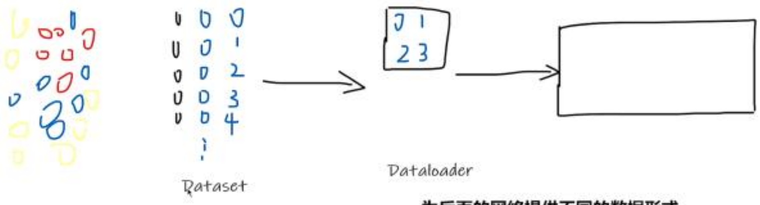
173.PyTorch初识

2025年4月24日 10:16

1.PyTorch加载数据初认识



垃圾（数据）

为后面的网络提供不同的数据形式

提供一种方式去获取数据及其label

如何获取每一个数据及其label

告诉我们总共有多少的数据

电脑 › 桌面 › hymenoptera_data › hymenoptera_data › train

名称	修改日期	类型	大小
ants	2019/12/7 20:02	文件夹	
bees	2019/12/7 20:02	文件夹	

```python
from torch.utils.data import Dataset
# import cv2
from PIL import Image
import os

class MyData(Dataset):
    def __init__(self,root_dir,label_dir):
        self.root_dir = root_dir
        self.label_dir =label_dir
        self.path = os.path.join(self.root_dir,self.label_dir)
        self.img_path = os.listdir(self.path)

    def __getitem__(self,idx):
        img_name = self.img_path[idx]
        img_item_path = os.path.join(self.root_dir,self.label_dir,img_name)
        img = Image.open(img_item_path)
        label = self.label_dir
        return img, label

    def __len__(self):
        return len(self.img_path)


root_dir = "dataset\\train"
ants_label_dir = "ants"
bees_label_dir = "bees"
ants_dataset = MyData(root_dir,ants_label_dir)
bees_dataset = MyData(root_dir,bees_label_dir)
train_dataset = ants_dataset + bees_dataset

img, label = train_dataset[5]
img.show()
```

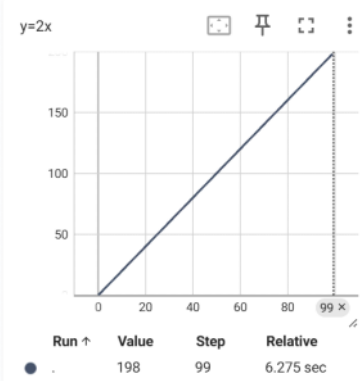
2.TensorBoard的使用（add_scalar）

1、Tensorboard 的使用

2、图像变换，transform的使用

```python
def add_scalar(self, tag, scalar_value, global_step=None, walltime=None):
    """Add scalar data to summary.

    Args:
        tag (string): Data identifier
        scalar_value (float or string/blobname): Value to save
        global_step (int): Global step value to record
        walltime (float): Optional override default walltime (time.time()
          with seconds after epoch of event
```


```python
from torch.utils.tensorboard import SummaryWriter


writer = SummaryWriter("logs")

# writer.add_image()
# y = 2x
for i in range(100):
    writer.add_scalar("y=2x",2*i,i)

writer.close()
```



终端控制台命令：

tensorboard --logdir=logs --port=6471

3.TensorBoard的使用（add_image）

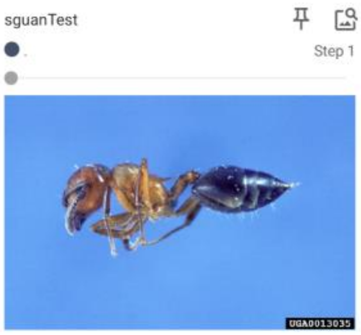
```python
def add_image(self, tag, img_tensor, global_step=None, walltime=None, dataformats='CHW'):
    """Add image data to summary.

    Note that this requires the ``pillow`` package.

    Args:
        tag (string): Data identifier
        img_tensor (torch.Tensor, numpy.array, or string/blobname): Image data
        global_step (int): Global step value to record
        walltime (float): Optional override default walltime (time.time())
          seconds after epoch of event
```

```python
In[2]: image_path = "data/train/ants_image/0013035.jpg"
In[3]: from PIL import Image
In[4]: img = Image.open(image_path)
In[5]: print(type(img))
<class 'PIL.JpegImagePlugin.JpegImageFile'>
```

