TensorLayerX Documentation

Release 0.5.6

TensorLayerX contributors

Jul 22, 2022

CONTENTS

Documentation Version: 0.5.6

TensorLayerX is a deep learning library designed for researchers and engineers that is compatible with multiple deep learning frameworks such as TensorFlow, MindSpore and PaddlePaddle, allowing users to run the code on different hardware like Nvidia-GPU and Huawei-Ascend. It provides popular DL and RL modules that can be easily customized and assembled for tackling real-world machine learning problems. More details can be found here.

TensorLayerX is a multi-backend AI framework, which can run on almost all operation systems and AI hardwares, and support hybrid-framework programming. The currently version supports TensorFlow, MindSpore, PaddlePaddle and PyTorch(partial) as the backends.

Note: If you got problem to read the docs online, you could download the repository on TensorLayerX, then go to `/docs/_build/html/index.html` to read the docs offline. The `_build` folder can be generated in `docs` using `make html`.

CHAPTER

ONE

USER GUIDE

The TensorLayerX user guide explains how to install TensorFlow, CUDA and cuDNN, how to build and train neural networks using TensorLayer3, and how to contribute to the library as a developer.

1.1 Installation

TensorLayerX has some prerequisites that need to be installed first, including TensorFlow , MindSpore, PaddlePaddle, PyTorch, numpy and matplotlib. For GPU support CUDA and cuDNN are required.

If you run into any trouble, please check the TensorFlow installation instructions, MindSpore installation instructions, PaddlePaddle installation instructions, PyTorch installation instructions, which cover installing the TensorFlow for a range of operating systems including Mac OX, Linux and Windows, or ask for help on tensorlayer@gmail.com or FAQ.

1.1.1 Install Backend

TensorLayerX supports multiple deep learning backends, default TensorFlow as backend also supports MindSpore, PaddlePaddle and PyTorch.

```
pip3 install tensorflow-gpu==2.0.0-beta1 # specific version  (YOU SHOULD INSTALL THIS␣
↪ONE NOW)
pip3 install tensorflow-gpu # GPU version
pip3 install tensorflow # CPU version
```

The installation instructions of TensorFlow are written to be very detailed on TensorFlow website. However, there are something need to be considered. For example, TensorFlow officially supports GPU acceleration for Linux, Mac OX and Windows at present. For ARM processor architecture, you need to install TensorFlow from source.

If you want to use mindspore backend, you should install mindspore==1.6.1.

```
pip install https://ms-release.obs.cn-north-4.myhuaweicloud.com/1.6.1/MindSpore/gpu/
↪x86_64/cuda-10.1/mindspore_gpu-1.6.1-cp37-cp37m-linux_x86_64.whl --trusted-host ms-
↪release.obs.cn-north-4.myhuaweicloud.com -i https://pypi.tuna.tsinghua.edu.cn/simple
```

If you want to use paddlepaddle backend, you should install paddlepaddle>=2.1.1

```
python -m pip install paddlepaddle -i https://mirror.baidu.com/pypi/simple
```

If you want to use PyTorch backend, you should install PyTorch>=1.8.0

```
pip3 install torch==1.8.2+cu102 torchvision==0.9.2+cu102 torchaudio===0.8.2 -f https:/
↪/download.pytorch.org/whl/lts/1.8/torch_lts.html
```

1.1.2 Install TensorLayerX

For stable version:

```
pip3 install tensorlayerx

pip install tensorlayerx -i https://pypi.tuna.tsinghua.edu.cn/simple  (faster in␣
↪China)
```

For latest version, please install from github.

```
pip3 install git+https://github.com/tensorlayer/TensorLayerX.git
```

For developers, you should clone the folder to your local machine and put it along with your project scripts.

```
git clone https://github.com/tensorlayer/TensorLayerX.git
```

Alternatively, you can build from the source.

```
# First clone the repository and change the current directory to the newly cloned␣
↪repository
git clone https://github.com/tensorlayer/TensorLayerX.git
cd tensorlayer

# Install virtualenv if necessary
sudo pip3 install virtualenv
# Then create a virtualenv called `venv`
virtualenv venv

# Activate the virtualenv

## Linux:
source venv/bin/activate

## Windows:
venv\Scripts\activate.bat

# basic installation
pip3 install .

# ============= IF TENSORFLOW IS NOT ALREADY INSTALLED ============= #

# for a machine **without** an NVIDIA GPU
pip3 install -e ".[all_cpu_dev]"

# for a machine **with** an NVIDIA GPU
pip3 install -e ".[all_gpu_dev]"
```

If you want install TensorLayer 2.X, It does not support multiple backends.

```
[stable version] pip3 install tensorlayer==2.x.x
```

If you want install TensorLayer 1.X, the simplest way to install TensorLayer 1.X is as follow. It will also install the numpy and matplotlib automatically.

```
[stable version] pip3 install tensorlayer==1.x.x
```

However, if you want to modify or extend TensorLayer 1.X, you can download the repository from Github and install it as follow.

```
cd to the root of the git tree
pip3 install -e .
```

This command will run the `setup.py` to install TensorLayerX. The `-e` reflects editable, then you can edit the source code in `tensorlayer` folder, and `import` the edited TensorLayerX.

1.1.3 GPU support

Thanks to NVIDIA supports, training a fully connected network on a GPU, which may be 10 to 20 times faster than training them on a CPU. For convolutional network, may have 50 times faster. This requires an NVIDIA GPU with CUDA and cuDNN support.

CUDA

The TensorFlow website also teach how to install the CUDA and cuDNN, please see TensorFlow GPU Support.

Download and install the latest CUDA is available from NVIDIA website:

- CUDA download and install

If CUDA is set up correctly, the following command should print some GPU information on the terminal:

```
python -c "import tensorflow"
```

cuDNN

Apart from CUDA, NVIDIA also provides a library for common neural network operations that especially speeds up Convolutional Neural Networks (CNNs). Again, it can be obtained from NVIDIA after registering as a developer (it take a while):

Download and install the latest cuDNN is available from NVIDIA website:

- cuDNN download and install

To install it, copy the `*.h` files to `/usr/local/cuda/include` and the `lib*` files to `/usr/local/cuda/lib64`.

1.1.4 Windows User

TensorLayer is built on the top of Python-version TensorFlow, so please install Python first. Note:We highly recommend installing Anaconda. The lowest version requirements of Python is py36.

Anaconda download

GPU support

Thanks to NVIDIA supports, training a fully connected network on a GPU, which may be 10 to 20 times faster than training them on a CPU. For convolutional network, may have 50 times faster. This requires an NVIDIA GPU with CUDA and cuDNN support.

1. Installing Microsoft Visual Studio

You should preinstall Microsoft Visual Studio (VS) before installing CUDA. The lowest version requirements is VS2010. We recommend installing VS2015 or VS2013. CUDA7.5 supports VS2010, VS2012 and VS2013. CUDA8.0 also supports VS2015.

2. Installing CUDA

Download and install the latest CUDA is available from NVIDIA website:

CUDA download

We do not recommend modifying the default installation directory.

3. Installing cuDNN

The NVIDIA CUDA® Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. Download and extract the latest cuDNN is available from NVIDIA website:

cuDNN download

After extracting cuDNN, you will get three folders (bin, lib, include). Then these folders should be copied to CUDA installation. (The default installation directory is *C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v8.0*)

1.2 Examples

We list some examples here, but more tutorials and applications can be found in Github examples.

1.2.1 Commonly used dataset and pretrained models

- MNIST, see OpenI. or MNIST.
- CIFAR10, see OpenI. or CIFAR10.
- YOLOv4 Pretrained Model, see OpenI. or Baidu. password: idsz
- VGG16 Pretrained Model, see OpenI. or Baidu. password: t36u
- VGG19 Pretrained Model, see OpenI. or Baidu. password: rb8w
- ResNet50 Pretrained Model, see OpenI. or Baidu. password: 3nui

1.2.2 Basics

- Multi-layer perceptron (MNIST), simple usage and supports multiple backends. Classification task, see mnist_mlp.py.
- Generative Adversarial Networks (MNIST), simple usage and supports multiple backends. See mnist_gan.py.
- Convolutional Network (CIFAR-10), simple usage and supports multiple backends. Classification task, see cifar10_cnn.py.
- Recurrent Neural Network (IMDB), simple usage and supports multiple backends. Text classification task, see imdb_LSTM_simple.py.

- Using tensorlayerx to automatic inference input shape. See automatic_inference_input _shape.py.
- Using ModuleList in tensorlayerx. See tutorial_ModuleList.py.
- Using Sequential in tensorlayerx. See mnist_Sequential.py.
- Using Dataflow in tensorlayerx. See mnist_dataflow.py.
- Using nested layer in tensorlayerx. See nested_usage_of_layer.py.
- Using tensorlayerx to save tensorflow model to pb. See tensorflow_model_save_to_pb.py.
- Using tensorlayerx to load model from npz. See tutorial_tensorlayer_model_load.py.

1.2.3 Pretrained Models

- VGG 16 (ImageNet). Classification task demo, see pretrained_vgg16. and VGG model, see vgg.py.
- Resnet50 (ImageNet). Classification task demo, see pretrained_resnet50.py. and Resnet model, see resnet.py.
- YOLOv4 (MS-COCO). Object Detection demo, see pretrained_yolov4.py. and YOLOv4 model, see yolo.py.
- All pretrained models in pretrained-models.

1.2.4 Vision

Warning:These examples below only support Tensorlayer 2.0. TensorlayerX is under development.

- Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization, see examples.
- ArcFace: Additive Angular Margin Loss for Deep Face Recognition, see InsignFace.
- BinaryNet. Model compression, see mnist cifar10.
- Ternary Weight Network. Model compression, see mnist cifar10.
- DoReFa-Net. Model compression, see mnist cifar10.
- QuanCNN. Model compression, sees mnist cifar10.
- Wide ResNet (CIFAR) by ritchieng.
- Spatial Transformer Networks by zsdonghao.
- U-Net for brain tumor segmentation by zsdonghao.
- Variational Autoencoder (VAE) for (CelebA) by yzwxx.
- Variational Autoencoder (VAE) for (MNIST) by BUPTLdy.
- Image Captioning - Reimplementation of Google's im2txt by zsdonghao.

1.2.5 Adversarial Learning

Warning:These examples below only support Tensorlayer 2.0. TensorlayerX is under development.

- DCGAN (CelebA). Generating images by Deep Convolutional Generative Adversarial Networks by zsdonghao.
- Generative Adversarial Text to Image Synthesis by zsdonghao.
- Unsupervised Image to Image Translation with Generative Adversarial Networks by zsdonghao.
- Improved CycleGAN with resize-convolution by luoxier.

- Super Resolution GAN by zsdonghao.
- BEGAN: Boundary Equilibrium Generative Adversarial Networks by 2wins.
- DAGAN: Fast Compressed Sensing MRI Reconstruction by nebulaV.

1.2.6 Natural Language Processing

Warning:These examples below only support Tensorlayer 2.0. TensorlayerX is under development.

- Word Embedding (Word2vec). Train a word embedding matrix, see tutorial_word2vec_basic.py.
- Restore Embedding matrix. Restore a pre-train embedding matrix, see tutorial_generate_text.py.
- Text Generation. Generates new text scripts, using LSTM network, see tutorial_generate_text.py.
- Chinese Text Anti-Spam by pakrchen.
- Chatbot in 200 lines of code for Seq2Seq.
- FastText Sentence Classification (IMDB), see tutorial_imdb_fasttext.py by tomtung.

1.2.7 Reinforcement Learning

Warning:These examples below only support Tensorlayer 2.0. TensorlayerX is under development.

- Policy Gradient / Network (Atari Ping Pong), see tutorial_atari_pong.py.
- Deep Q-Network (Frozen lake), see tutorial_frozenlake_dqn.py.
- Q-Table learning algorithm (Frozen lake), see tutorial_frozenlake_q_table.py.
- Asynchronous Policy Gradient using TensorDB (Atari Ping Pong) by nebulaV.
- AC for discrete action space (Cartpole), see tutorial_cartpole_ac.py.
- A3C for continuous action space (Bipedal Walker), see tutorial_bipedalwalker_a3c*.py.
- DAGGER for (Gym Torcs) by zsdonghao.
- TRPO for continuous and discrete action space by jjkke88.

1.2.8 Miscellaneous

Warning:These examples below only support Tensorlayer 2.0. TensorlayerX is under development.

- Sipeed : Run TensorLayer on AI Chips

1.3 Contributing

TensorLayerx is a major ongoing research project in Peking University and Pengcheng Laboratory, the first version was established at Imperial College London in 2016. The goal of the project is to develop a compositional language that is compatible with multiple deep learning frameworks, while complex learning systems can be built through composition of neural network modules.

Numerous contributors come from various horizons such as: Imperial College London, Tsinghua University, Carnegie Mellon University, Stanford, University of Technology of Compiegne, Google, Microsoft, Bloomberg and etc.

You can easily open a Pull Request (PR) on GitHub, every little step counts and will be credited. As an open-source project, we highly welcome and value contributions!

If you are interested in working with us, please contact us at: tensorlayer@gmail.com.

1.3.1 Project Maintainers

The TensorLayer project was started by Hao Dong at Imperial College London in June 2016.

For TensorLayerX, it is now actively developing and maintaining by the following people *(in alphabetical order)*:

- **Cheng Lai** (@Laicheng0830) - https://Laicheng0830.github.io
- **Hao Dong** (@zsdonghao) - https://zsdonghao.github.io
- **Jiarong Han** (@hanjr92) - https://hanjr92.github.io

For TensorLayer 2.x, it is now actively developing and maintaining by the following people who has more than 50 contributions:

- **Hao Dong** (@zsdonghao) - https://zsdonghao.github.io
- **Jingqing Zhang** (@JingqingZ) - https://jingqingz.github.io
- **Rundi Wu** (@ChrisWu1997) - http://chriswu1997.github.io
- **Ruihai Wu** (@warshallrho) - https://warshallrho.github.io/

For TensorLayer 1.x, it was actively developed and maintained by the following people *(in alphabetical order)*:

- **Akara Supratak** (@akaraspt) - https://akaraspt.github.io
- **Fangde Liu** (@fangde) - http://fangde.github.io/
- **Guo Li** (@lgarithm) - https://lgarithm.github.io
- **Hao Dong** (@zsdonghao) - https://zsdonghao.github.io
- **Jonathan Dekhtiar** (@DEKHTIARJonathan) - https://www.jonathandekhtiar.eu
- **Luo Mai** (@luomai) - http://www.doc.ic.ac.uk/~lm111/
- **Pan Wang** (@FerociousPanda) - http://github.com/FerociousPanda (UI)
- **Simiao Yu** (@nebulaV) - https://nebulav.github.io

Numerous other contributors can be found in the Github Contribution Graph.

1.3.2 What to contribute

Your method and example

If you have a new method or example in terms of Deep learning or Reinforcement learning, you are welcome to contribute.

- Provide your layers or examples, so everyone can use it.
- Explain how it would work, and link to a scientific paper if applicable.
- Keep the scope as narrow as possible, to make it easier to implement.

Report bugs

Report bugs at the GitHub or OpenI, we normally will fix it in 5 hours. If you are reporting a bug, please include:

- your TensorLayerX, TensorFlow, MindSpore, PaddlePaddle, PyTorch and Python version.
- steps to reproduce the bug, ideally reduced to a few Python commands.
- the results you obtain, and the results you expected instead.

If you are unsure whether the behavior you experience is a bug, or if you are unsure whether it is related to TensorLayer or TensorFlow, MindSpore, PaddlePaddle, PyTorch, please just ask on our mailing list first.

Fix bugs

Look through the GitHub issues or OpenI issues for bug reports. Anything tagged with "bug" is open to whoever wants to implement it. If you discover a bug in TensorLayerX you can fix yourself, by all means feel free to just implement a fix and not report it first.

Write documentation

Whenever you find something not explained well, misleading, glossed over or just wrong, please update it! The *Edit on Github* link on the top right of every documentation page and the *[source]* link for every documented entity in the API reference will help you to quickly locate the origin of any text.

1.3.3 How to contribute

Edit on Github

As a very easy way of just fixing issues in the documentation, use the *Edit on Github* link on the top right of a documentation page or the *[source]* link of an entity in the API reference to open the corresponding source file in Github, then click the *Edit this file* link to edit the file in your browser and send us a Pull Request. All you need for this is a free Github account.

For any more substantial changes, please follow the steps below to setup TensorLayerX for development.

Documentation

The documentation is generated with Sphinx. To build it locally, run the following commands:

```
pip install Sphinx
sphinx-quickstart

cd docs
make html
```

If you want to re-generate the whole docs, run the following commands:

```
cd docs
make clean
make html
```

To write the docs, we recommend you to install Local RTD VM.

Afterwards, open `docs/_build/html/index.html` to view the documentation as it would appear on readthedocs. If you changed a lot and seem to get misleading error messages or warnings, run `make clean html` to force Sphinx to recreate all files from scratch.

When writing docstrings, follow existing documentation as much as possible to ensure consistency throughout the library. For additional information on the syntax and conventions used, please refer to the following documents:

- reStructuredText Primer
- Sphinx reST markup constructs
- A Guide to NumPy/SciPy Documentation

Testing

TensorLayerX has a code coverage of 100%, which has proven very helpful in the past, but also creates some duties:

- Whenever you change any code, you should test whether it breaks existing features by just running the test scripts.
- Every bug you fix indicates a missing test case, so a proposed bug fix should come with a new test that fails without your fix.

Sending Pull Requests

When you're satisfied with your addition, the tests pass and the documentation looks good without any markup errors, commit your changes to a new branch, push that branch to your fork and send us a Pull Request via Github's web interface.

When filing your Pull Request, please include a description of what it does, to help us reviewing it. If it is fixing an open issue, say, issue #123, add *Fixes #123*, *Resolves #123* or *Closes #123* to the description text, so Github will close it when your request is merged.

1.4 Get Involved in Research

1.4.1 Ph.D. Postition @ PKU

Hi, I am Hao Dong, the founder of this project and a new faculty member in EECS, Peking University. I now have a few Ph.D. positions per year open for international students who would like to study AI. If you or your friends are interested in it, feel free to contact me. PKU is a top 30 university in the global ranking. The application is competitive, apply early is recommended. Please check the following links for more details.

- About the International Elite Ph.D. Program in Computer Science
- My homepage

Contact: hao.dong [AT] pku.edu.cn

1.4.2 Faculty Postition @ PKU

The Center on Frontiers of Computing Studies (CFCS), Peking University (PKU), China, is a university new initiative co-founded by Professors John Hopcroft (Turing Awardee) and Wen Gao (CAE, ACM/IEEE Fellow). The center aims at developing the excellence on two fronts: research and education. On the research front, the center will

provide a world-class research environment, where innovation and impactful research is the central aim, measured by professional reputation among world scholars, not by counting the number of publications and research funding. On the education front, the center deeply involves in the Turing Class, an elite undergraduate program that draws the cream of the crop from the PKU undergraduate talent pool. New curriculum and pedagogy are designed and practiced in this program, with the aim to cultivate a new generation of computer scientist/engineers that are solid in both theories and practices.

Positions and Qualification

The center invites applications for tenured/tenure-track faculty positions. We are seeking applicants from all areas of Computer Science, spanning theoretical foundations, systems, software, and applications, with special interests in artificial intelligence and machine learning. We are especially interested in applicants conducting research at the frontiers of Computer Science with other disciplines, such as data sciences, engineering, as well as mathematical, medical, physical, and social sciences.

Applicants are expected to have completed (or be completing) a Ph.D., have demonstrated the ability to pursue a program of research, and have a strong commitment to undergraduate and graduate teaching. A successful candidate will be expected to teach one to two courses at the undergraduate and graduate levels in each semester, and to build and lead a team of undergraduate and graduate students in innovative research.

We are also seeking qualified candidates for postdoctoral positions. Candidates should have a Ph.D. in a relevant discipline or expect a Ph. D within a year, with a substantive record of research accomplishments, and the ability to work collaboratively with faculty members in the center.

To Apply

Applicants should send a full curriculum vitae; copies of 3-5 key publications; 3-5 names and contact information of references; and a statement of research and teaching to: CFCS_recruiting[at]pku[dot]edu[dot]cn . To expedite the process, please arrange to have the reference letters sent directly to the above email address.

Application for a postdoctoral position should include a curriculum vita, brief statement of research, and three to five names and contact information of recommendation, and can be directly addressed to an individual faculty member.

We conduct review of applications monthly, immediately upon the recipient of all application materials at the beginning of each month. However, it is highly recommended that applicants submit complete applications sooner than later, as the positions are to be filled quickly.

1.4.3 Postdoc Postition @ ICL

Data science is therefore by nature at the core of all modern transdisciplinary scientific activities, as it involves the whole life cycle of data, from acquisition and exploration to analysis and communication of the results. Data science is not only concerned with the tools and methods to obtain, manage and analyse data: it is also about extracting value from data and translating it from asset to product.

Launched on 1st April 2014, the Data Science Institute (DSI) at Imperial College London aims to enhance Imperial's excellence in data-driven research across its faculties by fulfilling the following objectives.

The Data Science Institute is housed in purpose built facilities in the heart of the Imperial College campus in South Kensington. Such a central location provides excellent access to collabroators across the College and across London.

- To act as a focal point for coordinating data science research at Imperial College by facilitating access to funding, engaging with global partners, and stimulating cross-disciplinary collaboration.
- To develop data management and analysis technologies and services for supporting data driven research in the College.
- To promote the training and education of the new generation of data scientist by developing and coordinating new degree courses, and conducting public outreach programmes on data science.
- To advise College on data strategy and policy by providing world-class data science expertise.

- To enable the translation of data science innovation by close collaboration with industry and supporting commercialization.

If you are interested in working with us, please check our vacancies and other ways to get involved , or feel free to contact us.

1.4.4 Software Engineer @ SurgicalAI.cn

SurgicalAI is a startup founded by the data scientists and surgical robot experts from Imperial College. Our objective is AI democratise Surgery. By combining 5G, AI and Cloud Computing, SurgicalAI is building a platform enable junor surgeons to perfom complex procedures. As one of the most impactful startup, SurgicalAI is supported by Nvidia, AWS and top surgeons around the world.

Currently based in Hangzhou, China, we are building digital solution for cardiac surgery like TAVR, LAA and Orthopedidcs like TKA and UNA. A demo can be found at here <http://demo5g.surgicalai.cn>

We are activly looking for experts in robotic navigation, computer graphics and medical image analysis experts to join us, building digitalized surgical service platform for the aging world.

Home Page: http://www.surgicalai.cn

Demo Page: http://demo5g.surgicalai.cn

Contact: liufangde@surgicalai.cn

1.5 FAQ

1.5.1 How to effectively learn TensorLayerX

No matter what stage you are in, we recommend you to spend just 10 minutes to read the source code of TensorLayerX and the Understand layer / Your layer in this website, you will find the abstract methods are very simple for everyone. Reading the source codes helps you to better understand TensorFlow, MindSpore, PaddlePaddle and allows you to implement your own methods easily. For discussion, we recommend Gitter, Help Wanted Issues, QQ group and Wechat group.

Beginner

For people who new to deep learning, the contributors provided a number of tutorials in this website, these tutorials will guide you to understand convolutional neural network, recurrent neural network, generative adversarial networks and etc. If your already understand the basic of deep learning, we recommend you to skip the tutorials and read the example codes on Github , then implement an example from scratch.

Engineer

For people from industry, the contributors provided mass format-consistent examples covering computer vision, natural language processing and reinforcement learning. Besides, there are also many TensorFlow users already implemented product-level examples including image captioning, semantic/instance segmentation, machine translation, chatbot and etc., which can be found online. It is worth noting that a wrapper especially for computer vision Tf-Slim can be connected with TensorLayerX seamlessly. Therefore, you may able to find the examples that can be used in your project.

Researcher

For people from academia, TensorLayerX was originally developed by PhD students who facing issues with other libraries on implement novel algorithm. Installing TensorLayer in editable mode is recommended, so you can extend your methods in TensorLayerX. For research related to image processing such as image captioning, visual QA and etc., you may find it is very helpful to use the existing Tf-Slim pre-trained models with TensorLayerX (a specially layer for connecting Tf-Slim is provided).

1.5.2 Install Master Version

To use all new features of TensorLayerX, you need to install the master version from Github. Before that, you need to make sure you already installed git.

```
[stable version] pip3 install tensorlayerX
[master version] pip3 install git+https://github.com/tensorlayer/TensorLayerX.git
```

1.5.3 Editable Mode

- 1. Download the TensorLayerX folder from OpenI.
- 2. Before editing the TensorLayerX `.py` file.
  - If your script and TensorLayerX folder are in the same folder, when you edit the `.py` inside TensorLayerX folder, your script can access the new features.
  - If your script and TensorLayerX folder are not in the same folder, you need to run the following command in the folder contains `setup.py` before you edit `.py` inside TensorLayerX folder.

```
pip install -e .
```

1.5.4 Load Model

Note that, the `tl.files.load_npz()` can only able to load the npz model saved by `tl.files.save_npz()`. If you have a model want to load into your TensorLayerX network, you can first assign your parameters into a list in order, then use `tl.files.assign_params()` to load the parameters into your TensorLayerX model.

1.6 Define a model

TensorLayerX provides two ways to define a model. Sequential model allows you to build model in a fluent way while dynamic model allows you to fully control the forward process.

1.6.1 Sequential model

```
from tensorlayerx.nn import Sequential
from tensorlayerx.nn import Linear
import tensorlayerx as tlx

def get_model():
    layer_list = []
```

(continues on next page)

(continued from previous page)

```
    layer_list.append(Linear(out_features=800, act=tlx.nn.ReLU, in_features=784, name=
↪'Linear1'))
    layer_list.append(Linear(out_features=800, act=tlx.nn.ReLU, in_features=800, name=
↪'Linear2'))
    layer_list.append(Linear(out_features=10, act=tlx.nn.ReLU, in_features=800, name=
↪'Linear3'))
    MLP = Sequential(layer_list)
    return MLP
```

1.6.2 Dynamic model

In this case, you need to manually input the output shape of the previous layer to the new layer.

```
import tensorlayerx as tlx
from tensorlayerx.nn import Module
from tensorlayerx.nn import Dropout, Linear
class CustomModel(Module):

    def __init__(self):
        super(CustomModel, self).__init__()

        self.dropout1 = Dropout(p=0.2)
        self.linear1 = Linear(out_features=800, act=tlx.nn.ReLU, in_features=784)
        self.dropout2 = Dropout(p=0.2)
        self.linear2 = Linear(out_features=800, act=tlx.nn.ReLU, in_features=800)
        self.dropout3 = Dropout(p=0.2)
        self.linear3 = Linear(out_features=10, act=None, in_features=800)

    def forward(self, x, foo=False):
        z = self.dropout1(x)
        z = self.linear1(z)
        z = self.dropout2(z)
        z = self.linear2(z)
        z = self.dropout3(z)
        out = self.linear3(z)
        if foo:
            out = tlx.softmax(out)
        return out

MLP = CustomModel()
MLP.set_eval()
outputs = MLP(data, foo=True) # controls the forward here
outputs = MLP(data, foo=False)
```

1.6.3 Dynamic model do not manually input the output shape

In this case, you do not manually input the output shape of the previous layer to the new layer.

```
import tensorlayerx as tlx
from tensorlayerx.nn import Module
from tensorlayerx.nn import Dropout, Linear
class CustomModel(Module):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        super(CustomModel, self).__init__()

        self.dropout1 = Dropout(p=0.2)
        self.linear1 = Linear(out_features=800, act=tlx.nn.ReLU)
        self.dropout2 = Dropout(p=0.2)
        self.linear2 = Linear(out_features=800, act=tlx.nn.ReLU)
        self.dropout3 = Dropout(p=0.2)
        self.linear3 = Linear(out_features=10, act=None)

    def forward(self, x, foo=False):
        z = self.dropout1(x)
        z = self.linear1(z)
        z = self.dropout2(z)
        z = self.linear2(z)
        z = self.dropout3(z)
        out = self.linear3(z)
        if foo:
            out = tlx.softmax(out)
        return out

MLP = CustomModel()
MLP.init_build(tlx.nn.Input(shape=(1, 784))) # init_build must be called to␣
↪initialize the weights.
MLP.set_eval()
outputs = MLP(data, foo=True) # controls the forward here
outputs = MLP(data, foo=False)
```

1.6.4 Switching train/test modes

```
# method 1: switch before forward
MLP.set_train() # enable dropout, batch norm moving avg ...
output = MLP(train_data)
... # training code here
Model.set_eval()  # disable dropout, batch norm moving avg ...
output = MLP(test_data)
... # testing code here

# method 2: Using packaged training modules
model = tlx.model.Model(network=MLP, loss_fn=tlx.losses.softmax_cross_entropy_with_
↪logits, optimizer=optimizer)
model.train(n_epoch=n_epoch, train_dataset=train_ds)
```

1.6.5 Reuse weights

For dynamic model, call the layer multiple time in forward function

```
import tensorlayerx as tlx
from tensorlayerx.nn import Module, Linear, Concat
class MyModel(Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.linear_shared = Linear(out_features=800, act=tlx.nn.ReLU, in_
↪features=784)
```

(continues on next page)

(continued from previous page)

```
        self.linear1 = Linear(out_features=10, act=tlx.nn.ReLU, in_features=800)
        self.linear2 = Linear(out_features=10, act=tlx.nn.ReLU, in_features=800)
        self.cat = Concat()

    def forward(self, x):
        x1 = self.linear_shared(x) # call dense_shared twice
        x2 = self.linear_shared(x)
        x1 = self.linear1(x1)
        x2 = self.linear2(x2)
        out = self.cat([x1, x2])
        return out

model = MyModel()
```

1.6.6 Print model information

```
print(MLP) # simply call print function

# Model(
#   (_inputlayer): Input(shape=[None, 784], name='_inputlayer')
#   (dropout): Dropout(p=0.8, name='dropout')
#   (linear): Linear(out_features=800, relu, in_features='784', name='linear')
#   (dropout_1): Dropout(p=0.8, name='dropout_1')
#   (linear_1): Linear(out_features=800, relu, in_features='800', name='linear_1')
#   (dropout_2): Dropout(p=0.8, name='dropout_2')
#   (linear_2): Linear(out_features=10, None, in_features='800', name='linear_2')
# )
```

1.6.7 Get specific weights

We can get the specific weights by indexing or naming.

```
# indexing
all_weights = MLP.all_weights
some_weights = MLP.all_weights[1:3]
```

1.6.8 Save and restore model

We provide two ways to save and restore models

Save weights only

```
MLP.save_weights('./model_weights.npz') # by default, file will be in hdf5 format
MLP.load_weights('./model_weights.npz')
```

Save model weights (optional)

```
# When using packaged training modules. Saving and loading the model can be done as
↪follows
model = tlx.model.Model(network=MLP, loss_fn=tlx.losses.softmax_cross_entropy_with_
↪logits, optimizer=optimizer)
model.train(n_epoch=n_epoch, train_dataset=train_ds)
model.save_weights('./model.npz', format='npz_dict')
model.load_weights('./model.npz', format='npz_dict')
```

1.7 Advanced features

1.7.1 Customizing layer

Layers with weights

The fully-connected layer is $a = f(x*W+b)$, the most simple implementation is as follow.

```
import tensorlayerx as tlx
from tensorlayerx.nn import Module

class Linear(Module):
  """The :class:`Linear` class is a fully connected layer.

  Parameters
  ----------
  out_features : int
      The number of units of this layer.
  act : activation function
      The activation function of this layer.
  W_init : initializer or str
      The initializer for the weight matrix.
  b_init : initializer or None or str
      The initializer for the bias vector. If None, skip biases.
  in_features: int
      The number of channels of the previous layer.
      If None, it will be automatically detected when the layer is forwarded for the
↪first time.
  name : None or str
      A unique layer name. If None, a unique name will be automatically generated.
  """

  def __init__(
      self,
      out_features,
      act=None,
      W_init='truncated_normal',
      b_init='constant',
      in_features=None,
      name=None,  # 'linear',
  ):
      super(Linear, self).__init__(name, act=act) # auto naming, linear_1, linear_2 ..
↪.
      self.out_features = out_features
      self.in_features = in_features
      self.W_init = self.str_to_init(W_init)
```

(continues on next page)

(continued from previous page)

```
        self.b_init = self.str_to_init(b_init)
        self.build()
        self._built = True

    def build(self): # initialize the model weights here
        shape = [self.in_features, self.out_features]
        self.W = self._get_weights("weights", shape=tuple(shape), init=self.W_init)
        self.b = self._get_weights("biases", shape=(self.out_features, ), init=self.b_
↪init)

    def forward(self, inputs): # call function
        z = tlx.matmul(inputs, self.W) + self.b
        if self.act: # is not None
            z = self.act(z)
        return z
```

The full implementation is as follow, which supports both automatic inference input and dynamic models and allows users to control whether to use the bias, how to initialize the weight values.

```
class Linear(Module):
    """The :class:`Linear` class is a fully connected layer.

    Parameters
    ----------
    out_features : int
        The number of units of this layer.
    act : activation function
        The activation function of this layer.
    W_init : initializer or str
        The initializer for the weight matrix.
    b_init : initializer or None or str
        The initializer for the bias vector. If None, skip biases.
    in_features: int
        The number of channels of the previous layer.
        If None, it will be automatically detected when the layer is forwarded for the␣
↪first time.
    name : None or str
        A unique layer name. If None, a unique name will be automatically generated.

    Examples
    --------
    With TensorLayerx

    >>> net = tlx.nn.Input([100, 50], name='input')
    >>> linear = tlx.nn.Linear(out_features=800, act=tlx.nn.ReLU, in_features=50, name=
↪'linear_1')
    >>> tensor = tlx.nn.Linear(out_features=800, act=tlx.nn.ReLU, name='linear_2')(net)

    Notes
    -----
    If the layer input has more than two axes, it needs to be flatten by using␣
↪:class:`Flatten`.

    """

    def __init__(
```

(continues on next page)

(continued from previous page)

```
        self,
        out_features,
        act=None,
        W_init='truncated_normal',
        b_init='constant',
        in_features=None,
        name=None,  # 'linear',
    ):

        super(Linear, self).__init__(name, act=act)

        self.out_features = out_features
        self.W_init = self.str_to_init(W_init)
        self.b_init = self.str_to_init(b_init)
        self.in_features = in_features

        if self.in_features is not None:
            self.build(self.in_features)
            self._built = True

        logging.info(
            "Linear  %s: %d %s" %
            (self.name, self.out_features, self.act.__class__.__name__ if self.act is␣
↪not None else 'No Activation')
        )

    def __repr__(self):
        actstr = self.act.__class__.__name__ if self.act is not None else 'No Activation
↪'
        s = ('{classname}(out_features={out_features}, ' + actstr)
        if self.in_features is not None:
            s += ', in_features=\'{in_features}\''
        if self.name is not None:
            s += ', name=\'{name}\''
        s += ')'
        return s.format(classname=self.__class__.__name__, **self.__dict__)

    def build(self, inputs_shape):
        if self.in_features is None and len(inputs_shape) < 2:
            raise AssertionError("The dimension of input should not be less than 2")
        if self.in_features:
            shape = [self.in_features, self.out_features]
        else:
            self.in_features = inputs_shape[-1]
            shape = [self.in_features, self.out_features]

        self.W = self._get_weights("weights", shape=tuple(shape), init=self.W_init)

        self.b_init_flag = False
        if self.b_init:
            self.b = self._get_weights("biases", shape=(self.out_features, ), init=self.
↪b_init)
            self.b_init_flag = True
            self.bias_add = tlx.ops.BiasAdd(data_format='NHWC')

        self.act_init_flag = False
        if self.act:
```

(continues on next page)

(continued from previous page)

```
            self.act_init_flag = True

        self.matmul = tlx.ops.MatMul()

    def forward(self, inputs):
        if self._forward_state == False:
            if self._built == False:
                self.build(tlx.get_tensor_shape(inputs))
                self._built = True
            self._forward_state = True

        z = self.matmul(inputs, self.W)
        if self.b_init_flag:
            z = self.bias_add(z, self.b)
        if self.act_init_flag:
            z = self.act(z)
        return z
```

Layers with train/test modes

We use Dropout as an example here:

```
class Dropout(Module):
    """
    The :class:`Dropout` class is a noise layer which randomly set some
    activations to zero according to a probability.

    Parameters
    ----------
    p : float
        probability of an element to be zeroed. Default: 0.5
    seed : int or None
        The seed for random dropout.
    name : None or str
        A unique layer name.

    Examples
    --------
    >>> net = tlx.nn.Input([10, 200])
    >>> net = tlx.nn.Dropout(p=0.2)(net)

    """

    def __init__(self, p=0.5, seed=0, name=None):  #"dropout"):
        super(Dropout, self).__init__(name)
        self.p = p
        self.seed = seed

        self.build()
        self._built = True

        logging.info("Dropout %s: p: %f " % (self.name, self.p))

    def __repr__(self):
        s = ('{classname}(p={p}')
```

(continues on next page)

(continued from previous page)

```
        if self.name is not None:
            s += ', name=\'{name}\''
        s += ')'
        return s.format(classname=self.__class__.__name__, **self.__dict__)

    def build(self, inputs_shape=None):
        self.dropout = tlx.ops.Dropout(p=self.p, seed=self.seed)

    def forward(self, inputs):
        if self.is_train:
            outputs = self.dropout(inputs)
        else:
            outputs = inputs
        return outputs
```

1.7.2 Pre-trained CNN

Get entire CNN

```
import tensorlayerx as tlx
import numpy as np
from tensorlayerx.models.imagenet_classes import class_names
from examples.model_zoo import vgg16

vgg = vgg16(pretrained=True)
img = tlx.vision.load_image('data/tiger.jpeg')
img = tlx.utils.functional.resize(img, (224, 224), method='bilinear')
img = tlx.ops.convert_to_tensor(img, dtype = 'float32') / 255.
output = vgg(img, is_train=False)
```

CHAPTER

TWO

API REFERENCE

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

2.1 API - Activations

To make TensorLayerX simple, we minimize the number of activation functions as much as we can. So we encourage you to use Customizes activation function. For parametric activation, please read the layer APIs.

2.1.1 Your activation

Customizes activation function in TensorLayerX is very easy. The following example implements an activation that multiplies its input by 2.

```
from tensorlayerx.nn import Module
class DoubleActivation(Module):
  def __init__(self):
      pass
  def forward(self, x):
      return x * 2
double_activation = DoubleActivation()
```

For more complex activation, TensorFlow(MindSpore, PaddlePaddle, PyTorch) API will be required.

2.1.2 activation list

ELU([alpha])	Applies the element-wise function:
PRelu([num_parameters, init, data_format, name])	Applies the element-wise function:
PRelu6([channel_shared, in_channels, . . .])	The *PRelu6* class is Parametric Rectified Linear layer integrating ReLU6 behaviour.
PTRelu6([channel_shared, in_channels, . . .])	The *PTRelu6* class is Parametric Rectified Linear layer integrating ReLU6 behaviour.
ReLU()	This function is ReLU.
ReLU6()	This function is ReLU6.
Softplus()	This function is Softplus.
LeakyReLU([negative_slope])	Applies the element-wise function:

Continued on next page

Table 1 – continued from previous page

LeakyReLU6([alpha])	This activation function is a modified version `leaky_relu()` introduced by the following paper: Rectifier Nonlinearities Improve Neural Network Acoustic Models [A.L.Maas et al., 2013].
LeakyTwiceRelu6([alpha_low, alpha_high])	This activation function is a modified version `leaky_relu()` introduced by the following paper: Rectifier Nonlinearities Improve Neural Network Acoustic Models [A.L.Maas et al., 2013].
Ramp([v_min, v_max])	Ramp activation function.
Swish()	Swish function.
HardTanh()	Hard tanh activation function.
Tanh()	Applies the Hyperbolic Tangent (Tanh) function element-wise.
Sigmoid()	Computes sigmoid of x element-wise.
Softmax([axis])	Applies the Softmax function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range [0,1] and sum to 1.
Mish()	Applies the Mish function, element-wise.

2.1.3 TensorLayerX Activations

2.1.4 ELU

class tensorlayerx.nn.activation.**ELU**(*alpha=1.0*)

Applies the element-wise function:

$$\text{ELU}(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha * (\exp(x) - 1), & \text{if } x \leq 0 \end{cases}$$

Parameters

- **alpha** (*float*) – the α value for the ELU formulation. Default: 1.0
- **name** (*str*) – The function name (optional).

Returns A `Tensor` in the same type as `x`.

Return type Tensor

Examples

```
>>> net = tlx.nn.Input([10, 200])
>>> net = tlx.nn.ELU(alpha=0.5)(net)
```

2.1.5 PRelu

class tensorlayerx.nn.activation.**PRelu**(*num_parameters=1*, *init=0.25*, *data_format='channels_last'*, *name=None*)

Applies the element-wise function:

$$\text{PReLU}(x) = \max(0, x) + a * \min(0, x)$$

Parameters

- **num_parameters** (*int*) – number of *a* to learn. 1, or the number of channels at input. Default: 1
- **init** (*float*) – the initial value of *a*. Default: 0.25
- **data_format** (*str*) – Data format that specifies the layout of input. It may be 'channels_last' or 'channels_first'. Default is 'channels_last'.
- **name** (*None or str*) – A unique layer name.

Examples

```
>>> inputs = tlx.nn.Input([10, 5, 10])
>>> prelulayer = tlx.nn.PRelu(num_parameters=5, init=0.25, data_format='channels_
↪first')(inputs)
```

References

- Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification
- Convolutional Deep Belief Networks on CIFAR-10 [A. Krizhevsky, 2010]

2.1.6 PRelu6

class tensorlayerx.nn.activation.**PRelu6**(*channel_shared=False*, *in_channels=None*, *a_init='truncated_normal'*, *name=None*, *data_format='channels_last'*, *dim=2*)

The *PRelu6* class is Parametric Rectified Linear layer integrating ReLU6 behaviour.

This activation layer use a modified version `tlx.nn.LeakyReLU()` introduced by the following paper: Rectifier Nonlinearities Improve Neural Network Acoustic Models [A. L. Maas et al., 2013]

This activation function also use a modified version of the activation function `tf.nn.relu6()` introduced by the following paper: Convolutional Deep Belief Networks on CIFAR-10 [A. Krizhevsky, 2010]

This activation layer push further the logic by adding *leaky* behaviour both below zero and above six.

The function return the following results:

- When x < 0: `f(x) = alpha_low * x`.
- When x in [0, 6]: `f(x) = x`.
- When x > 6: `f(x) = 6`.

Parameters

- **channel_shared** (*boolean*) – If True, single weight is shared by all channels.
- **in_channels** (*int*) – The number of channels of the previous layer. If None, it will be automatically detected when the layer is forwarded for the first time.
- **a_init** (*initializer or str*) – The initializer for initializing the alpha(s).
- **name** (*None or str*) – A unique layer name.

Examples

```
>>> inputs = tlx.nn.Input([10, 5])
>>> prelulayer = tlx.nn.PRelu6(channel_shared=True, in_channels=5)(inputs)
```

References

- Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification
- Rectifier Nonlinearities Improve Neural Network Acoustic Models [A. L. Maas et al., 2013]
- Convolutional Deep Belief Networks on CIFAR-10 [A. Krizhevsky, 2010]

2.1.7 PTRelu6

class tensorlayerx.nn.activation.**PTRelu6**(*channel_shared=False*, *in_channels=None*, *data_format='channels_last'*, *a_init='truncated_normal'*, *name=None*)

The *PTRelu6* class is Parametric Rectified Linear layer integrating ReLU6 behaviour.

This activation layer use a modified version `tlx.nn.LeakyReLU()` introduced by the following paper: Rectifier Nonlinearities Improve Neural Network Acoustic Models [A. L. Maas et al., 2013]

This activation function also use a modified version of the activation function `tf.nn.relu6()` introduced by the following paper: Convolutional Deep Belief Networks on CIFAR-10 [A. Krizhevsky, 2010]

This activation layer push further the logic by adding *leaky* behaviour both below zero and above six.

The function return the following results:

- When x < 0: `f(x) = alpha_low * x`.
- When x in [0, 6]: `f(x) = x`.
- When x > 6: `f(x) = 6 + (alpha_high * (x-6))`.

This version goes one step beyond *PRelu6* by introducing leaky behaviour on the positive side when x > 6.

Parameters

- **channel_shared** (*boolean*) – If True, single weight is shared by all channels.
- **in_channels** (*int*) – The number of channels of the previous layer. If None, it will be automatically detected when the layer is forwarded for the first time.
- **a_init** (*initializer or str*) – The initializer for initializing the alpha(s).
- **name** (*None or str*) – A unique layer name.

Examples

```
>>> inputs = tlx.nn.Input([10, 5])
>>> prelulayer = tlx.nn.PTRelu6(channel_shared=True, in_channels=5)(inputs)
```

References

- Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification
- Convolutional Deep Belief Networks on CIFAR-10 [A. Krizhevsky, 2010]
- Rectifier Nonlinearities Improve Neural Network Acoustic Models [A. L. Maas et al., 2013]

2.1.8 ReLU

class tensorlayerx.nn.activation.**ReLU**

This function is ReLU.

The function return the following results:

- When x < 0: `f(x) = 0`.
- When x >= 0: `f(x) = x`.

Parameters **x** (*Tensor*) – Support input type `float`, `double`, `int32`, `int64`, `uint8`, `int16`, or `int8`.

Examples

```
>>> net = tlx.nn.Input([10, 200])
>>> net = tlx.nn.ReLU()(net)
```

Returns A `Tensor` in the same type as `x`.

Return type Tensor

2.1.9 ReLU6

class tensorlayerx.nn.activation.**ReLU6**

This function is ReLU6.

The function return the following results:

- ReLU6(x)=min(max(0,x),6)

Parameters **x** (*Tensor*) – Support input type `float`, `double`, `int32`, `int64`, `uint8`, `int16`, or `int8`.

Examples

```
>>> net = tlx.nn.Input([10, 200])
>>> net = tlx.nn.ReLU6()(net)
```

Returns A `Tensor` in the same type as `x`.

Return type Tensor

2.1.10 Softplus

class tensorlayerx.nn.activation.**Softplus**
This function is Softplus.

The function return the following results:

- softplus(x) = log(exp(x) + 1).

Parameters **x** (*Tensor*) – Support input type `float`, `double`, `int32`, `int64`, `uint8`, `int16`, or `int8`.

Examples

```
>>> net = tlx.nn.Input([10, 200])
>>> net = tlx.nn.Softplus()(net)
```

Returns A `Tensor` in the same type as `x`.

Return type Tensor

2.1.11 LeakyReLU

class tensorlayerx.nn.activation.**LeakyReLU**(*negative_slope=0.01*)
Applies the element-wise function:

$$\text{LeakyReLU}(x) = \max(0, x) + negative\_slope * \min(0, x)$$

Parameters

- **negative_slope** (*float*) – Controls the angle of the negative slope. Default: 1e-2
- **name** (*str*) – The function name (optional).

Examples

```
>>> net = tlx.nn.Input([10, 200])
>>> net = tlx.nn.LeakyReLU(alpha=0.5)(net)
```

Returns A `Tensor` in the same type as `x`.

Return type Tensor

References

- Rectifier Nonlinearities Improve Neural Network Acoustic Models [A. L. Maas et al., 2013]

2.1.12 LeakyReLU6

class tensorlayerx.nn.activation.**LeakyReLU6**(*alpha=0.2*)

This activation function is a modified version `leaky_relu()` introduced by the following paper: Rectifier Nonlinearities Improve Neural Network Acoustic Models [A. L. Maas et al., 2013]

This activation function also follows the behaviour of the activation function `tf.ops.relu6()` introduced by the following paper: Convolutional Deep Belief Networks on CIFAR-10 [A. Krizhevsky, 2010]

The function return the following results:

- When x < 0: `f(x) = alpha_low * x`.
- When x in [0, 6]: `f(x) = x`.
- When x > 6: `f(x) = 6`.

Parameters

- **x** (*Tensor*) – Support input type `float`, `double`, `int32`, `int64`, `uint8`, `int16`, or `int8`.
- **alpha** (*float*) – Slope.
- **name** (*str*) – The function name (optional).

Examples

```
>>> net = tlx.nn.Input([10, 200])
>>> net = tlx.nn.LeakyReLU6(alpha=0.5)(net)
```

Returns A `Tensor` in the same type as `x`.

