OneFlow API Reference

Oneflow Contributors

Feb 28, 2021

CONTENTS

CHAPTER

ONE

TROUBLESHOOTING

- `CUDNN_STATUS_NOT_INITIALIZED`
  - You might see error message like these:

```
I0729 22:37:45.483937439   56788 ev_epoll_linux.c:82]        Use of signals␣
↪is disabled. Epoll enginll not be used
E0729 22:37:45.515343 56788 version.cpp:82] Failed to get cuda runtime␣
↪version: CUDA driver version nsufficient for CUDA runtime version
F0729 22:38:31.209002 56788 improver.cpp:535] Check failed: mem_size > 0 (-
↪524288000 vs. 0)
```

```
F0723 19:05:56.194067 40970 cuda_util.cpp:82] Check failed: error == CUDNN_
↪STATUS_SUCCESS (1 vs. 0) CUDNN_STATUS_NOT_INITIALIZED
```

  - Please upgrade to Nvidia Linux x86_64 driver. Version >= 440.33 is recommended.
  - For more information, please refer to CUDA compatibility documentation.
- Failed to compile `.cu` files
  - Please refer to CUDA System Requirements . Make sure your linux distribution and libraries shipped with it meet the requirements.
  - If you are using tools like conda, please make sure libraries you install doesn't shade the proper installation comes with linux distribution or package management like apt-get.
  - Please build OneFlow with a newer version of CMake. You could download version 3.14 from here: https://github.com/Kitware/CMake/releases/download/v3.14.0/cmake-3.14.0-Linux-x86_64.tar.gz
- How do I know what compilers and flags are used to compile OneFlow?
  - run `make clean && make VERBOSE=1` to get exact compile commands with compiler path and flags
- How to compile OneFlow with RDMA support?
  - add cmake flag `-DBUILD_RDMA` to compile OneFlow
- Which version of g++ CMake is using to build OneFlow?
  - You should find a line like this in CMake output:

```
-- CMAKE_CXX_COMPILER_VERSION: [YOUR G++ VERSION NUMBER]
```

- Failed to compile NCCL
  - Try use less threads when compiling OneFlow third party. For instance, use

```
cmake -DTHIRD_PARTY=ON .. && make
```

instead of

```
cmake -DTHIRD_PARTY=ON .. && make -j$(nproc) `
```

- "CUDA_VERSION" "VERSION_GREATER_EQUAL" "10.0"
  - Please use a newer version of CMake
  - Make sure cmake is correctly included in PATH
- CUBLAS not found
  - Usually it happens when using CUDA 10.1 or newer
  - You should see error massage by CMake like this:

```
cuda lib not found: /usr/local/miniconda3/envs/dl/lib/libcublas_static.a or
/usr/local/cuda/lib64/libcublas_static.a
```

  - Make sure libcublas_static.a is in one of the two directories.
- When running OneFlow in gdb, there is no debug information for code location.
  - add cmake flag -DCMAKE_BUILD_TYPE=RELWITHDEBINFO or -DCMAKE_BUILD_TYPE=DEBUG and recompile
- libof_ccobj.a: File truncated
  - You might see error message like this:

```
/usr/bin/ar: libof_ccobj.a: File truncated
make[2]: *** [libof_ccobj.a] Error 1
make[2]: *** Deleting file `libof_ccobj.a'
make[1]: *** [CMakeFiles/of_ccobj.dir/all] Error 2
make: *** [all] Error 2
```

  - You should upgrade your GNU Binutils. Version 2.33.1 is recommended. If you are using conda, you could install it by running conda install -c conda-forge binutils
- Failed to compile because C++ 17 is enabled
  - In some cases, environment variable CXXFLAGS is not empty and contains --std c++17.
  - Check if it is empty by running echo $CXXFLAGS and clear it with unset CXXFLAGS.
- cmake outputs error No CMAKE_ASM_NASM_COMPILER could be found.
  - Install nasm. For instance, run sudo yum install nasm if you are on centos.
- No module named 'google.protobuf'
  - You might see error message like this:

```
Scanning dependencies of target generate_api
...
    from google.protobuf import descriptor as _descriptor
ModuleNotFoundError: No module named 'google.protobuf'
CMakeFiles/generate_api.dir/build.make:57: recipe for target 'CMakeFiles/
↪generate_api' failed
make[2]: *** [CMakeFiles/generate_api] Error 1
```

  - Install development dependencies by running:

```
pip3 install -r dev-requirements.txt
```

- Get gdb warning `ptrace: Operation not permitted.` and gdb command `bt` prints no backtrace
  - You might get this warning when debugging OneFlow with gdb inside a docker container. Try add these flags when launching your container:

    ```
    docker run --cap-add=SYS_PTRACE --security-opt seccomp=unconfined
    ```

  - Please refer to https://stackoverflow.com/questions/19215177/how-to-solve-ptrace-operation-not-permitted-when-trying-to-attach-gdb-to-a-pro
- It takes too long to download python packages when running `make`
  - If you are in China, you could run this to have pip download packages from domestic mirror of pypi:

    ```
    python3 -m pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/
    ↪simple
    ```

  - For more information on this, please refer to pypi

CHAPTER

TWO

ONEFLOW

class oneflow.**ConfigProto**

DESCRIPTOR = <google.protobuf.pyext._message.MessageDescriptor object>

io_conf
Field oneflow.ConfigProto.io_conf

load_lib_path
Field oneflow.ConfigProto.load_lib_path

profiler_conf
Field oneflow.ConfigProto.profiler_conf

resource
Field oneflow.ConfigProto.resource

session_id
Field oneflow.ConfigProto.session_id

class oneflow.**DeprecatedFixedTensorDef**(**args*, ***kwargs*)

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class oneflow.**DeprecatedMirroredTensorDef**(**args*, ***kwargs*)

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

class oneflow.**DeprecatedTensorListDef**(**args*, ***kwargs*)

__init__(**args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

oneflow.**FixedTensorDef**
alias of oneflow.python.framework.input_blob_def.DeprecatedFixedTensorDef

class oneflow.**FunctionConfig**
OneFlow function's configurations.

__init__() → None
Initialize self. See help(type(self)) for accurate signature.

property all_reduce_fp16

property all_reduce_group_min_mbyte

property all_reduce_group_num

property all_reduce_group_size_warmup

property all_reduce_lazy_ratio

property allow_cpu_return_op

property concurrency_width

property cudnn_buf_limit_mbyte

property cudnn_conv_enable_pseudo_half

property cudnn_conv_enable_true_half

property cudnn_conv_force_bwd_data_algo

property cudnn_conv_force_bwd_filter_algo

property cudnn_conv_force_fwd_algo

property cudnn_conv_heuristic_search_algo

property cudnn_conv_use_deterministic_algo_only

property default_data_type

property default_distribute_strategy

property default_initializer_conf

property default_logical_view

property default_placement_scope

property disable_all_reduce_sequence

property do_parallel_cast_before_widening_type_cast

property enable_all_reduce_group

property enable_auto_mixed_precision

property enable_cudnn

property enable_cudnn_conv_pseudo_half

property enable_cudnn_fused_normalization_add_relu

property enable_float_compute_for_half_gemm

property enable_fuse_add_to_output

property enable_fuse_cast_scale

property enable_fuse_model_update_ops

```
    property enable_gradients_stats_aggregation
    property enable_inplace
    property enable_inplace_in_reduce_struct
    property enable_keep_header_only
    property enable_nccl
    property enable_non_distributed_optimizer
    property enable_qat
    property enable_quantization_aware_training
    property enable_reused_mem
    property enable_true_half_config_when_conv
    property exp_run_conf
    property indexed_slices_optimizer_conf
    property non_distributed_optimizer_group_size_mbyte
    property optimizer_placement_optimization_mode
    property optimizer_placement_optimization_threshold
    property prune_cast_to_static_shape_ops
    property prune_parallel_cast_ops
    property qat
    property static_mem_alloc_algo_white_list
    property static_mem_alloc_policy_white_list
    property tensorrt
    property train
    property use_boxing_v2
    property use_memory_allocation_algorithm_v2
    property use_nccl_inter_node_communication
    property use_tensorrt
    property use_xla_jit
class oneflow.JobConfigProto

    DESCRIPTOR = <google.protobuf.pyext._message.MessageDescriptor object>
    class FlagName2flagValueEntry

        DESCRIPTOR = <google.protobuf.pyext._message.MessageDescriptor object>
        key
            Field oneflow.JobConfigProto.FlagName2flagValueEntry.key
        value
            Field oneflow.JobConfigProto.FlagName2flagValueEntry.value
```

concurrency_width
Field oneflow.JobConfigProto.concurrency_width

cudnn_buf_limit_mbyte
Field oneflow.JobConfigProto.cudnn_buf_limit_mbyte

cudnn_conv_enable_pseudo_half
Field oneflow.JobConfigProto.cudnn_conv_enable_pseudo_half

cudnn_conv_force_bwd_data_algo
Field oneflow.JobConfigProto.cudnn_conv_force_bwd_data_algo

cudnn_conv_force_bwd_filter_algo
Field oneflow.JobConfigProto.cudnn_conv_force_bwd_filter_algo

cudnn_conv_force_fwd_algo
Field oneflow.JobConfigProto.cudnn_conv_force_fwd_algo

cudnn_conv_heuristic_search_algo
Field oneflow.JobConfigProto.cudnn_conv_heuristic_search_algo

cudnn_conv_use_deterministic_algo_only
Field oneflow.JobConfigProto.cudnn_conv_use_deterministic_algo_only

default_data_type
Field oneflow.JobConfigProto.default_data_type

default_initialize_with_snapshot_path
Field oneflow.JobConfigProto.default_initialize_with_snapshot_path

default_initializer_conf
Field oneflow.JobConfigProto.default_initializer_conf

do_parallel_cast_before_widening_type_cast
Field oneflow.JobConfigProto.do_parallel_cast_before_widening_type_cast

enable_auto_mixed_precision
Field oneflow.JobConfigProto.enable_auto_mixed_precision

enable_cudnn
Field oneflow.JobConfigProto.enable_cudnn

enable_cudnn_fused_normalization_add_relu
Field oneflow.JobConfigProto.enable_cudnn_fused_normalization_add_relu

enable_float_compute_for_half_gemm
Field oneflow.JobConfigProto.enable_float_compute_for_half_gemm

enable_fuse_add_to_output
Field oneflow.JobConfigProto.enable_fuse_add_to_output

enable_fuse_cast_scale
Field oneflow.JobConfigProto.enable_fuse_cast_scale

enable_fuse_model_update_ops
Field oneflow.JobConfigProto.enable_fuse_model_update_ops

enable_gradients_stats_aggregation
Field oneflow.JobConfigProto.enable_gradients_stats_aggregation

enable_inplace
Field oneflow.JobConfigProto.enable_inplace

enable_inplace_in_reduce_struct
Field oneflow.JobConfigProto.enable_inplace_in_reduce_struct

enable_keep_header_only
Field oneflow.JobConfigProto.enable_keep_header_only

enable_quantization_aware_training
Field oneflow.JobConfigProto.enable_quantization_aware_training

enable_reuse_mem
Field oneflow.JobConfigProto.enable_reuse_mem

exp_run_conf
Field oneflow.JobConfigProto.exp_run_conf

flag_name2flag_value
Field oneflow.JobConfigProto.flag_name2flag_value

indexed_slices_optimizer_conf
Field oneflow.JobConfigProto.indexed_slices_optimizer_conf

job_name
Field oneflow.JobConfigProto.job_name

logical_object_id
Field oneflow.JobConfigProto.logical_object_id

memory_allocation_algorithm_conf
Field oneflow.JobConfigProto.memory_allocation_algorithm_conf

optimizer_placement_optimization_mode
Field oneflow.JobConfigProto.optimizer_placement_optimization_mode

optimizer_placement_optimization_threshold
Field oneflow.JobConfigProto.optimizer_placement_optimization_threshold

predict_conf
Field oneflow.JobConfigProto.predict_conf

prune_cast_to_static_shape_ops
Field oneflow.JobConfigProto.prune_cast_to_static_shape_ops

prune_parallel_cast_ops
Field oneflow.JobConfigProto.prune_parallel_cast_ops

qat_config
Field oneflow.JobConfigProto.qat_config

total_batch_num
Field oneflow.JobConfigProto.total_batch_num

train_conf
Field oneflow.JobConfigProto.train_conf

use_memory_allocation_algorithm_v2
Field oneflow.JobConfigProto.use_memory_allocation_algorithm_v2

xrt_config
Field oneflow.JobConfigProto.xrt_config

oneflow.**MirroredTensorDef**
alias of `oneflow.python.framework.input_blob_def.DeprecatedMirroredTensorDef`

oneflow.**MirroredTensorListDef**
alias of oneflow.python.framework.input_blob_def.DeprecatedTensorListDef

oneflow.**acc**(*one: oneflow_api.BlobDesc*, *max_acc_num: int*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.**amp_white_identity**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.**argsort**(*input: oneflow_api.BlobDesc*, *axis: int = -1*, *direction: str = 'ASCENDING'*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator sorts the input Blob at specified axis and return the indices of the sorted Blob.

Parameters

- **input** (*oneflow_api.BlobDesc*) – A Blob
- **axis** (*int, optional*) – dimension to be sorted. Defaults to the last dim (-1)
- **direction** (*str, optional*) – The direction in which to sort the Blob values. If the direction is "ASCENDING", The order of input will be sorted as ascending, else, the order of input will be sorted as descending. Defaults to "ASCENDING".
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The indices of the sorted Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def argsort_Job(x: tp.Numpy.Placeholder((5, ))
) -> tp.Numpy:
    return flow.argsort(input=x)

x = np.array([10, 2, 9, 3, 7]).astype("float32")
out = argsort_Job(x)

# out [1 3 4 2 0]
```

oneflow.**argwhere**(*condition: oneflow_api.BlobDesc*, *dtype: Optional[oneflow.python.framework.dtype.dtype] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator finds the indices of input Blob *condition* elements that are non-zero. It returns a List. Each element in the output is a coordinate that points to a non-zero element in the condition.

Parameters

- **condition** (*oneflow_api.BlobDesc*) – The input Blob.
- **dtype** (*Optional[dtype_util.dtype], optional*) – The data type of output. Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob. Its type is *ListNumpy*.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def argwhere_Job(x: tp.Numpy.Placeholder(shape=(2, 3), dtype=flow.float32),
) -> tp.ListNumpy:
    return flow.argwhere(x)


x = np.array([[0, 1, 0],
              [2, 0, 2]]).astype(np.float32)
out = argwhere_Job(x)

# out [array([[0, 1],
#             [1, 0],
#             [1, 2]], dtype=int32)]
```

oneflow.**assign**(*ref*, *value*, *dtype=None*, *name=None*)

oneflow.**broadcast_like**(*x: oneflow_api.BlobDesc*, *like: oneflow_api.BlobDesc*, *broadcast_axes: Optional[Sequence[int]] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator broadcast the input Blob *x* on the specified axis with input Blob *like*.

Parameters

- **x** (*oneflow_api.BlobDesc*) – The input Blob.
- **like** (*oneflow_api.BlobDesc*) – A Blob.
- **broadcast_axes** (*Optional[Sequence[int]], optional*) – The broadcast axis. Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Raises **ValueError** – The length of broadcast_axes must be greater than 0 and less than or equal to number of axes of like shape.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

Example 1:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def broadcast_like_Job(x: tp.Numpy.Placeholder(shape=(3, 1), dtype=flow.float32)
) -> tp.Numpy:
    like_tensor = flow.constant(value=1.0,
                                dtype=flow.float32,
                                shape=(3, 3))
    return flow.broadcast_like(x=x,
```

(continues on next page)

(continued from previous page)

```
                              like=like_tensor,
                              broadcast_axes=(1, ))


x = np.array([[1], [1], [1]]).astype(np.float32)
out = broadcast_like_Job(x)

# out [[[1 1 1]
#       [1 1 1]
#       [1 1 1]]]

# out.shape (3, 3)
```

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def broadcast_like_Job(x: tp.Numpy.Placeholder(shape=(3, 1, 1), dtype=flow.
↪float32)
) -> tp.Numpy:
    like_tensor = flow.constant(value=1.0,
                                dtype=flow.float32,
                                shape=(3, 3, 3))
    return flow.broadcast_like(x=x,
                              like=like_tensor,
                              broadcast_axes=(1, 2))


x = np.random.randn(3, 1, 1).astype(np.float32)
out = broadcast_like_Job(x)

# out.shape (3, 3, 3)
```

oneflow.**broadcast_to_compatible_with**(*x: oneflow_api.BlobDesc, compatible: Sequence[oneflow_api.BlobDesc], name: Optional[str] = None*) → oneflow_api.BlobDesc

Returns a 'Blob' with the shape can be broadcasted by other shapes

Parameters

- **x** (*oneflow_api.BlobDesc*) – a 'Blob'
- **compatible** (*Sequence[oneflow_api.BlobDesc]*) – Sequence of different shape
- **name** (*Optional[str], optional*) – This operator's name. Defaults to None.

Returns A 'Blob' with the biggest shape

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import oneflow.typing as tp

@flow.global_function()
def broadcast_to_compatible_with_Job(x: tp.Numpy.Placeholder((4, 1, 1))
)->tp.Numpy:
    blob_a = flow.constant(value=1, dtype=flow.float32, shape=(1, 2, 1))
    blob_b = flow.constant(value=1, dtype=flow.float32, shape=(1, 1, 3))

    return flow.math.broadcast_to_compatible_with(x, [blob_a, blob_b])

x = np.ones(shape=(4, 1, 1), dtype=np.float32)

out = broadcast_to_compatible_with_Job(x)

# out.shape (4, 2, 3)
```

oneflow.**cast**(*x: oneflow_api.BlobDesc*, *dtype: oneflow.python.framework.dtype.dtype*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

The op takes input x and casts it to the output with *dtype*

Parameters

- **x** (*oneflow_api.BlobDesc*) – Input Blob
- **dtype** (*dtype_util.dtype*) – Data type of the output
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def cast_Job(x: tp.Numpy.Placeholder((3, ), dtype=flow.float32)
)->tp.Numpy:
    return flow.cast(x, dtype=flow.int32)

x = np.array([1, 2, 3]).astype(np.float32)
out = cast_Job(x)

# out.dtype = "int32"
```

oneflow.**cast_to_current_logical_view**(*x: oneflow_api.BlobDesc*) → oneflow_api.BlobDesc

oneflow.**cast_to_static_shape**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator returns a *Blob* that has identical content and data type to input *Blob*, and whose shape is converted from dynamic to static

Parameters

- **x** (*oneflow_api.BlobDesc*) – The input Blob which has dynamic shape.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob which is identical to input blob but has static shape.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def cast_to_static_shape_func(
    x: tp.ListNumpy.Placeholder(shape=(3, 3), dtype=flow.float32),
) -> tp.Numpy:
    return flow.cast_to_static_shape(x)

x = np.array([[1, 1, 1],
              [2, 2, 2],
              [3, 3, 3]]).astype(np.float32)

out = cast_to_static_shape_func(x)

# out [[1 1 1]
#      [2 2 2]
#      [3 3 3]]
```

oneflow.**categorical_ordinal_encode**(*table: oneflow_api.BlobDesc*, *size: oneflow_api.BlobDesc*, *input_tensor: oneflow_api.BlobDesc*, *hash_precomputed: bool = True*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator maintains a hash table to encode the categorical ordinal Blob. It converts a discrete input value into a continuous integer ID.

Parameters

- **table** (*oneflow_api.BlobDesc*) – The hash table, you can assign it as a variable.
- **size** (*oneflow_api.BlobDesc*) – The size of hash table.
- **input_tensor** (*oneflow_api.BlobDesc*) – The input Blob.
- **hash_precomputed** (*bool, optional*) – We currently only support the 'True' mode. The internal hash value will no longer be computed. Defaults to True.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def categorical_ordinal_encode_Job(x: tp.Numpy.Placeholder((3, 3), dtype=flow.
↪int32)
) -> tp.Numpy:
    dtype = x.dtype
    with flow.scope.namespace("categorical_ordinal_encode"):
        table = flow.get_variable(
```

(continues on next page)

(continued from previous page)

```
            name="Table",
            shape=(16,),
            dtype=dtype,
            initializer=flow.constant_initializer(0, dtype=dtype),
            trainable=False,
            reuse=False,
        )
        size = flow.get_variable(
            name="Size",
            shape=(1,),
            dtype=dtype,
            initializer=flow.constant_initializer(0, dtype=dtype),
            trainable=False,
            reuse=False,
        )
        return flow.categorical_ordinal_encode(
            table=table, size=size, input_tensor=x, name="Encode",
        )

x = np.array([[7, 0, 2],
              [1, 7, 2],
              [0, 1, 7]]).astype(np.int32)

out = categorical_ordinal_encode_Job(x)

# out [[1 0 2]
#      [3 1 2]
#      [0 3 1]]
```

class oneflow.**char**

oneflow_proto_dtype = 1

oneflow.**clamp**(*values: oneflow_api.BlobDesc*, *min_value: Union[int, float, None] = None*, *max_value: Union[int, float, None] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This op clips Blob values to a specified min value and max value.

The equation is:

$$out = MIN(MAX(x, min), max)$$

Parameters

- **values** (*oneflow_api.BlobDesc*) – Input Blob
- **min_value** (*Optional[Union[int, float]], optional*) – The minimum value to clip by. Defaults to None.
- **max_value** (*Optional[Union[int, float]], optional*) – The maximum value to clip by. Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Raises **ValueError** – min_value and max_value *cannot be None at the same time*

Returns A clipped Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def clip_by_value_Job(x: tp.Numpy.Placeholder((4, ))
)->tp.Numpy:
    return flow.math.clip_by_value(x, min_value=-1, max_value=5)

x = np.array([-2, 1, 4, 7], dtype=np.float32)

out = clip_by_value_Job(x)

# out [-1. 1. 4. 5.]
```

oneflow.**clear_default_session**() → None
Clear the default session. All compiled OneFlow functions will be deleted.

oneflow.**clip**(*values: oneflow_api.BlobDesc*, *min_value: Union[int, float, None] = None*, *max_value: Union[int, float, None] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc
This op clips Blob values to a specified min value and max value.

The equation is:

$$out = MIN(MAX(x, min), max)$$

Parameters

- **values** (*oneflow_api.BlobDesc*) – Input Blob
- **min_value** (*Optional[Union[int, float]], optional*) – The minimum value to clip by. Defaults to None.
- **max_value** (*Optional[Union[int, float]], optional*) – The maximum value to clip by. Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Raises **ValueError** – min_value and max_value *cannot be None at the same time*

Returns A clipped Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def clip_by_value_Job(x: tp.Numpy.Placeholder((4, ))
)->tp.Numpy:
    return flow.math.clip_by_value(x, min_value=-1, max_value=5)

x = np.array([-2, 1, 4, 7], dtype=np.float32)

out = clip_by_value_Job(x)

# out [-1. 1. 4. 5.]
```

oneflow.**clip_by_scalar**(*values: oneflow_api.BlobDesc*, *min_value: Union[int, float, None] = None*, *max_value: Union[int, float, None] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This op clips Blob values to a specified min value and max value.

The equation is:

$$out = MIN(MAX(x, min), max)$$

Parameters

- **values** (*oneflow_api.BlobDesc*) – Input Blob
- **min_value** (*Optional[Union[int, float]], optional*) – The minimum value to clip by. Defaults to None.
- **max_value** (*Optional[Union[int, float]], optional*) – The maximum value to clip by. Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Raises **ValueError** – min_value and max_value *cannot be None at the same time*

Returns A clipped Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def clip_by_value_Job(x: tp.Numpy.Placeholder((4, ))
)->tp.Numpy:
    return flow.math.clip_by_value(x, min_value=-1, max_value=5)

x = np.array([-2, 1, 4, 7], dtype=np.float32)

out = clip_by_value_Job(x)

# out [-1. 1. 4. 5.]
```

oneflow.**clip_by_value**(*values: oneflow_api.BlobDesc*, *min_value: Union[int, float, None] = None*, *max_value: Union[int, float, None] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This op clips Blob values to a specified min value and max value.

The equation is:

$$out = MIN(MAX(x, min), max)$$

Parameters

- **values** (*oneflow_api.BlobDesc*) – Input Blob
- **min_value** (*Optional[Union[int, float]], optional*) – The minimum value to clip by. Defaults to None.
- **max_value** (*Optional[Union[int, float]], optional*) – The maximum value to clip by. Defaults to None.

- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Raises **ValueError** – min_value and max_value *cannot be None at the same time*

Returns A clipped Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def clip_by_value_Job(x: tp.Numpy.Placeholder((4, ))
)->tp.Numpy:
    return flow.math.clip_by_value(x, min_value=-1, max_value=5)

x = np.array([-2, 1, 4, 7], dtype=np.float32)

out = clip_by_value_Job(x)

# out [-1. 1. 4. 5.]
```

oneflow.**combined_margin_loss**(*x: oneflow_api.BlobDesc*, *label: oneflow_api.BlobDesc*, *m1: float = 1*, *m2: float = 0*, *m3: float = 0*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.**concat**(*inputs: Optional[Sequence[oneflow_api.BlobDesc]] = None*, *axis: int = 0*, *max_dim_size: Optional[int] = None*, *name: Optional[str] = None*, *values: Optional[Sequence[oneflow_api.BlobDesc]] = None*) → oneflow_api.BlobDesc

Concatenate two or more *Blob* s at specified axis.

Analogous to numpy.concatenate

Parameters

- **inputs** – a *list* of *Blob*
- **axis** – a *int*. *0* by default
- **max_dim_size** – hint of max dimension size along the given axis
- **name** – name of this operator. *None* by default
- **values** – deprecated param, use inputs instead

Returns A *Blob*

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def concat_Job() -> tp.Numpy:
    constant_blob_1 = flow.constant(value=1.5,
                                    shape=(1, 3, 3, 4),
                                    dtype=flow.float,
                                    name="blob1")
```

(continues on next page)

(continued from previous page)

```
    constant_blob_2 = flow.constant(value=2.5,
                                    shape=(1, 3, 3, 4),
                                    dtype=flow.float,
                                    name="blob2")
    return flow.concat(inputs=[constant_blob_1, constant_blob_2],
                       axis=3)


out = concat_Job()

# out.shape (1, 3, 3, 8)
```

oneflow.**consistent_user_op_builder**(*op_name*)

oneflow.**consistent_user_op_module_builder**(*op_type_name*)

oneflow.**constant**(*value: Union[int, float], dtype: Optional[oneflow.python.framework.dtype.dtype] = None, shape: Optional[Sequence[int]] = None, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator creates a constant Blob.

Parameters

- **value** (*Union[int, float]*) – The constant value of Blob.
- **dtype** (*Optional[dtype_util.dtype], optional*) – The data type of Blob. Defaults to None.
- **shape** (*Optional[Sequence[int]], optional*) – The shape of Blob. Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Raises **NotImplementedError** – The data type of value should be int or float.

Returns The result blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def constant_Job() -> tp.Numpy:
    constant_blob = flow.constant(value=1.5,
                                  shape=(1, 3, 3),
                                  dtype=flow.float)
    return constant_blob


out = constant_Job()

# out [[[1.5 1.5 1.5]
#       [1.5 1.5 1.5]
#       [1.5 1.5 1.5]]]
```

oneflow.**constant_initializer**(*value: float = 0*, *dtype: oneflow.python.framework.dtype.dtype = <class 'oneflow.python.framework.dtype.float32'>*) → oneflow.core.job.initializer_conf_pb2.InitializerConf

Initializer that generates blob with constant values.

Parameters

- **value** (*float, optional*) – A Python scalar. All elements of the initialized variable . Defaults to 0.
- **dtype** (*dtype_util.dtype, optional*) – Default data type. Defaults to dtype_util.float.

Raises **NotImplementedError** – Do not support such data type.

Returns An InitializerConf object.

Return type initializer_conf_util.InitializerConf

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp


def watch_handler(y: tp.Numpy):
    print("out", y)


@flow.global_function()
def constant_Job() -> None:
    init = flow.constant_initializer(2.5)
    blob = flow.get_variable(
        "blob-weight",
        shape=(3, ),
        initializer=init,
        trainable=True
    )
    flow.watch(blob, watch_handler)


checkpoint = flow.train.CheckPoint()
checkpoint.init()
constant_Job()

# out [2.5 2.5 2.5]
```

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def conv2d_constant_Job(x: tp.Numpy.Placeholder((1, 256, 32, 32))
) -> tp.Numpy:
    initializer = flow.constant_initializer(0.01)
    conv2d = flow.layers.conv2d(
```

(continues on next page)

(continued from previous page)

```
        x,
        filters=128,
        kernel_size=3,
        strides=1,
        padding='SAME',
        kernel_initializer=initializer,
        name="Conv2d"
    )
    return conv2d


x = np.random.randn(1, 256, 32, 32).astype(np.float32)
out = conv2d_constant_Job(x)

# out.shape (1, 128, 32, 32)
```

oneflow.**constant_like**(*like: oneflow_api.BlobDesc, value: Union[int, float], dtype: Optional[oneflow.python.framework.dtype.dtype] = None, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator creates a constant Blob that has the same shape as *like*.

Parameters

- **like** (*oneflow_api.BlobDesc*) – A Blob.
- **value** (*Union[int, float]*) – The constant value of Blob.
- **dtype** (*Optional[dtype_util.dtype], optional*) – The data type of Blob. Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Raises **NotImplementedError** – The data type of value should be int or float.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def constant_like_Job() -> tp.Numpy:
    constant_blob = flow.constant(value=1.5,
                                  shape=(1, 3, 3),
                                  dtype=flow.float)
    constant_like_blob = flow.constant_like(like=constant_blob,
                                            value=5.5,
                                            dtype=flow.float)
    return constant_like_blob


out = constant_like_Job()

# out [[[5.5 5.5 5.5]
```

(continues on next page)

(continued from previous page)

```
#        [5.5 5.5 5.5]
#        [5.5 5.5 5.5]]]
```

oneflow.**constant_scalar**(*value: Union[int, float]*, *dtype: Optional[oneflow.python.framework.dtype.dtype] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator creates a constant scalar Blob.

Parameters

- **value** (*Union[int, float]*) – The constant value of Blob.
- **dtype** (*Optional[dtype_util.dtype], optional*) – The data type of Blob. Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def constant_scalar_Job() -> tp.Numpy:
    constant_scalar = flow.constant_scalar(value=2.5,
                                           dtype=flow.float)
    return constant_scalar


out = constant_scalar_Job()

# out [2.5]
```

oneflow.**convert_oneflow_dtype_to_numpy_dtype**(*oneflow_dtype: oneflow.python.framework.dtype.dtype*)

oneflow.**count_not_finite**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.**current_global_function_desc**() → oneflow.python.framework.function_desc.FunctionDesc

oneflow.**current_machine_id**()

Get machine id of current machine/node

Returns [description]

Return type [type]

oneflow.**current_resource**() → oneflow.core.job.resource_pb2.Resource

Get current resources, such as:machine nums, cpu/gpu device nums, epoch network threed num, rdma params...

Returns [description]

Return type resource_util.Resource

oneflow.**current_scope**()
Return current scope

oneflow.**dim_gather**(*input: oneflow_api.BlobDesc*, *dim: int*, *index: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc
This operator gathers elements from *input* according to *index* along with the axis *dim*.

Take a 3-D blob as example, the output is specified by:

```
output[i][j][k] = input[index[i][j][k]][j][k]  # if dim == 0
output[i][j][k] = input[i][index[i][j][k]][k]  # if dim == 1
output[i][j][k] = input[i][j][index[i][j][k]]  # if dim == 2
```

The shape of *input* and *index* should be the same except in the *dim* dimension.

That is, if *input* is a n-dimension blob with shape $(x_0, x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n)$, and *dim* = *i*, then *index* must be a n-dimension blob with shape $(x_0, x_1, \ldots, x_{i-1}, k, x_{i+1}, \ldots, x_n)$ where $k \geq 1$.

The return Blob *output* will have the same shape with *index*.

Parameters

- **input** (*oneflow_api.BlobDesc*) – The input blob
- **dim** (*int*) – The axis along which to index
- **index** (*oneflow_api.BlobDesc*) – The index blob of elements to gather
- **name** (*Optional[str], optional*) – The name of the operation. Defaults to None.

Returns The elements gathered from *input* will be returned as the output Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def dim_gather_Job(input: tp.Numpy.Placeholder((2, 2), dtype=flow.float64),
                   index:tp.Numpy.Placeholder((2, 2), dtype=flow.int32))->tp.Numpy:
    return flow.dim_gather(input, 1, index)

input = np.array([[1, 2], [3, 4]]).astype(np.float64)
index = np.array([[1, 0], [0, 1]]).astype(np.int32)

out = dim_gather_Job(input, index)
# output
# [[2. 1.]
#  [3. 4.]]
```

oneflow.**distributed_partial_fc_sample**(*weight: oneflow_api.BlobDesc*, *label: oneflow_api.BlobDesc*, *num_sample: int*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.**double**
alias of oneflow.python.framework.dtype.float64

oneflow.**dtypes**()

oneflow.**dynamic_reshape**(*x: oneflow_api.BlobDesc, shape: Sequence[int], name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator reshapes a dynamic blob.

Parameters

- **x** (*oneflow_api.BlobDesc*) – The input Blob.
- **shape** (*Sequence[int]*) – The output shape.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def dynamic_reshape_Job(x: tp.Numpy.Placeholder(shape=(1, 3, 64, 64), dtype=flow.
↪float32)
) -> tp.Numpy:
    reshape_out1 = flow.dynamic_reshape(x, (-1, 64))
    variable1 = flow.get_variable(
        "var1",
        shape=(64, 32),
        dtype=flow.float,
        initializer=flow.random_uniform_initializer(minval=-10, maxval=10),
        trainable=True,
    )
    matmul_tensor = flow.matmul(reshape_out1, variable1)
    reshape_out2 = flow.dynamic_reshape(matmul_tensor, (-1, 8, 4))
    return reshape_out2

x = np.random.rand(1, 3, 64, 64).astype(np.float32)
out = dynamic_reshape_Job(x)

# out.shape (192, 8, 4)
```

oneflow.**eager_execution_enabled**() → bool

Get current setting of the job, if enable eager execution mode ,then return True

Returns [description]

Return type bool

oneflow.**eager_nccl_all_reduce**(*x: oneflow_api.BlobDesc, parallel_conf: str, name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.**elem_cnt**(*inputs: oneflow_api.BlobDesc, dtype: Optional[oneflow.python.framework.dtype.dtype] = None, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator returns the amount of elements in input Blob.

Parameters

- **inputs** (*oneflow_api.BlobDesc*) – The input Blob.
- **dtype** (*Optional[dtype_util.dtype], optional*) – The data type. Defaults to None.

- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob. Its type is *ListNumpy*.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def elem_cnt_Job(x: tp.Numpy.Placeholder(shape=(5, ), dtype=flow.float32),
) -> tp.ListNumpy:
    return flow.elem_cnt(inputs=x, dtype=flow.int32)

x = np.array([10, 20, -30, 40, 50]).astype(np.float32)
out = elem_cnt_Job(x)

# [array([5], dtype=int32)]
```

oneflow.**enable_eager_execution**(*val: bool = True*) → None

If True, job will execute in eager mode, else use lazy mode(static graph).

Parameters **val** (*bool, optional*) – Whether eager execution or not. Defaults to True.

oneflow.**expand_dims**(*input: oneflow_api.BlobDesc*, *axis: int*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator inserts a dimention at the specified axis in the input Blob. The size of new dimension can only be 1, and the amount of element in return value is the same as Blob *input*.

Parameters

- **input** (*oneflow_api.BlobDesc*) – The input Blob.
- **axis** (*int*) – The specified dimension index.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def expand_dim_Job(x: tp.Numpy.Placeholder(shape=(1, 3, 3), dtype=flow.int32),
) -> tp.Numpy:
    return flow.expand_dims(input=x,
                            axis=2)


x = np.array([[[1, 1, 1],
               [1, 1, 1],
               [1, 1, 1]]]).astype(np.int32)
```

(continues on next page)

(continued from previous page)

```
out = expand_dim_Job(x)

# out.shape (1, 3, 1, 3)
```

oneflow.**find_or_create_module**(*module_name: str, create: Callable[[], None], reuse: bool = False*)

oneflow.**flatten**(*input: oneflow_api.BlobDesc, start_dim: int = 0, end_dim: int = -1, name: Optional[str] = None*) → oneflow_api.BlobDesc

Flattens a contiguous range of dims in a Blob.

Parameters

- **input** – A *Blob*.
- **start_dim** – The first dim to flatten.
- **end_dim** – The last dim to flatten.
- **name** – A name for the operation (optional).

Returns A *Blob*, has the same type as *input*.

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def flatten_Job(input: tp.Numpy.Placeholder(shape=(4, 4, 3, 2), dtype=flow.
↪float32)
) -> tp.Numpy:
    flatten_blob = flow.flatten(input, start_dim=1, end_dim=-1)
    return flatten_blob


input = np.zeros((4, 4, 3, 2)).astype(np.float32)
out = flatten_Job(input)

# out.shape (4, 24)
```

oneflow.**float**

alias of oneflow.python.framework.dtype.float32

class oneflow.**float16**

oneflow_proto_dtype = 9

class oneflow.**float32**

oneflow_proto_dtype = 2

class oneflow.**float64**

oneflow_proto_dtype = 3

oneflow.**function_config**

alias of oneflow.python.framework.function_util.FunctionConfig

oneflow.**gather**(*params: oneflow_api.BlobDesc*, *indices: oneflow_api.BlobDesc*, *validate_indices: Optional[oneflow_api.BlobDesc] = None*, *axis: Optional[int] = None*, *batch_dims: int = 0*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator gathers slices from params *axis* according to indices.

Parameters

- **params** – A *Blob*. The blob from which to gather values. Must be at least rank *axis + 1*.
- **indices** – A *Blob*. Index blob. Must be in range [0, params.shape[axis]).
- **axis** – A *int*. The axis in params to gather indices from. Defaults to the first dimension. Supports negative indexes.
- **batch_dims** – An optional *int*. Defaults to 0.
- **name** – A name for the operation (optional).

Returns A blob. Has the same type as params.

For example:

Example 1:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def gather_Job(x: tp.Numpy.Placeholder(shape=(3, 3), dtype=flow.float32),
               indice: tp.Numpy.Placeholder(shape=(2, ), dtype=flow.int32)
) -> tp.Numpy:
    gather_blob = flow.gather(params=x,
                              indices=indice,
                              axis=1)
    return gather_blob


x = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]]).astype(np.float32)
indice = np.array([0, 2]).astype(np.int32)
out = gather_Job(x, indice)

# out [[1. 3.]
#      [4. 6.]
#      [7. 9.]]
```

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def gather_Job(x: tp.Numpy.Placeholder(shape=(3, 3), dtype=flow.float32),
               indice: tp.Numpy.Placeholder(shape=(2, ), dtype=flow.int32)
) -> tp.Numpy:
    gather_blob = flow.gather(params=x,
```

(continues on next page)

(continued from previous page)

```
                              indices=indice,
                              axis=0)
    return gather_blob


x = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]]).astype(np.float32)
indice = np.array([0, 2]).astype(np.int32)
out = gather_Job(x, indice)

# out [[1. 2. 3.]
#      [7. 8. 9.]]
```

oneflow.**gather_nd**(*params: oneflow_api.BlobDesc*, *indices: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator is a high-dimensional extension of *gather*, *indices* is a K-dimensional tensor, which is regarded as a index of input Blob *params*.

Each element defines a slice of *params*:

$$output[(i_0, i_1, ..., i_{K-2})] = param[indices(i_0, i_1, ..., i_{K-2})]$$

Parameters

- **params** (*oneflow_api.BlobDesc*) – The input Blob.
- **indices** (*oneflow_api.BlobDesc*) – The slice indices.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

Example 1:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def gather_nd_Job(x: tp.Numpy.Placeholder(shape=(3, 3), dtype=flow.float32),
                  indice: tp.Numpy.Placeholder(shape=(2, 1), dtype=flow.int32)
) -> tp.Numpy:
    gather_nd_blob = flow.gather_nd(params=x,
                                    indices=indice)
    return gather_nd_blob


x = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]]).astype(np.float32)
indice = np.array([[0], [2]]).astype(np.int32)
out = gather_nd_Job(x, indice)
```

(continues on next page)

(continued from previous page)

```
# out [[1. 2. 3.]
#      [7. 8. 9.]]
```

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def gather_nd_Job(x: tp.Numpy.Placeholder(shape=(3, 3), dtype=flow.float32),
                  indice: tp.Numpy.Placeholder(shape=(2, 2), dtype=flow.int32)
) -> tp.Numpy:
    gather_nd_blob = flow.gather_nd(params=x,
                                    indices=indice)
    return gather_nd_blob


x = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]]).astype(np.float32)
indice = np.array([[0, 2], [2, 1]]).astype(np.int32)
out = gather_nd_Job(x, indice)

# out [3. 8.]
```

Example3:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def gather_nd_Job(x: tp.Numpy.Placeholder(shape=(3, 3), dtype=flow.float32),
                  indice: tp.Numpy.Placeholder(shape=(3, 2), dtype=flow.int32)
) -> tp.Numpy:
    gather_nd_blob = flow.gather_nd(params=x,
                                    indices=indice)
    return gather_nd_blob


x = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]]).astype(np.float32)
indice = np.array([[0, 1], [1, 0], [2, 2]]).astype(np.int32)
out = gather_nd_Job(x, indice)

# out [2. 4. 9.]
```

oneflow.**get_all_variables**() → Dict[str, oneflow.python.framework.remote_blob.EagerConsistentBlob]

Get all variables of all jobs as a dict.

oneflow.**get_variable**(*name: str*, *shape: Optional[Sequence[int]] = None*, *dtype: Optional[oneflow.python.framework.dtype.dtype] = <class 'oneflow.python.framework.dtype.float32'>*, *initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None*, *regularizer: Optional[oneflow.core.job.regularizer_conf_pb2.RegularizerConf] = None*, *trainable: Optional[bool] = None*, *model_name: Optional[str] = None*, *random_seed: Optional[int] = None*, *distribute: oneflow_api.distribute.Distribute = <oneflow_api.distribute.BroadcastDistribute object>*, *reuse: bool = True*) → oneflow_api.BlobDesc

Create a variable or retrieve an existing one.

Parameters

- **name** – Name of this variable. One variable could be shared by multiple OneFlow functions. *None* by defaulit
- **shape** – Shape of the variable. *None* by defaulit
- **dtype** – Data type of the variable. *None* by defaulit
- **initializer** – A initializer object. For instance, a *ones_initializer()*. *None* by defaulit
- **trainable** – A *bool* to indicate if this variable is trainable. *True* by defaulit
- **model_name** – A *string*. *'weight'* or *'bias'*. *None* by defaulit
- **random_seed** – Random seed for random initializers. *None* by defaulit

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp


def watch_handler(y: tp.Numpy):
    print("out", y)


@flow.global_function()
def variable_Job() -> None:
    init = flow.constant_initializer(1.25)
    variable = flow.get_variable(
        "variable-weight",
        shape=(1, 3, 2, 2),
        initializer=init,
        trainable=True
    )
    flow.watch(variable, watch_handler)


checkpoint = flow.train.CheckPoint()
checkpoint.init()
variable_Job()

# out [[[[1.25 1.25]
#         [1.25 1.25]]

#       [[1.25 1.25]
```

(continues on next page)

(continued from previous page)

```
#         [1.25 1.25]]

#        [[1.25 1.25]
#         [1.25 1.25]]]]
```

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


def conv2d(input, filters, kernel_size, strides, padding, name):
    input_shape = input.shape
    weight_initializer = flow.truncated_normal(0.1)
    weight_regularizer = flow.regularizers.l2(0.0005)
    weight_shape = (filters,
                    input_shape[1],
                    kernel_size[0],
                    kernel_size[1])

    weight = flow.get_variable(
        name + "-weight",
        shape=weight_shape,
        initializer=weight_initializer,
        regularizer=weight_regularizer,
    )
    return flow.nn.conv2d(input, weight, strides, padding, name=name)


@flow.global_function()
def conv2d_Job(x: tp.Numpy.Placeholder((1, 64, 32, 32))
) -> tp.Numpy:
    conv = conv2d(x,
                  filters=128,
                  kernel_size=[3, 3],
                  strides=2,
                  padding='SAME',
                  name="Convlayer")
    return conv


x = np.random.randn(1, 64, 32, 32).astype(np.float32)
out = conv2d_Job(x)

# out.shape (1, 128, 16, 16)
```

oneflow.**global_function**(*type: str = 'predict'*, *function_config: oneflow.python.framework.function_util.FunctionConfig = None*) → Callable[[Callable], Callable]

Creates a callable OneFlow global function from a Python function.

For instance:

```
@oneflow.global_function(flow.FunctionConfig())
def train():
    # your model
```

Parameters **function_config** (FunctionConfig, *optional*) – a *FunctionConfig* object. Defaults to FunctionConfig().

Returns a callable which is called to execute the compiled function

Return type Callable[[Callable], Callable]

oneflow.**glorot_normal_initializer**(*data_format: str = ''*) → oneflow.core.job.initializer_conf_pb2.InitializerConf

Initializer that generates a Xavier normal distribution.

It also can be called as *oneflow.glorot_normal_initializer*.

The equation is:

$$W \sim N(0, \sqrt{\frac{2}{n_j + n_{j+1}}})$$

N means normal distribution

n_j means the amount of Nth layer parameters

Parameters **data_format** (*str, optional*) – The data format. Defaults to "".

Returns Initial configuration

Return type initializer_conf_util.InitializerConf

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp


def watch_handler(y: tp.Numpy):
    print("out", y)


@flow.global_function()
def xavier_normal_Job() -> None:
    init = flow.xavier_normal_initializer()
    blob = flow.get_variable(
        "blob-weight",
        shape=(3, 3),
        initializer=init,
        trainable=True
    )
    flow.watch(blob, watch_handler)


checkpoint = flow.train.CheckPoint()
checkpoint.init()
xavier_normal_Job()

# out [[ 0.5908121  -0.10804518 -0.6148571 ]
#      [ 1.4007381  -0.08172473  0.36579943]
#      [-0.6461796  -0.15923311  0.33653972]]
```

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def conv2d_xavier_normal_Job(x: tp.Numpy.Placeholder((1, 256, 32, 32))
) -> tp.Numpy:
    initializer = flow.xavier_normal_initializer()
    conv2d = flow.layers.conv2d(
        x,
        filters=128,
        kernel_size=3,
        strides=1,
        padding='SAME',
        kernel_initializer=initializer,
        name="Conv2d"
    )
    return conv2d


x = np.random.randn(1, 256, 32, 32).astype(np.float32)
out = conv2d_xavier_normal_Job(x)

# out.shape (1, 128, 32, 32)
```

oneflow.**glorot_uniform_initializer**(*data_format: str = ''*) → oneflow.core.job.initializer_conf_pb2.InitializerConf

Initializer that generates a Xavier uniform distribution.

It also can be called as *oneflow.glorot_uniform_initializer*.

The equation is:

$$W \sim U(-\sqrt{\frac{6}{n_j + n_{j+1}}}, \sqrt{\frac{6}{n_j + n_{j+1}}})$$

U means uniform distribution

n_j means the amount of Nth layer parameters

Parameters **data_format** (*str, optional*) – The data format. Defaults to "".

Returns Initial configuration

Return type initializer_conf_util.InitializerConf

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp


def watch_handler(y: tp.Numpy):
    print("out", y)


@flow.global_function()
```

(continues on next page)

(continued from previous page)

```
def xavier_uniform_Job() -> None:
    init = flow.xavier_uniform_initializer()
    blob = flow.get_variable(
        "blob-weight",
        shape=(3, 3),
        initializer=init,
        trainable=True
    )
    flow.watch(blob, watch_handler)


checkpoint = flow.train.CheckPoint()
checkpoint.init()
xavier_uniform_Job()

# out [[-0.14424723 -0.9532095  -0.08723891]
#      [-0.8011227  -0.29729813 -0.26769108]
#      [ 0.9208976  -0.5971756  -0.15077025]]
```

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def conv2d_xavier_uniform_Job(x: tp.Numpy.Placeholder((1, 256, 32, 32))
) -> tp.Numpy:
    initializer = flow.xavier_uniform_initializer()
    conv2d = flow.layers.conv2d(
        x,
        filters=128,
        kernel_size=3,
        strides=1,
        padding='SAME',
        kernel_initializer=initializer,
        name="Conv2d"
    )
    return conv2d


x = np.random.randn(1, 256, 32, 32).astype(np.float32)
out = conv2d_xavier_uniform_Job(x)

# out.shape (1, 128, 32, 32)
```

oneflow.**identity**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator returns a *Blob* that has identical content and data type to input *Blob*.

Analogous to tf.identity

Parameters

- **x** (*oneflow_api.BlobDesc*) – The input Blob.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def identity_Job(x: tp.Numpy.Placeholder(shape=(3, 3), dtype=flow.int32),
) -> tp.Numpy:
    return flow.identity(x)

x = np.array([[1, 1, 1],
              [2, 2, 2],
              [3, 3, 3]]).astype(np.int32)
out = identity_Job(x)

# out [[1 1 1]
#      [2 2 2]
#      [3 3 3]]
```

oneflow.**identity_n**(*inputs: Sequence[oneflow_api.BlobDesc], name: Optional[str] = None*) → List[oneflow_api.BlobDesc]

This operator is similar to *oneflow.identity*. The difference is that the input and output of *identity_n* is *List*.

Parameters

- **inputs** (*Iterable[oneflow_api.BlobDesc]*) – A List of input Blob.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A list of result Blob.

Return type List[oneflow_api.BlobDesc]

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp
from typing import List


@flow.global_function()
def identity_Job(x: tp.Numpy.Placeholder(shape=(1, 3), dtype=flow.int32),
                 y: tp.Numpy.Placeholder(shape=(1, 3), dtype=flow.int32),
                 z: tp.Numpy.Placeholder(shape=(1, 3), dtype=flow.int32)
) -> List[tp.Numpy]:
    return flow.identity_n([x, y, z])


x = np.array([[1, 1, 1]]).astype(np.int32)
y = np.array([[2, 2, 2]]).astype(np.int32)
z = np.array([[3, 3, 3]]).astype(np.int32)
out = identity_Job(x, y, z)

# out[0] [[1, 1, 1]]
# out[1] [[2, 2, 2]]
# out[2] [[3, 3, 3]]
```

oneflow.**image_batch_align**(*images: oneflow_api.BlobDesc, shape: Sequence[int], dtype: oneflow.python.framework.dtype.dtype, alignment: int, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator aligns the shape for a batch of images.

The aligned shape is computed as:

$$shape_{width} = int(\frac{(shape_{width} + alignment - 1)}{alignment}) * alignment$$
$$shape_{height} = int(\frac{(shape_{height} + alignment - 1)}{alignment}) * alignment$$

Parameters

- **images** (*oneflow_api.BlobDesc*) – The images.
- **shape** (*Sequence[int]*) – The maximum static shape of input images.
- **dtype** (*dtype_util.dtype*) – The data type.
- **alignment** (*int*) – The align factor.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import cv2
import numpy as np
import oneflow as flow
import oneflow.typing as tp


def _of_image_batch_align(images, input_shape, output_shape, alignment):
    func_config = flow.FunctionConfig()
    func_config.default_data_type(flow.float)
    func_config.default_logical_view(flow.scope.mirrored_view())

    @flow.global_function(function_config=func_config)
    def image_batch_align_job(
        images_def: tp.ListListNumpy.Placeholder(shape=input_shape, dtype=flow.
↪float)
    ) -> tp.ListNumpy:
        # Convert to tensor buffer
        images_buffer = flow.tensor_list_to_tensor_buffer(images_def)
        image = flow.image_batch_align(
            images_buffer, shape=output_shape[1:], dtype=flow.float,␣
↪alignment=alignment
        )
        return image

    image = image_batch_align_job([images])
    return image[0]


def _read_images_by_cv(image_files):
    images = [cv2.imread(image_file).astype(np.single) for image_file in image_
↪files]
```

(continues on next page)

(continued from previous page)

```
    return [np.expand_dims(image, axis=0) for image in images]


def _get_images_static_shape(images):
    image_shapes = [image.shape for image in images]
    image_static_shape = np.amax(image_shapes, axis=0)
    assert isinstance(
        image_static_shape, np.ndarray
    ), "image_shapes: {}, image_static_shape: {}".format(
        str(image_shapes), str(image_static_shape)
    )
    image_static_shape = image_static_shape.tolist()
    assert image_static_shape[0] == 1, str(image_static_shape)
    image_static_shape[0] = len(image_shapes)
    return image_static_shape

def _roundup(x, n):
    # compute the aligned shape
    return int((x + n - 1) / n) * n

if __name__ == "__main__":
    img = _read_images_by_cv(['./img/1.jpg', './img/2.jpg', './img/3.jpg'])
    img_shape = _get_images_static_shape(img) # In example is [3, 349, 367, 3]
    alignment = 16 # alignment factor
    aligned_image_shape = [
        img_shape[0],
        _roundup(img_shape[1], alignment),
        _roundup(img_shape[2], alignment),
        img_shape[3],
    ]
    image = _of_image_batch_align(img, tuple(img_shape), aligned_image_shape,␣
↪alignment)
```

oneflow.**image_decode**(*images_bytes_buffer: oneflow_api.BlobDesc*, *dtype: oneflow.python.framework.dtype.dtype = <class 'oneflow.python.framework.dtype.uint8'>*, *color_space: str = 'BGR'*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator decode the image.

Parameters

- **images_bytes_buffer** (*oneflow_api.BlobDesc*) – The input Blob. Its type should be *kTensorBuffer*. More details please refer to the code example.
- **dtype** (*dtype_util.dtype, optional*) – The data type. Defaults to dtype_util.uint8.
- **color_space** (*str, optional*) – The color space. Defaults to "BGR".
- **name** (*Optional[str], optional*) – The name for the opreation. Defaults to None.

Returns The decoded image list.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp
```

(continues on next page)

(continued from previous page)

```
import numpy as np
from PIL import Image


def _of_image_decode(images):
    image_files = [open(im, "rb") for im in images]
    images_bytes = [imf.read() for imf in image_files]
    static_shape = (len(images_bytes), max([len(bys) for bys in images_bytes]))
    for imf in image_files:
        imf.close()

    func_config = flow.FunctionConfig()
    func_config.default_data_type(flow.float)
    func_config.default_logical_view(flow.scope.mirrored_view())

    @flow.global_function(function_config=func_config)
    def image_decode_job(
        images_def: tp.ListListNumpy.Placeholder(shape=static_shape, dtype=flow.
↪int8)
    )->tp.ListListNumpy:
        # convert to tensor buffer
        images_buffer = flow.tensor_list_to_tensor_buffer(images_def)
        decoded_images_buffer = flow.image_decode(images_buffer)
        # Remember to set a shape
        # convert back to tensor list
        return flow.tensor_buffer_to_tensor_list(
            decoded_images_buffer, shape=(640, 640, 3), dtype=flow.uint8
        )

    images_np_arr = [
        np.frombuffer(bys, dtype=np.byte).reshape(1, -1) for bys in images_bytes
    ]
    decoded_images = image_decode_job([images_np_arr])
    return decoded_images[0]


if __name__ == "__main__":
    img = _of_image_decode(['./img/1.jpg'])
    print(img[0].shape) # Our image shape is (1, 349, 367, 3)
```

oneflow.**image_flip**(*image: oneflow_api.BlobDesc, flip_code: Union[int, oneflow_api.BlobDesc], name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator flips the images.