利用numpy.array()，对PIL图片进行转换

```
Shape:
    img_tensor: Default is :math:`(3, H, W)`. You can use ``torchvision.utils.make_grid()`` to
    convert a batch of tensor into 3xHxW format or call ``add_images`` and let us do the job.
    Tensor with :math:`(1, H, W)`, :math:`(H, W)`, :math:`(H, W, 3)` is also suitible as long as
    corresponding ``dataformats`` argument is passed. e.g. CHW, HWC, HW.
```

```python
from torch.utils.tensorboard import SummaryWriter
from PIL import Image
import numpy as np

writer = SummaryWriter("logs")

image_path = "data\\train\\ants_image\\0013035.jpg"
img_PIL = Image.open(image_path)
img = np.array(img_PIL)

writer.add_image("sguanTest",img,1,dataformats='HWC')
# y = 2x
# for i in range(100):
#     writer.add_scalar("y=2x",2*i,i)

writer.close()
```



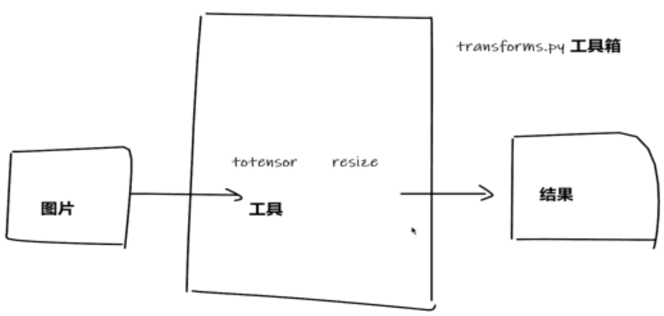
4.Torchvision的使用（transform）

torchvision 中的 transforms

```python
class FaceLandmarksDataset(Dataset):
    """Face Landmarks dataset."""

    def __init__(self, csv_file, root_dir, transform=None):
        """
        Args:
            csv_file (string): Path to the csv file with annotations.
            root_dir (string): Directory with all the images.
            transform (callable, optional): Optional transform to be applied
                on a sample.
        """
        self.landmarks_frame = pd.read_csv(csv_file)
        self.root_dir = root_dir
        self.transform = transform
```

```python
def __getitem__(self, index):
    path = self.imgs[index]
    img = self.loader(path)
    if self.transform is not None:
        img = self.transform(img)
    if self.return_paths:
        return img, path
    else:
```

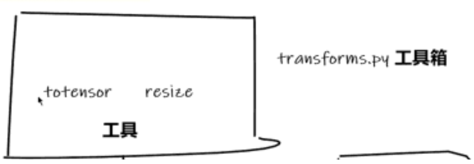
transforms结构及用法



- Compose(object)
- ToTensor(object)
- ToPILImage(object) 图片格式转换
- Normalize(object) 正则化
- Resize(object) 图片尺寸变换
- CenterCrop(object) 中心对称