Return type Tensor

References

- Rectifier Nonlinearities Improve Neural Network Acoustic Models [A. L. Maas et al., 2013]
- Convolutional Deep Belief Networks on CIFAR-10 [A. Krizhevsky, 2010]

2.1.13 LeakyTwiceRelu6

class tensorlayerx.nn.activation.**LeakyTwiceRelu6**(*alpha_low=0.2*, *alpha_high=0.2*)

This activation function is a modified version `leaky_relu()` introduced by the following paper: Rectifier Nonlinearities Improve Neural Network Acoustic Models [A. L. Maas et al., 2013]

This activation function also follows the behaviour of the activation function `tf.ops.relu6()` introduced by the following paper: Convolutional Deep Belief Networks on CIFAR-10 [A. Krizhevsky, 2010]

This function push further the logic by adding *leaky* behaviour both below zero and above six.

The function return the following results:

- When x < 0: `f(x) = alpha_low * x`.
- When x in [0, 6]: `f(x) = x`.
- When x > 6: `f(x) = 6 + (alpha_high * (x-6))`.

Parameters

- **x** (*Tensor*) – Support input type `float`, `double`, `int32`, `int64`, `uint8`, `int16`, or `int8`.
- **alpha_low** (*float*) – Slope for x < 0: `f(x) = alpha_low * x`.
- **alpha_high** (*float*) – Slope for x < 6: `f(x) = 6 (alpha_high * (x-6))`.
- **name** (*str*) – The function name (optional).

Examples

```
>>> net = tlx.nn.Input([10, 200])
>>> net = tlx.nn.LeakyTwiceRelu6(alpha_low=0.5, alpha_high=0.2)(net)
```

Returns A `Tensor` in the same type as `x`.

Return type Tensor

References

- Rectifier Nonlinearities Improve Neural Network Acoustic Models [A. L. Maas et al., 2013]
- Convolutional Deep Belief Networks on CIFAR-10 [A. Krizhevsky, 2010]

2.1.14 Ramp

class tensorlayerx.nn.activation.**Ramp**(*v_min=0*, *v_max=1*)

Ramp activation function.

Reference: [tf.clip_by_value]<https://www.tensorflow.org/api_docs/python/tf/clip_by_value>

Parameters

- **x** (*Tensor*) – input.
- **v_min** (*float*) – cap input to v_min as a lower bound.
- **v_max** (*float*) – cap input to v_max as a upper bound.

Returns A `Tensor` in the same type as `x`.

Return type Tensor

Examples

```
>>> inputs = tlx.nn.Input([10, 5])
>>> prelulayer = tlx.nn.Ramp()(inputs)
```

2.1.15 Swish

class tensorlayerx.nn.activation.**Swish**

Swish function.

See Swish: a Self-Gated Activation Function.

Parameters

- **x** (*Tensor*) – input.
- **name** (*str*) – function name (optional).

Examples

```
>>> net = tlx.nn.Input([10, 200])
>>> net = tlx.nn.Swish()(net)
```

Returns A `Tensor` in the same type as `x`.

Return type Tensor

2.1.16 HardTanh

class tensorlayerx.nn.activation.**HardTanh**

Hard tanh activation function.

Which is a ramp function with low bound of -1 and upper bound of 1, shortcut is *htanh*.

Parameters

- **x** (*Tensor*) – input.
- **name** (*str*) – The function name (optional).

Examples

```
>>> net = tlx.nn.Input([10, 200])
>>> net = tlx.nn.HardTanh()(net)
```

Returns A `Tensor` in the same type as `x`.

Return type Tensor

2.1.17 Mish

class tensorlayerx.nn.activation.**Mish**

Applies the Mish function, element-wise. Mish: A Self Regularized Non-Monotonic Neural Activation Function.

text{Mish}(x) = x * text{Tanh}(text{Softplus}(x))

Reference: [Mish: A Self Regularized Non-Monotonic Neural Activation Function .Diganta Misra, 2019]<https://arxiv.org/abs/1908.08681>

Parameters **x** (*Tensor*) – input.

Examples

```
>>> net = tlx.nn.Input([10, 200])
>>> net = tlx.nn.Mish()(net)
```

Returns A `Tensor` in the same type as `x`.

Return type Tensor

2.1.18 Tanh

class tensorlayerx.nn.activation.**Tanh**

Applies the Hyperbolic Tangent (Tanh) function element-wise.

Parameters **x** (*Tensor*) – Support input type `float`, `double`, `int32`, `int64`, `uint8`, `int16`, or `int8`.

Examples

```
>>> net = tlx.nn.Input([10, 200])
>>> net = tlx.nn.Tanh()(net)
```

Returns A `Tensor` in the same type as `x`.

Return type Tensor

2.1.19 Sigmoid

class tensorlayerx.nn.activation.**Sigmoid**

Computes sigmoid of x element-wise. Formula for calculating sigmoid(x) = 1/(1+exp(-x))

Parameters **x** (*Tensor*) – Support input type `float`, `double`, `int32`, `int64`, `uint8`, `int16`, or `int8`.

Examples

```
>>> net = tlx.nn.Input([10, 200])
>>> net = tlx.nn.Sigmoid()(net)
```

Returns A `Tensor` in the same type as `x`.

Return type Tensor

2.1.20 Softmax

class tensorlayerx.nn.activation.**Softmax**(*axis=-1*)

Applies the Softmax function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range [0,1] and sum to 1.

Parameters **axis** (*int*) – A dimension along which Softmax will be computed

Examples

```
>>> net = tlx.nn.Input([10, 200])
>>> net = tlx.nn.Softmax()(net)
```

Returns A `Tensor` in the same type as `x`.

Return type Tensor

2.1.21 Parametric activation

See `tensorlayerx.nn`.

2.2 API - Losses

To make TensorLayerX simple, we minimize the number of cost functions as much as we can. For more complex loss function, TensorFlow(MindSpore, PaddlePaddle, PyTorch) API will be required.

Note: Please refer to Getting Started for getting specific weights for weight regularization.

softmax_cross_entropy_with_logits(output, target)	Softmax cross-entropy operation, returns the TensorLayerX expression of cross-entropy for two distributions, it implements softmax internally.
sigmoid_cross_entropy(output, target[, ...])	Sigmoid cross-entropy operation, see `tf.ops.sigmoid_cross_entropy_with_logits`.
binary_cross_entropy(output, target[, reduction])	Binary cross entropy operation.
mean_squared_error(output, target[, reduction])	Return the TensorLayerX expression of mean-square-error (L2) of two batch of data.
normalized_mean_square_error(output, target)	Return the TensorLayerX expression of normalized mean-square-error of two distributions.
absolute_difference_error(output, target[, ...])	Return the TensorLayerX expression of absolute difference error (L1) of two batch of data.
dice_coe(output, target[, loss_type, axis, ...])	Soft dice (Sørensen or Jaccard) coefficient for comparing the similarity of two batch of data, usually be used for binary image segmentation i.e.
dice_hard_coe(output, target[, threshold, ...])	Non-differentiable Sørensen–Dice coefficient for comparing the similarity of two batch of data, usually be used for binary image segmentation i.e.
iou_coe(output, target[, threshold, axis, ...])	Non-differentiable Intersection over Union (IoU) for comparing the similarity of two batch of data, usually be used for evaluating binary image segmentation.
cross_entropy_seq(logits, target_seqs[, ...])	Returns the expression of cross-entropy of two sequences, implement softmax internally.
cross_entropy_seq_with_mask(logits, ...[, ...])	Returns the expression of cross-entropy of two sequences, implement softmax internally.
cosine_similarity(v1, v2)	Cosine similarity [-1, 1].
li_regularizer(scale[, scope])	Li regularization removes the neurons of previous layer.

Continued on next page

Table 2 – continued from previous page

lo_regularizer(scale)	Lo regularization removes the neurons of current layer.
maxnorm_regularizer([scale])	Max-norm regularization returns a function that can be used to apply max-norm regularization to weights.
maxnorm_o_regularizer(scale)	Max-norm output regularization removes the neurons of current layer.
maxnorm_i_regularizer(scale)	Max-norm input regularization removes the neurons of previous layer.

2.2.1 Softmax cross entropy

tensorlayerx.losses.**softmax_cross_entropy_with_logits**(*output*, *target*, *reduction='mean'*)

Softmax cross-entropy operation, returns the TensorLayerX expression of cross-entropy for two distributions, it implements softmax internally. See `tf.ops.sparse_softmax_cross_entropy_with_logits`.

Parameters

- **output** (*Tensor*) – A batch of distribution with shape: [batch_size, num of classes].
- **target** (*Tensor*) – A batch of index with shape: [batch_size,].
- **reduction** (*str*) – The optional values are "mean", "sum", and "none". If "none", do not perform reduction.

Examples

```
>>> import tensorlayerx as tlx
>>> logits = tlx.convert_to_tensor([[4.0, 2.0, 1.0], [0.0, 5.0, 1.0]])
>>> labels = tlx.convert_to_tensor([[1], [2]])
>>> loss = tlx.losses.softmax_cross_entropy_with_logits(logits, labels)
```

References

- About cross-entropy: https://en.wikipedia.org/wiki/Cross_entropy.
- The code is borrowed from: https://en.wikipedia.org/wiki/Cross_entropy.

2.2.2 Sigmoid cross entropy

tensorlayerx.losses.**sigmoid_cross_entropy**(*output*, *target*, *reduction='mean'*)

Sigmoid cross-entropy operation, see `tf.ops.sigmoid_cross_entropy_with_logits`.

Parameters

- **output** (*Tensor*) – A batch of distribution with shape: [batch_size, num of classes].
- **target** (*Tensor*) – same shape as the input.
- **reduction** (*str*) – The optional values are "mean", "sum", and "none". If "none", do not perform reduction.

Examples

```
>>> import tensorlayerx as tlx
>>> logits = tlx.convert_to_tensor([[4.0, 2.0, 1.0], [0.0, 5.0, 1.0]])
>>> labels = tlx.convert_to_tensor([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
>>> losses = tlx.losses.sigmoid_cross_entropy(logits, labels)
```

2.2.3 Binary cross entropy

tensorlayerx.losses.**binary_cross_entropy**(*output*, *target*, *reduction='mean'*)

Binary cross entropy operation.

Parameters

- **output** (*Tensor*) – Tensor with type of *float32* or *float64*.
- **target** (*Tensor*) – The target distribution, format the same with *output*.
- **reduction** (*str*) – The optional values are "mean", "sum", and "none". If "none", do not perform reduction.

Examples

```
>>> import tensorlayerx as tlx
>>> logits = tlx.convert_to_tensor([[0.4, 0.2, 0.8], [1.1, 0.5, 0.3]])
>>> labels = tlx.convert_to_tensor([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
>>> losses = tlx.losses.binary_cross_entropy(logits, labels)
```

References

- ericjang-DRAW

2.2.4 Mean squared error (L2)

tensorlayerx.losses.**mean_squared_error**(*output*, *target*, *reduction='mean'*)

Return the TensorLayerX expression of mean-square-error (L2) of two batch of data.

Parameters

- **output** (*Tensor*) – 2D, 3D or 4D tensor i.e. [batch_size, n_feature], [batch_size, height, width] or [batch_size, height, width, channel].
- **target** (*Tensor*) – The target distribution, format the same with *output*.
- **reduction** (*str*) – The optional values are "mean", "sum", and "none". If "none", do not perform reduction.

Examples

```
>>> import tensorlayerx as tlx
>>> logits = tlx.convert_to_tensor([[0.4, 0.2, 0.8], [1.1, 0.5, 0.3]])
>>> labels = tlx.convert_to_tensor([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
>>> losses = tlx.losses.mean_squared_error(logits, labels)
```

References

- Wiki Mean Squared Error

2.2.5 Normalized mean square error

tensorlayerx.losses.**normalized_mean_square_error**(*output*, *target*, *reduction='mean'*)

Return the TensorLayerX expression of normalized mean-square-error of two distributions.

Parameters

- **output** (*Tensor*) – 2D, 3D or 4D tensor i.e. [batch_size, n_feature], [batch_size, height, width] or [batch_size, height, width, channel].
- **target** (*Tensor*) – The target distribution, format the same with *output*.
- **reduction** (*str*) – The optional values are "mean", "sum", and "none". If "none", do not perform reduction.

Examples

```
>>> import tensorlayerx as tlx
>>> logits = tlx.convert_to_tensor([[0.4, 0.2, 0.8], [1.1, 0.5, 0.3]])
>>> labels = tlx.convert_to_tensor([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
>>> losses = tlx.losses.normalized_mean_square_error(logits, labels)
```

2.2.6 Absolute difference error (L1)

tensorlayerx.losses.**absolute_difference_error**(*output*, *target*, *reduction='mean'*)

Return the TensorLayerX expression of absolute difference error (L1) of two batch of data.

Parameters

- **output** (*Tensor*) – 2D, 3D or 4D tensor i.e. [batch_size, n_feature], [batch_size, height, width] or [batch_size, height, width, channel].
- **target** (*Tensor*) – The target distribution, format the same with *output*.
- **reduction** (*str*) – The optional values are "mean", "sum", and "none". If "none", do not perform reduction.

Examples

```
>>> import tensorlayerx as tlx
>>> logits = tlx.convert_to_tensor([[0.4, 0.2, 0.8], [1.1, 0.5, 0.3]])
>>> labels = tlx.convert_to_tensor([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
>>> losses = tlx.losses.absolute_difference_error(logits, labels)
```

2.2.7 Dice coefficient

tensorlayerx.losses.**dice_coe**(*output*, *target*, *loss_type='jaccard'*, *axis=(1, 2, 3)*, *smooth=1e-05*)

Soft dice (Sørensen or Jaccard) coefficient for comparing the similarity of two batch of data, usually be used for binary image segmentation i.e. labels are binary. The coefficient between 0 to 1, 1 means totally match.

Parameters

- **output** (*Tensor*) – A distribution with shape: [batch_size, ...], (any dimensions).
- **target** (*Tensor*) – The target distribution, format the same with *output*.
- **loss_type** (*str*) – `jaccard` or `sorensen`, default is `jaccard`.
- **axis** (*tuple of int*) – All dimensions are reduced, default `[1,2,3]`.
- **smooth** (*float*) –

  This small value will be added to the numerator and denominator.

  – If both output and target are empty, it makes sure dice is 1.

  – If either output or target are empty (all pixels are background), dice = `` `smooth/(small_value + smooth)` ``, then if smooth is very small, dice close to 0 (even the image values lower than the threshold), so in this case, higher smooth can have a higher dice.

Examples

```
>>> import tensorlayerx as tlx
>>> logits = tlx.convert_to_tensor([[0.4, 0.2, 0.8], [1.1, 0.5, 0.3]])
>>> labels = tlx.convert_to_tensor([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
>>> dice_loss = tlx.losses.dice_coe(logits, labels, axis=-1)
```

References

- Wiki-Dice

2.2.8 Hard Dice coefficient

`tensorlayerx.losses.`**`dice_hard_coe`**(*output*, *target*, *threshold=0.5*, *axis=(1, 2, 3)*, *smooth=1e-05*)

Non-differentiable Sørensen–Dice coefficient for comparing the similarity of two batch of data, usually be used for binary image segmentation i.e. labels are binary. The coefficient between 0 to 1, 1 if totally match.

Parameters

- **output** (*tensor*) – A distribution with shape: [batch_size, ...], (any dimensions).
- **target** (*tensor*) – The target distribution, format the same with *output*.
- **threshold** (*float*) – The threshold value to be true.
- **axis** (*tuple of integer*) – All dimensions are reduced, default `(1,2,3)`.
- **smooth** (*float*) – This small value will be added to the numerator and denominator, see `dice_coe`.

Examples

```
>>> import tensorlayerx as tlx
>>> logits = tlx.convert_to_tensor([[0.4, 0.2, 0.8], [1.1, 0.5, 0.3]])
>>> labels = tlx.convert_to_tensor([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
>>> dice_loss = tlx.losses.dice_hard_coe(logits, labels, axis=-1)
```

References

- Wiki-Dice

2.2.9 IOU coefficient

tensorlayerx.losses.**iou_coe**(*output*, *target*, *threshold=0.5*, *axis=(1, 2, 3)*, *smooth=1e-05*)

Non-differentiable Intersection over Union (IoU) for comparing the similarity of two batch of data, usually be used for evaluating binary image segmentation. The coefficient between 0 to 1, and 1 means totally match.

Parameters

- **output** (*tensor*) – A batch of distribution with shape: [batch_size, ...], (any dimensions).
- **target** (*tensor*) – The target distribution, format the same with *output*.
- **threshold** (*float*) – The threshold value to be true.
- **axis** (*tuple of integer*) – All dimensions are reduced, default `(1,2,3)`.
- **smooth** (*float*) – This small value will be added to the numerator and denominator, see `dice_coe`.

Examples

```
>>> import tensorlayerx as tlx
>>> logits = tlx.convert_to_tensor([[0.4, 0.2, 0.8], [1.1, 0.5, 0.3]])
>>> labels = tlx.convert_to_tensor([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
>>> dice_loss = tlx.losses.iou_coe(logits, labels, axis=-1)
```

Notes

- IoU cannot be used as training loss, people usually use dice coefficient for training, IoU and hard-dice for evaluating.

2.2.10 Cross entropy for sequence

tensorlayerx.losses.**cross_entropy_seq**(*logits*, *target_seqs*, *batch_size=None*)

Returns the expression of cross-entropy of two sequences, implement softmax internally. Normally be used for fixed length RNN outputs, see PTB example.

Parameters

- **logits** (*Tensor*) – 2D tensor with shape of *[batch_size * n_steps, n_classes]*.
- **target_seqs** (*Tensor*) – The target sequence, 2D tensor *[batch_size, n_steps]*, if the number of step is dynamic, please use `tl.losses.cross_entropy_seq_with_mask` instead.
- **batch_size** (*None or int.*) –

  Whether to divide the losses by batch size.

  - If integer, the return losses will be divided by *batch_size*.
  - If None (default), the return losses will not be divided by anything.

Examples

```
>>> import tensorlayerx as tlx
>>> # see `PTB example <https://github.com/tensorlayer/tensorlayer/blob/master/
↪example/tutorial_ptb_lstm.py>`__.for more details
>>> # outputs shape : (batch_size * n_steps, n_classes)
>>> # targets shape : (batch_size, n_steps)
>>> losses = tlx.losses.cross_entropy_seq(outputs, targets)
```

2.2.11 Cross entropy with mask for sequence

tensorlayerx.losses.**cross_entropy_seq_with_mask**(*logits*, *target_seqs*, *input_mask*, *return_details=False*, *name=None*)

Returns the expression of cross-entropy of two sequences, implement softmax internally. Normally be used for Dynamic RNN with Synced sequence input and output.

Parameters

- **logits** (*Tensor*) – 2D tensor with shape of [batch_size * ?, n_classes], *?* means dynamic IDs for each example. - Can be get from *DynamicRNNLayer* by setting `return_seq_2d` to *True*.
- **target_seqs** (*Tensor*) – int of tensor, like word ID. [batch_size, ?], *?* means dynamic IDs for each example.
- **input_mask** (*Tensor*) – The mask to compute loss, it has the same size with *target_seqs*, normally 0 or 1.
- **return_details** (*boolean*) –

  Whether to return detailed losses.

  - If False (default), only returns the loss.
  - If True, returns the loss, losses, weights and targets (see source code).

Examples

```
>>> import tensorlayerx as tlx
>>> import tensorflow as tf
>>> import numpy as np
>>> batch_size = 64
>>> vocab_size = 10000
>>> embedding_size = 256
>>> ni = tlx.nn.Input([batch_size, None], dtype=tf.int64)
>>> net_lits = []
>>> net_list.append(tlx.nn.Embedding(
...         vocabulary_size = vocab_size,
...         embedding_size = embedding_size,
...         name = 'seq_embedding'))
>>> net_list.append(tlx.nn.RNN(
...         cell =tf.keras.layers.LSTMCell(units=embedding_size, dropout=0.1),
...         return_seq_2d = True,
...         name = 'dynamicrnn'))
>>> net_list.append(tlx.nn.Dense(n_units=vocab_size, name="output"))
>>> model = tlx.nn.Sequential(net_list)
>>> input_seqs = np.random.randint(0, 10, size=(batch_size, 10), dtype=np.int64)
```

(continues on next page)

(continued from previous page)

```
>>> target_seqs = np.random.randint(0, 10, size=(batch_size, 10), dtype=np.int64)
>>> input_mask = np.random.randint(0, 2, size=(batch_size, 10), dtype=np.int64)
>>> outputs = model(input_seqs)
>>> loss = tlx.losses.cross_entropy_seq_with_mask(outputs, target_seqs, input_
↪mask)
```

2.2.12 Cosine similarity

tensorlayerx.losses.**cosine_similarity**(*v1*, *v2*)

Cosine similarity [-1, 1].

Parameters **v2** (*v1,*) – Tensor with the same shape [batch_size, n_feature].

References

- Wiki.

2.2.13 Regularization functions

For tf.nn.l2_loss, tf.contrib.layers.l1_regularizer, tf.contrib.layers.l2_regularizer and tf.contrib.layers.sum_regularizer, see tensorflow API. Maxnorm ^^^^^^^^^^ .. autofunction:: maxnorm_regularizer

Special

tensorlayerx.losses.**li_regularizer**(*scale*, *scope=None*)

Li regularization removes the neurons of previous layer. The *i* represents *inputs*. Returns a function that can be used to apply group li regularization to weights. The implementation follows TensorFlow contrib.

Parameters

- **scale** (*float*) – A scalar multiplier *Tensor*. 0.0 disables the regularizer.
- **scope** (*str*) – An optional scope name for this function.

Returns

Return type A function with signature *li(weights, name=None)* that apply Li regularization.

:raises ValueError : if scale is outside of the range [0.0, 1.0] or if scale is not a float.:

tensorlayerx.losses.**lo_regularizer**(*scale*)

Lo regularization removes the neurons of current layer. The *o* represents *outputs* Returns a function that can be used to apply group lo regularization to weights. The implementation follows TensorFlow contrib.

Parameters **scale** (*float*) – A scalar multiplier *Tensor*. 0.0 disables the regularizer.

Returns

Return type A function with signature *lo(weights, name=None)* that apply Lo regularization.

:raises ValueError : If scale is outside of the range [0.0, 1.0] or if scale is not a float.:

tensorlayerx.losses.**maxnorm_o_regularizer**(*scale*)

Max-norm output regularization removes the neurons of current layer. Returns a function that can be used to apply max-norm regularization to each column of weight matrix. The implementation follows TensorFlow contrib.

Parameters **scale** (*float*) – A scalar multiplier *Tensor*. 0.0 disables the regularizer.

Returns

Return type A function with signature *mn_o(weights, name=None)* that apply Lo regularization.

:raises ValueError : If scale is outside of the range [0.0, 1.0] or if scale is not a float.:

tensorlayerx.losses.**maxnorm_i_regularizer**(*scale*)

Max-norm input regularization removes the neurons of previous layer. Returns a function that can be used to apply max-norm regularization to each row of weight matrix. The implementation follows TensorFlow contrib.

Parameters **scale** (*float*) – A scalar multiplier *Tensor*. 0.0 disables the regularizer.

Returns

Return type A function with signature *mn_i(weights, name=None)* that apply Lo regularization.

:raises ValueError : If scale is outside of the range [0.0, 1.0] or if scale is not a float.:

2.3 API - Metrics

The tensorlayerx.metrics directory contians Accuracy, Auc, Precision and Recall. For more complex metrics, you can encapsulates metric logic and APIs by base class.

2.3.1 Metric list

Metric()	Base class for metric
Accuracy([topk])	Accuracy metric
Auc([curve, num_thresholds])	The auc metric is for binary classification.
Precision()	Precision score for binary classification task.
Recall()	Recall score for binary classification task.
acc(predicts, labels[, topk])	Accuracy function.

Metric

class tensorlayerx.metrics.**Metric**

Base class for metric

__init__()

Initializing the Metric.

update()

Update states for metric.

result()

Computes and returns the metric value.

reset()

Reset states and result.

Accuracy

class tensorlayerx.metrics.**Accuracy**(*topk=1*)

Accuracy metric

Parameters **topk** (*int*) – Specifies the top-k categorical accuracy to compute. Default is (1,).

Examples

```
>>> import tensorlayerx as tlx
>>> y_pred = tlx.ops.convert_to_tensor(np.array([[0.3, 0.2, 0.1, 0.4], [0.2, 0.2,␣
↪0.5, 0.1]]))
>>> y_true = tlx.ops.convert_to_tensor(np.array([[1], [3]]))
>>> metric = tlx.metrics.Accuracy()
>>> metric.update(y_pred, y_true)
>>> res = metric.result()
>>> metric.reset()
```

reset()

Resets all of the metric state.

result()

Computes the top-k categorical accuracy.

Returns

Return type computed result.

update(*y_pred*, *y_true*)

Updates the internal evaluation result *y_pred* and *y_true*.

Parameters

- **y_pred** (*Tensor*) – The predicted value.
- **y_true** (*Tensor*) – The ground truth.

Auc

class tensorlayerx.metrics.**Auc**(*curve='ROC'*, *num_thresholds=4095*)

The auc metric is for binary classification.

Parameters

- **curve** (*str*) – Specifies the mode of the curve to be computed. Only support 'ROC' now.
- **num_thresholds** (*int*) – The number of thresholds to use when discretizing the roc curve.

reset()

Reset states and result

result()

Return the area (a float score) under auc curve

Returns

Return type computed result.

update(*y_pred*, *y_true*)
Updates the auc curve with *y_pred* and *y_true*.

Parameters

- **y_pred** (*Tensor*) – The predicted value.
- **y_true** (*Tensor*) – The ground truth.

Precision

class tensorlayerx.metrics.**Precision**
Precision score for binary classification task.

Examples

```
>>> import tensorlayerx as tlx
>>> y_pred = tlx.ops.convert_to_tensor(np.array([0.3, 0.2, 0.1, 0.7]))
>>> y_true = tlx.ops.convert_to_tensor(np.array([1, 0, 0, 1]))
>>> metric = tlx.metrics.Precision()
>>> metric.update(y_pred, y_true)
>>> res = metric.result()
>>> metric.reset()
```

reset()
Resets all of the metric state.

result()
Return the precision

Returns

Return type computed result.

update(*y_pred*, *y_true*)
Update the states based on the current mini-batch prediction results.

Parameters

- **y_pred** (*Tensor*) – The predicted value.
- **y_true** (*Tensor*) – The ground truth.

Recall

class tensorlayerx.metrics.**Recall**
Recall score for binary classification task.

Examples

```
>>> import tensorlayerx as tlx
>>> y_pred = tlx.ops.convert_to_tensor(np.array([0.3, 0.2, 0.1, 0.7]))
>>> y_true = tlx.ops.convert_to_tensor(np.array([1, 0, 0, 1]))
>>> metric = tlx.metrics.Recall()
>>> metric.update(y_pred, y_true)
```

(continues on next page)

(continued from previous page)

```
>>> res = metric.result()
>>> metric.reset()
```

reset()
Resets all of the metric state.

result()
Return the recall

Returns

Return type computed result.

update(*y_pred*, *y_true*)
Update the states based on the current mini-batch prediction results.

Parameters

- **y_pred** (*Tensor*) – The predicted value.
- **y_true** (*Tensor*) – The ground truth.

acc

tensorlayerx.metrics.**acc**(*predicts*, *labels*, *topk=1*)
Accuracy function.

Parameters

- **predicts** (*Tensor*) – The predicted value.
- **labels** (*Tensor*) – The ground truth.
- **topk** (*int*) – The top k predictions for each class will be checked.

Returns

Return type The accuracy result.

Examples

```
>>> import tensorlayerx as tlx
>>> y_pred = tlx.ops.convert_to_tensor(np.array([[0.3, 0.2, 0.1, 0.4], [0.2, 0.2,␣
↪0.5, 0.1]]))
>>> y_true = tlx.ops.convert_to_tensor(np.array([[1], [3]]))
>>> acc = tlx.metrics.acc(y_pred, y_true, topk=1)
```

2.4 API - Dataflow

2.4.1 Dataflow list

DataLoader(dataset[, batch_size, shuffle, ...])	Data loader.
Dataset()	An abstract class to encapsulate methods and behaviors of datasets.

Continued on next page

Table 4 – continued from previous page

IterableDataset()	An abstract class to encapsulate methods and behaviors of iterable datasets.
TensorDataset(*tensors)	Generate a dataset from a list of tensors.
ChainDataset(datasets)	A Dataset which chains multiple iterable-tyle datasets.
ConcatDataset(datasets)	Concat multiple datasets into a new dataset
Subset(dataset, indices)	Subset of a dataset at specified indices.
random_split(dataset, lengths)	Randomly split a dataset into non-overlapping new datasets of given lengths.
Sampler()	Base class for all Samplers.
BatchSampler([sampler, batch_size, drop_last])	Wraps another sampler to yield a mini-batch of indices.
RandomSampler(data[, replacement, ...])	Samples elements randomly.
SequentialSampler(data)	Samples elements sequentially, always in the same order.
WeightedRandomSampler(weights, num_samples)	Samples elements from `[0,..,len(weights)-1]` with given probabilities (weights).
SubsetRandomSampler(indices)	Samples elements randomly from a given list of indices, without replacement.

2.4.2 Dataflow

DataLoader

class tensorlayerx.dataflow.**DataLoader**(*dataset*, *batch_size=1*, *shuffle=False*, *drop_last=False*, *sampler=None*, *batch_sampler=None*, *num_workers=0*, *collate_fn=None*, *time_out=0*, *worker_init_fn=None*, *prefetch_factor=2*, *persistent_workers=False*)

Data loader. Combines a dataset and a sampler, and provides an iterable over the given dataset.

The *tensorlayerx.dataflow.DataLoader* supports both map-style and iterable-style datasets with single- or multi-process loading, customizing loading order and optional automatic batching

Parameters

- **dataset** (*Dataset*) – dataset from which to load the data.
- **batch_size** (*int*) – how many samples per batch to load, default is 1.
- **shuffle** (*bool*) – set to `True` to have the data reshuffled at every epoch, default is `False`.
- **drop_last** (*bool*) – set to `True` to drop the last incomplete batch, if the dataset size is not divisible by the batch size. If `False` and the size of dataset is not divisible by the batch size, then the last batch will be smaller. default is `False`.
- **sampler** (*Sampler*) – defines the strategy to draw samples from the dataset. If specified, *shuffle* must not be specified.
- **batch_sampler** (*Sampler*) – returns a batch of indices at a time. If specified, *shuffle*, *batch_size*, *drop_last*, *sampler* must not be specified.
- **num_workers** (*int*) – how many subprocesses to use for data loading. `0` means that the data will be loaded in single process. default is `0`.
- **collate_fn** (*callable*) – merges a list of samples to form a mini-batch of Tensor(s). Used when using batched loading from a map-style dataset.

- **time_out** (*numeric*) – if positive, the timeout value for collecting a batch from workers. Should always be non-negative. default is `0`.
- **worker_init_fn** (*callable*) – If not `None`, this will be called on each worker subprocess with the worker id (an int in `[0, num_workers - 1]`) as input, after seeding and before data loading. default is `None`.
- **prefetch_factor** (*int*) – Number of samples loaded in advance by each worker. `2` means there will be a total of 2 * num_workers samples prefetched across all workers. default is `2`
- **persistent_workers** (*bool*) – If `True`, the data loader will not shutdown the worker processes after a dataset has been consumed once. This allows to maintain the workers *Dataset* instances alive. default is `False`.

Dataset

class tensorlayerx.dataflow.**Dataset**

An abstract class to encapsulate methods and behaviors of datasets. All datasets in map-style(dataset samples can be get by a given key) should be a subclass of 'tensorlayerx.dataflow.Dataset'. ALl subclasses should implement following methods: `__getitem__`: get sample from dataset with a given index. `__len__`: return dataset sample number. `__add__`: concat two datasets

Examples

With TensorLayerx

```
>>> from tensorlayerx.dataflow import Dataset
>>> class mnistdataset(Dataset):
>>>     def __init__(self, data, label,transform):
>>>         self.data = data
>>>         self.label = label
>>>         self.transform = transform
>>>     def __getitem__(self, index):
>>>         data = self.data[index].astype('float32')
>>>         data = self.transform(data)
>>>         label = self.label[index].astype('int64')
>>>         return data, label
>>>     def __len__(self):
>>>         return len(self.data)
>>> train_dataset = mnistdataset(data = X_train, label = y_train ,transform =␣
↪transform)
```

IterableDataset

class tensorlayerx.dataflow.**IterableDataset**

An abstract class to encapsulate methods and behaviors of iterable datasets. All datasets in iterable-style (can only get sample one by one sequentially, likea Python iterator) should be a subclass of *tensorlayerx.dataflow.IterableDataset*. All subclasses should implement following methods: `__iter__`: yield sample sequentially.

Examples

With TensorLayerx

>>>#example 1: >>> from tensorlayerx.dataflow import IterableDataset >>> class mnistdataset(IterableDataset): >>> def __init__(self, data, label,transform): >>> self.data = data >>> self.label = label >>> self.transform = transform >>> def __iter__(self): >>> for i in range(len(self.data)): >>> data = self.data[i].astype('float32') >>> data = self.transform(data) >>> label = self.label[i].astype('int64') >>> yield data, label >>> train_dataset = mnistdataset(data = X_train, label = y_train ,transform = transform) >>>#example 2: >>>iterable_dataset_1 = mnistdataset(data_1, label_1, transform_1) >>>iterable_dataset_2 = mnistdataset(data_2, label_2, transform_2) >>>new_iterable_dataset = iterable_dataset_1 + iterable_dataset_2

TensorDataset

class tensorlayerx.dataflow.**TensorDataset**(**tensors*)

Generate a dataset from a list of tensors. Each sample will be retrieved by indexing tensors along the first dimension.

Parameters ***tensor** (*list or tuple of tensors*) – tensors that have the same size of the first dimension.

Examples

With TensorLayerx

```
>>> import numpy as np
>>> import tensorlayerx as tlx
>>> data = np.random.random([10,224,224,3]).astype(np.float32)
>>> label = np.random.random((10,)).astype(np.int32)
>>> data = tlx.convert_to_tensor(data)
>>> label = tlx.convert_to_tensor(label)
>>> dataset = tlx.dataflow.TensorDataset([data, label])
>>> for i in range(len(dataset)):
>>>     x, y = dataset[i]
```

ChainDataset

class tensorlayerx.dataflow.**ChainDataset**(*datasets*)

A Dataset which chains multiple iterable-tyle datasets.

Parameters **datasets** (*list or tuple*) – sequence of datasets to be chainned.

Examples

With TensorLayerx

```
>>> import numpy as np
>>> from tensorlayerx.dataflow import IterableDataset, ChainDataset
>>> class mnistdataset(IterableDataset):
>>>     def __init__(self, data, label):
>>>         self.data = data
>>>         self.label = label
>>>     def __iter__(self):
>>>         for i in range(len(self.data)):
>>>             yield self.data[i] self.label[i]
>>> train_dataset1 = mnistdataset(data = X_train1, label = y_train1)
>>> train_dataset2 = mnistdataset(data = X_train2, label = y_train2)
>>> train_dataset = ChainDataset([train_dataset1, train_dataset2])
```

ConcatDataset

class tensorlayerx.dataflow.**ConcatDataset**(*datasets*)

Concat multiple datasets into a new dataset

Parameters **datasets** (*list or tuple*) – sequence of datasets to be concatenated

Examples

With TensorLayerx

```
>>> import numpy as np
>>> from tensorlayerx.dataflow import Dataset, ConcatDataset
>>> class mnistdataset(Dataset):
>>>     def __init__(self, data, label,transform):
>>>         self.data = data
>>>         self.label = label
>>>         self.transform = transform
>>>     def __getitem__(self, index):
>>>         data = self.data[index].astype('float32')
>>>         data = self.transform(data)
>>>         label = self.label[index].astype('int64')
>>>         return data, label
>>>     def __len__(self):
>>>         return len(self.data)
>>> train_dataset1 = mnistdataset(data = X_train1, label = y_train1 ,transform =␣
↪transform1)
>>> train_dataset2 = mnistdataset(data = X_train2, label = y_train2 ,transform =␣
↪transform2)
>>> train_dataset = ConcatDataset([train_dataset1, train_dataset2])
```

Subset

class tensorlayerx.dataflow.**Subset**(*dataset*, *indices*)

Subset of a dataset at specified indices.

Parameters

- **dataset** (Dataset) – The whole Dataset
- **indices** (*list or tuple*) – Indices in the whole set selected for subset

Examples

With TensorLayerx

```
>>> import numpy as np
>>> from tensorlayerx.dataflow import Dataset, Subset
>>> class mnistdataset(Dataset):
>>>     def __init__(self, data, label):
>>>         self.data = data
>>>         self.label = label
>>>     def __iter__(self):
>>>         for i in range(len(self.data)):
>>>             yield self.data[i] self.label[i]
```

(continues on next page)

(continued from previous page)

```
>>> train_dataset = mnistdataset(data = X_train, label = y_train)
>>> sub_dataset = Subset(train_dataset, indices=[1,2,3])
```

random_split

class tensorlayerx.dataflow.**random_split**

Randomly split a dataset into non-overlapping new datasets of given lengths.

Parameters

- **dataset** (Dataset) – dataset to be split
- **lengths** (*list or tuple*) – lengths of splits to be produced

Examples

With TensorLayerx

```
>>> import numpy as np
>>> from tensorlayerx.dataflow import Dataset, Subset
>>> random_split(range(10), [3, 7])
```

Sampler

class tensorlayerx.dataflow.**Sampler**

Base class for all Samplers. All subclasses should implement following methods: __iter__: providing a way to iterate over indices of dataset element __len__: the length of the returned iterators.

Examples

With TensorLayerx

```
>>> from tensorlayerx.dataflow import Sampler
>>> class MySampler(Sampler):
>>>     def __init__(self, data):
>>>         self.data = data
>>>     def __iter__(self):
>>>         return iter(range(len(self.data_source)))
>>>     def __len__(self):
>>>         return len(self.data)
```

BatchSampler

class tensorlayerx.dataflow.**BatchSampler**(*sampler=None*, *batch_size=1*, *drop_last=False*)

Wraps another sampler to yield a mini-batch of indices.

Parameters

- **sampler** (Sampler) – Base sampler.
- **batch_size** (*int*) – Size of mini-batch

- **drop_last** (*bool*) – If `True`, the sampler will drop the last batch if its size would be less than `batch_size`

Examples

With TensorLayerx

```
>>> from tensorlayerx.dataflow import BatchSampler, SequentialSampler
>>> list(BatchSampler(SequentialSampler(range(10)), batch_size=3, drop_
↪last=False))
>>> #[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9]]
>>> list(BatchSampler(SequentialSampler(range(10)), batch_size=3, drop_last=True))
>>> #[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

RandomSampler

class tensorlayerx.dataflow.**RandomSampler**(*data*, *replacement=False*, *num_samples=None*, *generator=None*)

Samples elements randomly. If without replacement, then sample from a shuffled dataset. If with replacement, then user can specify‘num_samples‘ to draw.

Parameters

- **data** (`Dataset`) – dataset to sample
- **replacement** (*bool*) – samples are drawn on-demand with replacement if `True`, default=‘‘False‘‘
- **num_samples** (*int*) – number of samples to draw, default=‘len(dataset)‘. This argument is supposed to be specified only when *replacement* is `True`.
- **generator** (*Generator*) – Generator used in sampling. Default is None.

Examples

With TensorLayerx

```
>>> from tensorlayerx.dataflow import RandomSampler, Dataset
>>> import numpy as np
>>> class mydataset(Dataset):
>>>     def __init__(self):
>>>         self.data = [np.random.random((224,224,3)) for i in range(100)]
>>>         self.label = [np.random.randint(1, 10, (1,)) for i in range(100)]
>>>     def __getitem__(self, item):
>>>         x = self.data[item]
>>>         y = self.label[item]
>>>         return x, y
>>>     def __len__(self):
>>>         return len(self.data)
>>> sampler = RandomSampler(data = mydataset())
```

SequentialSampler

class tensorlayerx.dataflow.**SequentialSampler**(*data*)

Samples elements sequentially, always in the same order.

Parameters **data** (Dataset) – dataset to sample

Examples

With TensorLayerx

```
>>> from tensorlayerx.dataflow import SequentialSampler, Dataset
>>> import numpy as np
>>> class mydataset(Dataset):
>>>     def __init__(self):
>>>         self.data = [np.random.random((224,224,3)) for i in range(100)]
>>>         self.label = [np.random.randint(1, 10, (1,)) for i in range(100)]
>>>     def __getitem__(self, item):
>>>         x = self.data[item]
>>>         y = self.label[item]
>>>         return x, y
>>>     def __len__(self):
>>>         return len(self.data)
>>> sampler = SequentialSampler(data = mydataset())
```

WeightedRandomSampler

class tensorlayerx.dataflow.**WeightedRandomSampler**(*weights*, *num_samples*, *replacement=True*)

Samples elements from `[0,..,len(weights)-1]` with given probabilities (weights).

Parameters

- **weights** (*list or tuple*) – a sequence of weights, not necessary summing up to one
- **num_samples** (*int*) – number of samples to draw
- **replacement** (*bool*) – if `True`, samples are drawn with replacement. If not, they are drawn without replacement, which means that when a sample index is drawn for a row, it cannot be drawn again for that row.

Examples

With TensorLayerx

```
>>> from tensorlayerx.dataflow import WeightedRandomSampler, Dataset
>>> import numpy as np
>>> sampler = list(WeightedRandomSampler(weights=[0.2,0.3,0.4,0.5,4.0], num_
↪samples=5, replacement=True))
>>> #[4, 4, 1, 4, 4]
>>> sampler = list(WeightedRandomSampler(weights=[0.2,0.3,0.4,0.5,0.6], num_
↪samples=5, replacement=False))
>>> #[4, 1, 3, 0, 2]
```

SubsetRandomSampler

class tensorlayerx.dataflow.**SubsetRandomSampler**(*indices*)

Samples elements randomly from a given list of indices, without replacement.

Parameters **indices** (*list or tuple*) – sequence of indices

2.5 API - Files

A collections of helper functions to work with dataset. Load benchmark dataset, save and restore model, save and load variables.

TensorLayer provides rich layer implementations trailed for various benchmarks and domain-specific problems. In addition, we also support transparent access to native TensorFlow parameters. For example, we provide not only layers for local response normalization, but also layers that allow user to apply `tf.ops.lrn` on `network.outputs`. More functions can be found in TensorFlow API.

load_mnist_dataset([shape, path])	Load the original mnist.
load_fashion_mnist_dataset([shape, path])	Load the fashion mnist.
load_cifar10_dataset([shape, path, plotable])	Load CIFAR-10 dataset.
load_cropped_svhn([path, include_extra])	Load Cropped SVHN.
load_matt_mahoney_text8_dataset([path])	Load Matt Mahoney's dataset.
load_imdb_dataset([path, nb_words, ...])	Load IMDB dataset.
load_nietzsche_dataset([path])	Load Nietzsche dataset.
load_flickr25k_dataset([tag, path, ...])	Load Flickr25K dataset.
load_flickr1M_dataset([tag, size, path, ...])	Load Flick1M dataset.
load_cyclegan_dataset([filename, path])	Load images from CycleGAN's database, see this link.
load_celebA_dataset([path])	Load CelebA dataset
load_mpii_pose_dataset([path, is_16_pos_only])	Load MPII Human Pose Dataset.
download_file_from_google_drive(ID, destination)	Download file from Google Drive.
save_npz([save_list, name])	Input parameters and the file name, save parameters into .npz file.
load_npz([path, name])	Load the parameters of a Model saved by tlx.files.save_npz().
assign_weights(weights, network)	Assign the given parameters to the TensorLayer network.
load_and_assign_npz([name, network])	Load model from npz and assign to a network.
save_npz_dict([save_list, name])	Input parameters and the file name, save parameters as a dictionary into .npz file.
load_and_assign_npz_dict([name, network, skip])	Restore the parameters saved by `tlx.files.save_npz_dict()`.
save_weights_to_hdf5(filepath, network)	Input filepath and save weights in hdf5 format.
load_hdf5_to_weights_in_order(filepath, network)	Load weights sequentially from a given file of hdf5 format
load_hdf5_to_weights(filepath, network[, skip])	Load weights by name from a given file of hdf5 format
save_any_to_npy([save_dict, name])	Save variables to *.npy* file.
load_npy_to_any([path, name])	Load *.npy* file.
file_exists(filepath)	Check whether a file exists by given file path.
folder_exists(folderpath)	Check whether a folder exists by given folder path.
del_file(filepath)	Delete a file by given file path.
del_folder(folderpath)	Delete a folder by given folder path.
read_file(filepath)	Read a file and return a string.
load_file_list([path, regx, printable, ...])	Return a file list in a folder by given a path and regular expression.
load_folder_list([path])	Return a folder list in a folder by given a folder path.

Continued on next page

Table 5 – continued from previous page

exists_or_mkdir(path[, verbose])	Check a folder by given name, if not exist, create the folder and return False, if directory exists, return True.
maybe_download_and_extract(filename, ...[, ...])	Checks if file exists in working_directory otherwise tries to dowload the file, and optionally also tries to extract the file if format is ".zip" or ".tar"
natural_keys(text)	Sort list of string with number in human order.

2.5.1 Load dataset functions

MNIST

tensorlayerx.files.**load_mnist_dataset**(*shape=(-1, 784)*, *path='data'*)

Load the original mnist.

Automatically download MNIST dataset and return the training, validation and test set with 50000, 10000 and 10000 digit images respectively.

Parameters

- **shape** (*tuple*) – The shape of digit images (the default is (-1, 784), alternatively (-1, 28, 28, 1)).
- **path** (*str*) – The path that the data is downloaded to.

Returns **X_train, y_train, X_val, y_val, X_test, y_test** – Return splitted training/validation/test set respectively.

Return type tuple

Examples

```
>>> X_train, y_train, X_val, y_val, X_test, y_test = tlx.files.load_mnist_
↪dataset(shape=(-1,784), path='datasets')
>>> X_train, y_train, X_val, y_val, X_test, y_test = tlx.files.load_mnist_
↪dataset(shape=(-1, 28, 28, 1))
```

Fashion-MNIST

tensorlayerx.files.**load_fashion_mnist_dataset**(*shape=(-1, 784)*, *path='data'*)

Load the fashion mnist.

Automatically download fashion-MNIST dataset and return the training, validation and test set with 50000, 10000 and 10000 fashion images respectively, examples.

Parameters

- **shape** (*tuple*) – The shape of digit images (the default is (-1, 784), alternatively (-1, 28, 28, 1)).
- **path** (*str*) – The path that the data is downloaded to.

Returns **X_train, y_train, X_val, y_val, X_test, y_test** – Return splitted training/validation/test set respectively.

Return type tuple

Examples

```
>>> X_train, y_train, X_val, y_val, X_test, y_test = tlx.files.load_fashion_mnist_
↪dataset(shape=(-1,784), path='datasets')
>>> X_train, y_train, X_val, y_val, X_test, y_test = tlx.files.load_fashion_mnist_
↪dataset(shape=(-1, 28, 28, 1))
```

CIFAR-10

tensorlayerx.files.**load_cifar10_dataset**(*shape=(-1, 32, 32, 3), path='data', plotable=False*)

Load CIFAR-10 dataset.

It consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

Parameters

- **shape** (*tupe*) – The shape of digit images e.g. (-1, 3, 32, 32) and (-1, 32, 32, 3).
- **path** (*str*) – The path that the data is downloaded to, defaults is `data/cifar10/`.
- **plotable** (*boolean*) – Whether to plot some image examples, False as default.

Examples

```
>>> X_train, y_train, X_test, y_test = tlx.files.load_cifar10_dataset(shape=(-1,␣
↪32, 32, 3))
```

References

- CIFAR website
- Data download link
- https://teratail.com/questions/28932

SVHN

tensorlayerx.files.**load_cropped_svhn**(*path='data', include_extra=True*)

Load Cropped SVHN.

The Cropped Street View House Numbers (SVHN) Dataset contains 32x32x3 RGB images. Digit '1' has label 1, '9' has label 9 and '0' has label 0 (the original dataset uses 10 to represent '0'), see uftdl website.

Parameters

- **path** (*str*) – The path that the data is downloaded to.
- **include_extra** (*boolean*) – If True (default), add extra images to the training set.

Returns **X_train, y_train, X_test, y_test** – Return splitted training/test set respectively.

Return type tuple

Examples

```
>>> X_train, y_train, X_test, y_test = tlx.files.load_cropped_svhn(include_
↪extra=False)
>>> tlx.vis.save_images(X_train[0:100], [10, 10], 'svhn.png')
```

Matt Mahoney's text8

tensorlayerx.files.**load_matt_mahoney_text8_dataset**(*path='data'*)

Load Matt Mahoney's dataset.

Download a text file from Matt Mahoney's website if not present, and make sure it's the right size. Extract the first file enclosed in a zip file as a list of words. This dataset can be used for Word Embedding.

Parameters **path** (*str*) – The path that the data is downloaded to, defaults is `data/mm_test8/`.

Returns The raw text data e.g. [.... 'their', 'families', 'who', 'were', 'expelled', 'from', 'jerusalem', ...]

Return type list of str

Examples

```
>>> words = tlx.files.load_matt_mahoney_text8_dataset()
>>> print('Data size', len(words))
```

IMBD

tensorlayerx.files.**load_imdb_dataset**(*path='data'*, *nb_words=None*, *skip_top=0*, *maxlen=None*, *test_split=0.2*, *seed=113*, *start_char=1*, *oov_char=2*, *index_from=3*)

Load IMDB dataset.

Parameters

- **path** (*str*) – The path that the data is downloaded to, defaults is `data/imdb/`.
- **nb_words** (*int*) – Number of words to get.
- **skip_top** (*int*) – Top most frequent words to ignore (they will appear as oov_char value in the sequence data).
- **maxlen** (*int*) – Maximum sequence length. Any longer sequence will be truncated.
- **seed** (*int*) – Seed for reproducible data shuffling.
- **start_char** (*int*) – The start of a sequence will be marked with this character. Set to 1 because 0 is usually the padding character.
- **oov_char** (*int*) – Words that were cut out because of the num_words or skip_top limit will be replaced with this character.
- **index_from** (*int*) – Index actual words with this index and higher.

Examples

```
>>> X_train, y_train, X_test, y_test = tlx.files.load_imdb_dataset(
...                                 nb_words=20000, test_split=0.2)
>>> print('X_train.shape', X_train.shape)
(20000,)  [[1, 62, 74, ... 1033, 507, 27],[1, 60, 33, ... 13, 1053, 7]..]
>>> print('y_train.shape', y_train.shape)
(20000,)  [1 0 0 ..., 1 0 1]
```

References

- Modified from keras.

Nietzsche

tensorlayerx.files.**load_nietzsche_dataset**(*path='data'*)

Load Nietzsche dataset.

Parameters **path** (*str*) – The path that the data is downloaded to, defaults is `data/nietzsche/`.

Returns The content.

Return type str

Examples

```
>>> see tutorial_generate_text.py
>>> words = tlx.files.load_nietzsche_dataset()
>>> words = basic_clean_str(words)
>>> words = words.split()
```

Flickr25k

tensorlayerx.files.**load_flickr25k_dataset**(*tag='sky'*, *path='data'*, *n_threads=50*, *printable=False*)

Load Flickr25K dataset.

Returns a list of images by a given tag from Flick25k dataset, it will download Flickr25k from the official website at the first time you use it.

Parameters

- **tag** (*str or None*) –

  What images to return.

  - If you want to get images with tag, use string like 'dog', 'red', see Flickr Search.
  - If you want to get all images, set to `None`.

- **path** (*str*) – The path that the data is downloaded to, defaults is `data/flickr25k/`.
- **n_threads** (*int*) – The number of thread to read image.
- **printable** (*boolean*) – Whether to print infomation when reading images, default is `False`.

Examples

Get images with tag of sky

```
>>> images = tlx.files.load_flickr25k_dataset(tag='sky')
```

Get all images

```
>>> images = tlx.files.load_flickr25k_dataset(tag=None, n_threads=100,␣
↪printable=True)
```

Flickr1M

tensorlayerx.files.**load_flickr1M_dataset**(*tag='sky'*, *size=10*, *path='data'*, *n_threads=50*, *printable=False*)

Load Flick1M dataset.

Returns a list of images by a given tag from Flickr1M dataset, it will download Flickr1M from the official website at the first time you use it.

Parameters

- **tag** (*str or None*) –

  What images to return.

  - If you want to get images with tag, use string like 'dog', 'red', see Flickr Search.
  - If you want to get all images, set to `None`.

- **size** (*int*) – integer between 1 to 10. 1 means 100k images ... 5 means 500k images, 10 means all 1 million images. Default is 10.
- **path** (*str*) – The path that the data is downloaded to, defaults is `data/flickr25k/`.
- **n_threads** (*int*) – The number of thread to read image.
- **printable** (*boolean*) – Whether to print infomation when reading images, default is `False`.

Examples

Use 200k images

```
>>> images = tlx.files.load_flickr1M_dataset(tag='zebra', size=2)
```

Use 1 Million images

```
>>> images = tlx.files.load_flickr1M_dataset(tag='zebra')
```

CycleGAN

tensorlayerx.files.**load_cyclegan_dataset**(*filename='summer2winter_yosemite'*, *path='data'*)

Load images from CycleGAN's database, see this link.

Parameters

- **filename** (*str*) – The dataset you want, see this link.
- **path** (*str*) – The path that the data is downloaded to, defaults is *data/cyclegan*

Examples

```
>>> im_train_A, im_train_B, im_test_A, im_test_B = load_cyclegan_dataset(filename=
↪'summer2winter_yosemite')
```

CelebA

tensorlayerx.files.**load_celebA_dataset**(*path='data'*)

Load CelebA dataset

Return a list of image path.

Parameters **path** (*str*) – The path that the data is downloaded to, defaults is `data/celebA/`.

MPII

tensorlayerx.files.**load_mpii_pose_dataset**(*path='data'*, *is_16_pos_only=False*)

Load MPII Human Pose Dataset.

Parameters

- **path** (*str*) – The path that the data is downloaded to.
- **is_16_pos_only** (*boolean*) – If True, only return the peoples contain 16 pose keypoints. (Usually be used for single person pose estimation)

Returns

- **img_train_list** (*list of str*) – The image directories of training data.
- **ann_train_list** (*list of dict*) – The annotations of training data.
- **img_test_list** (*list of str*) – The image directories of testing data.
- **ann_test_list** (*list of dict*) – The annotations of testing data.

Examples

```
>>> import pprint
>>> import tensorlayerx as tlx
>>> img_train_list, ann_train_list, img_test_list, ann_test_list = tlx.files.load_
↪mpii_pose_dataset()
>>> image = tlx.vis.read_image(img_train_list[0])
>>> tlx.vis.draw_mpii_pose_to_image(image, ann_train_list[0], 'image.png')
>>> pprint.pprint(ann_train_list[0])
```

References

- MPII Human Pose Dataset. CVPR 14
- MPII Human Pose Models. CVPR 16

- MPII Human Shape, Poselet Conditioned Pictorial Structures and etc
- MPII Keyponts and ID

Google Drive

tensorlayerx.files.**download_file_from_google_drive**(*ID*, *destination*)

Download file from Google Drive.

See `tlx.files.load_celebA_dataset` for example.

Parameters

- **ID** (*str*) – The driver ID.
- **destination** (*str*) – The destination for save file.

2.5.2 Load and save network

TensorFlow provides `.ckpt` file format to save and restore the models, while we suggest to use standard python file format `hdf5` to save models for the sake of cross-platform. Other file formats such as `.npz` are also available.