The flip code corresponds to the different flip mode:

0 (0x00): Non Flip

1 (0x01): Horizontal Flip

16 (0x10): Vertical Flip

17 (0x11): Both Horizontal and Vertical Flip

Parameters

- **image** (*oneflow_api.BlobDesc*) – The input images.
- **flip_code** (*Union[int, oneflow_api.BlobDesc]*) – The flip code.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import cv2
import numpy as np
import oneflow as flow
import oneflow.typing as tp


def _of_image_flip(images, image_shape, flip_code):
    func_config = flow.FunctionConfig()
    func_config.default_data_type(flow.float)
    func_config.default_logical_view(flow.scope.mirrored_view())

    @flow.global_function(function_config=func_config)
    def image_flip_job(
        images_def: tp.ListListNumpy.Placeholder(shape=image_shape, dtype=flow.
↪float)
    ) -> tp.ListListNumpy:
        images_buffer = flow.tensor_list_to_tensor_buffer(images_def)
        flip_images = flow.image_flip(images_buffer, flip_code)
        return flow.tensor_buffer_to_tensor_list(
            flip_images, shape=image_shape[1:], dtype=flow.float
        )

    image_tensor = image_flip_job([images])
    return image_tensor[0]


def _read_images_by_cv(image_files):
    images = [cv2.imread(image_file).astype(np.single) for image_file in image_
↪files]
    return [np.expand_dims(image, axis=0) for image in images]


def _get_images_static_shape(images):
    image_shapes = [image.shape for image in images]
    image_static_shape = np.amax(image_shapes, axis=0)
    assert isinstance(
        image_static_shape, np.ndarray
    ), "image_shapes: {}, image_static_shape: {}".format(
        str(image_shapes), str(image_static_shape)
    )
    image_static_shape = image_static_shape.tolist()
    assert image_static_shape[0] == 1, str(image_static_shape)
    image_static_shape[0] = len(image_shapes)
    return image_static_shape

if __name__ == "__main__":
    img = _read_images_by_cv(['./img/1.jpg', './img/2.jpg', './img/3.jpg'])
    img_shape = _get_images_static_shape(img) # In example is [3, 349, 367, 3]
    image = _of_image_flip(img,
                   tuple(img_shape),
                   flip_code=1)
```

oneflow.**image_normalize**(*image: oneflow_api.BlobDesc, std: Sequence[float], mean: Sequence[float], name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator normalizes the image.

Parameters

- **image** (*oneflow_api.BlobDesc*) – The input image.
- **std** (*Sequence[float]*) – The standard deviation of the images.
- **mean** (*Sequence[float]*) – The mean value of the images.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import cv2
import numpy as np
import oneflow as flow
import oneflow.typing as tp


def _of_image_normalize(images, image_shape, std, mean):
    func_config = flow.FunctionConfig()
    func_config.default_data_type(flow.float)
    func_config.default_logical_view(flow.scope.mirrored_view())

    @flow.global_function(function_config=func_config)
    def image_normalize_job(
        images_def: tp.ListListNumpy.Placeholder(shape=image_shape, dtype=flow.
↪float)
    ) -> tp.ListListNumpy:
        # Convert to tensor buffer
        images_buffer = flow.tensor_list_to_tensor_buffer(images_def)
        # Normalize the imagess
        norm_images = flow.image_normalize(images_buffer, std, mean)
        # Convert back to tensor list
        return flow.tensor_buffer_to_tensor_list(
            norm_images, shape=image_shape[1:], dtype=flow.float
        )

    image_tensor = image_normalize_job([images])
    return image_tensor[0]


def _read_images_by_cv(image_files):
    images = [cv2.imread(image_file).astype(np.single) for image_file in image_
↪files]
    return [np.expand_dims(image, axis=0) for image in images]


def _get_images_static_shape(images):
    image_shapes = [image.shape for image in images]
    image_static_shape = np.amax(image_shapes, axis=0)
    assert isinstance(
        image_static_shape, np.ndarray
    ), "image_shapes: {}, image_static_shape: {}".format(
        str(image_shapes), str(image_static_shape)
    )
```

(continues on next page)

(continued from previous page)

```
    image_static_shape = image_static_shape.tolist()
    assert image_static_shape[0] == 1, str(image_static_shape)
    image_static_shape[0] = len(image_shapes)
    return image_static_shape

if __name__ == "__main__":
    img = _read_images_by_cv(['./img/1.jpg', './img/2.jpg', './img/3.jpg'])
    img_shape = _get_images_static_shape(img) # In example is [3, 349, 367, 3]
    image = _of_image_normalize(img,
                                tuple(img_shape),
                                std=(102.9801, 115.9465, 122.7717),
                                mean=(1.0, 1.0, 1.0))
```

oneflow.**image_random_crop**(*input_blob: oneflow_api.BlobDesc*, *num_attempts: int = 10*, *seed: Optional[int] = None*, *random_area: Sequence[float] = None*, *random_aspect_ratio: Sequence[float] = None*, *name: str = 'ImageRandomCrop'*) → oneflow_api.BlobDesc

This operator crops the input image randomly.

Parameters

- **input_blob** (*oneflow_api.BlobDesc*) – The input Blob.
- **num_attempts** (*int, optional*) – The maximum number of random cropping attempts. Defaults to 10.
- **seed** (*Optional[int], optional*) – The random seed. Defaults to None.
- **random_area** (*Sequence[float], optional*) – The random cropping area. Defaults to None.
- **random_aspect_ratio** (*Sequence[float], optional*) – The random scaled ratio. Defaults to None.
- **name** (*str, optional*) – The name for the operation. Defaults to "ImageRandomCrop".

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np
import cv2


def _read_images_by_cv(image_files):
    images = [cv2.imread(image_file).astype(np.single) for image_file in image_
↪files]
    return [np.expand_dims(image, axis=0) for image in images]


def _get_images_static_shape(images):
    image_shapes = [image.shape for image in images]
    image_static_shape = np.amax(image_shapes, axis=0)
    assert isinstance(
        image_static_shape, np.ndarray
```

(continues on next page)

(continued from previous page)

```
    ), "image_shapes: {}, image_static_shape: {}".format(
        str(image_shapes), str(image_static_shape)
    )
    image_static_shape = image_static_shape.tolist()
    assert image_static_shape[0] == 1, str(image_static_shape)
    image_static_shape[0] = len(image_shapes)
    return image_static_shape

def _of_image_random_crop(images, image_static_shape):
    func_config = flow.FunctionConfig()
    func_config.default_data_type(flow.float)
    func_config.default_logical_view(flow.scope.mirrored_view())

    @flow.global_function(function_config=func_config)
    def image_random_crop_job(images_def: tp.ListListNumpy.
↪Placeholder(shape=image_static_shape, dtype=flow.float)
    ) -> tp.ListListNumpy:
        # The input Blob type should be "kTensorBuffer"
        # So we use oneflow.tensor_list_to_tensor_buffer to convert
        images_buffer = flow.tensor_list_to_tensor_buffer(images_def)
        # Do the random crop
        random_crop_buffer = flow.image.random_crop(
            images_buffer,
            random_area=[0.15, 0.80],
            random_aspect_ratio=[0.75, 1.55],
        )
        # We convert back to "tensorlist" type
        random_crop_images = flow.tensor_buffer_to_tensor_list(
            random_crop_buffer,
            shape=(image_static_shape[1], image_static_shape[2], image_static_
↪shape[-1]),
            dtype=flow.float,
        )
        return random_crop_images

    random_crop_images = image_random_crop_job([images])

    return random_crop_images

if __name__ == "__main__":
    img = _read_images_by_cv(['./img/1.jpg'])
    img_shape = _get_images_static_shape(img) # In example is (1, 234, 346, 3)
    random_crop_images = _of_image_random_crop(img, tuple(img_shape))
    # random_crop_images.shape is (234, 346, 3)
```

oneflow.**image_resize**(*image: oneflow_api.BlobDesc*, *target_size: Union[int, Sequence[int]] = None*, *min_size: Optional[int] = None*, *max_size: Optional[int] = None*, *keep_aspect_ratio: bool = False*, *resize_side: str = 'shorter'*, *channels: int = 3*, *dtype: Optional[oneflow.python.framework.dtype.dtype] = None*, *interpolation_type: str = 'auto'*, *name: Optional[str] = None*, *color_space: Optional[str] = None*, *interp_type: Optional[str] = None*, *resize_shorter: int = 0*, *resize_x: int = 0*, *resize_y: int = 0*) → Union[oneflow_api.BlobDesc, Sequence[oneflow_api.BlobDesc]]

Resize images to target size.

Parameters

- **image** – A *Tensor* consists of images to be resized.
- **target_size** – A list or tuple when *keep_aspect_ratio* is false or an int when *keep_aspect_ratio* is true. When *keep_aspect_ratio* is false, *target_size* has a form of *(target_width, target_height)* that image will resize to. When *keep_aspect_ratio* is true, the longer side or shorter side of the image will be resized to target size.
- **min_size** – An int, optional. Only works when *keep_aspect_ratio* is true and *resize_side* is "longer". If *min_size* is not None, the shorter side must be greater than or equal to *min_size*. Default is None.
- **max_size** – An int, optional. Only works when *keep_aspect_ratio* is true and *resize_side* is "shorter". If *max_size* is not None, the longer side must be less than or equal to *max_size*. Default is None.
- **keep_aspect_ratio** – A bool. If is false, indicate that image will be resized to fixed width and height, otherwise image will be resized keeping aspect ratio.
- **resize_side** – A str of "longer" or "shorter". Only works when *keep_aspect_ratio* is True. If *resize_side* is "longer", the longer side of image will be resized to *target_size*. If *resize_side* is "shorter", the shorter side of image will be resized to *target_size*.
- **channels** – An int. how many channels an image has
- **dtype** – *oneflow.dtype*. Indicate output resized image data type.
- **interpolation_type** – A str of "auto", "bilinear", "nearest_neighbor", "bicubic" or "area". Indicate interpolation method used to resize image.
- **name** – A str, optional. Name for the operation.
- **color_space** – Deprecated, a str of "RGB", "BGR" or "GRAY". Please use *channels* instead.
- **interp_type** – Deprecated, s str of "Linear", "Cubic" or "NN". Please use *interpolation_type* instead.
- **resize_shorter** – Deprecated, a int. Indicate target size that the shorter side of image will resize to. Please use *target_size* and *resize_side* instead.
- **resize_x** – Deprecated, a int. Indicate the target size that the width of image will resize to. Please use *target_size* instead.
- **resize_y** – Deprecated, a int. Indicate the target size that the height of image will resize to. Please use *target_size* instead.

Returns Tuple of resized images *Blob*, width and height scales *Blob* and new width and height *Blob* (new width and height *Blob* will be None when keep_aspect_ratio is false). If deprecated params are used, a single resized images *Blob* will be returned.

For example:

```
import oneflow as flow
import oneflow.typing as tp
from typing import Tuple


@flow.global_function(type="predict")
def ofrecord_reader_job() -> Tuple[tp.Numpy, tp.Numpy]:
    batch_size = 16
    color_space = "RGB"
    # our ofrecord file path is "./dataset/part-0"
```

(continues on next page)

(continued from previous page)

```
    ofrecord = flow.data.ofrecord_reader(
        "./imgdataset",
        batch_size=batch_size,
        data_part_num=1,
        part_name_suffix_length=-1,
        part_name_prefix='part-',
        random_shuffle=True,
        shuffle_after_epoch=True,
    )
    image = flow.data.OFRecordImageDecoderRandomCrop(
            ofrecord, "encoded", color_space=color_space
        )
    res_image, scale, new_size = flow.image.Resize(
            image, target_size=(224, 224)
        )
    label = flow.data.OFRecordRawDecoder(
        ofrecord, "class/label", shape=(1, ), dtype=flow.int32
    )

    return res_image, label

if __name__ == "__main__":
    images, labels = ofrecord_reader_job()
    # image.shape (16, 224, 224, 3)
```

oneflow.**image_target_resize**(*images: oneflow_api.BlobDesc*, *target_size: int*, *min_size: Optional[int] = None*, *max_size: Optional[int] = None*, *resize_side: str = 'shorter'*, *interpolation_type: str = 'auto'*, *name: Optional[str] = None*) → Sequence[oneflow_api.BlobDesc]

This operator resizes image to target size.

Parameters

- **images** (*oneflow_api.BlobDesc*) – The input Blob. Its type should be *kTensorBuffer*. More details please refer to the code example.
- **target_size** (*int*) – An int, the target size.
- **min_size** (*Optional[int], optional*) – If *min_size* is not None, the shorter side must be greater than or equal to *min_size*. Default is None. Defaults to None.
- **max_size** (*Optional[int], optional*) – If *max_size* is not None, the longer side must be less than or equal to *max_size*. Defaults to None.
- **resize_side** (*str, optional*) – A str of "longer" or "shorter". Only works when *keep_aspect_ratio* is True. If *resize_side* is "longer", the longer side of image will be resized to *target_size*. If *resize_side* is "shorter", the shorter side of image will be resized to *target_size*. Defaults to "shorter".
- **interpolation_type** (*str, optional*) – A str of "auto", "bilinear", "nearest_neighbor", "bicubic" or "area". Indicate interpolation method used to resize image. Defaults to "auto".
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Sequence includes the result Blob.

Return type Sequence[oneflow_api.BlobDesc]

For example:

```
import oneflow as flow
import oneflow.typing as tp
from typing import Tuple
import numpy as np
import cv2


def _read_images_by_cv(image_files):
    images = [cv2.imread(image_file).astype(np.single) for image_file in image_
↪files]
    return [np.expand_dims(image, axis=0) for image in images]


def _get_images_static_shape(images):
    image_shapes = [image.shape for image in images]
    image_static_shape = np.amax(image_shapes, axis=0)
    assert isinstance(
        image_static_shape, np.ndarray
    ), "image_shapes: {}, image_static_shape: {}".format(
        str(image_shapes), str(image_static_shape)
    )
    image_static_shape = image_static_shape.tolist()
    assert image_static_shape[0] == 1, str(image_static_shape)
    image_static_shape[0] = len(image_shapes)
    return image_static_shape

def _of_image_target_resize(images, image_static_shape, target_size, max_size):
    func_config = flow.FunctionConfig()
    func_config.default_data_type(flow.float)
    func_config.default_logical_view(flow.scope.mirrored_view())

    @flow.global_function(function_config=func_config)
    def image_target_resize_job(images_def: tp.ListListNumpy.
↪Placeholder(shape=image_static_shape, dtype=flow.float)
    ) -> Tuple[tp.ListListNumpy, tp.ListNumpy, tp.ListNumpy]:
        # The input Blob type should be "kTensorBuffer"
        # So we use oneflow.tensor_list_to_tensor_buffer to convert
        images_buffer = flow.tensor_list_to_tensor_buffer(images_def)

        resized_images_buffer, size, scale = flow.image_target_resize(
            images_buffer,
            target_size=target_size,
            max_size=max_size,
            resize_side="shorter",
        )
        # We convert back to "tensorlist" type
        resized_images = flow.tensor_buffer_to_tensor_list(
            resized_images_buffer,
            shape=(target_size, max_size, image_static_shape[-1]),
            dtype=flow.float,
        )
        return resized_images, size, scale

    resized_images, size, scale = image_target_resize_job([images])
    resized_image = resized_images[0]
    size = size[0]
    scale = scale[0]
```

(continues on next page)

(continued from previous page)

```
    return resized_images, size, scale

if __name__ == "__main__":
    img = _read_images_by_cv(['./img/1.jpg'])
    img_shape = _get_images_static_shape(img) # In example is [1, 349, 367, 3]
    target_size = 256
    max_size = 512
    resized_images, size, scale = _of_image_target_resize(img, tuple(img_shape),␣
↪target_size, max_size)
    # Here the shorter side is "349", we resize it to target_size(256)
    # The scale is 256 / 349 = 0.73
    # The longer side will be resized to 367 * scale = 269
    # get the first element from the resized_images (its type is `list.list`)
    print(resized_images[0][0].shape) # (1, 256, 269, 3)
```

oneflow.**in_top_k**(*targets: oneflow_api.BlobDesc, predictions: oneflow_api.BlobDesc, k: Optional[int], name: Optional[str] = None*) → oneflow_api.BlobDesc

Says whether the targets are in the top K predictions.

Parameters

- **targets** (*oneflow_api.BlobDesc*) – A Blob of type int32 or int64.
- **predictions** (*oneflow_api.BlobDesc*) – A Blob of type float32.
- **k** (*Optional[int], optional*) – Number of top elements to look at for computing precision.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob of type bool. Computed Precision at k as a bool Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def intopk_Job(
    targets: tp.Numpy.Placeholder((2,), dtype=flow.int32),
    predictions: tp.Numpy.Placeholder((2, 4), dtype=flow.float32),
) -> tp.Numpy:
    return flow.math.in_top_k(targets, predictions, 1)

targets = np.array([3, 1], dtype=np.int32)
predictions = np.array([[0.0, 1.0, 2.0, 3.0], [3.0, 2.0, 1.0, 0.0],], dtype=np.
↪float32)
out = intopk_Job(targets, predictions)

# out [1 0]
```

class oneflow.**int32**

oneflow_proto_dtype = 5

class oneflow.**int64**

oneflow_proto_dtype = 6

class oneflow.**int8**

oneflow_proto_dtype = 4

oneflow.**inter_job_reuse_mem_strategy**(*strategy_str: str*, *job_set: Optional[oneflow.core.job.job_set_pb2.JobSet] = None*, ***kwargs: _VT*) → None

Set memory sharing strategy for job set.

Parameters

- **strategy_str** – An optional *string* from: *mem_sharing_priority*, *parallelism_priority*
- **custom_parallelism.** (*or*) –
- **job_set** – A *JobSet* object. If None, set default job set.

oneflow.**is_deprecated**(*func_or_class*)

oneflow.**kaiming_initializer**(*shape: Sequence[int], distribution: str = 'random_normal', mode: str = 'fan_in', nonlinearity: str = 'leaky_relu', negative_slope: float = 0.0, data_format: str = 'NCHW'*) → None

Initialize weight according to the method described in *Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification* - He, K. et al. (2015), using a normal or uniform distribution.

When distribution is "random_normal"

The equation is:

$$W \sim N(0, \sqrt{\frac{2}{n}})$$

When distribution is "random_uniform"

The equation is:

$$W \sim U(-\sqrt{\frac{6}{n}}, \sqrt{\frac{6}{n}})$$

If mode is "fan_in", the "n" is the number of input units in the weight Blob.

If mode is "fan_out", the "n" is the number of output units in the weight Blob.

if mode is "fan_avg", the "n" is the average of the number of input and output units in the weight Blob

Parameters

- **shape** (*Sequence[int]*) – Blob shape.
- **distribution** (*str, optional*) – 'random_normal' or 'random_uniform'. Defaults to "random_normal".
- **mode** (*str, optional*) – 'fan_in', 'fan_out' or 'fan_avg'. Defaults to "fan_in".
- **nonlinearity** (*str, optional*) – None, 'tanh', 'sigmoid', 'relu' or 'leaky_relu'. Defaults to "leaky_relu".
- **negative_slope** (*float, optional*) – The negative slope of leaky_relu. Defaults to 0.0.

- **data_format** (*str, optional*) – 'NCHW', 'NHWC'. Defaults to "NCHW".

Raises **NotImplementedError** – Only support normal and uniform distribution

Returns flow.random_normal_initializer or flow.random_uniform_initializer

Return type [type]

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp


def watch_handler(y: tp.Numpy):
    print("out", y)


@flow.global_function()
def kaiming_Job() -> None:
    init = flow.kaiming_initializer(shape=(3, 3),
                                    mode="fan_avg",
                                    nonlinearity="relu")
    blob = flow.get_variable(
        "blob-weight",
        shape=(3, 3),
        initializer=init,
        trainable=True
    )
    flow.watch(blob, watch_handler)


checkpoint = flow.train.CheckPoint()
checkpoint.init()
kaiming_Job()

# out [[ 0.54521346  0.32585594  1.3474437 ]
#      [ 0.30729076 -0.19158769  0.2709008 ]
#      [-0.95830524 -0.05093324  0.28178614]]
```

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def conv2d_kaiming_Job(x: tp.Numpy.Placeholder((1, 256, 32, 32))
) -> tp.Numpy:
    initializer = flow.kaiming_initializer(shape=(1, 256, 32, 32))
    conv2d = flow.layers.conv2d(
        x,
        filters=128,
        kernel_size=3,
        strides=1,
        padding='SAME',
        kernel_initializer=initializer,
```

(continues on next page)

(continued from previous page)

```
        name="Conv2d"
    )
    return conv2d


x = np.random.randn(1, 256, 32, 32).astype(np.float32)
out = conv2d_kaiming_Job(x)

# out.shape (1, 128, 32, 32)
```

oneflow.**load_variables**(*value_dict: Dict[str, Union[oneflow.python.framework.remote_blob.EagerBlobTrait, oneflow.python.framework.check_point_v2.FileBackendVariableBlob, numpy.ndarray]], ignore_mismatch: bool = True*)

Load value in *value_dict* into oneflow variables. For example, if *value_dict* is {'x', np.ones(x_shape)}, the value of variable "x" will all ones. If *ignore_mismatch* is False, an exception will be raised when there is a name in *value_dict* not belonging to any variable.

oneflow.**masked_fill**(*x: oneflow_api.BlobDesc, mask: oneflow_api.BlobDesc, value: Union[float, int], name: Optional[str] = None*) → oneflow_api.BlobDesc

Fill a blob with a given value according to the given mask.

Parameters

- **x** (*oneflow_api.BlobDesc*) – Input Blob.
- **mask** (*oneflow_api.BlobDesc*) – Composed with 0 and 1, the input blob 'x' will be filled with the given value where the mask is 1.
- **value** (*Union[int, int]*) – The value to use for filling the input blob.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Attention: x and mask must be broadcastable to each other. mask must be int type (int8/int32/int64).

Returns The value-filled Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def masked_fill_Job(x: tp.Numpy.Placeholder((4, ), mask: tp.Numpy.Placeholder((4,␣
↪),
                    dtype = flow.int8))->tp.Numpy:
    return flow.masked_fill(x, mask, value=5)

x = np.array([1, 2, 3, 4], dtype=np.float32)
mask = np.array([1, 0, 0, 1], dtype=np.int8)

out = masked_fill_Job(x, mask)

# output [5 2 3 5]
```

oneflow.**matmul**(*a: oneflow_api.BlobDesc*, *b: oneflow_api.BlobDesc*, *transpose_a: bool = False*, *transpose_b: bool = False*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator applies matrix multiplication to two Blobs.

Parameters

- **a** (*oneflow_api.BlobDesc*) – A Blob
- **b** (*oneflow_api.BlobDesc*) – A Blob
- **transpose_a** (*bool, optional*) – Whether to transpose A Blob. Defaults to False.
- **transpose_b** (*bool, optional*) – Whether to transpose B Blob. Defaults to False.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def matmul_Job(A: tp.Numpy.Placeholder((3, 3)),
               B: tp.Numpy.Placeholder((3, 3))
) -> tp.Numpy:
    return flow.linalg.matmul(A, B)


A = np.array([[1, 0, 0],
              [0, 1, 1],
              [0, 0, 1]]).astype(np.float32)
B = np.array([[3, 4, 5],
              [6, 7, 8],
              [9, 10, 11]]).astype(np.float32)
out = matmul_Job(A, B)

# output [[ 3.  4.  5.]
#         [15. 17. 19.]
#         [ 9. 10. 11.]]
```

oneflow.**multi_count_not_finite**(*x: Optional[Sequence[oneflow_api.BlobDesc]] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.**nonzero**(*a: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator finds the indices of input Blob *condition* elements that are non-zero.

Parameters

- **a** (*oneflow_api.BlobDesc*) – The input Blob.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

oneflow.**object_bbox_flip**(*bbox: oneflow_api.BlobDesc, image_size: oneflow_api.BlobDesc, flip_code: Union[int, oneflow_api.BlobDesc], name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator flips the object bounding box.

The flip code corresponds to the different flip mode:

0 (0x00): Non Flip

1 (0x01): Horizontal Flip

16 (0x10): Vertical Flip

17 (0x11): Both Horizontal and Vertical Flip

Parameters

- **bbox** (*oneflow_api.BlobDesc*) – The bounding box.
- **image_size** (*oneflow_api.BlobDesc*) – The size of input image.
- **flip_code** (*Union[int, oneflow_api.BlobDesc]*) – The flip code.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import numpy as np
import oneflow as flow
import oneflow.typing as tp


def _of_object_bbox_flip(bbox_list, image_size, flip_code):
    bbox_shape = _get_bbox_static_shape(bbox_list)
    func_config = flow.FunctionConfig()
    func_config.default_data_type(flow.float)
    func_config.default_logical_view(flow.scope.mirrored_view())

    @flow.global_function(function_config=func_config)
    def object_bbox_flip_job(
        bbox_def: tp.ListListNumpy.Placeholder(
            shape=tuple(bbox_shape), dtype=flow.float
        ),
        image_size_def: tp.ListNumpy.Placeholder(
            shape=image_size.shape, dtype=flow.int32
        ),
    ) -> tp.ListListNumpy:
        bbox_buffer = flow.tensor_list_to_tensor_buffer(bbox_def)
        flip_bbox = flow.object_bbox_flip(bbox_buffer, image_size_def, flip_code)
        return flow.tensor_buffer_to_tensor_list(
            flip_bbox, shape=bbox_shape[1:], dtype=flow.float
        )

    input_bbox_list = [np.expand_dims(bbox, axis=0) for bbox in bbox_list]
    bbox_tensor = object_bbox_flip_job([input_bbox_list], [image_size])
    return bbox_tensor[0]


def _get_bbox_static_shape(bbox_list):
```

(continues on next page)

(continued from previous page)

```
    bbox_shapes = [bbox.shape for bbox in bbox_list]
    bbox_static_shape = np.amax(bbox_shapes, axis=0)
    assert isinstance(
        bbox_static_shape, np.ndarray
    ), "bbox_shapes: {}, bbox_static_shape: {}".format(
        str(bbox_shapes), str(bbox_static_shape)
    )
    bbox_static_shape = bbox_static_shape.tolist()
    bbox_static_shape.insert(0, len(bbox_list))
    return bbox_static_shape

if __name__ == "__main__":
    bbox = np.array([[[20.0, 40.0, 80.0, 160.0],
                      [30.0, 50.0, 70.0, 100.0]]]).astype(np.single) # [x1, y1, x2,␣
↪y2]
    image_size = np.array([[480, 620]]).astype(np.int32)
    bbox_flip =  _of_object_bbox_flip(bbox,
                                      image_size,
                                      flip_code=1) # Horizontal Flip
    print(bbox_flip[0][0])

    # [[399.  40. 459. 160.]
    #  [409.  50. 449. 100.]]
```

oneflow.**object_bbox_scale**(*bbox: oneflow_api.BlobDesc*, *scale: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator scales the input image and the corresponding bounding box. It returns the scaled bounding box.

Parameters

- **bbox** (*oneflow_api.BlobDesc*) – The bounding box.
- **scale** (*oneflow_api.BlobDesc*) – The scale factor.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import numpy as np
import oneflow as flow
import oneflow.typing as tp
import cv2
from typing import Tuple


def _read_images_by_cv(image_files):
    images = [cv2.imread(image_file).astype(np.single) for image_file in image_
↪files]
    return images


def _get_images_static_shape(images):
    image_shapes = [image.shape for image in images]
    image_static_shape = np.amax(image_shapes, axis=0)
    assert isinstance(
```

(continues on next page)

(continued from previous page)

```
        image_static_shape, np.ndarray
    ), "image_shapes: {}, image_static_shape: {}".format(
        str(image_shapes), str(image_static_shape)
    )
    image_static_shape = image_static_shape.tolist()
    image_static_shape.insert(0, len(image_shapes))
    return image_static_shape


def _get_bbox_static_shape(bbox_list):
    bbox_shapes = [bbox.shape for bbox in bbox_list]
    bbox_static_shape = np.amax(bbox_shapes, axis=0)
    assert isinstance(
        bbox_static_shape, np.ndarray
    ), "bbox_shapes: {}, bbox_static_shape: {}".format(
        str(bbox_shapes), str(bbox_static_shape)
    )
    bbox_static_shape = bbox_static_shape.tolist()
    bbox_static_shape.insert(0, len(bbox_list))
    return bbox_static_shape


def _of_target_resize_bbox_scale(images, bbox_list, target_size, max_size):
    image_shape = _get_images_static_shape(images)
    bbox_shape = _get_bbox_static_shape(bbox_list)

    func_config = flow.FunctionConfig()
    func_config.default_data_type(flow.float)
    func_config.default_logical_view(flow.scope.mirrored_view())

    @flow.global_function(function_config=func_config)
    def target_resize_bbox_scale_job(
        image_def: tp.ListListNumpy.Placeholder(
            shape=tuple(image_shape), dtype=flow.float
        ),
        bbox_def: tp.ListListNumpy.Placeholder(
            shape=tuple(bbox_shape), dtype=flow.float
        ),
    ) -> Tuple[tp.ListListNumpy, tp.ListNumpy]:
        images_buffer = flow.tensor_list_to_tensor_buffer(image_def)
        resized_images_buffer, new_size, scale = flow.image_target_resize(
            images_buffer, target_size=target_size, max_size=max_size
        )
        bbox_buffer = flow.tensor_list_to_tensor_buffer(bbox_def)
        scaled_bbox = flow.object_bbox_scale(bbox_buffer, scale)
        scaled_bbox_list = flow.tensor_buffer_to_tensor_list(
            scaled_bbox, shape=bbox_shape[1:], dtype=flow.float
        )
        return scaled_bbox_list, new_size

    input_image_list = [np.expand_dims(image, axis=0) for image in images]
    input_bbox_list = [np.expand_dims(bbox, axis=0) for bbox in bbox_list]
    output_bbox_list, output_image_size = target_resize_bbox_scale_job(
        [input_image_list], [input_bbox_list]
    )
    return output_bbox_list[0], output_image_size[0]
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    images = _read_images_by_cv(['./img/1.jpg', './img/2.jpg'])
    bbox = np.array([[[20.0, 40.0, 80.0, 160.0],
                      [30.0, 50.0, 70.0, 100.0]],
                     [[26.0, 40.0, 86.0, 160.0],
                      [36.0, 56.0, 76.0, 106.0]]]).astype(np.single) # [x1, y1, x2,␣
↪y2]
    bbox, size = _of_target_resize_bbox_scale(images, bbox, 280, 350)
    print(bbox[0])
    print(bbox[1])

    # [[[ 16.0218    32.09169   64.0872   128.36676 ]
    #   [ 24.032698  40.114613  56.076298  80.229225]]]

    # [[[ 24.186047  37.170418  80.        148.68167 ]
    #   [ 33.488373  52.038586  70.69768   98.5016  ]]]
```

oneflow.**object_segmentation_polygon_flip**(*poly: oneflow_api.BlobDesc, image_size: oneflow_api.BlobDesc, flip_code: Union[int, oneflow_api.BlobDesc], name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator flips the segmentation points in image.

The flip code corresponds to the different flip mode:

0 (0x00): Non Flip

1 (0x01): Horizontal Flip

16 (0x10): Vertical Flip

17 (0x11): Both Horizontal and Vertical Flip

Parameters

- **poly** (*oneflow_api.BlobDesc*) – The poly segmentation points.
- **image_size** (*oneflow_api.BlobDesc*) – The image size.
- **flip_code** (*Union[int, oneflow_api.BlobDesc]*) – The filp code.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import numpy as np
import oneflow as flow
import oneflow.typing as tp
import cv2


def _read_images_by_cv(image_files):
    images = [cv2.imread(image_file).astype(np.single) for image_file in image_
↪files]
    return [np.expand_dims(image, axis=0) for image in images]
```

(continues on next page)

(continued from previous page)

```
def _of_object_segm_poly_flip(poly_list, image_size, flip_code):
    poly_shape = _get_segm_poly_static_shape(poly_list)

    func_config = flow.FunctionConfig()
    func_config.default_data_type(flow.float)
    func_config.default_logical_view(flow.scope.mirrored_view())

    @flow.global_function(function_config=func_config)
    def object_segm_poly_flip_job(
        poly_def: tp.ListListNumpy.Placeholder(
            shape=tuple(poly_shape), dtype=flow.float
        ),
        image_size_def: tp.ListNumpy.Placeholder(
            shape=image_size.shape, dtype=flow.int32
        ),
    ) -> tp.ListListNumpy:
        poly_buffer = flow.tensor_list_to_tensor_buffer(poly_def)
        flip_poly = flow.object_segmentation_polygon_flip(
            poly_buffer, image_size_def, flip_code
        )
        return flow.tensor_buffer_to_tensor_list(
            flip_poly, shape=poly_shape[1:], dtype=flow.float
        )

    input_poly_list = [np.expand_dims(poly, axis=0) for poly in poly_list]
    poly_tensor = object_segm_poly_flip_job([input_poly_list], [image_size])
    return poly_tensor[0]


def _get_segm_poly_static_shape(poly_list):
    poly_shapes = [poly.shape for poly in poly_list]
    poly_static_shape = np.amax(poly_shapes, axis=0)
    assert isinstance(
        poly_static_shape, np.ndarray
    ), "poly_shapes: {}, poly_static_shape: {}".format(
        str(poly_shapes), str(poly_static_shape)
    )
    poly_static_shape = poly_static_shape.tolist()
    poly_static_shape.insert(0, len(poly_list))
    return poly_static_shape

if __name__ == "__main__":
    segm_poly_list = []
    segmentations = [[[20.0, 40.0], [80.0, 160.0], [100.0, 210.0]], # Image 1␣
↪segmentation point
                     [[25.0, 45.0], [85.0, 165.0], [105.0, 215.0]]] # Image 2␣
↪segmentation point
    for segmentation in segmentations:
        polygon = []
        for seg in segmentation:
            polygon.extend(seg)
        poly_array = np.array(polygon, dtype=np.single).reshape(-1, 2) # Reshape␣
↪it
        segm_poly_list.append(poly_array)

    image_size = np.array([[480, 620], # Image 1 size
```

(continues on next page)

(continued from previous page)

```
                        [640, 640]]).astype(np.int32) # Image 2 size
    of_segm_poly_list = _of_object_segm_poly_flip(
        segm_poly_list, image_size, flip_code=1
    ) # Horizontal Flip
    print(of_segm_poly_list[0])
    print(of_segm_poly_list[1])

    # of_segm_poly_list[0]
    # [[[460.  40.]
    #   [400. 160.]
    #   [380. 210.]]]

    # of_segm_poly_list[1]
    # [[[615.  45.]
    #   [555. 165.]
    #   [535. 215.]]]
```

oneflow.**object_segmentation_polygon_scale**(*poly: oneflow_api.BlobDesc*, *scale: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator scales the segmentation points in the images.

Parameters

- **poly** (*oneflow_api.BlobDesc*) – The poly segmentation points.
- **scale** (*oneflow_api.BlobDesc*) – The image scale.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import numpy as np
import oneflow as flow
import oneflow.typing as tp
import cv2
from typing import Tuple


def _read_images_by_cv(image_files):
    images = [cv2.imread(image_file).astype(np.single) for image_file in image_
↪files]
    return images


def _get_images_static_shape(images):
    image_shapes = [image.shape for image in images]
    image_static_shape = np.amax(image_shapes, axis=0)
    assert isinstance(
        image_static_shape, np.ndarray
    ), "image_shapes: {}, image_static_shape: {}".format(
        str(image_shapes), str(image_static_shape)
    )
    image_static_shape = image_static_shape.tolist()
    image_static_shape.insert(0, len(image_shapes))
```

(continues on next page)

(continued from previous page)

```
    return image_static_shape


def _get_segm_poly_static_shape(poly_list):
    poly_shapes = [poly.shape for poly in poly_list]
    poly_static_shape = np.amax(poly_shapes, axis=0)
    assert isinstance(
        poly_static_shape, np.ndarray
    ), "poly_shapes: {}, poly_static_shape: {}".format(
        str(poly_shapes), str(poly_static_shape)
    )
    poly_static_shape = poly_static_shape.tolist()
    poly_static_shape.insert(0, len(poly_list))
    return poly_static_shape


def _get_bbox_static_shape(bbox_list):
    bbox_shapes = [bbox.shape for bbox in bbox_list]
    bbox_static_shape = np.amax(bbox_shapes, axis=0)
    assert isinstance(
        bbox_static_shape, np.ndarray
    ), "bbox_shapes: {}, bbox_static_shape: {}".format(
        str(bbox_shapes), str(bbox_static_shape)
    )
    bbox_static_shape = bbox_static_shape.tolist()
    bbox_static_shape.insert(0, len(bbox_list))
    return bbox_static_shape


def _of_object_segm_poly_scale(images, poly_list, target_size, max_size):
    image_shape = _get_images_static_shape(images)
    print(image_shape)
    poly_shape = _get_segm_poly_static_shape(poly_list)
    print("Poly shape is ", poly_shape)
    func_config = flow.FunctionConfig()
    func_config.default_data_type(flow.float)
    func_config.default_logical_view(flow.scope.mirrored_view())

    @flow.global_function(function_config=func_config)
    def object_segm_poly_scale_job(
        image_def: tp.ListListNumpy.Placeholder(
            shape=tuple(image_shape), dtype=flow.float
        ),
        poly_def: tp.ListListNumpy.Placeholder(
            shape=tuple(poly_shape), dtype=flow.float
        ),
    ) -> Tuple[tp.ListListNumpy, tp.ListNumpy]:
        images_buffer = flow.tensor_list_to_tensor_buffer(image_def)
        resized_images_buffer, new_size, scale = flow.image_target_resize(
            images_buffer, target_size=target_size, max_size=max_size
        )
        poly_buffer = flow.tensor_list_to_tensor_buffer(poly_def)
        scaled_poly = flow.object_segmentation_polygon_scale(poly_buffer, scale)
        scaled_poly_list = flow.tensor_buffer_to_tensor_list(
            scaled_poly, shape=poly_shape[1:], dtype=flow.float
        )
        return scaled_poly_list, new_size
```

(continues on next page)

(continued from previous page)

```
    input_image_list = [np.expand_dims(image, axis=0) for image in images]
    input_poly_list = [np.expand_dims(poly, axis=0) for poly in poly_list]

    output_poly_list, output_image_size = object_segm_poly_scale_job(
        [input_image_list], [input_poly_list]
    )

    return output_poly_list[0], output_image_size

if __name__ == "__main__":
    images = _read_images_by_cv(['./img/1.jpg', './img/2.jpg'])
    segm_poly_list = []
    segmentations = [[[20.0, 40.0], [80.0, 160.0], [100.0, 210.0]], # Image 1␣
↪segmentation point
                     [[25.0, 45.0], [85.0, 165.0], [105.0, 215.0]]] # Image 2␣
↪segmentation point

    for segmentation in segmentations:
        polygon = []
        for seg in segmentation:
            polygon.extend(seg)
        poly_array = np.array(polygon, dtype=np.single).reshape(-1, 2) # Reshape␣
↪it
        segm_poly_list.append(poly_array)

    bbox, size = _of_object_segm_poly_scale(images, segm_poly_list, 280, 350)
```

oneflow.**object_segmentation_polygon_to_mask**(*poly: oneflow_api.BlobDesc*, *poly_index: oneflow_api.BlobDesc*, *image_size: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator converts the poly segment points to the segment mask array.

Parameters

- **poly** (*oneflow_api.BlobDesc*) – The poly segment points.
- **poly_index** (*oneflow_api.BlobDesc*) – The poly segment index.
- **image_size** (*oneflow_api.BlobDesc*) – The input image size.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

```
import numpy as np
import oneflow as flow
import oneflow.typing as tp
import cv2
from typing import Tuple


def _read_images_by_cv(image_files):
    images = [cv2.imread(image_file).astype(np.single) for image_file in image_
↪files]
    return images
```

(continues on next page)

(continued from previous page)

```
def _get_images_static_shape(images):
    image_shapes = [image.shape for image in images]
    image_static_shape = np.amax(image_shapes, axis=0)
    assert isinstance(
        image_static_shape, np.ndarray
    ), "image_shapes: {}, image_static_shape: {}".format(
        str(image_shapes), str(image_static_shape)
    )
    image_static_shape = image_static_shape.tolist()
    image_static_shape.insert(0, len(image_shapes))
    return image_static_shape


def _get_segm_poly_static_shape(poly_list, poly_index_list):
    assert len(poly_list) == len(poly_index_list)
    num_images = len(poly_list)
    max_poly_elems = 0
    for poly, poly_index in zip(poly_list, poly_index_list):
        assert len(poly.shape) == 2
        assert len(poly_index.shape) == 2, str(poly_index.shape)
        assert poly.shape[0] == poly_index.shape[0]
        assert poly.shape[1] == 2
        assert poly_index.shape[1] == 3
        max_poly_elems = max(max_poly_elems, poly.shape[0])
    return [num_images, max_poly_elems, 2], [num_images, max_poly_elems, 3]

def _segm_poly_to_tensor(img_segm_poly_list):
    poly_array_list = []
    poly_index_array_list = []
    for img_idx, segm_poly_list in enumerate(img_segm_poly_list):
        img_poly_elem_list = []
        img_poly_index_list = []

        for obj_idx, poly_list in enumerate(segm_poly_list):
            for poly_idx, poly in enumerate(poly_list):
                img_poly_elem_list.extend(poly)
                for pt_idx, pt in enumerate(poly):
                    if pt_idx % 2 == 0:
                        img_poly_index_list.append([pt_idx / 2, poly_idx, obj_
↪idx])

        img_poly_array = np.array(img_poly_elem_list, dtype=np.single).reshape(-1,
↪ 2)
        assert img_poly_array.size > 0, segm_poly_list
        poly_array_list.append(img_poly_array)

        img_poly_index_array = np.array(img_poly_index_list, dtype=np.int32)
        assert img_poly_index_array.size > 0, segm_poly_list
        poly_index_array_list.append(img_poly_index_array)

    return poly_array_list, poly_index_array_list


def _of_poly_to_mask_pipline(
    images, poly_list, poly_index_list, num_segms_list, target_size, max_size
```

(continues on next page)

(continued from previous page)

```
):
    print(len(images))
    print(len(poly_list))

    assert len(images) == len(poly_list)
    assert len(poly_list) == len(poly_index_list)
    image_shape = _get_images_static_shape(images)
    poly_shape, poly_index_shape = _get_segm_poly_static_shape(
        poly_list, poly_index_list
    )
    max_num_segms = max(num_segms_list)

    func_config = flow.FunctionConfig()
    func_config.default_logical_view(flow.scope.mirrored_view())
    func_config.default_data_type(flow.float)


    @flow.global_function(function_config=func_config)
    def poly_to_mask_job(
        image_def: tp.ListListNumpy.Placeholder(
            shape=tuple(image_shape), dtype=flow.float
        ),
        poly_def: tp.ListListNumpy.Placeholder(
            shape=tuple(poly_shape), dtype=flow.float
        ),
        poly_index_def: tp.ListListNumpy.Placeholder(
            shape=tuple(poly_index_shape), dtype=flow.int32
        ),
    ) -> Tuple[tp.ListListNumpy, tp.ListListNumpy]:
        images_buffer = flow.tensor_list_to_tensor_buffer(image_def)
        resized_images_buffer, new_size, scale = flow.image_target_resize(
            images_buffer, target_size=target_size, max_size=max_size
        )
        poly_buffer = flow.tensor_list_to_tensor_buffer(poly_def)
        poly_index_buffer = flow.tensor_list_to_tensor_buffer(poly_index_def)
        scaled_poly_buffer = flow.object_segmentation_polygon_scale(poly_buffer,␣
↪scale)
        mask_buffer = flow.object_segmentation_polygon_to_mask(
            scaled_poly_buffer, poly_index_buffer, new_size
        )
        mask_list = flow.tensor_buffer_to_tensor_list(
            mask_buffer, shape=(max_num_segms, target_size, max_size), dtype=flow.
↪int8
        )
        scaled_poly_list = flow.tensor_buffer_to_tensor_list(
            scaled_poly_buffer, shape=poly_shape[1:], dtype=flow.float
        )
        return mask_list, scaled_poly_list

    input_image_list = [np.expand_dims(image, axis=0) for image in images]
    input_poly_list = [np.expand_dims(poly, axis=0) for poly in poly_list]
    input_poly_index_list = [
        np.expand_dims(poly_index, axis=0) for poly_index in poly_index_list
    ]

    output_mask_list, output_poly_list = poly_to_mask_job(
        [input_image_list], [input_poly_list], [input_poly_index_list]
```

(continues on next page)

(continued from previous page)

```
    )

    return output_mask_list[0], output_poly_list[0]

if __name__ == "__main__":
    images = _read_images_by_cv(['./img/1.jpg', './img/2.jpg'])
    segm_poly_list = []

    segmentations = [[[20.0, 40.0, 80.0, 160.0, 100.0, 210.0, 120.0, 215.0]], #␣
↪Image 1 segmentation point
                     [[24.0, 42.0, 86.0, 168.0, 103.0, 223.0, 125.0, 235.0]]] #␣
↪Image 2 segmentation point

    for segmentation in segmentations:
        polygon = []
        for seg in segmentation:
            polygon.extend(seg)

        poly_array = np.array(polygon, dtype=np.single).reshape(-1, 2) # Reshape␣
↪it
        segm_poly_list.append([poly_array])

    poly_list, poly_index_list = _segm_poly_to_tensor(segm_poly_list)
    num_segms_list = [len(segm_poly_list) for segm_poly_list in segm_poly_list]
    target_size = 280
    max_size = 350
    of_mask_list, of_scaled_poly_list = _of_poly_to_mask_pipline(
        images, poly_list, poly_index_list, num_segms_list, target_size, max_size
    )
    of_mask_list = [
        mask_array.reshape(-1, mask_array.shape[-2], mask_array.shape[-1])
        for mask_array in of_mask_list
    ] # reshape it
```

oneflow.**one_hot**(*indices: oneflow_api.BlobDesc, depth: int, on_value: Union[int, float] = 1, off_value: Union[int, float] = 0, axis: int = -1, dtype: Optional[oneflow.python.framework.dtype.dtype] = None, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator generates a onehot Blob from input Blob.

If input Blob's rank is *N*, the corresponding onehot Blob's rank is *N+1*. The new axis is generated on the specified dimension according to the parameter *axis*.

The locations represented by *indices* take value *on_value*, while other locations take *off_value*

Parameters

- **indices** (*oneflow_api.BlobDesc*) – The input Blob.
- **depth** (*int*) – The length of onehot Blob.
- **on_value** (*Union[int, float], optional*) – The fill value when *indices[i] == i*. Defaults to 1.
- **off_value** (*Union[int, float], optional*) – The fill value when *indice[i] != i*. Defaults to 0.
- **axis** (*int, optional*) – The specified dimension that the new axis is generated on. Defaults to -1.

- **dtype** (*Optional[dtype_util.dtype], optional*) – The output data type, it can be "oneflow.int32", "oneflow.int64", "oneflow.float", "oneflow.double". Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Note: The data type of input blob should be *int32* or *int64*

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def onehot_Job(x: tp.Numpy.Placeholder((4, ), dtype=flow.int32)
) -> tp.Numpy:
    return flow.one_hot(indices=x,
                        depth=5,
                        axis=-1,
                        dtype=flow.int32)


x = np.array([0, 3, 1, 2]).astype(np.int32)
out = onehot_Job(x)

# out [[1 0 0 0 0]
#      [0 0 0 1 0]
#      [0 1 0 0 0]
#      [0 0 1 0 0]]
```

Example 2:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def onehot_Job(x: tp.Numpy.Placeholder((4, ), dtype=flow.int32)
) -> tp.Numpy:
    return flow.one_hot(indices=x,
                        depth=5,
                        axis=0,
                        dtype=flow.int32)


x = np.array([0, 3, 1, 2]).astype(np.int32)
out = onehot_Job(x)

# out [[1 0 0 0]
#      [0 0 1 0]
#      [0 0 0 1]
#      [0 1 0 0]
```

(continues on next page)

(continued from previous page)

```
#       [0 0 0 0]]
```

Returns [description]

Return type oneflow_api.BlobDesc

oneflow.**ones**(*shape: Sequence[int], dtype: Optional[oneflow.python.framework.dtype.dtype] = None, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator creates a Tensor filled with the scalar value *1*.

Parameters

- **shape** (*Sequence[int]*) – The shape of the Tensor.
- **dtype** (*Optional[dtype_util.dtype], optional*) – The data type. Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operator. Defaults to None.

Returns The result Blob filled with value *1*

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp


@flow.global_function()
def ones_job() -> tp.Numpy:
    return flow.ones(shape=(2, 3), dtype=flow.float32)


out = ones_job()

# output: [[1. 1. 1.]
#          [1. 1. 1.]]
```

oneflow.**ones_initializer**(*dtype: oneflow.python.framework.dtype.dtype = <class 'oneflow.python.framework.dtype.float32'>*) → oneflow.core.job.initializer_conf_pb2.InitializerConf

Initializer that generates blobs initialized to 1.

Parameters **dtype** (*dtype_util.dtype, optional*) – Default data type. Defaults to dtype_util.float.

Returns constant_initializer

Return type initializer_conf_util.InitializerConf

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp


def watch_handler(y: tp.Numpy):
```

(continues on next page)

(continued from previous page)

```
    print("out", y)


@flow.global_function()
def ones_Job() -> None:
    init = flow.ones_initializer()
    blob = flow.get_variable(
        "blob-weight",
        shape=(3, ),
        initializer=init,
        trainable=True
    )
    flow.watch(blob, watch_handler)


checkpoint = flow.train.CheckPoint()
checkpoint.init()
ones_Job()

# out [1. 1. 1.]
```

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def conv2d_one_Job(x: tp.Numpy.Placeholder((1, 256, 32, 32))
) -> tp.Numpy:
    initializer = flow.ones_initializer()
    conv2d = flow.layers.conv2d(
        x,
        filters=128,
        kernel_size=3,
        strides=1,
        padding='SAME',
        kernel_initializer=initializer,
        name="Conv2d"
    )
    return conv2d


x = np.random.randn(1, 256, 32, 32).astype(np.float32)
out = conv2d_one_Job(x)

# out.shape (1, 128, 32, 32)
```

oneflow.**ones_like**(*like: oneflow_api.BlobDesc*, *dtype: Optional[oneflow.python.framework.dtype.dtype] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator creates a Blob with all elements set to *1* that has the same shape as *like*.

Parameters

- **like** (*oneflow_api.BlobDesc*) – A Blob.
- **dtype** (*Optional[dtype_util.dtype], optional*) – The data type of Blob.

Defaults to None.

- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def ones_like_Job() -> tp.Numpy:
    constant_blob = flow.constant(value=1.5,
                                  shape=(1, 3, 3),
                                  dtype=flow.float)
    ones_like_blob = flow.ones_like(like=constant_blob,
                                    dtype=flow.float)
    return ones_like_blob


out = ones_like_Job()

# out [[[1. 1. 1.]
#       [1. 1. 1.]
#       [1. 1. 1.]]]
```

oneflow.**pack**(*input: oneflow_api.BlobDesc*, *pack_num: int*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.**pad**(*x: oneflow_api.BlobDesc*, *paddings: Sequence[int]*, *constant_value: Union[int, float] = 0*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator pads the input blob with constant value that user specifies. User can set the amount of padding by setting the parameter *paddings*.

Parameters

- **x** (*oneflow_api.BlobDesc*) – The input Blob
- **paddings** (*Sequence[int]*) – A list of integers to specify the padding width, its length must equal with the length of *x.shape*.
- **constant_value** (*Union[int, float], optional*) – The constant value to pad. Defaults to 0.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Raises **ValueError** – The parameter *paddings* must be a tuple or a list.

Returns The Blob after padding.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np
```

(continues on next page)

(continued from previous page)

```
@flow.global_function()
def pad_Job(x: tp.Numpy.Placeholder((3, 3))
) -> tp.Numpy:
    return flow.pad(x,
                    paddings=((2, 2), (1, 1)),
                    constant_value=5)


x = np.array([[1, 1, 1],
              [1, 1, 1],
              [1, 1, 1]]).astype(np.float32)
out = pad_Job(x)

# out [[5. 5. 5. 5. 5.]
#      [5. 5. 5. 5. 5.]
#      [5. 1. 1. 1. 5.]
#      [5. 1. 1. 1. 5.]
#      [5. 1. 1. 1. 5.]
#      [5. 5. 5. 5. 5.]
#      [5. 5. 5. 5. 5.]]
```

oneflow.**pad_grad**(*x: oneflow_api.BlobDesc, paddings: Sequence[int], constant_value: Union[int, float] = 0, name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.**parallel_cast**(*input: oneflow_api.BlobDesc, name: Optional[str] = None, distribute: Optional[oneflow_api.distribute.Distribute] = None, gradient_distribute: Optional[oneflow_api.distribute.Distribute] = None*) → oneflow_api.BlobDesc

oneflow.**random_normal_initializer**(*mean: float = 0.0, stddev: float = 1.0, seed: Optional[int] = None, dtype: Optional[oneflow.python.framework.dtype.dtype] = None*) → oneflow.core.job.initializer_conf_pb2.InitializerConf

Initializer that generates blob with a normal distribution.

Parameters

- **mean** (*float, optional*) – A python scalar. Mean of the random values to generate.. Defaults to 0.0.
- **stddev** (*float, optional*) – A python scalar. Standard deviation of the random values to generate. Defaults to 1.0.
- **seed** (*Optional[int], optional*) – None. Not support yet. Defaults to None.
- **dtype** (*Optional[dtype_util.dtype], optional*) – . Defaults to None.

Returns Initial configuration

Return type initializer_conf_util.InitializerConf

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp


def watch_handler(y: tp.Numpy):
```

(continues on next page)

(continued from previous page)

```
    print("out", y)


@flow.global_function()
def random_normal_Job() -> None:
    init = flow.random_normal_initializer(mean=1, stddev=1)
    blob = flow.get_variable(
        "blob-weight",
        shape=(3, ),
        initializer=init,
        trainable=True
    )
    flow.watch(blob, watch_handler)


checkpoint = flow.train.CheckPoint()
checkpoint.init()
random_normal_Job()

# out [1.4190257 2.7663114 1.7114428]
```

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def conv2d_random_normal_Job(x: tp.Numpy.Placeholder((1, 256, 32, 32))
) -> tp.Numpy:
    initializer = flow.random_normal_initializer(mean=0, stddev=1)

    conv2d = flow.layers.conv2d(
        x,
        filters=128,
        kernel_size=3,
        strides=1,
        padding='SAME',
        kernel_initializer=initializer,
        name="Conv2d"
    )
    return conv2d


x = np.random.randn(1, 256, 32, 32).astype(np.float32)
out = conv2d_random_normal_Job(x)

# out.shape (1, 128, 32, 32)
```

oneflow.**random_uniform_initializer**(*minval: float = 0*, *maxval: float = 1*, *dtype: oneflow.python.framework.dtype.dtype = <class 'oneflow.python.framework.dtype.float32'>*) → oneflow.core.job.initializer_conf_pb2.InitializerConf

Initializer that generates blobs with a uniform distribution.

Parameters

- **minval** (*float, optional*) – A python scalar. Lower bound of the range of random values to generate. Defaults to 0.
- **maxval** (*float, optional*) – A python scalar. Upper bound of the range of random values to generate. Defaults to 1.
- **dtype** (*dtype_util.dtype, optional*) – Default data type. Defaults to dtype_util.float.

Raises **NotImplementedError** – Do not support such data type.

Returns Initial configuration

Return type initializer_conf_util.InitializerConf

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp


def watch_handler(y: tp.Numpy):
    print("out", y)


@flow.global_function()
def random_uniform_Job() -> None:
    init = flow.random_uniform_initializer(minval=0, maxval=0.5)
    blob = flow.get_variable(
        "blob-weight",
        shape=(3, ),
        initializer=init,
        trainable=True
    )
    flow.watch(blob, watch_handler)


checkpoint = flow.train.CheckPoint()
checkpoint.init()
random_uniform_Job()

# out [0.07557311 0.3943565  0.31875622]
```

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def conv2d_random_uniform_Job(x: tp.Numpy.Placeholder((1, 256, 32, 32))
) -> tp.Numpy:
    initializer = flow.random_uniform_initializer(minval=0, maxval=0.5)

    conv2d = flow.layers.conv2d(
        x,
        filters=128,
```

(continues on next page)

(continued from previous page)

```
        kernel_size=3,
        strides=1,
        padding='SAME',
        kernel_initializer=initializer,
        name="Conv2d"
    )
    return conv2d


x = np.random.randn(1, 256, 32, 32).astype(np.float32)
out = conv2d_random_uniform_Job(x)

# out.shape (1, 128, 32, 32)
```

oneflow.**range**(*start*, *limit=None*, *delta=1*, *dtype=None*, *name='range'*) → oneflow_api.BlobDesc

This operator is similar to python *range*, the difference is that *oneflow.range* generates a Blob.

Parameters

- **start** (*[type]*) – The start of interval. Its type should be *int*.
- **limit** (*[type], optional*) – The limit of interval. Its type should be *int*.
- **delta** (*int, optional*) – The numerical spacing between elements. Defaults to 1.
- **dtype** (*[type], optional*) – The output's data type. Currently we only support *oneflow.int64*. Defaults to None.
- **name** (*str, optional*) – The name for the operation. Defaults to "range".

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp


@flow.global_function()
def range_job()->tp.Numpy:
    with flow.scope.placement("cpu", "0:0"):
        out = flow.range(10, dtype=flow.int64)

    return out

out = range_job()

# out [0 1 2 3 4 5 6 7 8 9]
```

Example2:

```
import oneflow as flow
import oneflow.typing as tp


@flow.global_function()
```

(continues on next page)

(continued from previous page)

```
def range_job()->tp.Numpy:
    with flow.scope.placement("cpu", "0:0"):
        out = flow.range(1, 10, 3, dtype=flow.int64)

    return out

out = range_job()

# out [1 4 7]
```

class oneflow.**record**

oneflow_proto_dtype = 8

oneflow.**reflection_pad2d**(*x: oneflow_api.BlobDesc, padding: Union[int, tuple, list], name: Optional[str] = None*) → oneflow_api.BlobDesc

Pads the input tensor using the reflection of the input boundary.