normalized是归一化，regularization是正则化，是添加惩罚函数改善优化问题目标函数

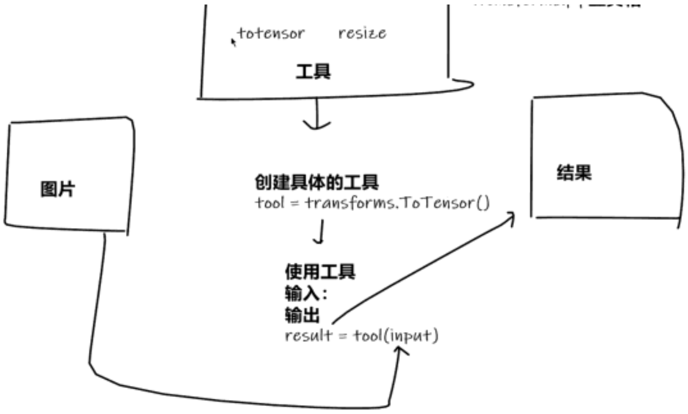
```python
# python的用法 -》 tensor数据类型
# 通过 transforms.ToTensor去看两个问题
# 1、 transforms该如何使用 (python)
# 2、为什么我们需要Tensor数据类型
```




```python
class ToPILImage(object):
    """Convert a tensor or an ndarray to PIL Image.

    Converts a torch.*Tensor of shape C x H x W or a numpy ndarray of shape
    H x W x C to a PIL Image while preserving the value range.

    Args:
        mode (`PIL.Image mode`_): color space and pixel depth of input data (optional).
            If ``mode`` is ``None`` (default) there are some assumptions made about the input data:
             - If the input has 4 channels, the ``mode`` is assumed to be ``RGBA``.
             - If the input has 3 channels, the ``mode`` is assumed to be ``RGB``.
             - If the input has 2 channels, the ``mode`` is assumed to be ``LA``.
             - If the input has 1 channel, the ``mode`` is determined by the data type (i.e ``int``, ``float``,
               ``short``).

    .. _PIL.Image mode: https://pillow.readthedocs.io/en/latest/handbook/concepts.html#concept-modes
    """
    def __init__(self, mode=None):
        self.mode = mode

    def __call__(self, pic):
        """
        Args:
            pic (Tensor or numpy.ndarray): Image to be converted to PIL Image.

        Returns:
            PIL Image: Image converted to PIL Image.

        """
        return F.to_pil_image(pic, self.mode)

class Compose(object):
    """Composes several transforms together.

    Args:
        transforms (list of ``Transform`` objects): list of transforms to compose.

    Example:
        >>> transforms.Compose([
        >>>     transforms.CenterCrop(10),
        >>>     transforms.ToTensor(),
        >>> ])
    """

    def __init__(self, transforms):
        self.transforms = transforms

    def __call__(self, img):
        for t in self.transforms:
            img = t(img)
        return img

    def __repr__(self):
        format_string = self.__class__.__name__ + '('
        for t in self.transforms:
            format_string += '\n'
            format_string += '    {0}'.format(t)
        format_string += '\n)'
        return format_string
```

```python
from torchvision import transforms
from PIL import Image

img_path = "dataset\\train\\ants\\0013035.jpg"
img = Image.open(img_path)

tensor_trans = transforms.ToTensor()
tensor_img = tensor_trans(img)

print(tensor_img)
```

```
(pytorch) E:\AE\Stm32 MDKcode\test_TrainPytorch>E:/ProgramData/anaconda3/
orch/Test_TransForms.py"
tensor([[[0.3137, 0.3137, 0.3137,  ..., 0.3176, 0.3098, 0.2980],
         [0.3176, 0.3176, 0.3176,  ..., 0.3176, 0.3098, 0.2980],
         [0.3216, 0.3216, 0.3216,  ..., 0.3137, 0.3098, 0.3020],
```




```
print(tensor_img)
(pytorch) E:\AE\Stm32 MDKcode\test_TrainPytorch>E:/ProgramData/anaconda3/
orch/Test_TransForms.py"
tensor([[[0.3137, 0.3137, 0.3137,  ..., 0.3176, 0.3098, 0.2980],
         [0.3176, 0.3176, 0.3176,  ..., 0.3176, 0.3098, 0.2980],
         [0.3216, 0.3216, 0.3216,  ..., 0.3137, 0.3098, 0.3020],
         ...,
         [0.3412, 0.3412, 0.3373,  ..., 0.1725, 0.3725, 0.3529],
         [0.3412, 0.3412, 0.3373,  ..., 0.3294, 0.3529, 0.3294],
         [0.3412, 0.3412, 0.3373,  ..., 0.3098, 0.3059, 0.3294]],

        [[0.5922, 0.5922, 0.5922,  ..., 0.5961, 0.5882, 0.5765],
         [0.5961, 0.5961, 0.5961,  ..., 0.5961, 0.5882, 0.5765],
         [0.6000, 0.6000, 0.6000,  ..., 0.5922, 0.5882, 0.5804],
         ...,
         [0.6275, 0.6275, 0.6235,  ..., 0.3608, 0.6196, 0.6157],
         [0.6275, 0.6275, 0.6235,  ..., 0.5765, 0.6275, 0.5961],
         [0.6275, 0.6275, 0.6235,  ..., 0.6275, 0.6235, 0.6314]],

        [[0.9137, 0.9137, 0.9137,  ..., 0.9176, 0.9098, 0.8980],
         [0.9176, 0.9176, 0.9176,  ..., 0.9176, 0.9098, 0.8980],
         [0.9216, 0.9216, 0.9216,  ..., 0.9137, 0.9098, 0.9020],
         ...,
         [0.9294, 0.9294, 0.9255,  ..., 0.5529, 0.9216, 0.8941],
         [0.9294, 0.9294, 0.9255,  ..., 0.8863, 1.0000, 0.9137],
         [0.9294, 0.9294, 0.9255,  ..., 0.9490, 0.9804, 0.9137]]])
```

4.Torchvision的使用（为什么要使用Tensor类型）

```
tensor_img = (Tensor) tensor([[[0.3137, 0.3137, 0.3137, ..., 0.3176, 0.3098, 0.2980],\n [0.3176, 0.3176, 0.3176, ..., 0.3176, 0.3098, 0.2980],\n [0.3216, 0.3216, 0.3216, ... View
  T = (Tensor) tensor([[[0.3137, 0.5922, 0.9137],\n [0.3176, 0.5961, 0.9176],\n [0.3216, 0.6000, 0.9216],\n ...\n [0.3412, 0.6275, 0.9294],\n [0.3412, ... View
  _backward_hooks = (NoneType) None
  _base = (NoneType) None
  _cdata = (int) 2501405032288
  _grad = (NoneType) None
  _grad_fn = (NoneType) None
  _version = (int) 0
  data = (Tensor) tensor([[[0.3137, 0.3137, 0.3137, ..., 0.3176, 0.3098, 0.2980],\n [0.3176, 0.3176, 0.3176, ..., 0.3176, 0.3098, 0.2980],\n [0.3216, 0.3216, 0.3216, ... View
  device = (device) cpu
  dtype = (dtype) torch.float32
  grad = (NoneType) None
  grad_fn = (NoneType) None
  is_cuda = (bool) False
  is_leaf = (bool) True
  is_mkldnn = (bool) False
  is_quantized = (bool) False
  is_sparse = (bool) False
  layout = (layout) torch.strided
  name = (NoneType) None
  names = (tuple) (None, None, None)
  ndim = (int) 3
  output_nr = (int) 0
  requires_grad = (bool) False
  shape = (Size) torch.Size([3, 512, 768])
```