```
## save model as .h5
tlx.files.save_weights_to_hdf5('model.h5', network.all_weights)
# restore model from .h5 (in order)
tlx.files.load_hdf5_to_weights_in_order('model.h5', network.all_weights)
# restore model from .h5 (by name)
tlx.files.load_hdf5_to_weights('model.h5', network.all_weights)

## save model as .npz
tlx.files.save_npz(network.all_weights , name='model.npz')
# restore model from .npz (method 1)
load_params = tl.files.load_npz(name='model.npz')
tlx.files.assign_weights(sess, load_params, network)
# restore model from .npz (method 2)
tlx.files.load_and_assign_npz(sess=sess, name='model.npz', network=network)

## you can assign the pre-trained parameters as follow
# 1st parameter
tlx.files.assign_weights(sess, [load_params[0]], network)
# the first three parameters
tlx.files.assign_weights(sess, load_params[:3], network)
```

Save network into list (npz)

tensorlayerx.files.**save_npz**(*save_list=None*, *name='model.npz'*)

Input parameters and the file name, save parameters into .npz file. Use tlx.utils.load_npz() to restore.

Parameters

- **save_list** (*list of tensor*) – A list of parameters (tensor) to be saved.
- **name** (*str*) – The name of the *.npz* file.

Examples

Save model to npz

```
>>> tlx.files.save_npz(network.all_weights, name='model.npz')
```

Load model from npz (Method 1)

```
>>> load_params = tlx.files.load_npz(name='model.npz')
>>> tlx.files.assign_weights(load_params, network)
```

Load model from npz (Method 2)

```
>>> tlx.files.load_and_assign_npz(name='model.npz', network=network)
```

References

Saving dictionary using numpy

Load network from list (npz)

tensorlayerx.files.**load_npz**(*path=''*, *name='model.npz'*)

Load the parameters of a Model saved by tlx.files.save_npz().

Parameters

- **path** (*str*) – Folder path to *.npz* file.
- **name** (*str*) – The name of the *.npz* file.

Returns A list of parameters in order.

Return type list of array

Examples

- See `tlx.files.save_npz`

References

- Saving dictionary using numpy

Assign a list of parameters to network

tensorlayerx.files.**assign_weights**(*weights*, *network*)

Assign the given parameters to the TensorLayer network.

Parameters

- **weights** (*list of array*) – A list of model weights (array) in order.
- **network** (Layer) – The network to be assigned.

Returns

- *1) list of operations if in graph mode* – A list of tf ops in order that assign weights. Support sess.run(ops) manually.
- *2) list of tf variables if in eager mode* – A list of tf variables (assigned weights) in order.

Examples

References

- Assign value to a TensorFlow variable

Load and assign a list of parameters to network

tensorlayerx.files.**load_and_assign_npz**(*name=None*, *network=None*)

Load model from npz and assign to a network.

Parameters

- **name** (*str*) – The name of the *.npz* file.
- **network** (Model) – The network to be assigned.

Examples

- See tlx.files.save_npz

Save network into dict (npz)

tensorlayerx.files.**save_npz_dict**(*save_list=None*, *name='model.npz'*)

Input parameters and the file name, save parameters as a dictionary into .npz file.

Use tlx.files.load_and_assign_npz_dict() to restore.

Parameters

- **save_list** (*list of parameters*) – A list of parameters (tensor) to be saved.
- **name** (*str*) – The name of the *.npz* file.

Load network from dict (npz)

tensorlayerx.files.**load_and_assign_npz_dict**(*name='model.npz'*, *network=None*, *skip=False*)

Restore the parameters saved by tlx.files.save_npz_dict().

Parameters

- **name** (*str*) – The name of the *.npz* file.
- **network** (Model) – The network to be assigned.
- **skip** (*boolean*) – If 'skip' == True, loaded weights whose name is not found in network's weights will be skipped. If 'skip' is False, error will be raised when mismatch is found. Default False.

Save network into OrderedDict (hdf5)

tensorlayerx.files.**save_weights_to_hdf5**(*filepath*, *network*)
Input filepath and save weights in hdf5 format.

Parameters

- **filepath** (*str*) – Filename to which the weights will be saved.
- **network** (*Model*) – TL model.

Load network from hdf5 in order

tensorlayerx.files.**load_hdf5_to_weights_in_order**(*filepath*, *network*)
Load weights sequentially from a given file of hdf5 format

Parameters

- **filepath** (*str*) – Filename to which the weights will be loaded, should be of hdf5 format.
- **network** (*Model*) – TL model.
- **Notes** – If the file contains more weights than given 'weights', then the redundant ones will be ignored if all previous weights match perfectly.

Load network from hdf5 by name

tensorlayerx.files.**load_hdf5_to_weights**(*filepath*, *network*, *skip=False*)
Load weights by name from a given file of hdf5 format

Parameters

- **filepath** (*str*) – Filename to which the weights will be loaded, should be of hdf5 format.
- **network** (*Model*) – TL model.
- **skip** (*bool*) – If 'skip' == True, loaded weights whose name is not found in 'weights' will be skipped. If 'skip' is False, error will be raised when mismatch is found. Default False.

2.5.3 Load and save variables

Save variables as .npy

tensorlayerx.files.**save_any_to_npy**(*save_dict=None*, *name='file.npy'*)
Save variables to *.npy* file.

Parameters

- **save_dict** (*directory*) – The variables to be saved.
- **name** (*str*) – File name.

Examples

```
>>> tlx.files.save_any_to_npy(save_dict={'data': ['a','b']}, name='test.npy')
>>> data = tlx.files.load_npy_to_any(name='test.npy')
>>> print(data)
{'data': ['a','b']}
```

Load variables from .npy

tensorlayerx.files.**load_npy_to_any**(*path=''*, *name='file.npy'*)

Load *.npy* file.

Parameters

- **path** (*str*) – Path to the file (optional).
- **name** (*str*) – File name.

Examples

- see tlx.files.save_any_to_npy()

2.5.4 Folder/File functions

Check file exists

tensorlayerx.files.**file_exists**(*filepath*)

Check whether a file exists by given file path.

Check folder exists

tensorlayerx.files.**folder_exists**(*folderpath*)

Check whether a folder exists by given folder path.

Delete file

tensorlayerx.files.**del_file**(*filepath*)

Delete a file by given file path.

Delete folder

tensorlayerx.files.**del_folder**(*folderpath*)

Delete a folder by given folder path.

Read file

tensorlayerx.files.**read_file**(*filepath*)

Read a file and return a string.

Examples

```
>>> data = tlx.files.read_file('data.txt')
```

Load file list from folder

tensorlayerx.files.**load_file_list**(*path=None*, *regx='\\.jpg'*, *printable=True*, *keep_prefix=False*)

Return a file list in a folder by given a path and regular expression.

Parameters

- **path** (*str or None*) – A folder path, if *None*, use the current directory.
- **regx** (*str*) – The regx of file name.
- **printable** (*boolean*) – Whether to print the files infomation.
- **keep_prefix** (*boolean*) – Whether to keep path in the file name.

Examples

```
>>> file_list = tlx.files.load_file_list(path=None, regx='w1pre_[0-9]+\.(npz)')
```

Load folder list from folder

tensorlayerx.files.**load_folder_list**(*path=''*)

Return a folder list in a folder by given a folder path.

Parameters **path** (*str*) – A folder path.

Check and Create folder

tensorlayerx.files.**exists_or_mkdir**(*path*, *verbose=True*)

Check a folder by given name, if not exist, create the folder and return False, if directory exists, return True.

Parameters

- **path** (*str*) – A folder path.
- **verbose** (*boolean*) – If True (default), prints results.

Returns True if folder already exist, otherwise, returns False and create the folder.

Return type boolean

Examples

```
>>> tlx.files.exists_or_mkdir("checkpoints/train")
```

Download or extract

tensorlayerx.files.**maybe_download_and_extract**(*filename*, *working_directory*, *url_source*, *extract=False*, *expected_bytes=None*)

Checks if file exists in working_directory otherwise tries to dowload the file, and optionally also tries to extract the file if format is ".zip" or ".tar"

Parameters

- **filename** (*str*) – The name of the (to be) dowloaded file.
- **working_directory** (*str*) – A folder path to search for the file in and dowload the file to
- **url** (*str*) – The URL to download the file from
- **extract** (*boolean*) – If True, tries to uncompress the dowloaded file is ".tar.gz/.tar.bz2" or ".zip" file, default is False.
- **expected_bytes** (*int or None*) – If set tries to verify that the downloaded file is of the specified size, otherwise raises an Exception, defaults is None which corresponds to no check being performed.

Returns File path of the dowloaded (uncompressed) file.

Return type str

Examples

```
>>> down_file = tlx.files.maybe_download_and_extract(filename='train-images-idx3-
↪ubyte.gz',
...                                                 working_directory='data/',
...                                                 url_source='http://yann.lecun.com/
↪exdb/mnist/')
>>> tlx.files.maybe_download_and_extract(filename='ADEChallengeData2016.zip',
...                                       working_directory='data/',
...                                       url_source='http://sceneparsing.
↪csail.mit.edu/data/',
...                                       extract=True)
```

2.5.5 Sort

List of string with number in human order

tensorlayerx.files.**natural_keys**(*text*)

Sort list of string with number in human order.

Examples

```
>>> l = ['im1.jpg', 'im31.jpg', 'im11.jpg', 'im21.jpg', 'im03.jpg', 'im05.jpg']
>>> l.sort(key=tlx.files.natural_keys)
['im1.jpg', 'im03.jpg', 'im05', 'im11.jpg', 'im21.jpg', 'im31.jpg']
>>> l.sort() # that is what we dont want
['im03.jpg', 'im05', 'im1.jpg', 'im11.jpg', 'im21.jpg', 'im31.jpg']
```

References

- link

2.5.6 Visualizing npz file

tensorlayerx.files.**npz_to_W_pdf**(*path=None*, *regx='w1pre_[0-9]+\\.(npz)'*)

Convert the first weight matrix of *.npz* file to *.pdf* by using *tlx.visualize.W()*.

Parameters

- **path** (*str*) – A folder path to *npz* files.
- **regx** (*str*) – Regx for the file name.

Examples

Convert the first weight matrix of w1_pre...npz file to w1_pre...pdf.

```
>>> tlx.files.npz_to_W_pdf(path='/Users/.../npz_file/', regx='w1pre_[0-9]+\.(npz)
↪')
```

2.6 API - NN

2.6.1 Layer list

Module([name, act])	The basic *Module* class represents a single layer of a neural network.
Sequential(*args)	The class *Sequential* is a linear stack of layers.
ModuleList([modules])	Holds submodules in a list.
ModuleDict([modules])	Holds submodules in a dictionary.
Parameter([name, act])	This function creates a parameter.
ParameterList([parameters])	Holds parameters in a list.
ParameterDict([parameters])	Holds parameters in a dictionary.
Input(shape[, init, dtype, name])	The *Input* class is the starting layer of a neural network.
OneHot([depth, on_value, off_value, axis, ...])	The *OneHot* class is the starting layer of a neural network, see `tf.one_hot`.
Word2vecEmbedding(num_embeddings, embedding_dim)	The *Word2vecEmbedding* class is a fully connected layer.
Embedding(num_embeddings, embedding_dim[, ...])	A simple lookup table that stores embeddings of a fixed dictionary and size.
AverageEmbedding(num_embeddings, embedding_dim)	The *AverageEmbedding* averages over embeddings of inputs.
Linear(out_features[, act, W_init, b_init, ...])	Applies a linear transformation to the incoming data: $y = xA^T + b$
Dropout([p, seed, name])	During training, randomly zeroes some of the elements of the input tensor with probability p using samples from a Bernoulli distribution.

Continued on next page

Table 6 – continued from previous page

GaussianNoise([mean, stddev, is_always, ...])	The GaussianNoise class is noise layer that adding noise with gaussian distribution to the activation.
DropconnectLinear([keep, out_features, act, ...])	The DropconnectLinear class is Dense with DropConnect behaviour which randomly removes connections between this layer and the previous layer according to a keeping probability.
UpSampling2d(scale[, method, antialias, ...])	The UpSampling2d class is a up-sampling 2D layer.
DownSampling2d(scale[, method, antialias, ...])	The DownSampling2d class is down-sampling 2D layer.
Conv1d([out_channels, kernel_size, stride, ...])	Applies a 1D convolution over an input signal composed of several input planes.
Conv2d([out_channels, kernel_size, stride, ...])	Applies a 2D convolution over an input signal composed of several input planes.
Conv3d([out_channels, kernel_size, stride, ...])	Applies a 3D convolution over an input signal composed of several input planes.
ConvTranspose1d([out_channels, kernel_size, ...])	Applies a 1D transposed convolution operator over an input image composed of several input planes.
ConvTranspose2d([out_channels, kernel_size, ...])	Applies a 2D transposed convolution operator over an input image composed of several input planes.
ConvTranspose3d([out_channels, kernel_size, ...])	Applies a 3D transposed convolution operator over an input image composed of several input planes.
DepthwiseConv2d([kernel_size, stride, act, ...])	Separable/Depthwise Convolutional 2D layer, see tf.nn.depthwise_conv2d.
SeparableConv1d([out_channels, kernel_size, ...])	The SeparableConv1d class is a 1D depthwise separable convolutional layer.
SeparableConv2d([out_channels, kernel_size, ...])	The SeparableConv2d class is a 2D depthwise separable convolutional layer.
DeformableConv2d([offset_layer, ...])	The DeformableConv2d class is a 2D Deformable Convolutional Networks.
GroupConv2d([out_channels, kernel_size, ...])	The GroupConv2d class is 2D grouped convolution, see here.
PadLayer([padding, mode, constant_values, name])	The PadLayer class is a padding layer for any mode and dimension.
ZeroPad1d(padding[, name, data_format])	The ZeroPad1d class is a 1D padding layer for signal [batch, length, channel].
ZeroPad2d(padding[, name, data_format])	The ZeroPad2d class is a 2D padding layer for image [batch, height, width, channel].
ZeroPad3d(padding[, name, data_format])	The ZeroPad3d class is a 3D padding layer for volume [batch, depth, height, width, channel].
MaxPool1d([kernel_size, stride, padding, ...])	Max pooling for 1D signal.
AvgPool1d([kernel_size, stride, padding, ...])	Avg pooling for 1D signal.
MaxPool2d([kernel_size, stride, padding, ...])	Max pooling for 2D image.
AvgPool2d([kernel_size, stride, padding, ...])	Avg pooling for 2D image [batch, height, width, channel].
MaxPool3d([kernel_size, stride, padding, ...])	Max pooling for 3D volume.
AvgPool3d([kernel_size, stride, padding, ...])	Avg pooling for 3D volume.
GlobalMaxPool1d([data_format, name])	The GlobalMaxPool1d class is a 1D Global Max Pooling layer.
GlobalAvgPool1d([data_format, name])	The GlobalAvgPool1d class is a 1D Global Avg Pooling layer.

Continued on next page

Table 6 – continued from previous page

GlobalMaxPool2d([data_format, name])	The GlobalMaxPool2d class is a 2D Global Max Pooling layer.
GlobalAvgPool2d([data_format, name])	The GlobalAvgPool2d class is a 2D Global Avg Pooling layer.
GlobalMaxPool3d([data_format, name])	The GlobalMaxPool3d class is a 3D Global Max Pooling layer.
GlobalAvgPool3d([data_format, name])	The GlobalAvgPool3d class is a 3D Global Avg Pooling layer.
AdaptiveAvgPool1d(output_size[, ...])	The AdaptiveAvgPool1d class is a 1D Adaptive Avg Pooling layer.
AdaptiveMaxPool1d(output_size[, ...])	The AdaptiveMaxPool1d class is a 1D Adaptive Max Pooling layer.
AdaptiveAvgPool2d(output_size[, ...])	The AdaptiveAvgPool2d class is a 2D Adaptive Avg Pooling layer.
AdaptiveMaxPool2d(output_size[, ...])	The AdaptiveMaxPool2d class is a 2D Adaptive Max Pooling layer.
AdaptiveAvgPool3d(output_size[, ...])	The AdaptiveAvgPool3d class is a 3D Adaptive Avg Pooling layer.
AdaptiveMaxPool3d(output_size[, ...])	The AdaptiveMaxPool3d class is a 3D Adaptive Max Pooling layer.
CornerPool2d([mode, name])	Corner pooling for 2D image [batch, height, width, channel], see here.
SubpixelConv1d([scale, act, in_channels, name])	It is a 1D sub-pixel up-sampling layer.
SubpixelConv2d([scale, data_format, act, name])	It is a 2D sub-pixel up-sampling layer, usually be used for Super-Resolution applications, see SRGAN for example.
BatchNorm([momentum, epsilon, act, ...])	This interface is used to construct a callable object of the BatchNorm class.
BatchNorm1d([momentum, epsilon, act, ...])	The BatchNorm1d applies Batch Normalization over 2D/3D input (a mini-batch of 1D inputs (optional) with additional channel dimension), of shape (N, C) or (N, L, C) or (N, C, L).
BatchNorm2d([momentum, epsilon, act, ...])	The BatchNorm2d applies Batch Normalization over 4D input (a mini-batch of 2D inputs with additional channel dimension) of shape (N, H, W, C) or (N, C, H, W).
BatchNorm3d([momentum, epsilon, act, ...])	The BatchNorm3d applies Batch Normalization over 5D input (a mini-batch of 3D inputs with additional channel dimension) with shape (N, D, H, W, C) or (N, C, D, H, W).
LayerNorm(normalized_shape[, epsilon, ...])	It implements the function of the Layer Normalization Layer and can be applied to mini-batch input data.
RNNCell(input_size, hidden_size[, bias, ...])	An Elman RNN cell with tanh or ReLU non-linearity.
LSTMCell(input_size, hidden_size[, bias, name])	A long short-term memory (LSTM) cell.
GRUCell(input_size, hidden_size[, bias, name])	A gated recurrent unit (GRU) cell.
RNN(input_size, hidden_size[, num_layers, ...])	Multilayer Elman network(RNN).
LSTM(input_size, hidden_size[, num_layers, ...])	Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.
GRU(input_size, hidden_size[, num_layers, ...])	Applies a multi-layer gated recurrent unit (GRU) RNN to an input sequence.

Continued on next page

Table 6 – continued from previous page

`MultiheadAttention`(embed_dim, num_heads[, ...])	Allows the model to jointly attend to information from different representation subspaces.
`Transformer`([d_model, nhead, ...])	A transformer model.
`TransformerEncoder`(encoder_layer, num_layers)	TransformerEncoder is a stack of N encoder layers
`TransformerDecoder`(decoder_layer, num_layers)	TransformerDecoder is a stack of N decoder layers
`TransformerEncoderLayer`(d_model, nhead, ...)	TransformerEncoderLayer is made up of self-attn and feedforward network.
`TransformerDecoderLayer`(d_model, nhead, ...)	TransformerDecoderLayer is made up of self-attn, multi-head-attn and feedforward network.
`Flatten`([name])	A layer that reshapes high-dimension input into a vector.
`Reshape`(shape[, name])	A layer that reshapes a given tensor.
`Transpose`([perm, conjugate, name])	A layer that transposes the dimension of a tensor.
`Shuffle`(group[, in_channels, name])	A layer that shuffle a 2D image [batch, height, width, channel], see here.
`Concat`([concat_dim, name])	A layer that concats multiple tensors according to given axis.
`Elementwise`([combine_fn, act, name])	A layer that combines multiple `Layer` that have the same output shapes according to an element-wise operation.
`ExpandDims`([axis, name])	The `ExpandDims` class inserts a dimension of 1 into a tensor's shape, see tf.expand_dims() .
`Tile`([multiples, name])	The `Tile` class constructs a tensor by tiling a given tensor, see tf.tile() .
`Stack`([axis, name])	The `Stack` class is a layer for stacking a list of rank-R tensors into one rank-(R+1) tensor, see tf.stack().
`UnStack`([num, axis, name])	The `UnStack` class is a layer for unstacking the given dimension of a rank-R tensor into rank-(R-1) tensors., see tf.unstack().
`Scale`([init_scale, name])	The `Scale` class is to multiple a trainable scale value to the layer outputs.
`BinaryLinear`([out_features, act, use_gemm, ...])	The `BinaryLinear` class is a binary fully connected layer, which weights are either -1 or 1 while inferencing.
`BinaryConv2d`([out_channels, kernel_size, ...])	The `BinaryConv2d` class is a 2D binary CNN layer, which weights are either -1 or 1 while inference.
`TernaryLinear`([out_features, act, use_gemm, ...])	The `TernaryLinear` class is a ternary fully connected layer, which weights are either -1 or 1 or 0 while inference.
`TernaryConv2d`([out_channels, kernel_size, ...])	The `TernaryConv2d` class is a 2D ternary CNN layer, which weights are either -1 or 1 or 0 while inference.
`DorefaLinear`([bitW, bitA, out_features, ...])	The `DorefaLinear` class is a binary fully connected layer, which weights are 'bitW' bits and the output of the previous layer are 'bitA' bits while inferencing.
`DorefaConv2d`([bitW, bitA, out_channels, ...])	The `DorefaConv2d` class is a 2D quantized convolutional layer, which weights are 'bitW' bits and the output of the previous layer are 'bitA' bits while inferencing.
`MaskedConv3d`(mask_type, out_channels[, ...])	MaskedConv3D.

2.6.2 Base Layer

Module

class tensorlayerx.nn.**Module**(*name=None*, *act=None*, **args*, ***kwargs*)

The basic *Module* class represents a single layer of a neural network. It should be subclassed when implementing new types of layers. :param name: A unique layer name. If None, a unique name will be automatically assigned. :type name: str or None

__init__()

Initializing the Layer.

__call__()

Forwarding the computation.

all_weights()

Return a list of Tensor which are all weights of this Layer.

trainable_weights()

Return a list of Tensor which are all trainable weights of this Layer.

nontrainable_weights()

Return a list of Tensor which are all nontrainable weights of this Layer.

build()

Abstract method. Build the Layer. All trainable weights should be defined in this function.

_get_weights()

Abstract method.Create weights for training parameters.

save_weights()

Input file_path, save model weights into a file of given format.

load_weights()

Load model weights from a given file, which should be previously saved by self.save_weights().

save_standard_weights()

Input file_path, save model weights into a npz_dict file. These parameters can support multiple backends.

load_standard_weights()

Load model weights from a given file, which should be previously saved by self.save_standard_weights().

forward()

Abstract method. Forward computation and return computation results.

Sequential

class tensorlayerx.nn.**Sequential**(**args*)

The class *Sequential* is a linear stack of layers. The *Sequential* can be created by passing a list of layer instances. The given layer instances will be automatically connected one by one. :param layers: A list of layers. :type layers: list of Layer :param name: A unique layer name. If None, a unique name will be automatically assigned. :type name: str or None

__init__()

Initializing the ModuleList.

weights()

A collection of weights of all the layer instances.

build()

Build the ModuleList. The layer instances will be connected automatically one by one.

forward()
Forward the computation. The computation will go through all layer instances.

Examples

```
>>> conv = tlx.layers.Conv2d(3, 2, 3, pad_mode='valid')
>>> bn = tlx.layers.BatchNorm2d(2)
>>> seq = tlx.nn.Sequential([conv, bn])
>>> x = tlx.layers.Input((1, 3, 4, 4))
>>> seq(x)
```

ModuleList

class tensorlayerx.nn.**ModuleList**(*modules=None*)
Holds submodules in a list.

ModuleList can be used like a regular Python list, support '__getitem__', '__setitem__', '__delitem__', '__len__', '__iter__' and '__iadd__', but module it contains are properly registered, and will be visible by all Modules methods.

Parameters **args** (*list*) – List of subclass of Module.

__init__()
Initializing the ModuleList.

insert()
Inserts a given layer before a given index in the list.

extend()
Appends layers from a Python iterable to the end of the list.

append()
Appends a given layer to the end of the list.

Examples

```
>>> from tensorlayerx.nn import Module, ModuleList, Linear
>>> import tensorlayerx as tlx
>>> d1 = Linear(out_features=800, act=tlx.ReLU, in_features=784, name='linear1')
>>> d2 = Linear(out_features=800, act=tlx.ReLU, in_features=800, name='linear2')
>>> d3 = Linear(out_features=10, act=tlx.ReLU, in_features=800, name='linear3')
>>> layer_list = ModuleList([d1, d2])
>>> # Inserts a given d2 before a given index in the list
>>> layer_list.insert(1, d2)
>>> layer_list.insert(2, d2)
>>> # Appends d2 from a Python iterable to the end of the list.
>>> layer_list.extend([d2])
>>> # Appends a given d3 to the end of the list.
>>> layer_list.append(d3)
```

ModuleDict

class tensorlayerx.nn.**ModuleDict**(*modules=None*)
Holds submodules in a dictionary.

ModuleDict can be used like a regular Python dictionary, support '__getitem__', '__setitem__', '__delitem__', '__len__', '__iter__' and '__contains__', but module it contains are properly registered, and will be visible by all Modules methods.

Parameters **args** (*dict*) – a mapping (dictionary) of (string: module) or an iterable of key-value pairs of type (string, module)

__init__()
Initializing the ModuleDict.

clear()
Remove all items from the ModuleDict.

pop()
Remove key from the ModuleDict and return its module.

keys()
Return an iterable of the ModuleDict keys.

items()
Return an iterable of the ModuleDict key/value pairs.

values()
Return an iterable of the ModuleDict values.

update()
Update the ModuleDict with the key-value pairs from a mapping or an iterable, overwriting existing keys.

Examples

```
>>> from tensorlayerx.nn import Module, ModuleDict, Linear
>>> import tensorlayerx as tlx
>>> class MyModule(Module):
>>>     def __init__(self):
>>>         super(MyModule, self).__init__()
>>>         self.dict = ModuleDict({
>>>                 'linear1':Linear(out_features=800, act=tlx.ReLU, in_
↪features=784, name='linear1'),
>>>                 'linear2':Linear(out_features=800, act=tlx.ReLU, in_
↪features=800, name='linear2')
>>>                 })
>>>     def forward(self, x, linear):
>>>         x = self.dict[linear](x)
>>>         return x
```

Parameter

tensorlayerx.nn.**Parameter**(*data=None*, *name=None*)
This function creates a parameter. The parameter is a learnable variable, which can have gradient, and can be optimized.

Parameters

- **data** (*Tensor*) – parameter tensor
- **requires_grad** (*bool*) – if the parameter requires gradient. Default: True

Returns

Return type Parameter

Examples

```
>>> import tensorlayerx as tlx
>>> para = tlx.nn.Parameter(data=tlx.ones((5,5)), requires_grad=True)
```

ParameterList

class tensorlayerx.nn.**ParameterList**(*parameters=None*)

Holds parameters in a list.

ParameterList can be indexed like a regular Python list. Support '__getitem__', '__setitem__', '__delitem__', '__len__', '__iter__' and '__iadd__'.

Parameters **Parameters** (*list*) – List of Parameter.

__init__()

Initializing the ParameterList.

extend(*parameter*)

Appends parameters from a Python iterable to the end of the list.

append(*parameters*)

Appends a given parameter to the end of the list.

Examples

```
>>> from tensorlayerx.nn import Module, ModuleList, Linear
>>> import tensorlayerx as tlx
>>> class MyModule(Module):
>>>     def __init__(self):
>>>         super(MyModule, self).__init__()
>>>         self.params2 = ParameterList([Parameter(tlx.ones((10,5))),␣
↪Parameter(tlx.ones((5,10)))])
>>>     def forward(self, x):
>>>         x = tlx.matmul(x, self.params2[0])
>>>         x = tlx.matmul(x, self.params2[1])
>>>         return x
```

ParameterDict

class tensorlayerx.nn.**ParameterDict**(*parameters=None*)

Holds parameters in a dictionary.

ParameterDict can be used like a regular Python dictionary, support '__getitem__', '__setitem__', '__delitem__', '__len__', '__iter__' and '__contains__',

Parameters **parameters** (*dict*) – a mapping (dictionary) of (string: parameter) or an iterable of key-value pairs of type (string, parameter)

__init__()

Initializing the ParameterDict.

clear()
Remove all items from the ParameterDict.

setdefault(*key*, *default=None*)
If key is in the ParameterDict, return its parameter. If not, insert *key* with a parameter *default* and return *default*. *default* defaults to *None*.

popitem()
Remove and return the last inserted *(key, parameter)* pair from the ParameterDict

pop(*key*)
Remove key from the ParameterDict and return its parameter.

get(key, default = None):
Return the parameter associated with key if present. Otherwise return default if provided, None if not.

fromkeys(*keys*, *default = None*)
Return a new ParameterDict with the keys provided

keys()
Return an iterable of the ParameterDict keys.

items()
Return an iterable of the ParameterDict key/value pairs.

values()
Return an iterable of the ParameterDict values.

update()
Update the ParameterDict with the key-value pairs from a mapping or an iterable, overwriting existing keys.

Examples

```
>>> from tensorlayerx.nn import Module, ParameterDict, Parameter
>>> import tensorlayerx as tlx
>>> class MyModule(Module):
>>>     def __init__(self):
>>>         super(MyModule, self).__init__()
>>>         self.dict = ParameterDict({
>>>                 'left': Parameter(tlx.ones((5, 10))),
>>>                 'right': Parameter(tlx.zeros((5, 10)))
>>>                 })
>>>     def forward(self, x, choice):
>>>         x = tlx.matmul(x, self.dict[choice])
>>>         return x
```

2.6.3 Input Layers

Input Layer

tensorlayerx.nn.**Input**(*shape*, *init=None*, *dtype=tensorflow.float32*, *name=None*)
The *Input* class is the starting layer of a neural network.

Parameters

- **shape** (*tuple (int)*) – Including batch size.

- **init** (*initializer or str or None*) – The initializer for initializing the input matrix
- **dtype** (*dtype*) – The type of input values. By default, tf.float32.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayer

```
>>> ni = tlx.nn.Input([10, 50, 50, 32], name='input')
>>> output shape : [10, 50, 50, 32]
```

One-hot Layer

class tensorlayerx.nn.**OneHot**(*depth=None*, *on_value=1.0*, *off_value=0.0*, *axis=-1*, *dtype=tensorflow.float32*, *name=None*)

The *OneHot* class is the starting layer of a neural network, see `tf.one_hot`. Useful link: *https://www.tensorflow.org/api_docs/python/tf/one_hot*.

Parameters

- **depth** (*None or int*) – If the input indices is rank N, the output will have rank N+1. The new axis is created at dimension *axis* (default: the new axis is appended at the end).
- **on_value** (*None or number*) – The value to represnt *ON*. If None, it will default to the value 1.
- **off_value** (*None or number*) – The value to represnt *OFF*. If None, it will default to the value 0.
- **axis** (*None or int*) – The axis.
- **dtype** (*None or TensorFlow dtype*) – The data type, None means tlx.float32.
- **name** (*str*) – A unique layer name.

Examples

```
>>> net = tlx.nn.Input([32], dtype=tlx.int32)
>>> onehot = tlx.nn.OneHot(depth=8)
>>> print(onehot)
OneHot(depth=8, name='onehot')
>>> tensor = tlx.nn.OneHot(depth=8)(net)
>>> print(tensor)
Tensor([...], shape=(32, 8), dtype=float32)
```

Word2Vec Embedding Layer

class tensorlayerx.nn.**Word2vecEmbedding**(*num_embeddings*, *embedding_dim*, *num_sampled=64*, *activate_nce_loss=True*, *nce_loss_args=None*, *E_init='random_uniform'*, *nce_W_init='truncated_normal'*, *nce_b_init='constant'*, *name=None*)

The *Word2vecEmbedding* class is a fully connected layer. For Word Embedding, words are input as integer

index. The output is the embedded word vector.

The layer integrates NCE loss by default (activate_nce_loss=True). If the NCE loss is activated, in a dynamic model, the computation of nce loss can be turned off in customised forward feeding by setting use_nce_loss=False when the layer is called. The NCE loss can be deactivated by setting activate_nce_loss=False.

Parameters

- **num_embeddings** (*int*) – size of the dictionary of embeddings.
- **embedding_dim** (*int*) – the size of each embedding vector.
- **num_sampled** (*int*) – The number of negative examples for NCE loss
- **activate_nce_loss** (*boolean*) – Whether activate nce loss or not. By default, True If True, the layer will return both outputs of embedding and nce_cost in forward feeding. If False, the layer will only return outputs of embedding. In a dynamic model, the computation of nce loss can be turned off in forward feeding by setting use_nce_loss=False when the layer is called. In a static model, once the model is constructed, the computation of nce loss cannot be changed (always computed or not computed).
- **nce_loss_args** (*dictionary*) – The arguments for tf.ops.nce_loss()
- **E_init** (*initializer or str*) – The initializer for initializing the embedding matrix
- **nce_W_init** (*initializer or str*) – The initializer for initializing the nce decoder weight matrix
- **nce_b_init** (*initializer or str*) – The initializer for initializing of the nce decoder bias vector
- **name** (*str*) – A unique layer name

outputs
The embedding layer outputs.

Type Tensor

normalized_embeddings
Normalized embedding matrix.

Type Tensor

nce_weights
The NCE weights only when activate_nce_loss is True.

Type Tensor

nce_biases
The NCE biases only when activate_nce_loss is True.

Type Tensor

Examples

Word2Vec With TensorLayer (Example in *examples/text_word_embedding/tutorial_word2vec_basic.py*)

```
>>> import tensorlayerx as tlx
>>> batch_size = 8
>>> embedding_dim = 50
```

(continues on next page)

(continued from previous page)

```
>>> inputs = tlx.nn.Input([batch_size], dtype=tlx.int32)
>>> labels = tlx.nn.Input([batch_size, 1], dtype=tlx.int32)
>>> emb_net = tlx.nn.Word2vecEmbedding(
>>>     num_embeddings=10000,
>>>     embedding_dim=embedding_dim,
>>>     num_sampled=100,
>>>     activate_nce_loss=True, # the nce loss is activated
>>>     nce_loss_args={},
>>>     E_init=tlx.initializers.random_uniform(minval=-1.0, maxval=1.0),
>>>     nce_W_init=tlx.initializers.truncated_normal(stddev=float(1.0 / np.
↪sqrt(embedding_dim))),
>>>     nce_b_init=tlx.initializers.constant(value=0.0),
>>>     name='word2vec_layer',
>>> )
>>> print(emb_net)
Word2vecEmbedding(num_embeddings=10000, embedding_dim=50, num_sampled=100,␣
↪activate_nce_loss=True, nce_loss_args={})
>>> embed_tensor = emb_net(inputs, use_nce_loss=False) # the nce loss is turned␣
↪off and no need to provide labels
>>> embed_tensor = emb_net([inputs, labels], use_nce_loss=False) # the nce loss␣
↪is turned off and the labels will be ignored
>>> embed_tensor, embed_nce_loss = emb_net([inputs, labels]) # the nce loss is␣
↪calculated
>>> outputs = tlx.nn.Linear(out_features=10, name="linear")(embed_tensor)
>>> model = tlx.model.Model(inputs=[inputs, labels], outputs=[outputs, embed_nce_
↪loss], name="word2vec_model") # a static model
>>> out = model([data_x, data_y], is_train=True) # where data_x is inputs and␣
↪data_y is labels
```

References

https://www.tensorflow.org/tutorials/representation/word2vec

Embedding Layer

class tensorlayerx.nn.**Embedding**(*num_embeddings*, *embedding_dim*, *E_init='random_uniform'*, *name=None*)

A simple lookup table that stores embeddings of a fixed dictionary and size.

This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.

Parameters

- **num_embeddings** (*int*) – size of the dictionary of embeddings.
- **embedding_dim** (*int*) – the size of each embedding vector.
- **E_init** (*initializer or str*) – The initializer for the embedding matrix.
- **E_init_args** (*dictionary*) – The arguments for embedding matrix initializer.
- **name** (*str*) – A unique layer name.

outputs

The embedding layer output is a 3D tensor in the shape: (batch_size, num_steps(num_words), embedding_dim).

Type tensor

Examples

```
>>> import tensorlayerx as tlx
>>> input = tlx.nn.Input([8, 100], dtype=tlx.int32)
>>> embed = tlx.nn.Embedding(num_embeddings=1000, embedding_dim=50, name='embed')
>>> print(embed)
Embedding(num_embeddings=1000, embedding_dim=50)
>>> tensor = embed(input)
>>> print(tensor)
Tensor([...], shape=(8, 100, 50), dtype=float32)
```

Average Embedding Layer

class tensorlayerx.nn.**AverageEmbedding**(*num_embeddings*, *embedding_dim*, *pad_value=0*, *E_init='random_uniform'*, *name=None*)

The *AverageEmbedding* averages over embeddings of inputs. This is often used as the input layer for model like DAN[1] and FastText[2].

Parameters

- **num_embeddings** (*int*) – size of the dictionary of embeddings.
- **embedding_dim** (*int*) – the size of each embedding vector.
- **pad_value** (*int*) – The scalar padding value used in inputs, 0 as default.
- **E_init** (*initializer or str*) – The initializer of the embedding matrix.
- **name** (*str*) – A unique layer name.

outputs

The embedding layer output is a 2D tensor in the shape: (batch_size, embedding_dim).

Type tensor

References

- [1] Iyyer, M., Manjunatha, V., Boyd-Graber, J., & Daum'e III, H. (2015). Deep Unordered Composition Rivals Syntactic Methods for Text Classification. In Association for Computational Linguistics.
- [2] Joulin, A., Grave, E., Bojanowski, P., & Mikolov, T. (2016). Bag of Tricks for Efficient Text Classification.

Examples

```
>>> import tensorlayerx as tlx
>>> batch_size = 8
>>> length = 5
>>> input = tlx.nn.Input([batch_size, length], dtype=tlx.int32)
>>> avgembed = tlx.nn.AverageEmbedding(num_embeddings=1000, embedding_dim=50,␣
↪name='avg')
>>> print(avgembed)
AverageEmbedding(num_embeddings=1000, embedding_dim=50, pad_value=0)
```

(continues on next page)

(continued from previous page)

```
>>> tensor = avgembed(input)
>>> print(tensor)
Tensor([...], shape=(8, 50), dtype=float32)
```

2.6.4 Convolutional Layers

Convolutions

Conv1d

class tensorlayerx.nn.**Conv1d**(*out_channels=32*, *kernel_size=5*, *stride=1*, *act=None*, *padding='SAME'*, *data_format='channels_last'*, *dilation=1*, *W_init='truncated_normal'*, *b_init='constant'*, *in_channels=None*, *name=None*)

Applies a 1D convolution over an input signal composed of several input planes.

Parameters

- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int*) – The kernel size
- **stride** (*int*) – The stride step
- **dilation** (*int*) – Specifying the dilation rate to use for dilated convolution.
- **act** (*activation function*) – The function that is applied to the layer activations
- **padding** (*str or int*) – The padding algorithm type: "SAME" or "VALID".
- **data_format** (*str*) – "channel_last" (NWC, default) or "channels_first" (NCW).
- **W_init** (*initializer or str*) – The initializer for the kernel weight matrix.
- **b_init** (*initializer or None or str*) – The initializer for the bias vector. If None, skip biases.
- **in_channels** (*int*) – The number of in channels.
- **name** (*None or str*) – A unique layer name

Examples

With TensorLayerx

```
>>> net = tlx.nn.Input([8, 100, 1], name='input')
>>> conv1d = tlx.nn.Conv1d(out_channels =32, kernel_size=5, stride=2, b_init=None,
↪ in_channels=1, name='conv1d_1')
>>> print(conv1d)
>>> tensor = tlx.nn.Conv1d(out_channels =32, kernel_size=5, stride=2, act=tlx.
↪ReLU, name='conv1d_2')(net)
>>> print(tensor)
```

Conv2d

class tensorlayerx.nn.**Conv2d**(*out_channels=32*, *kernel_size=(3, 3)*, *stride=(1, 1)*, *act=None*, *padding='SAME'*, *data_format='channels_last'*, *dilation=(1, 1)*, *W_init='truncated_normal'*, *b_init='constant'*, *in_channels=None*, *name=None*)

Applies a 2D convolution over an input signal composed of several input planes.

Parameters

- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*tuple or int*) – The kernel size (height, width).
- **stride** (*tuple or int*) – The sliding window stride of corresponding input dimensions. It must be in the same order as the `shape` parameter.
- **dilation** (*tuple or int*) – Specifying the dilation rate to use for dilated convolution.
- **act** (*activation function*) – The activation function of this layer.
- **padding** (*int, tuple or str*) – The padding algorithm type: "SAME" or "VALID". If padding is int or tuple, padding added to all four sides of the input. Default: 'SAME'
- **data_format** (*str*) – "channels_last" (NHWC, default) or "channels_first" (NCHW).
- **W_init** (*initializer or str*) – The initializer for the the kernel weight matrix.
- **b_init** (*initializer or None or str*) – The initializer for the the bias vector. If None, skip biases.
- **in_channels** (*int*) – The number of in channels.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerx

```
>>> net = tlx.nn.Input([8, 400, 400, 3], name='input')
>>> conv2d = tlx.nn.Conv2d(out_channels =32, kernel_size=(3, 3), stride=(2, 2), b_
↪init=None, in_channels=3, name='conv2d_1')
>>> print(conv2d)
>>> tensor = tlx.nn.Conv2d(out_channels =32, kernel_size=(3, 3), stride=(2, 2),␣
↪act=tlx.ReLU, name='conv2d_2')(net)
>>> print(tensor)
```

Conv3d

class tensorlayerx.nn.**Conv3d**(*out_channels=32*, *kernel_size=(3, 3, 3)*, *stride=(1, 1, 1)*, *act=None*, *padding='SAME'*, *data_format='channels_last'*, *dilation=(1, 1, 1)*, *W_init='truncated_normal'*, *b_init='constant'*, *in_channels=None*, *name=None*)

Applies a 3D convolution over an input signal composed of several input planes.

Parameters

- **out_channels** (*int*) – Number of channels produced by the convolution

- **kernel_size** (*tuple or int*) – The kernel size (depth, height, width).
- **stride** (*tuple or int*) – The sliding window stride of corresponding input dimensions. It must be in the same order as the `shape` parameter.
- **dilation** (*tuple or int*) – Specifying the dilation rate to use for dilated convolution.
- **act** (*activation function*) – The activation function of this layer.
- **padding** (*int, tuple or str*) – The padding algorithm type: "SAME" or "VALID".
- **data_format** (*str*) – "channels_last" (NDHWC, default) or "channels_first" (NCDHW).
- **W_init** (*initializer or str*) – The initializer for the the kernel weight matrix.
- **b_init** (*initializer or None or str*) – The initializer for the the bias vector. If None, skip biases.
- **in_channels** (*int*) – The number of in channels.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerx

```
>>> net = tlx.nn.Input([8, 20, 20, 20, 3], name='input')
>>> conv3d = tlx.nn.Conv3d(out_channels =32, kernel_size=(3, 3, 3), stride=(2, 2,␣
↪2), b_init=None, in_channels=3, name='conv3d_1')
>>> print(conv3d)
>>> tensor = tlx.nn.Conv3d(out_channels =32, kernel_size=(3, 3, 3), stride=(2, 2,␣
↪2), act=tlx.ReLU, name='conv3d_2')(net)
>>> print(tensor)
```

Deconvolutions

ConvTranspose1d

class tensorlayerx.nn.**ConvTranspose1d**(*out_channels=32*, *kernel_size=15*, *stride=1*, *act=None*, *padding='SAME'*, *data_format='channels_last'*, *dilation=1*, *W_init='truncated_normal'*, *b_init='constant'*, *in_channels=None*, *name=None*)

Applies a 1D transposed convolution operator over an input image composed of several input planes.

Parameters

- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int*) – The kernel size
- **stride** (*int or list*) – An int or list of *ints* that has length *1* or *3*. The number of entries by which the filter is moved right at each step.
- **dilation** (*int or list*) – Specifying the dilation rate to use for dilated convolution.
- **act** (*activation function*) – The function that is applied to the layer activations
- **padding** (*str*) – The padding algorithm type: "SAME" or "VALID".

- **data_format** (*str*) – "channel_last" (NWC, default) or "channels_first" (NCW).
- **W_init** (*initializer or str*) – The initializer for the kernel weight matrix.
- **b_init** (*initializer or None or str*) – The initializer for the bias vector. If None, skip biases.
- **in_channels** (*int*) – The number of in channels.
- **name** (*None or str*) – A unique layer name

Examples

With TensorLayerx

```
>>> net = tlx.nn.Input([8, 100, 1], name='input')
>>> conv1d = tlx.nn.ConvTranspose1d(out_channels=32, kernel_size=5, stride=2, b_
↪init=None, in_channels=1, name='Deonv1d_1')
>>> print(conv1d)
>>> tensor = tlx.nn.ConvTranspose1d(out_channels=32, kernel_size=5, stride=2,␣
↪act=tlx.ReLU, name='ConvTranspose1d_2')(net)
>>> print(tensor)
```

ConvTranspose2d

class tensorlayerx.nn.**ConvTranspose2d**(*out_channels=32*, *kernel_size=(3, 3)*, *stride=(1, 1)*, *act=None*, *padding='SAME'*, *data_format='channels_last'*, *dilation=(1, 1)*, *W_init='truncated_normal'*, *b_init='constant'*, *in_channels=None*, *name=None*)

Applies a 2D transposed convolution operator over an input image composed of several input planes.

Parameters

- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*tuple or int*) – The kernel size (height, width).
- **stride** (*tuple or int*) – The sliding window stride of corresponding input dimensions. It must be in the same order as the `shape` parameter.
- **dilation** (*tuple or int*) – Specifying the dilation rate to use for dilated convolution.
- **act** (*activation function*) – The activation function of this layer.
- **padding** (*int, tuple or str*) – The padding algorithm type: "SAME" or "VALID".
- **data_format** (*str*) – "channels_last" (NHWC, default) or "channels_first" (NCHW).
- **W_init** (*initializer or str*) – The initializer for the the kernel weight matrix.
- **b_init** (*initializer or None or str*) – The initializer for the the bias vector. If None, skip biases.
- **in_channels** (*int*) – The number of in channels.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerx

```
>>> net = tlx.nn.Input([8, 400, 400, 3], name='input')
>>> conv2d_transpose = tlx.nn.ConvTranspose2d(out_channels=32, kernel_size=(3, 3),
↪ stride=(2, 2), b_init=None, in_channels=3, name='conv2d_transpose_1')
>>> print(conv2d_transpose)
>>> tensor = tlx.nn.ConvTranspose2d(out_channels=32, kernel_size=(3, 3),␣
↪stride=(2, 2), act=tlx.ReLU, name='conv2d_transpose_2')(net)
>>> print(tensor)
```

ConvTranspose3d

class tensorlayerx.nn.**ConvTranspose3d**(*out_channels=32*, *kernel_size=(3, 3, 3)*, *stride=(1, 1, 1)*, *act=None*, *padding='SAME'*, *data_format='channels_last'*, *dilation=(1, 1, 1)*, *W_init='truncated_normal'*, *b_init='constant'*, *in_channels=None*, *name=None*)

Applies a 3D transposed convolution operator over an input image composed of several input planes.

Parameters

- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*tuple or int*) – The kernel size (depth, height, width).
- **stride** (*tuple or int*) – The sliding window stride of corresponding input dimensions. It must be in the same order as the `shape` parameter.
- **dilation** (*tuple or int*) – Specifying the dilation rate to use for dilated convolution.
- **act** (*activation function*) – The activation function of this layer.
- **padding** (*str*) – The padding algorithm type: "SAME" or "VALID".
- **data_format** (*str*) – "channels_last" (NDHWC, default) or "channels_first" (NCDHW).
- **W_init** (*initializer or str*) – The initializer for the the kernel weight matrix.
- **b_init** (*initializer or None or str*) – The initializer for the the bias vector. If None, skip biases.
- **in_channels** (*int*) – The number of in channels.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerx

```
>>> net = tlx.nn.Input([8, 20, 20, 20, 3], name='input')
>>> ConvTranspose3d = tlx.nn.ConvTranspose3d(out_channels=32, kernel_size=(3, 3,␣
↪3), stride=(2, 2, 2), b_init=None, in_channels=3, name='deconv3d_1')
>>> print(deconv3d)
>>> tensor = tlx.nn.ConvTranspose3d(out_channels=32, kernel_size=(3, 3, 3),␣
↪stride=(2, 2, 2), act=tlx.ReLU, name='ConvTranspose3d_2')(net)
>>> print(tensor)
```

Deformable Convolutions

DeformableConv2d

class tensorlayerx.nn.**DeformableConv2d**(*offset_layer=None*, *out_channels=32*, *kernel_size=(3, 3)*, *act=None*, *padding='SAME'*, *W_init='truncated_normal'*, *b_init='constant'*, *in_channels=None*, *name=None*)

The `DeformableConv2d` class is a 2D Deformable Convolutional Networks.

Parameters

- **offset_layer** (*tlx.Tensor*) – To predict the offset of convolution operations. The shape is (batchsize, input height, input width, 2*(number of element in the convolution kernel)) e.g. if apply a 3*3 kernel, the number of the last dimension should be 18 (2*3*3)
- **out_channels** (*int*) – The number of filters.
- **kernel_size** (*tuple or int*) – The filter size (height, width).
- **act** (*activation function*) – The activation function of this layer.
- **padding** (*str*) – The padding algorithm type: "SAME" or "VALID".
- **W_init** (*initializer or str*) – The initializer for the weight matrix.
- **b_init** (*initializer or None or str*) – The initializer for the bias vector. If None, skip biases.
- **in_channels** (*int*) – The number of in channels.
- **name** (*str*) – A unique layer name.

Examples

With TensorLayer

```
>>> net = tlx.nn.Input([5, 10, 10, 16], name='input')
>>> offset1 = tlx.nn.Conv2d(
...     out_channels=18, kernel_size=(3, 3), strides=(1, 1), padding='SAME', name=
↪'offset1'
... )(net)
>>> deformconv1 = tlx.nn.DeformableConv2d(
...     offset_layer=offset1, out_channels=32, kernel_size=(3, 3), name=
↪'deformable1'
... )(net)
>>> offset2 = tlx.nn.Conv2d(
...     out_channels=18, kernel_size=(3, 3), strides=(1, 1), padding='SAME', name=
↪'offset2'
... )(deformconv1)
>>> deformconv2 = tlx.nn.DeformableConv2d(
...     offset_layer=offset2, out_channels=64, kernel_size=(3, 3), name=
↪'deformable2'
... )(deformconv1)
```

References

- The deformation operation was adapted from the implementation in here

Notes

- The padding is fixed to 'SAME'.
- The current implementation is not optimized for memory usgae. Please use it carefully.

Depthwise Convolutions

DepthwiseConv2d

class tensorlayerx.nn.**DepthwiseConv2d**(*kernel_size=(3, 3)*, *stride=(1, 1)*, *act=None*, *padding='SAME'*, *data_format='channels_last'*, *dilation=(1, 1)*, *depth_multiplier=1*, *W_init='truncated_normal'*, *b_init='constant'*, *in_channels=None*, *name=None*)

Separable/Depthwise Convolutional 2D layer, see tf.nn.depthwise_conv2d.

Input: 4-D Tensor (batch, height, width, in_channels).

Output: 4-D Tensor (batch, new height, new width, in_channels * depth_multiplier).

Parameters

- **kernel_size** (*tuple or int*) – The filter size (height, width).
- **stride** (*tuple or int*) – The stride step (height, width).
- **act** (*activation function*) – The activation function of this layer.
- **padding** (*str*) – The padding algorithm type: "SAME" or "VALID".
- **data_format** (*str*) – "channels_last" (NHWC, default) or "channels_first" (NCHW).
- **dilation** (*tuple or int*) – The dilation rate in which we sample input values across the height and width dimensions in atrous convolution. If it is greater than 1, then all values of strides must be 1.
- **depth_multiplier** (*int*) – The number of channels to expand to.
- **W_init** (*initializer or str*) – The initializer for the weight matrix.
- **b_init** (*initializer or None or str*) – The initializer for the bias vector. If None, skip bias.
- **in_channels** (*int*) – The number of in channels.
- **name** (*str*) – A unique layer name.

Examples

With TensorLayer

```
>>> net = tlx.nn.Input([8, 200, 200, 32], name='input')
>>> depthwiseconv2d = tlx.nn.DepthwiseConv2d(
...     kernel_size=(3, 3), stride=(1, 1), dilation=(2, 2), act=tlx.ReLU, depth_
↪multiplier=2, name='depthwise'
... )(net)
>>> print(depthwiseconv2d)
>>> output shape : (8, 200, 200, 64)
```

References

- tflearn's grouped_conv_2d
- keras's separableconv2d

Group Convolutions

GroupConv2d

class tensorlayerx.nn.**GroupConv2d**(*out_channels=32*, *kernel_size=(1, 1)*, *stride=(1, 1)*, *n_group=1*, *act=None*, *padding='SAME'*, *data_format='channels_last'*, *dilation=(1, 1)*, *W_init='truncated_normal'*, *b_init='constant'*, *in_channels=None*, *name=None*)

The *GroupConv2d* class is 2D grouped convolution, see here.

Parameters

- **out_channels** (*int*) – The number of filters.
- **kernel_size** (*tuple or int*) – The filter size.
- **stride** (*tuple or int*) – The stride step.
- **n_group** (*int*) – The number of groups.
- **act** (*activation function*) – The activation function of this layer.
- **padding** (*str*) – The padding algorithm type: "SAME" or "VALID".
- **data_format** (*str*) – "channels_last" (NHWC, default) or "channels_first" (NCHW).
- **dilation** (*tuple or int*) – Specifying the dilation rate to use for dilated convolution.
- **W_init** (*initializer or str*) – The initializer for the weight matrix.
- **b_init** (*initializer or None or str*) – The initializer for the bias vector. If None, skip biases.
- **in_channels** (*int*) – The number of in channels.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayer

```
>>> net = tlx.nn.Input([8, 24, 24, 32], name='input')
>>> groupconv2d = tlx.nn.GroupConv2d(
...     out_channels=64, kernel_size=(3, 3), stride=(2, 2), n_group=2, name='group
↪'
... )(net)
>>> print(groupconv2d)
>>> output shape : (8, 12, 12, 64)
```

Separable Convolutions

SeparableConv1d

class tensorlayerx.nn.**SeparableConv1d**(*out_channels=32*, *kernel_size=1*, *stride=1*, *act=None*, *padding='SAME'*, *data_format='channels_last'*, *dilation=1*, *depth_multiplier=1*, *depthwise_init='truncated_normal'*, *pointwise_init='truncated_normal'*, *b_init='constant'*, *in_channels=None*, *name=None*)

The *SeparableConv1d* class is a 1D depthwise separable convolutional layer. This layer performs a depthwise convolution that acts separately on channels, followed by a pointwise convolution that mixes channels.

Parameters

- **out_channels** (*int*) – The dimensionality of the output space (i.e. the number of filters in the convolution).
- **kernel_size** (*int*) – Specifying the spatial dimensions of the filters. Can be a single integer to specify the same value for all spatial dimensions.
- **stride** (*int*) – Specifying the stride of the convolution. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any dilation value != 1.
- **act** (*activation function*) – The activation function of this layer.
- **padding** (*str*) – One of "valid" or "same" (case-insensitive).
- **data_format** (*str*) – One of channels_last (default) or channels_first. The ordering of the dimensions in the inputs. channels_last corresponds to inputs with shape (batch, height, width, channels) while channels_first corresponds to inputs with shape (batch, channels, height, width).
- **dilation** (*int*) – Specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any dilation value != 1 is incompatible with specifying any stride value != 1.
- **depth_multiplier** (*int*) – The number of depthwise convolution output channels for each input channel. The total number of depthwise convolution output channels will be equal to num_filters_in * depth_multiplier.
- **depthwise_init** (*initializer or str*) – for the depthwise convolution kernel.
- **pointwise_init** (*initializer or str*) – For the pointwise convolution kernel.
- **b_init** (*initializer or str*) – For the bias vector. If None, ignore bias in the pointwise part only.
- **in_channels** (*int*) – The number of in channels.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([8, 50, 64], name='input')
>>> separableconv1d = tlx.nn.SeparableConv1d(out_channels=32, kernel_size=3,␣
↪stride=2, padding='SAME', act=tlx.ReLU, name='separable_1d')(net)
>>> print(separableconv1d)
>>> output shape : (8, 25, 32)
```

SeparableConv2d

class tensorlayerx.nn.**SeparableConv2d**(*out_channels=32*, *kernel_size=(1, 1)*, *stride=(1, 1)*, *act=None*, *padding='VALID'*, *data_format='channels_last'*, *dilation=(1, 1)*, *depth_multiplier=1*, *depthwise_init='truncated_normal'*, *pointwise_init='truncated_normal'*, *b_init='constant'*, *in_channels=None*, *name=None*)

The *SeparableConv2d* class is a 2D depthwise separable convolutional layer. This layer performs a depthwise convolution that acts separately on channels, followed by a pointwise convolution that mixes channels.

Parameters

- **out_channels** (*int*) – The dimensionality of the output space (i.e. the number of filters in the convolution).
- **kernel_size** (*tuple or int*) – Specifying the spatial dimensions of the filters. Can be a single integer to specify the same value for all spatial dimensions.
- **stride** (*tuple or int*) – Specifying the stride of the convolution. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any dilation value != 1.
- **act** (*activation function*) – The activation function of this layer.
- **padding** (*str*) – One of "valid" or "same" (case-insensitive).
- **data_format** (*str*) – One of channels_last (default) or channels_first. The ordering of the dimensions in the inputs. channels_last corresponds to inputs with shape (batch, height, width, channels) while channels_first corresponds to inputs with shape (batch, channels, height, width).
- **dilation** (*tuple or int*) – Specifying the dilation rate to use for dilated convolution. Can be a single integer to specify the same value for all spatial dimensions. Currently, specifying any dilation value != 1 is incompatible with specifying any stride value != 1.
- **depth_multiplier** (*int*) – The number of depthwise convolution output channels for each input channel. The total number of depthwise convolution output channels will be equal to num_filters_in * depth_multiplier.
- **depthwise_init** (*initializer or str*) – for the depthwise convolution kernel.
- **pointwise_init** (*initializer or str*) – For the pointwise convolution kernel.
- **b_init** (*initializer or str*) – For the bias vector. If None, ignore bias in the pointwise part only.
- **in_channels** (*int*) – The number of in channels.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([8, 50, 50, 64], name='input')
>>> separableconv2d = tlx.nn.SeparableConv2d(out_channels=32, kernel_size=(3,3),␣
↪stride=(2,2), depth_multiplier = 3 , padding='SAME', act=tlx.ReLU, name=
↪'separable_2d')(net)
>>> print(separableconv2d)
>>> output shape : (8, 24, 24, 32)
```

SubPixel Convolutions

SubpixelConv1d

class tensorlayerx.nn.**SubpixelConv1d**(*scale=2*, *act=None*, *in_channels=None*, *name=None*)
It is a 1D sub-pixel up-sampling layer.

Calls a TensorFlow function that directly implements this functionality. We assume input has dim (batch, width, r)

Parameters

- **scale** (*int*) – The up-scaling ratio, a wrong setting will lead to Dimension size error.
- **act** (*activation function*) – The activation function of this layer.
- **in_channels** (*int*) – The number of in channels.
- **name** (*str*) – A unique layer name.

Examples

With TensorLayer

```
>>> net = tlx.nn.Input([8, 25, 32], name='input')
>>> subpixelconv1d = tlx.nn.SubpixelConv1d(scale=2, name='subpixelconv1d')(net)
>>> print(subpixelconv1d)
>>> output shape : (8, 50, 16)
```

References

Audio Super Resolution Implementation.

SubpixelConv2d

class tensorlayerx.nn.**SubpixelConv2d**(*scale=2*, *data_format='channels_last'*, *act=None*, *name=None*)
It is a 2D sub-pixel up-sampling layer, usually be used for Super-Resolution applications, see SRGAN for example.

Parameters

- **scale** (*int*) – factor to increase spatial resolution.

- **data_format** (*str*) – "channels_last" (NHWC, default) or "channels_first" (NCHW).
- **act** (*activation function*) – The activation function of this layer.
- **name** (*str*) – A unique layer name.

Examples

With TensorLayer

```
>>> net = tlx.nn.Input([2, 16, 16, 4], name='input1')
>>> subpixelconv2d = tlx.nn.SubpixelConv2d(scale=2, data_format='channels_last',␣
↪name='subpixel_conv2d1')(net)
>>> print(subpixelconv2d)
>>> output shape : (2, 32, 32, 1)
```

```
>>> net = tlx.nn.Input([2, 16, 16, 40], name='input2')
>>> subpixelconv2d = tlx.nn.SubpixelConv2d(scale=2, data_format='channels_last',␣
↪name='subpixel_conv2d2')(net)
>>> print(subpixelconv2d)
>>> output shape : (2, 32, 32, 10)
```

```
>>> net = tlx.nn.Input([2, 16, 16, 250], name='input3')
>>> subpixelconv2d = tlx.nn.SubpixelConv2d(scale=5, data_format='channels_last',␣
↪name='subpixel_conv2d3')(net)
>>> print(subpixelconv2d)
>>> output shape : (2, 80, 80, 10)
```

References

- Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network

MaskedConv3d

class tensorlayerx.nn.**MaskedConv3d**(*mask_type*, *out_channels*, *filter_size=(3, 3, 3)*, *stride=(1, 1, 1)*, *dilation=(1, 1, 1)*, *padding='SAME'*, *act=None*, *in_channels=None*, *data_format='channels_last'*, *kernel_initializer='he_normal'*, *bias_initializer='zeros'*, *name=None*)

MaskedConv3D. Reference: [1] Nguyen D T , Quach M , Valenzise G , et al. Lossless Coding of Point Cloud Geometry using a Deep Generative Model[J]. IEEE Transactions on Circuits and Systems for Video Technology, 2021, PP(99):1-1.

Parameters

- **mask_type** (*str*) – The mask type('A', 'B')
- **out_channels** (*int*) – The number of filters.
- **filter_size** (*tuple or int*) – The filter size (height, width).
- **stride** (*tuple or int*) – The sliding window stride of corresponding input dimensions. It must be in the same order as the `shape` parameter.
- **dilation** (*tuple or int*) – Specifying the dilation rate to use for dilated convolution.

- **act** (*activation function*) – The activation function of this layer.
- **padding** (*str*) – The padding algorithm type: "SAME" or "VALID".
- **data_format** (*str*) – "channels_last" (NDHWC, default) or "channels_first" (NCDHW).
- **kernel_initializer** (*initializer or str*) – The initializer for the the weight matrix.
- **bias_initializer** (*initializer or None or str*) – The initializer for the the bias vector. If None, skip biases.
- **in_channels** (*int*) – The number of in channels.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayer

```
>>> net = tlx.nn.Input([8, 20, 20, 20, 3], name='input')
>>> conv3d = tlx.nn.MaskedConv3d(mask_type='A', out_channels=32, filter_size=(3,␣
↪3, 3), stride=(2, 2, 2), bias_initializer=None, in_channels=3, name='conv3d_1')
>>> print(conv3d)
>>> tensor = tlx.nn.MaskedConv3d(mask_type='B', out_channels=32, filter_size=(3,␣
↪3, 3), stride=(2, 2, 2), act=tlx.ReLU, name='conv3d_2')(net)
>>> print(tensor)
```

2.6.5 Linear Layers

Linear Layer

class tensorlayerx.nn.**Linear**(*out_features*, *act=None*, *W_init='truncated_normal'*, *b_init='constant'*, *in_features=None*, *name=None*)

Applies a linear transformation to the incoming data: $y = xA^T + b$

Parameters

- **out_features** (*int*) – The number of units of this layer.
- **act** (*activation function*) – The activation function of this layer.
- **W_init** (*initializer or str*) – The initializer for the weight matrix.
- **b_init** (*initializer or None or str*) – The initializer for the bias vector. If None, skip biases.
- **in_features** (*int*) – The number of channels of the previous layer. If None, it will be automatically detected when the layer is forwarded for the first time.
- **name** (*None or str*) – A unique layer name. If None, a unique name will be automatically generated.

Examples

With TensorLayerx

```
>>> net = tlx.nn.Input([100, 50], name='input')
>>> linear = tlx.nn.Linear(out_features=800, act=tlx.ReLU, in_features=50, name=
↪'linear_1')
>>> tensor = tlx.nn.Linear(out_features=800, act=tlx.ReLU, name='linear_2')(net)
```

Notes

If the layer input has more than two axes, it needs to be flatten by using *Flatten*.

Drop Connect Linear Layer

class tensorlayerx.nn.**DropconnectLinear**(*keep=0.5*, *out_features=100*, *act=None*, *W_init='truncated_normal'*, *b_init='constant'*, *in_features=None*, *name=None*)

The *DropconnectLinear* class is `Dense` with DropConnect behaviour which randomly removes connections between this layer and the previous layer according to a keeping probability.

Parameters

- **keep** (*float*) – The keeping probability. The lower the probability it is, the more activations are set to zero.
- **out_features** (*int*) – The number of units of this layer.
- **act** (*activation function*) – The activation function of this layer.
- **W_init** (*weights initializer or str*) – The initializer for the weight matrix.
- **b_init** (*biases initializer or str*) – The initializer for the bias vector.
- **in_features** (*int*) – The number of channels of the previous layer. If None, it will be automatically detected when the layer is forwarded for the first time.
- **name** (*str*) – A unique layer name.

Examples

```
>>> net = tlx.nn.Input([10, 784], name='input')
>>> net = tlx.nn.DropconnectLinear(keep=0.8, out_features=800, act=tlx.ReLU, name=
↪'DropconnectLinear1')(net)
>>> output shape :(10, 800)
>>> net = tlx.nn.DropconnectLinear(keep=0.5, out_features=800, act=tlx.ReLU, name=
↪'DropconnectLinear2')(net)
>>> output shape :(10, 800)
>>> net = tlx.nn.DropconnectLinear(keep=0.5, out_features=10, name=
↪'DropconnectLinear3')(net)
>>> output shape :(10, 10)
```

References

- Wan, L. (2013). Regularization of neural networks using dropconnect

2.6.6 Dropout Layers

class tensorlayerx.nn.**Dropout**(*p=0.5*, *seed=0*, *name=None*)

During training, randomly zeroes some of the elements of the input tensor with probability p using samples from a Bernoulli distribution. Each channel will be zeroed out independently on every forward call.

Parameters

- **p** (*float*) – probability of an element to be zeroed. Default: 0.5
- **seed** (*int or None*) – The seed for random dropout.
- **name** (*None or str*) – A unique layer name.

Examples

```
>>> net = tlx.nn.Input([10, 200])
>>> net = tlx.nn.Dropout(p=0.2)(net)
```

2.6.7 Extend Layers

Expand Dims Layer

class tensorlayerx.nn.**ExpandDims**(*axis=-1*, *name=None*)

The *ExpandDims* class inserts a dimension of 1 into a tensor's shape, see tf.expand_dims() .

Parameters

- **axis** (*int*) – The dimension index at which to expand the shape of input.
- **name** (*str*) – A unique layer name. If None, a unique name will be automatically assigned.

Examples

```
>>> x = tlx.nn.Input([10, 3], name='in')
>>> y = tlx.nn.ExpandDims(axis=-1)(x)
[10, 3, 1]
```

Tile layer

class tensorlayerx.nn.**Tile**(*multiples=None*, *name=None*)

The *Tile* class constructs a tensor by tiling a given tensor, see tf.tile() .

Parameters

- **multiples** (*tensor*) – Must be one of the following types: int32, int64. 1-D Length must be the same as the number of dimensions in input.
- **name** (*None or str*) – A unique layer name.

Examples

```
>>> x = tlx.nn.Input([10, 3], name='in')
>>> y = tlx.nn.Tile(multiples=[2, 3])(x)
```

2.6.8 Image Resampling Layers

2D UpSampling

class tensorlayerx.nn.**UpSampling2d**(*scale*, *method='bilinear'*, *antialias=False*, *data_format='channels_last'*, *name=None*, *ksize=None*)

The *UpSampling2d* class is a up-sampling 2D layer.

See tf.image.resize_images.

Parameters

- **scale** (*int or tuple of int*) – (scale_height, scale_width) scale factor. scale_height = new_height/height, scale_width = new_width/width.
- **method** (*str*) –

  The resize method selected through the given string. Default 'bilinear'.

  - 'bilinear', Bilinear interpolation.
  - 'nearest', Nearest neighbor interpolation.
  - 'bicubic', Bicubic interpolation.
  - 'area', Area interpolation.
- **antialias** (*boolean*) – Whether to use an anti-aliasing filter when downsampling an image.
- **data_format** (*str*) – channels_last 'channel_last' (default) or channels_first.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayer

```
>>> ni = tlx.nn.Input([10, 50, 50, 32], name='input')
>>> ni = tlx.nn.UpSampling2d(scale=(2, 2))(ni)
>>> output shape : [10, 100, 100, 32]
```

2D DownSampling

class tensorlayerx.nn.**DownSampling2d**(*scale*, *method='bilinear'*, *antialias=False*, *data_format='channels_last'*, *name=None*, *ksize=None*)

The *DownSampling2d* class is down-sampling 2D layer.

See tf.image.resize_images.

Parameters

- **scale** (*int or tuple of int*) – (new_height, new_width) scale factor.scale_height = new_height/height, scale_width = new_width/width.
- **method** (*str*) –

  The resize method selected through the given string. Default 'bilinear'.

  - 'bilinear', Bilinear interpolation.

- 'nearest', Nearest neighbor interpolation.
- 'bicubic', Bicubic interpolation.
- 'area', Area interpolation.

- **antialias** (*boolean*) – Whether to use an anti-aliasing filter when downsampling an image.
- **data_format** (*str*) – channels_last 'channel_last' (default) or channels_first.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayer

```
>>> ni = tlx.nn.Input([10, 50, 50, 32], name='input')
>>> ni = tlx.nn.DownSampling2d(scale=(2, 2))(ni)
>>> output shape : [10, 25, 25, 32]
```

2.6.9 Merge Layers

Concat Layer

class tensorlayerx.nn.**Concat**(*concat_dim=-1*, *name=None*)

A layer that concats multiple tensors according to given axis.

Parameters

- **concat_dim** (*int*) – The dimension to concatenate.
- **name** (*None or str*) – A unique layer name.

Examples

```
>>> class CustomModel(Module):
>>>     def __init__(self):
>>>         super(CustomModel, self).__init__(name="custom")
>>>         self.linear1 = tlx.nn.Linear(in_features=20, out_features=10, act=tlx.
 ↪ReLU, name='relu1_1')
>>>         self.linear2 = tlx.nn.Linear(in_features=20, out_features=10, act=tlx.
 ↪ReLU, name='relu2_1')
>>>         self.concat = tlx.nn.Concat(concat_dim=1, name='concat_layer')
```

```
>>>     def forward(self, inputs):
>>>         d1 = self.linear1(inputs)
>>>         d2 = self.linear2(inputs)
>>>         outputs = self.concat([d1, d2])
>>>         return outputs
```

ElementWise Layer

class tensorlayerx.nn.**Elementwise**(*combine_fn=<function minimum>*, *act=None*, *name=None*)

A layer that combines multiple `Layer` that have the same output shapes according to an element-wise operation. If the element-wise operation is complicated, please consider to use `ElementwiseLambda`.

Parameters

- **combine_fn** (*a TensorFlow element-wise combine function*) – e.g. AND is `tlx.minimum` ; OR is `tlx.maximum` ; ADD is `tlx.add` ; MUL is `tlx.multiply` and so on. See TensorFlow Math API . If the combine function is more complicated, please consider to use `ElementwiseLambda`.
- **act** (*activation function*) – The activation function of this layer.
- **name** (*None or str*) – A unique layer name.

Examples

```
>>> import tensorlayerx as tlx
>>> class CustomModel(tlx.nn.Module):
>>>     def __init__(self):
>>>         super(CustomModel, self).__init__(name="custom")
>>>         self.linear1 = tlx.nn.Linear(in_features=20, out_features=10, act=tlx.
↪ReLU, name='relu1_1')
>>>         self.linear2 = tlx.nn.Linear(in_features=20, out_features=10, act=tlx.
↪ReLU, name='relu2_1')
>>>         self.element = tlx.nn.Elementwise(combine_fn=tlx.minimum, name=
↪'minimum')
```

```
>>>     def forward(self, inputs):
>>>         d1 = self.linear1(inputs)
>>>         d2 = self.linear2(inputs)
>>>         outputs = self.element([d1, d2])
>>>         return outputs
```

2.6.10 Noise Layer

class tensorlayerx.nn.**GaussianNoise**(*mean=0.0*, *stddev=1.0*, *is_always=True*, *seed=None*, *name=None*)

The *GaussianNoise* class is noise layer that adding noise with gaussian distribution to the activation.

Parameters

- **mean** (*float*) – The mean. Default is 0.0.
- **stddev** (*float*) – The standard deviation. Default is 1.0.
- **is_always** (*boolean*) – Is True, add noise for train and eval mode. If False, skip this layer in eval mode.
- **seed** (*int or None*) – The seed for random noise.
- **name** (*str*) – A unique layer name.

Examples

With TensorLayer

```
>>> net = tlx.nn.Input([64, 200], name='input')
>>> net = tlx.nn.Linear(in_features=200, out_features=100, act=tlx.ReLU, name=
↪'linear')(net)
>>> gaussianlayer = tlx.nn.GaussianNoise(name='gaussian')(net)
>>> print(gaussianlayer)
>>> output shape : (64, 100)
```

2.6.11 Normalization Layers

Batch Normalization

class tensorlayerx.nn.**BatchNorm**(*momentum=0.9*, *epsilon=1e-05*, *act=None*, *is_train=True*, *beta_init='zeros'*, *gamma_init='random_normal'*, *moving_mean_init='zeros'*, *moving_var_init='zeros'*, *num_features=None*, *data_format='channels_last'*, *name=None*)

This interface is used to construct a callable object of the BatchNorm class. For more details, refer to code examples. It implements the function of the Batch Normalization Layer and can be used as a normalizer function for conv2d and fully connected operations. The data is normalized by the mean and variance of the channel based on the current batch data.

the μ_β and σ_β^2 are the statistics of one mini-batch. Calculated as follows:

$$\mu_\beta \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \quad // \; mini-batch\; mean$$

$$\sigma_\beta^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\beta)^2 \quad // \; mini-batch\; variance$$

- x : mini-batch data
- m : the size of the mini-batch data

the μ_β and σ_β^2 are not the statistics of one mini-batch. They are global or running statistics (moving_mean and moving_variance). It usually got from the pre-trained model. Calculated as follows:

$$moving\_mean = moving\_mean * momentum + \mu_\beta * (1. - momentum) \quad //globalmean$$

$$moving\_variance = moving\_variance * momentum + \sigma_\beta^2 * (1. - momentum) \quad //globalvariance$$

The normalization function formula is as follows:

$$\hat{x_i} \leftarrow \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \epsilon}} \quad // \; normalize$$

$$y_i \leftarrow \gamma\hat{x_i} + \beta \quad // \; scale\; and\; shift$$

- ϵ : add a smaller value to the variance to prevent division by zero

- γ : trainable proportional parameter
- β : trainable deviation parameter

Parameters

- **momentum** (*float*) – The value used for the moving_mean and moving_var computation. Default: 0.9.
- **epsilon** (*float*) – a value added to the denominator for numerical stability. Default: 1e-5
- **act** (*activation function*) – The activation function of this layer.
- **is_train** (*boolean*) – Is being used for training or inference.
- **beta_init** (*initializer or str*) – The initializer for initializing beta, if None, skip beta. Usually you should not skip beta unless you know what happened.
- **gamma_init** (*initializer or str*) – The initializer for initializing gamma, if None, skip gamma. When the batch normalization layer is use instead of 'biases', or the next layer is linear, this can be disabled since the scaling can be done by the next layer. see Inception-ResNet-v2
- **moving_mean_init** (*initializer or str*) – The initializer for initializing moving mean, if None, skip moving mean.
- **moving_var_init** (*initializer or str*) – The initializer for initializing moving var, if None, skip moving var.
- **num_features** (*int*) – Number of features for input tensor. Useful to build layer if using BatchNorm1d, BatchNorm2d or BatchNorm3d, but should be left as None if using BatchNorm. Default None.
- **data_format** (*str*) – channels_last 'channel_last' (default) or channels_first.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10, 50, 50, 32], name='input')
>>> net = tlx.nn.BatchNorm()(net)
```

Notes

The *BatchNorm* is universally suitable for 3D/4D/5D input in static model, but should not be used in dynamic model where layer is built upon class initialization. So the argument 'num_features' should only be used for subclasses *BatchNorm1d*, *BatchNorm2d* and *BatchNorm3d*. All the three subclasses are suitable under all kinds of conditions.

References

- Source
- stackoverflow

Batch Normalization 1D

class tensorlayerx.nn.**BatchNorm1d**(*momentum=0.9*, *epsilon=1e-05*, *act=None*, *is_train=True*, *beta_init='zeros'*, *gamma_init='random_normal'*, *moving_mean_init='zeros'*, *moving_var_init='zeros'*, *num_features=None*, *data_format='channels_last'*, *name=None*)

The *BatchNorm1d* applies Batch Normalization over 2D/3D input (a mini-batch of 1D inputs (optional) with additional channel dimension), of shape (N, C) or (N, L, C) or (N, C, L). See more details in *BatchNorm*.

Examples

With TensorLayerX

```
>>> # in static model, no need to specify num_features
>>> net = tlx.nn.Input([10, 50, 32], name='input')
>>> net = tlx.nn.BatchNorm1d()(net)
>>> # in dynamic model, build by specifying num_features
>>> conv = tlx.nn.Conv1d(32, 5, 1, in_channels=3)
>>> bn = tlx.nn.BatchNorm1d(num_features=32)
```

Batch Normalization 2D

class tensorlayerx.nn.**BatchNorm2d**(*momentum=0.9*, *epsilon=1e-05*, *act=None*, *is_train=True*, *beta_init='zeros'*, *gamma_init='random_normal'*, *moving_mean_init='zeros'*, *moving_var_init='zeros'*, *num_features=None*, *data_format='channels_last'*, *name=None*)

The *BatchNorm2d* applies Batch Normalization over 4D input (a mini-batch of 2D inputs with additional channel dimension) of shape (N, H, W, C) or (N, C, H, W). See more details in *BatchNorm*.

Examples

With TensorLayer

```
>>> # in static model, no need to specify num_features
>>> net = tlx.nn.Input([10, 50, 50, 32], name='input')
>>> net = tlx.nn.BatchNorm2d()(net)
>>> # in dynamic model, build by specifying num_features
>>> conv = tlx.nn.Conv2d(32, (5, 5), (1, 1), in_channels=3)
>>> bn = tlx.nn.BatchNorm2d(num_features=32)
```

Batch Normalization 3D

class tensorlayerx.nn.**BatchNorm3d**(*momentum=0.9*, *epsilon=1e-05*, *act=None*, *is_train=True*, *beta_init='zeros'*, *gamma_init='random_normal'*, *moving_mean_init='zeros'*, *moving_var_init='zeros'*, *num_features=None*, *data_format='channels_last'*, *name=None*)

The *BatchNorm3d* applies Batch Normalization over 5D input (a mini-batch of 3D inputs with additional channel dimension) with shape (N, D, H, W, C) or (N, C, D, H, W). See more details in *BatchNorm*.

Examples

With TensorLayer

```
>>> # in static model, no need to specify num_features
>>> net = tlx.nn.Input([10, 50, 50, 50, 32], name='input')
>>> net = tlx.nn.BatchNorm3d()(net)
>>> # in dynamic model, build by specifying num_features
>>> conv = tlx.nn.Conv3d(32, (5, 5, 5), (1, 1), in_channels=3)
>>> bn = tlx.nn.BatchNorm3d(num_features=32)
```

2.6.12 Padding Layers

Pad Layer (Expert API)

Padding layer for any modes.

class tensorlayerx.nn.**PadLayer**(*padding=None*, *mode='CONSTANT'*, *constant_values=0*, *name=None*)

The *PadLayer* class is a padding layer for any mode and dimension. Please see tf.pad for usage.

Parameters

- **padding** (*list of lists of 2 ints, or a Tensor of type int32.*) – The int32 values to pad.
- **mode** (*str*) – "CONSTANT", "REFLECT", or "SYMMETRIC" (case-insensitive).
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayer

```
>>> net = tlx.nn.Input([10, 224, 224, 3], name='input')
>>> padlayer = tlx.nn.PadLayer([[0, 0], [3, 3], [3, 3], [0, 0]], "REFLECT", name=
↪'inpad')(net)
>>> print(padlayer)
>>> output shape : (10, 230, 230, 3)
```

1D Zero padding

class tensorlayerx.nn.**ZeroPad1d**(*padding*, *name=None*, *data_format='channels_last'*)

The *ZeroPad1d* class is a 1D padding layer for signal [batch, length, channel].

Parameters

- **padding** (*tuple of 2 ints*) –
  - If tuple of 2 ints, zeros to add at the beginning and at the end of the padding dimension.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayer

```
>>> net = tlx.nn.Input([10, 100, 1], name='input')
>>> pad1d = tlx.nn.ZeroPad1d(padding=(3, 3))(net)
>>> print(pad1d)
>>> output shape : (10, 106, 1)
```

2D Zero padding

class tensorlayerx.nn.**ZeroPad2d**(*padding*, *name=None*, *data_format='channels_last'*)

The *ZeroPad2d* class is a 2D padding layer for image [batch, height, width, channel].

Parameters

- **padding** (*tuple of 2 tuples of 2 ints.*) –
  - If tuple of 2 tuples of 2 ints, interpreted as `((top_pad, bottom_pad), (left_pad, right_pad))`.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayer

```
>>> net = tlx.nn.Input([10, 100, 100, 3], name='input')
>>> pad2d = tlx.nn.ZeroPad2d(padding=((3, 3), (4, 4)))(net)
>>> print(pad2d)
>>> output shape : (10, 106, 108, 3)
```

3D Zero padding

class tensorlayerx.nn.**ZeroPad3d**(*padding*, *name=None*, *data_format='channels_last'*)

The *ZeroPad3d* class is a 3D padding layer for volume [batch, depth, height, width, channel].

Parameters

- **padding** (*tuple of 2 tuples of 2 ints.*) –
  - If tuple of 2 tuples of 2 ints, interpreted as `((left_dim1_pad, right_dim1_pad), (left_dim2_pad, right_dim2_pad), (left_dim3_pad, right_dim3_pad))`.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayer

```
>>> net = tlx.nn.Input([10, 100, 100, 100, 3], name='input')
>>> pad3d = tlx.nn.ZeroPad3d(padding=((3, 3), (4, 4), (5, 5)))(net)
>>> print(pad3d)
>>> output shape : (10, 106, 108, 110, 3)
```

2.6.13 Pooling Layers

1D Max pooling

class tensorlayerx.nn.**MaxPool1d**(*kernel_size=3*, *stride=2*, *padding='SAME'*, *data_format='channels_last'*, *name=None*)

Max pooling for 1D signal.

Parameters

- **kernel_size** (*int*) – Pooling window size.
- **stride** (*int*) – Stride of the pooling operation.
- **padding** (*str or int*) – The padding method: 'VALID' or 'SAME'.
- **data_format** (*str*) – One of channels_last (default, [batch, length, channel]) or channels_first. The ordering of the dimensions in the inputs.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10, 50, 32], name='input')
>>> net = tlx.nn.MaxPool1d(kernel_size=3, stride=2, padding='SAME', name=
↪'maxpool1d')(net)
>>> output shape : [10, 25, 32]
```

1D Avg pooling

class tensorlayerx.nn.**AvgPool1d**(*kernel_size=3*, *stride=2*, *padding='SAME'*, *data_format='channels_last'*, *dilation_rate=1*, *name=None*)

Avg pooling for 1D signal.

Parameters

- **kernel_size** (*int*) – Pooling window size.
- **stride** (*int*) – Strides of the pooling operation.
- **padding** (*inttuple or str*) – The padding method: 'VALID' or 'SAME'.
- **data_format** (*str*) – One of channels_last (default, [batch, length, channel]) or channels_first. The ordering of the dimensions in the inputs.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10, 50, 32], name='input')
>>> net = tlx.nn.AvgPool1d(kernel_size=3, stride=2, padding='SAME')(net)
>>> output shape : [10, 25, 32]
```

2D Max pooling

class tensorlayerx.nn.**MaxPool2d**(*kernel_size=(3, 3), stride=(2, 2), padding='SAME', data_format='channels_last', name=None*)

Max pooling for 2D image.

Parameters

- **kernel_size** (*tuple or int*) – (height, width) for filter size.
- **stride** (*tuple or int*) – (height, width) for stride.
- **padding** (*inttuple or str*) – The padding method: 'VALID' or 'SAME'.
- **data_format** (*str*) – One of channels_last (default, [batch, height, width, channel]) or channels_first. The ordering of the dimensions in the inputs.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10, 50, 50, 32], name='input')
>>> net = tlx.nn.MaxPool2d(kernel_size=(3, 3), stride=(2, 2), padding='SAME')(net)
>>> output shape : [10, 25, 25, 32]
```

2D Avg pooling

class tensorlayerx.nn.**AvgPool2d**(*kernel_size=(3, 3), stride=(2, 2), padding='SAME', data_format='channels_last', name=None*)

Avg pooling for 2D image [batch, height, width, channel].

Parameters

- **kernel_size** (*tuple or int*) – (height, width) for filter size.
- **stride** (*tuple or int*) – (height, width) for stride.
- **padding** (*inttuple or str*) – The padding method: 'VALID' or 'SAME'.
- **data_format** (*str*) – One of channels_last (default, [batch, height, width, channel]) or channels_first. The ordering of the dimensions in the inputs.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10, 50, 50, 32], name='input')
>>> net = tlx.nn.AvgPool2d(kernel_size=(3, 3), stride=(2, 2), padding='SAME')(net)
>>> output shape : [10, 25, 25, 32]
```

3D Max pooling

class tensorlayerx.nn.**MaxPool3d**(*kernel_size=(3, 3, 3), stride=(2, 2, 2), padding='VALID', data_format='channels_last', name=None*)

Max pooling for 3D volume.

Parameters

- **kernel_size** (*tuple or int*) – Pooling window size.
- **stride** (*tuple or int*) – Strides of the pooling operation.
- **padding** (*inttuple or str*) – The padding method: 'VALID' or 'SAME'.
- **data_format** (*str*) – One of channels_last (default, [batch, depth, height, width, channel]) or channels_first. The ordering of the dimensions in the inputs.
- **name** (*None or str*) – A unique layer name.

Returns A max pooling 3-D layer with a output rank as 5.

Return type tf.Tensor

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10, 50, 50, 50, 32], name='input')
>>> net = tlx.nn.MaxPool3d(kernel_size=(3, 3, 3), stride=(2, 2, 2), padding='SAME
↪')(net)
>>> output shape : [10, 25, 25, 25, 32]
```

3D Avg pooling

class tensorlayerx.nn.**AvgPool3d**(*kernel_size=(3, 3, 3), stride=(2, 2, 2), padding='VALID', data_format='channels_last', name=None*)

Avg pooling for 3D volume.

Parameters

- **kernel_size** (*tuple or int*) – Pooling window size.
- **stride** (*tuple or int*) – Strides of the pooling operation.
- **padding** (*inttuple or str*) – The padding method: 'VALID' or 'SAME'.
- **data_format** (*str*) – One of channels_last (default, [batch, depth, height, width, channel]) or channels_first. The ordering of the dimensions in the inputs.
- **name** (*None or str*) – A unique layer name.

Returns A Avg pooling 3-D layer with a output rank as 5.

Return type tf.Tensor

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10, 50, 50, 50, 32], name='input')
>>> net = tlx.nn.AvgPool3d(kernel_size=(3, 3, 3), stride=(2, 2, 2), padding='SAME
↪')(net)
>>> output shape : [10, 25, 25, 25, 32]
```

1D Global Max pooling

class tensorlayerx.nn.**GlobalMaxPool1d**(*data_format='channels_last'*, *name=None*)

The *GlobalMaxPool1d* class is a 1D Global Max Pooling layer.

Parameters

- **data_format** (*str*) – One of channels_last (default, [batch, length, channel]) or channels_first. The ordering of the dimensions in the inputs.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10, 100, 30], name='input')
>>> net = tlx.nn.GlobalMaxPool1d()(net)
>>> output shape : [10, 30]
```

1D Global Avg pooling

class tensorlayerx.nn.**GlobalAvgPool1d**(*data_format='channels_last'*, *name=None*)

The *GlobalAvgPool1d* class is a 1D Global Avg Pooling layer.

Parameters

- **data_format** (*str*) – One of channels_last (default, [batch, length, channel]) or channels_first. The ordering of the dimensions in the inputs.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10, 100, 30], name='input')
>>> net = tlx.nn.GlobalAvgPool1d()(net)
>>> output shape : [10, 30]
```

2D Global Max pooling

class tensorlayerx.nn.**GlobalMaxPool2d**(*data_format='channels_last'*, *name=None*)

The *GlobalMaxPool2d* class is a 2D Global Max Pooling layer.

Parameters

- **data_format** (*str*) – One of channels_last (default, [batch, height, width, channel]) or channels_first. The ordering of the dimensions in the inputs.

- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10, 100, 100, 30], name='input')
>>> net = tlx.nn.GlobalMaxPool2d()(net)
>>> output shape : [10, 30]
```

2D Global Avg pooling

class tensorlayerx.nn.**GlobalAvgPool2d**(*data_format='channels_last'*, *name=None*)

The *GlobalAvgPool2d* class is a 2D Global Avg Pooling layer.

Parameters

- **data_format** (*str*) – One of channels_last (default, [batch, height, width, channel]) or channels_first. The ordering of the dimensions in the inputs.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10, 100, 100, 30], name='input')
>>> net = tlx.nn.GlobalAvgPool2d()(net)
>>> output shape : [10, 30]
```

3D Global Max pooling

class tensorlayerx.nn.**GlobalMaxPool3d**(*data_format='channels_last'*, *name=None*)

The *GlobalMaxPool3d* class is a 3D Global Max Pooling layer.

Parameters

- **data_format** (*str*) – One of channels_last (default, [batch, depth, height, width, channel]) or channels_first. The ordering of the dimensions in the inputs.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10, 100, 100, 100, 30], name='input')
>>> net = tlx.nn.GlobalMaxPool3d()(net)
>>> output shape : [10, 30]
```

3D Global Avg pooling

class tensorlayerx.nn.**GlobalAvgPool3d**(*data_format='channels_last'*, *name=None*)

The *GlobalAvgPool3d* class is a 3D Global Avg Pooling layer.

Parameters

- **data_format** (*str*) – One of channels_last (default, [batch, depth, height, width, channel]) or channels_first. The ordering of the dimensions in the inputs.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10, 100, 100, 100, 30], name='input')
>>> net = tlx.nn.GlobalAvgPool3d()(net)
>>> output shape : [10, 30]
```

1D Adaptive Max pooling

class tensorlayerx.nn.**AdaptiveMaxPool1d**(*output_size*, *data_format='channels_last'*, *name=None*)

The *AdaptiveMaxPool1d* class is a 1D Adaptive Max Pooling layer.