Parameters

- **x** (*oneflow_api.BlobDesc*) – input blob, only support "NCHW" format.
- **padding** (*Union[int, oneflow_api.BlobDesc]*) – The size or bundary of padding, if is int uses the same padding in all dimension;
- **4-dims tuple, uses** (*if*) –
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns [description]

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def pad_Job(x: tp.Numpy.Placeholder((1, 2, 3, 3))
) -> tp.Numpy:
    return flow.reflection_pad2d(x, padding=[0, 0, 1, 2])


x = np.arange(18).reshape((1, 2, 3, 3)).astype(np.float32)
out = pad_Job(x)

# out [[[[ 5.  4.  3.  4.  5.  4.  3.]
#    [ 2.  1.  0.  1.  2.  1.  0.]
#    [ 5.  4.  3.  4.  5.  4.  3.]
#    [ 8.  7.  6.  7.  8.  7.  6.]
#    [ 5.  4.  3.  4.  5.  4.  3.]]

#   [[14. 13. 12. 13. 14. 13. 12.]
#    [11. 10.  9. 10. 11. 10.  9.]
#    [14. 13. 12. 13. 14. 13. 12.]
```

(continues on next page)

(continued from previous page)

```
#     [17. 16. 15. 16. 17. 16. 15.]
#     [14. 13. 12. 13. 14. 13. 12.]]]]
```

oneflow.**repeat**(*input: oneflow_api.BlobDesc*, *repeat_num: int*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.**reshape**(*x: oneflow_api.BlobDesc, shape: Sequence[int], name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator reshapes a Blob. If the Blob is dynamic, it will call *flow.dynamic_reshape* automatically

We can set one dimension in *shape* as *-1*, the operator will infer the complete shape.

Parameters

- **x** – A *Blob*.
- **shape** – Shape of the output blob.
- **name** – A name for the operation (optional).

Returns A *Blob*, has the same type as *x*.

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def reshape_Job(x: tp.Numpy.Placeholder(shape=(4, 4), dtype=flow.float32)
) -> tp.Numpy:
    reshape_blob = flow.reshape(x,
                                shape=[2, 2, 2, -1])
    return reshape_blob


x = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12],
              [13, 14, 15, 16]]).astype(np.float32)
out = reshape_Job(x)

# out.shape (2, 2, 2, 2)
```

oneflow.**reshape_like**(*x: oneflow_api.BlobDesc*, *like: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator reshapes the Blob x to be the same as Blob *like* .

Parameters

- **x** (*oneflow_api.BlobDesc*) – The input Blob.
- **like** (*oneflow_api.BlobDesc*) – A Blob.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def reshape_like_Job(x: tp.Numpy.Placeholder(shape=(4, 4), dtype=flow.float32)
) -> tp.Numpy:
    like_blob = flow.constant(value=1,
                              dtype=flow.int8,
                              shape=(2, 2, 4))
    reshape_like_blob = flow.reshape_like(x,
                                          like=like_blob)
    return reshape_like_blob


x = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12],
              [13, 14, 15, 16]]).astype(np.float32)
out = reshape_like_Job(x)

# out.shape (2, 2, 4)
```

oneflow.**reverse**(*input: oneflow_api.BlobDesc, axis: Union[int, Sequence[int]], name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator reverses the elements on the assigned axis.

Parameters

- **input** (*oneflow_api.BlobDesc*) – The input Blob.
- **axis** (*Union[int, Sequence[int]]*) – The reverse axis.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Raises

- **ValueError** – The name must be a string.
- **ValueError** – The axis must be a int or a list/tuple of int.
- **ValueError** – The axis is out of range.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def reverse_Job(x: tp.Numpy.Placeholder(shape=(3, 3), dtype=flow.float32)) -> tp.
↪Numpy:
    reverse_blob = flow.reverse(x,
                                axis=0)
    return reverse_blob
```

(continues on next page)

(continued from previous page)

```
x = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]]).astype(np.float32)
out = reverse_Job(x)

# out [[7. 8. 9.]
#      [4. 5. 6.]
#      [1. 2. 3.]]
```

oneflow.**same_padding**(*x: oneflow_api.BlobDesc, padding: Sequence[int], data_format: str, kernel_size: Sequence[int], strides: Sequence[int], dilation_rate: Sequence[int], name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator do the padding in "SAME" mode, It can computes the pad width according to the *kernel_size* and *strides* to keep the size of feature map unchanged after convolution or other operations.

Parameters

- **x** (*oneflow_api.BlobDesc*) – The input blob.
- **padding** (*Sequence[int]*) – The padding mode. It should be "SAME_UPPER" or "SAME_LOWER"
- **data_format** (*[type]*) – The data format of input Blob. If the string starts with "NC", it means the data format is *channel first*, else the data format is *channel last*.
- **kernel_size** (*Sequence[int]*) – The kernel size of operations. Its type should be tuple or list.
- **strides** (*Sequence[int]*) – The strides of operations. Its type should be tuple or list.
- **dilation_rate** (*Sequence[int]*) – The dilation rate of operations. Its type should be tuple or list.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The Blob after padding.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def same_pad_Job(x: tp.Numpy.Placeholder((1, 1, 3, 3))
) -> tp.Numpy:
    return flow.same_padding(x,
                             padding="SAME_UPPER",
                             data_format="NCHW",
                             kernel_size=(3, 3),
                             strides=(1, 1),
                             dilation_rate=(1, 1))


x = np.ones(shape=(1, 1, 3, 3)).astype(np.float32)
```

(continues on next page)

(continued from previous page)

```
out = same_pad_Job(x)

# out [[[[0. 0. 0. 0. 0.]
#        [0. 1. 1. 1. 0.]
#        [0. 1. 1. 1. 0.]
#        [0. 1. 1. 1. 0.]
#        [0. 0. 0. 0. 0.]]]]
```

oneflow.**scatter_nd**(*indices: oneflow_api.BlobDesc, updates: oneflow_api.BlobDesc, shape: Sequence[int], name: Optional[str] = None*)

This operator inserts the elements in *updates* according to the *indices* and create a new Blob.

Parameters

- **indices** (*oneflow_api.BlobDesc*) – The indice of *updates*. Its type should be *flow.int*.
- **updates** (*oneflow_api.BlobDesc*) – The update Blob.
- **shape** (*Sequence[int]*) – The constant tensor shape, the constant tensor elements are all zero.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

Example 1:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def scatter_nd_Job(indice: tp.Numpy.Placeholder(shape=(3, 1), dtype=flow.int32),
                   update: tp.Numpy.Placeholder(shape=(3, ), dtype=flow.float32),
) -> tp.Numpy:
    scatter_blob = flow.scatter_nd(indices=indice,
                                   updates=update,
                                   shape=[8])
    return scatter_blob


indice_array = np.array([[1], [6], [4]]).astype(np.int32)
update_array = np.array([10.2, 5.1, 12.7]).astype(np.float32)
out = scatter_nd_Job(indice_array, update_array)

# [ 0.  10.2  0.   0.  12.7  0.   5.1  0. ]
```

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


```

(continues on next page)

(continued from previous page)

```
@flow.global_function()
def scatter_nd_Job(indice: tp.Numpy.Placeholder(shape=(3, 1), dtype=flow.int32),
                   update: tp.Numpy.Placeholder(shape=(3, 3), dtype=flow.float32),
) -> tp.Numpy:
    scatter_blob = flow.scatter_nd(indices=indice,
                                   updates=update,
                                   shape=[5, 3])
    return scatter_blob


indice_array = np.array([[0], [4], [2]]).astype(np.int32)
update_array = np.array([[1, 1, 1],
                         [2, 2, 2],
                         [3, 3, 3]]).astype(np.float32)
out = scatter_nd_Job(indice_array, update_array)

# out [[1. 1. 1.]
#      [0. 0. 0.]
#      [3. 3. 3.]
#      [0. 0. 0.]
#      [2. 2. 2.]]
```

oneflow.**slice**(*x: oneflow_api.BlobDesc, begin: Sequence[int], size: Sequence[int], name: Optional[str] = None*) → oneflow_api.BlobDesc

Extracts a slice from a tensor.

Parameters

- **x** – A *Blob*.
- **begin** – A list or a tuple, indicate each dimension slice begin, whose length must be equal to x's number of dimensions, the first element of begin must be set to None. (Because the internal op of OneFlow does not support 0-dimension slice at present.)
- **size** – A list or a tuple, indicate each dimension slice size, whose length must be equal to x's number of dimensions, the first element of beign must be set to None.
- **name** – A name for the operation (optional).

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def slice_Job(x: tp.Numpy.Placeholder(shape=(3, 3), dtype=flow.float32)
) -> tp.Numpy:
    slice_blob = flow.slice(x,
                            begin=[None, 0],
                            size=[None, 2])
    return slice_blob

x = np.array([[1, 2, 3],
```

(continues on next page)

(continued from previous page)

```
            [4, 5, 6],
            [7, 8, 9]]).astype(np.float32)
out = slice_Job(x)

# out [[1. 2.]
#      [4. 5.]
#      [7. 8.]]
```

oneflow.**slice_update**(*x: oneflow_api.BlobDesc, update: oneflow_api.BlobDesc, slice_tup_list: Sequence[Tuple[int, int, int]], name: Optional[str] = None*) → oneflow_api.BlobDesc

Update a slice of tensor *x*.

Parameters

- **x** – A *Blob*, whose slice will be updated.
- **update** – A *Blob*, indicate the update content.
- **slice_tup_list** – A list of slice tuple, indicate each dimension slice (start, stop, step).
- **name** – A name for the operation (optional).

oneflow.**slice_v2**(*x: oneflow_api.BlobDesc, slice_tup_list: Sequence[Tuple[int, int, int]], name: Optional[str] = None*) → oneflow_api.BlobDesc

Extracts a slice from a tensor. The *slice_tup_list* assigns the slice indices in each dimension, the format is (start, stop, step). The operator will slice the Blob according to the *slice_top_list*.

Parameters

- **x** – A *Blob*.
- **slice_tup_list** – A list of slice tuple, indicate each dimension slice (start, stop, step).
- **name** – A name for the operation (optional).

Returns The result Blob.

Return type oneflow_api.BlobDesc

Note: Because the internal op of OneFlow does not support 0-dimension slice at present, we should set the zero element in *slice_tup_list* as *None*.

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp
@flow.global_function()
def slicev2_Job(x: tp.Numpy.Placeholder(shape=(3, 6, 9), dtype=flow.float32)
) -> tp.Numpy:
    slicev2_blob = flow.slice_v2(x,
                                 slice_tup_list=[[None, None, None],
                                                 [0, 5, 2], # slice in dimension 1,
↪ extract [0, 2, 4]
                                                 [0, 6, 3]]) # slice in dimension␣
↪2, extract [0, 3]
    return slicev2_blob
x = np.random.randn(3, 6, 9).astype(np.float32)
out = slicev2_Job(x)

# out.shape (3, 3, 2)
```

oneflow.**smooth_l1_loss**(*prediction: oneflow_api.BlobDesc*, *label: oneflow_api.BlobDesc*, *beta: float = 1.0*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the smooth l1 loss.

The equation is:

$$out = \frac{(\beta * x)^2}{2}, |x| < \frac{1}{\beta^2}$$
$$out = |x| - \frac{0.5}{\beta^2}, otherwise$$

Parameters

- **prediction** (*oneflow_api.BlobDesc*) – The prediction Blob
- **label** (*oneflow_api.BlobDesc*) – The label Blob
- **beta** (*float, optional*) – The β in the equation. Defaults to 1.0.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def smooth_l1_loss_Job(prediction: tp.Numpy.Placeholder((5, )),
                       label: tp.Numpy.Placeholder((5, ))
) -> tp.Numpy:
    return flow.smooth_l1_loss(prediction=prediction,
                               label=label)


prediction = np.array([0.1, 0.4, 0.3, 0.5, 0.9]).astype(np.float32)
label = np.array([0.3, 0.9, 2.5, 0.4, 0.3]).astype(np.float32)
out = smooth_l1_loss_Job(prediction, label)

# out [0.02       0.12499999 1.7        0.005      0.17999998]
```

oneflow.**sort**(*input: oneflow_api.BlobDesc*, *axis: int = -1*, *direction: str = 'ASCENDING'*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator sorts the input Blob at specified axis.

Parameters

- **input** (*oneflow_api.BlobDesc*) – A Blob
- **axis** (*int, optional*) – dimension to be sorted. Defaults to the last dim (-1)
- **direction** (*str, optional*) – The direction in which to sort the Blob values. If the direction is "ASCENDING", The order of input will be sorted as ascending, else, the order of input will be sorted as descending. Defaults to "ASCENDING".
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The sorted Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def sort_Job(x: tp.Numpy.Placeholder((5, ))
) -> tp.Numpy:
    return flow.sort(input=x)

x = np.array([10, 2, 9, 3, 7]).astype("float32")
out = sort_Job(x)

# out [ 2.  3.  7.  9. 10.]
```

oneflow.**squeeze**(*input: oneflow_api.BlobDesc*, *axis: Optional[Sequence[int]] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator removes the specified dimention which size is 1 of the input Blob. If the *axis* is not specified, this operator will remove all the dimention which size is 1 of the input Blob.

The amount of element in return value is the same as Blob *input*.

Parameters

- **input** (*oneflow_api.BlobDesc*) – The input Blob.
- **axis** (*Optional[Sequence[int]], optional*) – The axis. Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

Example 1:

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def squeeze_Job(x: tp.Numpy.Placeholder(shape=(1, 1, 1, 3), dtype=flow.int32),
) -> tp.Numpy:
    return flow.squeeze(x, axis=[1, 2])


x = np.array([[[[1, 1, 1]]]]).astype(np.int32)
out = squeeze_Job(x)

# out.shape (1, 3)
```

oneflow.**stack**(*inputs: Sequence[oneflow_api.BlobDesc], axis: int = 0, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator stacks the multiple Blobs on the specified axis.

Parameters

- **inputs** (*Sequence[oneflow_api.BlobDesc]*) – A list of input Blob.
- **axis** (*int*) – The stack axis.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def stack_job(x: tp.Numpy.Placeholder(shape=(2, 4, 6)),
              y: tp.Numpy.Placeholder(shape=(2, 4, 6)))->tp.Numpy:
    out = flow.stack([x, y], axis=2)
    return out

x = np.ones(shape=(2, 4, 6), dtype=np.float32)
y = np.ones(shape=(2, 4, 6), dtype=np.float32)

out = stack_job(x, y)

# output.shape (2, 4, 2, 6)
```

Returns The result Blob.

Return type oneflow_api.BlobDesc

oneflow.**sync_default_session**() → None

Synchronize the default session. Block until every synchronous OneFlow function and its callback finishes running.

oneflow.**sync_dynamic_resize**(*inputs: oneflow_api.BlobDesc*, *size: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Parameters

- **inputs** (*oneflow_api.BlobDesc*) – The input Blob.
- **size** (*oneflow_api.BlobDesc*) – The size of new Blob.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob. Its type is *ListNumpy*.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def sync_dynamic_resize_Job(x: tp.Numpy.Placeholder(shape=(4, 3), dtype=flow.
↪float32),
                            size: tp.Numpy.Placeholder(shape=(1, ), dtype=flow.
↪int32),
```

(continues on next page)

(continued from previous page)

```
) -> tp.ListNumpy:
    resize_Blob = flow.sync_dynamic_resize(inputs=x,
                                           size=size)
    return resize_Blob

x = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9],
              [10, 11, 12]]).astype(np.float32)
size = np.array([2]).astype(np.int32)
out = sync_dynamic_resize_Job(x, size)

# out [array([[1., 2., 3.],
#             [4., 5., 6.]], dtype=float32)]
```

class oneflow.**tensor_buffer**

oneflow_proto_dtype = 10

oneflow.**tensor_buffer_to_tensor**(*x: oneflow_api.BlobDesc, dtype: oneflow.python.framework.dtype.dtype, instance_shape: Sequence[int], name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator converts the Blob's type from TensorBuffer to Tensor. Some operator's output data type is *TensorBuffer*, you can use this operator to convert back to *Tensor*.

Refer to Concept Explanation for more about TensorBuffer.

Parameters

- **x** (*oneflow_api.BlobDesc*) – Input *Blob*.
- **dtype** (*dtype_util.dtype*) – The data dtype.
- **instance_shape** (*Sequence[int]*) – The shape of each TensorBuffer instance.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A *Blob*.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def tensor_buffer_to_tensor_Job(x: tp.Numpy.Placeholder(shape=(4, 16, 64, 64),␣
↪dtype=flow.float32),
) -> tp.Numpy:
    x = flow.tensor_to_tensor_buffer(x,
                                     instance_dims=2)
    return flow.tensor_buffer_to_tensor(x,
                                        instance_shape=(64, 64),
                                        dtype=flow.float)
```

(continues on next page)

(continued from previous page)

```
x = np.random.randn(4, 16, 64, 64).astype(np.float32)
out = tensor_buffer_to_tensor_Job(x)

# out.shape (4, 16, 64, 64)
```

oneflow.**tensor_buffer_to_tensor_list**(*input: oneflow_api.BlobDesc, shape: Sequence[int], dtype: oneflow.python.framework.dtype.dtype, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator converts *TensorBuffer* to *TensorList*.

Refer to Concept Explanation for more about TensorList.

Parameters

- **input** (*oneflow_api.BlobDesc*) – The input Tensor Buffer.
- **shape** (*Sequence[int]*) – The shape of input Tensor Buffer.
- **dtype** (*dtype_util.dtype*) – The data type.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def tensorBuffer_to_tensorList_Job(x: tp.Numpy.Placeholder(shape=(4, 16, 64, 64),␣
↪dtype=flow.float32),
) -> tp.ListListNumpy:
    x = flow.tensor_to_tensor_buffer(x,
                                     instance_dims=3)
    out = flow.tensor_buffer_to_tensor_list(input=x,
                                            shape=(16, 64, 64),
                                            dtype=flow.float32)
    return out

x = np.random.randn(4, 16, 64, 64).astype(np.float32)
out = tensorBuffer_to_tensorList_Job(x)

# out[0][0].shape (1, 16, 64, 64)
```

oneflow.**tensor_list_split**(*input_tensor_list: oneflow_api.BlobDesc, name: Optional[str] = None*) → Tuple[oneflow_api.BlobDesc]

This operator splits the input *TensorList*.

Parameters

- **input_tensor_list** (*oneflow_api.BlobDesc*) – The input *TensorList*.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Tuple of *ListNumpy*.

Return type Tuple[oneflow_api.BlobDesc]

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp
from typing import Tuple


func_config = flow.FunctionConfig()
func_config.default_data_type(flow.float)
func_config.default_logical_view(flow.scope.mirrored_view())
@flow.global_function(function_config=func_config)
def tensorList_split_Job(x: tp.ListListNumpy.Placeholder(shape=(2, 5, 4),␣
↪dtype=flow.float32),
) -> Tuple[tp.ListNumpy, tp.ListNumpy]:
    return flow.tensor_list_split(x)


x = np.random.rand(1, 3, 2).astype(np.float32)
y = np.random.rand(1, 2, 2).astype(np.float32)
out = tensorList_split_Job([[x, y]])

# out[0][0].shape (3, 2)
# out[1][0].shape (2, 2)
```

oneflow.**tensor_list_to_tensor_buffer**(*input: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator converts *TensorList* to *TensorBuffer*.

Refer to Concept Explanation for more about TensorList.

Parameters

- **input** (*oneflow_api.BlobDesc*) – The input *TensorList*.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

func_config = flow.FunctionConfig()
func_config.default_data_type(flow.float)
func_config.default_logical_view(flow.scope.mirrored_view())
@flow.global_function(function_config=func_config)
def tensorList_to_tensorBuffer_Job(x: tp.ListListNumpy.Placeholder(shape=(2, 5,␣
↪4), dtype=flow.float32),
) -> tp.ListListNumpy:
    x = flow.tensor_list_to_tensor_buffer(input=x)
    return flow.tensor_buffer_to_tensor_list(x,
                                             shape=(5, 4),
                                             dtype=flow.float32)

x = np.random.rand(1, 3, 2).astype(np.float32)
```

(continues on next page)

(continued from previous page)

```
y = np.random.rand(1, 2, 2).astype(np.float32)
out = tensorList_to_tensorBuffer_Job([[x, y]])

# out[0][0].shape (1, 3, 2)
```

oneflow.**tensor_scatter_nd_add**(*params: oneflow_api.BlobDesc*, *indices: oneflow_api.BlobDesc*, *updates: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator adds elements from 'updates' to Blob 'params' based on the *indices*.

Parameters

- **params** (*oneflow_api.BlobDesc*) – The input Blob.
- **indices** (*oneflow_api.BlobDesc*) – The indice of *updates*. Its type should be *flow.int32*.
- **updates** (*oneflow_api.BlobDesc*) – The update Blob.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def tensor_scatter_nd_add_Job(x: tp.Numpy.Placeholder(shape=(5, 3), dtype=flow.
↪float32),
                              indice: tp.Numpy.Placeholder(shape=(3, 1), dtype=flow.
↪int32),
                              update: tp.Numpy.Placeholder(shape=(3, 3), dtype=flow.
↪float32),
) -> tp.Numpy:
    scatter_blob = flow.tensor_scatter_nd_add(params=x,
                                              indices=indice,
                                              updates=update)
    return scatter_blob

x = np.array([[1, 2, 3],
              [1, 2, 3],
              [1, 2, 3],
              [1, 2, 3],
              [1, 2, 3]]).astype(np.float32)
indice_array = np.array([[0], [4], [2]]).astype(np.int32)
update_array = np.array([[1, 1, 1],
                         [2, 2, 2],
                         [3, 3, 3]]).astype(np.float32)
out = tensor_scatter_nd_add_Job(x, indice_array, update_array)

# out [[2. 3. 4.]
#      [1. 2. 3.]
#      [4. 5. 6.]
```

(continues on next page)

(continued from previous page)

```
#       [1. 2. 3.]
#       [3. 4. 5.]]
```

oneflow.**tensor_scatter_nd_update**(*params: oneflow_api.BlobDesc*, *indices: oneflow_api.BlobDesc*, *updates: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator inserts the elements in *updates* according to the *indices* into the Blob *params*.

Parameters

- **params** (*oneflow_api.BlobDesc*) – The input Blob.
- **indices** (*oneflow_api.BlobDesc*) – The indice of *updates*. Its type should be *flow.int32*.
- **updates** (*oneflow_api.BlobDesc*) – The update Blob.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def tensor_scatter_nd_Job(x: tp.Numpy.Placeholder(shape=(5, 3), dtype=flow.
↪float32),
                          indice: tp.Numpy.Placeholder(shape=(3, 1), dtype=flow.
↪int32),
                          update: tp.Numpy.Placeholder(shape=(3, 3), dtype=flow.
↪float32),
) -> tp.Numpy:
    scatter_blob = flow.tensor_scatter_nd_update(params=x,
                                                 indices=indice,
                                                 updates=update)
    return scatter_blob

x = np.array([[1, 2, 3],
              [1, 2, 3],
              [1, 2, 3],
              [1, 2, 3],
              [1, 2, 3]]).astype(np.float32)
indice_array = np.array([[0], [4], [2]]).astype(np.int32)
update_array = np.array([[1, 1, 1],
                         [2, 2, 2],
                         [3, 3, 3]]).astype(np.float32)
out = tensor_scatter_nd_Job(x, indice_array, update_array)

# out [[1. 1. 1.]
#      [1. 2. 3.]
#      [3. 3. 3.]
#      [1. 2. 3.]
#      [2. 2. 2.]]
```

oneflow.**tensor_to_tensor_buffer**(*x: oneflow_api.BlobDesc*, *instance_dims: int*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator converts the Blob's type from Tensor to TensorBuffer.

Refer to Concept Explanation for more about TensorBuffer.

Parameters

- **x** (*oneflow_api.BlobDesc*) – Input *Blob*.
- **instance_dims** (*int*) – The dimensions of dynamic tensor instance.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def tensor_buffer_to_tensor_Job(x: tp.Numpy.Placeholder(shape=(4, 16, 64, 64),␣
↪dtype=flow.float32),
) -> tp.Numpy:
    x = flow.tensor_to_tensor_buffer(x,
                                     instance_dims=2)
    return flow.tensor_buffer_to_tensor(x,
                                        instance_shape=(64, 64),
                                        dtype=flow.float)

x = np.random.randn(4, 16, 64, 64).astype(np.float32)
out = tensor_buffer_to_tensor_Job(x)

# out.shape (4, 16, 64, 64)
```

oneflow.**transpose**(*a: oneflow_api.BlobDesc*, *perm: Sequence[int] = None*, *conjugate: bool = False*, *batch_axis_non_change: bool = False*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator transposes the specified axis of input Blob.

Parameters

- **a** (*oneflow_api.BlobDesc*) – The input Blob.
- **perm** (*Sequence[int], optional*) – The list of dimension permutation. Defaults to None.
- **conjugate** (*bool, optional*) – Still Unavailable. Defaults to False.
- **batch_axis_non_change** (*bool, optional*) – Whether to change the batch axis it is a temporary design for solving batch axis infer error in some situations. It will be removed after *batch_axis* has been depreciated. Defaults to False.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Raises **NotImplementedError** – The attribute *conjugate* still unavailable.

Returns A transposed blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def transpose_Job(x: tp.Numpy.Placeholder(shape=(1, 2, 3), dtype=flow.float32)
) -> tp.Numpy:
    transpose_blob = flow.transpose(x,
                                    perm=[2, 0, 1])
    return transpose_blob

x = np.random.randn(1, 2, 3).astype(np.float32)
out = transpose_Job(x)

# out.shape (3, 1, 2)
```

oneflow.**truncated_normal**(*stddev: float = 1.0*) → oneflow.core.job.initializer_conf_pb2.InitializerConf

oneflow.**truncated_normal_initializer**(*mean: float = 0.0*, *stddev: float = 1.0*) → oneflow.core.job.initializer_conf_pb2.InitializerConf

Initializer that generates a truncated normal distribution.

Parameters

- **mean** (*float, optional*) – A scalar (float). Defaults to 0.0.
- **stddev** (*float, optional*) – A scalar (float). Defaults to 1.0.

Returns Initial configuration

Return type initializer_conf_util.InitializerConf

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp


def watch_handler(y: tp.Numpy):
    print("out", y)


@flow.global_function()
def truncated_normal_Job() -> None:
    init = flow.truncated_normal_initializer(mean=1, stddev=1)
    blob = flow.get_variable(
        "blob-weight",
        shape=(3, ),
        initializer=init,
        trainable=True
    )
    flow.watch(blob, watch_handler)


checkpoint = flow.train.CheckPoint()
checkpoint.init()
```

(continues on next page)

(continued from previous page)

```
truncated_normal_Job()

# out [1.8303236  0.09787154 0.83049864]
```

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def conv2d_truncated_normal_Job(x: tp.Numpy.Placeholder((1, 256, 32, 32))
) -> tp.Numpy:
    initializer = flow.truncated_normal_initializer(mean=0, stddev=1)

    conv2d = flow.layers.conv2d(
        x,
        filters=128,
        kernel_size=3,
        strides=1,
        padding='SAME',
        kernel_initializer=initializer,
        name="Conv2d"
    )
    return conv2d


x = np.random.randn(1, 256, 32, 32).astype(np.float32)
out = conv2d_truncated_normal_Job(x)

# out.shape (1, 128, 32, 32)
```

class oneflow.**uint8**

oneflow_proto_dtype = 7

oneflow.**unpack**(*input: oneflow_api.BlobDesc*, *unpack_num: int*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.**unsorted_batch_segment_sum**(*data: oneflow_api.BlobDesc*, *segment_ids: oneflow_api.BlobDesc*, *num_segments: int*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

It is similar with *unsorted_segment_sum*, the difference is that *unsorted_batch_segment_sum* brings a *batch axis*. We can do the segment sum in different batch of data.

For example, the segment id is like:

```
[[0 0 0 1 2 2 3 3],
 [0 0 1 1 2 3 3 3]]
```

Parameters

- **data** (*oneflow_api.BlobDesc*) – Input Blob
- **segment_ids** (*oneflow_api.BlobDesc*) – A Blob with shape (d0, d1). The d0, d1 are the first and second dimension of data.

- **num_segments** (*int*) – num_segments should equal the number of distinct segment IDs.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def unsorted_batch_segment_sum_Job(data: tp.Numpy.Placeholder((3, 4)),
                                   segment_ids: tp.Numpy.Placeholder((3, 4),␣
↪dtype=flow.int32)
)->tp.Numpy:
    return flow.math.unsorted_batch_segment_sum(data, segment_ids, 2)

input_blob = np.array([[1, 2, 3, 4],
                       [1, 2, 3 ,4],
                       [1, 2, 3, 4]]).astype(np.float32)
segment_ids = np.array([[0, 0, 0, 1],
                        [0, 0, 1, 0],
                        [0, 1, 0, 0]]).astype(np.int32)
out = unsorted_batch_segment_sum_Job(input_blob, segment_ids)

# out [[6. 4.]
#      [7. 3.]
#      [8. 2.]]
```

oneflow.**unsorted_segment_sum**(*data: oneflow_api.BlobDesc*, *segment_ids: oneflow_api.BlobDesc*, *num_segments: int*, *axis: int = 0*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Computes the sum along segments of a Blob.

Parameters

- **data** (*oneflow_api.BlobDesc*) – Input Blob
- **segment_ids** (*oneflow_api.BlobDesc*) – A Blob should be the size of the first dimension, with consecutive IDs in the range 0 to k (k < d0).
- **num_segments** (*int*) – num_segments should equal the number of distinct segment IDs.
- **axis** (*int, optional*) – The axis of data. Defaults to 0.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob with the same type of data.

Return type oneflow_api.BlobDesc

For example:

```
# Example 1:
import oneflow as flow
import numpy as np
import oneflow.typing as tp
```

(continues on next page)

(continued from previous page)

```
@flow.global_function()
def unsorted_segment_sumJob(data: tp.Numpy.Placeholder((3, 4)),
                            segment_ids: tp.Numpy.Placeholder((4, ), dtype=flow.
↪int32)
)->tp.Numpy:
    return flow.math.unsorted_segment_sum(data, segment_ids, num_segments=2,␣
↪axis=1)

input_blob = np.array([[1, 2, 3, 4],
                       [5, 6, 7 ,8],
                       [9, 10, 11, 12]]).astype(np.float32)
segment_ids = np.array([0, 1, 0, 1]).astype(np.int32)
out = unsorted_segment_sumJob(input_blob, segment_ids)

# out [[ 4.  6.]
#      [12. 14.]
#      [20. 22.]]

# Example 2
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def unsorted_segment_sumJob(data: tp.Numpy.Placeholder((3, 4)),
                            segment_ids: tp.Numpy.Placeholder((3, ), dtype=flow.
↪int32)
)->tp.Numpy:
    return flow.math.unsorted_segment_sum(data, segment_ids, num_segments=2,␣
↪axis=0)

input_blob = np.array([[1, 2, 3, 4],
                       [5, 6, 7 ,8],
                       [9, 10, 11, 12]]).astype(np.float32)
segment_ids = np.array([0, 1, 0]).astype(np.int32)
out = unsorted_segment_sumJob(input_blob, segment_ids)

#  out [[10. 12. 14. 16.]
#       [ 5.  6.  7.  8.]]
```

oneflow.**unsorted_segment_sum_like**(*data: oneflow_api.BlobDesc*, *segment_ids: oneflow_api.BlobDesc*, *like: oneflow_api.BlobDesc*, *axis: int = 0*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Computes the sum along segments of a Blob, the output shape is the same as the *like* Blob.

Parameters

- **data** (*oneflow_api.BlobDesc*) – Input Blob
- **segment_ids** (*oneflow_api.BlobDesc*) – A Blob should be the size of the first dimension, with consecutive IDs in the range 0 to k (k < d0).
- **like** (*oneflow_api.BlobDesc*) – The input Blob which specifies shape
- **axis** (*int, optional*) – The axis of data. Defaults to 0.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def unsorted_segment_sum_like_Job(data: tp.Numpy.Placeholder((3, 4)),
                                  segment_ids: tp.Numpy.Placeholder((3, ),␣
↪dtype=flow.int32),
                                  like: tp.Numpy.Placeholder((2, 4), dtype=flow.
↪float32)
)->tp.Numpy:
    return flow.math.unsorted_segment_sum_like(data, segment_ids, like, axis=0)

input_blob = np.array([[1, 2, 3, 4],
                       [5, 6, 7 ,8],
                       [9, 10, 11, 12]]).astype(np.float32)
segment_ids = np.array([0, 1, 0]).astype(np.int32)
like = np.zeros(shape=(2, 4), dtype=np.float32)

out = unsorted_segment_sum_like_Job(input_blob, segment_ids, like)

# out [[10. 12. 14. 16.]
#      [ 5.  6.  7.  8.]]
```

oneflow.**user_op_builder**(*op_name*)

Build a wrapper of user op.

For instance::

def myargmax(input: oneflow_api.BlobDesc) -> oneflow_api.BlobDesc: return (flow.user_op_builder("myargmax") .Op("argmax") .Input("in", [input]) .Output("out") .Build() .InferAndTryRun() .RemoteBlobList()[0])

Parameters **op_name** (*str*) – name of new user op

Returns *UserOpConfBuilder* object used to build a wrapper of user op.

Return type UserOpConfBuilder

oneflow.**user_op_module_builder**(*op_type_name*)

oneflow.**variance_scaling_initializer**(*scale: float = 1.0, mode: str = 'fan_in', distribution: str = 'truncated_normal', data_format: str = ''*) → oneflow.core.job.initializer_conf_pb2.InitializerConf

Initializer that generates a truncated normal distribution or a random normal distribution or a random uniform distribution with a scale adapting to it.

When the distribution is "truncated_normal"

The equation is:

$$W \sim N(0, \sqrt{\frac{scale}{n}})$$

If mode is "fan_in", the "n" is the number of input units in the weight Blob.

If mode is "fan_out", the "n" is the number of output units in the weight Blob.

if mode is “fan_avg”, the “n” is the average of the number of input and output units in the weight Blob

Parameters

- **scale** (*float, optional*) – Scaling factor (positive float). Defaults to 1.0.
- **mode** (*str, optional*) – One of “fan_in”, “fan_out”, “fan_avg”. Defaults to “fan_in”.
- **distribution** (*str, optional*) – Random distribution to use. One of “truncated_normal”,. Defaults to “truncated_normal”.
- **data_format** (*str, optional*) – A string be one of “N...C” or “NC...”. Defaults to “”.

Returns Initial configuration

Return type initializer_conf_util.InitializerConf

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp


def watch_handler(y: tp.Numpy):
    print("out", y)


@flow.global_function()
def variance_scale_Job() -> None:
    init = flow.variance_scaling_initializer(scale=2.0, mode="fan_avg")
    blob = flow.get_variable(
        "blob-weight",
        shape=(3, 3),
        initializer=init,
        trainable=True
    )
    flow.watch(blob, watch_handler)


checkpoint = flow.train.CheckPoint()
checkpoint.init()
variance_scale_Job()

# out [[-0.13931477  0.12266728 -0.9434968 ]
#      [-0.49665168  0.10231158 -0.19194333]
#      [-0.7902896  -1.7034698  -0.38695997]]
```

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def conv2d_variance_scaling_Job(x: tp.Numpy.Placeholder((1, 256, 32, 32))
) -> tp.Numpy:
    initializer = flow.variance_scaling_initializer(mode="fan_out")
```

(continues on next page)

(continued from previous page)

```
    conv2d = flow.layers.conv2d(
        x,
        filters=128,
        kernel_size=3,
        strides=1,
        padding='SAME',
        kernel_initializer=initializer,
        name="Conv2d"
    )
    return conv2d


x = np.random.randn(1, 256, 32, 32).astype(np.float32)
out = conv2d_variance_scaling_Job(x)

# out.shape (1, 128, 32, 32)
```

oneflow.**watch**(*blob_watched: oneflow_api.BlobDesc*, *handler_or_prompt: Union[Callable*, *str*, *None] = None*) → None

Register callback for a blob. The callback function will be called after the computation produce the blob finishes. We can use it to watch the values of Blob.

Parameters

- **blob_watched** – a *Blob*
- **handler_or_prompt** – a function has an argument of a *Blob*

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp


def watch_handler(y: tp.Numpy):
    print("out", y)


@flow.global_function()
def watch_Job() -> None:
    init = flow.constant_initializer(2.5)
    variable = flow.get_variable(
        "variable-weight",
        shape=(5, ),
        initializer=init,
        trainable=True
    )
    flow.watch(variable, watch_handler)


checkpoint = flow.train.CheckPoint()
checkpoint.init()
watch_Job()

# out [2.5 2.5 2.5 2.5 2.5]
```

Example 2:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np

def watch_handler(y: tp.Numpy):
    print("out", y)


@flow.global_function()
def watch_Job(x: tp.Numpy.Placeholder((1, 3, 2, 2))
) -> None:
    initializer = flow.truncated_normal(0.1)
    conv2d = flow.layers.conv2d(
        x,
        filters=3,
        kernel_size=1,
        strides=1,
        padding='SAME',
        kernel_initializer=initializer,
        name="Conv2d"
    )

    flow.watch(conv2d, watch_handler)


checkpoint = flow.train.CheckPoint()
checkpoint.init()
x = np.ones(shape=(1, 3, 2, 2)).astype(np.float32)
watch_Job(x)

# out [[[[ 0.03757111  0.03757111]
#         [ 0.03757111  0.03757111]]

#       [[-0.36131713 -0.36131713]
#        [-0.36131713 -0.36131713]]

#       [[-0.12266113 -0.12266113]
#        [-0.12266113 -0.12266113]]]]
```

oneflow.**watch_diff**(*blob_watched: oneflow_api.BlobDesc*, *handler_or_prompt: Union[Callable, str, None] = None*) → None

Register callback for gradient of a blob. The callback will be called after the computation produce the gradient blob finishes.

Parameters

- **blob_watched** – a *Blob*
- **handler_or_prompt** – a function has an argument of a *Blob*

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp


BATCH_SIZE = 20
```

(continues on next page)

(continued from previous page)

```
def watch_diff_handler(blob: tp.Numpy):
    print("watch_diff_handler:", blob, blob.shape, blob.dtype)

@flow.global_function(type="train")
def train_job(
    images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.float),
    labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    initializer = flow.truncated_normal(0.1)
    with flow.scope.placement("gpu", "0:0"):
        reshape = flow.reshape(images, [images.shape[0], -1])
        hidden = flow.layers.dense(
            reshape,
            512,
            activation=flow.nn.relu,
            kernel_initializer=initializer,
            name="hidden",
        )
        logits = flow.layers.dense(
            hidden, 10, kernel_initializer=initializer, name="output"
        )
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(labels, logits,␣
↪name="softmax_loss")

    lr_scheduler = flow.optimizer.PiecewiseConstantScheduler([], [0.1])
    flow.optimizer.SGD(lr_scheduler, momentum=0).minimize(loss)
    flow.watch_diff(logits, watch_diff_handler)
    return loss


if __name__ == "__main__":
    checkpoint = flow.train.CheckPoint()
    checkpoint.init()
    (train_images, train_labels), (test_images, test_labels) = flow.data.load_
↪mnist(
            BATCH_SIZE
    )
    for i, (images, labels) in enumerate(zip(train_images, train_labels)):
        loss = train_job(images, labels)


# watch_diff_handler: [[-1.88834548e-01  2.71021971e-03  2.28271242e-02  7.
↪17673637e-03
#                       4.10183379e-03  8.93106461e-02  2.23669074e-02  3.
↪86103359e-03
#                       3.12465224e-02  5.23346756e-03] .....
```

Example 2:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


BATCH_SIZE = 20
```

(continues on next page)

(continued from previous page)

```
def watch_diff_handler(blob: tp.Numpy):
    print("watch_diff_handler:", blob)


@flow.global_function(type="train")
def watch_matmul_diff_job(
    images: tp.Numpy.Placeholder((3, 3), dtype=flow.float),
) -> None:
    with flow.scope.placement("cpu", "0:0"):
        weight_initializer = flow.constant_initializer(2)
        weight_shape = (3, BATCH_SIZE)
        weight = flow.get_variable(
            "matmultest-weight",
            shape=weight_shape,
            initializer=weight_initializer)
        output = flow.linalg.matmul(images, weight)

    lr_scheduler = flow.optimizer.PiecewiseConstantScheduler([], [0.1])
    flow.optimizer.SGD(lr_scheduler, momentum=0.9).minimize(output)
    flow.watch_diff(weight, watch_diff_handler)


if __name__ == "__main__":
    check_point = flow.train.CheckPoint()
    check_point.init()

    x = np.array([[1, 1, 1],
                  [1, 1, 1],
                  [1, 1, 1]]).astype(np.float32)
    watch_matmul_diff_job(x)

# watch_diff_handler: [[3. 3. 3.]
#                      [3. 3. 3.]
#                      [3. 3. 3.]]
```

Example 3:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


def watch_diff_handler(blob: tp.Numpy):
    print("watch_diff_handler:", blob, blob.shape, blob.dtype)


@flow.global_function(type="train")
def watch_conv_diff_job(
    images: tp.Numpy.Placeholder((1, 1, 4, 4), dtype=flow.float),
) -> None:
    with flow.scope.placement("gpu", "0:0"):
        weight_shape = (1, 1, 3, 3)
        weight_initializer = flow.truncated_normal(0.1)
        weight = flow.get_variable(
            name="conv-weight",
            shape=weight_shape,
            initializer=weight_initializer
```

(continues on next page)

(continued from previous page)

```
        )
        output = flow.nn.conv2d(images, weight, strides=1, padding="VALID")

    lr_scheduler = flow.optimizer.PiecewiseConstantScheduler([], [0.1])
    flow.optimizer.SGD(lr_scheduler, momentum=0.9).minimize(output)
    flow.watch_diff(weight, watch_diff_handler)


if __name__ == "__main__":
    check_point = flow.train.CheckPoint()
    check_point.init()

    x = np.array([[[[ 1.,  2.,  3.,  4.],
                    [ 5.,  6.,  7.,  8.],
                    [ 9., 10., 11., 12.],
                    [13., 14., 15., 16.]]]]).astype(np.float32)

    watch_conv_diff_job(x)

# watch_diff_handler: [[[[14. 18. 22.]
#                        [30. 34. 38.]
#                        [46. 50. 54.]]]]
```

oneflow.**where**(*condition: oneflow_api.BlobDesc*, *x: Optional[oneflow_api.BlobDesc] = None*, *y: Optional[oneflow_api.BlobDesc] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator returns the elements where condition is larger than 0.

If *x* and *y* is None, this operator is equal to *oneflow.argwhere*.

If *x* and *y* both are not None, If the element in condition is larger than 0, it will take the *x* element, else it will take the *y* element.

Parameters

- **condition** (*oneflow_api.BlobDesc*) – The input Blob.
- **x** (*Optional[oneflow_api.BlobDesc], optional*) – A Blob. Defaults to None.
- **y** (*Optional[oneflow_api.BlobDesc], optional*) – A Blob. Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Raises **ValueError** – It is not supported when exactly one of x or y is non-None

Returns The result Blob. Its type is *ListNumpy*.

Return type oneflow_api.BlobDesc

For example:

Example 1:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
```

(continues on next page)

(continued from previous page)

```
def where_Job(condition: tp.Numpy.Placeholder(shape=(5, ), dtype=flow.int32),
              x: tp.Numpy.Placeholder(shape=(5, ), dtype=flow.float32),
              y: tp.Numpy.Placeholder(shape=(5, ), dtype=flow.float32),
) -> tp.ListNumpy:
    return flow.where(condition=condition,
                      x=x,
                      y=y)


condition = np.array([3, 0, 1, 0, 1]).astype(np.int32)
x = np.array([10, 20, 30, 40, 50]).astype(np.float32)
y = np.array([100, 200, 300, 400, 500]).astype(np.float32)
out = where_Job(condition, x, y)

# out [array([ 10., 200.,  30., 400.,  50.], dtype=float32)]
```

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def where_Job(condition: tp.Numpy.Placeholder(shape=(5, ), dtype=flow.int32),
) -> tp.ListNumpy:
    return flow.where(condition=condition)


condition = np.array([3, 0, 1, 0, 1]).astype(np.int32)
out = where_Job(condition)

# out [array([[0],
#             [2],
#             [4]], dtype=int32)]
```

oneflow.**xavier_normal_initializer**(*data_format: str = ''*) → oneflow.core.job.initializer_conf_pb2.InitializerConf

Initializer that generates a Xavier normal distribution.

It also can be called as *oneflow.glorot_normal_initializer*.

The equation is:

$$W \sim N(0, \sqrt{\frac{2}{n_j + n_{j+1}}})$$

N means normal distribution

n_j means the amount of Nth layer parameters

Parameters **data_format** (*str, optional*) – The data format. Defaults to "".

Returns Initial configuration

Return type initializer_conf_util.InitializerConf

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp


def watch_handler(y: tp.Numpy):
    print("out", y)


@flow.global_function()
def xavier_normal_Job() -> None:
    init = flow.xavier_normal_initializer()
    blob = flow.get_variable(
        "blob-weight",
        shape=(3, 3),
        initializer=init,
        trainable=True
    )
    flow.watch(blob, watch_handler)


checkpoint = flow.train.CheckPoint()
checkpoint.init()
xavier_normal_Job()

# out [[ 0.5908121  -0.10804518 -0.6148571 ]
#      [ 1.4007381  -0.08172473  0.36579943]
#      [-0.6461796  -0.15923311  0.33653972]]
```

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def conv2d_xavier_normal_Job(x: tp.Numpy.Placeholder((1, 256, 32, 32))
) -> tp.Numpy:
    initializer = flow.xavier_normal_initializer()
    conv2d = flow.layers.conv2d(
        x,
        filters=128,
        kernel_size=3,
        strides=1,
        padding='SAME',
        kernel_initializer=initializer,
        name="Conv2d"
    )
    return conv2d


x = np.random.randn(1, 256, 32, 32).astype(np.float32)
out = conv2d_xavier_normal_Job(x)

# out.shape (1, 128, 32, 32)
```

oneflow.**xavier_uniform_initializer**(*data_format: str = ''*) → oneflow.core.job.initializer_conf_pb2.InitializerConf

Initializer that generates a Xavier uniform distribution.

It also can be called as *oneflow.glorot_uniform_initializer*.

The equation is:

$$W \sim U(-\sqrt{\frac{6}{n_j + n_{j+1}}}, \sqrt{\frac{6}{n_j + n_{j+1}}})$$

U means uniform distribution

n_j means the amount of Nth layer parameters

Parameters **data_format** (*str, optional*) – The data format. Defaults to "".

Returns Initial configuration

Return type initializer_conf_util.InitializerConf

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp


def watch_handler(y: tp.Numpy):
    print("out", y)


@flow.global_function()
def xavier_uniform_Job() -> None:
    init = flow.xavier_uniform_initializer()
    blob = flow.get_variable(
        "blob-weight",
        shape=(3, 3),
        initializer=init,
        trainable=True
    )
    flow.watch(blob, watch_handler)


checkpoint = flow.train.CheckPoint()
checkpoint.init()
xavier_uniform_Job()

# out [[-0.14424723 -0.9532095  -0.08723891]
#      [-0.8011227  -0.29729813 -0.26769108]
#      [ 0.9208976  -0.5971756  -0.15077025]]
```

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def conv2d_xavier_uniform_Job(x: tp.Numpy.Placeholder((1, 256, 32, 32))
```

(continues on next page)

(continued from previous page)

```
) -> tp.Numpy:
    initializer = flow.xavier_uniform_initializer()
    conv2d = flow.layers.conv2d(
        x,
        filters=128,
        kernel_size=3,
        strides=1,
        padding='SAME',
        kernel_initializer=initializer,
        name="Conv2d"
    )
    return conv2d


x = np.random.randn(1, 256, 32, 32).astype(np.float32)
out = conv2d_xavier_uniform_Job(x)

# out.shape (1, 128, 32, 32)
```

oneflow.**zeros**(*shape: Sequence[int], dtype: Optional[oneflow.python.framework.dtype.dtype] = None, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator creates a Tensor filled with the scalar value *0*.

Parameters

- **shape** (*Sequence[int]*) – The shape of the Tensor.
- **dtype** (*Optional[dtype_util.dtype], optional*) – The data type. Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operator. Defaults to None.

Returns The result Tensor filled with value *0*

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp


@flow.global_function()
def zeros_job() -> tp.Numpy:
    return flow.zeros(shape=(2, 3), dtype=flow.float32)


out = zeros_job()

# output: [[0. 0. 0.]
#          [0. 0. 0.]]
```

oneflow.**zeros_initializer**(*dtype: oneflow.python.framework.dtype.dtype = <class 'oneflow.python.framework.dtype.float32'>*) → oneflow.core.job.initializer_conf_pb2.InitializerConf

Initializer that generates blobs initialized to 0

Parameters **dtype** (*dtype_util.dtype, optional*) – Default data type. Defaults to dtype_util.float.

Returns constant_initializer

Return type initializer_conf_util.InitializerConf

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp


def watch_handler(y: tp.Numpy):
    print("out", y)


@flow.global_function()
def zeros_Job() -> None:
    init = flow.zeros_initializer()
    blob = flow.get_variable(
        "blob-weight",
        shape=(3, ),
        initializer=init,
        trainable=True
    )
    flow.watch(blob, watch_handler)


checkpoint = flow.train.CheckPoint()
checkpoint.init()
zeros_Job()

# out [0. 0. 0.]
```

Example 2:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def conv2d_zero_Job(x: tp.Numpy.Placeholder((1, 256, 32, 32))
) -> tp.Numpy:
    initializer = flow.zeros_initializer()
    conv2d = flow.layers.conv2d(
        x,
        filters=128,
        kernel_size=3,
        strides=1,
        padding='SAME',
        kernel_initializer=initializer,
        name="Conv2d"
    )
    return conv2d


x = np.random.randn(1, 256, 32, 32).astype(np.float32)
out = conv2d_zero_Job(x)
```

(continues on next page)

(continued from previous page)

```
# out.shape (1, 128, 32, 32)
```

oneflow.**zeros_like**(*like: oneflow_api.BlobDesc*, *dtype: Optional[oneflow.python.framework.dtype.dtype] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator creates a Blob that has the same shape as *like* whose all elements are set to *0*.