Tensor这个类型就是包含了我们反向神经网络理论基础的一个参数

```python
from torchvision import transforms
from PIL import Image
from torch.utils.tensorboard import SummaryWriter

img_path = "dataset\\train\\ants\\0013035.jpg"
img = Image.open(img_path)

writer = SummaryWriter("logs")


tensor_trans = transforms.ToTensor()
tensor_img = tensor_trans(img)

# print(tensor_img)
writer.add_image("Tensor_img",tensor_img)
writer.close()
```

```python
import cv2
cv_img = cv2.imread(img_path)
```

5.常见的Transforms函数

* 输入 * PIL * Image.open()
* 输出 * tensor * ToTensor()
* 作用 * narrays * cv.imread()

Python 中 __call__ 的用法

```python
class Person:
    def __call__(self, name):
        print("__call__"+" Hello " + name)

    def hello(self, name):
        print("hello"+ name)


person = Person()
person("zhangsan")
person.hello("lisi")
```

```
C:\Users\Zhiyao\Anaconda3\
__call__ Hello zhangsan
hellolisi
```

```python
class Normalize(object):
    """Normalize a tensor image with mean and standard deviation.
    Given mean: ``(M1,...,Mn)`` and std: ``(S1,..,Sn)`` for ``n`` channels, this transform
    will normalize each channel of the input ``torch.*Tensor`` i.e.
    ``input[channel] = (input[channel] - mean[channel]) / std[channel]``

    .. note::
        This transform acts out of place, i.e., it does not mutates the input tensor.
```

Normalize归一化，此处公式为正太分布标准化

```python
# Normalize
print(img_tensor[0][0][0])
trans_norm = transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
img_norm = trans_norm(img_tensor)
print(img_norm[0][0][0])
```

```
tensor(0.8275)
tensor(0.6549)
```

Compose([transforms参数1， transforms参数2,...])

```python
from torch.utils.tensorboard import SummaryWriter
from PIL import Image
from torchvision import transforms

writer = SummaryWriter("logs")
img = Image.open("dataset\\train\\ants\\6743948_2b8c096dda.jpg")

# ToTensor类型转换
trans_totensor = transforms.ToTensor()
img_tensor = trans_totensor(img)
writer.add_image("ToTensor",img_tensor)

# Normalize归一化
trans_norm = transforms.Normalize([0.5,0.5,0.5], [0.5,0.5,0.5]) # 提供3个标准差
img_norm = trans_norm(img_tensor)
writer.add_image("Normalize",img_norm)

# Resize强行修改图片尺寸
trans_resize = transforms.Resize((512,512))
# img PIL -> resize -> img_resize PIL
img_resize = trans_resize(img)
# img_resize -> totensor -> img_resize tensor
img_resize = trans_totensor(img_resize)
writer.add_image("Resize",img_resize)

# Compose-resize-2 仅对图片高缩减到512,使用的是等比压缩
trans_resize_2 = transforms.Resize(512)
trans_compose = transforms.Compose([trans_resize_2,trans_totensor])
img_resize_2 = trans_compose(img)
writer.add_image("Resize",img_resize_2,1)

# RandomCrop随机裁剪一个指定尺寸大小的图片
trans_random = transforms.RandomCrop(100)
trans_compose_2 = transforms.Compose([trans_random,trans_totensor])
for i in range(10):
    img_random = trans_compose_2(img)
    writer.add_image("RandomCrop",img_random,i)


writer.close()
```

关注输入和输出类型
多看官方文档
关注方法需要什么参数

不知道返回值的时候
* print
* print(type())
* debug

6.torchvision的数据集使用

PyTorch 核心模块
torchaudio 处理语音
torchtext 处理文字
torchvision 处理视觉
TorchElastic
TorchServe
PyTorch on XLA Devices