Parameters

- **output_size** (*int*) – The target output size. It must be an integer.
- **data_format** (*str*) – One of channels_last (default, [batch, width, channel]) or channels_first. The ordering of the dimensions in the inputs.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10, 32, 3], name='input')
>>> net = tlx.nn.AdaptiveMaxPool1d(output_size=16)(net)
>>> output shape : [10, 16, 3]
```

1D Adaptive Avg pooling

class tensorlayerx.nn.**AdaptiveAvgPool1d**(*output_size*, *data_format='channels_last'*, *name=None*)

The *AdaptiveAvgPool1d* class is a 1D Adaptive Avg Pooling layer.

Parameters

- **output_size** (*int*) – The target output size. It must be an integer.
- **data_format** (*str*) – One of channels_last (default, [batch, width, channel]) or channels_first. The ordering of the dimensions in the inputs.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10, 32, 3], name='input')
>>> net = tlx.nn.AdaptiveAvgPool1d(output_size=16)(net)
>>> output shape : [10, 16, 3]
```

2D Adaptive Max pooling

class tensorlayerx.nn.**AdaptiveMaxPool2d**(*output_size*, *data_format='channels_last'*, *name=None*)

The *AdaptiveMaxPool2d* class is a 2D Adaptive Max Pooling layer.

Parameters

- **output_size** (*int or list or tuple*) – The target output size. It cloud be an int [int,int](int, int).
- **data_format** (*str*) – One of channels_last (default, [batch, height, width, channel]) or channels_first. The ordering of the dimensions in the inputs.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10, 32, 32, 3], name='input')
>>> net = tlx.nn.AdaptiveMaxPool2d(output_size=16)(net)
>>> output shape : [10, 16, 16, 3]
```

2D Adaptive Avg pooling

class tensorlayerx.nn.**AdaptiveAvgPool2d**(*output_size*, *data_format='channels_last'*, *name=None*)

The *AdaptiveAvgPool2d* class is a 2D Adaptive Avg Pooling layer.

Parameters

- **output_size** (*int or list or tuple*) – The target output size. It cloud be an int [int,int](int, int).
- **data_format** (*str*) – One of channels_last (default, [batch, height, width, channel]) or channels_first. The ordering of the dimensions in the inputs.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10,32, 32, 3], name='input')
>>> net = tlx.nn.AdaptiveAvgPool2d(output_size=16)(net)
>>> output shape : [10,16, 16, 3]
```

3D Adaptive Max pooling

class tensorlayerx.nn.**AdaptiveMaxPool3d**(*output_size*, *data_format='channels_last'*, *name=None*)

The *AdaptiveMaxPool3d* class is a 3D Adaptive Max Pooling layer.

Parameters

- **output_size** (*int or list or tuple*) – The target output size. It cloud be an int [int,int,int](int, int, int).
- **data_format** (*str*) – One of channels_last (default, [batch, depth, height, width, channel]) or channels_first. The ordering of the dimensions in the inputs.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10,32, 32, 32, 3], name='input')
>>> net = tlx.nn.AdaptiveMaxPool3d(output_size=16)(net)
>>> output shape : [10, 16, 16, 16, 3]
```

3D Adaptive Avg pooling

class tensorlayerx.nn.**AdaptiveAvgPool3d**(*output_size*, *data_format='channels_last'*, *name=None*)

The *AdaptiveAvgPool3d* class is a 3D Adaptive Avg Pooling layer.

Parameters

- **output_size** (*int or list or tuple*) – The target output size. It cloud be an int [int,int,int](int, int, int).
- **data_format** (*str*) – One of channels_last (default, [batch, depth, height, width, channel]) or channels_first. The ordering of the dimensions in the inputs.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10,32, 32, 32, 3], name='input')
>>> net = tlx.nn.AdaptiveAvgPool3d(output_size=16)(net)
>>> output shape : [10, 16, 16, 16, 3]
```

2D Corner pooling

class tensorlayerx.nn.**CornerPool2d**(*mode='TopLeft'*, *name=None*)

Corner pooling for 2D image [batch, height, width, channel], see here.

Parameters

- **mode** (*str*) – TopLeft for the top left corner, Bottomright for the bottom right corner.

- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> net = tlx.nn.Input([10, 32, 32, 8], name='input')
>>> net = tlx.nn.CornerPool2d(mode='TopLeft',name='cornerpool2d')(net)
>>> output shape : [10, 32, 32, 8]
```

2.6.14 Quantized Nets

This is an experimental API package for building Quantized Neural Networks. We are using matrix multiplication rather than add-minus and bit-count operation at the moment. Therefore, these APIs would not speed up the inferencing, for production, you can train model via TensorLayer and deploy the model into other customized C/C++ implementation (We probably provide users an extra C/C++ binary net framework that can load model from TensorLayer).

Note that, these experimental APIs can be changed in the future.

Scale

class tensorlayerx.nn.**Scale**(*init_scale=0.05*, *name='scale'*)

The *Scale* class is to multiple a trainable scale value to the layer outputs. Usually be used on the output of binary net.

Parameters

- **init_scale** (*float*) – The initial value for the scale factor.
- **name** (*a str*) – A unique layer name.

Examples

```
>>> inputs = tlx.nn.Input([8, 3])
>>> linear = tlx.nn.Linear(out_features=10, in_channels=3)(inputs)
>>> outputs = tlx.nn.Scale(init_scale=0.5)(linear)
```

Binary Linear Layer

class tensorlayerx.nn.**BinaryLinear**(*out_features=100*, *act=None*, *use_gemm=False*, *W_init='truncated_normal'*, *b_init='constant'*, *in_features=None*, *name=None*)

The *BinaryLinear* class is a binary fully connected layer, which weights are either -1 or 1 while inferencing.

Note that, the bias vector would not be binarized.

Parameters

- **out_features** (*int*) – The number of units of this layer.
- **act** (*activation function*) – The activation function of this layer, usually set to `tf.act.sign` or apply `Sign` after *BatchNorm*.

- **use_gemm** (*boolean*) – If True, use gemm instead of `tf.matmul` for inference. (TODO).
- **W_init** (*initializer or str*) – The initializer for the weight matrix.
- **b_init** (*initializer or None or str*) – The initializer for the bias vector. If None, skip biases.
- **in_features** (*int*) – The number of channels of the previous layer. If None, it will be automatically detected when the layer is forwarded for the first time.
- **name** (*None or str*) – A unique layer name.

Examples

```
>>> net = tlx.nn.Input([10, 784], name='input')
>>> net = tlx.nn.BinaryLinear(out_features=800, act=tlx.ReLU, name='BinaryLinear1
↪')(net)
>>> output shape :(10, 800)
>>> net = tlx.nn.BinaryLinear(out_features=10, name='BinaryLineart')(net)
>>> output shape : (10, 10)
```

Binary (De)Convolutions

BinaryConv2d

class tensorlayerx.nn.**BinaryConv2d**(*out_channels=32*, *kernel_size=(3, 3)*, *stride=(1, 1)*, *act=None*, *padding='VALID'*, *data_format='channels_last'*, *dilation=(1, 1)*, *W_init='truncated_normal'*, *b_init='constant'*, *in_channels=None*, *name=None*)

The *BinaryConv2d* class is a 2D binary CNN layer, which weights are either -1 or 1 while inference.

Note that, the bias vector would not be binarized.

Parameters

- **out_channels** (*int*) – The number of filters.
- **kernel_size** (*tuple or int*) – The filter size (height, width).
- **stride** (*tuple or int*) – The sliding window strides of corresponding input dimensions. It must be in the same order as the `shape` parameter.
- **act** (*activation function*) – The activation function of this layer.
- **padding** (*str*) – The padding algorithm type: "SAME" or "VALID".
- **data_format** (*str*) – "channels_last" (NHWC, default) or "channels_first" (NCHW).
- **dilation** (*tuple or int*) – Specifying the dilation rate to use for dilated convolution.
- **W_init** (*initializer or str*) – The initializer for the the weight matrix.
- **b_init** (*initializer or None or str*) – The initializer for the the bias vector. If None, skip biases.
- **in_channels** (*int*) – The number of in channels.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayer

```
>>> net = tlx.nn.Input([8, 100, 100, 32], name='input')
>>> binaryconv2d = tlx.nn.BinaryConv2d(
    ... out_channels=64, kernel_size=(3, 3), stride=(2, 2), act=tlx.ReLU, in_
↪channels=32, name='binaryconv2d')(net)
>>> print(binaryconv2d)
>>> output shape : (8, 50, 50, 64)
```

Ternary Linear Layer

TernaryLinear

class tensorlayerx.nn.**TernaryLinear**(*out_features=100*, *act=None*, *use_gemm=False*, *W_init='truncated_normal'*, *b_init='constant'*, *in_features=None*, *name=None*)

The *TernaryLinear* class is a ternary fully connected layer, which weights are either -1 or 1 or 0 while inference. # TODO The TernaryDense only supports TensorFlow backend.

Note that, the bias vector would not be tenaried.

Parameters

- **out_features** (*int*) – The number of units of this layer.
- **act** (*activation function*) – The activation function of this layer, usually set to `tf.act.sign` or apply `SignLayer` after `BatchNormLayer`.
- **use_gemm** (*boolean*) – If True, use gemm instead of `tf.matmul` for inference. (TODO).
- **W_init** (*initializer or str*) – The initializer for the weight matrix.
- **b_init** (*initializer or None or str*) – The initializer for the bias vector. If None, skip biases.
- **in_features** (*int*) – The number of channels of the previous layer. If None, it will be automatically detected when the layer is forwarded for the first time.
- **name** (*None or str*) – A unique layer name.

Ternary Convolutions

TernaryConv2d

class tensorlayerx.nn.**TernaryConv2d**(*out_channels=32*, *kernel_size=(3, 3)*, *stride=(1, 1)*, *act=None*, *padding='SAME'*, *use_gemm=False*, *data_format='channels_last'*, *dilation=(1, 1)*, *W_init='truncated_normal'*, *b_init='constant'*, *in_channels=None*, *name=None*)

The *TernaryConv2d* class is a 2D ternary CNN layer, which weights are either -1 or 1 or 0 while inference.

Note that, the bias vector would not be tenarized.

Parameters

- **out_channels** (*int*) – The number of filters.
- **kernel_size** (*tuple or int*) – The filter size (height, width).
- **stride** (*tuple or int*) – The sliding window stride of corresponding input dimensions. It must be in the same order as the `shape` parameter.
- **act** (*activation function*) – The activation function of this layer.
- **padding** (*str*) – The padding algorithm type: "SAME" or "VALID".
- **use_gemm** (*boolean*) – If True, use gemm instead of `tf.matmul` for inference. TODO: support gemm
- **data_format** (*str*) – "channels_last" (NHWC, default) or "channels_first" (NCHW).
- **dilation_rate** (*tuple or int*) – Specifying the dilation rate to use for dilated convolution.
- **W_init** (*initializer or str*) – The initializer for the the weight matrix.
- **b_init** (*initializer or None or str*) – The initializer for the the bias vector. If None, skip biases.
- **in_channels** (*int*) – The number of in channels.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayer

```
>>> net = tlx.nn.Input([8, 12, 12, 32], name='input')
>>> ternaryconv2d = tlx.nn.TernaryConv2d(
...     out_channels=64, kernel_size=(5, 5), stride=(1, 1), act=tlx.ReLU, padding=
↪'SAME', name='ternaryconv2d'
... )(net)
>>> print(ternaryconv2d)
>>> output shape : (8, 12, 12, 64)
```

DorefaLinear

class tensorlayerx.nn.**DorefaLinear**(*bitW=1*, *bitA=3*, *out_features=100*, *act=None*, *use_gemm=False*, *W_init='truncated_normal'*, *b_init='constant'*, *in_features=None*, *name=None*)

The *DorefaLinear* class is a binary fully connected layer, which weights are 'bitW' bits and the output of the previous layer are 'bitA' bits while inferencing.

Note that, the bias vector would not be binarized.

Parameters

- **bitW** (*int*) – The bits of this layer's parameter
- **bitA** (*int*) – The bits of the output of previous layer
- **out_features** (*int*) – The number of units of this layer.
- **act** (*activation function*) – The activation function of this layer, usually set to `tf.act.sign` or apply `Sign` after *BatchNorm*.

- **use_gemm** (*boolean*) – If True, use gemm instead of `tf.matmul` for inferencing. (TODO).
- **W_init** (*initializer or str*) – The initializer for the weight matrix.
- **b_init** (*initializer or None or str*) – The initializer for the bias vector. If None, skip biases.
- **in_features** (*int*) – The number of channels of the previous layer. If None, it will be automatically detected when the layer is forwarded for the first time.
- **name** (*a str*) – A unique layer name.

Examples

```
>>> net = tlx.nn.Input([10, 784], name='input')
>>> net = tlx.nn.DorefaLinear(out_features=800, act=tlx.ReLU, name='DorefaLinear1
↪')(net)
>>> output shape :(10, 800)
>>> net = tlx.nn.DorefaLinear(out_features=10, name='DorefaLinear2')(net)
>>> output shape :(10, 10)
```

DoReFa Convolutions

DorefaConv2d

class tensorlayerx.nn.**DorefaConv2d**(*bitW=1*, *bitA=3*, *out_channels=32*, *kernel_size=(3, 3)*, *stride=(1, 1)*, *act=None*, *padding='SAME'*, *data_format='channels_last'*, *dilation=(1, 1)*, *W_init='truncated_normal'*, *b_init='constant'*, *in_channels=None*, *name=None*)

The *DorefaConv2d* class is a 2D quantized convolutional layer, which weights are 'bitW' bits and the output of the previous layer are 'bitA' bits while inferencing.

Note that, the bias vector would not be binarized.

Parameters

- **bitW** (*int*) – The bits of this layer's parameter
- **bitA** (*int*) – The bits of the output of previous layer
- **out_channels** (*int*) – The number of filters.
- **kernel_size** (*tuple or int*) – The filter size (height, width).
- **stride** (*tuple or int*) – The sliding window strides of corresponding input dimensions. It must be in the same order as the `shape` parameter.
- **act** (*activation function*) – The activation function of this layer.
- **padding** (*str*) – The padding algorithm type: "SAME" or "VALID".
- **data_format** (*str*) – "channels_last" (NHWC, default) or "channels_first" (NCHW).
- **dilation** (*tuple or int*) – Specifying the dilation rate to use for dilated convolution.
- **W_init** (*initializer or str*) – The initializer for the the weight matrix.
- **b_init** (*initializer or None or str*) – The initializer for the the bias vector. If None, skip biases.

- **in_channels** (*int*) – The number of in channels.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayer

```
>>> net = tlx.nn.Input([8, 12, 12, 32], name='input')
>>> dorefaconv2d = tlx.nn.DorefaConv2d(
...     out_channels=32, kernel_size=(5, 5), stride=(1, 1), act=tlx.ReLU, padding=
↪'SAME', name='dorefaconv2d'
... )(net)
>>> print(dorefaconv2d)
>>> output shape : (8, 12, 12, 32)
```

2.6.15 Recurrent Layers

Common Recurrent layer

RNNCell layer

class tensorlayerx.nn.**RNNCell**(*input_size*, *hidden_size*, *bias=True*, *act='tanh'*, *name=None*)

An Elman RNN cell with tanh or ReLU non-linearity.

Parameters

- **input_size** (*int*) – The number of expected features in the input *x*
- **hidden_size** (*int*) – The number of features in the hidden state *h*
- **bias** (*bool*) – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **act** (*activation function*) – The non-linearity to use. Can be either 'tanh' or 'relu'. Default: 'tanh'
- **name** (*None or str*) – A unique layer name

Returns

- **outputs** (*tensor*) – A tensor with shape *[batch_size, hidden_size]*.
- **states** (*tensor*) – A tensor with shape *[batch_size, hidden_size]*. Tensor containing the next hidden state for each element in the batch

forward(*inputs*, *states=None*)

Parameters

- **inputs** (*tensor*) – A tensor with shape *[batch_size, input_size]*.
- **states** (*tensor or None*) – A tensor with shape *[batch_size, hidden_size]*. When states is None, zero state is used. Defaults to None.

Examples

With TensorLayerx

```
>>> input = tlx.nn.Input([4, 16], name='input')
>>> prev_h = tlx.nn.Input([4,32])
>>> cell = tlx.nn.RNNCell(input_size=16, hidden_size=32, bias=True, act='tanh
↪', name='rnncell_1')
>>> y, h = cell(input, prev_h)
>>> print(y.shape)
```

LSTMCell layer

class tensorlayerx.nn.**LSTMCell**(*input_size*, *hidden_size*, *bias=True*, *name=None*)

A long short-term memory (LSTM) cell.

Parameters

- **input_size** (*int*) – The number of expected features in the input *x*
- **hidden_size** (*int*) – The number of features in the hidden state *h*
- **bias** (*bool*) – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **name** (*None or str*) – A unique layer name

Returns

- **outputs** (*tensor*) – A tensor with shape *[batch_size, hidden_size]*.
- **states** (*tensor*) – A tuple of two tensor *(h, c)*, each of shape *[batch_size, hidden_size]*. Tensors containing the next hidden state and next cell state for each element in the batch.

forward(*inputs*, *states=None*)

Parameters

- **inputs** (*tensor*) – A tensor with shape *[batch_size, input_size]*.
- **states** (*tuple or None*) – A tuple of two tensor *(h, c)*, each of shape *[batch_size, hidden_size]*. When states is None, zero state is used. Defaults: None.

Examples

With TensorLayerx

```
>>> input = tlx.nn.Input([4, 16], name='input')
>>> prev_h = tlx.nn.Input([4,32])
>>> prev_c = tlx.nn.Input([4,32])
>>> cell = tlx.nn.LSTMCell(input_size=16, hidden_size=32, bias=True, name=
↪'lstmcell_1')
>>> y, (h, c)= cell(input, (prev_h, prev_c))
>>> print(y.shape)
```

GRUCell layer

class tensorlayerx.nn.**GRUCell**(*input_size*, *hidden_size*, *bias=True*, *name=None*)

A gated recurrent unit (GRU) cell.

Parameters

- **input_size** (*int*) – The number of expected features in the input *x*
- **hidden_size** (*int*) – The number of features in the hidden state *h*
- **bias** (*bool*) – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **name** (*None or str*) – A unique layer name

Returns

- **outputs** (*tensor*) – A tensor with shape *[batch_size, hidden_size]*.
- **states** (*tensor*) – A tensor with shape *[batch_size, hidden_size]*. Tensor containing the next hidden state for each element in the batch

forward(*inputs*, *states=None*)

Parameters

- **inputs** (*tensor*) – A tensor with shape *[batch_size, input_size]*.
- **states** (*tensor or None*) – A tensor with shape *[batch_size, hidden_size]*. When states is None, zero state is used. Defaults: *None*.

Examples

With TensorLayerx

```
>>> input = tlx.nn.Input([4, 16], name='input')
>>> prev_h = tlx.nn.Input([4,32])
>>> cell = tlx.nn.GRUCell(input_size=16, hidden_size=32, bias=True, name=
↪'grucell_1')
>>> y, h= cell(input, prev_h)
>>> print(y.shape)
```

RNN layer

class tensorlayerx.nn.**RNN**(*input_size*, *hidden_size*, *num_layers=1*, *bias=True*, *batch_first=False*, *dropout=0.0*, *bidirectional=False*, *act='tanh'*, *name=None*)

Multilayer Elman network(RNN). It takes input sequences and initial states as inputs, and returns the output sequences and the final states.

Parameters

- **input_size** (*int*) – The number of expected features in the input *x*
- **hidden_size** (*int*) – The number of features in the hidden state *h*
- **num_layers** (*int*) – Number of recurrent layers. Default: 1
- **bias** (*bool*) – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`

- **batch_first** (*bool*) – If `True`, then the input and output tensors are provided as *[batch_size, seq, input_size]*, Default: `False`
- **dropout** (*float*) – If non-zero, introduces a *Dropout* layer on the outputs of each RNN layer except the last layer, with dropout probability equal to *dropout*. Default: 0
- **bidirectional** (*bool*) – If `True`, becomes a bidirectional RNN. Default: `False`
- **act** (*activation function*) – The non-linearity to use. Can be either 'tanh' or 'relu'. Default: 'tanh'
- **name** (*None or str*) – A unique layer name

Returns

- **outputs** (*tensor*) – the output sequence. if *batch_first* is True, the shape is *[batch_size, seq, num_directions * hidden_size]*, else, the shape is *[seq, batch_size, num_directions * hidden_size]*.
- **final_states** (*tensor*) – final states. The shape is *[num_layers * num_directions, batch_size, hidden_size]*. Note that if the RNN is Bidirectional, the forward states are (0,2,4,6,...) and the backward states are (1,3,5,7,....).

forward(*input*, *states=None*)

Parameters

- **inputs** (*tensor*) – the input sequence. if *batch_first* is True, the shape is *[batch_size, seq, input_size]*, else, the shape is *[seq, batch_size, input_size]*.
- **initial_states** (*tensor or None*) – the initial states. The shape is *[num_layers * num_directions, batch_size, hidden_size]*.If initial_state is not given, zero initial states are used. If the RNN is Bidirectional, num_directions should be 2, else it should be 1. Default: None.

Examples

With TensorLayer

```
>>> input = tlx.nn.Input([23, 32, 16], name='input')
>>> prev_h = tlx.nn.Input([4, 32, 32])
>>> cell = tlx.nn.RNN(input_size=16, hidden_size=32, bias=True, num_layers=2,␣
↪bidirectional = True, act='tanh', batch_first=False, dropout=0, name='rnn_1
↪')
>>> y, h= cell(input, prev_h)
>>> print(y.shape)
```

LSTM layer

class tensorlayerx.nn.**LSTM**(*input_size*, *hidden_size*, *num_layers=1*, *bias=True*, *batch_first=False*, *dropout=0.0*, *bidirectional=False*, *name=None*)

Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

Parameters

- **input_size** (*int*) – The number of expected features in the input *x*
- **hidden_size** (*int*) – The number of features in the hidden state *h*
- **num_layers** (*int*) – Number of recurrent layers. Default: 1

- **bias** (*bool*) – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** (*bool*) – If `True`, then the input and output tensors are provided as *[batch_size, seq, input_size]*, Default: `False`
- **dropout** (*float*) – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to *dropout*. Default: 0
- **bidirectional** (*bool*) – If `True`, becomes a bidirectional LSTM. Default: `False`
- **name** (*None or str*) – A unique layer name

Returns

- **outputs** (*tensor*) – the output sequence. if *batch_first* is True, the shape is *[batch_size, seq, num_directions * hidden_size]*, else, the shape is *[seq, batch_size, num_directions * hidden_size]*.
- **final_states** (*tensor*) – final states. A tuple of two tensor. The shape of each is *[num_layers * num_directions, batch_size, hidden_size]*. Note that if the LSTM is Bidirectional, the forward states are (0,2,4,6,. . .) and the backward states are (1,3,5,7,. . .).

forward(*input*, *states=None*)

Parameters

- **inputs** (*tensor*) – the input sequence. if *batch_first* is True, the shape is *[batch_size, seq, input_size]*, else, the shape is *[seq, batch_size, input_size]*.
- **initial_states** (*tensor or None*) – the initial states. A tuple of tensor (h, c), the shape of each is *[num_layers * num_directions, batch_size, hidden_size]*.If initial_state is not given, zero initial states are used. If the LSTM is Bidirectional, num_directions should be 2, else it should be 1. Default: None.

Examples

With TensorLayerx

```
>>> input = tlx.nn.Input([23, 32, 16], name='input')
>>> prev_h = tlx.nn.Input([4, 32, 32])
>>> prev_c = tlx.nn.Input([4, 32, 32])
>>> cell = tlx.nn.LSTM(input_size=16, hidden_size=32, bias=True, num_layers=2,
↪ bidirectional = True,  batch_first=False, dropout=0, name='lstm_1')
>>> y, (h, c)= cell(input, (prev_h, prev_c))
>>> print(y.shape)
```

GRU layer

class tensorlayerx.nn.**GRU**(*input_size*, *hidden_size*, *num_layers=1*, *bias=True*, *batch_first=False*, *dropout=0.0*, *bidirectional=False*, *name=None*)

Applies a multi-layer gated recurrent unit (GRU) RNN to an input sequence.

Parameters

- **input_size** (*int*) – The number of expected features in the input *x*
- **hidden_size** (*int*) – The number of features in the hidden state *h*
- **num_layers** (*int*) – Number of recurrent layers. Default: 1

- **bias** (*bool*) – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** (*bool*) – If `True`, then the input and output tensors are provided as *[batch_size, seq, input_size]*, Default: `False`
- **dropout** (*float*) – If non-zero, introduces a *Dropout* layer on the outputs of each GRU layer except the last layer, with dropout probability equal to *dropout*. Default: 0
- **bidirectional** (*bool*) – If `True`, becomes a bidirectional LSTM. Default: `False`
- **name** (*None or str*) – A unique layer name

Returns

- **outputs** (*tensor*) – the output sequence. if *batch_first* is True, the shape is *[batch_size, seq, num_directions * hidden_size]*, else, the shape is *[seq, batch_size, num_directions * hidden_size]*.
- **final_states** (*tensor*) – final states. A tuple of two tensor. The shape of each is *[num_layers * num_directions, batch_size, hidden_size]*. Note that if the GRU is Bidirectional, the forward states are (0,2,4,6,. . .) and the backward states are (1,3,5,7,. . . .).

forward(*input*, *states=None*)

Parameters

- **inputs** (*tensor*) – the input sequence. if *batch_first* is True, the shape is *[batch_size, seq, input_size]*, else, the shape is *[seq, batch_size, input_size]*.
- **initial_states** (*tensor or None*) – the initial states. A tuple of tensor (h, c), the shape of each is *[num_layers * num_directions, batch_size, hidden_size]*.If initial_state is not given, zero initial states are used. If the GRU is Bidirectional, num_directions should be 2, else it should be 1. Default: None.

Examples

With TensorLayerx

```
>>> input = tlx.nn.Input([23, 32, 16], name='input')
>>> prev_h = tlx.nn.Input([4, 32, 32])
>>> cell = tlx.nn.GRU(input_size=16, hidden_size=32, bias=True, num_layers=2,␣
↪bidirectional = True,  batch_first=False, dropout=0, name='GRU_1')
>>> y, h= cell(input, prev_h)
>>> print(y.shape)
```

2.6.16 Transformer Layers

Transformer layer

MultiheadAttention layer

class tensorlayerx.nn.**MultiheadAttention**(*embed_dim*, *num_heads*, *dropout=0.0*, *kdim=None*, *vdim=None*, *bias=True*, *batch_first=False*, *need_weights=True*, *name=None*)

Allows the model to jointly attend to information from different representation subspaces.

Parameters

- **embed_dim** (*int*) – total dimension of the model.
- **num_heads** (*int*) – The number of heads in multi-head attention.
- **dropout** (*float*) – a Dropout layer on attn_output_weights. Default: 0.0.
- **kdim** (*int*) – total number of features in key. Default: None.
- **vdim** (*int*) – total number of features in value. Default: None.
- **bias** (*bool*) – add bias as module parameter. Default: True.
- **batch_first** (*bool*) – If `True`, then the input and output tensors are provided as *[batch, seq, feature]*. Default: `False` *[seq, batch, feature]*.
- **need_weights** (*bool*) – Indicate whether to return the attention weights. Default `False`.
- **name** (*None or str*) – A unique layer name.

Examples

With TensorLayerX

```
>>> q = tlx.nn.Input(shape=(4,2,128))
>>> attn_mask = tlx.convert_to_tensor(np.zeros((4,4)),dtype='bool')
>>> layer = MultiheadAttention(embed_dim=128, num_heads=4)
>>> output = layer(q, attn_mask=attn_mask)
```

References

- Attention Is All You Need

forward(*q*, *k=None*, *v=None*, *attn_mask=None*, *key_padding_mask=None*)

Parameters

- **q** (*Tensor*) – The queries for multi-head attention. If *batch_first* is `True`, it is a tensor with shape *[batch_size, query_length, embed_dim]*. If *batch_first* is `False`, it is a tensor with shape *[query_length, batch_size, embed_dim]*. The data type should be float32 or float64.
- **k** (*Tensor*) – The keys for multi-head attention. It is a tensor with shape *[batch_size, key_length, kdim]*. If *batch_first* is `False`, it is a tensor with shape *[key_length, batch_size, kdim]*. The data type should be float32 or float64. If None, use *query* as *key*. Default is *None*.
- **v** (*Tensor*) – The values for multi-head attention. It is a tensor with shape *[batch_size, value_length, vdim]*. If *batch_first* is `False`, it is a tensor with shape *[value_length, batch_size, vdim]*. The data type should be float32 or float64. If None, use *value* as *key*. Default is *None*.
- **attn_mask** (*Tensor*) – 2D or 3D mask that prevents attention to certain positions. A 2D mask will be broadcasted for all the batches while a 3D mask allows to specify a different mask for the entries of each batch. if a 2D mask: (L, S) where L is the target sequence length, S is the source sequence length. if a 3D mask: $(N \cdot extnum\_heads, L, S)$. Where N is the batch size, L is the target sequence length, S is the source sequence length. `attn_mask` ensure that position i is allowed to attend the unmasked positions. If a

ByteTensor is provided, the non-zero positions are not allowed to attend while the zero positions will be unchanged. If a BoolTensor is provided, positions with `True` is not allowed to attend while `False` values will be unchanged. If a FloatTensor is provided, it will be added to the attention weight.

- **key_padding_mask** (*Tensor*) – if provided, specified padding elements in the key will be ignored by the attention. When given a binary mask and a value is True, the corresponding value on the attention layer will be ignored. When given a byte mask and a value is non-zero, the corresponding value on the attention layer will be ignored (N, S) where N is the batch size, S is the source sequence length. If a ByteTensor is provided, the non-zero positions will be ignored while the position with the zero positions will be unchanged. If a BoolTensor is provided, the positions with the value of `True` will be ignored while the position with the value of `False` will be unchanged.

Returns

- **attn_output** (*Tensor*) – (L, N, E) where L is the target sequence length, N is the batch size, E is the embedding dimension. (N, L, E) if `batch_first` is `True`.
- *attn_output_weights* – (N, L, S) where N is the batch size, L is the target sequence length, S is the source sequence length.

Transformer layer

class tensorlayerx.nn.**Transformer**(*d_model=512*, *nhead=8*, *num_encoder_layers=6*, *num_decoder_layers=6*, *dim_feedforward=2048*, *dropout=0.1*, *act='relu'*, *custom_encoder=None*, *custom_decoder=None*, *layer_norm_eps=1e-05*, *batch_first=False*)

A transformer model. User is able to modify the attributes as needed.

Parameters

- **d_model** (*int*) – the number of expected features in the encoder/decoder inputs.
- **nhead** (*int*) – the number of heads in the multiheadattention model.
- **num_encoder_layers** – the number of sub-encoder-layers in the encoder.
- **num_decoder_layers** – the number of sub-decoder-layers in the decoder.
- **dim_feedforward** (*int*) – the dimension of the feedforward network model.
- **dropout** (*float*) – a Dropout layer on attn_output_weights. Default: 0.0.
- **act** (*str*) – the activation function of encoder/decoder intermediate layer, 'relu' or 'gelu'. Default: 'relu'.
- **custom_encoder** (*Module or None*) – custom encoder.
- **custom_decoder** (*Module or None*) – custom decoder
- **layer_norm_eps** (*float*) – the eps value in layer normalization components. Default: 1e-5.
- **batch_first** (*bool*) – If `True`, then the input and output tensors are provided as *[batch, seq, feature]*. Default: `False` *[seq, batch, feature]*.

Examples

With TensorLayerX

```
>>> src = tlx.nn.Input(shape=(4,2,128))
>>> tgt = tlx.nn.Input(shape=(4,2,128))
>>> layer = Transformer(d_model=128, nhead=4)
>>> output = layer(src, tgt)
```

References

- Attention Is All You Need
- BERT

forward(*src*, *tgt*, *src_mask=None*, *tgt_mask=None*, *memory_mask=None*, *src_key_padding_mask=None*, *tgt_key_padding_mask=None*, *memory_key_padding_mask=None*)

Parameters

- **src** (*Tensor*) – the sequence to the encoder.
- **tgt** (*Tensor*) – the sequence to the decoder.
- **src_mask** (*Tensor*) – the additive mask for the src sequence.
- **tgt_mask** (*Tensor*) – the additive mask for the tgt sequence.
- **memory_mask** (*Tensor*) – the additive mask for the encoder output.
- **src_key_padding_mask** (*Tensor*) – mask for src keys per batch.
- **tgt_key_padding_mask** (*Tensor*) – mask for tgt keys per batch.
- **memory_key_padding_mask** (*Tensor*) – mask for memory keys per batch.

generate_square_subsequent_mask(*length*)

Generate a square mask for the sequence. The masked positions are filled with float('-inf'). Unmasked positions are filled with float(0.0).

Parameters **length** (*int*) – The length of sequence.

Examples

With TensorLayerX

```
>>> length = 5
>>> mask = transformer.generate_square_subsequent_mask(length)
>>> print(mask)
>>> [[  0. -inf -inf -inf -inf]
>>> [  0.   0. -inf -inf -inf]
>>> [  0.   0.   0. -inf -inf]
>>> [  0.   0.   0.   0. -inf]
>>> [  0.   0.   0.   0.   0.]]
```

TransformerEncoder layer

class tensorlayerx.nn.**TransformerEncoder**(*encoder_layer*, *num_layers*, *norm=None*)

TransformerEncoder is a stack of N encoder layers

Parameters

- **encoder_layer** (Module) – an instance of the TransformerEncoderLayer() class.
- **num_layers** (*int*) – the number of sub-encoder-layers in the encoder.
- **norm** (*None*) – the layer normalization component.

Examples

With TensorLayerX

```
>>> q = tlx.nn.Input(shape=(4,2,128))
>>> attn_mask = tlx.convert_to_tensor(np.zeros((4,4)),dtype='bool')
>>> encoder = TransformerEncoderLayer(128, 2, 256)
>>> encoder = TransformerEncoder(encoder, num_layers=3)
>>> output = encoder(q, mask=attn_mask)
```

forward(*src*, *mask=None*, *src_key_padding_mask=None*)

Parameters

- **src** (*Tensor*) – the sequence to the encoder.
- **mask** (*Tensor*) – the mask for the src sequence.
- **src_key_padding_mask** – the mask for the src keys per batch.

TransformerDecoder layer

class tensorlayerx.nn.**TransformerDecoder**(*decoder_layer*, *num_layers*, *norm=None*)

TransformerDecoder is a stack of N decoder layers

Parameters

- **decoder_layer** (Module) – an instance of the TransformerDecoderLayer() class.
- **num_layers** (*int*) – the number of sub-decoder-layers in the decoder.
- **norm** (*None*) – the layer normalization component.

Examples

With TensorLayerX

```
>>> q = tlx.nn.Input(shape=(4,2,128))
>>> decoder = TransformerDecoderLayer(128, 2, 256)
>>> decoder = TransformerDecoder(decoder, num_layers=3)
>>> output = decoder(q, q)
```

forward(*tgt*, *memory*, *tgt_mask=None*, *memory_mask=None*, *tgt_key_padding_mask=None*, *memory_key_padding_mask=None*)

Parameters

- **tgt** (*Tensor*) – the sequence to the decoder.
- **memory** (*Tensor*) – the sequence from the last layer of the encoder.
- **tgt_mask** (*Tensor*) – the mask for the tgt sequence.
- **memory_mask** (*Tensor*) – the mask for the memory sequence.
- **tgt_key_padding_mask** (*Tensor*) – the mask for the tgt keys per batch.
- **memory_key_padding_mask** (*Tensor*) – the mask for the memory keys per batch.

TransformerEncoderLayer layer

class tensorlayerx.nn.**TransformerEncoderLayer**(*d_model*, *nhead*, *dim_feedforward*, *dropout=0.1*, *act='relu'*, *layer_norm_eps=1e-05*, *batch_first=False*)

TransformerEncoderLayer is made up of self-attn and feedforward network. This standard encoder layer is based on the paper "Attention Is All You Need".

Parameters

- **d_model** (*int*) – total dimension of the model.
- **nhead** (*int*) – The number of heads in multi-head attention.
- **dim_feedforward** (*int*) – the dimension of the feedforward network model.
- **dropout** (*float*) – a Dropout layer on attn_output_weights. Default: 0.1.
- **act** (*str*) – The activation function in the feedforward network. 'relu' or 'gelu'. Default 'relu'.
- **layer_norm_eps** (*float*) – the eps value in layer normalization components. Default 1e-5.
- **batch_first** (*bool*) – If `True`, then the input and output tensors are provided as *[batch, seq, feature]*. Default: `False` *[seq, batch, feature]*.

Examples

With TensorLayerX

```
>>> q = tlx.nn.Input(shape=(4,2,128))
>>> attn_mask = tlx.convert_to_tensor(np.zeros((4,4)),dtype='bool')
>>> encoder = TransformerEncoderLayer(128, 2, 256)
>>> output = encoder(q, src_mask=attn_mask)
```

forward(*src*, *src_mask=None*, *src_key_padding_mask=None*)

Parameters

- **src** (*Tensor*) – the sequence to the encoder layer.
- **src_mask** (*Tensor or None*) – the mask for the src sequence.
- **src_key_padding_mask** (*Tensor or None*) – the mask for the src keys per batch.

TransformerDecoderLayer layer

class tensorlayerx.nn.**TransformerDecoderLayer**(*d_model*, *nhead*, *dim_feedforward*, *dropout=0.1*, *act='relu'*, *layer_norm_eps=1e-05*, *batch_first=False*)

TransformerDecoderLayer is made up of self-attn, multi-head-attn and feedforward network. This standard decoder layer is based on the paper "Attention Is All You Need".

Parameters

- **d_model** (*int*) – total dimension of the model.
- **nhead** (*int*) – The number of heads in multi-head attention.
- **dim_feedforward** (*int*) – the dimension of the feedforward network model.
- **dropout** (*float*) – a Dropout layer on attn_output_weights. Default: 0.1.
- **act** (*str*) – The activation function in the feedforward network. 'relu' or 'gelu'. Default 'relu'.
- **layer_norm_eps** (*float*) – the eps value in layer normalization components. Default 1e-5.
- **batch_first** (*bool*) – If `True`, then the input and output tensors are provided as *[batch, seq, feature]*. Default: `False` *[seq, batch, feature]*.

Examples

With TensorLayerX

```
>>> q = tlx.nn.Input(shape=(4,2,128))
>>> encoder = TransformerDecoderLayer(128, 2, 256)
>>> output = encoder(q, q)
```

forward(*tgt*, *memory*, *tgt_mask=None*, *memory_mask=None*, *tgt_key_padding_mask=None*, *memory_key_padding_mask=None*)

Parameters

- **tgt** (*Tensor*) – the sequence to the decoder layer.
- **memory** – the sequence from the last layer of the encoder.
- **tgt_mask** – the mask for the tgt sequence.
- **memory_mask** – the mask for the memory sequence.
- **tgt_key_padding_mask** – the mask for the tgt keys per batch.
- **memory_key_padding_mask** – the mask for the memory keys per batch.

2.6.17 Shape Layers

Flatten Layer

class tensorlayerx.nn.**Flatten**(*name=None*)

A layer that reshapes high-dimension input into a vector.

Then we often apply Dense, RNN, Concat and etc on the top of a flatten layer. [batch_size, mask_row, mask_col, n_mask] —> [batch_size, mask_row * mask_col * n_mask]

Parameters **name** (*None or str*) – A unique layer name.

Examples

```
>>> x = tlx.nn.Input([8, 4, 3], name='input')
>>> y = tlx.nn.Flatten(name='flatten')(x)
[8, 12]
```

Reshape Layer

class tensorlayerx.nn.**Reshape**(*shape*, *name=None*)

A layer that reshapes a given tensor.

Parameters

- **shape** (*tuple of int*) – The output shape, see `tf.reshape`.
- **name** (*str*) – A unique layer name.

Examples

```
>>> x = tlx.nn.Input([8, 4, 3], name='input')
>>> y = tlx.nn.Reshape(shape=[-1, 12], name='reshape')(x)
(8, 12)
```

Transpose Layer

class tensorlayerx.nn.**Transpose**(*perm=None*, *conjugate=False*, *name=None*)

A layer that transposes the dimension of a tensor.

See tf.transpose() .

Parameters

- **perm** (*list of int or None*) – The permutation of the dimensions, similar with `numpy.transpose`. If None, it is set to (n-1...0), where n is the rank of the input tensor.
- **conjugate** (*bool*) – By default False. If True, returns the complex conjugate of complex numbers (and transposed) For example [[1+1j, 2+2j]] –> [[1-1j], [2-2j]]
- **name** (*str*) – A unique layer name.

Examples

```
>>> x = tlx.nn.Input([8, 4, 3], name='input')
>>> y = tlx.nn.Transpose(perm=[0, 2, 1], conjugate=False, name='trans')(x)
(8, 3, 4)
```

Shuffle Layer

class tensorlayerx.nn.**Shuffle**(*group*, *in_channels=None*, *name=None*)
A layer that shuffle a 2D image [batch, height, width, channel], see here.

Parameters

- **group** (*int*) – The number of groups.
- **name** (*str*) – A unique layer name.

Examples

```
>>> x = tlx.nn.Input([1, 16, 16, 8], name='input')
>>> y = tlx.nn.Shuffle(group=2, name='shuffle')(x)
(1, 16, 16, 8)
```

2.6.18 Stack Layer

Stack Layer

class tensorlayerx.nn.**Stack**(*axis=1*, *name=None*)
The *Stack* class is a layer for stacking a list of rank-R tensors into one rank-(R+1) tensor, see tf.stack().

Parameters

- **axis** (*int*) – New dimension along which to stack.
- **name** (*str*) – A unique layer name.

Examples

```
>>> import tensorlayerx as tlx
>>> ni = tlx.nn.Input([10, 784], name='input')
>>> net1 = tlx.nn.Linear(10, name='linear1')(ni)
>>> net2 = tlx.nn.Linear(10, name='linear2')(ni)
>>> net3 = tlx.nn.Linear(10, name='linear3')(ni)
>>> net = tlx.nn.Stack(axis=1, name='stack')([net1, net2, net3])
(10, 3, 10)
```

Unstack Layer

class tensorlayerx.nn.**UnStack**(*num=None*, *axis=0*, *name=None*)
The *UnStack* class is a layer for unstacking the given dimension of a rank-R tensor into rank-(R-1) tensors., see tf.unstack().

Parameters

- **num** (*int or None*) – The length of the dimension axis. Automatically inferred if None (the default).
- **axis** (*int*) – Dimension along which axis to concatenate.
- **name** (*str*) – A unique layer name.

Returns The list of layer objects unstacked from the input.

Return type list of `Layer`

Examples

```
>>> ni = tlx.nn.Input([4, 10], name='input')
>>> nn = tlx.nn.Linear(n_units=5)(ni)
>>> nn = tlx.nn.UnStack(axis=1)(nn)  # unstack in channel axis
>>> len(nn)  # 5
>>> nn[0].shape  # (4,)
```

2.7 API - Model Training

TensorLayerX provides two model training interfaces, which can satisfy the training of various deep learning tasks.

Model(network[, loss_fn, optimizer, metrics])	High-Level API for Training or Testing.
WithLoss(backbone, loss_fn)	High-Level API for Training or Testing.
WithGrad(network[, loss_fn, optimizer])	Module that returns the gradients.
TrainOneStep(net_with_loss, optimizer, ...)	High-Level API for Training One Step.

2.7.1 Model

tensorlayerx.model.**Model**(*network*, *loss_fn=None*, *optimizer=None*, *metrics=None*, ***kwargs*)

High-Level API for Training or Testing.

Model groups layers into an object with training and inference features.

Parameters

- **network** (*tensorlayer model*) – The training or testing network.
- **loss_fn** (*function*) – Objective function
- **optimizer** (*class*) – Optimizer for updating the weights
- **metrics** (*class*) – Dict or set of metrics to be evaluated by the model during

tensorlayerx.model.**trin**()

Model training.

tensorlayerx.model.**eval**()

Model prediction.

tensorlayerx.model.**save_weights**()

Input file_path, save model weights into a file of given format. Use load_weights() to restore.

tensorlayerx.model.**load_weights**()

Load model weights from a given file, which should be previously saved by save_weights().

Examples

```
>>> import tensorlayerx as tlx
>>> class Net(Module):
>>>     def __init__(self):
>>>         super(Net, self).__init__()
>>>         self.conv = tlx.nn.Conv2d(out_channels=32, kernel_size=(3, 3),␣
↪stride=(2, 2), in_channels=5, name='conv2d')
>>>         self.bn = tlx.nn.BatchNorm2d(num_features=32, act=tlx.ReLU)
>>>         self.flatten = tlx.nn.Flatten()
>>>         self.fc = tlx.nn.Linear(out_features=12, in_features=32*224*224) #␣
↪padding=0
>>>
>>>     def construct(self, x):
>>>         x = self.conv(x)
>>>         x = self.bn(x)
>>>         x = self.flatten(x)
>>>         out = self.fc(x)
>>>         return out
>>>
>>> net = Net()
>>> loss = tlx.losses.softmax_cross_entropy_with_logits
>>> optim = tlx.optimizers.Momentum(params=net.trainable_weights, learning_rate=0.
↪1, momentum=0.9)
>>> model = Model(net, loss_fn=loss, optimizer=optim, metrics=None)
>>> dataset = get_dataset()
>>> model.train(2, dataset)
```

2.7.2 WithLoss

tensorlayerx.model.**WithLoss**(*backbone*, *loss_fn*)

High-Level API for Training or Testing.

Wraps the network with loss function. This Module accepts data and label as inputs and the computed loss will be returned.

Parameters

- **backbone** (*tensorlayer model*) – The tensorlayer network.
- **loss_fn** (*function*) – Objective function

tensorlayerx.model.**forward**()

Model inference.

Examples

```
>>> import tensorlayerx as tlx
>>> net = vgg16()
>>> loss_fn = tlx.losses.softmax_cross_entropy_with_logits
>>> net_with_loss = tlx.model.WithLoss(net, loss_fn)
```

2.7.3 WithGrad

tensorlayerx.model.**WithGrad**(*network*, *loss_fn=None*, *optimizer=None*)

Module that returns the gradients.

Parameters

- **network** (*tensorlayer model*) – The tensorlayer network.
- **loss_fn** (*function*) – Objective function
- **optimizer** (*class*) – Optimizer for updating the weights

Examples

```
>>> import tensorlayerx as tlx
>>> net = vgg16()
>>> loss_fn = tlx.losses.softmax_cross_entropy_with_logits
>>> optimizer = tlx.optimizers.Adam(learning_rate=1e-3)
>>> net_with_grad = tlx.model.WithGrad(net, loss_fn, optimizer)
>>> inputs, labels = tlx.nn.Input((128, 784), dtype=tlx.float32), tlx.nn.
↪Input((128, 1), dtype=tlx.int32)
>>> net_with_grad(inputs, labels)
```

2.7.4 TrainOneStep

tensorlayerx.model.**TrainOneStep**(*net_with_loss*, *optimizer*, *train_weights*)

High-Level API for Training One Step.

Wraps the network with an optimizer. It can be trained in one step using the optimizer to get the loss.

Parameters

- **net_with_loss** (*tensorlayer WithLoss*) – The training or testing network.
- **optimizer** (*class*) – Optimizer for updating the weights
- **train_weights** (*class*) – Dict or set of metrics to be evaluated by the model during

Examples

```
>>> import tensorlayerx as tlx
>>> net = vgg16()
>>> train_weights = net.trainable_weights
>>> loss_fn = tlx.losses.softmax_cross_entropy_with_logits
>>> optimizer = tlx.optimizers.Adam(learning_rate=1e-3)
>>> net_with_loss = tlx.model.WithLoss(net, loss_fn)
>>> train_one_step = tlx.model.TrainOneStep(net_with_loss, optimizer, train_
↪weights)
>>> inputs, labels = tlx.nn.Input((128, 784), dtype=tlx.float32), tlx.nn.
↪Input((128, 1), dtype=tlx.int32)
>>> train_one_step(inputs, labels)
```

2.8 API - Vision

2.8.1 Vision Transforms list

ToTensor([data_format])	Convert a `PIL Image` or `numpy.ndarray` to tensor.
Compose(transforms)	Composes several transforms together.
Crop(top, left, height, width)	Crops an image to a specified bounding box.
CentralCrop([size, central_fraction])	Crops the given image at the center.If the size is given, image will be cropped as size.
RandomCrop(size[, padding, pad_if_needed, ...])	Crop the given image at a random location.
Pad(padding[, padding_value, mode])	Pad the given image on all sides with the given "pad" value.
PadToBoundingbox(top, left, height, width[, ...])	Pad image with the specified height and width to target size.
Resize(size[, interpolation])	Resize the input image to the given size.
RandomResizedCrop(size[, scale, ratio, ...])	Crop the given image to random size and aspect ratio.
RgbToGray([num_output_channels])	Converts a image from RGB to grayscale.
HsvToRgb	Converts a image from HSV to RGB.
RgbToHsv	Converts a image from RGB to HSV.
AdjustBrightness([brightness_factor])	Adjust brightness of the image.
AdjustContrast([contrast_factor])	Adjust contrast of the image.
AdjustHue([hue_factor])	Adjust hue of the image.
AdjustSaturation([saturation_factor])	Adjust saturation of the image.
RandomBrightness([brightness_factor])	Random adjust brightness of the image.
RandomContrast([contrast_factor])	Random adjust contrast of the image.
RandomHue([hue_factor])	Random adjust hue of the image.
RandomSaturation([saturation_factor])	Random adjust saturation of the image.
ColorJitter([brightness, contrast, ...])	Randomly change the brightness, contrast, saturation and hue of an image.
FlipHorizontal	Flip an image horizontally.
FlipVertical	Flip an image vertically.
RandomFlipHorizontal([prob])	Horizontally flip the given image randomly with a given probability.
RandomFlipVertical([prob])	Vertically flip the given image randomly with a given probability.
Rotation([angle, interpolation, expand, ...])	Rotate the image by angle.
RandomRotation(degrees[, interpolation, ...])	Rotate the image by random angle.
RandomShift(shift[, interpolation, fill])	Shift the image by random translations.
RandomShear(shear[, interpolation, fill])	Shear the image by random angle.
RandomZoom(zoom[, interpolation, fill])	Zoom the image by random scale.
RandomAffine(degrees[, shift, zoom, shear, ...])	Random affine transformation of the image keeping center invariant.
Transpose(order)	Transpose image(s) by swapping dimension.
HWC2CHW	Transpose a image shape (H, W, C) to shape (C, H, W).
CHW2HWC	Transpose a image shape (C, H, W) to shape (H, W, C).
Normalize(mean, std[, data_format])	Normalize a tensor image with mean and standard deviation.
StandardizePerImage	For each 3-D image x in image, computes (x - mean) / adjusted_stddev, where mean is the average of all values in x.

2.8.2 Vision IO list

load_image(path)	Load an image
save_image(image, file_name, path)	Save an image
load_images(path[, n_threads])	Load images from file
save_images(images, file_names, path)	Save images

2.8.3 Vision Transforms

ToTensor

class tensorlayerx.vision.transforms.**ToTensor**(*data_format='HWC'*)

Convert a `PIL Image` or `numpy.ndarray` to tensor.

Parameters **data_format** (*str*) – Data format of output tensor, should be 'HWC' or 'CHW'. Default: 'HWC'.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.ToTensor(data_format='HWC')
>>> image = transform(image)
>>> print(image)
```

Compose

class tensorlayerx.vision.transforms.**Compose**(*transforms*)

Composes several transforms together.