Parameters

- **like** (*oneflow_api.BlobDesc*) – A Blob.
- **dtype** (*Optional[dtype_util.dtype], optional*) – The data type of Blob. Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def zeros_like_Job() -> tp.Numpy:
    constant_blob = flow.constant(value=1.5,
                                  shape=(1, 3, 3),
                                  dtype=flow.float)
    zeros_like_blob = flow.zeros_like(like=constant_blob,
                                      dtype=flow.float)
    return zeros_like_blob


out = zeros_like_Job()

# out [[[0. 0. 0.]
#       [0. 0. 0.]
#       [0. 0. 0.]]]
```

2.1 Types

oneflow.double

oneflow.float

oneflow.float32

oneflow.float64

oneflow.int32

oneflow.int64

oneflow.int8

CHAPTER

THREE

ONEFLOW.ENV

oneflow.env.**ctrl_port**(*val: int*) → None
Set port number used to control the execution across multiple machines. Same on every machine.

Parameters **val** – a port number accessible to peer machines

oneflow.env.**current_resource**() → oneflow.core.job.resource_pb2.Resource

Get current resources, such as:machine nums, cpu/gpu device nums, epoch network threed num, rdma params...

Returns [description]

Return type resource_util.Resource

oneflow.env.**data_port**(*val: int*) → None
Set port number used to data transfer among multiple machines. Same on every machine.

Parameters **val** – a port number accessible to peer machines

oneflow.env.**init**() → bool
Init environment for job

Returns [description]

Return type bool

oneflow.env.**log_dir**(*val: str*) → None
Specify a dir to store OneFlow's logging files. If not specified, it is *./log* by default.

Parameters **val** (*str*) – string , log file path

oneflow.env.**logbuflevel**(*val: int*) → None

Log messages at a level <= this flag are buffered. Log messages at a higher level are flushed immediately.

Parameters **val** (*int*) – int, number of level

oneflow.env.**logtostderr**(*val: int*) → None
Set whether log messages go to stderr instead of logfiles

Parameters **val** (*int*) – [description]

oneflow.env.**machine**(**val: list*) → None
Set machines' hostnames.

For instance:

```
oneflow.env.machine([{"addr": "192.168.1.1"}, {"addr": "192.168.1.2"}])
```

Parameters **val** – *list*, *tuple* or multiple arguments of *dict*. First in the list is the master machine.

CHAPTER

FOUR

ONEFLOW.CONFIG

oneflow.config.**collect_act_event**(*val: bool = True*) → None
Whether or not collect active event.

Parameters **val** (*bool, optional*) – True or False. Defaults to True.

oneflow.config.**comm_net_worker_num**(*val: int*) → None

Set up the workers number in epoll mode network, If use RDMA mode network, then doesn't need.

Parameters **val** (*int*) – number of workers

oneflow.config.**compute_thread_pool_size**(*val: int*) → None
Set up the size of compute thread pool

Parameters **val** (*int*) – size of thread pool

oneflow.config.**cpu_device_num**(*val: int*) → None
Set number of CPUs on each machine to run oneflow on. Usually you don't need to set this.

Parameters **val** (*int*) – number of CPUs. It is identical on every machine.

oneflow.config.**enable_debug_mode**(*val: bool*) → None
Whether use debug mode or not.

Parameters **val** (*bool*) – True or False

oneflow.config.**enable_legacy_model_io**(*val: bool = True*)
Whether or not use legacy model io.

Parameters **val** (*[type]*) – True or False

oneflow.config.**enable_model_io_v2**(*val*)
Whether or not use version2 of model input/output function.

Parameters **val** (*[type]*) – True or False

oneflow.config.**enable_numa_aware_cuda_malloc_host**(*val: bool = True*) → None
Whether or not let numa know that cuda allocated host's memory.

Parameters **val** (*bool, optional*) – True or False. Defaults to True.

oneflow.config.**enable_tensor_float_32_compute**(*val: bool = True*) → None
Whether or not to enable Tensor-float-32 on supported GPUs

Parameters **val** (*bool, optional*) – True or False. Defaults to True.

oneflow.config.**gpu_device_num**(*val: int*) → None
Set number of GPUs on each machine to run oneflow on.

Parameters

- **val** (*int*) – number of GPUs. It is identical on every machine. In other words,
- **can't specify different number of GPUs you would like to use on each machine.** (*you*) –

oneflow.config.**legacy_model_io_enabled**()

oneflow.config.**load_library**(*val: str*) → None
Load necessary library for job

Parameters **val** (*str*) – path to shared object file

oneflow.config.**load_library_now**(*val: str*) → None
Load necessary library for job now

Parameters **val** (*str*) – path to shared object file

oneflow.config.**machine_num**(*val: int*) → None
Set available number of machine/node for running job .

Parameters **val** (*int*) – available number of machines

oneflow.config.**max_mdsave_worker_num**(*val: int*) → None
Set up max number of workers for mdsave process.

Parameters **val** (*int*) – max number of workers

oneflow.config.**persistence_buf_byte**(*val: int*) → None
Set up buffer size for persistence.

Parameters **val** (*int*) – e.g. 1024(bytes)

oneflow.config.**rdma_mem_block_mbyte**(*val: int*) → None
Set up the memory block size in rdma mode.

Parameters **val** (*int*) – size of block, e.g. 1024(mb)

oneflow.config.**rdma_recv_msg_buf_mbyte**(*val: int*) → None
Set up the buffer size for receiving messages in rama mode

Parameters **val** (*int*) – buffer size, e.g. 1024(mb)

oneflow.config.**reserved_device_mem_mbyte**(*val: int*) → None
Set up the memory size of reserved device

Parameters **val** (*int*) – memory size, e.g. 1024(mb)

oneflow.config.**reserved_host_mem_mbyte**(*val: int*) → None
Set up the memory size of reserved host

Parameters **val** (*int*) – memory size, e.g. 1024(mb)

oneflow.config.**save_downloaded_file_to_local_fs**(*val: bool = True*) → None
Whether or not save downloaded file to local file system.

Parameters **val** (*bool, optional*) – True or False. Defaults to True.

oneflow.config.**thread_enable_local_message_queue**(*val: bool*) → None
Whether or not enable thread using local message queue.

Parameters **val** (*bool*) – True or False

oneflow.config.**use_rdma**(*val: bool = True*) → None

Whether use RDMA to speed up data transmission in cluster nodes or not. if not, then use normal epoll mode.

Parameters **val** (*bool, optional*) – Defaults to True.

CHAPTER

FIVE

ONEFLOW.OPTIMIZER

5.1 Optimizers

class oneflow.optimizer.**Adam**(*lr_scheduler: oneflow.python.ops.optimizer.LrScheduler*, *beta1=0.9*, *beta2=0.999*, *epsilon=1e-08*, *do_bias_correction=False*, *loss_scale_factor: Optional[float] = None*, *grad_clipping: Optional[oneflow.python.ops.optimizer.ClipGradientConf] = None*, *train_step_lbn: Optional[str] = None*, *loss_scale_policy: Optional[oneflow.python.ops.optimizer.LossScalePolicy] = None*)

The optimizer of the Adam algorithm.

This algorithm can adjust the learning rate of each parameter dynamically according to the 1st-moment estimates and the 2nd-moment estimates of gradient.

With bias correction, the equation of parameters updating is:

$$
\begin{aligned}
V_t &= \beta_1 * V_{t-1} + (1 - \beta_1) * grad \\
S_t &= \beta_2 * S_{t-1} + (1 - \beta_2) * grad \odot grad \\
\hat{V}_t &= \frac{V_t}{1 - \beta_1^t} \\
\hat{S}_t &= \frac{S_t}{1 - \beta_2^t} \\
\hat{g} &= learning\_rate * \frac{\hat{V}_t}{\sqrt{\hat{S}_t} + \epsilon} \\
param_{new} &= param_{old} - \hat{g}
\end{aligned}
$$

Without bias correction, the equation of parameters updating is:

$$
\begin{aligned}
V_t &= \beta_1 * V_{t-1} + (1 - \beta_1) * grad \\
S_t &= \beta_2 * S_{t-1} + (1 - \beta_2) * grad \odot grad \\
\hat{g} &= learning\_rate * \frac{V_t}{\sqrt{S_t} + \epsilon} \\
param_{new} &= param_{old} - \hat{g}
\end{aligned}
$$

More details please refer to Adam

Parameters

- **lr_scheduler** (*LrScheduler*) – The scheduler of learning rate.

- **beta1** (*float, optional*) – The exponential weighted average decay rate for the 1st-moment estimates (β_1). Defaults to 0.9.
- **beta2** (*float, optional*) – The exponential weighted average decay rate for the 2rd-moment estimates (β_2). Defaults to 0.999.
- **epsilon** (*[type], optional*) – A small float constant value for numerical stability (ϵ). Defaults to 1e-8.
- **do_bias_correction** (*bool, optional*) – Whether to do the bias correction. Defaults to False.
- **loss_scale_factor** (*Optional[float], optional*) – The scale factor of loss. Defaults to None.
- **grad_clipping** (*Optional[ClipGradientConf], optional*) – The gradient clipping strategy. Defaults to None.
- **train_step_lbn** (*Optional[Text], optional*) – [description]. Defaults to None.
- **loss_scale_policy** (*Optional[LossScalePolicy]*) – The policy of loss scale.

For example:

```
import oneflow as flow
import oneflow.typing as tp

@flow.global_function(type="train")
def train_job(
    images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.float),
    labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    with flow.scope.placement("gpu", "0:0"):
        logits = lenet(images, train=True)
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(
            labels, logits, name="softmax_loss"
        )

    # Set learning rate as 0.001
    lr_scheduler = flow.optimizer.PiecewiseConstantScheduler([], [0.001])
    # Set Adam optimizer
    flow.optimizer.Adam(lr_scheduler, do_bias_correction=False).minimize(loss)

    return loss
```

__init__(*lr_scheduler: oneflow.python.ops.optimizer.LrScheduler, beta1=0.9, beta2=0.999, epsilon=1e-08, do_bias_correction=False, loss_scale_factor: Optional[float] = None, grad_clipping: Optional[oneflow.python.ops.optimizer.ClipGradientConf] = None, train_step_lbn: Optional[str] = None, loss_scale_policy: Optional[oneflow.python.ops.optimizer.LossScalePolicy] = None*)

Initialize self. See help(type(self)) for accurate signature.

class oneflow.optimizer.**AdamW**(*lr_scheduler: oneflow.python.ops.optimizer.LrScheduler, beta1=0.9, beta2=0.999, epsilon=1e-08, do_bias_correction=False, loss_scale_factor: Optional[float] = None, weight_decay: Optional[float] = None, weight_decay_includes: Union[Sequence[str], str, None] = None, weight_decay_excludes: Union[Sequence[str], str, None] = None, grad_clipping: Optional[oneflow.python.ops.optimizer.ClipGradientConf] = None, train_step_lbn: Optional[str] = None, loss_scale_policy: Optional[oneflow.python.ops.optimizer.LossScalePolicy] = None*)

The optimizer of the Adam-weight-decay algorithm.

If we use L2 regularization,

it will be invalid due to the adaptive learning rate in Adam optimizer

(More details please refer to Adam-weight-decay).

So we use Adam-weight-decay algorithm to solve this problem.

With bias correction, the equation of parameters updating is:

$$
\begin{aligned}
V_t &= \beta_1 * V_{t-1} + (1 - \beta_1) * grad \\
S_t &= \beta_2 * S_{t-1} + (1 - \beta_2) * grad \odot grad \\
\hat{V}_t &= \frac{V_t}{1 - \beta_1^t} \\
\hat{S}_t &= \frac{S_t}{1 - \beta_2^t} \\
\hat{g} &= learning\_rate * (\frac{\hat{V}_t}{\sqrt{\hat{S}_t} + \epsilon} + \lambda * param_{old}) \\
param_{new} &= param_{old} - \hat{g}
\end{aligned}
$$

Without bias correction, the equation of parameters updating is:

$$
\begin{aligned}
V_t &= \beta_1 * V_{t-1} + (1 - \beta_1) * grad \\
S_t &= \beta_2 * S_{t-1} + (1 - \beta_2) * grad \odot grad \\
\hat{g} &= learning\_rate * (\frac{V_t}{\sqrt{S_t} + \epsilon} + \lambda * param_{old}) \\
param_{new} &= param_{old} - \hat{g}
\end{aligned}
$$

Parameters

- **lr_scheduler** (*LrScheduler*) – The scheduler of learning rate.
- **beta1** (*float, optional*) – The exponential weighted average decay rate for the 1st-moment estimates (β_1). Defaults to 0.9.
- **beta2** (*float, optional*) – The exponential weighted average decay rate for the 2rd-moment estimates (β_2). Defaults to 0.999.
- **epsilon** (*[type], optional*) – A small float constant value for numerical stability (ϵ). Defaults to 1e-8.
- **do_bias_correction** (*bool, optional*) – Whether to do the bias correction. Defaults to False.
- **loss_scale_factor** (*Optional[float], optional*) – The scale factor of loss. Defaults to None.

- **weight_decay** (*Optional[float], optional*) – The weight decay factor (In the equation is λ). Defaults to None.
- **weight_decay_includes** (*Optional[Union[Sequence[Text], Text]], optional*) – The name of the model parameters that use weight decay. Defaults to None.
- **weight_decay_excludes** (*Optional[Union[Sequence[Text], Text]], optional*) – The name of the model parameters that do not use weight decay. Defaults to None.
- **grad_clipping** (*Optional[ClipGradientConf], optional*) – The gradient clipping strategy. Defaults to None.
- **train_step_lbn** (*Optional[Text], optional*) – [description]. Defaults to None.
- **loss_scale_policy** (*Optional[LossScalePolicy]*) – The policy of loss scale.

Note: Only one of *weight_decay_includes* and *weight_decay_excludes* can be set. If both are None, all the model parameters will use weight decay.

For example:

```
import oneflow as flow
import oneflow.typing as tp

@flow.global_function(type="train")
def train_job(
    images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.float),
    labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    with flow.scope.placement("gpu", "0:0"):
        logits = lenet(images, train=True)
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(
            labels, logits, name="softmax_loss"
        )

    # Set learning rate as 0.001
    lr_scheduler = flow.optimizer.PiecewiseConstantScheduler([], [0.001])
    # Set AdamW optimizer, weight_decay factor is 0.00005
    flow.optimizer.AdamW(lr_scheduler,
            do_bias_correction=False, weight_decay=0.00005).minimize(loss)

    return loss
```

__init__(*lr_scheduler: oneflow.python.ops.optimizer.LrScheduler, beta1=0.9, beta2=0.999, epsilon=1e-08, do_bias_correction=False, loss_scale_factor: Optional[float] = None, weight_decay: Optional[float] = None, weight_decay_includes: Union[Sequence[str], str, None] = None, weight_decay_excludes: Union[Sequence[str], str, None] = None, grad_clipping: Optional[oneflow.python.ops.optimizer.ClipGradientConf] = None, train_step_lbn: Optional[str] = None, loss_scale_policy: Optional[oneflow.python.ops.optimizer.LossScalePolicy] = None*)

Initialize self. See help(type(self)) for accurate signature.

class oneflow.optimizer.**CosineScheduler**(*base_lr: float, steps: int, alpha: float = 0.0, warmup: Optional[oneflow.python.ops.optimizer.WarmupConf] = None*)

This operator creates a Cosine decayed learning rate scheduler.

Before the steps are specified by user, the learning rate will be updated as:

$$cos\_decay = 0.5 * (1 + cos(\pi * \frac{current\_batch}{decayed\_batch}))$$
$$decay\_factor = (1 - \alpha) * cos\_decay + \alpha$$
$$learning\_rate = base\_learning\_rate * decay\_factor$$

After the steps specified by user, the learning rate will be :

$$learning\_rate = base\_learning\_rate * \alpha$$

Parameters

- **base_lr** (*float*) – The base learning rate ($base\_learning\_rate$)
- **steps** (*int*) – The decay steps in the scheduler ($decayed\_batch$)
- **alpha** (*float, optional*) – The learning rate scale factor (α). Defaults to 0.0.
- **warmup** (*Optional[WarmupConf], optional*) – The warmup strategy. Defaults to None.

For example:

```
import oneflow as flow
import oneflow.typing as tp

@flow.global_function(type="train")
def train_job(
    images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.float),
    labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    with flow.scope.placement("gpu", "0:0"):
        logits = lenet(images, train=True)
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(
            labels, logits, name="softmax_loss"
        )

    lr_scheduler = flow.optimizer.CosineScheduler(base_lr=0.01,
                                                  steps=10,
                                                  alpha=0.1)
    flow.optimizer.Adam(lr_scheduler).minimize(loss)

    return loss
```

__init__(*base_lr: float*, *steps: int*, *alpha: float = 0.0*, *warmup: Optional[oneflow.python.ops.optimizer.WarmupConf] = None*)
Initialize self. See help(type(self)) for accurate signature.

property learning_rate_decay_conf

class oneflow.optimizer.**CustomScheduler**(*lbn: str*)

__init__(*lbn: str*)
Initialize self. See help(type(self)) for accurate signature.

property learning_rate_decay_conf

class oneflow.optimizer.**ExponentialScheduler**(*base_lr: float*, *steps: int*, *decay_rate: float*, *staircase=False*, *warmup: Optional[oneflow.python.ops.optimizer.WarmupConf] = None*)

This operator creates a exponential decayed learning rate scheduler.

The learning rate will be updated as follows:

If staircase is set to False, the equation is:

$$pow = \frac{current\_batch}{decay\_batch}$$
$$learning\_rate = base\_learning\_rate * decay\_rate^{pow}$$

If staircase is set to True, the equation is:

$$pow = floor(\frac{current\_batch}{decay\_batch})$$
$$learning\_rate = base\_learning\_rate * decay\_rate^{pow}$$

Parameters

- **base_lr** (*float*) – The base learning rate
- **steps** (*int*) – The decay steps
- **decay_rate** (*float*) – The decay rate
- **staircase** (*bool, optional*) – If staircase is True, the scheduler decay the learning rate at discrete intervals. Defaults to False.
- **warmup** (*Optional[WarmupConf], optional*) – The warmup strategy. Defaults to None.

For example:

__init__(*base_lr: float*, *steps: int*, *decay_rate: float*, *staircase=False*, *warmup: Optional[oneflow.python.ops.optimizer.WarmupConf] = None*)

Initialize self. See help(type(self)) for accurate signature.

property learning_rate_decay_conf

class oneflow.optimizer.**InverseTimeScheduler**(*base_lr: float*, *steps: int*, *decay_rate: float*, *staircase: bool = False*, *warmup: Optional[oneflow.python.ops.optimizer.WarmupConf] = None*)

This operator creates a inverse time decayed learning rate scheduler.

The learning rate will be updated as follows:

If staircase is set to False, the equation is:

$$step\_ratio = \frac{current\_batch}{decay\_batch}$$
$$learning\_rate = \frac{base\_learning\_rate}{1 + decay\_rate * step\_ratio}$$

If staircase is set to True, the equation is:

$$step\_ratio = \frac{current\_batch}{decay\_batch}$$
$$learning\_rate = \frac{base\_learning\_rate}{1 + floor(decay\_rate * step\_ratio)}$$

Parameters

- **base_lr** (*float*) – The base learning rate
- **steps** (*int*) – The decay steps
- **decay_rate** (*float*) – The decay rate
- **staircase** (*bool, optional*) – If staircase is True, the scheduler decay the learning rate at discrete intervals. Defaults to False.
- **warmup** (*Optional[WarmupConf], optional*) – The warmup strategy. Defaults to None.

For example:

```
import oneflow as flow
import oneflow.typing as tp

@flow.global_function(type="train")
def train_job(
        images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.
↪float),
        labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    with flow.scope.placement("gpu", "0:0"):
        logits = lenet(images, train=True)
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(
            labels, logits, name="softmax_loss"
        )

    lr_scheduler = flow.optimizer.InverseTimeScheduler(base_lr=0.1,
                                                       steps=5,
                                                       decay_rate=0.9)
    flow.optimizer.SGD(lr_scheduler, momentum=0.9).minimize(loss)

    return loss
```

__init__(*base_lr: float, steps: int, decay_rate: float, staircase: bool = False, warmup: Optional[oneflow.python.ops.optimizer.WarmupConf] = None*)

Initialize self. See help(type(self)) for accurate signature.

property learning_rate_decay_conf

class oneflow.optimizer.**LAMB**(*lr_scheduler: oneflow.python.ops.optimizer.LrScheduler, beta1: float = 0.9, beta2: float = 0.999, epsilon: float = 1e-06, loss_scale_factor: Optional[float] = None, grad_clipping: Optional[oneflow.python.ops.optimizer.ClipGradientConf] = None, train_step_lbn: Optional[str] = None, loss_scale_policy: Optional[oneflow.python.ops.optimizer.LossScalePolicy] = None*)

Parameters

- **lr_scheduler** (*LrScheduler*) – The scheduler of learning rate.
- **beta1** (*float, optional*) – The exponential weighted average decay rate for the 1st-moment estimates (β_1). Defaults to 0.9.
- **beta2** (*float, optional*) – The exponential weighted average decay rate for the 2rd-moment estimates (β_2). Defaults to 0.999.
- **epsilon** (*[type], optional*) – A small float constant value for numerical stability (ϵ). Defaults to 1e-6.

- **loss_scale_factor** (*Optional[float], optional*) – The scale factor of loss. Defaults to None.
- **grad_clipping** (*Optional[ClipGradientConf], optional*) – The gradient clipping strategy. Defaults to None.
- **train_step_lbn** (*Optional[Text], optional*) – [description]. Defaults to None.
- **loss_scale_policy** (*Optional[LossScalePolicy]*) – The policy of loss scale.

__init__(*lr_scheduler: oneflow.python.ops.optimizer.LrScheduler, beta1: float = 0.9, beta2: float = 0.999, epsilon: float = 1e-06, loss_scale_factor: Optional[float] = None, grad_clipping: Optional[oneflow.python.ops.optimizer.ClipGradientConf] = None, train_step_lbn: Optional[str] = None, loss_scale_policy: Optional[oneflow.python.ops.optimizer.LossScalePolicy] = None*)

Initialize self. See help(type(self)) for accurate signature.

class oneflow.optimizer.**LARS**(*lr_scheduler: oneflow.python.ops.optimizer.LrScheduler, momentum_beta: float = 0.9, epsilon: float = 1e-09, lars_coefficient: float = 0.0001, loss_scale_factor: Optional[float] = None, grad_clipping: Optional[oneflow.python.ops.optimizer.ClipGradientConf] = None, train_step_lbn: Optional[str] = None, loss_scale_policy: Optional[oneflow.python.ops.optimizer.LossScalePolicy] = None*)

The optimizer of the LARS algorithm.

The equation of parameters updating is:

$$local\_learning\_rate = learning\_rate * lars\_coeff * \frac{\|parm_{old}\|}{\epsilon + \|grad\|}$$

$$momentum_t = \beta * momentum_{t-1} + local\_learning\_rate * (grad)$$

$$param_{new} = param_{old} - momentum_t$$

Parameters

- **lr_scheduler** (*LrScheduler*) – The scheduler of learning rate.
- **momentum_beta** (*float, optional*) – The momentum factor (β). Defaults to 0.9.
- **epsilon** (*float, optional*) – A small float constant value for numerical stability (ϵ). Defaults to 1e-9.
- **lars_coefficient** (*float, optional*) – The coefficient factor, it defines how much we trust the layer to change its weights ($lars\_coeff$). Defaults to 0.0001.
- **loss_scale_factor** (*Optional[float], optional*) – The scale factor of loss. Defaults to None.
- **grad_clipping** (*Optional[ClipGradientConf], optional*) – The gradient clipping strategy. Defaults to None.
- **train_step_lbn** (*Optional[Text], optional*) – [description]. Defaults to None.
- **loss_scale_policy** (*Optional[LossScalePolicy]*) – The policy of loss scale.

For example:

```
import oneflow as flow
import oneflow.typing as tp
```

(continues on next page)

(continued from previous page)

```
@flow.global_function(type="train")
def train_job(
        images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.float),
        labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    with flow.scope.placement("gpu", "0:0"):
        logits = lenet(images, train=True)
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(
            labels, logits, name="softmax_loss"
        )
    # Set learning rate as 0.1
    lr_scheduler = flow.optimizer.PiecewiseConstantScheduler([], [0.1])
    # Set LARS optimizer, momentum factor is 0.9
    flow.optimizer.LARS(lr_scheduler, momentum_beta=0.9).minimize(loss)

    return loss
```

__init__(*lr_scheduler: oneflow.python.ops.optimizer.LrScheduler*, *momentum_beta: float = 0.9*, *epsilon: float = 1e-09*, *lars_coefficient: float = 0.0001*, *loss_scale_factor: Optional[float] = None*, *grad_clipping: Optional[oneflow.python.ops.optimizer.ClipGradientConf] = None*, *train_step_lbn: Optional[str] = None*, *loss_scale_policy: Optional[oneflow.python.ops.optimizer.LossScalePolicy] = None*)

Initialize self. See help(type(self)) for accurate signature.

class oneflow.optimizer.**LazyAdam**(*lr_scheduler: oneflow.python.ops.optimizer.LrScheduler*, *beta1: float = 0.9*, *beta2: float = 0.999*, *epsilon: float = 1e-08*, *loss_scale_factor: Optional[float] = None*, *grad_clipping: Optional[oneflow.python.ops.optimizer.ClipGradientConf] = None*, *train_step_lbn: Optional[str] = None*, *loss_scale_policy: Optional[oneflow.python.ops.optimizer.LossScalePolicy] = None*)

The optimizer of the LazyAdam algorithm.

This algorithm can adjust the learning rate of each parameter dynamically according to the 1st-moment estimates and the 2nd-moment estimates of the gradient.

The difference between Adam optimizer and LazyAdam optimizer is that LazyAdam only updates the element that has gradient in the current batch, it is faster than Adam optimizer.

$$
\begin{aligned}
V_t &= \beta_1 * V_{t-1} + (1 - \beta_1) * grad \\
S_t &= \beta_2 * S_{t-1} + (1 - \beta_2) * grad \odot grad \\
\hat{g} &= learning\_rate * \frac{V_t}{\sqrt{S_t} + \epsilon} \\
param_{new} &= param_{old} - \hat{g}
\end{aligned}
$$

Parameters

- **lr_scheduler** (*LrScheduler*) – The scheduler of learning rate.
- **beta1** (*float, optional*) – The exponential weighted average decay rate for the 1st-moment estimates (β_1). Defaults to 0.9.
- **beta2** (*float, optional*) – The exponential weighted average decay rate for the 2rd-moment estimates (β_2). Defaults to 0.999.

- **epsilon** (*[type], optional*) – A small float constant value for numerical stability (ϵ). Defaults to 1e-8.
- **loss_scale_factor** (*Optional[float], optional*) – The scale factor of loss. Defaults to None.
- **grad_clipping** (*Optional[ClipGradientConf], optional*) – The gradient clipping strategy. Defaults to None.
- **train_step_lbn** (*Optional[Text], optional*) – [description]. Defaults to None.
- **loss_scale_policy** (*Optional[LossScalePolicy]*) – The policy of loss scale.

For example:

```
import oneflow as flow
import oneflow.typing as tp

@flow.global_function(type="train")
def train_job(
    images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.float),
    labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    with flow.scope.placement("gpu", "0:0"):
        logits = lenet(images, train=True)
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(
            labels, logits, name="softmax_loss"
        )
    # Set learning rate as 0.001
    lr_scheduler = flow.optimizer.PiecewiseConstantScheduler([], [0.001])
    # Set LazyAdam optimizer
    flow.optimizer.LazyAdam(lr_scheduler).minimize(loss)

    return loss
```

__init__(*lr_scheduler: oneflow.python.ops.optimizer.LrScheduler, beta1: float = 0.9, beta2: float = 0.999, epsilon: float = 1e-08, loss_scale_factor: Optional[float] = None, grad_clipping: Optional[oneflow.python.ops.optimizer.ClipGradientConf] = None, train_step_lbn: Optional[str] = None, loss_scale_policy: Optional[oneflow.python.ops.optimizer.LossScalePolicy] = None*)

Initialize self. See help(type(self)) for accurate signature.

class oneflow.optimizer.**LinearCosineScheduler**(*base_lr: float, steps: int, num_periods: float = 0.5, alpha: float = 0.0, beta: float = 0.001, warmup: Optional[oneflow.python.ops.optimizer.WarmupConf] = None*)

This operator creates a linear cosine decayed learning rate scheduler.

The learning rate will be updated as follows:

$$current\_batch = min(current\_batch, decay\_batch)$$

$$linear\_decay = \frac{(decay\_batch - current\_batch)}{decay\_batch}$$

$$cosine\_decay = 0.5 * (1.0 + cos(2 * \pi * num\_periods * \frac{current\_batch}{decay\_batch}))$$

$$decay\_factor = (\alpha + linear\_decay) * cosine\_decay + \beta$$

$$learning\_rate = base\_learning\_rate * decay\_factor$$

Parameters

- **base_lr** (*float*) – The base learning rate
- **steps** (*int*) – The decay steps
- **num_periods** (*float, optional*) – The number of decay periods. Defaults to 0.5.
- **alpha** (*float, optional*) – The α in equation. Defaults to 0.0.
- **beta** (*float, optional*) – The β in equation. Defaults to 0.001.
- **warmup** (*Optional[WarmupConf], optional*) – The warmup strategy. Defaults to None.

For example:

```
import oneflow as flow
import oneflow.typing as tp

@flow.global_function(type="train")
def train_job(
        images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.
↪float),
        labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    with flow.scope.placement("gpu", "0:0"):
        logits = lenet(images, train=True)
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(
            labels, logits, name="softmax_loss"
        )

    lr_scheduler = flow.optimizer.LinearCosineScheduler(base_lr=0.1,
                                                        steps=10)
    flow.optimizer.SGD(lr_scheduler, momentum=0.9).minimize(loss)

    return loss
```

__init__(*base_lr: float*, *steps: int*, *num_periods: float = 0.5*, *alpha: float = 0.0*, *beta: float = 0.001*, *warmup: Optional[oneflow.python.ops.optimizer.WarmupConf] = None*)

Initialize self. See help(type(self)) for accurate signature.

property learning_rate_decay_conf

class oneflow.optimizer.**NaturalExpScheduler**(*base_lr: float*, *steps: int*, *decay_rate: float*, *staircase: bool = False*, *warmup: Optional[oneflow.python.ops.optimizer.WarmupConf] = None*)

This operator creates a natural exponential decayed learning rate scheduler.

The learning rate will be updated as follows:

If staircase is set to False, the equation is:

$$step\_ratio = \frac{current\_batch}{decay\_batch}$$

$$learning\_rate = base\_learning\_rate * e^{-decay\_rate * step\_ratio}$$

If staircase is set to True, the equation is:

$$step\_ratio = \frac{current\_batch}{decay\_batch}$$

$$learning\_rate = base\_learning\_rate * e^{-decay\_rate * floor(step\_ratio)}$$

Parameters

- **base_lr** (*float*) – The base learning rate
- **steps** (*int*) – The decay steps
- **decay_rate** (*float*) – The decay rate
- **staircase** (*bool, optional*) – If staircase is True, the scheduler decay the learning rate at discrete intervals. Defaults to False.
- **warmup** (*Optional[WarmupConf], optional*) – The warmup strategy. Defaults to None.

For example:

```
import oneflow as flow
import oneflow.typing as tp

@flow.global_function(type="train")
def train_job(
        images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.
↪float),
        labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    with flow.scope.placement("gpu", "0:0"):
        logits = lenet(images, train=True)
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(
            labels, logits, name="softmax_loss"
        )

    lr_scheduler = flow.optimizer.NaturalExpScheduler(base_lr=0.1,
                                                      steps=10,
                                                      decay_rate=0.5)
    flow.optimizer.SGD(lr_scheduler, momentum=0.9).minimize(loss)

    return loss
```

__init__(*base_lr: float, steps: int, decay_rate: float, staircase: bool = False, warmup: Optional[oneflow.python.ops.optimizer.WarmupConf] = None*)

Initialize self. See help(type(self)) for accurate signature.

property learning_rate_decay_conf

class oneflow.optimizer.**PiecewiseConstantScheduler**(*boundaries: Sequence[int], values: Sequence[float], warmup: Optional[oneflow.python.ops.optimizer.WarmupConf] = None*)

This operator creates a piecewise constant learning rate scheduler.

The change in learning rate can be described as follows:

```
boundaries = [1000, 2000]
values = [0.1, 0.01, 0.001]

if current_step < 1000:
    learning_rate = 0.1
elif 1000 < current_step < 2000:
    learning_rate = 0.01
```

(continues on next page)

(continued from previous page)

```
else:
    learning_rate = 0.001
```

Parameters

- **boundaries** (*Sequence[int]*) – A list of train steps.
- **values** (*Sequence[float]*) – A list of learning rate values during the different train step boundary.
- **warmup** (*Optional[WarmupConf], optional*) – The warmup strategy. Defaults to None.

For example:

```
import oneflow as flow
import oneflow.typing as tp

@flow.global_function(type="train")
def train_job(
        images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.float),
        labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    with flow.scope.placement("gpu", "0:0"):
        logits = lenet(images, train=True)
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(
            labels, logits, name="softmax_loss"
        )

    lr_scheduler = flow.optimizer.PiecewiseConstantScheduler(boundaries=[10, 20],
                                                             values=[0.1, 0.01, 0.
↪001])
    flow.optimizer.Adam(lr_scheduler).minimize(loss)

    return loss
```

__init__(*boundaries: Sequence[int], values: Sequence[float], warmup: Optional[oneflow.python.ops.optimizer.WarmupConf] = None*)
Initialize self. See help(type(self)) for accurate signature.

property learning_rate_decay_conf

class oneflow.optimizer.**PiecewiseScalingScheduler**(*base_lr: float, boundaries: Sequence[int], scale: Union[float, Sequence[float]], warmup: Optional[oneflow.python.ops.optimizer.WarmupConf] = None*)

This operator creates a piecewise scaled decayed learning rate scheduler.

The change in learning rate can be described as follows:

```
boundaries = [1000, 2000]
scale = [0.1, 0.01]
base_lr = 0.1

if current_step < 1000:
    learning_rate = base_lr
elif 1000 < current_step < 2000:
```

(continues on next page)

(continued from previous page)

```
    learning_rate = 0.1*base_lr
else:
    learning_rate = 0.01*base_lr
```

Parameters

- **base_lr** (*float*) – The base learning rate
- **boundaries** (*Sequence[int]*) – A list of train steps.
- **scale** (*Union[float, Sequence[float]]*) – A list of learning rate scaled factors during the different train step boundary.
- **warmup** (*Optional[WarmupConf], optional*) – The warmup strategy. Defaults to None.

For example:

```
import oneflow as flow
import oneflow.typing as tp

@flow.global_function(type="train")
def train_job(
    images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.float),
    labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    with flow.scope.placement("gpu", "0:0"):
        logits = lenet(images, train=True)
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(
            labels, logits, name="softmax_loss"
        )

    lr_scheduler = flow.optimizer.PiecewiseScalingScheduler(base_lr=0.1,
                                                            boundaries=[5, 10],
                                                            scale=[0.5, 0.1])
    flow.optimizer.SGD(lr_scheduler, momentum=0).minimize(loss)

    return loss
```

__init__(*base_lr: float, boundaries: Sequence[int], scale: Union[float, Sequence[float]], warmup: Optional[oneflow.python.ops.optimizer.WarmupConf] = None*)

Initialize self. See help(type(self)) for accurate signature.

property learning_rate_decay_conf

class oneflow.optimizer.**PolynomialScheduler**(*base_lr: float, steps: int, end_learning_rate: float = 0.0001, power: float = 1.0, cycle: bool = False, warmup: Optional[oneflow.python.ops.optimizer.WarmupConf] = None*)

This operator creates a polynomial decayed learning rate scheduler.

The learning rate will be updated as follows:

If cycle is *True*, the equation is:

$$decay\_batch = decay\_batch * ceil(\frac{current\_batch}{decay\_batch})$$

$$learning\_rate = (base\_lr - end\_lr) * (1 - \frac{current\_batch}{decay\_batch})^{pow} + end\_lr$$

If cycle is *False*, the equation is:

$$decay\_batch = min(decay\_batch, current\_batch)$$

$$learning\_rate = (base\_lr - end\_lr) * (1 - \frac{current\_batch}{decay\_batch})^{pow} + end\_lr$$

Parameters

- **base_lr** (*float*) – The base learning rate
- **steps** (*int*) – The decayed steps
- **end_learning_rate** (*float, optional*) – The final learning rate. Defaults to 0.0001.
- **power** (*float, optional*) – The power of polynomial. Defaults to 1.0.
- **cycle** (*bool, optional*) – If cycle is true, the scheduler will decay the learning rate every decay steps. Defaults to False.
- **warmup** (*Optional[WarmupConf], optional*) – The warmup strategy. Defaults to None.

For example:

```
import oneflow as flow
import oneflow.typing as tp

@flow.global_function(type="train")
def train_job(
        images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.
↪float),
        labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    with flow.scope.placement("gpu", "0:0"):
        logits = lenet(images, train=True)
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(
            labels, logits, name="softmax_loss"
        )

    lr_scheduler = flow.optimizer.PolynomialScheduler(base_lr=0.001,
                                                      steps=5,
                                                      end_learning_rate=0.
↪00001,
                                                      power=2)
    flow.optimizer.Adam(lr_scheduler).minimize(loss)

    return loss
```

__init__ (*base_lr: float*, *steps: int*, *end_learning_rate: float = 0.0001*, *power: float = 1.0*, *cycle: bool = False*, *warmup: Optional[oneflow.python.ops.optimizer.WarmupConf] = None*)

Initialize self. See help(type(self)) for accurate signature.

property learning_rate_decay_conf

class oneflow.optimizer.**RMSProp**(*lr_scheduler: oneflow.python.ops.optimizer.LrScheduler*, *decay_rate: float = 0.99*, *epsilon: float = 1e-08*, *centered: bool = False*, *loss_scale_factor: Optional[float] = None*, *grad_clipping: Optional[oneflow.python.ops.optimizer.ClipGradientConf] = None*, *train_step_lbn: Optional[str] = None*, *loss_scale_policy: Optional[oneflow.python.ops.optimizer.LossScalePolicy] = None*)

The optimizer of the RMSProp algorithm.

This algorithm uses mean squared gradient to adjust the learning rate.

The equation of parameters updating is:

if centered:

$$mg_t = mg * \beta_1 + (1 - \beta_1) * grad$$
$$denom_t = S_t - mg_t * mg_t$$

else:

$$denom_t = S_t$$

$$param_{new} = param_{old} - \frac{learning\_rate}{\sqrt{denom_t + \epsilon}} \odot grad$$

Parameters

- **lr_scheduler** (*LrScheduler*) – The scheduler of learning rate.
- **decay_rate** (*float, optional*) – The decay factor (β_1). Defaults to 0.99.
- **epsilon** (*float, optional*) – A small float constant value for numerical stability (ϵ). Defaults to 1e-8.
- **centered** (*bool, optional*) – If *True*, gradients are normalized by the estimated variance of the gradient; if False, by the uncentered second moment. Setting this to *True* may help with training, but is slightly more expensive in terms of computation and memory. Defaults to *False*.
- **loss_scale_factor** (*Optional[float], optional*) – The scale factor of loss. Defaults to None.
- **grad_clipping** (*Optional[ClipGradientConf], optional*) – The gradient clipping strategy. Defaults to None.
- **train_step_lbn** (*Optional[Text], optional*) – [description]. Defaults to None.
- **loss_scale_policy** (*Optional[LossScalePolicy]*) – The policy of loss scale.

For example:

```
import oneflow as flow
import oneflow.typing as tp

@flow.global_function(type="train")
def train_job(
    images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.float),
    labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    with flow.scope.placement("gpu", "0:0"):
        logits = lenet(images, train=True)
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(
            labels, logits, name="softmax_loss"
        )
    # Set learning rate as 0.001
    lr_scheduler = flow.optimizer.PiecewiseConstantScheduler([], [0.001])
    # Set RMSProp optimizer
    flow.optimizer.RMSProp(lr_scheduler).minimize(loss)

    return loss
```

__init__(*lr_scheduler: oneflow.python.ops.optimizer.LrScheduler*, *decay_rate: float = 0.99*, *epsilon: float = 1e-08*, *centered: bool = False*, *loss_scale_factor: Optional[float] = None*, *grad_clipping: Optional[oneflow.python.ops.optimizer.ClipGradientConf] = None*, *train_step_lbn: Optional[str] = None*, *loss_scale_policy: Optional[oneflow.python.ops.optimizer.LossScalePolicy] = None*)

Initialize self. See help(type(self)) for accurate signature.

class oneflow.optimizer.**SGD**(*lr_scheduler: oneflow.python.ops.optimizer.LrScheduler*, *loss_scale_factor: Optional[float] = None*, *momentum: float = 0.9*, *grad_clipping: Optional[oneflow.python.ops.optimizer.ClipGradientConf] = None*, *train_step_lbn: Optional[str] = None*, *loss_scale_policy: Optional[oneflow.python.ops.optimizer.LossScalePolicy] = None*)

The optimizer of the stochastic gradient descent algorithm.

This algorithm takes a random sample's gradient as an approximate estimate of the overall gradient in small batch gradient descent.

When the momentum = 0, the equation of parameters updating is:

$$param_{new} = param_{old} - learning\_rate * grad$$

With momentum, the equation of parameters updating is:

$$V_t = \beta * V_{t-1} + learning\_rate * g_t$$
$$param_{new} = param_{old} - V_t$$

Parameters

- **lr_scheduler** (*LrScheduler*) – The scheduler of learning rate.
- **loss_scale_factor** (*Optional[float], optional*) – The scale factor of loss. Defaults to None.
- **momentum** (*float, optional*) – Momentum factor (β). Defaults to 0.9.
- **grad_clipping** (*Optional[ClipGradientConf], optional*) – The gradient clipping strategy. Defaults to None.
- **train_step_lbn** (*Optional[Text], optional*) – [description]. Defaults to None.

- **loss_scale_policy** (*Optional[LossScalePolicy]*) – The policy of loss scale.

For example:

```
import oneflow as flow
import oneflow.typing as tp

@flow.global_function(type="train")
def train_job(
    images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.float),
    labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    with flow.scope.placement("gpu", "0:0"):
        logits = lenet(images, train=True)
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(
            labels, logits, name="softmax_loss"
        )

    # Set Learning rate as 0.1
    lr_scheduler = flow.optimizer.PiecewiseConstantScheduler([], [0.1])
    # Set Momentum=0.9 SGD optimizer
    flow.optimizer.SGD(lr_scheduler, momentum=0.9).minimize(loss)

    return loss
```

__init__(*lr_scheduler: oneflow.python.ops.optimizer.LrScheduler, loss_scale_factor: Optional[float] = None, momentum: float = 0.9, grad_clipping: Optional[oneflow.python.ops.optimizer.ClipGradientConf] = None, train_step_lbn: Optional[str] = None, loss_scale_policy: Optional[oneflow.python.ops.optimizer.LossScalePolicy] = None*)

Initialize self. See help(type(self)) for accurate signature.

class oneflow.optimizer.**SGDW**(*lr_scheduler: oneflow.python.ops.optimizer.LrScheduler, loss_scale_factor: Optional[float] = None, momentum: float = 0.9, weight_decay: Optional[float] = None, weight_decay_includes: Union[Sequence[str], str, None] = None, weight_decay_excludes: Union[Sequence[str], str, None] = None, grad_clipping: Optional[oneflow.python.ops.optimizer.ClipGradientConf] = None, train_step_lbn: Optional[str] = None, loss_scale_policy: Optional[oneflow.python.ops.optimizer.LossScalePolicy] = None*)

The optimizer of the stochastic-gradient-descent-weight-decay algorithm.

(More details please refer to Decoupled Weight Decay Regularization).

When the momentum = 0, the equation of parameters updating is:

$$param_{new} = param_{old} - learning\_rate * (grad + \lambda * param_{old}))$$

With momentum, the equation of parameters updating is:

$$V_t = \beta * V_{t-1} - learning\_rate * g_t$$
$$param_{new} = param_{old} + V_t - learning\_rate * \lambda * param_{old}$$

Parameters

- **lr_scheduler** (*LrScheduler*) – The scheduler of learning rate.
- **loss_scale_factor** (*Optional[float], optional*) – The scale factor of loss. Defaults to None.
- **momentum** (*float, optional*) – Momentum factor (β). Defaults to 0.9.

- **weight_decay** (*Optional[float], optional*) – The weight decay factor (In the equation is λ). Defaults to None.
- **weight_decay_includes** (*Optional[Union[Sequence[Text], Text]], optional*) – The name of the model parameters that use weight decay. Defaults to None.
- **weight_decay_excludes** (*Optional[Union[Sequence[Text], Text]], optional*) – The name of the model parameters that do not use weight decay. Defaults to None.
- **grad_clipping** (*Optional[ClipGradientConf], optional*) – The gradient clipping strategy. Defaults to None.
- **train_step_lbn** (*Optional[Text], optional*) – [description]. Defaults to None.
- **loss_scale_policy** (*Optional[LossScalePolicy]*) – The policy of loss scale.

Note: Only one of *weight_decay_includes* and *weight_decay_excludes* can be set. If both are None, all the model parameters will use weight decay.

For example:

```
import oneflow as flow
import oneflow.typing as tp

@flow.global_function(type="train")
def train_job(
    images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.float),
    labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    with flow.scope.placement("gpu", "0:0"):
        logits = lenet(images, train=True)
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(
            labels, logits, name="softmax_loss"
        )

    # Set Learning rate as 0.1
    lr_scheduler = flow.optimizer.PiecewiseConstantScheduler([], [0.1])
    # Set Momentum=0.9 SGDW optimizer, weight_decay factor is 0.00005
    flow.optimizer.SGDW(lr_scheduler, momentum=0.9, weight_decay=0.00005).
↪minimize(loss)

    return loss
```

__init__(*lr_scheduler: oneflow.python.ops.optimizer.LrScheduler, loss_scale_factor: Optional[float] = None, momentum: float = 0.9, weight_decay: Optional[float] = None, weight_decay_includes: Union[Sequence[str], str, None] = None, weight_decay_excludes: Union[Sequence[str], str, None] = None, grad_clipping: Optional[oneflow.python.ops.optimizer.ClipGradientConf] = None, train_step_lbn: Optional[str] = None, loss_scale_policy: Optional[oneflow.python.ops.optimizer.LossScalePolicy] = None*)

Initialize self. See help(type(self)) for accurate signature.

class oneflow.optimizer.warmup.**constant**(*steps*, *multiplier*)

This operator use the constant warmup strategy to adjust the learning rate.

Before the steps are specified by user, the learning rate is:

$$learning\_rate = base\_learning\_rate * multiplier$$

After the steps are specified by user, the learning rate is:

$$learning\_rate = base\_learning\_rate$$

Parameters

- **steps** (*int*) – [description]
- **multiplier** (*float*) – The scale factor $multiplier$, it should be greater than 0. and less than 1.

For example:

```
import oneflow as flow
import oneflow.typing as tp

@flow.global_function(type="train")
def train_job(
    images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.float),
    labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    with flow.scope.placement("gpu", "0:0"):
        logits = lenet(images, train=True)
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(
            labels, logits, name="softmax_loss"
        )

    # Before 10 epochs, the learning rate is 0.001
    # After 10 epochs, the learning rate is 0.01
    warmup_scheduler = flow.optimizer.warmup.constant(10, 0.1)
    lr_scheduler = flow.optimizer.PiecewiseConstantScheduler([], [0.01],␣
↪warmup=warmup_scheduler)
    flow.optimizer.Adam(lr_scheduler).minimize(loss)

    return loss
```

__init__(*steps*, *multiplier*)
Initialize self. See help(type(self)) for accurate signature.

property warmup_conf

class oneflow.optimizer.warmup.**linear**(*steps*, *start_multiplier*)
This operator uses the linear warmup strategy to adjust the learning rate.

When current train step is less than warmup steps, the learning rate will be updated as:

$$current\_multiplier = start\_multiplier + (1 - start\_multiplier) * \frac{train\_step}{warmup\_step}$$

$$current\_learning\_rate = learning\_rate * current\_multiplier$$

Parameters

- **steps** (*int*) – The warmup steps.
- **start_multiplier** (*float*) – The start multiplier($start\_multiplier$). It should be greater than 0. and less than 1.

For example:

```
import oneflow as flow
import oneflow.typing as tp

@flow.global_function(type="train")
def train_job(
    images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.float),
    labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    with flow.scope.placement("gpu", "0:0"):
        logits = lenet(images, train=True)
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(
            labels, logits, name="softmax_loss"
        )

    # Before 10 epochs, the learning rate will increase from 0.001 to 0.01 in␣
↪linear.
    warmup_scheduler = flow.optimizer.warmup.linear(10, 0.1)
    lr_scheduler = flow.optimizer.PiecewiseConstantScheduler([], [0.01],␣
↪warmup=warmup_scheduler)
    flow.optimizer.Adam(lr_scheduler).minimize(loss)

    return loss
```

__init__(*steps*, *start_multiplier*)

Initialize self. See help(type(self)) for accurate signature.

property warmup_conf

class oneflow.optimizer.grad_clipping.**by_global_norm**(*clip_norm*)

This operator limits the norm of *Input* with *clip_norm*.

If the norm of *Input* is less than the *clip_norm*,

the *Output* will be the same as *Input*.

If the norm of *Input* is greater than the *clip_norm*, the *Output* will be scaled.

The equation is:

$$Output = \frac{clip\_norm * Input}{norm(Input)}$$

Parameters **clip_norm** (*float*) – The maximum norm value.

For example:

```
import oneflow as flow
import oneflow.typing as tp

@flow.global_function(type="train")
def train_job(
    images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.float),
    labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    with flow.scope.placement("gpu", "0:0"):
        logits = lenet(images, train=True)
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(
            labels, logits, name="softmax_loss"
        )
```

(continues on next page)

(continued from previous page)

```
    # Set learning rate as 0.001
    lr_scheduler = flow.optimizer.PiecewiseConstantScheduler([], [0.001])
    # Set gradient_clip
    gradient_clip = flow.optimizer.grad_clipping.by_global_norm(1.0)
    # Set AdamW optimizer with gradient clip
    flow.optimizer.AdamW(lr_scheduler,
                do_bias_correction=False, weight_decay=0.00005,
                grad_clipping=gradient_clip).minimize(loss)

    return loss
```

__init__(*clip_norm*)

Initialize self. See help(type(self)) for accurate signature.

property clip_conf

CHAPTER

SIX

ONEFLOW.LOSSES

6.1 Operators for neural networks

oneflow.losses.**add_loss**(*loss: oneflow_api.BlobDesc*) → None

Mark a *Blob* as a loss. Auto grad starts at every loss blob. It doesn't has to be a product of typical "loss" operator like softmax loss but can also be a *Blob* produced by any operator.

Parameters **loss** – A *Blob*.

CHAPTER

SEVEN

ONEFLOW.MATH

oneflow.math.**abs**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator returns the absolute value of Blob.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def abs_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.abs(x)


x = np.array([-1, 2, -3]).astype(np.float32)
out = abs_Job(x)

# out [1. 2. 3.]
```

oneflow.math.**acos**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the acos value of Blob.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def acos_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.acos(x)


x = np.array([0.5, 0.6, 0.7]).astype(np.float32)
out = acos_Job(x)

# out [1.0471976 0.9272952 0.7953989]
# We take the first value as an example
# (arccos(0.5) * pi) / 180 = 1.0471976
```

oneflow.math.**acosh**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the inverse hyperbolic cosine value of Blob.

The equation is:

$$out = log(x + (x^2 - 1)^{\frac{1}{2}})$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob, the range is [1, inf]
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def acosh_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.acosh(x)


x = np.array([2, 3, 4]).astype(np.float32)
out = acosh_Job(x)

# out [1.316958  1.7627473 2.063437 ]
```

oneflow.math.**add**(*x: Union[int, float, oneflow_api.BlobDesc], y: Union[int, float, oneflow_api.BlobDesc], name: Optional[str] = None*) → oneflow_api.BlobDesc

Compute $X + Y$ element-wise, math.add supports broadcasting. The equation is:

$$out = X + Y$$

Parameters

- **x** (*Union[int, float, oneflow_api.BlobDesc]*) – A Blob.
- **y** (*Union[int, float, oneflow_api.BlobDesc]*) – A Blob has the same type of x.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob is added by x and y, and has the same type of x.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def addJob(x: tp.Numpy.Placeholder((3, )),
        y: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.add(x, y)

x = np.array([1, 2, 3]).astype(np.float32)
y = np.array([1, 1, 1]).astype(np.float32)
out = addJob(x, y)

# out [2., 3., 4.]
```

oneflow.math.**add_n**(*inputs: Sequence[oneflow_api.BlobDesc], name: Optional[str] = None*) → oneflow_api.BlobDesc

Add all the input tensors in element-wise.

Parameters

- **inputs** (*Sequence[oneflow_api.BlobDesc]*) – A list of Blob, each Blob has the same shape and type.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The sum of the inputs, has the same shape and type as the elements of inputs.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def add_n_Job(x: tp.Numpy.Placeholder((3, )),
            y: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.add_n([x, y])

x = np.array([1, 2, 3]).astype(np.float32)
y = np.array([1, 1, 1]).astype(np.float32)
out = add_n_Job(x, y)
print(out)

# out [2., 3., 4.]
```

oneflow.math.**argmax**(*input: oneflow_api.BlobDesc*, *axis: int = -1*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

The op computes the index with the largest value of a Blob at specified axis.

Parameters

- **input** (*oneflow_api.BlobDesc*) – Input Blob
- **axis** (*int, optional*) – dimension to be calculated. Defaults to the last dim (-1)
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob(dtype=int32) contains the index with the largest value of *input*

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def argmax_Job(x: tp.Numpy.Placeholder((2, 5))
)->tp.Numpy:
    return flow.math.argmax(x)

x = np.array([[1, 3, 8, 7, 2],
              [1, 9, 4, 3, 2]], dtype=np.float32)

out = argmax_Job(x)

# out [2 1]
```

oneflow.math.**asin**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the arcsin value of Blob.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def asin_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.asin(x)


x = np.array([0.5, 0.6, 0.7]).astype(np.float32)
out = asin_Job(x)
```

(continues on next page)

(continued from previous page)

```
# out [0.5235988  0.64350116 0.7753975 ]
# We take the first value as an example
# (arcsin(0.5) * pi) / 180 = 0.5235988
```

oneflow.math.**asinh**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the inverse hyperbolic sine value of Blob.

The equation is:

$$out = log(x + (x^2 + 1)^{\frac{1}{2}})$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def asinh_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.asinh(x)


x = np.array([2, 3, 4]).astype(np.float32)
out = asinh_Job(x)

# out [1.4436355 1.8184464 2.0947125]
```

oneflow.math.**atan**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the arctan value of Blob.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def atan_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
```

(continues on next page)

(continued from previous page)

```
    return flow.math.atan(x)


x = np.array([0.5, 0.6, 0.7]).astype(np.float32)
out = atan_Job(x)

# out [0.4636476  0.5404195  0.61072594]
# We take the first value as an example
# (arctan(0.5) * pi) / 180 = 0.4636476
```

oneflow.math.**atan2**(*x: oneflow_api.BlobDesc*, *y: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the values of $arctan(\frac{x}{y})$.

The equation is:

$$out = arctan(\frac{x}{y})$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **y** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def atan2Job(x: tp.Numpy.Placeholder((3,),), y: tp.Numpy.Placeholder((3, ))
)-> tp.Numpy:
    return flow.math.atan2(x, y)

x = np.array([1, 2, 3]).astype(np.float32)
y = np.array([4, 4, 4]).astype(np.float32)
out = atan2Job(x, y)


# out [0.24497867 0.4636476  0.6435011 ]
# We take the first value as an example
# (arctan(1/4) * pi) / 180 = 0.24497867
```

oneflow.math.**atanh**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the inverse hyperbolic tangent value of Blob.

The equation is:

$$out = \frac{1}{2} * log(\frac{1+x}{1-x})$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def atanh_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.atanh(x)


x = np.array([0.5, 0.6, 0.7]).astype(np.float32)
out = atanh_Job(x)

# out [0.54930615 0.6931472  0.8673005 ]
```

oneflow.math.**broadcast_to_compatible_with**(*x: oneflow_api.BlobDesc, compatible: Sequence[oneflow_api.BlobDesc], name: Optional[str] = None*) → oneflow_api.BlobDesc

Returns a 'Blob' with the shape can be broadcasted by other shapes

Parameters

- **x** (*oneflow_api.BlobDesc*) – a 'Blob'
- **compatible** (*Sequence[oneflow_api.BlobDesc]*) – Sequence of different shape
- **name** (*Optional[str], optional*) – This operator's name. Defaults to None.

Returns A 'Blob' with the biggest shape

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def broadcast_to_compatible_with_Job(x: tp.Numpy.Placeholder((4, 1, 1))
)->tp.Numpy:
    blob_a = flow.constant(value=1, dtype=flow.float32, shape=(1, 2, 1))
    blob_b = flow.constant(value=1, dtype=flow.float32, shape=(1, 1, 3))

    return flow.math.broadcast_to_compatible_with(x, [blob_a, blob_b])

x = np.ones(shape=(4, 1, 1), dtype=np.float32)

out = broadcast_to_compatible_with_Job(x)
```

(continues on next page)

(continued from previous page)

```
# out.shape (4, 2, 3)
```

oneflow.math.**ceil**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the ceiling value of Blob.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def ceil_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.ceil(x)


x = np.array([1.3, 1.5, 2.7]).astype(np.float32)
out = ceil_Job(x)

# out [2. 2. 3.]
```

oneflow.math.**clip_by_value**(*values: oneflow_api.BlobDesc*, *min_value: Union[int, float, None] = None*, *max_value: Union[int, float, None] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This op clips Blob values to a specified min value and max value.

The equation is:

$$out = MIN(MAX(x, min), max)$$

Parameters

- **values** (*oneflow_api.BlobDesc*) – Input Blob
- **min_value** (*Optional[Union[int, float]], optional*) – The minimum value to clip by. Defaults to None.
- **max_value** (*Optional[Union[int, float]], optional*) – The maximum value to clip by. Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Raises **ValueError** – min_value and max_value *cannot be None at the same time*

Returns A clipped Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def clip_by_value_Job(x: tp.Numpy.Placeholder((4, ))
)->tp.Numpy:
    return flow.math.clip_by_value(x, min_value=-1, max_value=5)

x = np.array([-2, 1, 4, 7], dtype=np.float32)

out = clip_by_value_Job(x)

# out [-1. 1. 4. 5.]
```

oneflow.math.**cos**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the cosine value of Blob.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def cos_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.cos(x)


x = np.array([1/3*np.pi, 0.25*np.pi, 1.25*np.pi]).astype(np.float32)
out = cos_Job(x)

# out [ 0.49999997  0.70710677 -0.7071068 ]
```

oneflow.math.**cosh**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes hyperbolic cosine value of Blob.

The equation is:

$$out = \frac{e^x + e^{-x}}{2}$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def cosh_Job(x: tp.Numpy.Placeholder((3,))
             ) -> tp.Numpy:
    return flow.math.cosh(x)


x = np.array([1, 2, 3]).astype(np.float32)
out = cosh_Job(x)

# out [ 1.5430806  3.7621958 10.067662 ]
```

oneflow.math.**divide**(*x: Union[int, float, oneflow_api.BlobDesc], y: Union[int, float, oneflow_api.BlobDesc], name: Optional[str] = None*) → oneflow_api.BlobDesc

Computes the division of x by y.

The equation is:

$$out = \frac{X}{Y}$$

Parameters

- **x** (*Union[int, float, oneflow_api.BlobDesc]*) – A Blob.
- **y** (*Union[int, float, oneflow_api.BlobDesc]*) – A Blob.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob with same shape as input x.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def divideJob(x: tp.Numpy.Placeholder((3, )),
              y: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.divide(x, y)

x = np.array([25, 16, 9]).astype(np.float32)
y = np.array([10, 4, 2]).astype(np.float32)
out = divideJob(x, y)

# out [2.5, 4., 4.5]
```

oneflow.math.**equal**(*x: oneflow_api.BlobDesc, y: oneflow_api.BlobDesc, name: Optional[str] = None*) → oneflow_api.BlobDesc

Returns the truth value of $x == y$ element-wise.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **y** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob with int8 type.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def equal_Job(x: tp.Numpy.Placeholder((3, )),
              y: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.equal(x, y)

x = np.array([1, 2, 3]).astype(np.float32)
y = np.array([1, 2, 1]).astype(np.float32)
out = equal_Job(x, y)

# out [1 1 0]
```

oneflow.math.**erf**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the Gauss error value of Blob.

The equation is:

$$out = \frac{2}{\sqrt{\pi}} * \int_0^x e^{-z^2} \mathrm{d}z$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def erf_Job(x: tp.Numpy.Placeholder((3,))
            ) -> tp.Numpy:
    return flow.math.erf(x)


x = np.array([1, 2, 3]).astype(np.float32)
out = erf_Job(x)

# out [0.8427008 0.9953223 0.9999779]
```

oneflow.math.**erfc**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the $1 - erf(x)$, for more details of *erf* function please refer to *math.erf*.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def erfc_Job(x: tp.Numpy.Placeholder((3,))
             ) -> tp.Numpy:
    return flow.math.erfc(x)


x = np.array([1, 2, 3]).astype(np.float32)
out = erfc_Job(x)

# out [1.5729921e-01 4.6777353e-03 2.2090495e-05]
```

oneflow.math.**exp**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the exponential of Blob.

The equation is:

$$out = e^x$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def exp_Job(x: tp.Numpy.Placeholder((3,))
            ) -> tp.Numpy:
    return flow.math.exp(x)


x = np.array([1, 2, 3]).astype(np.float32)
out = exp_Job(x)
```

(continues on next page)

(continued from previous page)

```
# out [ 2.7182817  7.389056  20.085537 ]
```

oneflow.math.**expm1**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes $y = e^x - 1$.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def expm1_Job(x: tp.Numpy.Placeholder((3,))
              ) -> tp.Numpy:
    return flow.math.expm1(x)


x = np.array([1, 2, 3]).astype(np.float32)
out = expm1_Job(x)

# out [ 1.7182819  6.389056  19.085537 ]
```

oneflow.math.**floor**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the largest integer not greater than input Blob.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def floor_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.floor(x)


x = np.array([1.3, 1.5, 2.7]).astype(np.float32)
```

(continues on next page)

(continued from previous page)

```
out = floor_Job(x)

# out [1. 1. 2.]
```

oneflow.math.**floordiv**(*x: oneflow_api.BlobDesc*, *y: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the result of x/y, rounding toward the most negative integer value

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **y** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def floor_div_Job(x: tp.Numpy.Placeholder((3,)),
                  y: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.floordiv(x, y)


x = np.array([4, 3, 5]).astype(np.float32)
y = np.array([3, 2, 2]).astype(np.float32)
out = floor_div_Job(x, y)

# out [1. 1. 2.]
```

oneflow.math.**fused_scale_tril**(*x: oneflow_api.BlobDesc*, *diagonal: int = 0*, *fill_value: Union[int, float] = 0*, *scale: Union[int, float] = 1*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.math.**gelu**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Gelu activation operator.

The equation is:

$$out = 0.5 * x * (1 + tanh(\sqrt{\frac{2}{\pi}} * (x + 0.044715x^3)))$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – Input Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def geluJob(x: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.gelu(x)

x = np.array([-0.5, 0, 0.5]).astype(np.float32)
out = geluJob(x)

# out [-0.15426877, 0., 0.34573123]
```

oneflow.math.**gelu_grad**(*x: oneflow_api.BlobDesc*, *dy: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.math.**greater**(*x: oneflow_api.BlobDesc*, *y: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Returns the truth value of $x > y$ element-wise.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **y** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob with int8 type.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def greater_Job(x: tp.Numpy.Placeholder((3, )),
                y: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.greater(x, y)

x = np.array([1, 1, 4]).astype(np.float32)
y = np.array([1, 2, 3]).astype(np.float32)
out = greater_Job(x, y)

# out [0 0 1]
```

oneflow.math.**greater_equal**(*x: oneflow_api.BlobDesc*, *y: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Returns the truth value of $x >= y$ element-wise.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **y** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob with int8 type.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def greater_equal_Job(x: tp.Numpy.Placeholder((3, )),
                      y: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.greater_equal(x, y)

x = np.array([1, 1, 4]).astype(np.float32)
y = np.array([1, 2, 3]).astype(np.float32)
out = greater_equal_Job(x, y)

# out [1 0 1]
```

oneflow.math.**in_top_k**(*targets: oneflow_api.BlobDesc, predictions: oneflow_api.BlobDesc, k: Optional[int], name: Optional[str] = None*) → oneflow_api.BlobDesc

Says whether the targets are in the top K predictions.