The CIFAR-10 dataset CIFAR是主要用于物体识别的

The CIFAR-10 dataset consists of 60000 32x32 colour images i

The dataset is divided into five training batches and one test b
some training batches may contain more images from one cla

Here are the classes in the dataset, as well as 10 random imag

airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck

* torchvision.models torchvision构建目标检测的模块
    * Classification
    * Semantic Segmentation
    * Object Detection, Instance Segmentation and Person Keypoint Detection
    * Video classification

```python
from torch.utils.data import Dataset
import torchvision
from torch.utils.tensorboard import SummaryWriter
from PIL import Image

trans_dataset = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor()
])

trans_set = torchvision.datasets.CIFAR10(root=".\\dataset",train=True,transform=trans_dataset,download=True)
test_set = torchvision.datasets.CIFAR10(root=".\\dataset",train=False,transform=trans_dataset,download=True)

# print(test_set[0])
writer = SummaryWriter("logs")
for i in range(10):
    img,target = test_set[i]
    writer.add_image("test_set",img,i)

writer.close()
```

Captions

CLASS `torchvision.datasets.CocoCaptions(root: str, annFile: str, transform: Union[Callable, NoneType] = None, target_transform: Union[Callable, NoneType] = None, transforms: Union[Callable, NoneType] = None)` [SOURCE]

MS Coco Captions Dataset.

Parameters:
* **root** (*string*) – Root directory where images are downloaded to.
* **annFile** (*string*) – Path to json annotation file.
* **transform** (*callable, optional*) – A function/transform that takes in an PIL image and returns a transformed version. E.g, `transforms.ToTensor`
* **target_transform** (*callable, optional*) – A function/transform that takes in the target and transforms it.
* **transforms** (*callable, optional*) – A function/transform that takes input sample and its target as entry and returns a transformed version.

Example

```python
import torchvision.datasets as dset
import torchvision.transforms as transforms
cap = dset.CocoCaptions(root = 'dir where images are',
                        annFile = 'json annotation file',
                        transform=transforms.ToTensor())

print('Number of samples: ', len(cap))
img, target = cap[3] # load 4th sample

print("Image Size: ", img.size())
```

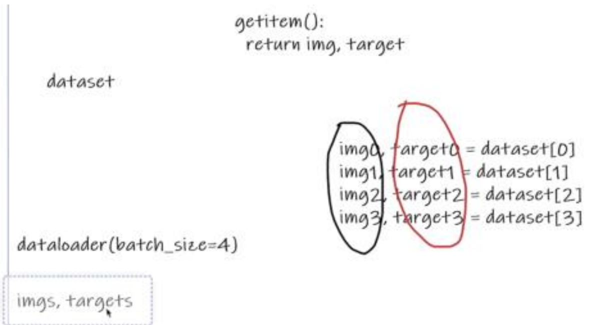
```python
writer = SummaryWriter("logs")
for i in range(10):
    img,target = test_set[i]
    writer.add_image("test_set",img,i)

writer.close()
```

```python
                      transform=transforms.ToTensor())

print('Number of samples: ', len(cap))
img, target = cap[3] # load 4th sample

print("Image Size: ", img.size())
print(target)
```