Parameters **transforms** (*list of 'transform' objects*) – list of transforms to compose.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.Compose([tlx.vision.transforms.
↪ToTensor(data_format='HWC'),tlx.vision.transforms.CentralCrop(size = 100)])
>>> image = transform(image)
>>> print(image)
>>> image shape : (100, 100, 3)
```

Crop

class tensorlayerx.vision.transforms.**Crop**(*top*, *left*, *height*, *width*)

Crops an image to a specified bounding box.

Parameters

- **top** (*int*) – Vertical coordinate of the top-left corner of the bounding box in image.
- **left** (*int*) – Horizontal coordinate of the top-left corner of the bounding box in image.
- **height** (*int*) – Height of the bounding box.
- **width** (*int*) – Width of the bounding box.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.Crop(top=10, left=10, height=100, width=100)
>>> image = transform(image)
>>> print(image)
>>> image shape : (100, 100, 3)
```

CentralCrop

class tensorlayerx.vision.transforms.**CentralCrop**(*size=None*, *central_fraction=None*)

Crops the given image at the center.If the size is given, image will be cropped as size. If the central_fraction is given, image will cropped as (H * central_fraction, W * fraction). Size has a higher priority.

Parameters

- **size** (*int or sequence of int*) –
  - The output size of the cropped image.
  - If size is an integer, a square crop of size (size, size) is returned.
  - If size is a sequence of length 2, it should be (height, width).
- **central_fraction** (*float*) – float (0, 1], fraction of size to crop

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.CentralCrop(size = (50, 50))
>>> image = transform(image)
>>> print(image)
>>> image shape : (50, 50, 3)
```

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.CentralCrop(central_fraction=0.5)
>>> image = transform(image)
>>> print(image)
>>> image shape : (112, 112, 3)
```

RandomCrop

class tensorlayerx.vision.transforms.**RandomCrop**(*size*, *padding=None*, *pad_if_needed=False*, *fill=0*, *padding_mode='constant'*)

Crop the given image at a random location.

Parameters

- **size** (*int or sequence*) –
  - Desired output size of the crop.
  - If size is an int instead of sequence like (h, w), a square crop (size, size) is made.
  - If provided a sequence of length 1, it will be interpreted as (size[0], size[0]).
- **padding** (*int or sequence, optional*) –
  - Optional padding on each border of the image.
  - If a single int is provided this is used to pad all borders.
  - If sequence of length 2 is provided this is the padding on left/right and top/bottom respectively.
  - If a sequence of length 4 is provided, it is used to pad left, top, right, bottom borders respectively.
  - Default: 0.
- **pad_if_needed** (*boolean*) – It will pad the image if smaller than the desired size to avoid raising an exception. Since cropping is done after padding, the padding seems to be done at a random offset.
- **fill** (*number or sequence*) – Pixel fill value for constant fill. Default is 0. If a tuple of length 3, it is used to fill R, G, B channels respectively.
- **padding_mode** (*str*) – Type of padding. Default is "constant"."constant", "reflect", "symmetric" are supported.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.RandomCrop(size=50, padding=10, pad_if_
↪needed=False, fill=0, padding_mode='constant')
>>> image = transform(image)
>>> print(image)
>>> image shape : (70,70,3)
```

Pad

class tensorlayerx.vision.transforms.**Pad**(*padding*, *padding_value=0*, *mode='constant'*)

Pad the given image on all sides with the given "pad" value.

Parameters

- **padding** (*int or sequenece*) –

  - Padding on each border.
  - If a single int is provided this is used to pad all borders.
  - If sequence of length 2 is provided this is the padding on left/right and top/bottom respectively.
  - If a sequence of length 4 is provided this is the padding for the left, top, right and bottom borders respectively.
- **padding_value** (*number or sequenece*) – Pixel fill value for constant fill. Default is 0. If a tuple or list of length 3, it is used to fill R, G, B channels respectively. tuple and list only is supported for PIL Image. This value is only used when the mode is constant.
- **mode** (*str*) – Type of padding. Default is "constant"."constant", "reflect", "symmetric" , "edge" are supported.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.Pad(padding=10, padding_value=0, mode=
↪'constant')
>>> image = transform(image)
>>> print(image)
>>> image shape : (244, 244, 3)
```

PadToBoundingbox

class tensorlayerx.vision.transforms.**PadToBoundingbox**(*top*, *left*, *height*, *width*, *padding_value=0*)

Pad image with the specified height and width to target size.

Parameters

- **top** (*int*) – Number of rows to add on top.
- **left** (*int*) – Number of columns to add on the left.
- **height** (*int*) – Height of output image.
- **width** (*int*) – Width of output image.
- **padding_value** (*int or sequence*) – value to pad.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand( 224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.PadToBoundingbox(top=10, left=10,␣
↪height=300, width=300, padding_value=0)
>>> image = transform(image)
>>> print(image)
>>> image shape : (300, 300, 3)
```

Resize

class tensorlayerx.vision.transforms.**Resize**(*size*, *interpolation='bilinear'*)

Resize the input image to the given size.

Parameters

- **size** (*int or sequenece*) –
  - Desired output size.
  - If size is a sequence like (h, w), output size will be matched to this.
  - If size is an int, smaller edge of the image will be matched to this number.
  - i.e, if height > width, then image will be rescaled to (size * height / width, size).
- **interpolation** (*str*) – Interpolation method. Default: 'bilinear'. 'nearest', 'bilinear', 'bicubic', 'area' and 'lanczos' are supported.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.Resize(size = (100,100), interpolation=
↪'bilinear')
>>> image = transform(image)
>>> print(image)
>>> image shape : (100, 100, 3)
```

RandomResizedCrop

class tensorlayerx.vision.transforms.**RandomResizedCrop**(*size*, *scale=(0.08, 1.0)*, *ratio=(0.75, 1.3333333333333333)*, *interpolation='bilinear'*)

Crop the given image to random size and aspect ratio.

Parameters

- **size** (*int or sequence*) –
  - Desired output size of the crop.
  - If size is an int instead of sequence like (h, w), a square crop (size, size) is made.
  - If provided a sequence of length 1, it will be interpreted as (size[0], size[0]).
- **scale** (*tuple of float*) – scale range of the cropped image before resizing, relatively to the origin image.
- **ratio** (*tuple of float*) – aspect ratio range of the cropped image before resizing.
- **interpolation** (*str*) – Type of interpolation. Default is "bilinear"."nearest","bilinear" and "bicubic" are supported.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.RandomResizedCrop(size = (100, 100), scale␣
↪= (0.08, 1.0), ratio = (3./4.,4./3.), interpolation = 'bilinear')
>>> image = transform(image)
>>> print(image)
>>> image shape : (100,100,3)
```

RgbToGray

class tensorlayerx.vision.transforms.**RgbToGray**(*num_output_channels=1*)

Converts a image from RGB to grayscale.

Parameters **num_output_channels** (*int*) – (1 or 3) number of channels desired for output image. Default is 1.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.RgbToGray(num_output_channels=1)
>>> image = transform(image)
>>> print(image)
>>> image shape : (224, 224, 1)
```

HsvToRgb

class tensorlayerx.vision.transforms.**HsvToRgb**

Converts a image from HSV to RGB.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.HsvToRgb()
>>> image = transform(image)
>>> print(image)
>>> image shape : (224, 224, 3)
```

RgbToHsv

class tensorlayerx.vision.transforms.**RgbToHsv**

Converts a image from RGB to HSV.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.RgbToHsv()
>>> image = transform(image)
>>> print(image)
>>> image shape : (224, 224, 3)
```

AdjustBrightness

class tensorlayerx.vision.transforms.**AdjustBrightness**(*brightness_factor=1*)

Adjust brightness of the image.

Parameters **brightness_factor** (*float*) – How much to adjust the brightness. Can be any non negative number. 1 gives the original image. Default is 1.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.AdjustBrightness(brightness_factor=1)
>>> image = transform(image)
>>> print(image)
```

AdjustContrast

class tensorlayerx.vision.transforms.**AdjustContrast**(*contrast_factor=1*)

Adjust contrast of the image.

Parameters **contrast_factor** (*float*) – How much to adjust the contrast. Can be any non negative number. 1 gives the original image. Default is 1.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.AdjustContrast(contrast_factor=1)
>>> image = transform(image)
>>> print(image)
```

AdjustHue

class tensorlayerx.vision.transforms.**AdjustHue**(*hue_factor=0*)

Adjust hue of the image.

Parameters **hue_factor** (*float*) – How much to shift the hue channel. Should be in [-0.5, 0.5]. 0.5 and -0.5 give complete reversal of hue channel in HSV space in positive and negative direction respectively. 0 means no shift. Therefore, both -0.5 and 0.5 will give an image with complementary colors while 0 gives the original image. Default is 0.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.AdjustHue(hue_factor=0)
>>> image = transform(image)
>>> print(image)
```

AdjustSaturation

class tensorlayerx.vision.transforms.**AdjustSaturation**(*saturation_factor=1*)

Adjust saturation of the image.

Parameters **saturation_factor** (*float*) – How much to adjust the saturation. Can be any non negative number. 1 gives the original image. Default is 1.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.AdjustSaturation(saturation_factor=1)
>>> image = transform(image)
>>> print(image)
```

RandomBrightness

class tensorlayerx.vision.transforms.**RandomBrightness**(*brightness_factor=(1, 1)*)

Random adjust brightness of the image.

Parameters **brightness_factor** (*float or sequence*) –

- Brightness adjustment factor (default=(1, 1)).
- If it is a float, the factor is uniformly chosen from the range [max(0, 1-brightness_factor), 1+brightness_factor].
- If it is a sequence, it should be [min, max] for the range.Should be non negative numbers.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.RandomBrightness(brightness_factor=(0.5, 2))
>>> image = transform(image)
>>> print(image)
```

RandomContrast

class tensorlayerx.vision.transforms.**RandomContrast**(*contrast_factor=(1, 1)*)

Random adjust contrast of the image.

Parameters **contrast_factor** (*float or sequence*) –

- Contrast adjustment factor (default=(1, 1)).
- If it is a float, the factor is uniformly chosen from the range [max(0, 1-contrast_factor), 1+contrast_factor].
- If it is a sequence, it should be [min, max] for the range.Should be non negative numbers.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.RandomContrast(contrast_factor=(0.5, 2))
>>> image = transform(image)
>>> print(image)
```

RandomHue

class tensorlayerx.vision.transforms.**RandomHue**(*hue_factor=(0, 0)*)

Random adjust hue of the image.

Parameters **hue_factor** (*float or sequence*) –

- Hue adjustment factor (default=(0, 0)).
- If it is a float, the factor is uniformly chosen from the range [-hue_factor, hue_factor].
- If it is a sequence, it should be [min, max] for the range.Should have 0<= hue <= 0.5 or -0.5 <= min <= max <= 0.5.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.RandomHue(hue_factor=(-0.5, 0.5))
>>> image = transform(image)
>>> print(image)
```

RandomSaturation

class tensorlayerx.vision.transforms.**RandomSaturation**(*saturation_factor=(1, 1)*)

Random adjust saturation of the image.

Parameters **saturation_factor** (*float or sequence*) –

- Saturation adjustment factor (default=(1, 1)).
- If it is a float, the factor is uniformly chosen from the range [max(0, 1-saturation_factor), 1+saturation_factor].
- If it is a sequence, it should be [min, max] for the range.Should be non negative numbers.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.RandomSaturation(saturation_factor=(0.5, 2))
>>> image = transform(image)
>>> print(image)
```

ColorJitter

class tensorlayerx.vision.transforms.**ColorJitter**(*brightness=0*, *contrast=0*, *saturation=0*, *hue=0*)

Randomly change the brightness, contrast, saturation and hue of an image.

Parameters

- **brightness** (*float or sequence*) –
  - Brightness adjustment factor (default=(1, 1)).
  - If it is a float, the factor is uniformly chosen from the range [max(0, 1-brightness_factor), 1+brightness_factor].
  - If it is a sequence, it should be [min, max] for the range.Should be non negative numbers.
- **contrast** (*float or sequence*) –
  - Contrast adjustment factor (default=(1, 1)).
  - If it is a float, the factor is uniformly chosen from the range [max(0, 1-contrast_factor), 1+contrast_factor].
  - If it is a sequence, it should be [min, max] for the range.Should be non negative numbers.
- **saturation** (*float or sequence*) –
  - Saturation adjustment factor (default=(1, 1)).
  - If it is a float, the factor is uniformly chosen from the range [max(0, 1-saturation_factor), 1+saturation_factor].
  - If it is a sequence, it should be [min, max] for the range.Should be non negative numbers.
- **hue** (*float or sequence*) –
  - Hue adjustment factor (default=(0, 0)).

- If it is a float, the factor is uniformly chosen from the range [-hue_factor, hue_factor].
- If it is a sequence, it should be [min, max] for the range.Should have 0<= hue <= 0.5 or -0.5 <= min <= max <= 0.5.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.ColorJitter(brightness=(1,5), contrast=(1,
↪5), saturation=(1,5), hue=(-0.2,0.2))
>>> image = transform(image)
>>> print(image)
```

FlipHorizontal

class tensorlayerx.vision.transforms.**FlipHorizontal**

Flip an image horizontally.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.FlipHorizontal()
>>> image = transform(image)
>>> print(image)
```

FlipVertical

class tensorlayerx.vision.transforms.**FlipVertical**

Flip an image vertically.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.FlipVertical()
>>> image = transform(image)
>>> print(image)
```

RandomFlipHorizontal

class tensorlayerx.vision.transforms.**RandomFlipHorizontal**(*prob=0.5*)

Horizontally flip the given image randomly with a given probability.

Parameters **prob** (*float*) – probability of the image being flipped. Default value is 0.5

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.RandomFlipHorizontal(prob = 0.5)
>>> image = transform(image)
>>> print(image)
```

RandomFlipVertical

class tensorlayerx.vision.transforms.**RandomFlipVertical**(*prob=0.5*)

Vertically flip the given image randomly with a given probability.

Parameters **prob** (*float*) – probability of the image being flipped. Default value is 0.5

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.RandomFlipVertical(prob = 0.5)
>>> image = transform(image)
>>> print(image)
```

Rotation

class tensorlayerx.vision.transforms.**Rotation**(*angle=0*, *interpolation='bilinear'*, *expand=False*, *center=None*, *fill=0*)

Rotate the image by angle.

Parameters

- **degrees** (*number*) – degrees to rotate.
- **interpolation** (*str*) – Interpolation method. Default is 'bilinear'. 'nearest','bilinear' are supported.
- **expand** (*boolean*) –
  - If true, expands the output to make it large enough to hold the entire rotated image.
  - If false or omitted, make the output image the same size as the input image.
  - Note that the expand flag assumes rotation around the center and no translation.
- **center** (*sequence or None*) – Optional center of rotation, (x, y). Origin is the upper left corner. Default is the center of the image.
- **fill** (*number or sequence*) – Pixel fill value for the area outside the rotated image. Default is 0.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.Rotation(angle = 0, interpolation =
↪'bilinear', expand = False, center = None, fill = 0)
>>> image = transform(image)
>>> print(image)
```

RandomRotation

class tensorlayerx.vision.transforms.**RandomRotation**(*degrees*, *interpolation='bilinear'*, *expand=False*, *center=None*, *fill=0*)

Rotate the image by random angle.

Parameters

- **degrees** (*number or sequence*) –
  - Range of degrees to select from.
  - If degrees is a number, the range of degrees will be (-degrees, +degrees).
  - If degrees is a sequence, the range of degrees will (degrees[0], degrees[1]).
- **interpolation** (*str*) – Interpolation method. Default is 'bilinear'. 'nearest','bilinear' are supported.
- **expand** (*boolean*) –
  - If true, expands the output to make it large enough to hold the entire rotated image.
  - If false or omitted, make the output image the same size as the input image.
  - Note that the expand flag assumes rotation around the center and no translation.
- **center** (*sequence or None*) – Optional center of rotation, (x, y). Origin is the upper left corner. Default is the center of the image.
- **fill** (*number or sequence*) – Pixel fill value for the area outside the rotated image. Default is 0.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.RandomRotation(degrees=30, interpolation=
↪'bilinear', expand=False, center=None, fill=0)
>>> image = transform(image)
>>> print(image)
```

RandomShift

class tensorlayerx.vision.transforms.**RandomShift**(*shift*, *interpolation='bilinear'*, *fill=0*)

Shift the image by random translations.

Parameters

- **shift** (*list or tuple*) – Maximum absolute fraction for horizontal and vertical translations. shift=(a, b), then horizontal shift is randomly sampled in the range -img_width * a < dx < img_width * a. vertical shift is randomly sampled in the range -img_height * b < dy < img_height * b.
- **interpolation** (*str*) – Interpolation method. Default is 'bilinear'. 'nearest','bilinear' are supported.
- **fill** (*number or sequence*) – Pixel fill value for the area outside the sheared image. Default is 0.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.RandomShift(shift=(0.2, 0.2), interpolation=
↪'bilinear', fill=0)
>>> image = transform(image)
>>> print(image)
```

RandomShear

class tensorlayerx.vision.transforms.**RandomShear**(*shear*, *interpolation='bilinear'*, *fill=0*)

Shear the image by random angle.

Parameters

- **shear** (*number or sequnence*) –
  - Range of degrees to select from.
  - If shear is a number, a shear parallel to the x axis in the range (-shear, +shear) will be applied.
  - If shear is a sequence of 2 values a shear parallel to the x axis in the range (shear[0], shear[1]) will be applied.
  - If shear is a sequence of 4 values, a x-axis shear in (shear[0], shear[1]) and y-axis shear in (shear[2], shear[3]) will be applied.
- **interpolation** (*str*) – Interpolation method. Default is 'bilinear'.'nearest','bilinear' are supported.
- **fill** (*number or sequence*) – Pixel fill value for the area outside the sheared image. Default is 0.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.RandomShear(shear=30, interpolation=
↪'bilinear', fill=0)
>>> image = transform(image)
>>> print(image)
```

RandomZoom

class tensorlayerx.vision.transforms.**RandomZoom**(*zoom*, *interpolation='bilinear'*, *fill=0*)

Zoom the image by random scale.

Parameters

- **zoom** (*list or tuple*) – Scaling factor interval, e.g (a, b), then scale is randomly sampled from the range a <= scale <= b.
- **interpolation** (*str*) – Interpolation method. Default is 'bilinear'. 'nearest','bilinear' are supported.
- **fill** (*number or sequence*) – Pixel fill value for the area outside the sheared image. Default is 0.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.RandomZoom(zoom=(0.2, 0.5), interpolation=
↪'bilinear', fill=0)
>>> image = transform(image)
>>> print(image)
```

RandomAffine

class tensorlayerx.vision.transforms.**RandomAffine**(*degrees*, *shift=None*, *zoom=None*, *shear=None*, *interpolation='bilinear'*, *fill=0*)

Random affine transformation of the image keeping center invariant.

Parameters

- **degrees** (*number or sequnence*) –
  - Range of degrees to select from.
  - If degrees is a number, the range of degrees will be (-degrees, +degrees).
  - If degrees is a sequence, the range of degrees will (degrees[0], degrees[1]).
  - Set to 0 to deactivate rotations.
- **shift** (*sequence or None*) –

- Maximum absolute fraction for horizontal and vertical translations.
- shift=(a, b), then horizontal shift is randomly sampled in the range -img_width * a < dx < img_width * a.
- vertical shift is randomly sampled in the range -img_height * b < dy < img_height * b.
- Will not shift by default.

- **shear** (*number or sequnence or None*) –
  - Range of degrees to select from.
  - If degrees is a number, a shear parallel to the x axis in the range (-shear, +shear) will be applied.
  - If shear is a sequence of 2 values a shear parallel to the x axis in the range (shear[0], shear[1]) will be applied.
  - If shear is a sequence of 4 values, a x-axis shear in (shear[0], shear[1]) and y-axis shear in (shear[2], shear[3]) will be applied.
  - Will not apply shear by default.
- **zoom** (*sequence or None*) – Scaling factor interval, e.g (a, b), then scale is randomly sampled from the range a <= scale <= b. Will not zoom by default.
- **interpolation** (*str*) – Interpolation method. Default is 'bilinear'. 'nearest','bilinear' are supported.
- **fill** (*number or sequence*) – Pixel fill value for the area outside the sheared image. Default is 0.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.RandomAffine(degrees=30, shift=(0.2,0.2),␣
↪zoom=(0.2, 0.5), shear=30, interpolation='bilinear', fill=0)
>>> image = transform(image)
>>> print(image)
```

Transpose

class tensorlayerx.vision.transforms.**Transpose**(*order*)

Transpose image(s) by swapping dimension.

Parameters **order** (*sequenece of int*) – Desired output dimension order.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.Transpose(order=(2, 0 ,1))
>>> image = transform(image)
>>> print(image)
>>> image shape : (3, 224, 224)
```

HWC2CHW

class tensorlayerx.vision.transforms.**HWC2CHW**
Transpose a image shape (H, W, C) to shape (C, H, W).

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.HWC2CHW()
>>> image = transform(image)
>>> print(image)
>>> image shape : (3, 224, 224)
```

CHW2HWC

class tensorlayerx.vision.transforms.**CHW2HWC**
Transpose a image shape (C, H, W) to shape (H, W, C).

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand(3, 224, 224) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.CHW2HWC()
>>> image = transform(image)
>>> print(image)
>>> image shape : (224, 224, 3)
```

Normalize

class tensorlayerx.vision.transforms.**Normalize**(*mean*, *std*, *data_format='HWC'*)
Normalize a tensor image with mean and standard deviation.

Parameters

- **mean** (*number or sequence*) – If mean is a number, mean will be applied for all channels. Sequence of means for each channel.
- **std** (*number or sequnece*) – If std is a number, std will be applied for all channels.Sequence of standard deviations for each channel.

- **data_format** (*str*) – Data format of input image, should be 'HWC' or 'CHW'. Default: 'HWC'.

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand( 224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.Normalize(mean = (155.0, 155.0, 155.0), std
↪= (75.0, 75.0, 75.0),data_format='HWC')
>>> image = transform(image)
>>> print(image)
```

StandardizePerImage

class tensorlayerx.vision.transforms.**StandardizePerImage**

For each 3-D image x in image, computes (x - mean) / adjusted_stddev, where mean is the average of all values in x. adjusted_stddev = max(stddev, 1.0/sqrt(N)) is capped away from 0 to protect against division by 0 when handling uniform images. N is the number of elements in x. stddev is the standard deviation of all values in x

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> image = (np.random.rand( 224, 224, 3) * 255.).astype(np.uint8)
>>> transform = tlx.vision.transforms.StandardizePerImage()
>>> image = transform(image)
>>> print(image)
```

2.8.4 Vision IO

load_image

class tensorlayerx.vision.utils.**load_image**

Load an image

Parameters

- **path** (*str*) – path of the image.
- **Returns** (*numpy.ndarray*) –
- **-------** – a numpy RGB image

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> path = './data/1.png'
>>> image = tlx.vision.load_image(path)
>>> print(image)
```

save_image

class tensorlayerx.vision.utils.**save_image**

Save an image

Parameters

- **image** (*numpy.ndarray*) – The image to save
- **file_name** (*str*) – image name to save
- **path** (*str*) – path to save image

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> load_path = './data/1.png'
>>> save_path = './test/'
>>> image = tlx.vision.load_image(path)
>>> tlx.vision.save_image(image, file_name='1.png',path=save_path)
```

load_images

class tensorlayerx.vision.utils.**load_images**

Load images from file

Parameters

- **path** (*str*) – path of the images.
- **n_threads** (*int*) – The number of threads to read image.
- **Returns** (*list*) –
- **-------** – a list of numpy RGB images

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> load_path = './data/'
>>> image = tlx.vision.load_images(path)
```

save_image

class tensorlayerx.vision.utils.**save_images**
Save images

Parameters

- **images** (*list*) – a list of numpy RGB images
- **file_names** (*list*) – a list of image names to save
- **path** (*str*) – path to save images

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> load_path = './data/'
>>> save_path = './test/'
>>> images = tlx.vision.load_images(path)
>>> name_list = user_define
>>> tlx.vision.save_images(images, file_names=name_list,path=save_path)
```

2.9 API - Initializers

To make TensorLayerX simple, TensorLayerX only warps some basic initializers. For more complex activation, TensorFlow(MindSpore, PaddlePaddle, PyTorch) API will be required.

Initializer	Initializer base class: all initializers inherit from this class.
Zeros	Initializer that generates tensors initialized to 0.
Ones	Initializer that generates tensors initialized to 1.
Constant([value])	Initializer that generates tensors initialized to a constant value.
RandomUniform([minval, maxval, seed])	Initializer that generates tensors with a uniform distribution.
RandomNormal([mean, stddev, seed])	Initializer that generates tensors with a normal distribution.
TruncatedNormal([mean, stddev, seed])	Initializer that generates a truncated normal distribution.
HeNormal([a, mode, nonlinearity, seed])	He normal initializer.
HeUniform([a, mode, nonlinearity, seed])	He uniform initializer.
deconv2d_bilinear_upsampling_initializer(shape)	Returns the initializer that can be passed to DeConv2dLayer for initializing the weights in correspondence to channel-wise bilinear up-sampling.
XavierNormal([gain, seed])	This class implements the Xavier weight initializer from the paper by Xavier Glorot and Yoshua Bengio.using a normal distribution.
XavierUniform([gain, seed])	This class implements the Xavier weight initializer from the paper by Xavier Glorot and Yoshua Bengio.using a uniform distribution.

2.9.1 Initializer

class tensorlayerx.nn.initializers.**Initializer**
Initializer base class: all initializers inherit from this class.

2.9.2 Zeros

class tensorlayerx.nn.initializers.**Zeros**
Initializer that generates tensors initialized to 0.

Examples

```
>>> import tensorlayerx as tlx
>>> init = tlx.initializers.zeros()
>>> print(init(shape=(5, 10), dtype=tlx.float32))
```

2.9.3 Ones

class tensorlayerx.nn.initializers.**Ones**
Initializer that generates tensors initialized to 1.

Examples

```
>>> import tensorlayerx as tlx
>>> init = tlx.initializers.ones()
>>> print(init(shape=(5, 10), dtype=tlx.float32))
```

2.9.4 Constant

class tensorlayerx.nn.initializers.**Constant**(*value=0*)
Initializer that generates tensors initialized to a constant value.

Parameters **value** – A python scalar or a numpy array. The assigned value.

Examples

```
>>> import tensorlayerx as tlx
>>> init = tlx.initializers.constant(value=10)
>>> print(init(shape=(5, 10), dtype=tlx.float32))
```

2.9.5 RandomUniform

class tensorlayerx.nn.initializers.**RandomUniform**(*minval=-0.05*, *maxval=0.05*, *seed=None*)
Initializer that generates tensors with a uniform distribution.

Parameters

- **minval** – A python scalar or a scalar tensor. Lower bound of the range of random values to generate.
- **maxval** – A python scalar or a scalar tensor. Upper bound of the range of random values to generate.
- **seed** – A Python integer. Used to seed the random generator.

Examples :

```
>>> import tensorlayerx as tlx
>>> init = tlx.initializers.random_uniform(minval=-0.05, maxval=0.05)
>>> print(init(shape=(5, 10), dtype=tlx.float32))
```

2.9.6 RandomNormal

class tensorlayerx.nn.initializers.**RandomNormal**(*mean=0.0*, *stddev=0.05*, *seed=None*)

Initializer that generates tensors with a normal distribution.

Parameters

- **mean** – A python scalar or a scalar tensor. Mean of the random values to generate.
- **stddev** – A python scalar or a scalar tensor. Standard deviation of the random values to generate.
- **seed** – A Python integer. Used to seed the random generator.

Examples :

```
>>> import tensorlayerx as tlx
>>> init = tlx.initializers.random_normal(mean=0.0, stddev=0.05)
>>> print(init(shape=(5, 10), dtype=tlx.float32))
```

2.9.7 TruncatedNormal

class tensorlayerx.nn.initializers.**TruncatedNormal**(*mean=0.0*, *stddev=0.05*, *seed=None*)

Initializer that generates a truncated normal distribution.

These values are similar to values from a *RandomNormal* except that values more than two standard deviations from the mean are discarded and re-drawn. This is the recommended initializer for neural network weights and filters.

Parameters

- **mean** – A python scalar or a scalar tensor. Mean of the random values to generate.
- **stddev** – A python scalar or a scalar tensor. Standard deviation of the andom values to generate.
- **seed** – A Python integer. Used to seed the random generator.

Examples

```
>>> import tensorlayerx as tlx
>>> init = tlx.initializers.truncated_normal(mean=0.0, stddev=0.05)
>>> print(init(shape=(5, 10), dtype=tlx.float32))
```

2.9.8 HeNormal

class tensorlayerx.nn.initializers.**HeNormal**(*a=0*, *mode='fan_in'*, *nonlinearity='leaky_relu'*, *seed=None*)

He normal initializer.

The resulting tensor will have values sampled from $\mathcal{N}(0, \text{std}^2)$ where

$$\text{std} = \frac{\text{gain}}{\sqrt{fan\_mode}}$$

Parameters

- **a** – int or float the negative slope of the rectifier used after this layer (only used with `'leaky_relu'`)
- **mode** – str either `'fan_in'` (default) or `'fan_out'`. Choosing `'fan_in'` preserves the magnitude of the variance of the weights in the forward pass. Choosing `'fan_out'` preserves the magnitudes in the backwards pass.
- **nonlinearity** – str the non-linear function name, recommended to use only with `'relu'` or `'leaky_relu'` (default).
- **seed** – int Used to seed the random generator.

Examples

```
>>> import tensorlayerx as tlx
>>> init = tlx.initializers.HeNormal(a=0, mode='fan_out', nonlinearity='relu')
>>> print(init(shape=(5, 10), dtype=tlx.float32))
```

2.9.9 HeUniform

class tensorlayerx.nn.initializers.**HeUniform**(*a=0*, *mode='fan_in'*, *nonlinearity='leaky_relu'*, *seed=None*)

He uniform initializer. The resulting tensor will have values sampled from $\mathcal{U}(-\text{bound}, \text{bound})$ where

$$\text{bound} = \text{gain} \times \sqrt{\frac{3}{fan\_mode}}$$

Parameters

- **a** – int or float the negative slope of the rectifier used after this layer (only used with `'leaky_relu'`)
- **mode** – str either `'fan_in'` (default) or `'fan_out'`. Choosing `'fan_in'` preserves the magnitude of the variance of the weights in the forward pass. Choosing `'fan_out'` preserves the magnitudes in the backwards pass.
- **nonlinearity** – str the non-linear function name, recommended to use only with `'relu'` or `'leaky_relu'` (default).
- **seed** – int Used to seed the random generator.

Examples

```
>>> import tensorlayerx as tlx
>>> init = tlx.initializers.HeUniform(a=0, mode='fan_in', nonlinearity='relu')
>>> print(init(shape=(5, 10), dtype=tlx.float32))
```

2.9.10 deconv2d_bilinear_upsampling_initializer

tensorlayerx.nn.initializers.**deconv2d_bilinear_upsampling_initializer**(*shape*)

Returns the initializer that can be passed to DeConv2dLayer for initializing the weights in correspondence to channel-wise bilinear up-sampling. Used in segmentation approaches such as [FCN](https://arxiv.org/abs/1605.06211)

Parameters **shape** (*tuple of int*) – The shape of the filters, [height, width, output_channels, in_channels]. It must match the shape passed to DeConv2dLayer.

Returns A constant initializer with weights set to correspond to per channel bilinear upsampling when passed as W_int in DeConv2dLayer

Return type tf.constant_initializer

2.9.11 XavierNormal

class tensorlayerx.nn.initializers.**XavierNormal**(*gain=1.0*, *seed=None*)

This class implements the Xavier weight initializer from the paper by Xavier Glorot and Yoshua Bengio.using a normal distribution.

The resulting tensor will have values sampled from $\mathcal{N}(0, \text{std}^2)$ where

$$\text{std} = \text{gain} \times \sqrt{\frac{2}{fan\_in + fan\_out}}$$

Parameters

- **gain** – float an optional scaling factor
- **seed** – int Used to seed the random generator.

2.9.12 XavierUniform

class tensorlayerx.nn.initializers.**XavierUniform**(*gain=1.0*, *seed=None*)

This class implements the Xavier weight initializer from the paper by Xavier Glorot and Yoshua Bengio.using a uniform distribution.

The resulting tensor will have values sampled from $\mathcal{U}(-a, a)$ where

$$\text{bound} = \text{gain} \times \sqrt{\frac{3}{fan\_mode}}$$

Parameters

- **gain** – float an optional scaling factor
- **seed** – int Used to seed the random generator.

2.10 API - Operations

Deep learning and Reinforcement learning library for Researchers and Engineers

get_tensor_shape(x)	Get the shape of tensor
zeros(shape[, dtype, device])	Creates a tensor with all elements set to zero.
zeros_like(x[, dtype])	This OP returns a Tensor filled with the value 0, with the same shape and data type (use dtype if dtype is not None) as x.
ones(shape[, dtype, device])	Creates a tensor with all elements set to ones.
ones_like(x[, dtype])	This OP returns a Tensor filled with the value 1, with the same shape and data type (use dtype if dtype is not None) as x.
constant(value[, dtype, shape, device])	Creates a constant tensor from a tensor-like object.
random_uniform(shape[, minval, maxval, ...])	Outputs random values from a uniform distribution.
random_normal(shape[, mean, stddev, dtype, seed])	Outputs random values from a normal distribution.
truncated_normal(shape[, mean, stddev, ...])	Outputs random values from a truncated normal distribution.
he_normal(shape[, a, mode, nonlinearity, ...])	He normal initializer.
xavier_normal(shape[, gain, dtype, seed])	Xavier normal.
xavier_uniform(shape[, gain, dtype, seed])	Xavier uniform.
Variable(initial_value, name[, trainable, ...])	Creates a new variable with value initial_value.
abs(x)	Computes the absolute value of a tensor.
acos(x)	Computes acos of x element-wise.
acosh(x)	Computes inverse hyperbolic cosine of x element-wise.
add(value, bias)	Returns x + y element-wise.
angle(x)	Returns the element-wise argument of a complex (or real) tensor.
argmax(x[, axis, dtype])	Returns the index with the largest value across axes of a tensor.
argmin(x[, axis, dtype])	Returns the index with the smallest value across axes of a tensor.
asin(x)	Returns the index with the smallest value across axes of a tensor.
asinh(x)	Computes inverse hyperbolic sine of x element-wise.
atan(x)	Computes the trignometric inverse tangent of x element-wise.
atanh(x)	Computes inverse hyperbolic tangent of x element-wise.
arange(start[, limit, delta, dtype])	Creates a sequence of numbers.
ceil(x)	Return the ceiling of x as an Integral.
cos(x)	Computes cos of x element-wise.
cosh(x)	Computes hyperbolic cosine of x element-wise.
count_nonzero(x[, axis, keepdims, dtype])	Computes number of nonzero elements across dimensions of a tensor.
cumprod(x[, axis, exclusive, reverse])	Compute the cumulative product of the tensor x along axis.
cumsum(x[, axis, exclusive, reverse])	Compute the cumulative sum of the tensor x along axis.
divide(x, y)	Computes Python style division of x by y.
equal(x, y)	Returns the truth value of (x == y) element-wise.
exp(x)	Computes exponential of x element-wise.
floor(x)	Return the floor of x as an Integral.

Continued on next page

Table 11 – continued from previous page

floordiv(x, y)	Divides x / y elementwise, rounding toward the most negative integer.
floormod(x, y)	Returns element-wise remainder of division.
greater(x, y)	Returns the truth value of (x >= y) element-wise.
greater_equal(x, y)	Returns the truth value of (x >= y) element-wise.
is_inf(x)	Returns which elements of x are Inf.
is_nan(x)	Returns which elements of x are NaN.
l2_normalize(x[, axis, eps])	Normalizes along dimension axis using an L2 norm.
less(x, y)	Returns the truth value of (x < y) element-wise.
less_equal(x, y)	Returns the truth value of (x <= y) element-wise.
log(x)	Computes natural logarithm of x element-wise.
log_sigmoid(x)	Computes log sigmoid of x element-wise.
maximum(x, y)	Returns the max of x and y (i.e.
minimum(x, y)	Returns the min of x and y (i.e.
multiply(x, y)	Returns an element-wise x * y.
negative(x)	Computes numerical negative value element-wise.
not_equal(x, y)	Returns the truth value of (x != y) element-wise.
pow(x, y)	Computes the power of one value to another.
real(x)	Computes numerical negative value element-wise.
reciprocal(x)	Computes the reciprocal of x element-wise.
reshape(tensor, shape)	Reshapes a tensor.
concat(values, axis)	Concatenates tensors along one dimension.
convert_to_tensor(value[, dtype, device])	Converts the given value to a Tensor.
convert_to_numpy(value)	Converts the given Tensor to a numpy.
reduce_max(x[, axis, keepdims])	Computes the maximum of elements across dimensions of a tensor.
reduce_mean(input_tensor[, axis, keepdims])	Computes the mean of elements across dimensions of a tensor.
reduce_min(x[, axis, keepdims])	Computes the minimum of elements across dimensions of a tensor.
reduce_prod(x[, axis, keepdims])	Computes the multiply of elements across dimensions of a tensor.
reduce_std(x[, axis, keepdims])	Computes the standard deviation of elements across dimensions of a tensor.
reduce_sum(x[, axis, keepdims])	Computes the standard deviation of elements across dimensions of a tensor.
reduce_variance(x[, axis, keepdims])	Computes the variance of elements across dimensions of a tensor.
round(x)	Rounds the values of a tensor to the nearest integer, element-wise.
rsqrt(x)	Computes reciprocal of square root of x element-wise.
segment_max(x, segment_ids)	Computes the maximum along segments of a tensor.
segment_mean(x, segment_ids)	Computes the mean along segments of a tensor.
segment_min(x, segment_ids)	Computes the minimum along segments of a tensor.
segment_prod(x, segment_ids)	Computes the product along segments of a tensor.
segment_sum(x, segment_ids)	Computes the sum along segments of a tensor.
sigmoid(x)	Computes sigmoid of x element-wise.
sign(x)	Computes sign of a tensor element-wise.
sin(x)	Computes sine of a tensor element-wise.
sinh(x)	Computes hyperbolic sine of a tensor element-wise.
softplus(x)	Computes softplus of a tensor element-wise.

Continued on next page

Table 11 – continued from previous page

sqrt(x)	Computes square root of a tensor element-wise.
square(x)	Computes square of a tensor element-wise.
squared_difference(x, y)	Computes difference and square between tensor x and tensor y.
subtract(x, y)	Returns x - y element-wise.
tan(x)	Computes tan of a tensor element-wise.
tanh(x)	Computes hyperbolic tangent of a tensor element-wise.
any(x[, axis, keepdims])	Computes logical_or of a tensor element-wise.
all(x[, axis, keepdims])	Computes logical_and of a tensor element-wise.
logical_and(x, y)	Returns the truth value of x AND y element-wise.
logical_not(x)	Returns the truth value of NOT x element-wise.
logical_or(x, y)	Returns the truth value of x OR y element-wise.
logical_xor(x, y)	Returns the truth value of NOT x element-wise.
argsort(x[, axis, descending])	Returns the indices of a tensor that give its sorted order along an axis.
bmm(x, y)	Applies batched matrix multiplication to two tensors.
matmul(a, b[, transpose_a, transpose_b])	Multiplies matrix a by matrix b, producing a * b.
triu(x[, diagonal])	This op returns the upper triangular part of a matrix (2-D tensor) or batch of matrices x, the other elements of the result tensor are set to 0.
tril(x[, diagonal])	This op returns the lower triangular part of a matrix (2-D tensor) or batch of matrices x, the other elements of the result tensor are set to 0.
tile(input, multiples)	Constructs a tensor by tiling a given tensor.
where(condition, x, y)	Return a tensor of elements selected from either x or y, depending on condition.
stack(values[, axis])	Stacks a list of rank-R tensors into one rank-(R+1) tensor.
split(value, num_or_size_splits[, axis])	Splits a tensor into sub tensors.
squeeze(x[, axis])	Removes dimensions of size 1 from the shape of a tensor.
expand_dims(input, axis)	Inserts a dimension of 1 into a tensor's shape.
gather(params, indices[, axis])	Gather slices from params axis axis according to indices.
unsorted_segment_sum(x, segment_ids, ...)	Computes the sum along segments of a tensor.
unsorted_segment_mean(x, segment_ids, ...)	Computes the mean along segments of a tensor.
unsorted_segment_min(x, segment_ids, ...)	Computes the min along segments of a tensor.
unsorted_segment_max(x, segment_ids, ...)	Computes the max along segments of a tensor.
set_seed(seed) **param seed** The random seed to set.	
is_tensor(x) **param x** A python object to check.	
tensor_scatter_nd_update(tensor, indices, ...) **param tensor** tensor to update.	
scatter_update(tensor, indices, updates)	Applies sparse updates to a variable
diag(input[, diagonal]) **param input** the input tensor.	

Continued on next page

Table 11 – continued from previous page

mask_select(x, mask[, axis])	**param x** N-D Tensor.
eye(n[, m, dtype])	**param n** the number of rows
einsum(equation, *operands)	Sums the product of the elements of the input operands along dimensions specified using a notation based on the Einstein summation convention.
set_device([device, id])	This function can specify the global device which the OP will run.
get_device()	This function can get the specified global device.
to_device(tensor[, device, id])	Returns a copy of Tensor in specified device.

2.10.1 TensorLayerX Tensor Operations

get_tensor_shape

tensorlayerx.**get_tensor_shape**(*x*)

Get the shape of tensor

Parameters **x** (*tensor*) – type float16, float32, float64, int32, complex64, complex128.

Returns

Return type list.

Examples

```
>>> import tensorlayerx as tlx
>>> x_in = tlx.layers.Input((32, 3, 3, 32))
>>> x_shape = tlx.ops.get_tensor_shape(x_in)
```

zeros

tensorlayerx.**zeros**(*shape*, *dtype='float32'*, *device=None*)

Creates a tensor with all elements set to zero.

Parameters

- **shape** (*A list of integers*) – a tuple of integers, or a 1-D Tensor of type int32.
- **dtype** (*tensor or str*) – The DType of an element in the resulting Tensor
- **device** (*str or None*) – create a tensor on 'cpu' or 'gpu', defautl is None.

Returns

Return type A Tensor with all elements set to zero.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.zeros((32, 3, 3, 32), dtype=tlx.int32)
>>> y = tlx.ops.zeros((10, 25, 25, 10), dtype='float32')
```

ones

tensorlayerx.**ones**(*shape*, *dtype='float32'*, *device=None*)

Creates a tensor with all elements set to ones.

Parameters

- **shape** (*A list of integers*) – a tuple of integers, or a 1-D Tensor of type int32.
- **dtype** (*tensor or str*) – The DType of an element in the resulting Tensor
- **device** (*str or None*) – create a tensor on 'cpu' or 'gpu', defautl is None.

Returns

Return type A Tensor with all elements set to zero.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.ones((32, 3, 3, 32), dtype=tlx.int32)
>>> y = tlx.ops.ones((10, 25, 25, 10), dtype='float32')
```

constant

tensorlayerx.**constant**(*value*, *dtype='float32'*, *shape=None*, *device=None*)

Creates a constant tensor from a tensor-like object.

Parameters

- **value** (*list*) – A constant value (or list) of output type dtype.
- **dtype** (*tensor or str*) – The type of the elements of the resulting tensor.
- **shape** (*tuple*) – Optional dimensions of resulting tensor.
- **device** (*str or None*) – create a tensor on 'cpu' or 'gpu', defautl is None.

Returns

Return type A Constant Tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(0.5, (32, 3, 3, 32), dtype=tlx.int32)
>>> y = tlx.ops.constant(0.5, (10, 25, 25, 10), dtype='float32')
```

random_uniform

tensorlayerx.**random_uniform**(*shape*, *minval=0*, *maxval=None*, *dtype='float32'*, *seed=None*)

Outputs random values from a uniform distribution.

Parameters

- **shape** (*tuple*) – A 1-D integer Tensor or Python array. The shape of the output tensor.
- **minval** (*float*) – The lower bound on the range of random values to generate (inclusive). Defaults to 0.
- **maxval** (*float*) – The upper bound on the range of random values to generate (exclusive). Defaults to 1 if dtype is floating point.
- **dtype** (*tensor or str*) – The type of the output: float16, float32, float64, int32, or int64.
- **seed** (*int*) – Used in combination with tf.random.set_seed to create a reproducible sequence of tensors across multiple calls.

Returns

Return type A tensor of the specified shape filled with random uniform values.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.random_uniform((32, 3, 3, 32), maxval=1.0, dtype=tlx.int32)
>>> y = tlx.ops.random_uniform((10, 25, 25, 10), maxval=1.0, dtype='float32')
```

random_normal

tensorlayerx.**random_normal**(*shape*, *mean=0.0*, *stddev=1.0*, *dtype='float32'*, *seed=None*)

Outputs random values from a normal distribution.