Parameters

- **targets** (*oneflow_api.BlobDesc*) – A Blob of type int32 or int64.
- **predictions** (*oneflow_api.BlobDesc*) – A Blob of type float32.
- **k** (*Optional[int], optional*) – Number of top elements to look at for computing precision.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob of type bool. Computed Precision at k as a bool Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def intopk_Job(
    targets: tp.Numpy.Placeholder((2,), dtype=flow.int32),
    predictions: tp.Numpy.Placeholder((2, 4), dtype=flow.float32),
) -> tp.Numpy:
    return flow.math.in_top_k(targets, predictions, 1)

targets = np.array([3, 1], dtype=np.int32)
predictions = np.array([[0.0, 1.0, 2.0, 3.0], [3.0, 2.0, 1.0, 0.0],], dtype=np.
↪float32)
out = intopk_Job(targets, predictions)

# out [1 0]
```

oneflow.math.**l2_normalize**(*input: oneflow_api.BlobDesc*, *axis: Optional[int] = None*, *epsilon: float = 1e-12*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Use L2 norm to normalizes along dimension *axis*

The equation is:

$$out = \frac{x}{\sqrt{\Sigma x^2 + \epsilon}}$$

Parameters

- **input** (*oneflow_api.BlobDesc*) – Input Blob
- **axis** (*Optional[int], optional*) – The axis on which to apply L2 normalization. Defaults to None.
- **epsilon** (*float, optional*) – The epsilon value is used to avoid division by zero. Defaults to 1e-12.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The normalized Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def l2_normalize_Job(x: tp.Numpy.Placeholder((4, ))
)->tp.Numpy:
    return flow.math.l2_normalize(x, axis=0)

x = np.array([1, 2, 3, 4], dtype=np.float32)

out = l2_normalize_Job(x)

# out [0.18257418 0.36514837 0.5477226  0.73029673]
```

oneflow.math.**less**(*x: oneflow_api.BlobDesc*, *y: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Returns the truth value of $x < y$ element-wise.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **y** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob with int8 type.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp
```

(continues on next page)

(continued from previous page)

```
@flow.global_function()
def less_Job(x: tp.Numpy.Placeholder((3, )),
             y: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.less(x, y)

x = np.array([1, 2, 3]).astype(np.float32)
y = np.array([1, 2, 4]).astype(np.float32)
out = less_Job(x, y)

# out [0 0 1]
```

oneflow.math.**less_equal**(*x: oneflow_api.BlobDesc*, *y: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Returns the truth value of $x <= y$ element-wise.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **y** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob with int8 type.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def less_equal_Job(x: tp.Numpy.Placeholder((3, )),
                   y: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.less_equal(x, y)

x = np.array([1, 2, 3]).astype(np.float32)
y = np.array([1, 1, 4]).astype(np.float32)
out = less_equal_Job(x, y)

# out [1 0 1]
```

oneflow.math.**lgamma**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the $Gamma(x)$ value.

The equation is:

$$out = \int_0^\infty t^{x-1} * e^{-t} \mathrm{d}t$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def lgamma_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.lgamma(x)


x = np.array([1.3, 1.5, 2.7]).astype(np.float32)
out = lgamma_Job(x)

# out [-0.1081748  -0.12078223  0.4348206 ]
```

oneflow.math.**log**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the log value of input Blob.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def log_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.log(x)


x = np.array([1.3, 1.5, 2.7]).astype(np.float32)
out = log_Job(x)

# out [0.26236424 0.40546513 0.9932518 ]
```

oneflow.math.**log1p**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the $log(x) + 1$ value of input Blob.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def log1p_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.log1p(x)


x = np.array([1.3, 1.5, 2.7]).astype(np.float32)
out = log1p_Job(x)

# out [0.8329091  0.91629076 1.3083328 ]
```

oneflow.math.**log_sigmoid**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the log sigmoid value of input Blob.

The equation is:

$$out = log(\frac{1}{1 + e^{-x}})$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def log_sigmoid_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.log_sigmoid(x)


x = np.array([1.3, 1.5, 2.7]).astype(np.float32)
out = log_sigmoid_Job(x)

# out [-0.24100842 -0.20141333 -0.0650436 ]
```

oneflow.math.**logical_and**(*x: oneflow_api.BlobDesc*, *y: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Logical AND function.

Each element is calculated by:

$$out = X \wedge Y$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **y** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob with int8 type.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def logical_and_Job(x: tp.Numpy.Placeholder((3, )),
                    y: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.logical_and(x, y)

x = np.array([1, 0, 1]).astype(np.float32)
y = np.array([0, 0, 1]).astype(np.float32)
out = logical_and_Job(x, y)

# out [0 0 1]
```

oneflow.math.**maximum**(*x: oneflow_api.BlobDesc*, *y: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Returns the max of x and y element-wise, this op supports broadcasting.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **y** (*oneflow_api.BlobDesc*) – A Blob. Must have the same type of x
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob, has the same type of x.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def maximum_Job(x: tp.Numpy.Placeholder((3, )),
                y: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.maximum(x, y)

x = np.array([2, 3, 4]).astype(np.float32)
y = np.array([4, 2, 1]).astype(np.float32)
out = maximum_Job(x, y)

# out [4. 3. 4.]
```

oneflow.math.**minimum**(*x: oneflow_api.BlobDesc*, *y: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Returns the min of x and y element-wise, this op supports broadcasting.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **y** (*oneflow_api.BlobDesc*) – A Blob. Must have the same type of x
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob, has the same type of x.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def minimum_Job(x: tp.Numpy.Placeholder((3, )),
                y: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.minimum(x, y)

x = np.array([2, 3, 4]).astype(np.float32)
y = np.array([4, 2, 1]).astype(np.float32)
out = minimum_Job(x, y)

# out [2. 2. 1.]
```

oneflow.math.**mod**(*x: Union[int, float, oneflow_api.BlobDesc], y: Union[int, float, oneflow_api.BlobDesc], name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator mods two Blobs.

The equation is:

$$out = X \bmod Y$$

Parameters

- **x** (*Union[int, float, oneflow_api.BlobDesc]*) – A Blob
- **y** (*Union[int, float, oneflow_api.BlobDesc]*) – A Blob has the same type of x
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Raises

- **NotImplementedError** – x must be an int or a float
- **NotImplementedError** – y must be an int or a float

Returns A Blob with same type as input x.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def modJob(x: tp.Numpy.Placeholder((3, )),
        y: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.mod(x, y)

x = np.array([16, 9, 5]).astype(np.float32)
y = np.array([6, 4, 3]).astype(np.float32)
out = modJob(x, y)

# out [4., 1., 2.]
```

oneflow.math.**multiply**(*x: Union[int, float, oneflow_api.BlobDesc], y: Union[int, float, oneflow_api.BlobDesc], name: Optional[str] = None*) → oneflow_api.BlobDesc

Compute $x \times y$ element-wise.

The equation is:

$$out = X \times Y$$

Parameters

- **x** (*Union[int, float, oneflow_api.BlobDesc]*) – A Blob.
- **y** (*Union[int, float, oneflow_api.BlobDesc]*) – A Blob has the same type of x.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob after multiplying, has the same type as x.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def multiplyJob(x: tp.Numpy.Placeholder((3, )),
                y: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.multiply(x, y)

x = np.array([1, 2, 3]).astype(np.float32)
y = np.array([2, 3, 3]).astype(np.float32)
out = multiplyJob(x, y)

# out [2., 6., 9.]
```

oneflow.math.**negative**(*x: oneflow_api.BlobDesc, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the negative value of Blob.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def negative_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.negative(x)


x = np.array([1.3, 1.5, 2.7]).astype(np.float32)
out = negative_Job(x)

# out [-1.3 -1.5 -2.7]
```

oneflow.math.**not_equal**(*x: oneflow_api.BlobDesc*, *y: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Returns the truth value of $x! = y$ element-wise.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **y** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob with int8 type.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def not_equal_Job(x: tp.Numpy.Placeholder((3, )),
                  y: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.not_equal(x, y)

x = np.array([1, 2, 3]).astype(np.float32)
y = np.array([1, 2, 1]).astype(np.float32)
out = not_equal_Job(x, y)

# out [0 0 1]
```

oneflow.math.**polyval**(*coeffs: Union[List, Tuple], x: oneflow_api.BlobDesc, name: Optional[str] = None*) → oneflow_api.BlobDesc

Computes the elementwise value of a polynomial.

Parameters

- **coeffs** (*Union[List, Tuple]*) – The coefficients of the polynomial.
- **x** (*oneflow_api.BlobDesc*) – A Blob.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob, has the same data type of x.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def polyval_Job(
    x: tp.Numpy.Placeholder((3,), dtype=flow.float32)
) -> tp.Numpy:
    coeffs = [1.0, 3.0, -2.0]
    return flow.math.polyval(coeffs, x)

x = np.array([1.0, 2.0, 3.0]).astype(np.float32)
out = polyval_Job(x)

# output [ 2.  8. 16.]
```

oneflow.math.**pow**(*x: oneflow_api.BlobDesc, y: Union[oneflow_api.BlobDesc, float], name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the Pow result.

The equation is:

$$out = x^y$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **y** (*Union[oneflow_api.BlobDesc, float]*) – A Blob or float value, the exponential factor of Pow
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

Example 1:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def powJob(x: tp.Numpy.Placeholder((3,), ), y: tp.Numpy.Placeholder((3,))
        ) -> tp.Numpy:
```

(continues on next page)

(continued from previous page)

```
    return flow.math.pow(x, y)


x = np.array([2, 3, 4]).astype(np.float32)
y = np.array([2, 3, 4]).astype(np.float32)
out = powJob(x, y)

# out [  4.  27. 256.]
```

Example 2:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def scalar_pow_job(x: tp.Numpy.Placeholder(shape=(3, )))->tp.Numpy:
    with flow.scope.placement("cpu", "0:0"):
        out = flow.math.pow(x, 2.0)
    return out


x = np.array([1, 2, 3]).astype(np.float32)
out = scalar_pow_job(x)

# out [1. 4. 9.]
```

oneflow.math.**reciprocal**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the reciprocal of x.

The equation is:

$$out = \frac{1}{x}$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def reciprocal_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.reciprocal(x)
```

(continues on next page)

(continued from previous page)

```
x = np.array([1, 2, 4]).astype(np.float32)
out = reciprocal_Job(x)

# out [1.   0.5  0.25]
```

oneflow.math.**reciprocal_no_nan**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the safe reciprocal of x. If x is zero, the reciprocal will be also set to zero.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def reciprocal_no_nan_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.reciprocal_no_nan(x)


x = np.array([0, 2, 4]).astype(np.float32)
out = reciprocal_no_nan_Job(x)

# out [0.   0.5  0.25]
```

oneflow.math.**reduce_all**(*x: oneflow_api.BlobDesc*, *axis: Union[int, Sequence[int], None] = None*, *keepdims: bool = False*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the *logical and* of input Blob along the specified axis

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **axis** (*Optional[Union[int, Sequence[int]]], optional*) – The dimension along which the logical and value is computed. Defaults to None.
- **keepdims** (*bool, optional*) – Whether to keep the reduced dimension in the output Blob. Defaults to False.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result of logical and value on the specified axis of input Blob

Return type oneflow_api.BlobDesc

Note: The input Blob dtype is int8

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def reduce_all_Job(x: tp.Numpy.Placeholder((3, 3), dtype=flow.int8)
) -> tp.Numpy:
    return flow.math.reduce_all(x, axis=1, keepdims=True)


x = np.array([[1, 0, 0], [0, 0, 0], [1, 1, 1]]).astype(np.int8)
out = reduce_all_Job(x)

# out [[0]
#      [0]
#      [1]]
```

oneflow.math.**reduce_any**(*x: oneflow_api.BlobDesc, axis: Union[int, Sequence[int], None] = None, keepdims: bool = False, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the *logical or* of input Blob along the specified axis

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **axis** (*Optional[Union[int, Sequence[int]]], optional*) – The dimension along which the logical and value is computed. Defaults to None.
- **keepdims** (*bool, optional*) – Whether to keep the reduced dimension in the output Blob. Defaults to False.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result of logical or on the specified axis of input Blob

Return type oneflow_api.BlobDesc

Note: The input Blob dtype is int8

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def reduce_any_Job(x: tp.Numpy.Placeholder((3, 3), dtype=flow.int8)
) -> tp.Numpy:
    return flow.math.reduce_any(x, axis=1, keepdims=True)


x = np.array([[1, 0, 0], [0, 0, 0], [1, 0, 1]]).astype(np.int8)
out = reduce_any_Job(x)
```

(continues on next page)

(continued from previous page)

```
# out [[1]
#      [0]
#      [1]]
```

oneflow.math.**reduce_euclidean_norm**(*input_tensor: oneflow_api.BlobDesc, axis: Union[int, Sequence[int], None] = None, keepdims: bool = False, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the Euclidean norm of input Blob along the specified axis

The equation is:

$$out = \sqrt{\sum_{t=0}^{n} x_t^2}$$

Parameters

- **input_tensor** (*oneflow_api.BlobDesc*) – A Blob
- **axis** (*Optional[Union[int, Sequence[int]]], optional*) – The dimension along which the Euclidean norm is computed. Defaults to None.
- **keepdims** (*bool, optional*) – Whether to keep the reduced dimension in the output Blob. Defaults to False.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result of Euclidean norm on the specified axis of input Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def reduce_euclidean_norm_Job(x: tp.Numpy.Placeholder((3, 2))
) -> tp.Numpy:
    return flow.math.reduce_euclidean_norm(x, axis=1, keepdims=True)


x = np.array([[3, 4], [5, 12], [8, 15]]).astype(np.float32)
out = reduce_euclidean_norm_Job(x)

# out [[ 5.]
#      [13.]
#      [17.]]
```

oneflow.math.**reduce_logsumexp**(*input_tensor: oneflow_api.BlobDesc, axis: Union[int, Sequence[int], None] = None, keepdims: bool = False, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the log of exponential sum of input Blob along the specified axis

The equation is:

$$out = log(\sum_{t=0}^{t=n} e^{x_t})$$

Parameters

- **input_tensor** (*oneflow_api.BlobDesc*) – A Blob
- **axis** (*Optional[Union[int, Sequence[int]]], optional*) – The dimension along which the log of exponential sum is computed. Defaults to None.
- **keepdims** (*bool, optional*) – Whether to keep the reduced dimension in the output Blob. Defaults to False.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result of log of exponential sum on the specified axis of input Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def reduce_logsumexp_Job(x: tp.Numpy.Placeholder((3, 2))
) -> tp.Numpy:
    return flow.math.reduce_logsumexp(x, axis=1, keepdims=True)


x = np.array([[0, 0], [1, 1], [2, 2]]).astype(np.float32)
out = reduce_logsumexp_Job(x)

# out [[0.6931472]
#      [1.6931472]
#      [2.6931472]]
```

oneflow.math.**reduce_max**(*x: oneflow_api.BlobDesc, axis: Union[int, Sequence[int], None] = None, keepdims: bool = False, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the maximum value of input Blob along the specified axis

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **axis** (*Optional[Union[int, Sequence[int]]], optional*) – The dimension along which the maximum value is computed. Defaults to None.
- **keepdims** (*bool, optional*) – Whether to keep the reduced dimension in the output Blob. Defaults to False.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result of maximum value on the specified axis of input Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp
```

(continues on next page)

(continued from previous page)

```
@flow.global_function()
def reduce_max_Job(x: tp.Numpy.Placeholder((3, 3))
) -> tp.Numpy:
    return flow.math.reduce_max(x, axis=1, keepdims=True)


x = np.array([[2, 1, 4], [5, 3, 7], [7, 4, 9]]).astype(np.float32)
out = reduce_max_Job(x)

# out [[4.]
#      [7.]
#      [9.]]
```

oneflow.math.**reduce_mean**(*input_blob: oneflow_api.BlobDesc*, *axis: Union[collections.abc.Sized, int, None] = None*, *keepdims: bool = False*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the mean of input Blob along the specified axis

Parameters

- **input_blob** (*oneflow_api.BlobDesc*) – A Blob
- **axis** (*Optional[Union[collections.Sized, int]], optional*) – The dimension along which the mean value is computed. Defaults to None.
- **keepdims** (*bool, optional*) – Whether to keep the reduced dimension in the output Blob. Defaults to False.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result of average on the specified axis of input Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def reduce_mean_Job(x: tp.Numpy.Placeholder((3, 3))
) -> tp.Numpy:
    return flow.math.reduce_mean(x, axis=1, keepdims=True)


x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]).astype(np.float32)
out = reduce_mean_Job(x)

# out [[2.]
#      [5.]
#      [8.]]
```

oneflow.math.**reduce_min**(*x: oneflow_api.BlobDesc*, *axis: Union[int, Sequence[int], None] = None*, *keepdims: bool = False*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the minimum value of input Blob along the specified axis

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **axis** (*Optional[Union[int, Sequence[int]]], optional*) – The dimension along which the minimum value is computed. Defaults to None.
- **keepdims** (*bool, optional*) – Whether to keep the reduced dimension in the output Blob. Defaults to False.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result of minimum value on the specified axis of input Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def reduce_min_Job(x: tp.Numpy.Placeholder((3, 3))
) -> tp.Numpy:
    return flow.math.reduce_min(x, axis=1, keepdims=True)


x = np.array([[2, 1, 3], [5, 3, 6], [7, 4, 9]]).astype(np.float32)
out = reduce_min_Job(x)

# out [[1.]
#      [3.]
#      [4.]]
```

oneflow.math.**reduce_prod**(*x: oneflow_api.BlobDesc, axis: Union[int, Sequence[int], None] = None, keepdims: bool = False, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the product of input Blob along the specified axis

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **axis** (*Optional[Union[int, Sequence[int]]], optional*) – The dimension along which the product is computed. Defaults to None.
- **keepdims** (*bool, optional*) – Whether to keep the reduced dimension in the output Blob. Defaults to False.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result of product value on the specified axis of input Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp
```

(continues on next page)

(continued from previous page)

```
@flow.global_function()
def reduce_product_Job(x: tp.Numpy.Placeholder((3, 3))
) -> tp.Numpy:
    return flow.math.reduce_prod(x, axis=1, keepdims=True)


x = np.array([[1, 2, 3], [3, 4, 5], [6, 3, 2]]).astype(np.float32)
out = reduce_product_Job(x)

# out [[ 6.]
#      [60.]
#      [36.]]
```

oneflow.math.**reduce_std**(*input_tensor: oneflow_api.BlobDesc, axis: Union[int, Sequence[int], None] = None, keepdims: bool = False, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the standard deviation of input Blob along the specified axis

The equation is:

$$out = \sqrt{\frac{1}{n} * \sum_{i=1}^{n}(x_i - mean)^2}$$

Parameters

- **input_tensor** (*oneflow_api.BlobDesc*) – A Blob
- **axis** (*Optional[Union[int, Sequence[int]]], optional*) – The dimension along which the standard deviation is computed. Defaults to None.
- **keepdims** (*bool, optional*) – Whether to keep the reduced dimension in the output Blob. Defaults to False.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result of standard deviation on the specified axis of input Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def reduce_std_Job(x: tp.Numpy.Placeholder((3, 3))
) -> tp.Numpy:
    return flow.math.reduce_std(x, axis=1, keepdims=True)


x = np.array([[0, 5, 10], [5, 5, 5], [12, 3, 0]]).astype(np.float32)
out = reduce_std_Job(x)

# out [[4.0824833]
#      [0.       ]
#      [5.0990195]]
```

oneflow.math.**reduce_sum**(*input_tensor: oneflow_api.BlobDesc, axis: Union[int, Sequence[int], None] = None, keepdims: bool = False, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the sum of elements across dimensions of a tensor

Parameters

- **input_tensor** (*oneflow_api.BlobDesc*) – A Blob
- **axis** (*Optional[Union[int, Sequence[int]]], optional*) – The dimension along which the sum value is computed. Defaults to None.
- **keepdims** (*bool, optional*) – Whether to keep the reduced dimension in the output Blob. Defaults to False.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result of sum on the specified axis of input Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def reduce_sum_Job(x: tp.Numpy.Placeholder((3, 3))
) -> tp.Numpy:
    return flow.math.reduce_sum(x, axis=1, keepdims=True)


x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]).astype(np.float32)
out = reduce_sum_Job(x)

# out [[ 6.]
#      [15.]
#      [24.]]
```

oneflow.math.**reduce_variance**(*input_tensor: oneflow_api.BlobDesc, axis: Union[int, Sequence[int], None] = None, keepdims: bool = False, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the variance of input Blob along the specified axis

The equation is:

$$out = \frac{1}{n} * \sum_{i=1}^{n}(x_i - mean)^2$$

Parameters

- **input_tensor** (*oneflow_api.BlobDesc*) – A Blob
- **axis** (*Optional[Union[int, Sequence[int]]], optional*) – The dimension along which the variance is computed. Defaults to None.
- **keepdims** (*bool, optional*) – Whether to keep the reduced dimension in the output Blob. Defaults to False.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result of variance on the specified axis of input Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def reduce_variance_Job(x: tp.Numpy.Placeholder((3, 3))
) -> tp.Numpy:
    return flow.math.reduce_variance(x, axis=1, keepdims=True)


x = np.array([[0, 5, 10], [5, 5, 5], [12, 3, 0]]).astype(np.float32)
out = reduce_variance_Job(x)

# out [[16.666668]
#      [ 0.      ]
#      [26.      ]]
```

oneflow.math.**reduced_shape_elem_cnt**(*input_blob: oneflow_api.BlobDesc*, *axis: Optional[Sequence[int]] = None*, *dtype: Optional[oneflow.python.framework.dtype.dtype] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Computes the product of input_blob's dimensions along the parameter *axis*. By default, all the dimensions will be computed.

Parameters

- **input_blob** (*oneflow_api.BlobDesc*) – Input Blob
- **axis** (*Optional[Sequence[int]], optional*) – The dimensions along which the op is performed. Defaults to None.
- **dtype** (*Optional[dtype_util.dtype], optional*) – The data type. Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob

Return type oneflow_api.BlobDesc

For example:

```
# Example 1:
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def elem_cnt_Job(x: tp.Numpy.Placeholder((3, 4, 5))
)->tp.Numpy:
    return flow.math.reduced_shape_elem_cnt(x, axis=[0, 1])

x = np.ones(shape=(3, 4, 5), dtype=np.float32)
out = elem_cnt_Job(x) # 3 x 4 = 12
```

(continues on next page)

(continued from previous page)

```
# out [12]

# Example 2:
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def elem_cnt_Job(x: tp.Numpy.Placeholder((3, 4, 5))
)->tp.Numpy:
    return flow.math.reduced_shape_elem_cnt(x)

x = np.ones(shape=(3, 4, 5), dtype=np.float32)
out = elem_cnt_Job(x) # 3 x 4 x 5 = 60

# out [60]
```

oneflow.math.**relu**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Relu activation

The equation is:

$$out = max(X, 0)$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – Input Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns An activated Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def reluJob(x: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.relu(x)

x = np.array([-1, 0, 5]).astype(np.float32)
out = reluJob(x)

# out [0., 0., 5.]
```

oneflow.math.**rint**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the closest integer to Blob.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def rint_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.rint(x)


x = np.array([1.49999, 1.500001, 2.7]).astype(np.float32)
out = rint_Job(x)

# out [1. 2. 3.]
```

oneflow.math.**round**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator rounds the value of Blob to the nearest integer.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def round_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.round(x)


x = np.array([1.49999, 1.500001, 2.7]).astype(np.float32)
out = round_Job(x)

# out [1. 2. 3.]
```

oneflow.math.**rsqrt**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the reciprocal of square root value of Blob.

The equation is:

$$out = \frac{1}{\sqrt{x}}$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob

- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def rsqrt_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.rsqrt(x)


x = np.array([4, 16, 25]).astype(np.float32)
out = rsqrt_Job(x)

# out [0.5  0.25 0.2 ]
```

oneflow.math.**sigmoid**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Sigmoid activation

The equation is:

$$out = \frac{1}{1 + e^{-x}}$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – Input Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns An activated Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def sigmoidJob(x: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.sigmoid(x)

x = np.array([-1, 0, 1]).astype(np.float32)
out = sigmoidJob(x)

# out [0.26894143, 0.5, 0.7310586]
```

oneflow.math.**sigmoid_grad**(*y: oneflow_api.BlobDesc*, *dy: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.math.**sigmoid_v2**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the sigmoid value of Blob.

The equation is:

$$out = \frac{1}{1 + e^{-x}}$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def sigmoidv2_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.sigmoid_v2(x)

x = np.array([-0.5, 0, 0.5]).astype(np.float32)
out = sigmoidv2_Job(x)

# out [0.37754068 0.5        0.62245935]
```

oneflow.math.**sign**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator returns the sign of Blob.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def sign_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.sign(x)


x = np.array([-2, 0, 2]).astype(np.float32)
```

(continues on next page)

(continued from previous page)

```
out = sign_Job(x)

# out [-1.  0.  1.]
```

oneflow.math.**sin**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the sin value of Blob.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def sin_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.sin(x)


x = np.array([-1/6*np.pi, 0, 1/6*np.pi]).astype(np.float32)
out = sin_Job(x)

# out [-0.5  0.   0.5]
```

oneflow.math.**sinh**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the hyperbolic sine value of Blob.

The equation is:

$$out = \frac{e^x - e^{-x}}{2}$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
```

(continues on next page)

(continued from previous page)

```
def sinh_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.sinh(x)


x = np.array([-1, 0, 1]).astype(np.float32)
out = sinh_Job(x)

# out [-1.1752012  0.         1.1752012]
```

oneflow.math.**softplus**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the softplus value of Blob.

The equation is:

$$out = log(e^x + 1)$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def softplus_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.softplus(x)


x = np.array([-1, 0, 1]).astype(np.float32)
out = softplus_Job(x)

# out [0.31326166 0.6931472  1.3132616 ]
```

oneflow.math.**sqrt**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the sqrt root value of Blob.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def sqrt_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.sqrt(x)


x = np.array([4, 16, 25]).astype(np.float32)
out = sqrt_Job(x)

# out [2. 4. 5.]
```

oneflow.math.**square**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the square value of Blob.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def square_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.square(x)


x = np.array([2, 3, 4]).astype(np.float32)
out = square_Job(x)

# out [ 4.  9. 16.]
```

oneflow.math.**squared_difference**(*x: Union[int, float, oneflow_api.BlobDesc], y: Union[int, float, oneflow_api.BlobDesc], name: Optional[str] = None*) → oneflow_api.BlobDesc

This op computes $(x - y)^2$ element-wise.

Parameters

- **x** (*Union[int, float, oneflow_api.BlobDesc]*) – A Blob
- **y** (*Union[int, float, oneflow_api.BlobDesc]*) – A Blob with the same type of x
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def squared_difference_Job(x: tp.Numpy.Placeholder((4, )),
                           y: tp.Numpy.Placeholder((4, ))
)->tp.Numpy:
    return flow.math.squared_difference(x, y)

x = np.array([1, 2, 3, 4], dtype=np.float32)
y = np.array([2, 4, 6, 8], dtype=np.float32)

out = squared_difference_Job(x, y)

# out [ 1.  4.  9. 16.]
```

oneflow.math.**subtract**(*x: Union[int, float, oneflow_api.BlobDesc], y: Union[int, float, oneflow_api.BlobDesc], name: Optional[str] = None*) → oneflow_api.BlobDesc

Compute $X - Y$ element-wise.

The equation is:

$$out = X - Y$$

Parameters

- **x** (*Union[int, float, oneflow_api.BlobDesc]*) – A Blob.
- **y** (*Union[int, float, oneflow_api.BlobDesc]*) – A Blob has the same type of x.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob after subtracting, has the same type as x.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def subtractJob(x: tp.Numpy.Placeholder((3, )),
                y: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.subtract(x, y)

x = np.array([1, 2, 3]).astype(np.float32)
y = np.array([2, 4, 1]).astype(np.float32)
out = subtractJob(x, y)

# out [-1., -2., 2.]
```

oneflow.math.**tan**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the tan value of Blob.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def tan_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.tan(x)


x = np.array([-1/4*np.pi, 0, 1/4*np.pi]).astype(np.float32)
out = tan_Job(x)

# out [-1.  0.  1.]
```

oneflow.math.**tanh**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the hyperbolic tangent value of Blob.

The equation is:

$$out = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def tanh_Job(x: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.tanh(x)
```

(continues on next page)

(continued from previous page)

```
x = np.array([-1, 0, 1]).astype(np.float32)
out = tanh_Job(x)

# out [-0.7615942  0.         0.7615942]
```

oneflow.math.**tanh_v2**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the hyperbolic tangent value of Blob.

The equation is:

$$out = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

oneflow.math.**top_k**(*input: oneflow_api.BlobDesc*, *axis: int = -1*, *k: int = 1*, *sorted: bool = True*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Finds the indices of the k largest entries at specified axis, the difference between other framework is that oneflow only return the indices.

Parameters

- **input** (*oneflow_api.BlobDesc*) – The input Blob
- **axis** (*int, optional*) – dimension to be calculated. Defaults to the last dim (-1)
- **k** (*int, optional*) – Number of top elements to look for along the last dimension. Defaults to 1.
- **sorted** (*bool, optional*) – If true the resulting k elements will be sorted by the values in descending order. Defaults to True.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob(dtype=int32) contains the indices of the k largest elements.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def topk_Job(x: tp.Numpy.Placeholder((5, ))
)->tp.Numpy:
    return flow.math.top_k(x, 2)

x = np.array([1, 3, 8, 7, 2], dtype=np.float32)
out = topk_Job(x)

# out [2 3]
```

oneflow.math.**tril**(*x: oneflow_api.BlobDesc*, *diagonal: int = 0*, *fill_value: Union[int, float] = 0*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Compute lower triangle of an matrix.

Parameters

- **x** (*oneflow_api.BlobDesc*) – Input Blob.
- **diagonal** (*int*) – Diagonal offset, when diagonal > 0, diagonal offset up, otherwise, offset downward.
- **fill_value** (*Union[int, float]*) – The value filled into the upper triangle.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Attention: The dimension of x must greater or equal to 2.

Returns The lower triangle blob of input.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp
@flow.global_function()
def tril_Job(x: tp.Numpy.Placeholder((4, 4))
)->tp.Numpy:
    return flow.math.tril(x, 0)
x = np.array([[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]],
             dtype=np.float32)
out = tril_Job(x).get()

# output [[1, 0, 0, 0],
          [1, 2, 0, 0],
          [1, 2, 3, 0],
          [1, 2, 3, 4]]
```

oneflow.math.**two_stage_reduce_max**(*x: oneflow_api.BlobDesc, axis: Union[int, Sequence[int], None] = None, keepdims: bool = False, name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.math.**two_stage_reduce_min**(*x: oneflow_api.BlobDesc, axis: Union[int, Sequence[int], None] = None, keepdims: bool = False, name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.math.**unsorted_batch_segment_sum**(*data: oneflow_api.BlobDesc*, *segment_ids: oneflow_api.BlobDesc*, *num_segments: int*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

It is similar with *unsorted_segment_sum*, the difference is that *unsorted_batch_segment_sum* brings a *batch axis*. We can do the segment sum in different batch of data.

For example, the segment id is like:

```
[[0 0 0 1 2 2 3 3],
 [0 0 1 1 2 3 3 3]]
```

Parameters

- **data** (*oneflow_api.BlobDesc*) – Input Blob
- **segment_ids** (*oneflow_api.BlobDesc*) – A Blob with shape (d0, d1). The d0, d1 are the first and second dimension of data.
- **num_segments** (*int*) – num_segments should equal the number of distinct segment IDs.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def unsorted_batch_segment_sum_Job(data: tp.Numpy.Placeholder((3, 4)),
                                   segment_ids: tp.Numpy.Placeholder((3, 4),␣
↪dtype=flow.int32)
)->tp.Numpy:
    return flow.math.unsorted_batch_segment_sum(data, segment_ids, 2)

input_blob = np.array([[1, 2, 3, 4],
                       [1, 2, 3 ,4],
                       [1, 2, 3, 4]]).astype(np.float32)
segment_ids = np.array([[0, 0, 0, 1],
                        [0, 0, 1, 0],
                        [0, 1, 0, 0]]).astype(np.int32)
out = unsorted_batch_segment_sum_Job(input_blob, segment_ids)

# out [[6. 4.]
#      [7. 3.]
#      [8. 2.]]
```

oneflow.math.**unsorted_segment_sum**(*data: oneflow_api.BlobDesc*, *segment_ids: oneflow_api.BlobDesc*, *num_segments: int*, *axis: int = 0*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Computes the sum along segments of a Blob.

Parameters

- **data** (*oneflow_api.BlobDesc*) – Input Blob
- **segment_ids** (*oneflow_api.BlobDesc*) – A Blob should be the size of the first dimension, with consecutive IDs in the range 0 to k (k < d0).
- **num_segments** (*int*) – num_segments should equal the number of distinct segment IDs.
- **axis** (*int, optional*) – The axis of data. Defaults to 0.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob with the same type of data.

Return type oneflow_api.BlobDesc

For example:

```
# Example 1:
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def unsorted_segment_sumJob(data: tp.Numpy.Placeholder((3, 4)),
                            segment_ids: tp.Numpy.Placeholder((4, ), dtype=flow.
↪int32)
)->tp.Numpy:
    return flow.math.unsorted_segment_sum(data, segment_ids, num_segments=2,␣
↪axis=1)

input_blob = np.array([[1, 2, 3, 4],
                       [5, 6, 7 ,8],
                       [9, 10, 11, 12]]).astype(np.float32)
segment_ids = np.array([0, 1, 0, 1]).astype(np.int32)
out = unsorted_segment_sumJob(input_blob, segment_ids)

# out [[ 4.  6.]
#      [12. 14.]
#      [20. 22.]]

# Example 2
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def unsorted_segment_sumJob(data: tp.Numpy.Placeholder((3, 4)),
                            segment_ids: tp.Numpy.Placeholder((3, ), dtype=flow.
↪int32)
)->tp.Numpy:
    return flow.math.unsorted_segment_sum(data, segment_ids, num_segments=2,␣
↪axis=0)

input_blob = np.array([[1, 2, 3, 4],
                       [5, 6, 7 ,8],
                       [9, 10, 11, 12]]).astype(np.float32)
segment_ids = np.array([0, 1, 0]).astype(np.int32)
out = unsorted_segment_sumJob(input_blob, segment_ids)

#  out [[10. 12. 14. 16.]
#       [ 5.  6.  7.  8.]]
```

oneflow.math.**unsorted_segment_sum_like**(*data: oneflow_api.BlobDesc*, *segment_ids: oneflow_api.BlobDesc*, *like: oneflow_api.BlobDesc*, *axis: int = 0*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Computes the sum along segments of a Blob, the output shape is the same as the *like* Blob.

Parameters

- **data** (*oneflow_api.BlobDesc*) – Input Blob
- **segment_ids** (*oneflow_api.BlobDesc*) – A Blob should be the size of the first dimension, with consecutive IDs in the range 0 to k (k < d0).
- **like** (*oneflow_api.BlobDesc*) – The input Blob which specifies shape

- **axis** (*int, optional*) – The axis of data. Defaults to 0.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def unsorted_segment_sum_like_Job(data: tp.Numpy.Placeholder((3, 4)),
                                  segment_ids: tp.Numpy.Placeholder((3, ),␣
↪dtype=flow.int32),
                                  like: tp.Numpy.Placeholder((2, 4), dtype=flow.
↪float32)
)->tp.Numpy:
    return flow.math.unsorted_segment_sum_like(data, segment_ids, like, axis=0)

input_blob = np.array([[1, 2, 3, 4],
                       [5, 6, 7 ,8],
                       [9, 10, 11, 12]]).astype(np.float32)
segment_ids = np.array([0, 1, 0]).astype(np.int32)
like = np.zeros(shape=(2, 4), dtype=np.float32)

out = unsorted_segment_sum_like_Job(input_blob, segment_ids, like)

# out [[10. 12. 14. 16.]
#      [ 5.  6.  7.  8.]]
```

oneflow.math.**xdivy**(*x: oneflow_api.BlobDesc*, *y: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the result of x/y

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **y** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def xdivy_Job(x: tp.Numpy.Placeholder((3,)),
              y: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.xdivy(x, y)
```

(continues on next page)

(continued from previous page)

```
x = np.array([4, 3, 5]).astype(np.float32)
y = np.array([3, 2, 2]).astype(np.float32)
out = xdivy_Job(x, y)

# out [1.3333334 1.5       2.5      ]
```

oneflow.math.**xlogy**(*x: oneflow_api.BlobDesc*, *y: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the result of $x * log(y)$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **y** (*oneflow_api.BlobDesc*) – A Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def xlogy_Job(x: tp.Numpy.Placeholder((3,)),
              y: tp.Numpy.Placeholder((3,))
) -> tp.Numpy:
    return flow.math.xlogy(x, y)


x = np.array([2, 2, 2]).astype(np.float32)
y = np.array([4, 8, 16]).astype(np.float32)
out = xlogy_Job(x, y)

# out [2.7725887 4.158883  5.5451775]
```

CHAPTER

EIGHT

ONEFLOW.NN

8.1 Operators for neural networks

oneflow.nn.**BCELoss**(*input: oneflow_api.BlobDesc, target: oneflow_api.BlobDesc, weight: <module 'oneflow.python.framework.remote_blob' from '/home/docs/checkouts/readthedocs.org/user_builds/oneflow/envs/master/lib/python3.7/site-packages/oneflow/python/framework/remote_blob.py'> = None, reduction: str = 'mean', name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the binary cross entropy loss.

The equation is:

if reduction = "none":

$$out = -(Target_i * log(Input_i) + (1 - Target_i) * log(1 - Input_i))$$

if reduction = "mean":

$$out = -\frac{1}{n}\sum_{i=1}^{n}(Target_i * log(Input_i) + (1 - Target_i) * log(1 - Input_i))$$

if reduction = "sum":

$$out = -\sum_{i=1}^{n}(Target_i * log(Input_i) + (1 - Target_i) * log(1 - Input_i))$$

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def bce_loss_job(input: tp.Numpy.Placeholder(shape=(2, 3)),
                         target: tp.Numpy.Placeholder(shape=(2, 3)),
                         weight: tp.Numpy.Placeholder(shape=(2, 3)))->tp.Numpy:
    sigmoid_input = flow.math.sigmoid(input)
    return flow.nn.BCELoss(sigmoid_input, target, weight, reduction='mean')


np_input = np.array([[1.2, 0.2, -0.3],
                     [0.7, 0.6, -2]]).astype(np.float32)
```

(continues on next page)

(continued from previous page)

```
np_target = np.array([[0, 1, 0],
                      [1, 0, 1]]).astype(np.float32)

np_weight = np.array([[2, 2, 2],
                      [2, 2, 2]]).astype(np.float32)

# output [2.0611262]
```

Parameters

- **input** (*oneflow_api.BlobDesc*) – The input Blob.
- **target** (*oneflow_api.BlobDesc*) – The target value.
- **weight** (*remote_blob_util, optional*) – The manual rescaling weight to the loss. Default to None, whose corresponding weight value is 1.
- **reduction** (*str, optional*) – The reduce type, it can be one of "none", "mean", "sum". Defaults to "mean".
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Attention: The input value must be in the range of (0, 1). Or the loss function may return *nan* value.

Returns The result Blob.

Return type oneflow_api.BlobDesc

oneflow.nn.**BCEWithLogitsLoss**(*input: oneflow_api.BlobDesc*, *target: oneflow_api.BlobDesc*, *weight: <module 'oneflow.python.framework.remote_blob' from '/home/docs/checkouts/readthedocs.org/user_builds/oneflow/envs/master/lib/python3.7/site-packages/oneflow/python/framework/remote_blob.py'> = None*, *pos_weight: <module 'oneflow.python.framework.remote_blob' from '/home/docs/checkouts/readthedocs.org/user_builds/oneflow/envs/master/lib/python3.7/site-packages/oneflow/python/framework/remote_blob.py'> = None*, *reduction: str = 'mean'*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator combines the *Sigmoid* and *BCELoss* together. For numerical stability, we apply some math tricks instead of using *Sigmoid* layer with *BCELoss*.

The equation is:

if reduction = "none":

$$out = -weight * [Pos\_weight * y * log\sigma(x) + (1 - y) * log(1 - \sigma(x))]$$

if reduction = "mean":

$$out = -\frac{weight}{n}\sum_{i=1}^{n}[Pos\_weight * y * log\sigma(x) + (1 - y) * log(1 - \sigma(x))]$$

if reduction = "sum":

$$out = -weight * \sum_{i=1}^{n}[Pos\_weight * y * log\sigma(x) + (1 - y) * log(1 - \sigma(x))]$$

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def bce_with_logits_loss_job(input: tp.Numpy.Placeholder(shape=(2, 3)),
                             target: tp.Numpy.Placeholder(shape=(2, 3)),
                             weight: tp.Numpy.Placeholder(shape=(2, 3)),
                             pos_weight: tp.Numpy.Placeholder(shape=(3, )))->tp.
↪Numpy:
    return flow.nn.BCEWithLogitsLoss(input, target, weight, pos_weight, reduction=
↪'mean')


np_input = np.array([[1.2, 0.2, -0.3],
                     [0.7, 0.6, -2]]).astype(np.float32)

np_target = np.array([[0, 1, 0],
                      [1, 0, 1]]).astype(np.float32)

np_weight = np.array([[2, 2, 2],
                      [2, 2, 2]]).astype(np.float32)

np_pos_weight = np.array([1.2, 1.3, 1.4]).astype(np.float32)

out = bce_with_logits_loss_job(np_input, np_target, np_weight, np_pos_weight)

# output [2.4314096]
```

Parameters

- **input** (*oneflow_api.BlobDesc*) – The input Tensor.
- **target** (*oneflow_api.BlobDesc*) – The target Tensor.
- **weight** (*remote_blob_util, optional*) – The manual rescaling weight to the loss. Defaults to None.
- **pos_weight** (*remote_blob_util, optional*) – The manual rescaling weight to the positive examples. Defaults to None.
- **reduction** (*str, optional*) – The reduce type, it can be one of "none", "mean", "sum". Defaults to "mean".
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

oneflow.nn.**GroupNorm**(*x: oneflow_api.BlobDesc*, *num_groups: int = 32*, *eps: float = 1e-05*, *affine: bool = True*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Applies Group Normalization over a ND(N>=3) input.

Parameters

- **x** (*oneflow_api.BlobDesc*) – input tensor with shape (N,C,), where C means the number of channels.

- **eps** (*float*) – A value added to the denominator for numerical stability. Default: 1e-5.
- **affine** (*bool*) – A boolean value that when set to True, this module has learnable affine parameters, initialized the same way as done for batch normalization. Default: True.
- **name** (*Optional[str], optional*) – Name of this op.

Returns The normalized input tensor.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def group_norm_Job(x: tp.Numpy.Placeholder((4, 4, 32, 32))
) -> tp.Numpy:
    group_norm = flow.nn.GroupNorm(
        x,
        num_group=2,
        eps=1e-5,
        affine=True,
    )
    return group_norm

x = np.random.random(size=(4, 4, 32, 32)).astype(np.float32)
out = group_norm_Job(x)
```

oneflow.nn.**InstanceNorm1d**(*x: oneflow_api.BlobDesc*, *eps: float = 1e-05*, *affine: bool = True*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Applies Instance Normalization over a 3D input.

Parameters

- **x** (*oneflow_api.BlobDesc*) – 3D input tensor with NCL data layout.
- **eps** (*float*) – A value added to the denominator for numerical stability. Default: 1e-5.
- **affine** (*bool*) – A boolean value that when set to True, this module has learnable affine parameters, initialized the same way as done for batch normalization. Default: True.
- **name** (*Optional[str], optional*) – Name of this op.

Returns The normalized input tensor.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def instance_norm_Job(x: tp.Numpy.Placeholder((4, 2, 32))
) -> tp.Numpy:
    instance_norm = flow.nn.InstanceNorm1d(
        x,
        eps=1e-5,
```

(continues on next page)

(continued from previous page)

```
        affine=True,
    )
    return instance_norm

x = np.random.random(size=(4, 2, 32)).astype(np.float32)
out = instance_norm_Job(x)
```

oneflow.nn.**InstanceNorm2d**(*x: oneflow_api.BlobDesc*, *eps: float = 1e-05*, *affine: bool = True*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Applies Instance Normalization over a 4D input.

Parameters

- **x** (*oneflow_api.BlobDesc*) – 4D input tensor with NCHW data layout.
- **eps** (*float*) – A value added to the denominator for numerical stability. Default: 1e-5.
- **affine** (*bool*) – A boolean value that when set to True, this module has learnable affine parameters, initialized the same way as done for batch normalization. Default: True.
- **name** (*Optional[str], optional*) – Name of this op.

Returns The normalized input tensor.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def instance_norm_Job(x: tp.Numpy.Placeholder((4, 2, 32, 32))
) -> tp.Numpy:
    instance_norm = flow.nn.InstanceNorm2d(
        x,
        eps=1e-5,
        affine=True,
    )
    return instance_norm

x = np.random.random(size=(4, 2, 32, 32)).astype(np.float32)
out = instance_norm_Job(x)
```

oneflow.nn.**InstanceNorm3d**(*x: oneflow_api.BlobDesc*, *eps: float = 1e-05*, *affine: bool = True*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Applies Instance Normalization over a 5D input.

Parameters

- **x** (*oneflow_api.BlobDesc*) – 5D input tensor with NCDHW data layout.
- **eps** (*float*) – A value added to the denominator for numerical stability. Default: 1e-5.
- **affine** (*bool*) – A boolean value that when set to True, this module has learnable affine parameters, initialized the same way as done for batch normalization. Default: True.
- **name** (*Optional[str], optional*) – Name of this op.

Returns The normalized input tensor.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def instance_norm_Job(x: tp.Numpy.Placeholder((4, 2, 32, 32, 32))
) -> tp.Numpy:
    instance_norm = flow.nn.InstanceNorm2d(
        x,
        eps=1e-5,
        affine=True,
    )
    return instance_norm

x = np.random.random(size=(4, 2, 32, 32, 32)).astype(np.float32)
out = instance_norm_Job(x)
```

oneflow.nn.**KLDivLoss**(*input: oneflow_api.BlobDesc*, *target: oneflow_api.BlobDesc*, *log_target: bool = False*, *reduction: str = 'mean'*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the Kullback-Leiber divergence loss.

The equation is:

If $log\_target = True$:

$$loss = e^{target} * (target - input)$$

If $log\_target = False$:

$$loss = target * (log(target) - input)$$

Attention: In *log_target = False* case, the element in loss will set to be *0* when the element in target is less than *0*

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def of_kldivloss(input: tp.Numpy.Placeholder(shape=(3, 3)),
                 target: tp.Numpy.Placeholder(shape=(3, 3))) -> tp.Numpy:
    return flow.nn.KLDivLoss(input, target, log_target=False, reduction='none')


input = np.array([[0.1, 0.2, 0.7],
            [0.8, 0.9, 0.5],
            [0.5, 0.15, 0.35]]).astype(np.float32)
target = np.array([[0.3, 0.1, 0.6],
            [-0.3, 0.4, 0.4],
            [0.35, 0.25, 0.4]]).astype(np.float32)
```

(continues on next page)

(continued from previous page)

```
out = of_kldivloss(input, target)

# output [[-0.39119187 -0.25025854 -0.7264954 ]
#         [ 0.         -0.72651625 -0.56651634]
#         [-0.54243773 -0.3840736  -0.5065163 ]]
```

Parameters

- **input** (*oneflow_api.BlobDesc*) – The input tensor.
- **target** (*oneflow_api.BlobDesc*) – The target tensor.
- **log_target** (*bool, optional*) – Whether the *target* is passed in the log space. Defaults to False.
- **reduction** (*str, optional*) – The reduce type, it can be one of "none", "mean", "sum". Defaults to "mean".
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result tensor.

Return type oneflow_api.BlobDesc

oneflow.nn.**L1Loss**(*input: oneflow_api.BlobDesc*, *target: oneflow_api.BlobDesc*, *reduction: str = 'mean'*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the L1 Loss between each element in *input* and *target*.

The equation is:

if reduction = "none":

$$output = |Target - Input|$$

if reduction = "mean":

$$output = \frac{1}{n}\sum_{i=1}^{n}|Target_i - Input_i|$$

if reduction = "sum":

$$output = \sum_{i=1}^{n}|Target_i - Input_i|$$

Parameters

- **input** (*oneflow_api.BlobDesc*) – The input Blob.
- **target** (*oneflow_api.BlobDesc*) – The target value.
- **reduction** (*str*) – The reduce type, it can be one of "none", "mean", "sum". Defaults to "mean".
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def l1_job(x: tp.Numpy.Placeholder(shape=(3, 3)),
        y: tp.Numpy.Placeholder(shape=(3, 3))) -> tp.Numpy:
    out = flow.nn.L1Loss(x, y, reduction="mean", name="l1")

    return out


input = np.array([[1, 1, 1], [2, 2, 2], [7, 7, 7]]).astype(np.float32)
target = np.array([[4, 4, 4], [4, 4, 4], [4, 4, 4]]).astype(np.float32)

out = l1_job(input, target)

# output [2.6666667]
```

Example 2:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def l1_job(x: tp.Numpy.Placeholder(shape=(3, 3)),
        y: tp.Numpy.Placeholder(shape=(3, 3))) -> tp.Numpy:
    out = flow.nn.L1Loss(x, y, reduction="sum", name="l1")

    return out


input = np.array([[1, 1, 1], [2, 2, 2], [7, 7, 7]]).astype(np.float32)
target = np.array([[4, 4, 4], [4, 4, 4], [4, 4, 4]]).astype(np.float32)

out = l1_job(input, target)

# output [24.]
```

oneflow.nn.**MSELoss**(*input: oneflow_api.BlobDesc*, *target: oneflow_api.BlobDesc*, *reduction: str = 'mean'*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the mean squared error between each element in *input* and *target*.

The equation is:

if reduction = "none":

$$out = (Target_i - Input_i)^2$$

if reduction = "mean":

$$out = \frac{1}{n}\sum_{i=1}^{n}(Target_i - Input_i)^2$$

if reduction = “sum”:

$$out = \sum_{i=1}^{n}(Target_i - Input_i)^2$$

Parameters

- **input** (*oneflow_api.BlobDesc*) – The input Blob.
- **target** (*oneflow_api.BlobDesc*) – The target value.
- **reduction** (*str*) –
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

Example 1:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def mseloss_job(input: tp.Numpy.Placeholder(shape=(3, 3)),
                target: tp.Numpy.Placeholder(shape=(3, 3)))->tp.Numpy:
    out = flow.nn.MSELoss(input, target, reduction="mean")
    return out

input = np.array([[1, 1, 1], [2, 2, 2], [7, 7, 7]]).astype(np.float32)
target = np.array([[4, 4, 4], [4, 4, 4], [4, 4, 4]]).astype(np.float32)

out = mseloss_job(input, target)

# output [7.3333335]
```

Example 2:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def mseloss_job(input: tp.Numpy.Placeholder(shape=(3, 3)),
                target: tp.Numpy.Placeholder(shape=(3, 3)))->tp.Numpy:
    out = flow.nn.MSELoss(input, target, reduction="sum")
    return out

input = np.array([[1, 1, 1], [2, 2, 2], [7, 7, 7]]).astype(np.float32)
target = np.array([[4, 4, 4], [4, 4, 4], [4, 4, 4]]).astype(np.float32)

out = mseloss_job(input, target)

# output [66.]
```

oneflow.nn.**MarginRankingLoss**(*input1: oneflow_api.BlobDesc*, *input2: oneflow_api.BlobDesc*, *target: oneflow_api.BlobDesc*, *margin: float = 0.0*, *reduction: str = 'mean'*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the Margin Ranking loss.

The equation is:

if reduction = "none":

$$out = \max\left(0, -y * (x_1 - x_2) + margin\right)$$

if reduction = "mean":

$$out = \frac{1}{n}\sum_{i=1}^{n} \max\left(0, -y * (x_1 - x_2) + margin\right)$$

if reduction = "sum":

$$out = \sum_{i=1}^{n} \max\left(0, -y * (x_1 - x_2) + margin\right)$$

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def margin_ranking_loss_job(input1: tp.Numpy.Placeholder(shape=(3, 3)),
                            input2: tp.Numpy.Placeholder(shape=(3, 3)),
                            target: tp.Numpy.Placeholder(shape=(3, 3)))->tp.Numpy:
    out = flow.nn.MarginRankingLoss(input1, input2, target, margin=1.0)
    return out

np_input1 = np.array([[1, 2, 3],
                      [4, 5, 6],
                      [7, 8, 9]]).astype(np.float32)
np_input2 = np.array([[2, 2, 2],
                      [2, 2, 2],
                      [2, 2, 2]]).astype(np.float32)
np_target = np.array([[3, 3, 3],
                      [3, 3, 3],
                      [3, 3, 3]]).astype(np.float32)

out = margin_ranking_loss_job(np_input1, np_input2, np_target)

# output [0.5555556]
```

Parameters

- **input1** (*oneflow_api.BlobDesc*) – The ranking score of input1 Blob.
- **input2** (*oneflow_api.BlobDesc*) – The ranking score of input2 Blob.
- **target** (*oneflow_api.BlobDesc*) – The target Blob.
- **margin** (*float*) – The margin value. Defaults to 0.0.
- **reduction** (*str, optional*) – The reduce type, it can be one of "none", "mean", "sum". Defaults to "mean".

- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

class oneflow.nn.**Module**(*name=None*)

__init__(*name=None*)
Initialize self. See help(type(self)) for accurate signature.

property call_seq_no

forward(**args*)

property module_name

oneflow.nn.**PixelShuffle**(*input: oneflow_api.BlobDesc*, *upscale_factor: int*, *name: Optional[str] = None*) → oneflow_api.BlobDesc
This operator do the pixel shuffle, the shape of input(B, C*r*r, H, W) is arranged to (B, C, H*r, W*r). It can be used to do the sub-pixel convolution.

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def PixelShuffleJob(input: tp.Numpy.Placeholder(shape=(3, 4, 2, 2), dtype=flow.
↪float32))->tp.Numpy:
    out = flow.nn.PixelShuffle(input, upscale_factor=2)

    return out

input = np.random.uniform(size=(3, 4, 2, 2)).astype(np.float32)
out = PixelShuffleJob(input)

# out.shape (3, 1, 4, 4)
```

Parameters

- **input** (*oneflow_api.BlobDesc*) – The input Blob.
- **upscale_factor** (*int*) – The upscale factor.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

oneflow.nn.**PixelShufflev2**(*input: oneflow_api.BlobDesc*, *h_upscale_factor: int*, *w_upscale_factor: int*, *name: Optional[str] = None*) → oneflow_api.BlobDesc
This operator is similar to *oneflow.nn.PixelShuffle*. The difference is that in *oneflow.nn.PixelShuffle*, the upscale factor of height and width is the same. But in *oneflow.nn.PixelShufflev2*, you can set different upscale factor for height and width.