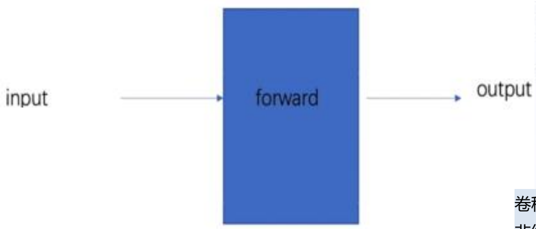
7.dataloader的使用（神经网络的加载器）

- **dataset** (*Dataset*) – dataset from which to load the data.
- **batch_size** (*int, optional*) – how many samples per batch to load (default: `1`).
- **shuffle** (*bool, optional*) – set to `True` to have the data reshuffled at every epoch (default: `False`).
- **sampler** (*Sampler or Iterable, optional*) – defines the strategy to draw samples from the dataset. Can be any `Iterable` with `__len__` implemented. If specified, `shuffle` must not be specified.
- **batch_sampler** (*Sampler or Iterable, optional*) – like `sampler`, but returns a batch of indices at a time. Mutually exclusive with `batch_size`, `shuffle`, `sampler`, and `drop_last`.
- **num_workers** (*int, optional*) – how many subprocesses to use for data loading. `0` means that the data will be loaded in the main process. (default: `0`)
- **collate_fn** (*callable, optional*) – merges a list of samples to form a mini-batch of Tensor(s). Used when using batched loading from a map-style dataset.
- **pin_memory** (*bool, optional*) – If `True`, the data loader will copy Tensors into CUDA pinned memory before returning them. If your data elements are a custom type, or your `collate_fn` returns a batch that is a custom type, see the example below.
- **drop_last** (*bool, optional*) – 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: `False`)
- **timeout** (*numeric, optional*) – if positive, the timeout value for collecting a batch from workers. Should always be non-negative. (default: `0`)
- **worker_init_fn** (*callable, optional*) – 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: `None`)
- **prefetch_factor** (*int, optional, keyword-only arg*) – 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: `2`)
- **persistent_workers** (*bool, optional*) – 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: `False`)

```python
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter
import torchvision


# 准备的一个测试集
test_data = torchvision.datasets.CIFAR10(root=".\\dataset",train=False,transform=torchvision.transforms.ToTensor())
test_loader = DataLoader(dataset=test_data,batch_size=4,shuffle=True,num_workers=0,drop_last=False)
# batch_size是指每次取数据多少个
# drop_last最后几张不足batch_size的时候，是否舍去
# shuffle是否要打乱每次取数据的顺序

# 测试数据集中第一张图片及targe
img,targe = test_data[0]
print(img.shape)
print(targe)

writer = SummaryWriter("logs")

#DataLoader中取样本数据
step = 0
for data in test_loader:
    imgs,targes = data
    # print(imgs.shape)
    # print(targes)
    writer.add_images("test_data",imgs,step)
    step += 1

writer.close()
```



8.神经网络的基本骨架nn.Module的使用




```python
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        return F.relu(self.conv2(x))
```

卷积

非线性处理

前向传播层

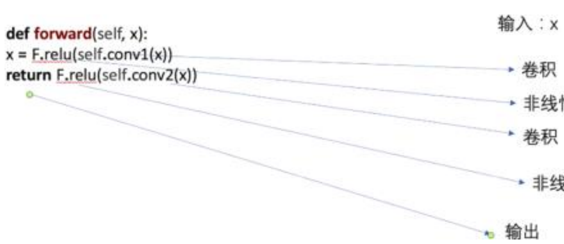
火炬.nn

- 器皿
- 卷积层
- 池化层
- 填充层
- 非线性激活 （加权和、非线性）
- 非线性激活 （其他）
- 归一化图层
- 循环层
- 变压器层
- 线性层
- Dropout 图层
- 稀疏层
- 距离函数
- 损失函数
- Vision Layers
- 随机排列图层
- DataParallel Layers（多 GPU，分布式）
- 公用事业
- 量化函数
- 惰性模块初始化
  - 别名

torch.nn

- Containers
- Convolution Layers
- Pooling layers
- Padding Layers
- Non-linear Activations (weighted sum, nonlinearity)
- Non-linear Activations (other)
- Normalization Layers
- Recurrent Layers
- Transformer Layers
- Linear Layers
- Dropout Layers
- Sparse Layers
- Distance Functions
- Loss Functions
- Vision Layers
- Shuffle Layers
- DataParallel Layers (multi-GPU, distributed)
- Utilities
- Quantized Functions
- Lazy Modules Initialization
  - Aliases

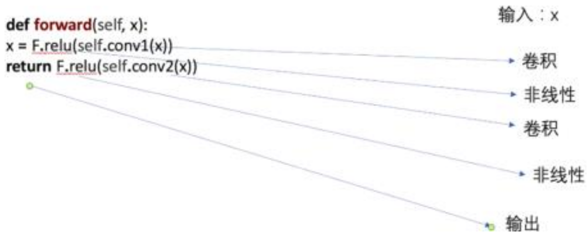
```python
import torch
from torch import nn

class Sguan(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self,input):
        output = input + 1
        return output

sguan = Sguan()
x = torch.tensor(1,0)
output = sguan(x)
print(output)
# print神经网络输出
```



- 非线性激活（其他）
- 归一化图层
- 循环层
- 变压器层
- 线性层
- Dropout 图层
- 稀疏层
- 距离函数
- 损失函数
- Vision Layers
- 随机排列图层
- DataParallel Layers（多 GPU，分布式）
- 公用事业
- 量化函数
- 惰性模块初始化
  - 别名

- Non-linear Activations (other)
- Normalization Layers
- Recurrent Layers
- Transformer Layers
- Linear Layers
- Dropout Layers
- Sparse Layers
- Distance Functions
- Loss Functions
- Vision Layers
- Shuffle Layers
- DataParallel Layers (multi-GPU, distributed)
- Utilities
- Quantized Functions
- Lazy Modules Initialization
  - Aliases

```python
import torch
from torch import nn

class Sguan(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self,input):
        output = input + 1
        return output

sguan = Sguan()
x = torch.tensor(1,0)
output = sguan(x)
print(output)
# print神经网络输出
```



9.神经网络CNN卷积操作

conv2d

`torch.nn.functional.conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1) → Tensor`

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

This operator supports TensorFloat32.