Parameters

- **shape** (*tuple*) – A 1-D integer Tensor or Python array. The shape of the output tensor.
- **mean** (*float*) – The mean of the normal distribution
- **stddev** (*float*) – The standard deviation of the normal distribution.
- **dtype** (*tensor or str*) – The type of the output.
- **seed** (*A Python integer*) – Used to create a random seed for the distribution

Returns

Return type A tensor of the specified shape filled with random normal values.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.random_normal((32, 3, 3, 32), dtype=tlx.int32)
>>> y = tlx.ops.random_normal((10, 25, 25, 10), dtype='float32')
```

truncated_normal

tensorlayerx.**truncated_normal**(*shape*, *mean=0.0*, *stddev=1.0*, *dtype='float32'*, *seed=None*)

Outputs random values from a truncated normal distribution.

Parameters

- **shape** (*tuple*) – A 1-D integer Tensor or Python array. The shape of the output tensor.
- **mean** (*float*) – The mean of the normal distribution
- **stddev** (*float*) – The standard deviation of the normal distribution.
- **dtype** (*tensor or str*) – The type of the output.
- **seed** (*A Python integer*) – Used to create a random seed for the distribution

Returns

Return type A tensor of the specified shape filled with random truncated normal values.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.truncated_normal((32, 3, 3, 32), dtype=tlx.int32)
>>> y = tlx.ops.truncated_normal((10, 25, 25, 10), dtype='float32')
```

he_normal

tensorlayerx.**he_normal**(*shape*, *a=0*, *mode='fan_in'*, *nonlinearity='leaky_relu'*, *dtype='float32'*, *seed=None*)

He normal initializer.

Parameters

- **seed** (*A Python integer.*) – Used to seed the random generator.
- **shape** (*tuple*) – A 1-D integer Tensor or Python array. The shape of the output tensor.
- **dtype** (*tensor or str*) – The type of the output.

Returns

Return type A tensor of the specified shape filled with he normal values.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.he_normal((32, 3, 3, 32), dtype=tlx.int32)
>>> y = tlx.ops.he_normal((10, 25, 25, 10), dtype='float32')
```

xavier_normal

tensorlayerx.**xavier_normal**(*shape*, *gain=1.0*, *dtype='float32'*, *seed=None*)

Xavier normal.

Parameters

- **seed** (*A Python integer.*) – Used to seed the random generator.
- **shape** (*tuple*) – A 1-D integer Tensor or Python array. The shape of the output tensor.
- **dtype** (*tensor or str*) – The type of the output.

Returns

Return type A tensor of the specified shape filled with xavier normal values.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.xavier_normal((32, 3, 3, 32), dtype=tlx.int32)
>>> y = tlx.ops.xavier_normal((10, 25, 25, 10), dtype='float32')
```

xavier_uniform

tensorlayerx.**xavier_uniform**(*shape*, *gain=1.0*, *dtype='float32'*, *seed=None*)

Xavier uniform.

Parameters

- **seed** (*A Python integer.*) – Used to seed the random generator.
- **shape** (*tuple*) – A 1-D integer Tensor or Python array. The shape of the output tensor.
- **dtype** (*tensor or str*) – The type of the output.

Returns

Return type A tensor of the specified shape filled with xavier uniform values.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.xavier_uniform((32, 3, 3, 32), dtype=tlx.int32)
>>> y = tlx.ops.xavier_uniform((10, 25, 25, 10), dtype='float32')
```

Variable

tensorlayerx.**Variable**(*initial_value*, *name*, *trainable=True*, *device=None*)

Creates a new variable with value initial_value.

Parameters

- **initial_value** (*tensor*) – A Tensor, or Python object convertible to a Tensor
- **name** (*str*) – Optional name for the variable. Defaults to 'Variable' and gets uniquified automatically.
- **device** (*str or None*) – create a tensor on 'cpu' or 'gpu', defautl is None.

Returns

Return type Variable

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.Variable(tlx.ops.ones(shape=(10, 20)), name='w')
```

abs

tensorlayerx.**abs**(*x*)

Computes the absolute value of a tensor.

Parameters **x** (*tensor*) – A Tensor or SparseTensor of type float16, float32, float64, int32, int64, complex64 or complex128.

Returns

Return type A Tensor or SparseTensor of the same size, type and sparsity as x, with absolute values.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.layers.Input((32, 3, 3, 32))
>>> y = tlx.ops.abs(x)
```

acos

tensorlayerx.**acos**(*x*)

Computes acos of x element-wise.

Parameters **x** (*tensor*) – Must be one of the following types: bfloat16, half, float32, float64, uint8, int8, int16, int32, int64, complex64, complex128, string.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.layers.Input((32, 3, 3, 32))
>>> y = tlx.ops.acos(x)
```

acosh

tensorlayerx.**acosh**(*x*)

Computes inverse hyperbolic cosine of x element-wise.

Parameters **x** (*tensor*) – A Tensor. Must be one of the following types: bfloat16, half, float32, float64, complex64, complex128.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.layers.Input((32, 3, 3, 32))
>>> y = tlx.ops.acosh(x)
```

add

tensorlayerx.**add**(*value*, *bias*)

Returns x + y element-wise.

Parameters

- **value** (*tensor.*) – Must be one of the following types: bfloat16, half, float32, float64, uint8, int8, int16, int32, int64, complex64, complex128, string.
- **bias** (*tensor*) – Must have the same type as a

Returns

Return type A Tensor. Has the same type as a.

Examples

```
>>> import tensorlayerx as tlx
>>> value = tlx.ones(shape=(10, 20))
>>> bias = tlx.ones(shape=(20))
>>> x = tlx.ops.add(value, bias)
```

angle

tensorlayerx.**angle**(*x*)

Returns the element-wise argument of a complex (or real) tensor.

Parameters **x** (*tensor*) – A Tensor. Must be one of the following types: float, double, complex64, complex128.

Returns

Return type A Tensor of type float32 or float64.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[2.15 + 3.57j, 3.89 + 6.54j])
>>> y = tlx.ops.angle(x)
```

argmax

tensorlayerx.**argmax**(*x*, *axis=None*, *dtype='int64'*)

Returns the index with the largest value across axes of a tensor.

Parameters

- **x** (*tensor*) – A Tensor
- **axis** (*int*) – An integer, the axis to reduce across. Default to 0.
- **dtype** (*tensor or str*) – An optional output dtype (nt32 or int64). Defaults to int64.

Returns

Return type A Tensor of type output_type.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[10, 20, 5, 6, 15])
>>> y = tlx.ops.argmax(x)
```

argmin

tensorlayerx.**argmin**(*x*, *axis=None*, *dtype='int64'*)

Returns the index with the smallest value across axes of a tensor.

Parameters

- **x** (*tensor*) – A Tensor
- **axis** (*int*) – An integer, the axis to reduce across. Default to 0.
- **dtype** (*tensor or str*) – An optional output dtype (nt32 or int64). Defaults to int64.

Returns

Return type A Tensor of type output_type.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[10, 20, 5, 6, 15])
>>> y = tlx.ops.argmin(x)
```

asin

tensorlayerx.**asin**(*x*)

Returns the index with the smallest value across axes of a tensor.

Parameters **x** (*tensor*) – Must be one of the following types: bfloat16, half, float32, float64, int8, int16, int32, int64, complex64, complex128.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[10, 20, 5, 6, 15])
>>> y = tlx.ops.asin(x)
```

asinh

tensorlayerx.**asinh**(*x*)

Computes inverse hyperbolic sine of x element-wise.

Parameters **x** (*tensor*) – Must be one of the following types: bfloat16, half, float32, float64, complex64, complex128.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[0.9142202  0.72091234])
>>> y = tlx.ops.asinh(x)
```

atan

tensorlayerx.**atan**(*x*)

Computes the trignometric inverse tangent of x element-wise.

Parameters **x** (*tensor*) – Must be one of the following types: bfloat16, half, float32, float64, int8, int16, int32, int64, complex64, complex128.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[0.9142202  0.72091234])
>>> y = tlx.ops.atan(x)
```

atanh

tensorlayerx.**atanh**(*x*)

Computes inverse hyperbolic tangent of x element-wise.

Parameters **x** (*tensor*) – Must be one of the following types: bfloat16, half, float32, float64, complex64, complex128.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[0.9142202  0.72091234])
>>> y = tlx.ops.atanh(x)
```

arange

tensorlayerx.**arange**(*start*, *limit=None*, *delta=1*, *dtype=None*)

Creates a sequence of numbers.

Parameters

- **start** (*tensor*) – A 0-D Tensor (scalar). Acts as first entry in the range if limit is not None; otherwise, acts as range limit and first entry defaults to 0.
- **limit** (*tensor*) – A 0-D Tensor (scalar). Upper limit of sequence, exclusive. If None, defaults to the value of start while the first entry of the range defaults to 0.
- **delta** (*tensor*) – A 0-D Tensor (scalar). Number that increments start. Defaults to 1.
- **dtype** (*type*) – The type of the elements of the resulting tensor.

Returns

Return type An 1-D Tensor of type dtype.

ceil

tensorlayerx.**ceil**(*x*)

Return the ceiling of x as an Integral. This is the smallest integer >= x.

cos

tensorlayerx.**cos**(*x*)

Computes cos of x element-wise.

Parameters **x** (*tensor*) – Must be one of the following types: bfloat16, half, float32, float64, complex64, complex128.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[0.9142202  0.72091234])
>>> y = tlx.ops.cos(x)
```

cosh

tensorlayerx.**cosh**(*x*)

Computes hyperbolic cosine of x element-wise.

Parameters **x** (*tensor*) – Must be one of the following types: bfloat16, half, float32, float64, complex64, complex128.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[0.9142202  0.72091234])
>>> y = tlx.ops.cosh(x)
```

count_nonzero

tensorlayerx.**count_nonzero**(*x*, *axis=None*, *keepdims=None*, *dtype='int64'*)

Computes number of nonzero elements across dimensions of a tensor.

Parameters

- **x** (*tensor*) – The tensor to reduce. Should be of numeric type, bool, or string.
- **axis** (*int*) – The dimensions to reduce. If None (the default), reduces all dimensions. Must be in the range [-rank(input), rank(input)).
- **keepdims** (*bool*) – If true, retains reduced dimensions with length 1.
- **dtype** (*tensor or str*) – The output dtype; defaults to tf.int64.

Returns

Return type The reduced tensor (number of nonzero values)

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=["", "a", "c", "b", " "])
>>> y = tlx.ops.count_nonzero(x)
```

cumprod

tensorlayerx.**cumprod**(*x*, *axis=0*, *exclusive=False*, *reverse=False*)

Compute the cumulative product of the tensor x along axis.

Parameters

- **x** (*tensor*) –

  Must be one of the following types: float32, float64, int64, int32, uint8, uint16, int16, int8, complex64, complex128, qint8, quint8, qint32, half.
- **axis** (*int*) – A Tensor of type int32 (default: 0). Must be in the range [-rank(x), rank(x)).
- **exclusive** (*bool*) – If True, perform exclusive cumprod.
- **reverse** (*bool*) – A bool (default: False).

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[3, 2, 1])
>>> y = tlx.ops.cumprod(x)
>>> y = tlx.ops.cumprod(x, exclusive=True, reverse=True)
```

cumsum

tensorlayerx.**cumsum**(*x*, *axis=0*, *exclusive=False*, *reverse=False*)

Compute the cumulative sum of the tensor x along axis.

Parameters

- **x** (*tensor*) –

  Must be one of the following types: float32, float64, int64, int32, uint8, uint16, int16, int8, complex64, complex128, qint8, quint8, qint32, half.

- **axis** (*int*) – A Tensor of type int32 (default: 0). Must be in the range [-rank(x), rank(x)).
- **exclusive** (*bool*) – If True, perform exclusive cumprod.
- **reverse** (*bool*) – A bool (default: False).

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[1, 2, 3])
>>> y = tlx.ops.cumsum(x)
>>> y = tlx.ops.cumsum(x, exclusive=True, reverse=True)
```

divide

tensorlayerx.**divide**(*x*, *y*)

Computes Python style division of x by y.

Parameters

- **x** (*tensor*) – A Tensor
- **y** (*tensor*) – A Tensor

Returns

Return type A Tensor with same shape as input

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[0.9142202  0.72091234])
>>> y = tlx.ops.divide(x, x)
```

equal

tensorlayerx.**equal**(*x*, *y*)

Returns the truth value of (x == y) element-wise.

Parameters

- **x** (*tensor*) – A Tensor or SparseTensor or IndexedSlices.
- **y** (*tensor*) – A Tensor or SparseTensor or IndexedSlices.

Returns

Return type A Tensor of type bool with the same size as that of x or y.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[1, 2, 3])
>>> y = tlx.ops.equal(x, x)
```

exp

tensorlayerx.**exp**(*x*)

Computes exponential of x element-wise.

Parameters **x** (*tensor*) – Must be one of the following types: bfloat16, half, float32, float64, complex64, complex128.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[1, 2, 3])
>>> y = tlx.ops.exp(x)
```

floor

tensorlayerx.**floor**(*x*)

Return the floor of x as an Integral. This is the largest integer <= x.

floordiv

tensorlayerx.**floordiv**(*x*, *y*)

Divides x / y elementwise, rounding toward the most negative integer.

Parameters

- **x** (*tensor*) – Tensor numerator of real numeric type.
- **y** (*tensor*) – Tensor denominator of real numeric type.

Returns

Return type x / y rounded toward -infinity.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[1, 2, 3])
>>> y = tlx.ops.floordiv(x, x)
```

floormod

tensorlayerx.**floormod**(*x*, *y*)

Returns element-wise remainder of division. When x < 0 xor y < 0 is true, this follows Python semantics in that the result here is consistent with a flooring divide. E.g. floor(x / y) * y + mod(x, y) = x.

Parameters

- **x** (*tensor*) – Must be one of the following types: int8, int16, int32, int64, uint8, uint16, uint32, uint64, bfloat16, half, float32, float64.
- **y** (*tensor*) – A Tensor. Must have the same type as x.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[1, 2, 3])
>>> y = tlx.ops.floormod(x, x)
```

greater

tensorlayerx.**greater**(*x*, *y*)

Returns the truth value of (x >= y) element-wise.

Parameters

- **x** (*tensor*) – Must be one of the following types: float32, float64, int32, uint8, int16, int8, int64, bfloat16, uint16, half, uint32, uint64.
- **y** (*tensor*) – A Tensor. Must have the same type as x.

Returns

Return type A Tensor of type bool.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[1, 2, 3])
>>> y = tlx.ops.greater(x, x)
```

greater_equal

tensorlayerx.**greater_equal**(*x*, *y*)

Returns the truth value of (x >= y) element-wise.

Parameters

- **x** (*tensor*) – Must be one of the following types: float32, float64, int32, uint8, int16, int8, int64, bfloat16, uint16, half, uint32, uint64.
- **y** (*tensor*) – A Tensor. Must have the same type as x.

Returns

Return type A Tensor of type bool.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[1, 2, 3])
>>> y = tlx.ops.greater_equal(x, x)
```

is_inf

tensorlayerx.**is_inf**(*x*)

Returns which elements of x are Inf.

Parameters **x** (*tensor*) – A Tensor. Must be one of the following types: bfloat16, half, float32, float64.

Returns

Return type A Tensor of type bool.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.constant(value=[1, 2, 3, np.inf])
>>> y = tlx.ops.is_inf(x)
```

is_nan

tensorlayerx.**is_nan**(*x*)

Returns which elements of x are NaN.

Parameters **x** (*tensor*) – A Tensor. Must be one of the following types: bfloat16, half, float32, float64.

Returns

Return type A Tensor of type bool.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.constant(value=[1, 2, 3, np.nan])
>>> y = tlx.ops.is_nan(x)
```

l2_normalize

tensorlayerx.**l2_normalize**(*x*, *axis=None*, *eps=1e-12*)

Normalizes along dimension axis using an L2 norm. For a 1-D tensor with axis = 0, computes output = x / sqrt(max(sum(x**2), epsilon))

Parameters

- **x** (*tensor*) – A Tensor
- **axis** (*int*) – Dimension along which to normalize. A scalar or a vector of integers.
- **eps** (*float*) – A lower bound value for the norm. Will use sqrt(epsilon) as the divisor if norm < sqrt(epsilon).

Returns

Return type A Tensor with the same shape as x.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.constant(value=[1, 2, 3, np.nan])
>>> y = tlx.ops.l2_normalize(x)
```

less

tensorlayerx.**less**(*x*, *y*)

Returns the truth value of (x < y) element-wise.

Parameters

- **x** (*tensor*) – Must be one of the following types: float32, float64, int32, uint8, int16, int8, int64, bfloat16, uint16, half, uint32, uint64.
- **y** (*tensor*) – A Tensor. Must have the same type as x.

Returns

Return type A Tensor of type bool.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[1, 2, 3])
>>> y = tlx.ops.less(x, x)
```

less_equal

tensorlayerx.**less_equal**(*x*, *y*)

Returns the truth value of (x <= y) element-wise.

Parameters

- **x** (*tensor*) – Must be one of the following types: float32, float64, int32, uint8, int16, int8, int64, bfloat16, uint16, half, uint32, uint64.
- **y** (*tensor*) – A Tensor. Must have the same type as x.

Returns

Return type A Tensor of type bool.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[1, 2, 3])
>>> y = tlx.ops.less_equal(x, x)
```

log

tensorlayerx.**log**(*x*)

Computes natural logarithm of x element-wise.

Parameters **x** (*tensor*) – Must be one of the following types: bfloat16, half, float32, float64, complex64, complex128.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[1, 2, 3])
>>> y = tlx.ops.log(x)
```

log_sigmoid

tensorlayerx.**log_sigmoid**(*x*)

Computes log sigmoid of x element-wise.

Parameters **x** (*tensor*) – A Tensor with type float32 or float64.

Returns

Return type A Tensor with the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[1, 2, 3])
>>> y = tlx.ops.log_sigmoid(x)
```

maximum

tensorlayerx.**maximum**(*x*, *y*)

Returns the max of x and y (i.e. x > y ? x : y) element-wise.

Parameters

- **x** (*tensor*) – Must be one of the following types: float32, float64, int32, uint8, int16, int8, int64, bfloat16, uint16, half, uint32, uint64.
- **y** (*tensor*) – A Tensor. Must have the same type as x.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[1, 2, 3])
>>> y = tlx.ops.maximum(x, x)
```

minimum

tensorlayerx.**minimum**(*x*, *y*)

Returns the min of x and y (i.e. x < y ? x : y) element-wise.

Parameters

- **x** (*tensor.*) – Must be one of the following types: bfloat16, half, float32, float64, int32, int64.
- **y** (*A Tensor.*) – Must have the same type as x.

Returns

Return type A Tensor. Has the same type as x

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant([0., 0., 0., 0.])
>>> y = tlx.ops.constant([-5., -2., 0., 3.])
>>> z = tlx.ops.minimum(x, y)
```

multiply

tensorlayerx.**multiply**(*x*, *y*)

Returns an element-wise x * y.

Parameters

- **x** (*tensor*) – Must be one of the following types: bfloat16, half, float32, float64, uint8, int8, uint16, int16, int32, int64, complex64, complex128.
- **y** (*tensor*) – A Tensor. Must have the same type as x.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[0.9142202  0.72091234])
>>> y = tlx.ops.multiply(x, x)
```

negative

tensorlayerx.**negative**(*x*)

Computes numerical negative value element-wise.

Parameters **x** (*tensor*) – Must be one of the following types: bfloat16, half, float32, float64, int8, int16, int32, int64, complex64, complex128.

Returns

- *A Tensor. Has the same type as x.*
- *If x is a SparseTensor, returns SparseTensor(x.indices, tf.math.negative(x.values, ...), x.dense_shape)*

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[1, 2, 3])
>>> y = tlx.ops.negative(x)
```

not_equal

tensorlayerx.**not_equal**(*x*, *y*)

Returns the truth value of (x != y) element-wise.

Parameters

- **x** (*tensor*) – A Tensor or SparseTensor or IndexedSlices.
- **y** (*tensor*) – A Tensor or SparseTensor or IndexedSlices.

Returns

Return type A Tensor of type bool with the same size as that of x or y.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[1, 2, 3])
>>> y = tlx.ops.constant(value=[1, 3, 5])
>>> x = tlx.ops.not_equal(x, y)
```

pow

tensorlayerx.**pow**(*x*, *y*)

Computes the power of one value to another.

Parameters

- **x** (*tensor*) – A Tensor of type float16, float32, float64, int32, int64, complex64, or complex128.
- **y** (*tensor*) – A Tensor of type float16, float32, float64, int32, int64, complex64, or complex128.

Returns

Return type A Tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[1, 2, 3])
>>> y = tlx.ops.constant(value=[1, 3, 5])
>>> x = tlx.ops.pow(x, y)
```

real

tensorlayerx.**real**(*x*)

Computes numerical negative value element-wise.

Parameters **x** (*tensor*) – A Tensor. Must have numeric type.

Returns

Return type A Tensor of type float32 or float64.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[-2.25 + 4.75j, 3.25 + 5.75j])
>>> y = tlx.ops.real(x)
```

reciprocal

tensorlayerx.**reciprocal**(*x*)

Computes the reciprocal of x element-wise.

Parameters **x** (*tensor*) – A Tensor. Must be one of the following types: bfloat16, half, float32, float64, int8, int16, int32, int64, complex64, complex128.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant(value=[-2.25, 3.25])
>>> y = tlx.ops.reciprocal(x)
```

reshape

tensorlayerx.**reshape**(*tensor*, *shape*)

Reshapes a tensor.

Parameters

- **tensor** (*tensor*) – A Tensor.
- **shape** (*tensor*) – Defines the shape of the output tensor.

Returns

Return type A Tensor. Has the same type as tensor

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant([0., 1., 2., 3.])
>>> z = tlx.ops.reshape(x, [2, 2])
```

concat

tensorlayerx.**concat**(*values*, *axis*)

Concatenates tensors along one dimension.

Parameters

- **values** (*list*) – A list of Tensor objects or a single Tensor
- **axis** (*int*) – 0-D int32 Tensor. Dimension along which to concatenate

Returns

Return type A Tensor resulting from concatenation of the input tensors.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant([0., 0., 0., 0.])
>>> y = tlx.ops.constant([-5., -2., 0., 3.])
>>> z = tlx.ops.concat([x, y], 0)
```

convert_to_tensor

tensorlayerx.**convert_to_tensor**(*value*, *dtype=None*, *device=None*)

Converts the given value to a Tensor.

Parameters

- **value** (*object*) – An object whose type has a registered Tensor conversion function.
- **dtype** (*optional*) – Optional element type for the returned tensor. If missing, the type is inferred from the type of value.
- **device** (*str or None*) – create a tensor on 'cpu' or 'gpu', defautl is None.

Returns

Return type A Tensor based on value.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = np.ones(shape=(10, 10))
>>> y = tlx.ops.convert_to_tensor(x)
```

convert_to_numpy

tensorlayerx.**convert_to_numpy**(*value*)

Converts the given Tensor to a numpy.

Parameters **value** (*object*) – An object whose type has a registered Tensor conversion function.

Returns

Return type A value based on tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.ones(shape=(10, 10))
>>> y = tlx.ops.convert_to_numpy(x)
```

reduce_max

tensorlayerx.**reduce_max**(*x*, *axis=None*, *keepdims=False*)

Computes the maximum of elements across dimensions of a tensor.

Parameters

- **x** (*tensor*) – The tensor to reduce. Should have real numeric type.
- **axis** (*int*) – The dimensions to reduce. If None (the default), reduces all dimensions. Must be in the range [-rank(x), rank(x)).
- **keepdims** (*boolean*) – If true, keep these reduced dimensions and the length is 1. If false, don't keep these dimensions. Default : False, don't keep these reduced dimensions.

Returns

Return type The reduced tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.random.randn(3, 4))
>>> x1 = tlx.ops.reduce_max(x, axis=1, keepdims=False)
>>> x2 = tlx.ops.reduce_max(x, axis=1, keepdims=True)
```

reduce_mean

tensorlayerx.**reduce_mean**(*input_tensor*, *axis=None*, *keepdims=False*)

Computes the mean of elements across dimensions of a tensor.

Parameters

- **x** (*tensor*) – The tensor to reduce. Should have numeric type.
- **axis** (*list*) – The dimensions to reduce. If None (the default), reduces all dimensions. Must be in the range [-rank(x), rank(x)).
- **keepdims** (*boolean*) – If true, keep these reduced dimensions and the length is 1. If false, don't keep these dimensions. Default : False, don't keep these reduced dimensions.

Returns

Return type The reduced tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.random.randn(3, 4))
>>> x1 = tlx.ops.reduce_mean(x, axis=1, keepdims=False)
>>> x2 = tlx.ops.reduce_mean(x, axis=1, keepdims=True)
```

reduce_min

tensorlayerx.**reduce_min**(*x*, *axis=None*, *keepdims=False*)

Computes the minimum of elements across dimensions of a tensor.

Parameters

- **x** (*tensor*) – The tensor to reduce. Should have real numeric type.
- **axis** (*int*) – The dimensions to reduce. If None (the default), reduces all dimensions. Must be in the range [-rank(x), rank(x)).
- **keepdims** (*boolean*) – If true, keep these reduced dimensions and the length is 1. If false, don't keep these dimensions. Default : False, don't keep these reduced dimensions.

Returns

Return type The reduced tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.random.randn(3, 4))
>>> x1 = tlx.ops.reduce_min(x, axis=1, keepdims=False)
>>> x2 = tlx.ops.reduce_min(x, axis=1, keepdims=True)
```

reduce_prod

tensorlayerx.**reduce_prod**(*x*, *axis=None*, *keepdims=False*)

Computes the multiply of elements across dimensions of a tensor.

Parameters

- **x** (*tensor*) – The tensor to reduce. Should have real numeric type.
- **axis** (*int*) – The dimensions to reduce. If None (the default), reduces all dimensions. Must be in the range [-rank(x), rank(x)).
- **keepdims** (*boolean*) – If true, keep these reduced dimensions and the length is 1. If false, don't keep these dimensions. Default : False, don't keep these reduced dimensions.

Returns

Return type The reduced tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.random.randn(3, 4))
>>> x1 = tlx.ops.reduce_prod(x, axis=1, keepdims=False)
>>> x2 = tlx.ops.reduce_prod(x, axis=1, keepdims=True)
```

reduce_std

tensorlayerx.**reduce_std**(*x*, *axis=None*, *keepdims=False*)

Computes the standard deviation of elements across dimensions of a tensor.

Parameters

- **x** (*tensor*) – The tensor to reduce. Should have real numeric type.
- **axis** (*int*) – The dimensions to reduce. If None (the default), reduces all dimensions. Must be in the range [-rank(x), rank(x)).
- **keepdims** (*boolean*) – If true, keep these reduced dimensions and the length is 1. If false, don't keep these dimensions. Default : False, don't keep these reduced dimensions.

Returns

Return type The reduced tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.random.randn(3, 4))
>>> x1 = tlx.ops.reduce_std(x, axis=1, keepdims=False)
>>> x2 = tlx.ops.reduce_std(x, axis=1, keepdims=True)
```

reduce_sum

tensorlayerx.**reduce_sum**(*x*, *axis=None*, *keepdims=False*)

Computes the standard deviation of elements across dimensions of a tensor.

Parameters

- **x** (*tensor*) – The tensor to reduce. Should have real numeric type.
- **axis** (*int*) – The dimensions to reduce. If None (the default), reduces all dimensions. Must be in the range [-rank(x), rank(x)).
- **keepdims** (*boolean*) – If true, keep these reduced dimensions and the length is 1. If false, don't keep these dimensions. Default : False, don't keep these reduced dimensions.

Returns

Return type The reduced tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.random.randn(3, 4))
>>> x1 = tlx.ops.reduce_sum(x, axis=1, keepdims=False)
>>> x2 = tlx.ops.reduce_sum(x, axis=1, keepdims=True)
```

reduce_variance

tensorlayerx.**reduce_variance**(*x*, *axis=None*, *keepdims=False*)

Computes the variance of elements across dimensions of a tensor.

Parameters

- **x** (*tensor*) – The tensor to reduce. Should have real numeric type.
- **axis** (*int*) – The dimensions to reduce. If None (the default), reduces all dimensions. Must be in the range [-rank(x), rank(x)).
- **keepdims** (*boolean*) – If true, keep these reduced dimensions and the length is 1. If false, don't keep these dimensions. Default : False, don't keep these reduced dimensions.

Returns

Return type The reduced tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.random.randn(3, 4))
>>> x1 = tlx.ops.reduce_variance(x, axis=1, keepdims=False)
>>> x2 = tlx.ops.reduce_variance(x, axis=1, keepdims=True)
```

round

tensorlayerx.**round**(*x*)

Rounds the values of a tensor to the nearest integer, element-wise.

Parameters **x** (*tensor*) – The tensor to round. Should have real numeric type.

Returns

Return type A Tensor of same shape and type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([0.9, 2.5, 2.3, 1.5, -4.5]))
>>> x = tlx.ops.round(x)
```

rsqrt

tensorlayerx.**rsqrt**(*x*)

Computes reciprocal of square root of x element-wise.

Parameters **x** (*tensor*) – The tensor to rsqrt. Should have real numeric type.

Returns

Return type A Tensor of same shape and type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([0.9, 2.5, 2.3, 1.5]))
>>> x = tlx.ops.rsqrt(x)
```

segment_max

tensorlayerx.**segment_max**(*x*, *segment_ids*)

Computes the maximum along segments of a tensor.

Parameters

- **x** (*tensor*) – The tensor to segment_max. Should have real numeric type.

- **segment_ids** (*tensor*) – A 1-D tensor whose size is equal to the size of data's first dimension. Values should be sorted and can be repeated.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([[1,2,3,4], [4, 3, 2, 1], [5,6,7,8]]))
>>> id = tlx.ops.convert_to_tensor([0, 0, 1])
>>> x = tlx.ops.segment_max(x, id)
>>> print(x)
>>> [[4, 3, 3, 4],
>>> [5, 6, 7, 8]]
```

segment_mean

tensorlayerx.**segment_mean**(*x*, *segment_ids*)

Computes the mean along segments of a tensor.

Parameters

- **x** (*tensor*) – The tensor to segment_mean. Should have real numeric type.
- **segment_ids** (*tensor*) – A 1-D tensor whose size is equal to the size of data's first dimension. Values should be sorted and can be repeated.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([[1.0 , 2, 3, 4], [4, 3, 2, 1], [5, 6,␣
↪7, 8]]))
>>> id = tlx.ops.convert_to_tensor([0, 0, 1])
>>> x = tlx.ops.segment_mean(x, id)
>>> print(x)
>>> [[2.5, 2.5, 2.5, 2.5],
>>> [5, 6, 7, 8]]
```

segment_min

tensorlayerx.**segment_min**(*x*, *segment_ids*)

Computes the minimum along segments of a tensor.

Parameters

- **x** (*tensor*) – The tensor to segment_minimum. Should have real numeric type.

- **segment_ids** (*tensor*) – A 1-D tensor whose size is equal to the size of data's first dimension. Values should be sorted and can be repeated.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([[1, 2, 3, 4], [4, 3, 2, 1], [5, 6, 7,␣
↪8]]))
>>> id = tlx.ops.convert_to_tensor([0, 0, 1])
>>> x = tlx.ops.segment_minimum(x, id)
>>> print(x)
>>> [[1, 2, 2, 1],
>>> [5, 6, 7, 8]]
```

segment_prod

tensorlayerx.**segment_prod**(*x*, *segment_ids*)

Computes the product along segments of a tensor.

Parameters

- **x** (*tensor*) – The tensor to segment_prod. Should have real numeric type.
- **segment_ids** (*tensor*) – A 1-D tensor whose size is equal to the size of data's first dimension. Values should be sorted and can be repeated.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([[1, 2, 3, 4], [4, 3, 2, 1], [5, 6, 7,␣
↪8]]))
>>> id = tlx.ops.convert_to_tensor([0, 0, 1])
>>> x = tlx.ops.segment_prod(x, id)
>>> print(x)
>>> [[4, 6, 6, 4],
>>> [5, 6, 7, 8]]
```

segment_sum

tensorlayerx.**segment_sum**(*x*, *segment_ids*)

Computes the sum along segments of a tensor.

Parameters

- **x** (*tensor*) – The tensor to segment_sum. Should have real numeric type.

- **segment_ids** (*tensor*) – A 1-D tensor whose size is equal to the size of data's first dimension. Values should be sorted and can be repeated.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([[1, 2, 3, 4], [4, 3, 2, 1], [5, 6, 7,␣
↪8]]))
>>> id = tlx.ops.convert_to_tensor([0, 0, 1])
>>> x = tlx.ops.segment_sum(x, id)
>>> print(x)
>>> [[5, 5, 5, 5],
>>> [5, 6, 7, 8]]
```

sigmoid

tensorlayerx.**sigmoid**(*x*)

Computes sigmoid of x element-wise.

Parameters **x** (*tensor*) – A Tensor with type float16, float32, float64, complex64, or complex128.

Returns

Return type A Tensor with the same type as x.

sign

tensorlayerx.**sign**(*x*)

Computes sign of a tensor element-wise.

Parameters **x** (*tensor*) – The tensor to sign. y = sign(x) = -1 if x < 0; 0 if x == 0; 1 if x > 0.

Returns

Return type A Tensor with the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([-128.0, 0.0, 128.0]), dtype='float32')
>>> x = tlx.ops.sign(x)
>>> print(x)
>>> [-1., 0., 1.]
```

sin

tensorlayerx.**sin**(*x*)

Computes sine of a tensor element-wise.

Parameters **x** (*tensor*) – The tensor to sin. Input range is (-inf, inf) and output range is [-1,1].

Returns

Return type A Tensor with the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([-1.0, 0.0, 1.0]), dtype='float32')
>>> x = tlx.ops.sin(x)
>>> print(x)
>>> [-0.84147096, 0., 0.84147096]
```

sinh

tensorlayerx.**sinh**(*x*)

Computes hyperbolic sine of a tensor element-wise.

Parameters **x** (*tensor*) – The tensor to hyperbolic sin. Input range is (-inf, inf) and output range is [-inf,inf].

Returns

Return type A Tensor with the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([-1.0, 0.0, 1.0]), dtype='float32')
>>> x = tlx.ops.sinh(x)
>>> print(x)
>>> [-1.1752012, 0., 1.1752012]
```

softplus

tensorlayerx.**softplus**(*x*)

Computes softplus of a tensor element-wise.

Parameters **x** (*tensor*) – The tensor to softplus. softplus(x) = log(exp(x) + 1).

Returns

Return type A Tensor with the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([-1.0, 0.0, 1.0]), dtype='float32')
>>> x = tlx.ops.softplus(x)
```

(continues on next page)

(continued from previous page)

```
>>> print(x)
>>> [0.3132617, 0.6931472, 1.3132616]
```

sqrt

tensorlayerx.**sqrt**(*x*)

Computes square root of a tensor element-wise.

Parameters **x** (*tensor*) – Must be one of the following types: bfloat16, half, float32, float64, complex64, complex128.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([0.0, 1.0, 4.0]), dtype=tlx.float32)
>>> x = tlx.ops.sqrt(x)
>>> print(x)
>>> [0.0, 1.0, 2.0]
```

square

tensorlayerx.**square**(*x*)

Computes square of a tensor element-wise.

Parameters **x** (*tensor*) – The tensor to square.

Returns

Return type A Tensor with the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([-1.0, 0.0, 1.0]), dtype='float32')
>>> x = tlx.ops.square(x)
>>> print(x)
>>> [1.0, 0.0, 1.0]
```

squared_difference

tensorlayerx.**squared_difference**(*x*, *y*)

Computes difference and square between tensor x and tensor y. return square(x - y)

Parameters

- **x** (*tensor*) – A Tensor.

- **y** (*tensor*) – A Tensor. Must have the same type as x.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([[1,0,1], [2,3,4]]), dtype='float32')
>>> y = tlx.ops.convert_to_tensor(np.array([[-1,0,1], [2,3,4]]), dtype='float32')
>>> res = tlx.ops.squared_difference(x, y)
>>> print(res)
>>> [[4.0, 0.0, 0.0],
>>> [0.0, 0.0, 0.0]]
```

subtract

tensorlayerx.**subtract**(*x*, *y*)

Returns x - y element-wise.

Parameters

- **x** (*tensor*) – A tensor.
- **y** (*tensor*) – A Tensor. Must have the same type as x.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([[1,0,1], [2,3,4]]), dtype='float32')
>>> y = tlx.ops.convert_to_tensor(np.array([[-1,0,1], [2,3,4]]), dtype='float32')
>>> res = tlx.ops.subtract(x, y)
>>> print(res)
>>> [[-2.0, 0.0, 0.0],
>>> [0.0, 0.0, 0.0]]
```

tan

tensorlayerx.**tan**(*x*)

Computes tan of a tensor element-wise.

Parameters **x** (*tensor*) – The tensor to tan.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([1,0,1]), dtype='float32')
>>> res = tlx.ops.tan(x)
>>> print(res)
>>> [-1.5574077, 0.0, 1.5574077]
```

tanh

tensorlayerx.**tanh**(*x*)

Computes hyperbolic tangent of a tensor element-wise.

Parameters **x** (*tensor*) – The tensor to tanh.

Returns

Return type A Tensor. Has the same type as x.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([1,0,1]), dtype="float32")
>>> res = tlx.ops.tanh(x)
>>> print(res)
>>> [-0.7615942, 0.0, 0.7615942]
```

any

tensorlayerx.**any**(*x*, *axis=None*, *keepdims=False*)

Computes logical_or of a tensor element-wise.

Parameters

- **x** (*tensor*) – The boolean tensor to reduce.
- **axis** (*int or None*) – The dimensions to reduce. If None (the default), reduces all dimensions. Must be in the range [-rank(x),rank(x)).
- **keepdims** (*boolean*) – If true, retains reduced dimensions with length 1.

Returns

Return type The reduced tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([1,0,1]), dtype='bool')
>>> res = tlx.ops.any(x, axis = None, keepdims = False)
>>> print(res)
>>> True
```

all

tensorlayerx.**all**(*x*, *axis=None*, *keepdims=False*)

Computes logical_and of a tensor element-wise.

Parameters

- **x** (*tensor*) – The boolean tensor to reduce.
- **axis** (*int or None*) – The dimensions to reduce. If None (the default), reduces all dimensions. Must be in the range [-rank(x),rank(x)).
- **keepdims** (*boolean*) – If true, retains reduced dimensions with length 1.

Returns

Return type The reduced tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.ops.convert_to_tensor(np.array([1,0,1]), dtype='bool')
>>> res = tlx.ops.all(x, axis = None, keepdims = False)
>>> print(res)
>>> False
```

logical_and

tensorlayerx.**logical_and**(*x*, *y*)

Returns the truth value of x AND y element-wise.

Parameters

- **x** (*tensor*) – A tf.Tensor of type bool.
- **y** (*tensor*) – A tf.Tensor of type bool.

Returns

Return type A Tensor of type bool.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.constant([False, False, True, True])
>>> y = tlx.constant([False, True, False, True])
>>> res = tlx.ops.logical_and(x, y)
>>> print(res)
>>> [False, False, False, True]
```

logical_not

tensorlayerx.**logical_not**(*x*)

Returns the truth value of NOT x element-wise.

Parameters **x** (*tensor*) – A tf.Tensor of type bool.

Returns

Return type A Tensor of type bool.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.constant([False, False, True, True])
>>> res = tlx.ops.logical_not(x, y)
>>> print(res)
>>> [True, True, False, False]
```

logical_or

tensorlayerx.**logical_or**(*x*, *y*)

Returns the truth value of x OR y element-wise.

Parameters

- **x** (*tensor*) – A tf.Tensor of type bool.
- **y** (*tensor*) – A tf.Tensor of type bool.

Returns

Return type A Tensor of type bool.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.constant([False, False, True, True])
>>> y = tlx.constant([False, True, False, True])
>>> res = tlx.ops.logical_or(x, y)
>>> print(res)
>>> [False, True, True, True]
```

logical_xor

tensorlayerx.**logical_xor**(*x*, *y*)

Returns the truth value of NOT x element-wise. x ^ y = (x | y) & ~(x & y)

Parameters **x** (*tensor*) – A tf.Tensor of type bool.

Returns

Return type A Tensor of type bool.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.constant([False, False, True, True])
>>> y = tlx.constant([False, True, False, True])
>>> res = tlx.ops.logical_xor(x, y)
```

(continues on next page)

(continued from previous page)

```
>>> print(res)
>>> [False, True, True, False]
```

argsort

tensorlayerx.**argsort**(*x*, *axis=-1*, *descending=False*)

Returns the indices of a tensor that give its sorted order along an axis.

Parameters

- **x** (*tensor*) – An input N-D Tensor
- **axis** (*int or None*) – The axis along which to sort. The default is -1, which sorts the last axis.
- **descending** (*boolean*) – Descending is a flag, if set to true, algorithm will sort by descending order, else sort by ascending order. Default is false.

Returns

Return type A Tensor with the same shape as values.

Examples

```
>>> import tensorlayerx as tlx
>>> x = [1, 10, 26.9, 2.8, 166.32, 62.3]
>>> y = tlx.ops.argsort(x, descending = False)
>>> print(y)
>>> [0, 3, 1, 2, 5, 4]
```

bmm

tensorlayerx.**bmm**(*x*, *y*)

Applies batched matrix multiplication to two tensors. Both of the two input tensors must be three-dementional and share the same batch size. if x is a (b, m, k) tensor, y is a (b, k, n) tensor, the output will be a (b, m, n) tensor.

Parameters

- **x** (*tensor*) – The input Tensor.
- **y** (*tensor*) – The input Tensor.

Returns

Return type The product Tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.convert_to_tensor([[[1.0, 1.0, 1.0],[2.0, 2.0, 2.0]],[[3.0, 3.0, 3.0],
↪[4.0, 4.0, 4.0]]])
>>> y = tlx.convert_to_tensor([[[1.0, 1.0],[2.0, 2.0],[3.0, 3.0]],[[4.0, 4.0],[5.
↪0, 5.0],[6.0, 6.0]]])
>>> res = tlx.ops.bmm(x, y)
```

(continues on next page)

(continued from previous page)

```
>>> print(res)
>>> [[[6. , 6. ],
>>> [12., 12.]],
>>> [[45., 45.],
>>> [60., 60.]]]
```

matmul

tensorlayerx.**matmul**(*a*, *b*, *transpose_a=False*, *transpose_b=False*)

Multiplies matrix a by matrix b, producing a * b.

Parameters

- **a** (*tensor*) – type float16, float32, float64, int32, complex64, complex128 and rank > 1.
- **b** (*tensor*) – with same type and rank as a.
- **transpose_a** (*boolean*) – If True, a is transposed before multiplication.
- **transpose_b** (*boolean*) – If True, b is transposed before multiplication.

Returns

Return type A Tensor of the same type as a and b

Examples

```
>>> import tensorlayerx as tlx
>>> import numpy as np
>>> x = tlx.convert_to_tensor(np.random.random([2,3,2]), dtype="float32")
>>> y = tlx.convert_to_tensor(np.random.random([2,2,3]), dtype="float32")
>>> z = tlx.ops.matmul(x, y)
>>> print(z.shape)
>>> [2,3,3]
```

triu

tensorlayerx.**triu**(*x*, *diagonal=0*)

This op returns the upper triangular part of a matrix (2-D tensor) or batch of matrices x, the other elements of the result tensor are set to 0. The upper triangular part of the matrix is defined as the elements on and above the diagonal.

Parameters

- **x** (*tensor*) – The tensor to triu.
- **diagonal** (*int*) – The diagonal to consider, default value is 0. If diagonal = 0, all elements on and above the main diagonal are retained. A positive value excludes just as many diagonals above the main diagonal, and similarly a negative value includes just as many diagonals below the main diagonal.

Returns

Return type Results of upper triangular operation by the specified diagonal of input tensor x, it's data type is the same as x's Tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.convert_to_tensor(np.arange(1, 10, dtype="int32").reshape(3,-1))
>>> y = tlx.ops.triu(x, diagonal=1)
>>> print(y)
>>> [[0, 2, 3],
>>> [ 0, 0, 6],
>>> [ 0, 0, 0]]
```

tril

tensorlayerx.**tril**(*x*, *diagonal=0*)

This op returns the lower triangular part of a matrix (2-D tensor) or batch of matrices x, the other elements of the result tensor are set to 0. The lower triangular part of the matrix is defined as the elements on and below the diagonal.

Parameters

- **x** (*tensor*) – The tensor to tril.
- **diagonal** (*int*) – The diagonal to consider, default value is 0. If diagonal = 0, all elements on and below the main diagonal are retained. A positive value includes just as many diagonals above the main diagonal, and similarly a negative value excludes just as many diagonals below the main diagonal.

Returns

Return type Results of lower triangular operation by the specified diagonal of input tensor x, it's data type is the same as x's Tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.convert_to_tensor(np.arange(1, 10, dtype="int32").reshape(3,-1))
>>> y = tlx.ops.tril(x, diagonal=1)
>>> print(y)
>>> [[0, 0, 0],
>>> [ 4, 0, 0],
>>> [ 7, 8, 0]]
```

tile

tensorlayerx.**tile**(*input*, *multiples*)

Constructs a tensor by tiling a given tensor.

Parameters

- **input** (*tensor*) – A Tensor. 1-D or higher.
- **multiples** (*tensor or tuple or list*) – The number of repeating times. If repeat_times is a list or tuple, all its elements should be integers or 1-D Tensors with the data type int32. If repeat_times is a Tensor, it should be an 1-D Tensor with the data type int32. Length must be the same as the number of dimensions in input.

Returns

Return type A Tensor. Has the same type as input.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant([[1,2,3],[1,2,3]])
>>> y = tlx.ops.tile(x, [2, 1])
>>> [[1, 2, 3],
>>>  [1, 2, 3],
>>>  [1, 2, 3],
>>>  [1, 2, 3]]
```

where

tensorlayerx.**where**(*condition*, *x*, *y*)

Return a tensor of elements selected from either x or y, depending on condition.

Parameters

- **condition** (*tensor of bool*) – When True (nonzero), yield x, otherwise yield y
- **x** (*tensor*) – values selected at indices where condition is True
- **y** (*tensor*) – values selected at indices where condition is False

Returns

Return type A tensor of shape equal to the broadcasted shape of condition, x, y

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.convert_to_tensor([0.9, 0.1, 3.2, 1.2])
>>> y = tlx.convert_to_tensor([1.0, 1.0, 1.0, 1.0])
>>> res = tlx.ops.where(x>1, x, y)
>>> print(res)
>>> [1.0, 1.0, 3.2, 1.2]
```

ones_like

tensorlayerx.**ones_like**(*x*, *dtype=None*)

This OP returns a Tensor filled with the value 1, with the same shape and data type (use dtype if dtype is not None) as x.

Parameters

- **x** (*tensor*) – The input tensor which specifies shape and dtype.
- **dtype** (*str*) – A type for the returned Tensor.If dtype is None, the data type is the same as x. Default is None.

Returns

Return type A Tensor filled with the value 1, with the same shape and data type (use dtype if dtype is not None) as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.convert_to_tensor([0.9, 0.1, 3.2, 1.2])
>>> res = tlx.ops.ones_like(x, dtype="int32")
>>> print(res)
>>> [1, 1, 1, 1]
```

zeros_like

tensorlayerx.**zeros_like**(*x*, *dtype=None*)

This OP returns a Tensor filled with the value 0, with the same shape and data type (use dtype if dtype is not None) as x.

Parameters

- **x** (*tensor*) – The input tensor which specifies shape and dtype.
- **dtype** (*str*) – A type for the returned Tensor.If dtype is None, the data type is the same as x. Default is None.

Returns

Return type A Tensor filled with the value 0, with the same shape and data type (use dtype if dtype is not None) as x.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.convert_to_tensor([0.9, 0.1, 3.2, 1.2])
>>> res = tlx.ops.zeros_like(x, dtype="int32")
>>> print(res)
>>> [0, 0, 0, 0]
```

stack

tensorlayerx.**stack**(*values*, *axis=0*)

Stacks a list of rank-R tensors into one rank-(R+1) tensor.

Parameters

- **values** (*list*) – A list of Tensor objects with the same shape and type.
- **axis** (*int*) – An int. The axis to stack along. Defaults to the first dimension. Negative values wrap around, so the valid range is [-(R+1), R+1).

Returns

Return type A stacked Tensor with the same type as values.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant([1,2,3])
>>> y = tlx.ops.constant([1,2,3])
>>> res = tlx.ops.stack([x, y])
>>> [[1, 2, 3],
>>>  [1, 2, 3]]
```

split

tensorlayerx.**split**(*value*, *num_or_size_splits*, *axis=0*)

Splits a tensor into sub tensors.

Parameters

- **value** (*tensor*) – The Tensor to split.
- **num_or_size_splits** (*int or list*) – Either an integer indicating the number of splits along split_dim or a 1-D integer Tensor or Python list containing the sizes of each output tensor along split_dim.
- **axis** (*int*) – The dimension along which to split. Must be in the range [-rank(value), rank(value)). Defaults to 0.

Returns

Return type Tensor objects resulting from splitting value.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.ones([3, 9, 5])
>>> y1, y2, y3 = tlx.ops.split(x, 3, axis=1)
>>> y1, y2, y3 = tlx.ops.split(x, [1,3,5], axis=1)
```

squeeze

tensorlayerx.**squeeze**(*x*, *axis=None*)

Removes dimensions of size 1 from the shape of a tensor.