Parameters

- **input** (*oneflow_api.BlobDesc*) – The input Blob.

- **h_upscale_factor** (*int*) – The upscale factor of height.
- **w_upscale_factor** (*int*) – The upscale factor of width.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def PixelShufflev2Job(input: tp.Numpy.Placeholder(shape=(3, 16, 2, 4), dtype=flow.
↪float32))->tp.Numpy:
    out = flow.nn.PixelShufflev2(input, h_upscale_factor=2, w_upscale_factor=4)

    return out

input = np.random.uniform(size=(3, 16, 2, 4)).astype(np.float32)
out = PixelShuffleJob(input)

# out.shape (3, 2, 4, 16)
```

Returns The result Blob.

Return type oneflow_api.BlobDesc

oneflow.nn.**TripletMarginLoss**(*anchor: oneflow_api.BlobDesc*, *positive: oneflow_api.BlobDesc*, *negative: oneflow_api.BlobDesc*, *margin: float = 1.0*, *p: float = 2.0*, *eps: float = 1e-06*, *swap: bool = False*, *reduction: str = 'mean'*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the Triplet Margin Loss.

The equation is:

if reduction = "none":

$$output = \max\{\|a_i - p_i\|_p - \|a_i - n_i\|_p + \text{margin}, 0\}$$

if reduction = "mean":

$$output = \frac{1}{n}\sum_{i=1}^{n} \max\{\|a_i - p_i\|_p - \|a_i - n_i\|_p + \text{margin}, 0\}$$

if reduction = "sum":

$$output = \sum_{i=1}^{n} \max\{\|a_i - p_i\|_p - \|a_i - n_i\|_p + \text{margin}, 0\}$$

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
```

(continues on next page)

(continued from previous page)

```
def triplet_loss_job(anchor: tp.Numpy.Placeholder(shape=(3, 3)),
                     pos: tp.Numpy.Placeholder(shape=(3, 3)),
                     neg: tp.Numpy.Placeholder(shape=(3, 3)))->tp.Numpy:
    out = flow.nn.TripletMarginLoss(anchor, pos, neg, margin=1.0, p=2.0)
    return out

np_anchor = np.array([[1, 2, 3],
                      [4, 5, 6],
                      [7, 8, 9]]).astype(np.float32)
np_pos = np.array([[2, 2, 2],
                   [2, 2, 2],
                   [2, 2, 2]]).astype(np.float32)
np_neg = np.array([[3, 3, 3],
                   [3, 3, 3],
                   [3, 3, 3]]).astype(np.float32)

out = triplet_loss_job(np_anchor, np_pos, np_neg)

# output [1.8449262]
```

Parameters

- **anchor** (*oneflow_api.BlobDesc*) – The anchor Blob.
- **positive** (*oneflow_api.BlobDesc*) – The positive sample Blob.
- **negative** (*oneflow_api.BlobDesc*) – The negative sample Blob.
- **margin** (*float, optional*) – The margin value. Defaults to 1.0.
- **p** (*float, optional*) – The norm degree for computing distance. Defaults to 2.0.
- **eps** (*float, optional*) – A small value use in norm computation. Defaults to 1e-6.
- **swap** (*bool, optional*) – Whether to swap the distance.
- **more details you can check the Paper Learning shallow convolutional feature descriptors with triplet losses. Defaults to False.** (*For*) –
- **reduction** (*str, optional*) – The reduce type, it can be one of "none", "mean", "sum". Defaults to "mean".
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

oneflow.nn.**avg_pool1d**(*input: oneflow_api.BlobDesc, ksize: Union[int, Sequence[int]], strides: Union[int, Sequence[int]], padding: Union[str, Sequence[Sequence[int]]], data_format: str = 'NCW', name: Optional[str] = None*) → oneflow_api.BlobDesc

Performs the average pooling on the input *Blob*.

Parameters

- **input** (*oneflow_api.BlobDesc*) – A 3-D *Blob* of the format specified by data_format.

- **ksize** (*Union[int, Sequence[int]]*) – An int or list of ints that has length 1 or 3. The size of the window for each dimension of the input *Blob*.
- **strides** (*Union[int, Sequence[int]]*) – An int or list of ints that has length 1 or 3. The stride of the sliding window for each dimension of the input *Blob*.
- **padding** (*str*) – '*VALID*' or '*SAME*'.
- **data_format** (*str, optional*) – '*NWC*' or '*NCW*'. Defaults to '*NWC*'.
- **name** (*Optional[str], optional*) – This operator's name(optional). Defaults to None.

Raises **NotImplementedError** – TODO: fix cuDNN bugs in pooling_1d

Returns A *Blob* of format specified by data_format. The max pooled output *Blob*.

Return type oneflow_api.BlobDesc

oneflow.nn.**avg_pool2d**(*input: oneflow_api.BlobDesc, ksize: Union[int, Tuple[int, int]], strides: Union[int, Tuple[int, int]], padding: Union[str, Tuple[Tuple[int, int], Tuple[int, int], Tuple[int, int], Tuple[int, int]]], data_format: str = 'NCHW', ceil_mode: bool = False, name: Optional[str] = None*) → oneflow_api.BlobDesc

Performs the 2d-average pooling on the input.

Parameters

- **input** (*oneflow_api.BlobDesc*) – A 4-D *Blob* of shape [batch, height, width, channels].
- **ksize** (*Union[int, IntPair]*) – An int or list of ints that has length 1, 2. The size of the window for each dimension of the input *Blob*.
- **strides** (*Union[int, IntPair]*) – An int or list of ints that has length 1, 2. The stride of the sliding window for each dimension of the input *Blob*.
- **padding** (*str*) – '*VALID*' or '*SAME*' or '*SAME_LOWER*' or '*SAME_UPPER*' or Tuple[IntPair, IntPair, IntPair, IntPair]. The padding algorithm.
- **data_format** (*str, optional*) – '*NHWC*' or '*NCHW*'. Defaults to "NCHW".
- **name** (*Optional[str], optional*) – This operator's name(optional). Defaults to None.

Returns A *Blob* with the same type as '*value*'. The average pooled output *Blob*.

Return type oneflow_api.BlobDesc

For example:

```
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def avgpool2d_Job(x: tp.Numpy.Placeholder((1, 32, 128, 128))
) -> tp.Numpy:
    pool_out = flow.nn.avg_pool2d(
        input=x,
        ksize=3,
        strides=2,
        padding='SAME',
        data_format='NCHW'
```

(continues on next page)

(continued from previous page)

```
    )

    return pool_out


x = np.random.randn(1, 32, 128, 128).astype(np.float32)
out = avgpool2d_Job(x)

# out.shape (1, 32, 64, 64)
```

oneflow.nn.**avg_pool3d**(*input: oneflow_api.BlobDesc, ksize: Union[int, Sequence[int]], strides: Union[int, Sequence[int]], padding: Union[str, Sequence[Sequence[int]]], data_format: str = 'NCDHW', ceil_mode: bool = False, name: Optional[str] = None*) → oneflow_api.BlobDesc

Performs the 3d-average pooling on the input.

Parameters

- **input** (*oneflow_api.BlobDesc*) – A 5-D *Blob* of shape [batch, height, width, channels].
- **ksize** (*Union[int, Sequence[int]]*) – An int or list of ints that has length 1, 3 or 5. The size of the window for each dimension of the input *Blob*.
- **strides** (*Union[int, Sequence[int]]*) – An int or list of ints that has length 1, 3 or 5. The stride of the sliding window for each dimension of the input *Blob*.
- **padding** (*str*) – '*VALID*' or '*SAME*' or '*SAME_LOWER*' or '*SAME_UPPER or Sequence[Sequence[int]]*'.
- **data_format** (*str, optional*) – '*NDHWC*' or '*NCDHW*'. Defaults to "NCDHW".
- **name** (*Optional[str], optional*) – This operator's name(optional).Defaults to None.

Returns A *Blob* with the same type as value. The average pooled output *Blob*.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def avgpool3d_Job(x: tp.Numpy.Placeholder((1, 32, 10, 128, 128))
) -> tp.Numpy:
    pool_out = flow.nn.avg_pool3d(
        input=x,
        ksize=3,
        strides=2,
        padding='SAME',
        data_format='NCDHW'
    )

    return pool_out
```

(continues on next page)

(continued from previous page)

```
x = np.random.randn(1, 32, 10, 128, 128).astype(np.float32)
out = avgpool3d_Job(x)

# out.shape (1, 32, 5, 64, 64)
```

oneflow.nn.**batch_normalization**(*x: oneflow_api.BlobDesc*, *mean: oneflow_api.BlobDesc*, *variance: oneflow_api.BlobDesc*, *offset: Optional[oneflow_api.BlobDesc] = None*, *scale: Optional[oneflow_api.BlobDesc] = None*, *variance_epsilon: Optional[float] = 1e-05*, *axis: int = 1*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This op does not fully align with tf.nn.batch_normalization.

The *mean*, *variable*, *offset* and *scale* are always 1D. Users need to specify *axis* to 1 for NCHW data format.

Parameters

- **x** (*oneflow_api.BlobDesc*) – Input *Blob* of arbitrary dimensionality.
- **mean** (*oneflow_api.BlobDesc*) – A 1D mean *Blob*.
- **variance** (*oneflow_api.BlobDesc*) – A 1D variance *Blob*.
- **offset** (*Optional[oneflow_api.BlobDesc]*) – An 1D offset *Blob*, often denoted in equations, or None. If present, will be added to the normalized *Blob*.
- **scale** (*Optional[oneflow_api.BlobDesc]*) – A 1D scale *Blob*, often denoted in equations, or None. If present, the scale is applied to the normalized *Blob*.
- **variance_epsilon** (*float*) – A small float number to avoid dividing by 0.
- **axis** (*int, optional*) – 1 for '*NCHW*' data format. Defaults to 1.
- **name** (*Optional[str], optional*) – This operator's name.

Returns the normalized, scaled, offset *Blob*.

Return type oneflow_api.BlobDesc

Note: This api is more flexible, if you're new to OneFlow, it's more recommend to use *oneflow.layers.batch_normalization*

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def batch_norm_Job(x: tp.Numpy.Placeholder((1, 5))
) -> tp.Numpy:
    bn_mean, bn_variance = flow.nn.moments(x, axes=[1])
    batch_norm = flow.nn.batch_normalization(
        x,
        mean=bn_mean,
        variance=bn_variance,
        axis=0
    )
```

(continues on next page)

(continued from previous page)

```
    return batch_norm


x = np.array([[1, 2, 3, 4, 5]]).astype(np.float32)
out = batch_norm_Job(x)

# out [[-1.41421  -0.707105  0.        0.707105  1.41421 ]]
```

oneflow.nn.**bias_add**(*value: oneflow_api.BlobDesc*, *bias: oneflow_api.BlobDesc*, *data_format: Optional[str] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator adds a bias to Blob.

Parameters

- **value** (*oneflow_api.BlobDesc*) – A *Blob*.
- **bias** (*oneflow_api.BlobDesc*) – A 1-D *Blob* with size matching the channel dimension of value. And has the same type as value unless value is a quantized type.
- **data_format** (*Optional[str], optional*) – A string. *'N...C'* or *'NC...'*. Defaults to None.
- **name** (*Optional[str], optional*) – This operator's name. Defaults to None.

Raises **ValueError** – ValueError if data format is unrecognized, if value has less than two dimensions with *'N..C'*/None data_format or value has less than three dimensions with *'NC..'* data_format, if bias is a vector, or if the size of bias does not match the size of the channel dimension of value.

Returns A *Blob* with the same type as value.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def bias_add_Job(x: tp.Numpy.Placeholder((1, 64, 128, 128))
) -> tp.Numpy:
    bias_initializer = flow.truncated_normal(0.1)
    bias_regularizer = flow.regularizers.l2(0.0005)
    bias = flow.get_variable(
            "Add_bias",
            shape=(64,),
            initializer=bias_initializer,
            regularizer=bias_regularizer,
        )
    bias_out = flow.nn.bias_add(x, bias)
    return bias_out


x = np.random.randn(1, 64, 128, 128).astype(np.float32)
out = bias_add_Job(x)

# out.shape (1, 64, 128, 128)
```

oneflow.nn.**compat_conv2d**(*input: oneflow_api.BlobDesc, filters: oneflow_api.BlobDesc, strides: Union[int, Sequence[int]], padding: str, data_format: str = 'NCHW', dilations: Union[int, Sequence[int], None] = None, groups: int = 1, name: Optional[str] = None*) → oneflow_api.BlobDesc

Computes a 2-D convolution given *input* and 4-D *filters Blob*.

Parameters

- **input** (*oneflow_api.BlobDesc*) – A *Blob* of rank at least 4.
- **filters** (*oneflow_api.BlobDesc*) – A *Blob* with the same type as *input* and has the shape *[out_channels, in_channels//groups, filter_height, filter_width] for NCHW, or [out_channels, filter_height, filter_width, in_channels//groups] for NHWC*
- **strides** (*Union[int, Sequence[int]]*) – An int or list of *ints* that has length *1*, or *2*. The stride of the sliding window for each dimension of *input*.
- **padding** (*str*) – *"SAME"* or *"VALID"* indicating the type of padding algorithm to use, or a list indicating the explicit paddings at the start and end of each dimension.
- **data_format** (*str, optional*) – *"NHWC"* or *"NCHW"*. Defaults to *"NCHW"*.
- **dilations** (*Optional[Union[int, Sequence[int]]], optional*) – The dilation factor for each dimension of‘input‘. Defaults to None.
- **groups** (*int, optional*) – int value greater than 0. Defaults to 1.
- **name** (*Optional[str], optional*) – This operator's name. Defaults to None.

Raises

- **ValueError** – strides must be an int or a list.
- **ValueError** – data_format must be "NHWC" or "NCHW".
- **ValueError** – dilations length must be 2 when passed as a list.
- **ValueError** – dilations must be an int or a list.
- **ValueError** – data_format NHWC not support groups > 1.
- **ValueError** – invalid data_format.
- **ValueError** – padding must be "SAME" or "VALID".

Returns A *Blob* with the same type as *input* and the same outer batch shape.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


def conv2d(input, filters, kernel_size, strides, padding, name):
    input_shape = input.shape
    weight_initializer = flow.truncated_normal(0.1)
    weight_regularizer = flow.regularizers.l2(0.0005)
    weight_shape = (filters,
                    input_shape[1],
                    kernel_size[0],
                    kernel_size[1])
```

(continues on next page)

(continued from previous page)

```
    weight = flow.get_variable(
        name + "-weight",
        shape=weight_shape,
        initializer=weight_initializer,
        regularizer=weight_regularizer,
    )
    return flow.nn.compat_conv2d(input, weight, strides, padding, name=name)


@flow.global_function()
def conv2d_Job(x: tp.Numpy.Placeholder((1, 64, 32, 32))
) -> tp.Numpy:
    conv = conv2d(x,
                  filters=128,
                  kernel_size=[3, 3],
                  strides=2,
                  padding='SAME',
                  name="Convlayer")
    return conv


x = np.random.randn(1, 64, 32, 32).astype(np.float32)
out = conv2d_Job(x)

# out.shape (1, 128, 16, 16)
```

oneflow.nn.**conv1d**(*input: oneflow_api.BlobDesc, filters: oneflow_api.BlobDesc, strides: Union[int, Tuple[int]], padding: Union[str, Tuple[Tuple[int, int], Tuple[int, int], Tuple[int, int]]], data_format: str = 'NCW', dilations: Union[int, Tuple[int], None] = None, groups: int = 1, name: Optional[str] = None*) → oneflow_api.BlobDesc

1D convolution layer.

Parameters

- **input** (*oneflow_api.BlobDesc*) – A 3D input *Blob*. [batch_num, channel, width]
- **filters** (*oneflow_api.BlobDesc*) – A *Blob* with the same type as *input* and has the shape [out_channels, in_channels//groups, filter_width] for *NCW*, or [out_channels, filter_width, in_channels//groups] for *NWC*
- **strides** (*Union[int, Tuple[int]]*) – An int or list of *ints* that has length *1*. The stride of the sliding window for each dimension of *input*.
- **padding** (*Union[str, Tuple[IntPair, IntPair, IntPair]]*) – padding: *string "SAME"* or *"SAME_LOWER"* or *"SAME_UPPER"* or *"VALID" or Tuple[IntPair, IntPair, IntPair]* indicating the type of padding algorithm to use, or a list indicating the explicit paddings at the start and end of each dimension.
- **data_format** (*str, optional*) – *"NWC" or "NCW"*. Defaults to *"NCW"*.
- **dilations** (*Optional[Union[int, Tuple[int]]], optional*) – An int or list of *ints* that has length *1*. The dilation factor for each dimension of *input*. Defaults to None.
- **groups** (*int, optional*) – int value greater than 0. Defaults to 1.
- **name** (*Optional[str], optional*) – This operator's name. Defaults to None.

Raises

- **ValueError** – strides must be an int or a list.
- **ValueError** – padding must be "SAME" or "SAME_LOWER" or "SAME_UPPER" or "VALID" or Tuple[IntPair, IntPair, IntPair, IntPair].
- **ValueError** – data_format must be "NWC" or "NCW".
- **ValueError** – dilations must be an int or a list.
- **ValueError** – invalid data_format.
- **ValueError** – data_format NWC not support groups > 1
- **ValueError** – invalid data_format.

Returns A *Blob* with the same type as *input* and the same outer batch shape.

Return type oneflow_api.BlobDesc

Note: This api is more flexible, if you're new to OneFlow, it's more recommend to use *oneflow.layers.conv1d*

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


def conv1d(input, filters, kernel_size, strides, padding, name):
    input_shape = input.shape
    weight_initializer = flow.truncated_normal(0.1)
    weight_regularizer = flow.regularizers.l2(0.0005)
    weight_shape = (filters,
                    input_shape[1],
                    kernel_size)

    weight = flow.get_variable(
        name + "-weight",
        shape=weight_shape,
        initializer=weight_initializer,
        regularizer=weight_regularizer,
    )
    return flow.nn.conv1d(input, weight, strides, padding, name=name)


@flow.global_function()
def conv1d_Job(x: tp.Numpy.Placeholder((1, 64, 32))
) -> tp.Numpy:
    conv = conv1d(x,
                  filters=32,
                  kernel_size=3,
                  strides=1,
                  padding='SAME',
                  name="Convlayer")
    return conv


x = np.random.randn(1, 64, 32).astype(np.float32)
out = conv1d_Job(x)
```

(continues on next page)

(continued from previous page)

```
# out.shape (1, 32, 32)
```

oneflow.nn.**conv2d**(*input: oneflow_api.BlobDesc, filters: oneflow_api.BlobDesc, strides: Union[int, Tuple[int, int]], padding: Union[str, Tuple[Tuple[int, int], Tuple[int, int], Tuple[int, int], Tuple[int, int]]], data_format: str = 'NCHW', dilations: Union[int, Tuple[int, int], None] = None, groups: int = 1, name: Optional[str] = None*) → oneflow_api.BlobDesc

2D convolution layer.

Parameters

- **input** (*oneflow_api.BlobDesc*) – A 4D input *Blob*. [batch_num, channel, height, width]
- **filters** (*oneflow_api.BlobDesc*) – A *Blob* with the same type as *input* and has the shape *[out_channels, in_channels//groups, filter_height, filter_width] for NCHW, or [out_channels, filter_height, filter_width, in_channels//groups] for NHWC*
- **strides** (*Union[int, IntPair]*) – An int or list of *ints* that has length *2*. The stride of the sliding window for each dimension of *input*.
- **padding** (*Union[str, Tuple[IntPair, IntPair, IntPair, IntPair]]*) – padding: *string "SAME"* or *"SAME_LOWER"* or *"SAME_UPPER"* or *"VALID" or Tuple[IntPair, IntPair, IntPair, IntPair]* indicating the type of padding algorithm to use, or a list indicating the explicit paddings at the start and end of each dimension.
- **data_format** (*str, optional*) – *"NHWC" or "NCHW"*. Defaults to *"NCHW"*.
- **dilations** (*Optional[Union[int, IntPair]], optional*) – An int or list of *ints* that has length *2*. The dilation factor for each dimension of *input*. Defaults to None.
- **groups** (*int, optional*) – int value greater than 0. Defaults to 1.
- **name** (*Optional[str], optional*) – This operator's name. Defaults to None.

Raises

- **ValueError** – strides must be an int or a list.
- **ValueError** – padding must be "SAME" or "SAME_LOWER" or "SAME_UPPER" or "VALID" or Tuple[IntPair, IntPair, IntPair, IntPair].
- **ValueError** – data_format must be "NHWC" or "NCHW".
- **ValueError** – dilations must be an int or a list.
- **ValueError** – invalid data_format.
- **ValueError** – data_format NHWC not support groups > 1
- **ValueError** – invalid data_format.

Returns A *Blob* with the same type as *input* and the same outer batch shape.

Return type oneflow_api.BlobDesc

Note: This api is more flexible, if you're new to OneFlow, it's more recommend to use *oneflow.layers.conv2d*.

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


def conv2d(input, filters, kernel_size, strides, padding, name):
    input_shape = input.shape
    weight_initializer = flow.truncated_normal(0.1)
    weight_regularizer = flow.regularizers.l2(0.0005)
    weight_shape = (filters,
                    input_shape[1],
                    kernel_size[0],
                    kernel_size[1])

    weight = flow.get_variable(
        name + "-weight",
        shape=weight_shape,
        initializer=weight_initializer,
        regularizer=weight_regularizer,
    )
    return flow.nn.conv2d(input, weight, strides, padding, name=name)


@flow.global_function()
def conv2d_Job(x: tp.Numpy.Placeholder((1, 64, 32, 32))
) -> tp.Numpy:
    conv = conv2d(x,
                  filters=128,
                  kernel_size=[3, 3],
                  strides=2,
                  padding='SAME',
                  name="Convlayer")
    return conv


x = np.random.randn(1, 64, 32, 32).astype(np.float32)
out = conv2d_Job(x)

# out.shape (1, 128, 16, 16)
```

oneflow.nn.**conv2d_transpose**(*value: Optional[oneflow_api.BlobDesc] = None, filter: Optional[oneflow_api.BlobDesc] = None, output_shape: Tuple[int, int, int, int] = None, strides: Union[int, Sequence[int], None] = None, padding: str = 'VALID', data_format: str = 'NCHW', name: Optional[str] = None, input: Optional[oneflow_api.BlobDesc] = None, filters: Optional[oneflow_api.BlobDesc] = None, dilations: Union[int, Sequence[int], None] = None*) → oneflow_api.BlobDesc

2d transposed convolution.

Parameters

- **value** (*Optional[oneflow_api.BlobDesc], optional*) – 4-d *Blob*. Defaults to None.
- **filter** (*Optional[oneflow_api.BlobDesc], optional*) – Filter of transposed convolution, usually a variable. Defaults to None.
- **output_shape** (*Tuple[int, int, int, int]*) – A 1-D *Blob* representing the output shape of the deconvolution op. Defaults to None.

- **strides** (*Optional[Union[int, Sequence[int]]], optional*) – *int* or *int list*. Defaults to None.
- **padding** (*str, optional*) – *'VALID'* or *'SAME'*. Defaults to "VALID".
- **data_format** (*str, optional*) – *'NHWC'* or *'NCHW'*. Defaults to "NCHW".
- **name** (*Optional[str], optional*) – This operator's name(optional). Defaults to None.
- **input** (*Optional[oneflow_api.BlobDesc], optional*) – Alias for value. Defaults to None.
- **filters** (*Optional[oneflow_api.BlobDesc], optional*) – Alias for filter. Defaults to None.
- **dilations** (*Optional[Union[int, Sequence[int]]], optional*) – The dilation factor for each dimension of input. Defaults to None.

Raises

- **ValueError** – shapes of *filter* and *input* must match.
- **ValueError** – dilations must be an int or a list.
- **ValueError** – data_format must be "NHWC" or "NCHW".
- **ValueError** – padding must be "SAME" or "VALID".

Returns A *Blob* with the same type as *value*.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


def deconv2d(input, filters, kernel_size, strides, padding, name):
    input_shape = input.shape
    weight_initializer = flow.truncated_normal(0.1)
    weight_regularizer = flow.regularizers.l2(0.0005)
    weight_shape = (filters,
                    input_shape[1],
                    kernel_size[0],
                    kernel_size[1])

    weight = flow.get_variable(
        name + "-weight",
        shape=weight_shape,
        initializer=weight_initializer,
        regularizer=weight_regularizer,
    )
    return flow.nn.conv2d_transpose(value=input,
                                    output_shape=(1, 32, 64, 64),
                                    filter=weight,
                                    strides=strides,
                                    padding=padding,
                                    name=name)
```

(continues on next page)

(continued from previous page)

```
@flow.global_function()
def deconv2d_Job(x: tp.Numpy.Placeholder((1, 32, 32, 32),)
) -> tp.Numpy:
    deconv = deconv2d(x,
                      filters=32,
                      kernel_size=[3, 3],
                      strides=2,
                      padding='SAME',
                      name="Convlayer")
    return deconv


x = np.random.randn(1, 32, 32, 32).astype(np.float32)
out = deconv2d_Job(x)

# out.shape (1, 32, 64, 64)
```

oneflow.nn.**conv3d**(*input: oneflow_api.BlobDesc, filters: oneflow_api.BlobDesc, strides: Union[int, Sequence[int]], padding: Union[str, Tuple[Tuple[int, int], Tuple[int, int], Tuple[int, int], Tuple[int, int], Tuple[int, int]]], data_format: str = 'NCDHW', dilations: Union[int, Sequence[int], None] = None, groups: int = 1, name: Optional[str] = None*) → oneflow_api.BlobDesc

3D convolution layer.

Parameters

- **input** (*oneflow_api.BlobDesc*) – A 5D input *Blob*. [batch_num, channel, depth, height, width]
- **filters** (*oneflow_api.BlobDesc*) – A *Blob* with the same type as *input* and has the shape *[out_channels, in_channels//groups, filter_depth, filter_height, filter_width] for NCDHW, or [out_channels, filter_depth, filter_height, filter_width, in_channels//groups] for NDHWC*
- **strides** (*Union[int, Sequence[int]]*) – An *int* or *list of ints* that has length *3*. The stride of the sliding window for each dimension of *input*.
- **padding** (*Union[str, Tuple[IntPair, IntPair, IntPair, IntPair, IntPair]]*) – padding: *string "SAME"* or *"SAME_LOWER"* or *"SAME_UPPER"* or *"VALID" or Tuple[IntPair, IntPair, IntPair, IntPair, IntPair]* indicating the type of padding algorithm to use, or a list indicating the explicit paddings at the start and end of each dimension.
- **data_format** (*str, optional*) – *"NDHWC" or "NCDHW"*. Defaults to *"NCDHW"*.
- **dilations** (*Optional[Union[int, Sequence[int]]], optional*) – An int or list of *ints* that has length *3*. The dilation factor for each dimension of *input*. Defaults to None.
- **groups** (*int, optional*) – int value greater than 0. Defaults to 1.
- **name** (*Optional[str], optional*) – This operator's name. Defaults to None.

Raises

- **ValueError** – strides must be an int or a list.
- **ValueError** – padding must be "SAME" or "SAME_LOWER" or "SAME_UPPER" or "VALID" or Tuple[IntPair, IntPair, IntPair, IntPair, IntPair].

- **ValueError** – data_format must be "NDHWC" or "NCDHW".
- **ValueError** – dilations must be an int or a list.
- **ValueError** – invalid data_format.
- **ValueError** – data_format NDHWC not support groups > 1
- **ValueError** – invalid data_format.

Returns A *Blob* with the same type as *input* and the same outer batch shape.

Return type oneflow_api.BlobDesc

Note: This api is more flexible, if you're new to OneFlow, it's more recommend to use *oneflow.layers.conv3d*

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


def conv3d(input, filters, kernel_size, strides, padding, name):
    input_shape = input.shape
    weight_initializer = flow.truncated_normal(0.1)
    weight_regularizer = flow.regularizers.l2(0.0005)
    weight_shape = (filters,
                    input_shape[1],
                    kernel_size[0],
                    kernel_size[1],
                    kernel_size[2])

    weight = flow.get_variable(
        name + "-weight",
        shape=weight_shape,
        initializer=weight_initializer,
        regularizer=weight_regularizer,
    )
    return flow.nn.conv3d(input, weight, strides, padding, name=name)


@flow.global_function()
def conv3d_Job(x: tp.Numpy.Placeholder((1, 64, 10, 16, 16))
) -> tp.Numpy:
    conv = conv3d(x,
                  filters=128,
                  kernel_size=[3, 3, 3],
                  strides=1,
                  padding='SAME',
                  name="Convlayer")
    return conv


x = np.random.randn(1, 64, 10, 16, 16).astype(np.float32)
out = conv3d_Job(x)

# out.shape (1, 128, 10, 16, 16)
```

oneflow.nn.**dropout**(*x: oneflow_api.BlobDesc*, *rate: float*, *noise_shape: Optional[oneflow_api.BlobDesc] = None*, *seed: Optional[int] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

For preventing overfitting, randomly set elements to zero.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A floating point *Blob*.
- **rate** (*float*) – A scalar *Blob* with the same type as x. The probability that each element is dropped.
- **noise_shape** (*Optional[oneflow_api.BlobDesc], optional*) – optional: A 1-D *Blob*, representing the shape for randomly generated keep/drop flags. Defaults to None.Defaults to None.
- **seed** (*Optional[int], optional*) – Optional int value. Defaults to None.
- **name** (*Optional[str], optional*) – This operator's name(optional). Defaults to None.

Returns A *Blob* of the same shape of x.

Return type oneflow_api.BlobDesc

Raises **ValueError** – If rate is not in [0, 1) or if x is not a floating point *Blob*. Rate=1 is not allowed.

For example:

```
import oneflow as flow


def lenet(data, train=False):
    initializer = flow.truncated_normal(0.1)
    conv1 = flow.layers.conv2d(
        data,
        32,
        5,
        padding="SAME",
        activation=flow.nn.relu,
        name="conv1",
        kernel_initializer=initializer,
    )
    pool1 = flow.nn.max_pool2d(
        conv1, ksize=2, strides=2, padding="SAME", name="pool1", data_format="NCHW
↪"
    )
    conv2 = flow.layers.conv2d(
        pool1,
        64,
        5,
        padding="SAME",
        activation=flow.nn.relu,
        name="conv2",
        kernel_initializer=initializer,
    )
    pool2 = flow.nn.max_pool2d(
        conv2, ksize=2, strides=2, padding="SAME", name="pool2", data_format="NCHW
↪"
    )
```

(continues on next page)

(continued from previous page)

```
    reshape = flow.reshape(pool2, [pool2.shape[0], -1])
    hidden = flow.layers.dense(
        reshape,
        512,
        activation=flow.nn.relu,
        kernel_initializer=initializer,
        name="dense1",
    )
    if train:
        hidden = flow.nn.dropout(hidden, rate=0.5, name="dropout")

    return flow.layers.dense(hidden, 10, kernel_initializer=initializer, name=
↪"dense2")
```

oneflow.nn.**elu**(*x: oneflow_api.BlobDesc*, *alpha: float = 1.0*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

The ELU activation.

The formula is:

$$\text{ELU}(x) = \begin{cases} x & \text{if } x0 \\ \alpha * (exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def elu_job(x: tp.Numpy.Placeholder(shape=(3, )))->tp.Numpy:
    return flow.nn.elu(x, alpha=1.0)


x = np.array([-3.5, 1, 3.5]).astype(np.float32)
out = elu_job(x)

# output [-0.9698026  1.         3.5       ]
```

Parameters

- **x** (*oneflow_api.BlobDesc*) – The input Tensor.
- **alpha** (*float, optional*) – The *alpha* value for the ELU formula. Defaults to 1.0.
- **name** (*Optional[str], optional*) – The name for the operator. Defaults to None.

Returns The activated Tensor.

Return type oneflow_api.BlobDesc

oneflow.nn.**fused_scale_tril**(*x: oneflow_api.BlobDesc*, *diagonal: int = 0*, *fill_value: Union[int, float] = 0*, *scale: Union[int, float] = 1*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.nn.**hardsigmoid**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

The Hardsigmoid activation.

The formula is:

$$\text{Hardsigmoid}(x) = \begin{cases} 0 & \text{if } x \leq -3 \\ 1 & \text{if } x \geq +3 \\ \frac{x}{6} + \frac{1}{2} & \text{otherwise} \end{cases}$$

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def hardsigmoid_job(x: tp.Numpy.Placeholder(shape=(3, )))->tp.Numpy:
    out = flow.math.hardsigmoid(x)

    return out


x = np.array([-3.1, 0, 3.3]).astype(np.float32)
out = hardsigmoid_job(x)

# output [0.  0.5 1. ]
```

Parameters

- **x** (*oneflow_api.BlobDesc*) – The input Tensor.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The activated Tensor.

Return type oneflow_api.BlobDesc

oneflow.nn.**hardswish**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

The Hardswish activation.

The formula is:

$$\text{Hardswish}(x) = \begin{cases} 0 & \text{if } x \leq -3 \\ x & \text{if } x \geq +3 \\ x * (x+3)/6 & \text{otherwise} \end{cases}$$

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def hardswish_job(x: tp.Numpy.Placeholder(shape=(3, )))->tp.Numpy:
    return flow.nn.hardswish(x)


x = np.array([-3.5, 1, 3.5]).astype(np.float32)
```

(continues on next page)

(continued from previous page)

```
out = hardswish_job(x)

# output [0.        0.6666667 3.5       ]
```

Parameters

- **x** (*oneflow_api.BlobDesc*) – The input Tensor.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The activated Tensor.

Return type oneflow_api.BlobDesc

oneflow.nn.**hardtanh**(*x: oneflow_api.BlobDesc*, *min_val: float = -1.0*, *max_val: float = 1.0*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

The Hardtanh activation.

The equation is:

$$\text{HardTanh}(x) = \begin{cases} max\_val & \text{if } x > max\_val \\ -min\_val & \text{if } x < min\_val \\ x & \text{otherwise} \end{cases}$$

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def hardtanh_job(x: tp.Numpy.Placeholder(shape=(2, 3)))->tp.Numpy:
    return flow.nn.hardtanh(x, min_val=-1.25, max_val=1.2)


x = np.array([[-1.5, -1.1, 0.6],
              [1.2, 1.3, 1.5]]).astype(np.float32)
out = hardtanh_job(x)

# output [[-1.25 -1.1   0.6 ]
#         [ 1.2   1.2   1.2 ]]
```

Parameters

- **x** (*oneflow_api.BlobDesc*) – The input Tensor.
- **min_val** (*float, optional*) – The minimum value of the linear region range. Defaults to -1.
- **max_val** (*float, optional*) – The maximum value of the linear region range. Defaults to 1.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The activated tensor.

Return type oneflow_api.BlobDesc

oneflow.nn.**layer_norm**(*inputs: oneflow_api.BlobDesc*, *gamma: Optional[oneflow_api.BlobDesc] = None*, *beta: Optional[oneflow_api.BlobDesc] = None*, *begin_norm_axis: int = 1*, *begin_params_axis: int = -1*, *epsilon: float = 1e-05*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Layer Normalization.

Parameters

- **inputs** (*oneflow_api.BlobDesc*) – Input *Blob*.
- **gamma** (*Optional[oneflow_api.BlobDesc]*) –
- **beta** (*Optional[oneflow_api.BlobDesc]*) –
- **begin_norm_axis** (*int, optional*) – An integer specifies which axis to normalize at first. Defaults to 1.
- **begin_params_axis** (*int, optional*) – An integer specifies which axis params at . Defaults to -1.
- **epsilon** (*float, optional*) – A small float is added to avoid division by zero. Defaults to 1e-5.
- **name** (*Optional[str], optional*) – This operator's name. Defaults to None.

Returns A normalized *Blob* with same shape of input.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def layer_norm_Job(x: tp.Numpy.Placeholder((1, 64, 128, 128))
) -> tp.Numpy:
    layer_norm = flow.nn.layer_norm(
        x,
        name="LayerNorm1"
    )
    return layer_norm


x = np.random.randn(1, 64, 128, 128).astype(np.float32)
out = layer_norm_Job(x)

# out.shape (1, 64, 128, 128)
```

oneflow.nn.**leaky_relu**(*x: oneflow_api.BlobDesc*, *alpha: float = 0.2*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Leaky ReLU activation.

$$out = max(x, alpha * x)$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – A *Blob* representing preactivation values.
- **alpha** (*float, optional*) – Slope of the activation function at x < 0 with float type. Default value is 0.2.

- **name** (*Optional[str], optional*) – This operator's name(optional). Defaults to None.

Returns The activation *Blob*.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def leaky_relu_Job(x: tp.Numpy.Placeholder((5, ),)
) -> tp.Numpy:
    leaky_relu = flow.nn.leaky_relu(x, alpha=0.2)

    return leaky_relu


x = np.array([-10, -5, 0, 5, 10]).astype(np.float32)
out = leaky_relu_Job(x)

# out [-2. -1.  0.  5. 10.]
```

oneflow.nn.**logsoftmax**(*logits: oneflow_api.BlobDesc*, *axis: Optional[int] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Computes logsoftmax activations.

For each element, we apply:

$$LogSoftmax(x_i) = Log(\frac{e^i}{\sum_1^j e^j})$$

Parameters

- **logits** (*oneflow_api.BlobDesc*) – A non-empty *Blob*.
- **axis** (*Optional[int], optional*) – The dimension logsoftmax would be performed on. Defaults to None.
- **name** (*Optional[str], optional*) – This operator's name(optional). Defaults to None.

Returns A *Blob* has the same type and shape as logits.

Return type oneflow_api.BlobDesc

Raises **InvalidArgumentError** – if logits is empty or axis is beyond the last dimension of logits.

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def logsoftmax_Job(x: tp.Numpy.Placeholder((1, 5))
```

(continues on next page)

(continued from previous page)

```
) -> tp.Numpy:
    logsoftmax_out = flow.nn.logsoftmax(x, axis=1)
    return logsoftmax_out


x = np.array([[1, 2, 1, 5, 4]]).astype(np.float32)
out = logsoftmax_Job(x)

# out [[-4.374523  -3.3745232 -4.374523  -0.3745232 -1.374523 ]]
```

oneflow.nn.**max_pool1d**(*input: oneflow_api.BlobDesc, ksize: Union[int, Sequence[int]], strides: Union[int, Sequence[int]], padding: Union[str, Sequence[Sequence[int]]], data_format: str = 'NWC', name: Optional[str] = None*) → oneflow_api.BlobDesc

Performs the 1d-max pooling on the input.

Parameters

- **input** (*oneflow_api.BlobDesc*) – A 3-D *Blob* of the format specified by data_format.
- **ksize** (*Union[int, Sequence[int]]*) – An int or list of ints that has length 1 or 3. The size of the window for each dimension of the input *Blob*.
- **strides** (*Union[int, Sequence[int]]*) – An int or list of ints that has length 1 or 3. The stride of the sliding window for each dimension of the input *Blob*.
- **padding** (*str*) – '*VALID*' or '*SAME*'. The padding algorithm.
- **data_format** (*str, optional*) – An optional string from: '*NWC*', '*NCW*'. Defaults to '*NWC*'.
- **name** (*Optional[str], optional*) – This operator's name(optional).Defaults to None.

Raises **NotImplementedError** – TODO: fix cuDNN bugs in pooling_1d

Returns A *Blob* of format specified by data_format. The max pooled output *Blob*.

Return type oneflow_api.BlobDesc

oneflow.nn.**max_pool2d**(*input: oneflow_api.BlobDesc, ksize: Union[int, Tuple[int, int]], strides: Union[int, Tuple[int, int]], padding: Union[str, Tuple[Tuple[int, int], Tuple[int, int], Tuple[int, int], Tuple[int, int]]], data_format: str = 'NCHW', ceil_mode: bool = False, name: Optional[str] = None*) → oneflow_api.BlobDesc

Performs the 2d-max pooling on the input *Blob*.

Parameters

- **input** (*oneflow_api.BlobDesc*) – A 4-D *Blob* of the format specified by data_format.
- **ksize** (*Union[int, IntPair]*) – An int or list of ints that has length 1, 2. The size of the window for each dimension of the input *Blob*.
- **strides** (*Union[int, IntPair]*) – An int or list of ints that has length 1, 2. The stride of the sliding window for each dimension of the input *Blob*.
- **padding** (*str*) – '*VALID*' or '*SAME*' or '*SAME_LOWER*' or '*SAME_UPPER*' or Tuple[IntPair, IntPair, IntPair, IntPair]'. The padding algorithm.

- **data_format** (*str, optional*) – 'NHWC', 'NCHW' or 'NCHW_VECT_C'. Defaults to "NCHW".
- **name** (*Optional[str], optional*) – This operator's name(optional). Defaults to None.

Returns A *Blob* of format specified by data_format. The max pooled output *Blob*.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def maxpool2d_Job(x: tp.Numpy.Placeholder((1, 32, 128, 128))
) -> tp.Numpy:
    pool_out = flow.nn.max_pool2d(
        input=x,
        ksize=3,
        strides=2,
        padding='SAME',
        data_format='NCHW'
    )

    return pool_out


x = np.random.randn(1, 32, 128, 128).astype(np.float32)
out = maxpool2d_Job(x)

# out.shape (1, 32, 64, 64)
```

oneflow.nn.**max_pool3d**(*input: oneflow_api.BlobDesc, ksize: Union[int, Sequence[int]], strides: Union[int, Sequence[int]], padding: Union[str, Sequence[Sequence[int]]], data_format: str = 'NCDHW', ceil_mode: bool = False, name: Optional[str] = None*) → oneflow_api.BlobDesc

Performs the 3d-max pooling on the input.

Parameters

- **input** (*oneflow_api.BlobDesc*) – A 5-D *Blob* of the format specified by data_format.
- **ksize** (*Union[int, Sequence[int]]*) – An int or list of ints that has length 1, 3 or 5. The size of the window for each dimension of the input *Blob*.
- **strides** (*Union[int, Sequence[int]]*) – An int or list of ints that has length 1, 3 or 5. The stride of the sliding window for each dimension of the input *Blob*.
- **padding** (*str*) – 'VALID' or 'SAME' or 'SAME_LOWER' or 'SAME_UPPER' or 'Sequence[Sequence[int]]'.
- **data_format** (*str, optional*) – "NDHWC" or "NCDHW". Defaults to "NCDHW".
- **name** (*Optional[str], optional*) – This operator's name(optional).

Returns A *Blob* of format specified by data_format. The max pooled output *Blob*.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def maxpool3d_Job(x: tp.Numpy.Placeholder((1, 32, 10, 128, 128))
) -> tp.Numpy:
    pool_out = flow.nn.max_pool3d(
        input=x,
        ksize=3,
        strides=2,
        padding='SAME',
        data_format='NCDHW'
    )

    return pool_out


x = np.random.randn(1, 32, 10, 128, 128).astype(np.float32)
out = maxpool3d_Job(x)

# out.shape (1, 32, 5, 64, 64)
```

oneflow.nn.**mish**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

The Mish activation function.

The equation is:

$$out = x * tanh(ln(1 + e^x))$$

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def mish_job(x: tp.Numpy.Placeholder(shape=(5, )))->tp.Numpy:
    return flow.nn.mish(x)


x = np.array([-0.5, 0, 0.5, 1.0, 1.5]).astype(np.float32)
out = mish_job(x)
```

Parameters

- **x** (*oneflow_api.BlobDesc*) – The input Blob.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

oneflow.nn.**moments**(*x: oneflow_api.BlobDesc, axes: List[int], keepdims: Optional[bool] = False, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator computes the mean and variance value of input Blob.

Parameters

- **x** (*oneflow_api.BlobDesc*) – A Blob
- **axes** (*List*) – Array of ints. Axes along which to compute the mean and variance
- **keepdims** (*bool, optional*) – Whether to keep the same dimensanality as the input x. Defaults to False.
- **name** (*str, optional*) – The operator's name. Defaults to None.

Returns Two Blobs, mean and variance.

Return type remote_blob

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp
from typing import Tuple


@flow.global_function()
def moments_Job(x: tp.Numpy.Placeholder((5,))
) -> Tuple[tp.Numpy, tp.Numpy]:
    return flow.nn.moments(x, axes=[0])


x = np.array([1, 2, 3, 4, 5]).astype(np.float32)
mean, variance = moments_Job(x)

# mean: [3.]
# variance: [2.]
```

oneflow.nn.**random_mask_like**(*like: oneflow_api.BlobDesc, rate: float, seed: Optional[int] = None, noise_shape: Optional[Sequence] = None, name: Optional[str] = None*) → oneflow_api.BlobDesc

Random mask *Blob* with same shape as '*like*'.

Parameters

- **like** (*oneflow_api.BlobDesc*) – A *Blob*.
- **rate** (*float*) – A float value for the probability that each element is dropped.
- **seed** (*Optional[int], optional*) – Optional, int value. Defaults to None.
- **noise_shape** (*Optional[Sequence], optional*) – Optional, A 1-D *Blob*, representing the shape for randomly generated keep/drop flags. Defaults to None.
- **name** (*Optional[str], optional*) – This operator's name(optional). Defaults to None.

Returns A random mask *Blob* of the same shape of *like*.

Return type oneflow_api.BlobDesc

Raises **ValueError** – If rate is not in [0, 1). Rate=1 is not allowed.

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def random_mask_like_Job(like: tp.Numpy.Placeholder((5, 5), dtype=flow.float32)
) -> tp.Numpy:

    return flow.nn.random_mask_like(like=like,
                                    rate=0.5)


like = np.ones(shape=(5, 5)).astype(np.float32)
random_mask = random_mask_like_Job(like)

# out [[0 0 0 0 0]
#      [1 1 1 0 0]
#      [1 0 1 1 0]
#      [0 0 0 0 1]
#      [1 0 1 1 1]]
```

oneflow.nn.**relu**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Relu activation

The equation is:

$$out = max(X, 0)$$

Parameters

- **x** (*oneflow_api.BlobDesc*) – Input Blob
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns An activated Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def reluJob(x: tp.Numpy.Placeholder((3, ))
)->tp.Numpy:
    return flow.math.relu(x)

x = np.array([-1, 0, 5]).astype(np.float32)
out = reluJob(x)

# out [0., 0., 5.]
```

oneflow.nn.**relu6**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Relu6 activation, it clips the value around (0, 6).

The equation is:

$$\text{Relu6}(x) = \begin{cases} 6 & \text{if } x > 6 \\ 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def relu6_job(x: tp.Numpy.Placeholder(shape=(2, 3)))->tp.Numpy:
    return flow.nn.relu6(x)

x = np.array([[-1, -0.5, 0.0],
              [0.5, 6.0, 7]]).astype(np.float32)

out = relu6_job(x)

# output [[0.  0.  0. ]
#         [0.5 6.  6. ]]
```

Parameters

- **x** (*oneflow_api.BlobDesc*) – The input Tensor.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The activated Tensor.

Return type oneflow_api.BlobDesc

oneflow.nn.**sigmoid_cross_entropy_with_logits**(*labels: oneflow_api.BlobDesc*, *logits: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Computes sigmoid cross entropy given logits.

Parameters

- **labels** (*oneflow_api.BlobDesc*) – A *Blob* of the same type and shape as logits.
- **logits** (*oneflow_api.BlobDesc*) – A *Blob* of type float.
- **name** (*Optional[str], optional*) – This operator's name(optional). Defaults to None.

Returns A *Blob* of the same shape as logits with the componentwise logistic losses.

Return type oneflow_api.BlobDesc

Raises **ValueError** – If logits and labels do not have the same shape.

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

```

(continues on next page)

(continued from previous page)

```
@flow.global_function()
def sigmoid_cross_entropy_Job(input: tp.Numpy.Placeholder((3, 2), dtype=flow.
↪float32),
                              labels: tp.Numpy.Placeholder((3, 2), dtype=flow.
↪float32)
) -> tp.Numpy:
    loss = flow.nn.sigmoid_cross_entropy_with_logits(labels=labels,
                                                     logits=input)
    return loss


x = np.array([[4, 1],
              [3, 2],
              [1, 5]]).astype(np.float32)
labels = np.array([[0.7, 0.3],
                   [0.4, 0.6],
                   [0.2, 0.8]]).astype(np.float32)
loss = sigmoid_cross_entropy_Job(x, labels)

# out [[0.612735   0.90472794]
#      [0.89778364 0.6990613 ]
#      [0.97783387 0.51372755]]
```

oneflow.nn.**softmax**(*logits: oneflow_api.BlobDesc*, *axis: Optional[int] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Computes softmax activations.

For each element, we apply:

$$S_i = \frac{e^i}{\sum_1^j e^j}$$

Parameters

- **logits** (*oneflow_api.BlobDesc*) – A non-empty *Blob*.
- **axis** (*Optional[int], optional*) – The dimension softmax would be performed on. Defaults to None.
- **name** (*Optional[str], optional*) – This operator's name(optional). Defaults to None.

Returns A *Blob* has the same type and shape as logits.

Return type oneflow_api.BlobDesc

Raises **InvalidArgumentError** – if logits is empty or axis is beyond the last dimension of logits.

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def softmax_Job(x: tp.Numpy.Placeholder((1, 5))
```

(continues on next page)

(continued from previous page)

```
) -> tp.Numpy:
    softmax_out = flow.nn.softmax(x, axis=1)

    return softmax_out


x = np.array([[1, 2, 1, 5, 4]]).astype(np.float32)
out = softmax_Job(x)

# out [[0.01259415 0.03423444 0.01259415 0.68761706 0.2529602 ]]
```

oneflow.nn.**softmax_cross_entropy_with_logits**(*labels: oneflow_api.BlobDesc*, *logits: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Computes softmax cross entropy between logits and labels.

Parameters

- **labels** (*oneflow_api.BlobDesc*) – Each vector along the class dimension should hold a valid probability distribution.
- **logits** (*oneflow_api.BlobDesc*) – Per-label activations, typically a linear output. logits has same shape and dtype as labels.
- **name** (*Optional[str], optional*) – This operator's name(optional). Defaults to None.

Returns A *Blob* that contains the softmax cross entropy loss. Its type is the same as logits and its shape is the same as labels except that it does not have the last dimension of labels.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def softmax_cross_entropy_Job(input: tp.Numpy.Placeholder((3, 3), dtype=flow.
↪float32),
                              labels: tp.Numpy.Placeholder((3, 3), dtype=flow.
↪float32)
) -> tp.Numpy:
    loss = flow.nn.softmax_cross_entropy_with_logits(labels=labels,
                                                     logits=input)
    return loss


x = np.array([[4, 1, 2],
              [3, 2, 3],
              [1, 5, 10]]).astype(np.float32)
labels = np.array([[0.9, 0.05, 0.05],
                   [0.3, 0.4, 0.3],
                   [0.8, 0.1, 0.1]]).astype(np.float32)
loss = softmax_cross_entropy_Job(x, labels)

# out [0.73441553 1.1240788  1.4488925 ]
```

oneflow.nn.**softmax_grad**(*y: oneflow_api.BlobDesc*, *dy: oneflow_api.BlobDesc*, *axis: Optional[int] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Computes gradient of softmax activations.

Parameters

- **y** (*oneflow_api.BlobDesc*) – A *Blob* representing the softmax of x.
- **dy** (*oneflow_api.BlobDesc*) – gradient of y.
- **axis** (*Optional[int], optional*) – The dimension softmax would be performed on. Defaults to None.
- **name** (*Optional[str], optional*) – This operator's name(optional).

Returns A *Blob* representing the gradient of x.

Return type oneflow_api.BlobDesc

oneflow.nn.**sparse_cross_entropy**(*labels: oneflow_api.BlobDesc*, *prediction: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Computes sparse cross entropy

Parameters

- **labels** (*oneflow_api.BlobDesc*) – A *Blob* of shape [d_0, d_1, ..., d_{r-1}] (where r is rank of labels and result). Each entry in labels must be an index in [0, num_classes).
- **prediction** (*oneflow_api.BlobDesc*) – A *Blob* with the rank that is equal to the rank of the labels plus one.
- **name** (*Optional[str], optional*) – This operator's name(optional). Defaults to None.

Returns A *Blob* of the same shape as labels.

Return type oneflow_api.BlobDesc

Note: The labels data type should be *oneflow.int32*.

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def sparse_cross_entropy_Job(input: tp.Numpy.Placeholder((5, 2), dtype=flow.
↪float32),
                             labels: tp.Numpy.Placeholder((5,), dtype=flow.int32)
) -> tp.Numpy:
    loss = flow.nn.sparse_cross_entropy(labels=labels,
                                        prediction=input)
    return loss


x = np.array([[0.3, 0.7],
              [0.4, 0.6],
              [0.5, 0.5],
```

(continues on next page)

(continued from previous page)

```
            [0.1, 0.9],
            [0.2, 0.8]]).astype(np.float32)
labels = np.array([0, 1, 1, 0, 1]).astype(np.int32)
loss = sparse_cross_entropy_Job(x, labels)

# out [1.2039728  0.5108256  0.6931472  2.3025851  0.22314353]
```

oneflow.nn.**sparse_softmax_cross_entropy_with_logits**(*labels: oneflow_api.BlobDesc*, *logits: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Computes sparse softmax cross entropy between logits and labels.

Parameters

- **labels** (*oneflow_api.BlobDesc*) – *Blob* of shape [d_0, d_1, ..., d_{r-1}] (where r is rank of labels and result). Each entry in labels must be an index in [0, num_classes).
- **logits** (*oneflow_api.BlobDesc*) – Unscaled log probabilities of shape [d_0, d_1, ..., d_{r-1},num_classes].
- **name** (*Optional[str], optional*) – This operator's name(optional). Defaults to None.

Raises **ValueError** – If logits are scalars (need to have rank >= 1) or if the rank of the labels is not equal to the rank of the logits minus one.

Returns A *Blob* of the same shape as labels and of the same type as logits with the softmax cross entropy loss.

Return type oneflow_api.BlobDesc

Note: The labels data type should be *oneflow.int32*.

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def sparse_softmax_cross_entropy_Job(input: tp.Numpy.Placeholder((3, 3),␣
↪dtype=flow.float32),
                                     labels: tp.Numpy.Placeholder((3, ),␣
↪dtype=flow.int32)
) -> tp.Numpy:
    loss = flow.nn.sparse_softmax_cross_entropy_with_logits(labels=labels,
                                                            logits=input)
    return loss


x = np.array([[4, 1, 2],
            [3, 2, 3],
            [1, 5, 10]]).astype(np.float32)
labels = np.array([0, 1, 2]).astype(np.int32)
loss = sparse_softmax_cross_entropy_Job(x, labels)
```

(continues on next page)

(continued from previous page)

```
# out [0.65784633 1.2842525  0.5557927 ]
```

oneflow.nn.**swish**(*x: oneflow_api.BlobDesc*, *beta: float = 1.0*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

The Swish activation function.

The equation is:

$$out = x * sigmoid(\beta * x)$$

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np


@flow.global_function()
def swish_job(x: tp.Numpy.Placeholder(shape=(5, )))->tp.Numpy:
    return flow.nn.swish(x)
x = np.array([-0.5, 0, 0.5, 1, 1.5]).astype(np.float32)


out = swish_job(x)
# output [-0.18877034  0.          0.31122968  0.7310586   1.2263618 ]
```

Parameters

- **x** (*oneflow_api.BlobDesc*) – The input Blob.
- **beta** (*float, optional*) – The smooth factor. Defaults to 1.0.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

oneflow.nn.**torch_conv2d_transpose**(*value=None*, *filter=None*, *output_padding=None*, *strides=None*, *padding_needed=None*, *data_format='NCHW'*, *name=None*, *input=None*, *filters=None*, *dilations=None*)

oneflow.nn.**tril**(*x: oneflow_api.BlobDesc*, *diagonal: int = 0*, *fill_value: Union[int, float] = 0*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

Compute lower triangle of an matrix.

Parameters

- **x** (*oneflow_api.BlobDesc*) – Input Blob.
- **diagonal** (*int*) – Diagonal offset, when diagonal > 0, diagonal offset up, otherwise, offset downward.
- **fill_value** (*Union[int, float]*) – The value filled into the upper triangle.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Attention: The dimension of x must greater or equal to 2.