See `Conv2d` for details and output shape.

NOTE

In some circumstances when given tensors on a CUDA device and using CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting `torch.backends.cudnn.deterministic = True`. See Reproducibility for more information.

Parameters

- **input** – input tensor of shape $(\text{minibatch}, \text{in\_channels}, iH, iW)$
- **weight** – filters of shape $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, kH, kW)$
- **bias** – optional bias tensor of shape (out_channels). Default: `None`
- **stride** – the stride of the convolving kernel. Can be a single number or a tuple *(sH, sW)*. Default: 1
- **padding** – implicit paddings on both sides of the input. Can be a single number or a tuple *(padH, padW)*. Default: 0
- **dilation** – the spacing between kernel elements. Can be a single number or a tuple *(dH, dW)*. Default: 1
- **groups** – split input into groups, in_channels should be divisible by the number of groups. Default: 1

输入，权重(卷积核)，偏置，步径

CONV2D

CLASS `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')` [SOURCE]

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

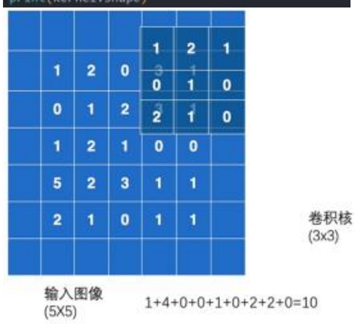
In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

```python
import torch
import torch.nn.functional as F

img_input = torch.tensor([[1,2,0,3,1],
                          [0,1,2,3,1],
                          [1,2,1,0,0],
                          [5,2,3,1,1],
                          [2,1,0,1,1]])

kernel = torch.tensor([[1,2,1],
                       [0,1,0],
                       [2,1,0]])

# 图片重定义reshap
img_input = torch.reshape(img_input,(1,1,5,5))
kernel = torch.reshape(kernel,(1,1,3,3))

# 打印shape参数，方便我们观看
print(img_input.shape)
print(kernel.shape)
```

(Input+2P-K/S) +1可以计算输出的大小



1+4+0+0+1+0+2+2+0=10

2+0+3+0+2+0+4+1=12

0+6+1+0+3+0+2+0+0=12

- **padding** – implicit paddings on both sides of the input. Can be a single number or a tuple *(padH, padW)*. Default: 0

Padding图像填充

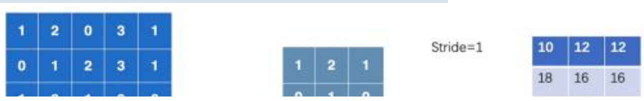
为什么需要填充？答：如果不填充白边，那么边缘的图像特征就会缺失

10.神经网络CNN卷积层

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

CLASS `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')`

输入图片的通道数，输出图片的期望通道数，卷积核的大小"(3"就是"3x3"的)，步径，填充数...



径，填充数...

CLASS `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros')`

in_channels (*int*) – Number of channels in the input image

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

kernel_size (*int or tuple*) – Size of the convolving kernel

stride (*int or tuple, optional*) – Stride of the convolution. Default: 1

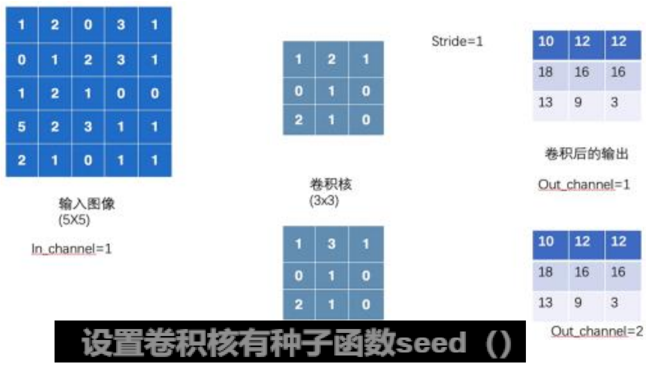
padding (*int or tuple, optional*) – Zero-padding added to both sides of the input. Default: 0