Parameters

- **x** (*tensor*) – The input Tensor.
- **axis** (*int or list or tuple*) – An integer or list/tuple of integers, indicating the dimensions to be squeezed. Default is None. The range of axis is [ndim(x),ndim(x)). If axis is negative, axis=axis+ndim(x). If axis is None, all the dimensions of x of size 1 will be removed.

Returns

Return type Squeezed Tensor with the same data type as input Tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.ones(shape=[1,2,3])
>>> res = tlx.ops.squeeze(x, axis=0)
>>> print(res.shape)
>>> [2, 3]
```

expand_dims

tensorlayerx.**expand_dims**(*input*, *axis*)

Inserts a dimension of 1 into a tensor's shape.

Parameters

- **input** (*tensor*) – A Tensor.
- **axis** (*int*) – 0-D (scalar). Specifies the dimension index at which to expand the shape of input. Must be in the range [-rank(input) - 1, rank(input)].

Returns

Return type A Tensor with the same data as input, but its shape has an additional dimension of size 1 added.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.ones([1,2,3])
>>> res = tlx.ops.expand_dims(x, axis=0)
>>> print(res.shape)
>>>  [1, 1, 2, 3]
```

unsorted_segment_sum

tensorlayerx.**unsorted_segment_sum**(*x*, *segment_ids*, *num_segments*)

Computes the sum along segments of a tensor.

Parameters

- **x** (*tensor*) – A Tensor.
- **segment_ids** (*Tensor or list or tuple*) – Must be one of the following types: int32, int64.
- **num_segments** (*int or tensor*) – should equal the number of distinct segment IDs.

Returns

Return type A Tensor. Has the same type as data.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant([1,2,3])
>>> res = tlx.ops.unsorted_segment_sum(x, (0, 0, 1), num_segments=2)
>>> print(res)
>>> [2, 3]
```

unsorted_segment_mean

tensorlayerx.**unsorted_segment_mean**(*x*, *segment_ids*, *num_segments*)

Computes the mean along segments of a tensor.

Parameters

- **x** (*tensor*) – A Tensor.
- **segment_ids** (*Tensor or list or tuple*) – Must be one of the following types: int32, int64.
- **num_segments** (*int or tensor*) – should equal the number of distinct segment IDs.

Returns

Return type A Tensor. Has the same type as data.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant([1.,2.,3.])
>>> res = tlx.ops.unsorted_segment_mean(x, (0, 0, 1), num_segments=2)
>>> print(res)
>>> [1.5, 3]
```

unsorted_segment_min

tensorlayerx.**unsorted_segment_min**(*x*, *segment_ids*, *num_segments*)

Computes the min along segments of a tensor.

Parameters

- **x** (*tensor*) – A Tensor.
- **segment_ids** (*Tensor or list or tuple*) – Must be one of the following types: int32, int64.
- **num_segments** (*int or tensor*) – should equal the number of distinct segment IDs.

Returns

Return type A Tensor. Has the same type as data.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant([1.,2.,3.])
>>> res = tlx.ops.unsorted_segment_min(x, (0, 0, 1), num_segments=2)
>>> print(res)
>>> [1, 3]
```

unsorted_segment_max

tensorlayerx.**unsorted_segment_max**(*x*, *segment_ids*, *num_segments*)

Computes the max along segments of a tensor.

Parameters

- **x** (*tensor*) – A Tensor.
- **segment_ids** (*Tensor or list or tuple*) – Must be one of the following types: int32, int64.
- **num_segments** (*int or tensor*) – should equal the number of distinct segment IDs.

Returns

Return type A Tensor. Has the same type as data.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.constant([1.,2.,3.])
>>> res = tlx.ops.unsorted_segment_max(x, (0, 0, 1), num_segments=2)
>>> print(res)
>>> [2, 3]
```

set_seed

tensorlayerx.**set_seed**(*seed*)

Parameters **seed** (*int*) – The random seed to set.

Examples

```
>>> import tensorlayerx as tlx
>>> tlx.ops.set_seed(42)
```

is_tensor

tensorlayerx.**is_tensor**(*x*)

Parameters **x** (*input*) – A python object to check.

Returns

Return type a bool Value. if x is tensor return True, else return False.

Examples

```
>>> import tensorlayerx as tlx
>>> tlx.ops.is_tensor(a)
```

tensor_scatter_nd_update

tensorlayerx.**tensor_scatter_nd_update**(*tensor*, *indices*, *updates*)

Parameters

- **tensor** (*Tensor*) – tensor to update.
- **indices** (*list*) – indices to update.
- **updates** (*Tensor*) – value to apply at the indices

Returns

Return type updated Tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> tensor = tlx.ops.ones(shape=(5,3))
>>> indices = [[0],[ 4],[ 2]]
>>> updates = tlx.ops.convert_to_tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> new_tensor = tlx.ops.tensor_scatter_nd_update(tensor, indices, updates)
>>>  [[1. 2. 3.]
>>>  [1. 1. 1.]
>>>  [7. 8. 9.]
>>>  [1. 1. 1.]
>>>  [4. 5. 6.]]
```

scatter_update

tensorlayerx.**scatter_update**(*tensor*, *indices*, *updates*)

Applies sparse updates to a variable

Parameters

- **tensor** (*Tensor*) – A Tensor. The dim of tensor must be 1.
- **indices** (*Tensor*) – Indices into the tensor.
- **updates** (*Tensor*) – Updated values

Returns

Return type Tensor after updated.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.ones((5,))
>>> indices = tlx.ops.convert_to_tensor([0,  4,  2])
>>> updates = tlx.ops.convert_to_tensor([1., 4., 7.])
>>> res = tlx.ops.scatter_update(x, indices, updates)
>>> [1. 1. 7. 1. 4.]
```

diag

tensorlayerx.**diag**(*input*, *diagonal=0*)

Parameters

- **input** (*Tensor*) – the input tensor.
- **diagonal** (*int*) – the diagonal to consider. Defualt is 0.

Returns

Return type the output tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> tensor = tlx.ops.convert_to_tensor([[1,2,3],[4,5,6],[7,8,9]]))
>>> new_tensor = tlx.ops.diag(tensor)
>>> [1, 5, 9]
```

mask_select

tensorlayerx.**mask_select**(*x*, *mask*, *axis=0*)

Parameters

- **x** (*Tensor*) – N-D Tensor.
- **mask** (*Tensor*) – N-D boolean Tensor or 1-D boolean Tensor
- **axis** – the axis in tensor to mask from. By default, axis is 0.

Returns

Return type the output tensor.

Examples

```
>>> import tensorlayerx as tlx
>>> tensor = tlx.ops.convert_to_tensor([[1,2,3],[4,5,6],[7,8,9]]))
>>> mask = tlx.ops.convert_to_tensor(np.array([True, False, True]), dtype=tlx.
↪bool)
>>> new_tensor = tlx.ops.mask_select(tensor, mask)
>>> [[1, 2, 3],[7, 8, 9]]
```

eye

tensorlayerx.**eye**(*n*, *m=None*, *dtype=None*)

Parameters

- **n** (*int*) – the number of rows
- **m** (*int or None*) – the number of columns with default being n
- **dtype** (*str or None*) – the desired data type of returned tensor. Default: if None, use float32

Returns

Return type A 2-D tensor with ones on the diagonal and zeros elsewhere

Examples

```
>>> import tensorlayerx as tlx
>>> tlx.ops.eye(2)
>>> [[1,0],
>>> [0,1]]
```

einsum

tensorlayerx.**einsum**(*equation*, **operands*)

Sums the product of the elements of the input operands along dimensions specified using a notation based on the Einstein summation convention.

Parameters

- **equation** (*An attribute*) – represent the operation you want to do. the value can contain only letters([a-z][A-Z]), commas(,), ellipsis(...), and arrow(->). the letters represent inputs's tensor dimension, commas(,)represent separate tensors, ellipsis(...) indicates the tensor dimension that you do not care about, the left of the arrow(->) indicates the input tensors, and the right of it indicates the desired output dimension.
- **operands** (*list*) – input tensor used for calculation. the data type of the tensor must be the same.

Returns

Return type Tensor, the shape of it can be obtained from the equation, and the data type is the same as input tensors.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.nn.Input((5,))
>>> y = tlx.nn.Input((4,))
>>> out = tlx.ops.einsum('i,j->ij', x, y)
>>> cal_enisum = tlx.ops.Einsum('i,j->ij')
>>> out = cal_enisum(x, y)
```

set_device

tensorlayerx.**set_device**(*device='GPU'*, *id=0*)

This function can specify the global device which the OP will run.

Parameters

- **device** (*str*) – Specific running device. It can be 'CPU', 'GPU' and 'Ascend'(In mindspore backend).
- **id** (*int*) – Device id.

get_device

tensorlayerx.**get_device**()

This function can get the specified global device.

Returns

Return type The global device.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.get_device()
>>> "CPU"
```

to_device

tensorlayerx.**to_device**(*tensor*, *device='GPU'*, *id=0*)

Returns a copy of Tensor in specified device.

Parameters

- **tensor** (*Tensor*) – A tensor.
- **device** (*str*) – The specified device. Support 'GPU' and 'CPU'. Default is 'GPU'.
- **id** (*int*) – The id of specified device. Default is 0.

Examples

```
>>> import tensorlayerx as tlx
>>> x = tlx.ops.ones((5,))
>>> x = tlx.ops.to_device(x, device="GPU", id=0)
```

2.11 API - Optimizers

TensorLayer provides rich layer implementations trailed for various benchmarks and domain-specific problems. In addition, we also support transparent access to native TensorFlow parameters. For example, we provide not only layers for local response normalization, but also layers that allow user to apply `tf.ops.lrn` on `network.outputs`. More functions can be found in TensorFlow API.

TensorLayerX provides simple API and tools to ease research, development and reduce the time to production. Therefore, we provide the latest state of the art optimizers that work with Tensorflow, MindSpore, PaddlePaddle and PyTorch. The optimizers functions provided by Tensorflow, MindSpore, PaddlePaddle and PyTorch can be used in TensorLayerX. We have also wrapped the optimizers functions for each framework, which can be found in tensorlayerx.optimizers. In addition, we provide the latest state of Optimizers Dynamic Learning Rate that work with Tensorflow, MindSpore, PaddlePaddle and PyTorch.

2.11.1 Optimizers List

Adadelta([lr, rho, eps, weight_decay, grad_clip])	Optimizer that implements the Adadelta algorithm.
Adagrad([lr, initial_accumulator, eps, ...])	Optimizer that implements the Adagrad algorithm.
Adam([lr, beta_1, beta_2, eps, ...])	Optimizer that implements the Adam algorithm.
Adamax([lr, beta_1, beta_2, eps, ...])	Optimizer that implements the Adamax algorithm.
Ftrl([lr, lr_power, ...])	Optimizer that implements the FTRL algorithm.
Nadam([lr, beta_1, beta_2, eps, ...])	Optimizer that implements the NAdam algorithm.
RMSprop([lr, rho, momentum, eps, centered, ...])	Optimizer that implements the RMSprop algorithm.
SGD([lr, momentum, weight_decay, grad_clip])	Gradient descent (with momentum) optimizer.
Momentum([lr, momentum, nesterov, ...])	Optimizer that implements the Momentum algorithm.
Lamb()	Optimizer that implements the Layer-wise Adaptive Moments (LAMB).
LARS()	LARS is an optimization algorithm employing a large batch optimization technique.

2.11.2 Optimizers Dynamic Learning Rate List

LRScheduler([learning_rate, last_epoch, verbose])	LRScheduler Base class.
StepDecay(learning_rate, step_size[, gamma, ...])	Update the learning rate of `optimizer` by `gamma` every `step_size` number of epoch.
CosineAnnealingDecay(learning_rate, T_max[, ...])	Set the learning rate using a cosine annealing schedule, where η_{max} is set to the initial learning_rate.
NoamDecay(d_model, warmup_steps[, ...])	Applies Noam Decay to the initial learning rate.
PiecewiseDecay(boundaries, values[, ...])	Piecewise learning rate scheduler.
NaturalExpDecay(learning_rate, gamma[, ...])	Applies natural exponential decay to the initial learning rate.
InverseTimeDecay(learning_rate, gamma[, ...])	Applies inverse time decay to the initial learning rate.
PolynomialDecay(learning_rate, decay_steps)	Applies polynomial decay to the initial learning rate.
LinearWarmup(learning_rate, warmup_steps, ...)	Linear learning rate warm up strategy.
ExponentialDecay(learning_rate, gamma[, ...])	Update learning rate by *gamma* each epoch.
MultiStepDecay(learning_rate, milestones[, ...])	Update the learning rate by `gamma` once `epoch` reaches one of the milestones.
LambdaDecay(learning_rate, lr_lambda[, ...])	Sets the learning rate of `optimizer` by function `lr_lambda` .
ReduceOnPlateau(learning_rate[, mode, ...])	Reduce learning rate when `metrics` has stopped descending.

Adadelta

class tensorlayerx.optimizers.**Adadelta**(*lr=0.001*, *rho=0.95*, *eps=1e-07*, *weight_decay=0.0*, *grad_clip=None*)

Optimizer that implements the Adadelta algorithm. Equivalent to tf.optimizers.Adadelta.

References

- https://tensorflow.google.cn/api_docs/python/tf/keras/optimizers/Adadelta?hl=en

Parameters

- **lr** (*A Tensor, floating point value*) – The learning rate. Defaults to 0.001.
- **rho** (*float or constant float tensor*) – A Tensor or a floating point value. The decay rate.
- **eps** (*float*) – A small constant for numerical stability.Defaults to 1e-7.
- **weight_decay** (*float*) – weight decay (L2 penalty) (default: 0.0)
- **grad_clip** (*GradientClip or None*) – Gradient cliping strategy.There are three cliping strategies (*tlx.ops.ClipGradByValue* , *tlx.ops.ClipGradByNorm*, *tlx.ops.ClipByGlobalNorm*). Default None, meaning there is no gradient clipping.

Examples

With TensorLayerx

```
>>> import tensorlayerx as tlx
>>> optimizer = tlx.optimizers.Adadelta(0.001)
>>> optimizer.apply_gradients(zip(grad, train_weights))
```

Adagrad

class tensorlayerx.optimizers.**Adagrad**(*lr=0.001*, *initial_accumulator=0.1*, *eps=1e-07*, *weight_decay=0.0*, *grad_clip=None*)

Optimizer that implements the Adagrad algorithm. Equivalent to tf.optimizers.Adagrad.

References

- https://tensorflow.google.cn/api_docs/python/tf/keras/optimizers/Adagrad?hl=en

Parameters

- **lr** (*A Tensor, floating point value*) – The learning rate. Defaults to 0.001.
- **initial_accumulator_value** (*float*) – Floating point value. Starting value for the accumulators (per-parameter momentum values). Must be non-negative.Defaults to 0.95.
- **eps** (*float*) – A small constant for numerical stability.Defaults to 1e-7.
- **weight_decay** (*float*) – weight decay (L2 penalty) (default: 0.0)
- **grad_clip** (*GradientClip or None*) – Gradient cliping strategy.There are three cliping strategies (*tlx.ops.ClipGradByValue* , *tlx.ops.ClipGradByNorm*, *tlx.ops.ClipByGlobalNorm*). Default None, meaning there is no gradient clipping.

Examples

With TensorLayerx

```
>>> import tensorlayerx as tlx
>>> optimizer = tlx.optimizers.Adagrad(0.001)
>>> optimizer.apply_gradients(zip(grad, train_weights))
```

Adam

class tensorlayerx.optimizers.**Adam**(*lr=0.001*, *beta_1=0.9*, *beta_2=0.999*, *eps=1e-07*, *weight_decay=0.0*, *grad_clip=None*)

Optimizer that implements the Adam algorithm. Equivalent to tf.optimizers.Adam.

References

- https://tensorflow.google.cn/api_docs/python/tf/keras/optimizers/Adam?hl=en

Parameters

- **lr** (*A Tensor, floating point value*) – The learning rate. Defaults to 0.001.
- **beta_1** (*float or constant float tensor*) – The exponential decay rate for the 1st moment estimates. Defaults to 0.9.
- **beta_2** (*float or constant float tensor*) – The exponential decay rate for the 2nd moment estimates. Defaults to 0.999.
- **eps** (*float*) – A small constant for numerical stability.Defaults to 1e-7.
- **weight_decay** (*float*) – weight decay (L2 penalty) (default: 0.0)
- **grad_clip** (*GradientClip or None*) – Gradient cliping strategy.There are three cliping strategies (*tlx.ops.ClipGradByValue* , *tlx.ops.ClipGradByNorm*, *tlx.ops.ClipByGlobalNorm*). Default None, meaning there is no gradient clipping.

Examples

With TensorLayerx

```
>>> import tensorlayerx as tlx
>>> optimizer = tlx.optimizers.Adam(0.001)
>>> optimizer.apply_gradients(zip(grad, train_weights))
```

Adamax

class tensorlayerx.optimizers.**Adamax**(*lr=0.001*, *beta_1=0.9*, *beta_2=0.999*, *eps=1e-07*, *weight_decay=0.0*, *grad_clip=None*)

Optimizer that implements the Adamax algorithm. Equivalent to tf.optimizers.Adamax.

References

- https://tensorflow.google.cn/api_docs/python/tf/keras/optimizers/Adamax?hl=en

Parameters

- **lr** (*A Tensor, floating point value*) – The learning rate. Defaults to 0.001.
- **beta_1** (*float or constant float tensor*) – The exponential decay rate for the 1st moment estimates. Defaults to 0.9.
- **beta_2** (*float or constant float tensor*) – The exponential decay rate for the exponentially weighted infinity norm. Defaults to 0.999.
- **eps** (*float*) – A small constant for numerical stability.Defaults to 1e-7.
- **weight_decay** (*float*) – weight decay (L2 penalty) (default: 0.0)
- **grad_clip** (*GradientClip or None*) – Gradient cliping strategy.There are three cliping strategies (*tlx.ops.ClipGradByValue* , *tlx.ops.ClipGradByNorm*, *tlx.ops.ClipByGlobalNorm*). Default None, meaning there is no gradient clipping.

Examples

With TensorLayerx

```
>>> import tensorlayerx as tlx
>>> optimizer = tlx.optimizers.Adamax(0.001)
>>> optimizer.apply_gradients(zip(grad, train_weights))
```

Ftrl

class tensorlayerx.optimizers.**Ftrl**(*lr=0.001*, *lr_power=-0.5*, *initial_accumulator_value=0.1*, *l1_regularization_strength=0.0*, *l2_regularization_strength=0.0*, *beta=0.0*, *l2_shrinkage_regularization_strength=0.0*, *weight_decay=0.0*, *grad_clip=None*)

Optimizer that implements the FTRL algorithm. Equivalent to tf.optimizers.Ftrl.

References

- https://tensorflow.google.cn/api_docs/python/tf/keras/optimizers/Ftrl?hl=en

Parameters

- **lr** (*A Tensor, floating point value*) – The learning rate. Defaults to 0.001.
- **lr_power** (*float*) – Controls how the learning rate decreases during training. Use zero for a fixed learning rate.
- **initial_accumulator_value** (*float*) – The starting value for accumulators. Only zero or positive values are allowed.
- **l1_regularization_strength** (*float*) – A float value, must be greater than or equal to zero. Defaults to 0.0.

- **l2_regularization_strength** (*float*) – A float value, must be greater than or equal to zero. Defaults to 0.0.
- **l2_shrinkage_regularization_strength** (*float*) – This differs from L2 above in that the L2 above is a stabilization penalty, whereas this L2 shrinkage is a magnitude penalty. When input is sparse shrinkage will only happen on the active weights.
- **beta** (*float*) – A float value, representing the beta value from the paper. Defaults to 0.0.
- **weight_decay** (*float*) – weight decay (L2 penalty) (default: 0.0)
- **grad_clip** (*GradientClip or None*) – Gradient cliping strategy.There are three cliping strategies (*tlx.ops.ClipGradByValue* , *tlx.ops.ClipGradByNorm*, *tlx.ops.ClipByGlobalNorm*). Default None, meaning there is no gradient clipping.

Examples

With TensorLayerx

```
>>> import tensorlayerx as tlx
>>> optimizer = tlx.optimizers.Ftrl(0.001)
>>> optimizer.apply_gradients(zip(grad, train_weights))
```

Nadam

class tensorlayerx.optimizers.**Nadam**(*lr=0.001*, *beta_1=0.9*, *beta_2=0.999*, *eps=1e-07*, *weight_decay=0.0*, *grad_clip=None*)

Optimizer that implements the NAdam algorithm. Equivalent to tf.optimizers.Nadam.

References

- https://tensorflow.google.cn/api_docs/python/tf/keras/optimizers/Nadam?hl=en

Parameters

- **lr** (*A Tensor, floating point value*) – The learning rate. Defaults to 0.001.
- **beta_1** (*float or constant float tensor*) – The exponential decay rate for the 1st moment estimates. Defaults to 0.9.
- **beta_2** (*float or constant float tensor*) – The exponential decay rate for the exponentially weighted infinity norm. Defaults to 0.999.
- **eps** (*float*) – A small constant for numerical stability.Defaults to 1e-7.
- **weight_decay** (*float*) – weight decay (L2 penalty) (default: 0.0)
- **grad_clip** (*GradientClip or None*) – Gradient cliping strategy.There are three cliping strategies (*tlx.ops.ClipGradByValue* , *tlx.ops.ClipGradByNorm*, *tlx.ops.ClipByGlobalNorm*). Default None, meaning there is no gradient clipping.

Examples

With TensorLayerx

```
>>> import tensorlayerx as tlx
>>> optimizer = tlx.optimizers.Nadam(0.001)
>>> optimizer.apply_gradients(zip(grad, train_weights))
```

RMSprop

class tensorlayerx.optimizers.**RMSprop**(*lr=0.001*, *rho=0.9*, *momentum=0.0*, *eps=1e-07*, *centered=False*, *weight_decay=0.0*, *grad_clip=None*)

Optimizer that implements the RMSprop algorithm. Equivalent to tf.optimizers.RMSprop.

References

- https://tensorflow.google.cn/api_docs/python/tf/keras/optimizers/RMSprop?hl=en

Parameters

- **lr** (*A Tensor, floating point value*) – The learning rate. Defaults to 0.001.
- **rho** (*float*) – Discounting factor for the history/coming gradient. Defaults to 0.9.
- **momentum** (*float*) – A scalar or a scalar Tensor. Defaults to 0.0.
- **eps** (*float*) – A small constant for numerical stability.Defaults to 1e-7.
- **centered** (*bool*) – 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.
- **weight_decay** (*float*) – weight decay (L2 penalty) (default: 0.0)
- **grad_clip** (*GradientClip or None*) – Gradient cliping strategy.There are three cliping strategies (*tlx.ops.ClipGradByValue* , *tlx.ops.ClipGradByNorm*, *tlx.ops.ClipByGlobalNorm*). Default None, meaning there is no gradient clipping.

Examples

With TensorLayerx

```
>>> import tensorlayerx as tlx
>>> optimizer = tlx.optimizers.RMSprop(0.001)
>>> optimizer.apply_gradients(zip(grad, train_weights))
```

SGD

class tensorlayerx.optimizers.**SGD**(*lr=0.01*, *momentum=0.0*, *weight_decay=0.0*, *grad_clip=None*)

Gradient descent (with momentum) optimizer. Equivalent to tf.optimizers.SGD.

References

- https://tensorflow.google.cn/api_docs/python/tf/keras/optimizers/SGD?hl=en

Parameters

- **lr** (*A Tensor, floating point value*) – The learning rate. Defaults to 0.001.
- **momentum** (*float*) – float hyperparameter >= 0 that accelerates gradient descent in the relevant direction and dampens oscillations. Defaults to 0, i.e., vanilla gradient descent.
- **weight_decay** (*float*) – weight decay (L2 penalty) (default: 0.0)
- **grad_clip** (*GradientClip or None*) – Gradient cliping strategy.There are three cliping strategies (*tlx.ops.ClipGradByValue* , *tlx.ops.ClipGradByNorm*, *tlx.ops.ClipByGlobalNorm*). Default None, meaning there is no gradient clipping.

Examples

With TensorLayerx

```
>>> import tensorlayerx as tlx
>>> optimizer = tlx.optimizers.SGD(0.01)
>>> optimizer.apply_gradients(zip(grad, train_weights))
```

Momentum

class tensorlayerx.optimizers.**Momentum**(*lr=0.01*, *momentum=0.0*, *nesterov=False*, *weight_decay=0.0*, *grad_clip=None*)

Optimizer that implements the Momentum algorithm. Equivalent to tf.compat.v1.train.MomentumOptimizer

References

- https://tensorflow.google.cn/api_docs/python/tf/compat/v1/train/MomentumOptimizer?hl=en&version=nightly

Parameters

- **lr** (*A Tensor, floating point value*) – The learning rate. Defaults to 0.001.
- **momentum** (*float*) – A Tensor or a floating point value. The momentum. Defaults to 0
- **use_locking** (*bool*) – If True use locks for update operations.
- **use_nesterov** (*bool*) – If True use Nesterov Momentum. See (Sutskever et al., 2013). This implementation always computes gradients at the value of the variable(s) passed to the optimizer. Using Nesterov Momentum makes the variable(s) track the values called theta_t + mu*v_t in the paper. This implementation is an approximation of the original formula, valid for high values of momentum. It will compute the "adjusted gradient" in NAG by assuming that the new gradient will be estimated by the current average gradient plus the product of momentum and the change in the average gradient.
- **weight_decay** (*float*) – weight decay (L2 penalty) (default: 0.0)
- **grad_clip** (*GradientClip or None*) – Gradient cliping strategy.There are three cliping strategies (*tlx.ops.ClipGradByValue* , *tlx.ops.ClipGradByNorm*, *tlx.ops.ClipByGlobalNorm*). Default None, meaning there is no gradient clipping.

Examples

With TensorLayerx

```
>>> import tensorlayerx as tlx
>>> optimizer = tlx.optimizers.Momentum(0.01, momentum=0.9)
>>> optimizer.apply_gradients(zip(grad, train_weights))
```

Lamb

class tensorlayerx.optimizers.**Lamb**

Optimizer that implements the Layer-wise Adaptive Moments (LAMB).

References

- https://tensorflow.google.cn/addons/api_docs/python/tfa/optimizers/LAMB?hl=en

LARS

class tensorlayerx.optimizers.**LARS**

LARS is an optimization algorithm employing a large batch optimization technique. Refer to paper LARGE BATCH TRAINING OF CONVOLUTIONAL NETWORKS.

References

- https://www.mindspore.cn/docs/api/zh-CN/r1.5/api_python/nn/mindspore.nn.LARS.html?highlight=lars#mindspore.nn.LARS

LRScheduler

class tensorlayerx.optimizers.lr.**LRScheduler**(*learning_rate=0.1*, *last_epoch=-1*, *verbose=False*)

LRScheduler Base class. Define the common interface of a learning rate scheduler.

User can import it by `from tl.optimizer.lr import LRScheduler`,

then overload it for your subclass and have a custom implementation of `get_lr()`.

References

- https://www.paddlepaddle.org.cn/documentation/docs/zh/api/paddle/optimizer/lr/LRScheduler_cn.html

Parameters

- **learning_rate** (*A floating point value*) – The learning rate. Defaults to 0.1.
- **last_epoch** (*int*) – The index of last epoch. Can be set to restart training. Default: -1, means initial learning rate.
- **verbose** (*bool*) – If `True`, prints a message to stdout for each update. Default: `False`.

Examples

With TensorLayerX

```
>>> #Here is an example of a simple ``StepDecay`` implementation.
>>> import tensorlayerx as tlx
>>> from tensorlayerx.optimizers.lr import LRScheduler
>>> class StepDecay(LRScheduler):
>>>     def __init__(self, learning_rate, step_size, gamma = 0.1, last_epoch = -1,
↪ verbose=False):
>>>         if not isinstance(step_size, int):
>>>             raise TypeError("The type of 'step_size' must be 'int', but␣
↪received %s." %type(step_size))
>>>         if gamma >= 1.0 :
>>>             raise ValueError('gamma should be < 1.0.')
>>>         self.step_size = step_size
>>>         self.gamma = gamma
>>>         super(StepDecay, self).__init__(learning_rate, last_epoch, verbose)
>>>     def get_lr(self):
>>>         i = self.last_epoch // self.step_size
>>>         return self.base_lr * (self.gamma**i)
```

StepDecay

class tensorlayerx.optimizers.lr.**StepDecay**(*learning_rate*, *step_size*, *gamma=0.1*, *last_epoch=-1*, *verbose=False*)

Update the learning rate of `optimizer` by `gamma` every `step_size` number of epoch.

$$new\_learning\_rate = learning\_rate * gamma^{epoch//step_size}$$

References

- https://www.paddlepaddle.org.cn/documentation/docs/zh/api/paddle/optimizer/lr/StepDecay_cn.html

Parameters

- **learning_rate** (*float*) – The learning rate.
- **step_size** (*int*) – the interval to update.
- **gamma** (*float*) – The Ratio that the learning rate will be reduced. `new_lr = origin_lr * gamma` . It should be less than 1.0. Default: 0.1.
- **last_epoch** (*int*) – The index of last epoch. Can be set to restart training. Default: -1, means initial learning rate.
- **verbose** (*bool*) – If `True`, prints a message to stdout for each update. Default: `False` .

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> scheduler = tlx.optimizers.lr.StepDecay(learning_rate = 0.1, step_size = 10, ␣
↪gamma = 0.1, last_epoch = -1, verbose = False)
```

(continues on next page)

(continued from previous page)

```
>>> sgd = tlx.optimizers.SGD(learning_rate=scheduler,momentum=0.2)
>>> for epoch in range(100):
>>>     for batch in range(100):
>>>         # train model
>>>          scheduler.step() # If you update learning rate each batch
>>>     #scheduler.step()    # If you update learning rate each epoch
```

CosineAnnealingDecay

class tensorlayerx.optimizers.lr.**CosineAnnealingDecay**(*learning_rate*, *T_max*, *eta_min=0*, *last_epoch=-1*, *verbose=False*)

Set the learning rate using a cosine annealing schedule, where η_{max} is set to the initial learning_rate. T_{cur} is the number of epochs since the last restart in SGDR.

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right), \quad T_{cur} \neq (2k+1)T_{max};\ \eta_{t+1} = \eta_t + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 - \cos\left(\frac{1}{T_{ma}}\right)\right.$$

References

- https://www.paddlepaddle.org.cn/documentation/docs/zh/api/paddle/optimizer/lr/CosineAnnealingDecay_cn.html

Parameters

- **learning_rate** (*float or int*) – The initial learning rate, that is η_{max} . It can be set to python float or int number.
- **T_max** (*int*) – Maximum number of iterations. It is half of the decay cycle of learning rate.
- **eta_min** (*float or int*) – Minimum learning rate, that is η_{min} . Default: 0.
- **last_epoch** (*int*) – The index of last epoch. Can be set to restart training. Default: -1, means initial learning rate.
- **verbose** (*bool*) – If `True`, prints a message to stdout for each update. Default: `False` .

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> scheduler = tlx.optimizers.lr.CosineAnnealingDecay(learning_rate = 0.1, T_max␣
↪= 10, eta_min=0, last_epoch=-1, verbose=False)
>>> sgd = tlx.optimizers.SGD(learning_rate=scheduler,momentum=0.2)
>>> for epoch in range(100):
>>>     for step in range(100):
>>>         # train model
>>>          scheduler.step() # If you update learning rate each step
>>>     #scheduler.step()    # If you update learning rate each epoch
```

NoamDecay

class tensorlayerx.optimizers.lr.**NoamDecay**(*d_model*, *warmup_steps*, *learning_rate=1.0*, *last_epoch=-1*, *verbose=False*)

Applies Noam Decay to the initial learning rate.

$$new\_learning\_rate = learning\_rate * d_{model}^{-0.5} * min(epoch^{-0.5}, epoch * warmup\_steps^{-1.5})$$

References

- https://www.paddlepaddle.org.cn/documentation/docs/zh/api/paddle/optimizer/lr/NoamDecay_cn.html
- 'Attention is all you need'<https://arxiv.org/pdf/1706.03762.pdf>_

Parameters

- **d_model** (*int*) – The dimensionality of input and output feature vector of model. It is a python int number.
- **warmup_steps** (*int*) – The number of warmup steps. A super parameter. It is a python int number
- **learning_rate** (*float*) – The initial learning rate. It is a python float number. Default: 1.0.
- **last_epoch** (*int*) – The index of last epoch. Can be set to restart training. Default: -1, means initial learning rate.
- **verbose** (*bool*) – If `True`, prints a message to stdout for each update. Default: `False` .

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> scheduler = tlx.optimizers.lr.NoamDecay(d_model=0.01, warmup_steps=100,␣
↪verbose=True)
>>> sgd = tlx.optimizers.SGD(learning_rate=scheduler,momentum=0.2)
>>> for epoch in range(100):
>>>     for step in range(100):
>>>        # train model
>>>         scheduler.step() # If you update learning rate each step
>>>     #scheduler.step()    # If you update learning rate each epoch
```

PiecewiseDecay

class tensorlayerx.optimizers.lr.**PiecewiseDecay**(*boundaries*, *values*, *last_epoch=-1*, *verbose=False*)

Piecewise learning rate scheduler.

```
boundaries = [100, 200]
values = [1.0, 0.5, 0.1]
if epoch < 100:
    learning_rate = 1.0
elif 100 <= global_step < 200:
    learning_rate = 0.5
```

(continues on next page)

(continued from previous page)

```
else:
    learning_rate = 0.1
```

References

- https://www.paddlepaddle.org.cn/documentation/docs/zh/api/paddle/optimizer/lr/PiecewiseDecay_cn.html

Parameters

- **boundaries** (*list*) – A list of steps numbers.
- **values** (*list*) – A list of learning rate values that will be picked during different epoch boundaries.
- **last_epoch** (*int*) – The index of last epoch. Can be set to restart training. Default: -1, means initial learning rate.
- **verbose** (*bool*) – If `True`, prints a message to stdout for each update. Default: `False` .

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> scheduler = tlx.optimizers.lr.PiecewiseDecay(boundaries=[100, 200], values=[0.
↪1, 0.5, 0.1], verbose=True)
>>> sgd = tlx.optimizers.SGD(learning_rate=scheduler,momentum=0.2)
>>> for epoch in range(100):
>>>     for step in range(100):
>>>        # train model
>>>         scheduler.step() # If you update learning rate each step
>>>     #scheduler.step()    # If you update learning rate each epoch
```

NaturalExpDecay

class tensorlayerx.optimizers.lr.**NaturalExpDecay**(*learning_rate*, *gamma*, *last_epoch=-1*, *verbose=False*)

Applies natural exponential decay to the initial learning rate.

$$new\_learning\_rate = learning\_rate * e^{-gamma*epoch}$$

References

- https://www.paddlepaddle.org.cn/documentation/docs/zh/api/paddle/optimizer/lr/NaturalExpDecay_cn.html

Parameters

- **learning_rate** (*float*) – The initial learning rate.
- **gamma** (*float*) – A Ratio to update the learning rate. Default: 0.1.

- **last_epoch** (*int*) – The index of last epoch. Can be set to restart training. Default: -1, means initial learning rate.
- **verbose** (*bool*) – If `True`, prints a message to stdout for each update. Default: `False` .

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> scheduler = tlx.optimizers.lr.NaturalExpDecay(learning_rate=0.1, gamma=0.1,␣
↪verbose=True)
>>> sgd = tlx.optimizers.SGD(learning_rate=scheduler,momentum=0.2)
>>> for epoch in range(100):
>>>     for step in range(100):
>>>         # train model
>>>          scheduler.step() # If you update learning rate each step
>>>     #scheduler.step()    # If you update learning rate each epoch
```

InverseTimeDecay

class tensorlayerx.optimizers.lr.**InverseTimeDecay**(*learning_rate*, *gamma*, *last_epoch=-1*, *verbose=False*)

Applies inverse time decay to the initial learning rate.

$$new\_learning\_rate = \frac{learning\_rate}{1 + gamma * epoch}$$

References

- https://www.paddlepaddle.org.cn/documentation/docs/zh/api/paddle/optimizer/lr/InverseTimeDecay_cn.html

Parameters

- **learning_rate** (*float*) – The initial learning rate.
- **gamma** (*float*) – A Ratio to update the learning rate. Default: 0.1.
- **last_epoch** (*int*) – The index of last epoch. Can be set to restart training. Default: -1, means initial learning rate.
- **verbose** (*bool*) – If `True`, prints a message to stdout for each update. Default: `False` .

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> scheduler = tlx.optimizers.lr.InverseTimeDecay(learning_rate=0.1, gamma=0.1,␣
↪verbose=True)
>>> sgd = tlx.optimizers.SGD(learning_rate=scheduler,momentum=0.2)
>>> for epoch in range(100):
>>>     for step in range(100):
```

(continues on next page)

(continued from previous page)

```
>>>         # train model
>>>          scheduler.step() # If you update learning rate each step
>>>      #scheduler.step()    # If you update learning rate each epoch
```

PolynomialDecay

class tensorlayerx.optimizers.lr.**PolynomialDecay**(*learning_rate*, *decay_steps*, *end_lr=0.0001*, *power=1.0*, *cycle=False*, *last_epoch=-1*, *verbose=False*)

Applies polynomial decay to the initial learning rate.

If cycle is set to True, then:

$$decay\_steps = decay\_steps * math.ceil(\frac{epoch}{decay\_steps})$$

$$new\_learning\_rate = (learning\_rate - end\_lr) * (1 - \frac{epoch}{decay\_steps})^{power} + end\_lr$$

If cycle is set to False, then:

$$epoch = min(epoch, decay\_steps)$$

$$new\_learning\_rate = (learning\_rate - end\_lr) * (1 - \frac{epoch}{decay\_steps})^{power} + end\_lr$$

References

- https://www.paddlepaddle.org.cn/documentation/docs/zh/api/paddle/optimizer/lr/PolynomialDecay_cn.html

Parameters

- **learning_rate** (*float*) – The initial learning rate.
- **decay_steps** (*int*) – The decay step size. It determines the decay cycle.
- **end_lr** (*float*) – The minimum final learning rate. Default: 0.0001.
- **power** (*float*) – Power of polynomial. Default: 1.0.
- **cycle** (*bool*) – Whether the learning rate rises again. If True, then the learning rate will rise when it decrease to `end_lr` . If False, the learning rate is monotone decreasing. Default: False.
- **last_epoch** (*int*) – The index of last epoch. Can be set to restart training. Default: -1, means initial learning rate.
- **verbose** (*bool*) – If `True`, prints a message to stdout for each update. Default: `False` .

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> scheduler = tlx.optimizers.lr.PolynomialDecay(learning_rate=0.1, decay_
↪steps=50, verbose=True)
>>> sgd = tlx.optimizers.SGD(learning_rate=scheduler,momentum=0.2)
>>> for epoch in range(100):
>>>     for step in range(100):
>>>        # train model
>>>         scheduler.step() # If you update learning rate each step
>>>     #scheduler.step()    # If you update learning rate each epoch
```

LinearWarmup

class tensorlayerx.optimizers.lr.**LinearWarmup**(*learning_rate*, *warmup_steps*, *start_lr*, *end_lr*, *last_epoch=-1*, *verbose=False*)

Linear learning rate warm up strategy. Update the learning rate preliminarily before the normal learning rate scheduler.

When epoch < warmup_steps, learning rate is updated as:

$$lr = start\_lr + (end\_lr - start\_lr) * \frac{epoch}{warmup\_steps}$$

where start_lr is the initial learning rate, and end_lr is the final learning rate;

When epoch >= warmup_steps, learning rate is updated as:

$$lr = learning_rate$$

where `learning_rate` is float or any subclass of `LRScheduler` .

References

- https://www.paddlepaddle.org.cn/documentation/docs/zh/api/paddle/optimizer/lr/LinearWarmup_cn.html
- Bag of Tricks for Image Classification with Convolutional Neural Networks

Parameters

- **learning_rate** (*float*) – The initial learning rate.
- **warmup_steps** (*int*) – total steps of warm up.
- **start_lr** (*float*) – Initial learning rate of warm up.
- **end_lr** (*float*) – Final learning rate of warm up.
- **last_epoch** (*int*) – The index of last epoch. Can be set to restart training. Default: -1, means initial learning rate.
- **verbose** (*bool*) – If `True`, prints a message to stdout for each update. Default: `False` .

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> scheduler = tlx.optimizers.lr.LinearWarmup(learning_rate=0.1, warmup_steps=20,
↪ start_lr=0.0, end_lr=0.5, verbose=True)
>>> sgd = tlx.optimizers.SGD(learning_rate=scheduler,momentum=0.2)
>>> for epoch in range(100):
>>>     for step in range(100):
>>>        # train model
>>>         scheduler.step() # If you update learning rate each step
>>>     #scheduler.step()    # If you update learning rate each epoch
```

ExponentialDecay

class tensorlayerx.optimizers.lr.**ExponentialDecay**(*learning_rate*, *gamma*, *last_epoch=-1*, *verbose=False*)

Update learning rate by *gamma* each epoch.

When epoch < warmup_steps, learning rate is updated as:

$$new\_learning\_rate = last\_learning\_rate * gamma$$

References

- https://www.paddlepaddle.org.cn/documentation/docs/zh/api/paddle/optimizer/lr/ExponentialDecay_cn.html

Parameters

- **learning_rate** (*float*) – The initial learning rate.
- **gamma** (*float*) – The Ratio that the learning rate will be reduced. It should be less than 1.0. Default: 0.1.
- **last_epoch** (*int*) – The index of last epoch. Can be set to restart training. Default: -1, means initial learning rate.
- **verbose** (*bool*) – If `True`, prints a message to stdout for each update. Default: `False` .

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> scheduler = tlx.optimizers.lr.ExponentialDecay(learning_rate=0.1, gamma=0.9,␣
↪verbose=True)
>>> sgd = tlx.optimizers.SGD(learning_rate=scheduler,momentum=0.2)
>>> for epoch in range(100):
>>>     for step in range(100):
>>>        # train model
>>>         scheduler.step() # If you update learning rate each step
>>>     #scheduler.step()    # If you update learning rate each epoch
```

MultiStepDecay

class tensorlayerx.optimizers.lr.**MultiStepDecay**(*learning_rate*, *milestones*, *gamma=0.1*, *last_epoch=-1*, *verbose=False*)

Update the learning rate by `gamma` once `epoch` reaches one of the milestones. The algorithm can be described as the code below.

```
learning_rate = 0.1
milestones = [50, 100]
gamma = 0.1
if epoch < 50:
    learning_rate = 0.1
elif epoch < 100:
    learning_rate = 0.01
else:
    learning_rate = 0.001
```

References

- https://www.paddlepaddle.org.cn/documentation/docs/zh/api/paddle/optimizer/lr/MultiStepDecay_cn.html

Parameters

- **learning_rate** (*float*) – The initial learning rate.
- **milestones** (*list*) – List or tuple of each boundaries. Must be increasing.
- **gamma** (*float*) – The Ratio that the learning rate will be reduced. It should be less than 1.0. Default: 0.1.
- **last_epoch** (*int*) – The index of last epoch. Can be set to restart training. Default: -1, means initial learning rate.
- **verbose** (*bool*) – If `True`, prints a message to stdout for each update. Default: `False` .

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> scheduler = tlx.optimizers.lr.MultiStepDecay(learning_rate=0.1,␣
↪milestones=[50, 100], gamma=0.1, verbose=True)
>>> sgd = tlx.optimizers.SGD(learning_rate=scheduler,momentum=0.2)
>>> for epoch in range(100):
>>>     for step in range(100):
>>>        # train model
>>>         scheduler.step() # If you update learning rate each step
>>>     #scheduler.step()    # If you update learning rate each epoch
```

LambdaDecay

class tensorlayerx.optimizers.lr.**LambdaDecay**(*learning_rate*, *lr_lambda*, *last_epoch=-1*, *verbose=False*)

Sets the learning rate of `optimizer` by function `lr_lambda` . `lr_lambda` is funciton which receives

epoch .

The algorithm can be described as the code below.

```
learning_rate = 0.5        # init learning_rate
lr_lambda = lambda epoch: 0.95 ** epoch

learning_rate = 0.5        # epoch 0, 0.5*0.95**0
learning_rate = 0.475      # epoch 1, 0.5*0.95**1
learning_rate = 0.45125    # epoch 2, 0.5*0.95**2
```

References

- https://www.paddlepaddle.org.cn/documentation/docs/zh/api/paddle/optimizer/lr/LambdaDecay_cn.html

Parameters

- **learning_rate** (*float*) – The initial learning rate.
- **lr_lambda** (*function*) – A function which computes a factor by epoch , and then multiply the initial learning rate by this factor.
- **last_epoch** (*int*) – The index of last epoch. Can be set to restart training. Default: -1, means initial learning rate.
- **verbose** (*bool*) – If True, prints a message to stdout for each update. Default: False .

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> scheduler = tlx.optimizers.lr.LambdaDecay(learning_rate=0.1, lr_lambda=lambda
↪x:0.9**x, verbose=True)
>>> sgd = tlx.optimizers.SGD(learning_rate=scheduler,momentum=0.2)
>>> for epoch in range(100):
>>>     for step in range(100):
>>>        # train model
>>>         scheduler.step() # If you update learning rate each step
>>>     #scheduler.step()    # If you update learning rate each epoch
```

ReduceOnPlateau

class tensorlayerx.optimizers.lr.**ReduceOnPlateau**(*learning_rate*, *mode='min'*, *factor=0.1*, *patience=10*, *threshold=0.0001*, *threshold_mode='rel'*, *cooldown=0*, *min_lr=0*, *epsilon=1e-08*, *verbose=False*)

Reduce learning rate when metrics has stopped descending. Models often benefit from reducing the learning rate by 2 to 10 times once model performance has no longer improvement.

The metrics is the one which has been pass into step , it must be 1-D Tensor with shape [1]. When metrics stop descending for a patience number of epochs, the learning rate will be reduced to learning_rate * factor . (Specially, mode can also be set to 'max , in this case, when metrics stop ascending for a patience number of epochs, the learning rate will be reduced.)

In addition, After each reduction, it will wait a `cooldown` number of epochs before resuming above operation.

References

- https://www.paddlepaddle.org.cn/documentation/docs/zh/api/paddle/optimizer/lr/LambdaDecay_cn.html

Parameters

- **learning_rate** (*float*) – The initial learning rate.
- **mode** (*str*) –

  'min' or 'max' can be selected. Normally, it is 'min' , which means that the learning rate will reduce wh
  Specially, if it's set to `'max'` , the learning rate will reduce when `loss` stops ascending. Default: `'min'` .
- **factor** (*float*) – The Ratio that the learning rate will be reduced.It should be less than 1.0. Default: 0.1.
- **patience** (*int*) – When `loss` doesn't improve for this number of epochs, learing rate will be reduced. Default: 10.
- **threshold** (*float*) – `threshold` and `threshold_mode` will determine the minimum change of `loss` . This make tiny changes of `loss` will be ignored. Default: 1e-4.
- **threshold_mode** (*str*) – `'rel'` or `'abs'` can be selected. In `'rel'` mode, the minimum change of `loss` is `last_loss * threshold` , where `last_loss` is `loss` in last epoch. In `'abs'` mode, the minimum change of `loss` is `threshold` . Default: `'rel'` .
- **cooldown** (*int*) – The number of epochs to wait before resuming normal operation. Default: 0.
- **min_lr** (*float*) – The lower bound of the learning rate after reduction. Default: 0.
- **epsilon** (*float*) – Minimal decay applied to lr. If the difference between new and old lr is smaller than epsilon, the update is ignored. Default: 1e-8.
- **verbose** (*bool*) – If `True`, prints a message to stdout for each update. Default: `False` .

Examples

With TensorLayerX

```
>>> import tensorlayerx as tlx
>>> scheduler = tlx.optimizers.lr.ReduceOnPlateau(learning_rate=1.0, factor=0.5,␣
↪patience=5, verbose=True)
>>> sgd = tlx.optimizers.SGD(learning_rate=scheduler,momentum=0.2)
>>> for epoch in range(100):
>>>     for step in range(100):
>>>         # train model
>>>          scheduler.step() # If you update learning rate each step
>>>     #scheduler.step()    # If you update learning rate each epoch
```

CHAPTER

THREE

COMMAND-LINE REFERENCE

TensorLayerX provides a handy command-line tool *tlx* to perform some common tasks.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z