Returns The lower triangle blob of input.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp
@flow.global_function()
def tril_Job(x: tp.Numpy.Placeholder((4, 4))
)->tp.Numpy:
    return flow.math.tril(x, 0)
x = np.array([[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]],
              dtype=np.float32)
out = tril_Job(x).get()

# output [[1, 0, 0, 0],
          [1, 2, 0, 0],
          [1, 2, 3, 0],
          [1, 2, 3, 4]]
```

CHAPTER

NINE

ONEFLOW.LAYERS

9.1 Operators with variables

oneflow.layers.**batch_normalization**(*inputs: oneflow_api.BlobDesc*, *axis: int = -1*, *momentum: float = 0.99*, *epsilon: float = 0.001*, *center: bool = True*, *scale: bool = True*, *beta_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None*, *gamma_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None*, *beta_regularizer: Optional[oneflow.core.job.regularizer_conf_pb2.RegularizerConf] = None*, *gamma_regularizer: Optional[oneflow.core.job.regularizer_conf_pb2.RegularizerConf] = None*, *moving_mean_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None*, *moving_variance_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None*, *trainable: bool = True*, *training: bool = True*, *name: str = 'BatchNorm'*) → oneflow_api.BlobDesc

The BatchNormalization Layer.

This layer can be used in conv or dense layer.

The input data will be normalized by the mean and variance of the current batch data

Parameters

- **inputs** (*oneflow_api.BlobDesc*) – Input *Blob*.
- **axis** (*int, optional*) – An int specifies the axis that should be normalized . Default is -1, which normalizes the last axis.
- **momentum** (*float, optional*) – A float specifies the momentum for the moving average. Defaults to 0.99.
- **epsilon** (*float, optional*) – A small float added to avoid division by zero. Defaults to 0.001.
- **center** (*bool, optional*) – A boolean specifies whether to add offset to normalized *Blob*. Defaults to True.
- **scale** (*bool, optional*) – A boolean specifies whether to multiply normalized *Blob* by gamma. Defaults to True.
- **beta_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for beta. Defaults to None.

- **gamma_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for gamma. Defaults to None.
- **beta_regularizer** (*Optional[regularizer_conf_util.RegularizerConf], optional*) – Regularizer for beta. Defaults to None.
- **gamma_regularizer** (*Optional[regularizer_conf_util.RegularizerConf], optional*) – Regularizer for gamma. Defaults to None.
- **moving_mean_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for moving mean. Defaults to None.
- **moving_variance_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for moving variance. Defaults to None.
- **trainable** (*bool, optional*) – A boolean specifies whether to train variables. Defaults to True.
- **training** (*bool, optional*) – A boolean specifies whether now is training the model. Defaults to True.
- **name** (*Optional[str], optional*) – This layer's name. Defaults to None.

Returns A *Blob* with same shape of input.

Return type oneflow_api.BlobDesc

Raises **ValueError** – If axis is out of dimension of input.

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def batch_norm_Job(x: tp.Numpy.Placeholder((1, 64, 128, 128))
) -> tp.Numpy:
    initializer = flow.truncated_normal(0.1)
    conv2d = flow.layers.conv2d(
        x,
        filters=128,
        kernel_size=3,
        strides=2,
        padding='SAME',
        kernel_initializer=initializer,
        name="Conv2d"
    )
    batch_norm = flow.layers.batch_normalization(
        conv2d,
        axis=1
    )
    return batch_norm


x = np.random.randn(1, 64, 128, 128).astype(np.float32)
out = batch_norm_Job(x)

# out.shape (1, 128, 64, 64)
```

oneflow.layers.**batch_normalization_add_relu**(*inputs: oneflow_api.BlobDesc*, *addend: Optional[oneflow_api.BlobDesc] = None*, *axis: int = -1*, *momentum: float = 0.99*, *epsilon: float = 0.001*, *center: bool = True*, *scale: bool = True*, *beta_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None*, *gamma_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None*, *beta_regularizer: Optional[oneflow.core.job.regularizer_conf_pb2.RegularizerConf] = None*, *gamma_regularizer: Optional[oneflow.core.job.regularizer_conf_pb2.RegularizerConf] = None*, *moving_mean_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None*, *moving_variance_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None*, *trainable: bool = True*, *training: bool = True*, *name: str = 'BatchNorm'*) → oneflow_api.BlobDesc

Fused flow.layers.batch_normalization + flow.math.add + flow.math.relu

Parameters

- **inputs** (*oneflow_api.BlobDesc*) – Input *Blob*.
- **addend** (*oneflow_api.BlobDesc*) – *Blob* add to batch_normalization output.
- **axis** (*int, optional*) – An int specifies the axis that should be normalized . Default is -1, which normalizes the last axis.
- **momentum** (*float, optional*) – A float specifies the momentum for the moving average. Defaults to 0.99.
- **epsilon** (*float, optional*) – A small float added to avoid division by zero. Defaults to 0.001.
- **center** (*bool, optional*) – A boolean specifies whether to add offset to normalized *Blob*. Defaults to True.
- **scale** (*bool, optional*) – A boolean specifies whether to multiply normalized *Blob* by gamma. Defaults to True.
- **beta_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for beta. Defaults to None.
- **gamma_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for gamma. Defaults to None.
- **beta_regularizer** (*Optional[regularizer_conf_util.RegularizerConf], optional*) – Regularizer for beta. Defaults to None.
- **gamma_regularizer** (*Optional[regularizer_conf_util.RegularizerConf], optional*) – Regularizer for gamma. Defaults to None.
- **moving_mean_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for moving mean. Defaults to None.
- **moving_variance_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for moving variance. Defaults to None.

- **trainable** (*bool, optional*) – A boolean specifies whether to train variables. Defaults to True.
- **training** (*bool, optional*) – A boolean specifies whether now is training the model. Defaults to True.
- **name** (*Optional[str], optional*) – This layer's name. Defaults to None.

Returns A *Blob* with same shape of input.

Return type oneflow_api.BlobDesc

Raises **ValueError** – If axis is out of dimension of input.

oneflow.layers.**batch_normalization_relu**(*inputs: oneflow_api.BlobDesc*, *axis: int = -1*, *momentum: float = 0.99*, *epsilon: float = 0.001*, *center: bool = True*, *scale: bool = True*, *beta_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None*, *gamma_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None*, *beta_regularizer: Optional[oneflow.core.job.regularizer_conf_pb2.RegularizerConf] = None*, *gamma_regularizer: Optional[oneflow.core.job.regularizer_conf_pb2.RegularizerConf] = None*, *moving_mean_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None*, *moving_variance_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None*, *trainable: bool = True*, *training: bool = True*, *name: str = 'BatchNorm'*) → oneflow_api.BlobDesc

Fused flow.layers.batch_normalization + flow.math.relu

Parameters

- **inputs** (*oneflow_api.BlobDesc*) – Input *Blob*.
- **axis** (*int, optional*) – An int specifies the axis that should be normalized . Default is -1, which normalizes the last axis.
- **momentum** (*float, optional*) – A float specifies the momentum for the moving average. Defaults to 0.99.
- **epsilon** (*float, optional*) – A small float added to avoid division by zero. Defaults to 0.001.
- **center** (*bool, optional*) – A boolean specifies whether to add offset to normalized *Blob*. Defaults to True.
- **scale** (*bool, optional*) – A boolean specifies whether to multiply normalized *Blob* by gamma. Defaults to True.
- **beta_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for beta. Defaults to None.
- **gamma_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for gamma. Defaults to None.
- **beta_regularizer** (*Optional[regularizer_conf_util.RegularizerConf], optional*) – Regularizer for beta. Defaults to None.

- **gamma_regularizer** (*Optional[regularizer_conf_util.RegularizerConf], optional*) – Regularizer for gamma. Defaults to None.
- **moving_mean_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for moving mean. Defaults to None.
- **moving_variance_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for moving variance. Defaults to None.
- **trainable** (*bool, optional*) – A boolean specifies whether to train variables. Defaults to True.
- **training** (*bool, optional*) – A boolean specifies whether now is training the model. Defaults to True.
- **name** (*Optional[str], optional*) – This layer's name. Defaults to None.

Returns A *Blob* with same shape of input.

Return type oneflow_api.BlobDesc

Raises **ValueError** – If axis is out of dimension of input.

oneflow.layers.**categorical_ordinal_encoder**(*input_tensor: oneflow_api.BlobDesc*, *capacity: int*, *hash_precomputed: bool = True*, *name: str = 'CategoricalOrdinalEncoder'*) → oneflow_api.BlobDesc

This operator uses *oneflow.categorical_ordinal_encode* to encapsulate a categorical_ordinal_encoder. More details please refer to *oneflow.categorical_ordinal_encode*

Parameters

- **input_tensor** (*oneflow_api.BlobDesc*) – The input Blob.
- **capacity** (*int*) – The capacity of hash table.
- **hash_precomputed** (*bool, optional*) – We currently only support the 'True' mode. The internal hash value will no longer be computed. Defaults to True.
- **name** (*str, optional*) – The name for the operation. Defaults to "CategoricalOrdinalEncoder".

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp

@flow.global_function()
def categorical_ordinal_encoder_Job(x: tp.Numpy.Placeholder((3, 3), dtype=flow.
↪int32)
) -> tp.Numpy:
    return flow.layers.categorical_ordinal_encoder(x, 16)

x = np.array([[7, 0, 2],
              [1, 7, 2],
              [0, 1, 7]]).astype(np.int32)
```

(continues on next page)

(continued from previous page)

```
out = categorical_ordinal_encoder_Job(x)

# out [[1 0 2]
#      [3 1 2]
#      [0 3 1]]
```

oneflow.layers.**conv1d**(*inputs: oneflow_api.BlobDesc, filters: int, kernel_size: Union[int, Tuple[int]] = 1, strides: Union[int, Tuple[int]] = 1, padding: Union[str, Tuple[Tuple[int, int], Tuple[int, int], Tuple[int, int]]] = 'VALID', data_format: str = 'NCW', dilation_rate: Union[int, Tuple[int], None] = None, groups: int = 1, activation: Optional[Callable[[oneflow_api.BlobDesc, str], oneflow_api.BlobDesc]] = None, use_bias: bool = True, kernel_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None, bias_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None, kernel_regularizer: Optional[oneflow.core.job.regularizer_conf_pb2.RegularizerConf] = None, bias_regularizer: Optional[oneflow.core.job.regularizer_conf_pb2.RegularizerConf] = None, trainable: bool = True, name: str = 'Conv1d', weight_name: Optional[str] = None, bias_name: Optional[str] = None*) → oneflow_api.BlobDesc

1D convolution layer.

This layer computes a 1-D convolution with 3D input Blob and filters.

Parameters

- **inputs** (*oneflow_api.BlobDesc*) – A 3D input *Blob*.
- **filters** (*int*) – An integer specifies the dimensionality of the output space.
- **kernel_size** (*Union[int, List[int], Tuple[int]], optional*) – An integer or tuple/list specifies the height and width of the convolution window. When it is an integer, a square window is applied to the input. Defaults to 1.
- **strides** (*Union[int, List[int], Tuple[int]], optional*) – An integer or tuple/list specifies the strides of the convolution window along the height and width. When it is an integer, the same value for the all spatial dimesions is applied. Defaults to 1.
- **padding** (*str, Tuple[IntPair, IntPair, IntPair], optional*) – padding: *string "SAME"* or *"SAME_LOWER"* or *"SAME_UPPER"* or *"VALID" or Tuple[IntPair, IntPair, IntPair]* indicating the type of padding algorithm to use, or a list indicating the explicit paddings at the start and end of each dimension. Defaults to "VALID".
- **data_format** (*str, optional*) – A string specifies the format of the input *Blob*, one of "NCW" or "NWC" (default: "NCW"). "NCW" cooresponds to channels_first, i.e. the input *Blob* with shape (batch_size, channels, width). "NWC" cooresponds to channels_last, i.e. the input *Blob* with shape (batch_size, channels, width). Defaults to "NCW".
- **dilation_rate** (*Optional[Union[int, Tuple[int]]], optional*) – An integer or tuple/list specifies the dilation rate for the dilated convolution. When it is an integer, the same dilation rate is applied for the all dimensions. Defaults to 1.
- **groups** (*int, optional*) – A positive integer specifies number of groups for the Group conv. Defaults to 1.
- **activation** (*Optional[Callable[[oneflow_api.BlobDesc, str], oneflow_api.BlobDesc]], optional*) – Activation function. Defaults to None.

- **use_bias** (*bool, optional*) – A boolean specifies whether to use a bias vector. Defaults to True.
- **kernel_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for the kernel weights matrix. Defaults to None.
- **bias_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for the bias vector. Defaults to None.
- **kernel_regularizer** (*Optional[regularizer_conf_util.RegularizerConf], optional*) – Regularizer for the kernel weights matrix. Defaults to None.
- **bias_regularizer** (*Optional[regularizer_conf_util.RegularizerConf], optional*) – Regularizer for the bias vector . Defaults to None.
- **trainable** (*bool, optional*) – A boolean specifies whether to train variables. Defaults to True.
- **name** (*Optional[str], optional*) – This layer's name. Defaults to None.

Raises

- **ValueError** – If the type of kernel_size is not one of integer, list, tuple.
- **ValueError** – The number of groups must be positive and number of filters must be divisible by it.
- **ValueError** – If data_format is not one of 'NCW', 'NWC'.
- **ValueError** – If number of input channels is not divisible by number of groups or less than number of groups.
- **ValueError** – Number of group must be one when data_format is 'NWC'.

Returns A 3D *Blob* with the shape of (batch_size, filters, new_width).

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def conv1d_Job(x: tp.Numpy.Placeholder((1, 64, 32))
) -> tp.Numpy:
    initializer = flow.truncated_normal(0.1)
    conv1d = flow.layers.conv1d(
        x,
        filters=128,
        kernel_size=3,
        strides=1,
        padding='SAME',
        kernel_initializer=initializer,
        name="Conv1d"
    )
```

(continues on next page)

(continued from previous page)

```
    return conv1d


x = np.random.randn(1, 64, 32).astype(np.float32)
out = conv1d_Job(x)

# out.shape (1, 128, 32)
```

oneflow.layers.**conv2d**(*inputs: oneflow_api.BlobDesc, filters: int, kernel_size: Union[int, Tuple[int, int]] = 1, strides: Union[int, Tuple[int, int]] = 1, padding: Union[str, Tuple[Tuple[int, int], Tuple[int, int], Tuple[int, int], Tuple[int, int]]] = 'VALID', data_format: str = 'NCHW', dilation_rate: Union[int, Tuple[int, int], None] = None, groups: int = 1, activation: Optional[Callable[[oneflow_api.BlobDesc, str], oneflow_api.BlobDesc]] = None, use_bias: bool = True, kernel_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None, bias_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None, kernel_regularizer: Optional[oneflow.core.job.regularizer_conf_pb2.RegularizerConf] = None, bias_regularizer: Optional[oneflow.core.job.regularizer_conf_pb2.RegularizerConf] = None, trainable: bool = True, name: str = 'Conv2d', weight_name: Optional[str] = None, bias_name: Optional[str] = None*) → oneflow_api.BlobDesc

2D convolution layer.

This layer computes a 2D convolution with 4D input Blob and filters.

Parameters

- **inputs** (*oneflow_api.BlobDesc*) – A 4D input *Blob*.
- **filters** (*int*) – An integer specifies the dimensionality of the output space.
- **kernel_size** (*Union[int, List[int], Tuple[int]], optional*) – An integer or tuple/list specifies the height and width of the convolution window. When it is an integer, a square window is applied to the input. Defaults to 1.
- **strides** (*Union[int, List[int], Tuple[int]], optional*) – An integer or tuple/list specifies the strides of the convolution window along the height and width. When it is an integer, the same value for the all spatial dimesions is applied. Defaults to 1.
- **padding** (*str, Tuple[IntPair, IntPair, IntPair, IntPair], optional*) – padding: *string "SAME"* or *"SAME_LOWER"* or *"SAME_UPPER"* or *"VALID" or Tuple[IntPair, IntPair, IntPair]* indicating the type of padding algorithm to use, or a list indicating the explicit paddings at the start and end of each dimension. Defaults to "VALID".
- **data_format** (*str, optional*) – A string specifies the format of the input *Blob*, one of "NCHW" or "NHWC" (default: "NCHW"). "NCHW" cooresponds to channels_first, i.e. the input *Blob* with shape (batch_size, channels, height, width). "NHWC" cooresponds to channels_last, i.e. the input *Blob* with shape (batch_size, height, width, channels). Defaults to "NCHW".
- **dilation_rate** (*int, optional*) – An integer or tuple/list specifies the dilation rate for the dilated convolution. When it is an integer, the same dilation rate is applied for the all dimensions. Defaults to 1.
- **groups** (*int, optional*) – A positive integer specifies number of groups for the Group conv. Defaults to 1.

- **activation** (*Optional[Callable[[oneflow_api.BlobDesc, str], oneflow_api.BlobDesc]], optional*) – Activation function. Defaults to None.
- **use_bias** (*bool, optional*) – A boolean specifies whether to use a bias vector. Defaults to True.
- **kernel_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for the kernel weights matrix. Defaults to None.
- **bias_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for the bias vector. Defaults to None.
- **kernel_regularizer** (*Optional[regularizer_conf_util.RegularizerConf], optional*) – Regularizer for the kernel weights matrix. Defaults to None.
- **bias_regularizer** (*Optional[regularizer_conf_util.RegularizerConf], optional*) – Regularizer for the bias vector . Defaults to None.
- **trainable** (*bool, optional*) – A boolean specifies whether to train variables. Defaults to True.
- **name** (*Optional[str], optional*) – This layer's name. Defaults to None.
- **weight_name** (*Optional[str], optional*) – This weight's name. Defaults to None.
- **bias_name** (*Optional[str], optional*) – This bias's name. Defaults to None.

Raises

- **ValueError** – If the type of kernel_size is not one of integer, list, tuple.
- **ValueError** – The number of groups must be positive and number of filters must be divisible by it.
- **ValueError** – If data_format is not one of 'NCHW', 'NHWC'.
- **ValueError** – If number of input channels is not divisible by number of groups or less than number of groups.
- **ValueError** – Number of group must be one when data_format is 'NHWC'.

Returns A 4D *Blob* with the shape of (batch_size, filters, new_height, new_width).

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def conv2d_Job(x: tp.Numpy.Placeholder((1, 256, 32, 32))
) -> tp.Numpy:
    initializer = flow.truncated_normal(0.1)
    conv2d = flow.layers.conv2d(
        x,
```

(continues on next page)

(continued from previous page)

```
        filters=128,
        kernel_size=3,
        strides=1,
        padding='SAME',
        kernel_initializer=initializer,
        name="Conv2d"
    )
    return conv2d


x = np.random.randn(1, 256, 32, 32).astype(np.float32)
out = conv2d_Job(x)

# out.shape (1, 128, 32, 32)
```

oneflow.layers.**conv3d**(*inputs: oneflow_api.BlobDesc, filters: int, kernel_size: Union[int, Sequence[int]] = 1, strides: Union[int, Sequence[int]] = 1, padding: Union[str, Tuple[Tuple[int, int], Tuple[int, int], Tuple[int, int], Tuple[int, int], Tuple[int, int]]] = 'VALID', data_format: str = 'NCDHW', dilation_rate: Union[int, Tuple[int, int], None] = None, groups: int = 1, activation: Optional[Callable[[oneflow_api.BlobDesc, str], oneflow_api.BlobDesc]] = None, use_bias: bool = True, kernel_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None, bias_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None, kernel_regularizer: Optional[oneflow.core.job.regularizer_conf_pb2.RegularizerConf] = None, bias_regularizer: Optional[oneflow.core.job.regularizer_conf_pb2.RegularizerConf] = None, trainable: bool = True, name: str = 'Conv3d', weight_name: Optional[str] = None, bias_name: Optional[str] = None*) → oneflow_api.BlobDesc

3D convolution layer.

This layer computes 3D convolution with 5D input Blob and filters

Parameters

- **inputs** (*oneflow_api.BlobDesc*) – A 5D input *Blob*.
- **filters** (*int*) – An integer specifies the dimensionality of the output space.
- **kernel_size** (*Union[int, List[int], Sequence[int]], optional*) – An integer or tuple/list specifies the height and width of the convolution window. When it is an integer, a square window is applied to the input. Defaults to 1.
- **strides** (*Union[int, List[int], Sequence[int]], optional*) – An integer or tuple/list specifies the strides of the convolution window along the height and width. When it is an integer, the same value for the all spatial dimesions is applied. Defaults to 1.
- **padding** (*str, Tuple[IntPair, IntPair, IntPair, IntPair, IntPair], optional*) – padding: *string "SAME"* or *"SAME_LOWER"* or *"SAME_UPPER"* or *"VALID" or Tuple[IntPair, IntPair, IntPair, IntPair, IntPair]* indicating the type of padding algorithm to use, or a list indicating the explicit paddings at the start and end of each dimension. Defaults to "VALID".
- **data_format** (*str, optional*) – A string specifies the format of the input *Blob*, one of "NCDHW" or "NDHWC" (default: "NCDHW"). "NCDHW" cooresponds to channels_first, i.e. the input *Blob* with shape (batch_size, channels, depth, height, width). "NDHWC" cooresponds to channels_last, i.e. the input *Blob* with shape (batch_size, channels, depth, height, width). Defaults to "NCDHW".

- **dilation_rate** (*int, optional*) – An integer or tuple/list specifies the dilation rate for the dilated convolution. When it is an integer, the same dilation rate is applied for the all dimensions. Defaults to 1.
- **groups** (*int, optional*) – A positive integer specifies number of groups for the Group conv. Defaults to 1.
- **activation** (*Optional[Callable[[oneflow_api.BlobDesc, str], oneflow_api.BlobDesc]], optional*) – Activation function. Defaults to None.
- **use_bias** (*bool, optional*) – A boolean specifies whether to use a bias vector. Defaults to True.
- **kernel_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for the kernel weights matrix. Defaults to None.
- **bias_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for the bias vector. Defaults to None.
- **kernel_regularizer** (*Optional[regularizer_conf_util.RegularizerConf], optional*) – Regularizer for the kernel weights matrix. Defaults to None.
- **bias_regularizer** (*Optional[regularizer_conf_util.RegularizerConf], optional*) – Regularizer for the bias vector . Defaults to None.
- **trainable** (*bool, optional*) – A boolean specifies whether to train variables. Defaults to True.
- **name** (*Optional[str], optional*) – This layer's name. Defaults to None.
- **weight_name** (*Optional[str], optional*) – This weight's name. Defaults to None.
- **bias_name** (*Optional[str], optional*) – This bias's name. Defaults to None.

Raises

- **ValueError** – If the type of kernel_size is not one of integer, list, tuple.
- **ValueError** – The number of groups must be positive and number of filters must be divisible by it.
- **ValueError** – If data_format is not one of 'NCDHW', 'NDHWC'.
- **ValueError** – If number of input channels is not divisible by number of groups or less than number of groups.
- **ValueError** – Number of group must be one when data_format is 'NDHWC'.

Returns A 5D *Blob* with the shape of (batch_size, filters, new_height, new_width).

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp
```

(continues on next page)

(continued from previous page)

```
@flow.global_function()
def conv3d_Job(x: tp.Numpy.Placeholder((1, 64, 16, 16, 16))
) -> tp.Numpy:
    initializer = flow.truncated_normal(0.1)
    conv3d = flow.layers.conv3d(
        x,
        filters=128,
        kernel_size=3,
        strides=1,
        padding='SAME',
        kernel_initializer=initializer,
        name="Conv3d"
    )
    return conv3d


x = np.random.randn(1, 64, 16, 16, 16).astype(np.float32)
out = conv3d_Job(x)

# out.shape (1, 128, 16, 16, 16)
```

oneflow.layers.**dense**(*inputs: oneflow_api.BlobDesc, units: int, activation: Optional[Callable[[oneflow_api.BlobDesc, str], oneflow_api.BlobDesc]] = None, use_bias: bool = True, kernel_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None, bias_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None, kernel_regularizer: Optional[oneflow.core.job.regularizer_conf_pb2.RegularizerConf] = None, bias_regularizer: Optional[oneflow.core.job.regularizer_conf_pb2.RegularizerConf] = None, trainable: bool = True, name: str = 'Dense', model_distribute: oneflow_api.distribute.Distribute = <oneflow_api.distribute.BroadcastDistribute object>*) → oneflow_api.BlobDesc

Fully-connected layer.

The fully-connected layer multiplies input Blob with weight matrix and produces an Output Blob.

Parameters

- **inputs** (*oneflow_api.BlobDesc*) – A 2D input *Blob*.
- **units** (*int*) – A positive integer for the dimensionality of the output space.
- **activation** (*Optional[oneflow_api.BlobDesc], optional*) – Activation function. Defaults to None.
- **use_bias** (*bool, optional*) – A boolean specifies whether to use a bias vector. Defaults to True.
- **kernel_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for the kernel weights matrix. Defaults to None.
- **bias_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – Initializer for the bias vector. Defaults to None.
- **kernel_regularizer** (*Optional[regularizer_conf_util.RegularizerConf], optional*) – Regularizer function applied to the kernel weights matrix. Defaults to None.

- **bias_regularizer** (*Optional[regularizer_conf_util.RegularizerConf], optional*) – Regularizer for the bias vector. Defaults to None.
- **trainable** (*bool, optional*) – A boolean specifies whether to train the variables. Defaults to True.
- **name** (*Optional[str], optional*) – This layer's name. Defaults to None.
- **model_distribute** (*oneflow_api.distribute.Distribute, optional*) – Define the way to ditribute the model. Defaults to oneflow_api.distribute.broadcast().

Returns A N-D *Blob* with the shape of (batch_size, units).

Return type oneflow_api.BlobDesc

Raises

- **ValueError** – The dimension of input *Blob* must be less than 2.
- **VaueError** – Model distribute must be in auto, broadcast, split.
- **ValueError** – The input must be a 2D *Blob* when the model distribute is split.

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def dense_Job(x: tp.Numpy.Placeholder((1, 256))
) -> tp.Numpy:
    initializer = flow.truncated_normal(0.1)
    hidden = flow.layers.dense(
        x,
        512,
        activation=flow.nn.relu,
        kernel_initializer=initializer,
        name="dense1",
    )
    return hidden


x = np.random.randn(1, 256).astype(np.float32)
out = dense_Job(x)

# out.shape (1, 512)
```

oneflow.layers.**layer_norm**(*inputs: oneflow_api.BlobDesc*, *center: bool = True*, *scale: bool = True*, *trainable: bool = True*, *begin_norm_axis: int = 1*, *begin_params_axis: int = -1*, *epsilon: float = 1e-05*, *name: str = 'LayerNorm'*) → oneflow_api.BlobDesc

Layer Normalization.

Parameters

- **inputs** (*oneflow_api.BlobDesc*) – Input *Blob*.
- **center** (*bool, optional*) – A boolean specifies whether to shift input *Blob*. Defaults to True.

- **scale** (*bool, optional*) – A boolean specifies whether to scale input *Blob*. Defaults to True.
- **trainable** (*bool, optional*) – A boolean specifies whether to train variables. Defaults to True.
- **begin_norm_axis** (*int, optional*) – An integer specifies which axis to normalize at first. Defaults to 1.
- **begin_params_axis** (*int, optional*) – An integer specifies which axis params at . Defaults to -1.
- **epsilon** (*float, optional*) – A small float is added to avoid division by zero. Defaults to 1e-5.
- **name** (*Optional[str], optional*) – This layer's name. Defaults to None.

Returns A normalized *Blob* with same shape of input.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def layer_norm_Job(x: tp.Numpy.Placeholder((1, 64, 128, 128))
) -> tp.Numpy:
    layer_norm = flow.layers.layer_norm(
        x,
        name="LayerNorm1"
    )
    return layer_norm


x = np.random.randn(1, 64, 128, 128).astype(np.float32)
out = layer_norm_Job(x)

# out.shape (1, 64, 128, 128)
```

oneflow.layers.**layer_norm_grad**(*dy: oneflow_api.BlobDesc*, *x: oneflow_api.BlobDesc*, *mean: oneflow_api.BlobDesc*, *inv_variance: oneflow_api.BlobDesc*, *begin_norm_axis: int = 1*, *name: str = 'LayerNormGrad'*) → oneflow_api.BlobDesc

Layer normalization

Parameters

- **dy** (*oneflow_api.BlobDesc*) – Upstream derivstives.
- **x** (*oneflow_api.BlobDesc*) – Input *Blob*.
- **mean** (*oneflow_api.BlobDesc*) – Mean over neurons.
- **inv_variance** (*oneflow_api.BlobDesc*) – Variance over neurons.
- **begin_norm_axis** (*int, optional*) – An integer specifies which axis to normalize at first. Defaults to 1.
- **name** (*Optional[str], optional*) – This layer's name. Defaults to None.

Returns Gradient with respect to input *Blob*.

Return type oneflow_api.BlobDesc

oneflow.layers.**layer_norm_param_grad**(*dy: oneflow_api.BlobDesc*, *norm: oneflow_api.BlobDesc*, *gamma: oneflow_api.BlobDesc*, *begin_params_axis: int = -1*, *name: str = 'LayerNormParamGrad'*) → Tuple[oneflow_api.BlobDesc, oneflow_api.BlobDesc, oneflow_api.BlobDesc]

Backward pass for layer normalization

Parameters

- **dy** (*oneflow_api.BlobDesc*) – Upstream derivstives.
- **norm** (*oneflow_api.BlobDesc*) – Normalized output.
- **gamma** (*oneflow_api.BlobDesc*) – Scale parameter.
- **begin_params_axis** (*int, optional*) – From which parameters to begin with. Defaults to -1.
- **name** (*Optional[str], optional*) – This layer's name. Defaults to 'LayerNormParamGrad'.

Returns normalized_diff: Gradient with respect to input *Blob*. beta_diff: Gradient with respect to shift parameter beta. gamma_diff: Gradient with respect to scale parameter gamma.

Return type Tuple[oneflow_api.BlobDesc]

oneflow.layers.**prelu**(*inputs: oneflow_api.BlobDesc*, *alpha_initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None*, *alpha_regularizer: Optional[oneflow.core.job.regularizer_conf_pb2.RegularizerConf] = None*, *shared_axes: Optional[Sequence[int]] = None*, *trainable: bool = True*, *name: str = 'PRelu'*, *model_distribute: oneflow_api.distribute.Distribute = <oneflow_api.distribute.BroadcastDistribute object>*) → oneflow_api.BlobDesc

The Prelu(Parametric Rectified Linear Unit) activation.

The α is a parameter that can be trained in network

The equation is

$$out = max(0, x) + \alpha * min(0, x)$$

Parameters

- **inputs** (*oneflow_api.BlobDesc*) – The input Blob.
- **alpha_initializer** (*Optional[initializer_conf_util.InitializerConf], optional*) – The initializer of alpha. Defaults to None.
- **alpha_regularizer** (*Optional[regularizer_conf_util.RegularizerConf], optional*) – The regularizer of alpha. Defaults to None.
- **shared_axes** (*Optional[Sequence[int]], optional*) – The axis along which to share learnable parameters for the prelu activation function. Defaults to None.
- **trainable** (*bool, optional*) – Whether to train the parameter α. Defaults to True.
- **name** (*str, optional*) – The name for the operation. Defaults to "PRelu".
- **model_distribute** (*oneflow_api.distribute.Distribute, optional*) – Define the way to ditribute the model. Defaults to oneflow_api.distribute.broadcast().

Returns The activated Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp

BATCH_SIZE = 100


def lenet(data, train=False):
    initializer = flow.truncated_normal(0.1)
    conv1 = flow.layers.conv2d(
        data,
        32,
        5,
        padding="SAME",
        name="conv1",
        kernel_initializer=initializer,
    )
    prelu1 = flow.layers.prelu(conv1,
                           alpha_initializer=initializer,
                           shared_axes=[2, 3],
                           name="Prelu1")
    pool1 = flow.nn.max_pool2d(
        prelu1, ksize=2, strides=2, padding="SAME", name="pool1", data_format=
↪"NCHW"
    )
    conv2 = flow.layers.conv2d(
        pool1,
        64,
        5,
        padding="SAME",
        name="conv2",
        kernel_initializer=initializer,
    )
    prelu2 = flow.layers.prelu(conv2,
                           alpha_initializer=initializer,
                           shared_axes=[2, 3],
                           name="Prelu2")
    pool2 = flow.nn.max_pool2d(
        prelu2, ksize=2, strides=2, padding="SAME", name="pool2", data_format=
↪"NCHW"
    )
    reshape = flow.reshape(pool2, [pool2.shape[0], -1])
    hidden = flow.layers.dense(
        reshape,
        512,
        activation=flow.nn.relu,
        kernel_initializer=initializer,
        name="dense1",
    )
    if train:
        hidden = flow.nn.dropout(hidden, rate=0.5, name="dropout")
    return flow.layers.dense(hidden, 10, kernel_initializer=initializer, name=
↪"dense2")
```

(continues on next page)

(continued from previous page)

```
@flow.global_function(type="train")
def train_job(
        images: tp.Numpy.Placeholder((BATCH_SIZE, 1, 28, 28), dtype=flow.float),
        labels: tp.Numpy.Placeholder((BATCH_SIZE,), dtype=flow.int32),
) -> tp.Numpy:
    with flow.scope.placement("gpu", "0:0"):
        logits = lenet(images, train=True)
        loss = flow.nn.sparse_softmax_cross_entropy_with_logits(
            labels, logits, name="softmax_loss"
        )

    lr_scheduler = flow.optimizer.PiecewiseConstantScheduler([], [0.1])
    flow.optimizer.SGD(lr_scheduler, momentum=0.9).minimize(loss)
    return loss
```

oneflow.layers.**upsample_2d**(*x: oneflow_api.BlobDesc*, *size: Sequence[int] = (2, 2)*, *data_format: str = 'NCHW'*, *interpolation: str = 'nearest'*, *name: str = 'Upsample2D'*)

The Upsample Layer, this layer can upsample the feature map to a specified scale.

Parameters

- **x** (*[type]*) – Input *Blob*.
- **size** (*tuple, optional*) – (height_scale, width_scale) Defaults to (2, 2).
- **data_format** (*str, optional*) – A string specifies the format of the input *Blob*, one of "NCHW" or "NHWC" (default: "NCHW"). "NCHW" cooresponds to channels_first, i.e. the input *Blob* with shape (batch_size, channels, height, width). "NHWC" cooresponds to channels_last, i.e. the input *Blob* with shape (batch_size, height, width, channels).. Defaults to "NCHW".
- **interpolation** (*str, optional*) – Image interpolation algorithm to enlarge the image size. Defaults to "nearest". "nearest" and "bilinear" are available now.
- **name** (*[type], optional*) – This layer's name. Defaults to None.

Raises

- **ValueError** – interpolation must be "nearest" or "bilinear".
- **ValueError** – data_format must be "NHWC" or "NCHW"

Returns oneflow_api.BlobDesc: A *Blob* which is the upsampled *x*. If *size* is (2, 2), the shape of return value is [N, C, 2H, 2W].

Return type [type]

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def upsample_Job(x: tp.Numpy.Placeholder((1, 32, 32, 32))
) -> tp.Numpy:
    upsample = flow.layers.upsample_2d(
```

(continues on next page)

(continued from previous page)

```
        x,
        size=(2, 2),
        name="Upsample1"
    )
    return upsample


x = np.random.randn(1, 32, 32, 32).astype(np.float32)
out = upsample_Job(x)

# out.shape (1, 32, 64, 64)
```

CHAPTER

TEN

ONEFLOW.DATA

class oneflow.data.**BlobConf**(*name: str, shape: Sequence[int], dtype: oneflow.python.framework.dtype.dtype, codec: Union[oneflow.python.ops.data_ops.ImageCodec, oneflow.python.ops.data_ops.RawCodec], preprocessors: Optional[Sequence[Union[oneflow.python.ops.data_ops.ImagePreprocessor, oneflow.python.ops.data_ops.ImageResizePreprocessor, oneflow.python.ops.data_ops.NormByChannelPreprocessor]]] = None*)

__init__(*name: str, shape: Sequence[int], dtype: oneflow.python.framework.dtype.dtype, codec: Union[oneflow.python.ops.data_ops.ImageCodec, oneflow.python.ops.data_ops.RawCodec], preprocessors: Optional[Sequence[Union[oneflow.python.ops.data_ops.ImagePreprocessor, oneflow.python.ops.data_ops.ImageResizePreprocessor, oneflow.python.ops.data_ops.NormByChannelPreprocessor]]] = None*) → None

Initialize self. See help(type(self)) for accurate signature.

to_proto() → oneflow.core.operator.op_conf_pb2.BlobConf

class oneflow.data.**BytesListCodec**

__init__() → None

Initialize self. See help(type(self)) for accurate signature.

to_proto(*proto: Optional[oneflow.core.operator.op_conf_pb2.EncodeConf] = None*) → oneflow.core.operator.op_conf_pb2.EncodeConf

class oneflow.data.**ImageCodec**(*image_preprocessors: Union[List[oneflow.python.ops.data_ops.ImageResizePreprocessor], Tuple[oneflow.python.ops.data_ops.ImageResizePreprocessor], None] = None*)

__init__(*image_preprocessors: Union[List[oneflow.python.ops.data_ops.ImageResizePreprocessor], Tuple[oneflow.python.ops.data_ops.ImageResizePreprocessor], None] = None*) → None

Initialize self. See help(type(self)) for accurate signature.

to_proto(*proto: Optional[oneflow.core.operator.op_conf_pb2.EncodeConf] = None*) → oneflow.core.operator.op_conf_pb2.EncodeConf

oneflow.data.**ImageDecoderRandomCropResize**(*input_blob: oneflow_api.BlobDesc*, *target_width: int*, *target_height: int*, *num_attempts: Optional[int] = None*, *seed: Optional[int] = None*, *random_area: Optional[Sequence[float]] = None*, *random_aspect_ratio: Optional[Sequence[float]] = None*, *num_workers: Optional[int] = None*, *warmup_size: Optional[int] = None*, *max_num_pixels: Optional[int] = None*, *name: Optional[str] = None*) → Tuple[oneflow_api.BlobDesc]

class oneflow.data.**ImagePreprocessor**(*preprocessor: str*)

__init__(*preprocessor: str*) → None
Initialize self. See help(type(self)) for accurate signature.

to_proto(*proto: Optional[oneflow.core.record.image_pb2.ImagePreprocess] = None*) → oneflow.core.record.image_pb2.ImagePreprocess

class oneflow.data.**ImageResizePreprocessor**(*width: int*, *height: int*)

__init__(*width: int*, *height: int*) → None
Initialize self. See help(type(self)) for accurate signature.

to_proto(*proto: Optional[oneflow.core.record.image_pb2.ImagePreprocess] = None*) → oneflow.core.record.image_pb2.ImagePreprocess

class oneflow.data.**NormByChannelPreprocessor**(*mean_values: Union[List[float], Tuple[float]], std_values: Union[List[float], Tuple[float]] = (1.0, 1.0, 1.0), data_format: str = 'channels_last'*)

__init__(*mean_values: Union[List[float], Tuple[float]], std_values: Union[List[float], Tuple[float]] = (1.0, 1.0, 1.0), data_format: str = 'channels_last'*) → None
Initialize self. See help(type(self)) for accurate signature.

to_proto(*proto: Optional[oneflow.core.operator.op_conf_pb2.PreprocessConf] = None*) → oneflow.core.operator.op_conf_pb2.PreprocessConf

oneflow.data.**OFRecordBytesDecoder**(*input_blob: oneflow_api.BlobDesc*, *blob_name: str*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.data.**OFRecordImageDecoder**(*input_blob: oneflow_api.BlobDesc*, *blob_name: str*, *color_space: str = 'BGR'*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator is an image decoder.

Parameters

- **input_blob** (*oneflow_api.BlobDesc*) – The input Blob
- **blob_name** (*str*) – The name of the input Blob
- **color_space** (*str, optional*) – The color space, such as “RGB”, “BGR”. Defaults to “BGR”.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp
from typing import Tuple


@flow.global_function(type="predict")
def image_decoder_job() -> Tuple[tp.Numpy, tp.Numpy]:
    batch_size = 16
    color_space = "RGB"
    # our ofrecord file path is "./dataset/part-0"
    ofrecord = flow.data.ofrecord_reader(
        "./imgdataset",
        batch_size=batch_size,
        data_part_num=1,
        part_name_suffix_length=-1,
        part_name_prefix='part-',
        random_shuffle=True,
        shuffle_after_epoch=True,
    )
    image = flow.data.OFRecordImageDecoder(
            ofrecord, "encoded", color_space=color_space
        )
    res_image, scale, new_size = flow.image.Resize(
            image, target_size=(224, 224)
        )
    label = flow.data.OFRecordRawDecoder(
        ofrecord, "class/label", shape=(1, ), dtype=flow.int32
    )

    return res_image, label

if __name__ == "__main__":
    images, labels = image_decoder_job()
    # image.shape (16, 224, 224, 3)
```

oneflow.data.**OFRecordImageDecoderRandomCrop**(*input_blob: oneflow_api.BlobDesc, blob_name: str, color_space: str = 'BGR', num_attempts: int = 10, seed: Optional[int] = None, random_area: Sequence[float] = [0.08, 1.0], random_aspect_ratio: Sequence[float] = [0.75, 1.333333], name: str = 'OFRecordImageDecoderRandomCrop'*) → oneflow_api.BlobDesc

This operator is an image decoder with random crop.

Parameters

- **input_blob** (*oneflow_api.BlobDesc*) – The input Blob
- **blob_name** (*str*) – The name of the Blob
- **color_space** (*str, optional*) – The color space, such as “RGB”, “BGR”. Defaults to “BGR”.
- **num_attempts** (*int, optional*) – The maximum number of random cropping attempts. Defaults to 10.
- **seed** (*Optional[int], optional*) – The random seed. Defaults to None.

- **random_area** (*Sequence[float], optional*) – The random cropping area. Defaults to [0.08, 1.0].
- **random_aspect_ratio** (*Sequence[float], optional*) – The random scaled ratio. Defaults to [0.75, 1.333333].
- **name** (*str, optional*) – The name for the operation. Defaults to "OFRecordImageDecoderRandomCrop".

Returns The random cropped Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp
from typing import Tuple


@flow.global_function(type="predict")
def ofrecord_reader_job() -> Tuple[tp.Numpy, tp.Numpy]:
    batch_size = 16
    color_space = "RGB"
    # our ofrecord file path is "./dataset/part-0"
    ofrecord = flow.data.ofrecord_reader(
        "./imgdataset",
        batch_size=batch_size,
        data_part_num=1,
        part_name_suffix_length=-1,
        part_name_prefix='part-',
        random_shuffle=True,
        shuffle_after_epoch=True,
    )
    image = flow.data.OFRecordImageDecoderRandomCrop(
            ofrecord, "encoded", color_space=color_space
        )
    res_image, scale, new_size = flow.image.Resize(
            image, target_size=(224, 224)
        )
    label = flow.data.OFRecordRawDecoder(
        ofrecord, "class/label", shape=(1, ), dtype=flow.int32
    )

    return res_image, label

if __name__ == "__main__":
    images, labels = ofrecord_reader_job()
    # images.shape (16, 224, 224, 3)
```

oneflow.data.**OFRecordRawDecoder**(*input_blob: oneflow_api.BlobDesc, blob_name: str, shape: Sequence[int], dtype: oneflow.python.framework.dtype.dtype, dim1_varying_length: bool = False, auto_zero_padding: bool = False, name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.data.**OneRecDecoder**(*input_blob*, *key*, *dtype*, *shape*, *is_dynamic=False*, *reshape=None*, *batch_padding=None*, *name=None*)

class oneflow.data.**RawCodec**(*auto_zero_padding: bool = False*)

__init__(*auto_zero_padding: bool = False*) → None
Initialize self. See help(type(self)) for accurate signature.

to_proto(*proto: Optional[oneflow.core.operator.op_conf_pb2.EncodeConf] = None*) → oneflow.core.operator.op_conf_pb2.EncodeConf

oneflow.data.**coco_reader**(*annotation_file: str, image_dir: str, batch_size: int, shuffle: bool = True, random_seed: Optional[int] = None, group_by_aspect_ratio: bool = True, stride_partition: bool = True, name: str = None*) → oneflow_api.BlobDesc

oneflow.data.**decode_random**(*shape: Sequence[int], dtype: oneflow.python.framework.dtype.dtype, batch_size: int = 1, initializer: Optional[oneflow.core.job.initializer_conf_pb2.InitializerConf] = None, tick: Optional[oneflow_api.BlobDesc] = None, name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.data.**image_decoder_random_crop_resize**(*input_blob: oneflow_api.BlobDesc, target_width: int, target_height: int, num_attempts: Optional[int] = None, seed: Optional[int] = None, random_area: Optional[Sequence[float]] = None, random_aspect_ratio: Optional[Sequence[float]] = None, num_workers: Optional[int] = None, warmup_size: Optional[int] = None, max_num_pixels: Optional[int] = None, name: Optional[str] = None*) → Tuple[oneflow_api.BlobDesc]

oneflow.data.**load_mnist**(*train_batch_size=100, test_batch_size=100, data_format='NCHW', url='https://oneflow-public.oss-cn-beijing.aliyuncs.com/datasets/mnist.npz', hash_check='63d4344077849053dc3036b247fa012b2b381de53fd055a66b539dffd76cf08e', out_dir='.'*)

Load mnist dataset, return images and labels, if dataset doesn't exist, then download it to directory that out_dir specified

Parameters

- **train_batch_size** (*int, optional*) – batch size for train. Defaults to 100.
- **test_batch_size** (*int, optional*) – batch size for test or evaluate. Defaults to 100.
- **data_format** (*str, optional*) – data format. Defaults to "NCHW".
- **url** (*str, optional*) – url to get mnist.npz. Defaults to "https://oneflow-public.oss-cn-beijing.aliyuncs.com/datasets/mnist.npz".
- **hash_check** (*str, optional*) – file hash value. Defaults to "63d4344077849053dc3036b247fa012b2b381de53fd055a66b539dffd76cf08e".
- **out_dir** (*str, optional*) – dir to save downloaded file. Defaults to "./".

Returns (train_images, train_labels), (test_images, test_labels)

Return type [type]

oneflow.data.**ofrecord_bytes_decoder**(*input_blob: oneflow_api.BlobDesc, blob_name: str, name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.data.**ofrecord_image_classification_reader**(*ofrecord_dir: str*, *image_feature_name: str*, *label_feature_name: str*, *batch_size: int = 1*, *data_part_num: int = 1*, *part_name_prefix: str = 'part-'*, *part_name_suffix_length: int = -1*, *random_shuffle: bool = False*, *shuffle_buffer_size: int = 1024*, *shuffle_after_epoch: bool = False*, *color_space: str = 'BGR'*, *decode_buffer_size_per_thread: int = 32*, *num_decode_threads_per_machine: Optional[int] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator creates a reader for image classification tasks.

Parameters

- **ofrecord_dir** (*str*) – The directory of ofrecord file.
- **image_feature_name** (*str*) – The name of the image feature.
- **label_feature_name** (*str*) – The name of the label feature.
- **batch_size** (*int, optional*) – The batch_size. Defaults to 1.
- **data_part_num** (*int, optional*) – The amounts of data part. Defaults to 1.
- **part_name_prefix** (*str, optional*) – The prefix of data part name. Defaults to "part-".
- **part_name_suffix_length** (*int, optional*) – The suffix name of data part name. Defaults to -1.
- **random_shuffle** (*bool, optional*) – Whether to random shuffle the data. Defaults to False.
- **shuffle_buffer_size** (*int, optional*) – The buffer size for shuffle data. Defaults to 1024.
- **shuffle_after_epoch** (*bool, optional*) – Whether to shuffle the data after each epoch. Defaults to False.
- **color_space** (*str, optional*) – The color space. Defaults to "BGR".
- **decode_buffer_size_per_thread** (*int, optional*) – The decode buffer size for per thread. Defaults to 32.
- **num_decode_threads_per_machine** (*Optional[int], optional*) – The amounts of decode threads for each machine. Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp
from typing import Tuple


@flow.global_function(type="predict")
def image_classifier_job() -> Tuple[tp.Numpy, tp.Numpy]:
    image, label = flow.data.ofrecord_image_classification_reader(
        ofrecord_dir="./imgdataset",
        image_feature_name="encoded",
        label_feature_name="class/label",
        batch_size=8,
        data_part_num=1,
        part_name_prefix="part-",
        part_name_suffix_length=-1,
        random_shuffle=False,
        shuffle_after_epoch=False,
        color_space="RGB",
        decode_buffer_size_per_thread=16,
    )
    res_image, scale, new_size = flow.image.Resize(
            image, target_size=(224, 224)
        )
    return res_image, label


if __name__ == "__main__":
    images, labels = image_classifier_job()
    # images.shape (8, 224, 224, 3)
```

oneflow.data.**ofrecord_image_decoder**(*input_blob: oneflow_api.BlobDesc*, *blob_name: str*, *color_space: str = 'BGR'*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator is an image decoder.

Parameters

- **input_blob** (*oneflow_api.BlobDesc*) – The input Blob
- **blob_name** (*str*) – The name of the input Blob
- **color_space** (*str, optional*) – The color space, such as "RGB", "BGR". Defaults to "BGR".
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp
from typing import Tuple


@flow.global_function(type="predict")
def image_decoder_job() -> Tuple[tp.Numpy, tp.Numpy]:
    batch_size = 16
```

(continues on next page)

(continued from previous page)

```
    color_space = "RGB"
    # our ofrecord file path is "./dataset/part-0"
    ofrecord = flow.data.ofrecord_reader(
        "./imgdataset",
        batch_size=batch_size,
        data_part_num=1,
        part_name_suffix_length=-1,
        part_name_prefix='part-',
        random_shuffle=True,
        shuffle_after_epoch=True,
    )
    image = flow.data.OFRecordImageDecoder(
            ofrecord, "encoded", color_space=color_space
        )
    res_image, scale, new_size = flow.image.Resize(
            image, target_size=(224, 224)
        )
    label = flow.data.OFRecordRawDecoder(
        ofrecord, "class/label", shape=(1, ), dtype=flow.int32
    )

    return res_image, label

if __name__ == "__main__":
    images, labels = image_decoder_job()
    # image.shape (16, 224, 224, 3)
```

oneflow.data.**ofrecord_image_decoder_random_crop**(*input_blob: oneflow_api.BlobDesc, blob_name: str, color_space: str = 'BGR', num_attempts: int = 10, seed: Optional[int] = None, random_area: Sequence[float] = [0.08, 1.0], random_aspect_ratio: Sequence[float] = [0.75, 1.333333], name: str = 'OFRecordImageDecoderRandomCrop'*) → oneflow_api.BlobDesc

This operator is an image decoder with random crop.

Parameters

- **input_blob** (*oneflow_api.BlobDesc*) – The input Blob
- **blob_name** (*str*) – The name of the Blob
- **color_space** (*str, optional*) – The color space, such as "RGB", "BGR". Defaults to "BGR".
- **num_attempts** (*int, optional*) – The maximum number of random cropping attempts. Defaults to 10.
- **seed** (*Optional[int], optional*) – The random seed. Defaults to None.
- **random_area** (*Sequence[float], optional*) – The random cropping area. Defaults to [0.08, 1.0].
- **random_aspect_ratio** (*Sequence[float], optional*) – The random scaled ratio. Defaults to [0.75, 1.333333].

- **name** (*str, optional*) – The name for the operation. Defaults to "OFRecordImageDecoderRandomCrop".

Returns The random cropped Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp
from typing import Tuple


@flow.global_function(type="predict")
def ofrecord_reader_job() -> Tuple[tp.Numpy, tp.Numpy]:
    batch_size = 16
    color_space = "RGB"
    # our ofrecord file path is "./dataset/part-0"
    ofrecord = flow.data.ofrecord_reader(
        "./imgdataset",
        batch_size=batch_size,
        data_part_num=1,
        part_name_suffix_length=-1,
        part_name_prefix='part-',
        random_shuffle=True,
        shuffle_after_epoch=True,
    )
    image = flow.data.OFRecordImageDecoderRandomCrop(
            ofrecord, "encoded", color_space=color_space
        )
    res_image, scale, new_size = flow.image.Resize(
            image, target_size=(224, 224)
        )
    label = flow.data.OFRecordRawDecoder(
        ofrecord, "class/label", shape=(1, ), dtype=flow.int32
    )

    return res_image, label

if __name__ == "__main__":
    images, labels = ofrecord_reader_job()
    # images.shape (16, 224, 224, 3)
```

oneflow.data.**ofrecord_loader**(*ofrecord_dir: str, batch_size: int = 1, data_part_num: int = 1, part_name_prefix: str = 'part-', part_name_suffix_length: int = -1, shuffle: bool = False, shuffle_buffer_size: int = 1024, name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.data.**ofrecord_raw_decoder**(*input_blob: oneflow_api.BlobDesc, blob_name: str, shape: Sequence[int], dtype: oneflow.python.framework.dtype.dtype, dim1_varying_length: bool = False, auto_zero_padding: bool = False, name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.data.**ofrecord_reader**(*ofrecord_dir: str, batch_size: int = 1, data_part_num: int = 1, part_name_prefix: str = 'part-', part_name_suffix_length: int = -1, random_shuffle: bool = False, shuffle_buffer_size: int = 1024, shuffle_after_epoch: bool = False, name: Optional[str] = None*) → oneflow_api.BlobDesc

Get ofrecord object from ofrecord dataset.

Parameters

- **ofrecord_dir** (*str*) – Path to ofrecord dataset.
- **batch_size** (*int, optional*) – Batch size. Defaults to 1.
- **data_part_num** (*int, optional*) – Number of dataset's partitions. Defaults to 1.
- **part_name_prefix** (*str, optional*) – Prefix of dataset's parition file. Defaults to "part-".
- **part_name_suffix_length** (*int, optional*) – Total length of padded suffix number , -1 means no padding. eg: 3 for *part-001*. Defaults to -1.
- **random_shuffle** (*bool, optional*) – Determines records shuffled or not. Defaults to False.
- **shuffle_buffer_size** (*int, optional*) – Shuffle buffer size. Defaults to 1024.
- **shuffle_after_epoch** (*bool, optional*) – Shuffled or not after each epoch. Defaults to False.
- **name** (*Optional[str], optional*) – Optional name. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp
from typing import Tuple


@flow.global_function(type="predict")
def ofrecord_reader_job() -> Tuple[tp.Numpy, tp.Numpy]:
    batch_size = 16
    with flow.scope.placement("cpu", "0:0"):
        # our ofrecord file path is "./dataset/part-0"
        ofrecord = flow.data.ofrecord_reader(
            "./dataset/",
            batch_size=batch_size,
            data_part_num=1,
            part_name_suffix_length=-1,
            part_name_prefix='part-',
            random_shuffle=True,
            shuffle_after_epoch=True,
        )
        # image shape is (28*28, )
        image = flow.data.OFRecordRawDecoder(
            ofrecord, "images", shape=(784, ), dtype=flow.int32
        )
        # label shape is (1, )
        label = flow.data.OFRecordRawDecoder(
            ofrecord, "labels", shape=(1, ), dtype=flow.int32
        )

        return image, label
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    images, labels = ofrecord_reader_job()
    print("In per batch, images shape is", images.shape)
    print("In per batch, labels shape is", labels.shape)

    # In per batch, images shape is (16, 784)
    # In per batch, labels shape is (16, 1)
```

oneflow.data.**onerec_decoder**(*input_blob*, *key*, *dtype*, *shape*, *is_dynamic=False*, *reshape=None*, *batch_padding=None*, *name=None*)

oneflow.data.**onerec_reader**(*files*, *batch_size=1*, *random_shuffle=False*, *shuffle_mode='instance'*, *shuffle_buffer_size=1024*, *shuffle_after_epoch=False*, *verify_example=True*, *name=None*)

CHAPTER

ELEVEN

ONEFLOW.DISTRIBUTE

oneflow.distribute.**assert_is_valid_distribute**(*distribute: oneflow_api.distribute.Distribute*) → None

oneflow.distribute.**auto**() → oneflow_api.distribute.AutoDistribute

Generate a broadcast scheme.

Returns Auto distribute scheme object, often required by *with_distribute* method of *Blob* or *oneflow.get_variable*.

Return type AutoDistribute

oneflow.distribute.**broadcast**() → oneflow_api.distribute.BroadcastDistribute

Generate a broadcast scheme.

Returns Broadcast scheme object, often required by *with_distribute* method of *Blob* or *oneflow.get_variable*.

Return type BroadcastDistribute

Example:: segment_ids = segment_ids.with_distribute(flow.distribute.broadcast())

oneflow.distribute.**split**(*axis: int*) → oneflow_api.distribute.SplitDistribute

Generate a split scheme in which op will be splitted at *axis*.

Parameters **axis** (*int*) – At *axis* the op will be splitted.

Returns Split scheme object, often required by *with_distribute* method of *Blob* or *oneflow.get_variable*.

Return type SplitDistribute

Example:: weight = weight.with_distribute(distribute.split(1))

CHAPTER

TWELVE

ONEFLOW.ADVANCED

12.1 Advanced features

oneflow.advanced.**distribute_add**(*xs: Sequence[oneflow_api.BlobDesc], name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.advanced.**distribute_clone**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → Tuple[oneflow_api.BlobDesc]

oneflow.advanced.**distribute_concat**(*xs: Sequence[oneflow_api.BlobDesc], axis: int = 0, name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.advanced.**distribute_map**(*xs: Union[Sequence[oneflow_api.BlobDesc], oneflow_api.BlobDesc], f: Callable[[oneflow_api.BlobDesc, oneflow_api.BlobDesc], oneflow_api.BlobDesc], axis: int = 0*) → Tuple[oneflow_api.BlobDesc]

oneflow.advanced.**distribute_split**(*x: oneflow_api.BlobDesc*, *axis: int = 0*, *name: Optional[str] = None*) → Tuple[oneflow_api.BlobDesc]

CHAPTER

THIRTEEN

ONEFLOW.TYPING

13.1 Typing system

class oneflow.typing.**Bundle**

One or a collection of typing.Numpy/typing.ListNumpy/typing.ListListNumpy, such as x, [x], (x,), {"key": x} and the mixed form of them.

class oneflow.typing.**Callback**

class oneflow.typing.**ListListNumpy**

ListListNumpy is a type hint for numpy output of a OneFlow global function For instance:

```
@oneflow.global_function()
def foo() -> oneflow.typing.ListListNumpy:
    mirrored_tensor_lists = ... # your network
    return mirrored_tensor_lists

mirrored_tensor_lists = foo() # get a list of list of numpy.ndarray
for tensor_list in mirrored_tensor_lists:
    for tensor in tensor_list:
        print(mirrored_tensors)
```

Placeholder(*dtype=<class 'oneflow.python.framework.dtype.float32'>*, *batch_axis: Optional[int] = 0*)

ListListNumpy.Placeholder is a typing function for numpy input of a OneFlow global function. A *list* of *list* of *numpy.ndarray* takes a *ListListNumpy.Placeholder*'s place. Each *numpy.ndarray* in the *list* could have any shape as long as it has the same rank and a smaller/equal size. For instance:

```
@oneflow.global_function()
def foo(
    image_blob: oneflow.typing.ListListNumpy.Placeholder(
        (2, 255, 255, 3), dtype=flow.float32
    )
):
    # your network

input1 = np.random.randn(2, 255, 255, 3).astype(np.float32)
input2 = np.random.randn(2, 251, 251, 3).astype(np.float32)
foo([[input1]])
foo([[input2]])
```

class oneflow.typing.**ListNumpy**

ListNumpy is a type hint for numpy output of a OneFlow global function For instance:

```
@oneflow.global_function()
def foo() -> oneflow.typing.ListNumpy:
    mirrored_tensors = ... # your network
    return mirrored_tensors

mirrored_tensors = foo() # get a list of numpy.ndarray
for tensor in mirrored_tensors:
    print(mirrored_tensors)
```

Placeholder(*dtype=<class 'oneflow.python.framework.dtype.float32'>*, *batch_axis: Optional[int] = 0*)

ListNumpy.Placeholder is a typing function for numpy input of a OneFlow global function. A *list* of *numpy.ndarray* takes a *ListNumpy.Placeholder*'s place. Each *numpy.ndarray* in the *list* could have any shape as long as it has the same rank and a smaller/equal size. For instance:

```
@oneflow.global_function()
def foo(
    image_blob: oneflow.typing.ListNumpy.Placeholder(
        (2, 255, 255, 3), dtype=flow.float32
    )
):
    # your network

input1 = np.random.randn(2, 255, 255, 3).astype(np.float32)
input2 = np.random.randn(2, 251, 251, 3).astype(np.float32)
foo([input1])
foo([input2])
```

class oneflow.typing.**Numpy**

Numpy is a type hint for numpy output of a OneFlow global function For instance:

```
@oneflow.global_function()
def foo() -> oneflow.typing.Numpy:
    loss = ... # your network
    return loss

loss = foo() # get a numpy.ndarray
print(loss)
```

Placeholder(*dtype=<class 'oneflow.python.framework.dtype.float32'>*, *batch_axis: Optional[int] = 0*)

Numpy.Placeholder is a typing function for numpy input of a OneFlow global function. A *numpy.ndarray* takes a *Numpy.Placeholder*'s place must have a identical shape. For instance:

```
@oneflow.global_function()
def foo(
    image_blob: oneflow.typing.Numpy.Placeholder(
        (2, 255, 255, 3), dtype=flow.float32
    )
):
    # your network

foo(np.random.randn(2, 255, 255, 3).astype(np.float32))
```

CHAPTER

FOURTEEN

ONEFLOW.TENSORRT

14.1 TensorRT integration

oneflow.tensorrt.**cache_int8_calibration**()

oneflow.tensorrt.**write_int8_calibration**(*path*)

CHAPTER

FIFTEEN

ONEFLOW.DEPRECATED

15.1 Deprecated APIs

oneflow.deprecated.**delete_worker**(*ssh_port=22*) → None

oneflow.deprecated.**init_worker**(*scp_binary: bool = True*, *use_uuid: bool = True*, *ssh_port=22*) → None

CHAPTER

SIXTEEN

ONEFLOW.EXPERIMENTAL

16.1 Experimental features

class oneflow.experimental.**CustomOpModule**(*op_module_name*, *module_path=''*)

__init__(*op_module_name*, *module_path=''*)
Initialize self. See help(type(self)) for accurate signature.

build_load()

cpp_def()

cpp_kernel()

gpu_kernel()

py_api()

py_kernel()

oneflow.experimental.**custom_op_module**
alias of oneflow.python.ops.util.custom_op_module.CustomOpModule

oneflow.experimental.**dynamic_binary_concat**(*input_blob_list: Sequence[oneflow_api.BlobDesc], source_blob: oneflow.python.framework.input_blob_def.ArgBlobDef, source_sbp: str = 'S:0', name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.experimental.**dynamic_binary_split**(*x: oneflow.python.framework.input_blob_def.ArgBlobDef, base_shift: int = 2, out_num: int = 2, name: Optional[str] = None*) → List[oneflow_api.BlobDesc]

oneflow.experimental.**eager_assign_121**(*ref*, *value*)

oneflow.experimental.**enable_typing_check**(*val: bool = True*) → None
enable typing check for global_function

oneflow.experimental.**get_interface_blob_value**(*op_name*)

oneflow.experimental.**indexed_slices_reduce_sum**(*indices: oneflow.python.framework.input_blob_def.ArgBlobDef, values: oneflow.python.framework.input_blob_def.ArgBlobDef, name: Optional[str] = None*) → Tuple[oneflow_api.BlobDesc]

oneflow.experimental.**logical_slice**(*x: oneflow_api.BlobDesc, slice_tup_list: Sequence[Tuple[int, int, int]], name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.experimental.**logical_slice_assign**(*x: oneflow_api.BlobDesc, value: oneflow_api.BlobDesc, slice_tup_list: Sequence[Tuple[int, int, int]], name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.experimental.**set_interface_blob_value**(*op_name*, *ndarray*)

oneflow.experimental.**square_sum**(*x: oneflow_api.BlobDesc*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

oneflow.experimental.**ssp_variable_proxy**(*var: oneflow_api.BlobDesc*, *buffer_size: int = 1*, *name=None*) → Tuple[oneflow_api.BlobDesc, oneflow_api.BlobDesc]

return ref_blob, value_blob

oneflow.experimental.**unique_with_counts**(*x: oneflow.python.framework.input_blob_def.ArgBlobDef*, *out_idx: oneflow.python.framework.dtype.dtype = <class 'oneflow.python.framework.dtype.int32'>*, *name: Optional[str] = None*) → Tuple[oneflow_api.BlobDesc]

CHAPTER

SEVENTEEN

ONEFLOW.SCOPE

17.1 Scope

class oneflow.scope.**DistributeConsistentStrategy**

Create a scope in consistent view. All operators within the scope will be automatically parallelized among diffierent accelerators for best performance and least data transfer.

Usage:

```
with oneflow.scope.consistent_view():
    ...
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

class oneflow.scope.**DistributeMirroredStrategy**

Create a scope in mirrored view. All operators within the scope will be mirrored among diffierent accelerators. Usage:

```
with oneflow.scope.mirrored_view():
    ...
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

oneflow.scope.**consistent_view**

alias of oneflow.python.framework.distribute.DistributeConsistentStrategy

oneflow.scope.**consistent_view_enabled**() → bool

Returns *True* if consistent strategy is enabled in current context where this function is called.

Return type bool

oneflow.scope.**mirrored_view**

alias of oneflow.python.framework.distribute.DistributeMirroredStrategy

oneflow.scope.**mirrored_view_enabled**() → bool

Returns *True* if mirrored strategy is enabled in current context where this function is called.

Return type bool

oneflow.scope.**namespace**(*name: str*) → None

Create a namespace. All variables within the namespace will have a prefix *[SCOPE NAME]-*. This is for convenience only and has no other effect on the system. Usage:

```
with oneflow.scope.namespace("scope1"):
    ...
    with oneflow.scope.namespace("scope2"):
        ...
```

Parameters **name** – Name of this namespace

oneflow.scope.**placement**(*device_tag: str*, *machine_device_ids: str*) → oneflow.python.framework.placement_context.PlacementScope

Create a scope. All ops within the scope will run on specified device that placed by "device_tag" and "machine_device_ids".

Parameters

- **device_tag** (*str*) – Device tag, "cpu" or "gpu" only
- **machine_device_ids** (*str*) – List of string that specifies what machine & device(s) to use, the format is "List[<NODE INDEX>:<DEVICE START INDEX>-<DEVICE END INDEX>, <NODE INDEX>:<DEVICE START INDEX>-<DEVICE END INDEX>, ...]", For example, "0:0" means use the device 0 of machine 0, and "1:4-6" means use device 4, 5, 6 of machine 1.

Returns Placement scope

Return type placement_ctx.DevicePriorPlacementScope

For example:

If you run program on single machine, you can assign the specified device like this:

```
with flow.scope.placement("gpu", "0:0"):
    logits = lenet(images, train=False)
    loss = flow.nn.sparse_softmax_cross_entropy_with_logits(labels, logits, name=
↪"softmax_loss")
    flow.losses.add_loss(loss)
```

Or you run distributed program, you can assign the specified devices like this:

```
# configure machines ids, ips, etc.
with flow.scope.placement("gpu", ['0:0-7', '1:0-7']):
    logits = lenet(images, train=False)
    loss = flow.nn.sparse_softmax_cross_entropy_with_logits(labels, logits, name=
↪"softmax_loss")
    flow.losses.add_loss(loss)
```

CHAPTER

EIGHTEEN

ONEFLOW.SYSCONFIG

18.1 System configurations

oneflow.sysconfig.**get_compile_flags**() → List[str]

oneflow.sysconfig.**get_include**() → str

oneflow.sysconfig.**get_lib**() → str

oneflow.sysconfig.**get_link_flags**() → List[str]

CHAPTER

NINETEEN

ONEFLOW.ONNX

19.1 ONNX integration

oneflow.onnx.**export**(*job_func: Callable*, *model_save_dir: str*, *onnx_filename: str*, *continue_on_error: bool = False*, *opset: Optional[int] = None*, *extra_opset: Optional[int] = None*, *shape_override: Optional[Dict[str, List[int]]] = None*, *external_data: bool = False*)

Export a oneflow model into ONNX format.

Parameters

- **job_func** – The job function
- **model_save_dir** – The directory containing oneflow model weights. Users are expected to call check_point.save(dir), wait for the model saving finishing, and pass the argument 'dir' as model_save_dir.
- **onnx_filename** – a string for the output filename
- **continue_on_error** – if an op can't be processed (aka there is no mapping), continue
- **opset** – the opset to be used (int, default is oneflow.python.onnx.constants.PREFERRED_OPSET)
- **extra_opset** – list of extra opset's, for example the opset's used by custom ops
- **shape_override** – dict with inputs that override the shapes given by oneflow
- **external_data** – Save weights as ONNX external data, usually to bypass the 2GB file size limit of protobuf.

CHAPTER

TWENTY

ONEFLOW.RANDOM

20.1 Random

oneflow.random.**CoinFlip**(*batch_size: int = 1*, *seed: Optional[int] = None*, *probability: float = 0.5*, *name: str = 'CoinFlip'*) → oneflow_api.BlobDesc

This operator performs the horizontal flip.

Parameters

- **batch_size** (*int, optional*) – The batch size. Defaults to 1.
- **seed** (*Optional[int], optional*) – The random seed. Defaults to None.
- **probability** (*float, optional*) – The flip probability. Defaults to 0.5.
- **name** (*str, optional*) – The name for the operation. Defaults to "CoinFlip".

Returns [description]

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp
from typing import Tuple


@flow.global_function(type="predict")
def coin_flip_job() -> Tuple[tp.Numpy, tp.Numpy]:
    batch_size = 1
    color_space = "RGB"
    # our ofrecord file path is "./dataset/part-0"
    ofrecord = flow.data.ofrecord_reader(
        "./imgdataset",
        batch_size=batch_size,
        data_part_num=1,
        part_name_suffix_length=-1,
        part_name_prefix='part-',
        shuffle_after_epoch=True,
    )
    image = flow.data.OFRecordImageDecoder(
            ofrecord, "encoded", color_space=color_space
        )
    res_image, scale, new_size = flow.image.Resize(
            image, target_size=(512, 512)
        )
```

(continues on next page)

(continued from previous page)

```
    label = flow.data.OFRecordRawDecoder(
        ofrecord, "class/label", shape=(1, ), dtype=flow.int32
    )
    coin_flip = flow.random.CoinFlip(
        batch_size=batch_size,
        probability=0.8
    )
    normal = flow.image.CropMirrorNormalize(
            res_image,
            mirror_blob=coin_flip,
            color_space=color_space,
            crop_h= 256,
            crop_w= 256,
            crop_pos_y=0.5,
            crop_pos_x=0.5,
            mean=[123.68, 116.779, 103.939],
            std=[58.393, 57.12, 57.375],
            output_dtype=flow.float,
        )

    return normal, label

if __name__ == "__main__":
    images, labels = coin_flip_job()
```

oneflow.random.**bernoulli**(*x: oneflow_api.BlobDesc*, *seed: Optional[int] = None*, *dtype: Optional[oneflow.python.framework.dtype.dtype] = None*, *name: str = 'Bernoulli'*) → oneflow_api.BlobDesc

This operator returns a Blob with binaray random numbers (0 / 1) from a Bernoulli distribution.

Parameters

- **x** (*oneflow_api.BlobDesc*) – The input Blob.
- **seed** (*Optional[int], optional*) – The random seed. Defaults to None.
- **dtype** (*Optional[dtype_util.dtype], optional*) – The data type. Defaults to None.
- **name** (*str, optional*) – The name for the operation. Defaults to "Bernoulli".

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def bernoulli_Job(x: tp.Numpy.Placeholder(shape=(3, 3), dtype=flow.float32),
) -> tp.Numpy:
    out = flow.random.bernoulli(x)
    return out
```

(continues on next page)

(continued from previous page)

```
x = np.array([[0.25, 0.45, 0.3],
              [0.55, 0.32, 0.13],
              [0.75, 0.15, 0.1]]).astype(np.float32)
out = bernoulli_Job(x)

# Because our random seed is not fixed, so the return value is different each␣
↪time.
# out [[1. 0. 0.]
#      [0. 0. 1.]
#      [0. 0. 0.]]
```

oneflow.random.**coin_flip**(*batch_size: int = 1*, *seed: Optional[int] = None*, *probability: float = 0.5*, *name: str = 'CoinFlip'*) → oneflow_api.BlobDesc

This operator performs the horizontal flip.

Parameters

- **batch_size** (*int, optional*) – The batch size. Defaults to 1.
- **seed** (*Optional[int], optional*) – The random seed. Defaults to None.
- **probability** (*float, optional*) – The flip probability. Defaults to 0.5.
- **name** (*str, optional*) – The name for the operation. Defaults to "CoinFlip".

Returns [description]

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp
from typing import Tuple


@flow.global_function(type="predict")
def coin_flip_job() -> Tuple[tp.Numpy, tp.Numpy]:
    batch_size = 1
    color_space = "RGB"
    # our ofrecord file path is "./dataset/part-0"
    ofrecord = flow.data.ofrecord_reader(
        "./imgdataset",
        batch_size=batch_size,
        data_part_num=1,
        part_name_suffix_length=-1,
        part_name_prefix='part-',
        shuffle_after_epoch=True,
    )
    image = flow.data.OFRecordImageDecoder(
            ofrecord, "encoded", color_space=color_space
        )
    res_image, scale, new_size = flow.image.Resize(
            image, target_size=(512, 512)
        )
    label = flow.data.OFRecordRawDecoder(
        ofrecord, "class/label", shape=(1, ), dtype=flow.int32
    )
    coin_flip = flow.random.CoinFlip(
```

(continues on next page)

(continued from previous page)

```
        batch_size=batch_size,
        probability=0.8
    )
    normal = flow.image.CropMirrorNormalize(
            res_image,
            mirror_blob=coin_flip,
            color_space=color_space,
            crop_h= 256,
            crop_w= 256,
            crop_pos_y=0.5,
            crop_pos_x=0.5,
            mean=[123.68, 116.779, 103.939],
            std=[58.393, 57.12, 57.375],
            output_dtype=flow.float,
        )

    return normal, label

if __name__ == "__main__":
    images, labels = coin_flip_job()
```

oneflow.random.**gen_seed**(*seed: Optional[int] = None*)

oneflow.random.**generate_random_batch_permutation_indices**(*value: oneflow_api.BlobDesc*, *seed: Optional[int] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator generates a random permutation of indices in batch axis.

Parameters

- **value** (*oneflow_api.BlobDesc*) – The input Blob.
- **seed** (*Optional[int], optional*) – The random seed. Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob. Its type is *ListNumpy*.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def random_indice_Job(x: tp.Numpy.Placeholder(shape=(4, 3), dtype=flow.int32),
) -> tp.ListNumpy:
    return flow.random.generate_random_batch_permutation_indices(value=x)

x = np.array([[1, 1, 1],
              [2, 2, 2],
              [3, 3, 3],
              [4, 4, 4]]).astype(np.int32)
```

(continues on next page)

(continued from previous page)

```
out = random_indice_Job(x)

# out [array([3, 0, 2, 1], dtype=int32)]
```

oneflow.random.**shuffle**(*value: oneflow_api.BlobDesc*, *seed: Optional[int] = None*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator shuffle the elements in input Blob.

Parameters

- **value** (*oneflow_api.BlobDesc*) – The input Blob.
- **seed** (*Optional[int], optional*) – The random seed. Defaults to None.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def shuffle_Job(x: tp.Numpy.Placeholder(shape=(3, 3), dtype=flow.int32),
) -> tp.Numpy:
    return flow.random.shuffle(x)

x = np.array([[1, 1, 1],
              [2, 2, 2],
              [3, 3, 3]]).astype(np.int32)
out = shuffle_Job(x)

# out [[3 3 3]
#      [1 1 1]
#      [2 2 2]]
```

CHAPTER

TWENTYONE

ONEFLOW.SYSTEM

21.1 System configurations

oneflow.system.**assign**(*ref*, *value*, *validate_shape=None*, *use_locking=None*, *name=None*)

CHAPTER

TWENTYTWO

ONEFLOW.REGULARIZERS

22.1 Regularizers

oneflow.regularizers.**l1**(*l: float = 0.01*) → oneflow.core.job.regularizer_conf_pb2.RegularizerConf

This operator creates a L1 weight regularizer.

Parameters **l** (*float, optional*) – The L1 regularization coefficient. Defaults to 0.01.

Returns A regularizer that can be used in other layers or operators.

Return type regularizer_conf_util.RegularizerConf

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def conv2d_l1_Job(x: tp.Numpy.Placeholder((1, 256, 32, 32))
) -> tp.Numpy:
    initializer = flow.truncated_normal(0.1)
    regularizer = flow.regularizers.l1(l=0.001)
    conv2d = flow.layers.conv2d(
        x,
        filters=128,
        kernel_size=3,
        strides=1,
        padding='SAME',
        kernel_initializer=initializer,
        kernel_regularizer=regularizer,
        name="Conv2d"
    )
    return conv2d


x = np.random.randn(1, 256, 32, 32).astype(np.float32)
out = conv2d_l1_Job(x)
```

oneflow.regularizers.**l1_l2**(*l1: float = 0.01, l2: float = 0.01*) → oneflow.core.job.regularizer_conf_pb2.RegularizerConf

This operator creates a L1 and L2 weight regularizer.

Parameters

- **l1** (*float, optional*) – The L1 regularization coefficient. Defaults to 0.01.

- **l2** (*float, optional*) – The L2 regularization coefficient. Defaults to 0.01.

Returns A regularizer that can be used in other layers or operators.

Return type regularizer_conf_util.RegularizerConf

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def conv2d_l1_l2_Job(x: tp.Numpy.Placeholder((1, 256, 32, 32))
) -> tp.Numpy:
    initializer = flow.truncated_normal(0.1)
    regularizer = flow.regularizers.l1_l2(l1=0.001, l2=0.001)
    conv2d = flow.layers.conv2d(
        x,
        filters=128,
        kernel_size=3,
        strides=1,
        padding='SAME',
        kernel_initializer=initializer,
        kernel_regularizer=regularizer,
        name="Conv2d"
    )
    return conv2d


x = np.random.randn(1, 256, 32, 32).astype(np.float32)
out = conv2d_l1_l2_Job(x)
```

oneflow.regularizers.**l2**(*l: float = 0.01*) → oneflow.core.job.regularizer_conf_pb2.RegularizerConf

This operator creates a L2 weight regularizer.

Parameters **l** (*float, optional*) – The L2 regularization coefficient. Defaults to 0.01.

Returns A regularizer that can be used in other layers or operators.

Return type regularizer_conf_util.RegularizerConf

For example:

```
import oneflow as flow
import numpy as np
import oneflow.typing as tp


@flow.global_function()
def conv2d_l2_Job(x: tp.Numpy.Placeholder((1, 256, 32, 32))
) -> tp.Numpy:
    initializer = flow.truncated_normal(0.1)
    regularizer = flow.regularizers.l2(l=0.001)
    conv2d = flow.layers.conv2d(
        x,
        filters=128,
        kernel_size=3,
        strides=1,
```

(continues on next page)

(continued from previous page)

```
        padding='SAME',
        kernel_initializer=initializer,
        kernel_regularizer=regularizer,
        name="Conv2d"
    )
    return conv2d


x = np.random.randn(1, 256, 32, 32).astype(np.float32)
out = conv2d_l2_Job(x)
```

CHAPTER

TWENTYTHREE

ONEFLOW.IMAGE

23.1 Image processing

oneflow.image.**CropMirrorNormalize**(*input_blob: oneflow_api.BlobDesc, mirror_blob: Optional[oneflow_api.BlobDesc] = None, color_space: str = 'BGR', output_layout: str = 'NCHW', crop_h: int = 0, crop_w: int = 0, crop_pos_y: float = 0.5, crop_pos_x: float = 0.5, mean: Sequence[float] = [0.0], std: Sequence[float] = [1.0], output_dtype: oneflow.python.framework.dtype.dtype = <class 'oneflow.python.framework.dtype.float32'>, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator performs the cropping, normalization, and horizontal flip for input Blob.

If *crop_h* and *crop_w* are provided, the image cropping position is specified by "crop_pos_y" and "crop_pos_x".

The position is computed as follows:

$$crop_x = crop\_pos\_x * (Width - crop\_w)$$
$$crop_y = crop\_pos\_y * (Height - crop\_h)$$

The *Width* and *Height* is the width and height of input Blob.

Parameters

- **input_blob** (*oneflow_api.BlobDesc*) – The input Blob.
- **mirror_blob** (*Optional[oneflow_api.BlobDesc], optional*) – The operation for horizontal flip, if it is *None*, the operator will not perform the horizontal flip. Defaults to None.
- **color_space** (*str, optional*) – The color space for input Blob. Defaults to "BGR".
- **output_layout** (*str, optional*) – The output format. Defaults to "NCHW".
- **crop_h** (*int, optional*) – The image cropping window height. Defaults to 0.
- **crop_w** (*int, optional*) – The image cropping window width. Defaults to 0.
- **crop_pos_y** (*float, optional*) – The vertical position of the image cropping window, the value range is normalized to (0.0, 1.0). Defaults to 0.5.
- **crop_pos_x** (*float, optional*) – The horizontal position of the image cropping window, the value range is normalized to (0.0, 1.0). Defaults to 0.5.
- **mean** (*Sequence[float], optional*) – The mean value for normalization. Defaults to [0.0].

- **std** (*Sequence[float], optional*) – The standard deviation values for normalization. Defaults to [1.0].
- **output_dtype** (*dtype_util.dtype, optional*) – The datatype of output Blob. Defaults to dtype_util.float.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Raises **NotImplementedError** – The data type of input Blob should be *tensor_buffer* or *uint8*

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp
from typing import Tuple


@flow.global_function(type="predict")
def crop_mirror_job() -> Tuple[tp.Numpy, tp.Numpy]:
    batch_size = 1
    color_space = "RGB"
    # our ofrecord file path is "./dataset/part-0"
    ofrecord = flow.data.ofrecord_reader(
        "./imgdataset",
        batch_size=batch_size,
        data_part_num=1,
        part_name_suffix_length=-1,
        part_name_prefix='part-',
        shuffle_after_epoch=True,
    )
    image = flow.data.OFRecordImageDecoder(
            ofrecord, "encoded", color_space=color_space
        )
    res_image, scale, new_size = flow.image.Resize(
            image, target_size=(512, 512)
        )
    label = flow.data.OFRecordRawDecoder(
        ofrecord, "class/label", shape=(1, ), dtype=flow.int32
    )
    rng = flow.random.CoinFlip(batch_size=batch_size)
    normal = flow.image.CropMirrorNormalize(
            res_image,
            mirror_blob=rng,
            color_space=color_space,
            crop_h= 256,
            crop_w= 256,
            crop_pos_y=0.5,
            crop_pos_x=0.5,
            mean=[123.68, 116.779, 103.939],
            std=[58.393, 57.12, 57.375],
            output_dtype=flow.float,
        )

    return normal, label

if __name__ == "__main__":
```

(continues on next page)

(continued from previous page)

```
    images, labels = crop_mirror_job()
    # images.shape (1, 3, 256, 256)
```

oneflow.image.**Resize**(*image: oneflow_api.BlobDesc*, *target_size: Union[int, Sequence[int]] = None*, *min_size: Optional[int] = None*, *max_size: Optional[int] = None*, *keep_aspect_ratio: bool = False*, *resize_side: str = 'shorter'*, *channels: int = 3*, *dtype: Optional[oneflow.python.framework.dtype.dtype] = None*, *interpolation_type: str = 'auto'*, *name: Optional[str] = None*, *color_space: Optional[str] = None*, *interp_type: Optional[str] = None*, *resize_shorter: int = 0*, *resize_x: int = 0*, *resize_y: int = 0*) → Union[oneflow_api.BlobDesc, Sequence[oneflow_api.BlobDesc]]

Resize images to target size.

Parameters

- **image** – A *Tensor* consists of images to be resized.
- **target_size** – A list or tuple when *keep_aspect_ratio* is false or an int when *keep_aspect_ratio* is true. When *keep_aspect_ratio* is false, *target_size* has a form of *(target_width, target_height)* that image will resize to. When *keep_aspect_ratio* is true, the longer side or shorter side of the image will be resized to target size.
- **min_size** – An int, optional. Only works when *keep_aspect_ratio* is true and *resize_side* is "longer". If *min_size* is not None, the shorter side must be greater than or equal to *min_size*. Default is None.
- **max_size** – An int, optional. Only works when *keep_aspect_ratio* is true and *resize_side* is "shorter". If *max_size* is not None, the longer side must be less than or equal to *max_size*. Default is None.
- **keep_aspect_ratio** – A bool. If is false, indicate that image will be resized to fixed width and height, otherwise image will be resized keeping aspect ratio.
- **resize_side** – A str of "longer" or "shorter". Only works when *keep_aspect_ratio* is True. If *resize_side* is "longer", the longer side of image will be resized to *target_size*. If *resize_side* is "shorter", the shorter side of image will be resized to *target_size*.
- **channels** – An int. how many channels an image has
- **dtype** – *oneflow.dtype*. Indicate output resized image data type.
- **interpolation_type** – A str of "auto", "bilinear", "nearest_neighbor", "bicubic" or "area". Indicate interpolation method used to resize image.
- **name** – A str, optional. Name for the operation.
- **color_space** – Deprecated, a str of "RGB", "BGR" or "GRAY". Please use *channels* instead.
- **interp_type** – Deprecated, s str of "Linear", "Cubic" or "NN". Please use *interpolation_type* instead.
- **resize_shorter** – Deprecated, a int. Indicate target size that the shorter side of image will resize to. Please use *target_size* and *resize_side* instead.
- **resize_x** – Deprecated, a int. Indicate the target size that the width of image will resize to. Please use *target_size* instead.
- **resize_y** – Deprecated, a int. Indicate the target size that the height of image will resize to. Please use *target_size* instead.

Returns Tuple of resized images *Blob*, width and height scales *Blob* and new width and height *Blob* (new width and height *Blob* will be None when keep_aspect_ratio is false). If deprecated params are used, a single resized images *Blob* will be returned.

For example:

```
import oneflow as flow
import oneflow.typing as tp
from typing import Tuple


@flow.global_function(type="predict")
def ofrecord_reader_job() -> Tuple[tp.Numpy, tp.Numpy]:
    batch_size = 16
    color_space = "RGB"
    # our ofrecord file path is "./dataset/part-0"
    ofrecord = flow.data.ofrecord_reader(
        "./imgdataset",
        batch_size=batch_size,
        data_part_num=1,
        part_name_suffix_length=-1,
        part_name_prefix='part-',
        random_shuffle=True,
        shuffle_after_epoch=True,
    )
    image = flow.data.OFRecordImageDecoderRandomCrop(
            ofrecord, "encoded", color_space=color_space
        )
    res_image, scale, new_size = flow.image.Resize(
            image, target_size=(224, 224)
        )
    label = flow.data.OFRecordRawDecoder(
        ofrecord, "class/label", shape=(1, ), dtype=flow.int32
    )

    return res_image, label

if __name__ == "__main__":
    images, labels = ofrecord_reader_job()
    # image.shape (16, 224, 224, 3)
```

oneflow.image.**batch_align**(*images: oneflow_api.BlobDesc, shape: Sequence[int], dtype: oneflow.python.framework.dtype.dtype, alignment: int, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator aligns the shape for a batch of images.

The aligned shape is computed as:

$$shape_{width} = int(\frac{(shape_{width} + alignment - 1)}{alignment}) * alignment$$

$$shape_{height} = int(\frac{(shape_{height} + alignment - 1)}{alignment}) * alignment$$

Parameters

- **images** (*oneflow_api.BlobDesc*) – The images.
- **shape** (*Sequence[int]*) – The maximum static shape of input images.
- **dtype** (*dtype_util.dtype*) – The data type.

- **alignment** (*int*) – The align factor.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import cv2
import numpy as np
import oneflow as flow
import oneflow.typing as tp


def _of_image_batch_align(images, input_shape, output_shape, alignment):
    func_config = flow.FunctionConfig()
    func_config.default_data_type(flow.float)
    func_config.default_logical_view(flow.scope.mirrored_view())

    @flow.global_function(function_config=func_config)
    def image_batch_align_job(
        images_def: tp.ListListNumpy.Placeholder(shape=input_shape, dtype=flow.
↪float)
    ) -> tp.ListNumpy:
        # Convert to tensor buffer
        images_buffer = flow.tensor_list_to_tensor_buffer(images_def)
        image = flow.image_batch_align(
            images_buffer, shape=output_shape[1:], dtype=flow.float,␣
↪alignment=alignment
        )
        return image

    image = image_batch_align_job([images])
    return image[0]


def _read_images_by_cv(image_files):
    images = [cv2.imread(image_file).astype(np.single) for image_file in image_
↪files]
    return [np.expand_dims(image, axis=0) for image in images]


def _get_images_static_shape(images):
    image_shapes = [image.shape for image in images]
    image_static_shape = np.amax(image_shapes, axis=0)
    assert isinstance(
        image_static_shape, np.ndarray
    ), "image_shapes: {}, image_static_shape: {}".format(
        str(image_shapes), str(image_static_shape)
    )
    image_static_shape = image_static_shape.tolist()
    assert image_static_shape[0] == 1, str(image_static_shape)
    image_static_shape[0] = len(image_shapes)
    return image_static_shape

def _roundup(x, n):
    # compute the aligned shape
```

(continues on next page)

(continued from previous page)

```
    return int((x + n - 1) / n) * n

if __name__ == "__main__":
    img = _read_images_by_cv(['./img/1.jpg', './img/2.jpg', './img/3.jpg'])
    img_shape = _get_images_static_shape(img) # In example is [3, 349, 367, 3]
    alignment = 16 # alignment factor
    aligned_image_shape = [
        img_shape[0],
        _roundup(img_shape[1], alignment),
        _roundup(img_shape[2], alignment),
        img_shape[3],
    ]
    image = _of_image_batch_align(img, tuple(img_shape), aligned_image_shape,␣
↪alignment)
```

oneflow.image.**crop_mirror_normalize**(*input_blob: oneflow_api.BlobDesc, mirror_blob: Optional[oneflow_api.BlobDesc] = None, color_space: str = 'BGR', output_layout: str = 'NCHW', crop_h: int = 0, crop_w: int = 0, crop_pos_y: float = 0.5, crop_pos_x: float = 0.5, mean: Sequence[float] = [0.0], std: Sequence[float] = [1.0], output_dtype: oneflow.python.framework.dtype.dtype = <class 'oneflow.python.framework.dtype.float32'>, name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator performs the cropping, normalization, and horizontal flip for input Blob.

If *crop_h* and *crop_w* are provided, the image cropping position is specified by "crop_pos_y" and "crop_pos_x".

The position is computed as follows:

$$crop_x = crop\_pos\_x * (Width - crop\_w)$$
$$crop_y = crop\_pos\_y * (Height - crop\_h)$$

The *Width* and *Height* is the width and height of input Blob.

Parameters

- **input_blob** (*oneflow_api.BlobDesc*) – The input Blob.
- **mirror_blob** (*Optional[oneflow_api.BlobDesc], optional*) – The operation for horizontal flip, if it is *None*, the operator will not perform the horizontal flip. Defaults to None.
- **color_space** (*str, optional*) – The color space for input Blob. Defaults to "BGR".
- **output_layout** (*str, optional*) – The output format. Defaults to "NCHW".
- **crop_h** (*int, optional*) – The image cropping window height. Defaults to 0.
- **crop_w** (*int, optional*) – The image cropping window width. Defaults to 0.
- **crop_pos_y** (*float, optional*) – The vertical position of the image cropping window, the value range is normalized to (0.0, 1.0). Defaults to 0.5.
- **crop_pos_x** (*float, optional*) – The horizontal position of the image cropping window, the value range is normalized to (0.0, 1.0). Defaults to 0.5.
- **mean** (*Sequence[float], optional*) – The mean value for normalization. Defaults to [0.0].

- **std** (*Sequence[float], optional*) – The standard deviation values for normalization. Defaults to [1.0].
- **output_dtype** (*dtype_util.dtype, optional*) – The datatype of output Blob. Defaults to dtype_util.float.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Raises **NotImplementedError** – The data type of input Blob should be *tensor_buffer* or *uint8*

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp
from typing import Tuple


@flow.global_function(type="predict")
def crop_mirror_job() -> Tuple[tp.Numpy, tp.Numpy]:
    batch_size = 1
    color_space = "RGB"
    # our ofrecord file path is "./dataset/part-0"
    ofrecord = flow.data.ofrecord_reader(
        "./imgdataset",
        batch_size=batch_size,
        data_part_num=1,
        part_name_suffix_length=-1,
        part_name_prefix='part-',
        shuffle_after_epoch=True,
    )
    image = flow.data.OFRecordImageDecoder(
            ofrecord, "encoded", color_space=color_space
        )
    res_image, scale, new_size = flow.image.Resize(
            image, target_size=(512, 512)
        )
    label = flow.data.OFRecordRawDecoder(
        ofrecord, "class/label", shape=(1, ), dtype=flow.int32
    )
    rng = flow.random.CoinFlip(batch_size=batch_size)
    normal = flow.image.CropMirrorNormalize(
            res_image,
            mirror_blob=rng,
            color_space=color_space,
            crop_h= 256,
            crop_w= 256,
            crop_pos_y=0.5,
            crop_pos_x=0.5,
            mean=[123.68, 116.779, 103.939],
            std=[58.393, 57.12, 57.375],
            output_dtype=flow.float,
        )

    return normal, label

if __name__ == "__main__":
```

(continues on next page)

(continued from previous page)

```
    images, labels = crop_mirror_job()
    # images.shape (1, 3, 256, 256)
```

oneflow.image.**decode**(*images_bytes_buffer: oneflow_api.BlobDesc*, *dtype: oneflow.python.framework.dtype.dtype = <class 'oneflow.python.framework.dtype.uint8'>*, *color_space: str = 'BGR'*, *name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator decode the image.

Parameters

- **images_bytes_buffer** (*oneflow_api.BlobDesc*) – The input Blob. Its type should be *kTensorBuffer*. More details please refer to the code example.
- **dtype** (*dtype_util.dtype, optional*) – The data type. Defaults to dtype_util.uint8.
- **color_space** (*str, optional*) – The color space. Defaults to "BGR".
- **name** (*Optional[str], optional*) – The name for the opreation. Defaults to None.

Returns The decoded image list.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np
from PIL import Image


def _of_image_decode(images):
    image_files = [open(im, "rb") for im in images]
    images_bytes = [imf.read() for imf in image_files]
    static_shape = (len(images_bytes), max([len(bys) for bys in images_bytes]))
    for imf in image_files:
        imf.close()

    func_config = flow.FunctionConfig()
    func_config.default_data_type(flow.float)
    func_config.default_logical_view(flow.scope.mirrored_view())

    @flow.global_function(function_config=func_config)
    def image_decode_job(
        images_def: tp.ListListNumpy.Placeholder(shape=static_shape, dtype=flow.
↪int8)
    )->tp.ListListNumpy:
        # convert to tensor buffer
        images_buffer = flow.tensor_list_to_tensor_buffer(images_def)
        decoded_images_buffer = flow.image_decode(images_buffer)
        # Remember to set a shape
        # convert back to tensor list
        return flow.tensor_buffer_to_tensor_list(
            decoded_images_buffer, shape=(640, 640, 3), dtype=flow.uint8
        )

    images_np_arr = [
```

(continues on next page)

(continued from previous page)

```
        np.frombuffer(bys, dtype=np.byte).reshape(1, -1) for bys in images_bytes
    ]
    decoded_images = image_decode_job([images_np_arr])
    return decoded_images[0]


if __name__ == "__main__":
    img = _of_image_decode(['./img/1.jpg'])
    print(img[0].shape) # Our image shape is (1, 349, 367, 3)
```

oneflow.image.**flip**(*image: oneflow_api.BlobDesc, flip_code: Union[int, oneflow_api.BlobDesc], name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator flips the images.

The flip code corresponds to the different flip mode:

0 (0x00): Non Flip

1 (0x01): Horizontal Flip

16 (0x10): Vertical Flip

17 (0x11): Both Horizontal and Vertical Flip

Parameters

- **image** (*oneflow_api.BlobDesc*) – The input images.
- **flip_code** (*Union[int, oneflow_api.BlobDesc]*) – The flip code.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import cv2
import numpy as np
import oneflow as flow
import oneflow.typing as tp


def _of_image_flip(images, image_shape, flip_code):
    func_config = flow.FunctionConfig()
    func_config.default_data_type(flow.float)
    func_config.default_logical_view(flow.scope.mirrored_view())

    @flow.global_function(function_config=func_config)
    def image_flip_job(
        images_def: tp.ListListNumpy.Placeholder(shape=image_shape, dtype=flow.
↪float)
    ) -> tp.ListListNumpy:
        images_buffer = flow.tensor_list_to_tensor_buffer(images_def)
        flip_images = flow.image_flip(images_buffer, flip_code)
        return flow.tensor_buffer_to_tensor_list(
            flip_images, shape=image_shape[1:], dtype=flow.float
        )

    image_tensor = image_flip_job([images])
```

(continues on next page)

(continued from previous page)

```
    return image_tensor[0]


def _read_images_by_cv(image_files):
    images = [cv2.imread(image_file).astype(np.single) for image_file in image_
↪files]
    return [np.expand_dims(image, axis=0) for image in images]


def _get_images_static_shape(images):
    image_shapes = [image.shape for image in images]
    image_static_shape = np.amax(image_shapes, axis=0)
    assert isinstance(
        image_static_shape, np.ndarray
    ), "image_shapes: {}, image_static_shape: {}".format(
        str(image_shapes), str(image_static_shape)
    )
    image_static_shape = image_static_shape.tolist()
    assert image_static_shape[0] == 1, str(image_static_shape)
    image_static_shape[0] = len(image_shapes)
    return image_static_shape

if __name__ == "__main__":
    img = _read_images_by_cv(['./img/1.jpg', './img/2.jpg', './img/3.jpg'])
    img_shape = _get_images_static_shape(img)  # In example is [3, 349, 367, 3]
    image = _of_image_flip(img,
                   tuple(img_shape),
                   flip_code=1)
```

oneflow.image.**normalize**(*image: oneflow_api.BlobDesc, std: Sequence[float], mean: Sequence[float], name: Optional[str] = None*) → oneflow_api.BlobDesc

This operator normalizes the image.

Parameters

- **image** (*oneflow_api.BlobDesc*) – The input image.
- **std** (*Sequence[float]*) – The standard deviation of the images.
- **mean** (*Sequence[float]*) – The mean value of the images.
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns The result Blob

Return type oneflow_api.BlobDesc

For example:

```
import cv2
import numpy as np
import oneflow as flow
import oneflow.typing as tp


def _of_image_normalize(images, image_shape, std, mean):
    func_config = flow.FunctionConfig()
    func_config.default_data_type(flow.float)
    func_config.default_logical_view(flow.scope.mirrored_view())
```

(continues on next page)

(continued from previous page)

```
    @flow.global_function(function_config=func_config)
    def image_normalize_job(
        images_def: tp.ListListNumpy.Placeholder(shape=image_shape, dtype=flow.
↪float)
    ) -> tp.ListListNumpy:
        # Convert to tensor buffer
        images_buffer = flow.tensor_list_to_tensor_buffer(images_def)
        # Normalize the imagess
        norm_images = flow.image_normalize(images_buffer, std, mean)
        # Convert back to tensor list
        return flow.tensor_buffer_to_tensor_list(
            norm_images, shape=image_shape[1:], dtype=flow.float
        )

    image_tensor = image_normalize_job([images])
    return image_tensor[0]


def _read_images_by_cv(image_files):
    images = [cv2.imread(image_file).astype(np.single) for image_file in image_
↪files]
    return [np.expand_dims(image, axis=0) for image in images]


def _get_images_static_shape(images):
    image_shapes = [image.shape for image in images]
    image_static_shape = np.amax(image_shapes, axis=0)
    assert isinstance(
        image_static_shape, np.ndarray
    ), "image_shapes: {}, image_static_shape: {}".format(
        str(image_shapes), str(image_static_shape)
    )
    image_static_shape = image_static_shape.tolist()
    assert image_static_shape[0] == 1, str(image_static_shape)
    image_static_shape[0] = len(image_shapes)
    return image_static_shape

if __name__ == "__main__":
    img = _read_images_by_cv(['./img/1.jpg', './img/2.jpg', './img/3.jpg'])
    img_shape = _get_images_static_shape(img) # In example is [3, 349, 367, 3]
    image = _of_image_normalize(img,
                                tuple(img_shape),
                                std=(102.9801, 115.9465, 122.7717),
                                mean=(1.0, 1.0, 1.0))
```

oneflow.image.**random_crop**(*input_blob: oneflow_api.BlobDesc*, *num_attempts: int = 10*, *seed: Optional[int] = None*, *random_area: Sequence[float] = None*, *random_aspect_ratio: Sequence[float] = None*, *name: str = 'ImageRandomCrop'*) → oneflow_api.BlobDesc

This operator crops the input image randomly.

Parameters

- **input_blob** (*oneflow_api.BlobDesc*) – The input Blob.
- **num_attempts** (*int, optional*) – The maximum number of random cropping attempts. Defaults to 10.

- **seed** (*Optional[int], optional*) – The random seed. Defaults to None.
- **random_area** (*Sequence[float], optional*) – The random cropping area. Defaults to None.
- **random_aspect_ratio** (*Sequence[float], optional*) – The random scaled ratio. Defaults to None.
- **name** (*str, optional*) – The name for the operation. Defaults to "ImageRandomCrop".

Returns The result Blob.

Return type oneflow_api.BlobDesc

For example:

```
import oneflow as flow
import oneflow.typing as tp
import numpy as np
import cv2


def _read_images_by_cv(image_files):
    images = [cv2.imread(image_file).astype(np.single) for image_file in image_
↪files]
    return [np.expand_dims(image, axis=0) for image in images]


def _get_images_static_shape(images):
    image_shapes = [image.shape for image in images]
    image_static_shape = np.amax(image_shapes, axis=0)
    assert isinstance(
        image_static_shape, np.ndarray
    ), "image_shapes: {}, image_static_shape: {}".format(
        str(image_shapes), str(image_static_shape)
    )
    image_static_shape = image_static_shape.tolist()
    assert image_static_shape[0] == 1, str(image_static_shape)
    image_static_shape[0] = len(image_shapes)
    return image_static_shape

def _of_image_random_crop(images, image_static_shape):
    func_config = flow.FunctionConfig()
    func_config.default_data_type(flow.float)
    func_config.default_logical_view(flow.scope.mirrored_view())

    @flow.global_function(function_config=func_config)
    def image_random_crop_job(images_def: tp.ListListNumpy.
↪Placeholder(shape=image_static_shape, dtype=flow.float)
    ) -> tp.ListListNumpy:
        # The input Blob type should be "kTensorBuffer"
        # So we use oneflow.tensor_list_to_tensor_buffer to convert
        images_buffer = flow.tensor_list_to_tensor_buffer(images_def)
        # Do the random crop
        random_crop_buffer = flow.image.random_crop(
            images_buffer,
            random_area=[0.15, 0.80],
            random_aspect_ratio=[0.75, 1.55],
        )
```

(continues on next page)

(continued from previous page)

```
        # We convert back to "tensorlist" type
        random_crop_images = flow.tensor_buffer_to_tensor_list(
            random_crop_buffer,
            shape=(image_static_shape[1], image_static_shape[2], image_static_
↪shape[-1]),
            dtype=flow.float,
        )
        return random_crop_images

    random_crop_images = image_random_crop_job([images])

    return random_crop_images

if __name__ == "__main__":
    img = _read_images_by_cv(['./img/1.jpg'])
    img_shape = _get_images_static_shape(img) # In example is (1, 234, 346, 3)
    random_crop_images = _of_image_random_crop(img, tuple(img_shape))
    # random_crop_images.shape is (234, 346, 3)
```

oneflow.image.**resize**(*image: oneflow_api.BlobDesc*, *target_size: Union[int, Sequence[int]] = None*, *min_size: Optional[int] = None*, *max_size: Optional[int] = None*, *keep_aspect_ratio: bool = False*, *resize_side: str = 'shorter'*, *channels: int = 3*, *dtype: Optional[oneflow.python.framework.dtype.dtype] = None*, *interpolation_type: str = 'auto'*, *name: Optional[str] = None*, *color_space: Optional[str] = None*, *interp_type: Optional[str] = None*, *resize_shorter: int = 0*, *resize_x: int = 0*, *resize_y: int = 0*) → Union[oneflow_api.BlobDesc, Sequence[oneflow_api.BlobDesc]]

Resize images to target size.

Parameters

- **image** – A *Tensor* consists of images to be resized.
- **target_size** – A list or tuple when *keep_aspect_ratio* is false or an int when *keep_aspect_ratio* is true. When *keep_aspect_ratio* is false, *target_size* has a form of *(target_width, target_height)* that image will resize to. When *keep_aspect_ratio* is true, the longer side or shorter side of the image will be resized to target size.
- **min_size** – An int, optional. Only works when *keep_aspect_ratio* is true and *resize_side* is "longer". If *min_size* is not None, the shorter side must be greater than or equal to *min_size*. Default is None.
- **max_size** – An int, optional. Only works when *keep_aspect_ratio* is true and *resize_side* is "shorter". If *max_size* is not None, the longer side must be less than or equal to *max_size*. Default is None.
- **keep_aspect_ratio** – A bool. If is false, indicate that image will be resized to fixed width and height, otherwise image will be resized keeping aspect ratio.
- **resize_side** – A str of "longer" or "shorter". Only works when *keep_aspect_ratio* is True. If *resize_side* is "longer", the longer side of image will be resized to *target_size*. If *resize_side* is "shorter", the shorter side of image will be resized to *target_size*.
- **channels** – An int. how many channels an image has
- **dtype** – *oneflow.dtype*. Indicate output resized image data type.
- **interpolation_type** – A str of "auto", "bilinear", "nearest_neighbor", "bicubic" or "area". Indicate interpolation method used to resize image.

- **name** – A str, optional. Name for the operation.
- **color_space** – Deprecated, a str of "RGB", "BGR" or "GRAY". Please use *channels* instead.
- **interp_type** – Deprecated, s str of "Linear", "Cubic" or "NN". Please use *interpolation_type* instead.
- **resize_shorter** – Deprecated, a int. Indicate target size that the shorter side of image will resize to. Please use *target_size* and *resize_side* instead.
- **resize_x** – Deprecated, a int. Indicate the target size that the width of image will resize to. Please use *target_size* instead.
- **resize_y** – Deprecated, a int. Indicate the target size that the height of image will resize to. Please use *target_size* instead.

Returns Tuple of resized images *Blob*, width and height scales *Blob* and new width and height *Blob* (new width and height *Blob* will be None when keep_aspect_ratio is false). If deprecated params are used, a single resized images *Blob* will be returned.

For example:

```
import oneflow as flow
import oneflow.typing as tp
from typing import Tuple


@flow.global_function(type="predict")
def ofrecord_reader_job() -> Tuple[tp.Numpy, tp.Numpy]:
    batch_size = 16
    color_space = "RGB"
    # our ofrecord file path is "./dataset/part-0"
    ofrecord = flow.data.ofrecord_reader(
        "./imgdataset",
        batch_size=batch_size,
        data_part_num=1,
        part_name_suffix_length=-1,
        part_name_prefix='part-',
        random_shuffle=True,
        shuffle_after_epoch=True,
    )
    image = flow.data.OFRecordImageDecoderRandomCrop(
            ofrecord, "encoded", color_space=color_space
        )
    res_image, scale, new_size = flow.image.Resize(
            image, target_size=(224, 224)
        )
    label = flow.data.OFRecordRawDecoder(
        ofrecord, "class/label", shape=(1, ), dtype=flow.int32
    )

    return res_image, label

if __name__ == "__main__":
    images, labels = ofrecord_reader_job()
    # image.shape (16, 224, 224, 3)
```

oneflow.image.**target_resize**(*images: oneflow_api.BlobDesc*, *target_size: int*, *min_size: Optional[int] = None*, *max_size: Optional[int] = None*, *resize_side: str = 'shorter'*, *interpolation_type: str = 'auto'*, *name: Optional[str] = None*) → Sequence[oneflow_api.BlobDesc]

This operator resizes image to target size.

Parameters

- **images** (*oneflow_api.BlobDesc*) – The input Blob. Its type should be *kTensorBuffer*. More details please refer to the code example.
- **target_size** (*int*) – An int, the target size.
- **min_size** (*Optional[int], optional*) – If *min_size* is not None, the shorter side must be greater than or equal to *min_size*. Default is None. Defaults to None.
- **max_size** (*Optional[int], optional*) – If *max_size* is not None, the longer side must be less than or equal to *max_size*. Defaults to None.
- **resize_side** (*str, optional*) – A str of "longer" or "shorter". Only works when *keep_aspect_ratio* is True. If *resize_side* is "longer", the longer side of image will be resized to *target_size*. If *resize_side* is "shorter", the shorter side of image will be resized to *target_size*. Defaults to "shorter".
- **interpolation_type** (*str, optional*) – A str of "auto", "bilinear", "nearest_neighbor", "bicubic" or "area". Indicate interpolation method used to resize image. Defaults to "auto".
- **name** (*Optional[str], optional*) – The name for the operation. Defaults to None.

Returns A Sequence includes the result Blob.

Return type Sequence[oneflow_api.BlobDesc]

For example:

```
import oneflow as flow
import oneflow.typing as tp
from typing import Tuple
import numpy as np
import cv2


def _read_images_by_cv(image_files):
    images = [cv2.imread(image_file).astype(np.single) for image_file in image_
↪files]
    return [np.expand_dims(image, axis=0) for image in images]


def _get_images_static_shape(images):
    image_shapes = [image.shape for image in images]
    image_static_shape = np.amax(image_shapes, axis=0)
    assert isinstance(
        image_static_shape, np.ndarray
    ), "image_shapes: {}, image_static_shape: {}".format(
        str(image_shapes), str(image_static_shape)
    )
    image_static_shape = image_static_shape.tolist()
    assert image_static_shape[0] == 1, str(image_static_shape)
    image_static_shape[0] = len(image_shapes)
    return image_static_shape
```

(continues on next page)

(continued from previous page)

```
def _of_image_target_resize(images, image_static_shape, target_size, max_size):
    func_config = flow.FunctionConfig()
    func_config.default_data_type(flow.float)
    func_config.default_logical_view(flow.scope.mirrored_view())

    @flow.global_function(function_config=func_config)
    def image_target_resize_job(images_def: tp.ListListNumpy.
↪Placeholder(shape=image_static_shape, dtype=flow.float)
    ) -> Tuple[tp.ListListNumpy, tp.ListNumpy, tp.ListNumpy]:
        # The input Blob type should be "kTensorBuffer"
        # So we use oneflow.tensor_list_to_tensor_buffer to convert
        images_buffer = flow.tensor_list_to_tensor_buffer(images_def)

        resized_images_buffer, size, scale = flow.image_target_resize(
            images_buffer,
            target_size=target_size,
            max_size=max_size,
            resize_side="shorter",
        )
        # We convert back to "tensorlist" type
        resized_images = flow.tensor_buffer_to_tensor_list(
            resized_images_buffer,
            shape=(target_size, max_size, image_static_shape[-1]),
            dtype=flow.float,
        )
        return resized_images, size, scale

    resized_images, size, scale = image_target_resize_job([images])
    resized_image = resized_images[0]
    size = size[0]
    scale = scale[0]

    return resized_images, size, scale

if __name__ == "__main__":
    img = _read_images_by_cv(['./img/1.jpg'])
    img_shape = _get_images_static_shape(img) # In example is [1, 349, 367, 3]
    target_size = 256
    max_size = 512
    resized_images, size, scale = _of_image_target_resize(img, tuple(img_shape),␣
↪target_size, max_size)
    # Here the shorter side is "349", we resize it to target_size(256)
    # The scale is 256 / 349 = 0.73
    # The longer side will be resized to 367 * scale = 269
    # get the first element from the resized_images (its type is `list.list`)
    print(resized_images[0][0].shape) # (1, 256, 269, 3)
```

CHAPTER

TWENTYFOUR

ONEFLOW.TRAIN

class oneflow.train.**CheckPoint**

Create a *CheckPoint* object to manage checkpoint manually.

__init__() → None

Initialize self. See help(type(self)) for accurate signature.

init() → None

Initialize models by default initializer of op or Job.

load(*path: str*) → None

load a checkpoint from *path* and initialize models.

Parameters **path** – A *string* of path to load checkpoint.

save(*path: str*) → None

save a checkpoint to *path*.

Parameters **path** – A *string* of path to save checkpoint.

class oneflow.train.**SimpleCheckPointManager**(*root_path: str*, *prefix: str = 'snapshot_'*)

SimpleCheckPointManager is a simple automatic checkpoint manager.

Parameters

- **root_path** – root path of snapshot
- **prefix** – prefix of snapshot

__init__(*root_path: str*, *prefix: str = 'snapshot_'*) → None

Initialize self. See help(type(self)) for accurate signature.

initialize_or_restore() → None

latest_checkpoint() → Optional[str]

list_checkpoints() → List[str]

save() → None

CHAPTER

TWENTYFIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z