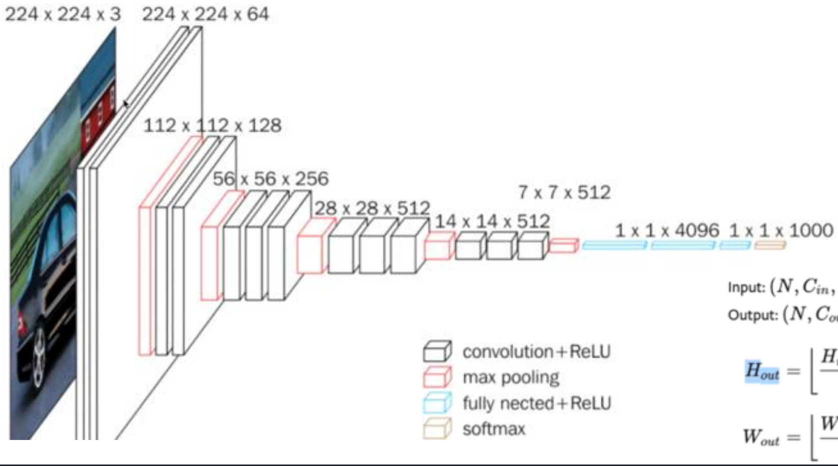
padding_mode (*string, optional*) – `'zeros'`, `'reflect'`, `'replicate'` or `'circular'`. Default: `'zeros'`

dilation (*int or tuple, optional*) – Spacing between kernel elements. Default: 1

groups (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1

bias (*bool, optional*) – If `True`, adds a learnable bias to the output. Default: `True`





Input: $(N, C_{in}, H_{in}, W_{in})$

Output: $(N, C_{out}, H_{out}, W_{out})$ where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

```python
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter
import torch
import torchvision
from torch.nn import Conv2d
from torch import nn

dataset = torchvision.datasets.CIFAR10(root=".\\dataset",train=False,
                                       transform=torchvision.transforms.ToTensor(),download=True)

dataloader = DataLoader(dataset,batch_size=64,shuffle=True)

# 卷积层CNN函数搭建
class Sguan(nn.Module):
    def __init__(self):
        super(Sguan,self).__init__()
        self.conv1 = Conv2d(in_channels=3,out_channels=6,kernel_size=3,stride=1,padding=1)

    def forward(self,x):
        x = self.conv1(x)
        return x

# 具体卷积操作CNN
sguan_cnn = Sguan()
Step = 0
writer = SummaryWriter("logs")
for data in dataloader:
    imgs,targes = data
    output = sguan_cnn(imgs)
    # print(imgs.shape)
    # print(output.shape)
    output_show = output[:, :3, :, :]  # 选择前3个通道
    writer.add_images("nn_input",imgs,Step)
    writer.add_images("nn_output",output_show,Step)
    Step += 1

writer.close()
```

11.神经网络池化层（示例为最大池化）

CLASS `torch.nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False)` [SOURCE]

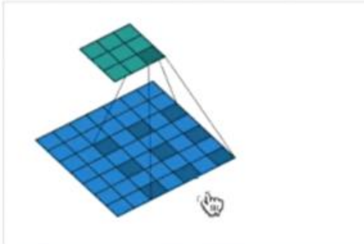
nn.MaxPool2d

nn.MaxUnpool2d

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

In the simplest case, the output value of the layer with input size (N, C, H, W), output (N, C, H_{out}, W_{out}) and `kernel_size` (kH, kW) can be precisely described as:

$$out(N_i, C_j, h, w) = \max_{m=0,\ldots,kH-1} \max_{n=0,\ldots,kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times h + m, \text{stride}[1] \times w + n)$$



No padding, no stride, dilation

kernel_size – the size of the window to take a max over

stride – the stride of the window. Default value is `kernel_size`

padding – implicit zero padding to be added on both sides

这个和卷积层CNN不同的是，stride默认是池化核的大小，而不是1

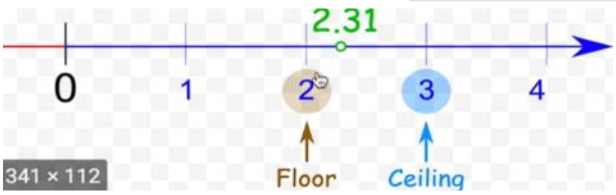
dilation – a parameter that controls the stride of elements in the window

return_indices – if `True`, will return the max indices along with the outputs. Useful for `torch.nn.MaxUnpool2d` later

如果说卷积是"提取重要特征"，那么池化就是一股脑的"压缩"

ceil_mode – when True, will use *ceil* instead of *floor* to compute the output shape

dilation是我们的一个空洞卷积



ceil_mode就是设置我们的一个向上取整or向下取整的一个方式

输入图像 (5X5)

最大池化操作

Ceil_model =True

Ceil_model =False

池化核(3x3), kernel_size=3

Shape:

- Input: (N, C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}), where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 * \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 * \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

在保留图片特征的同时，尽量降低图像的大小

```python
import torch
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter
from torch import nn
from torch.nn import MaxPool2d
import torchvision

# 1测试的数据集
dataset = torchvision.datasets.CIFAR10(root=".\\dataset",train=False,
                                       transform=torchvision.transforms.ToTensor(),download=True)
dataloader = DataLoader(dataset,batch_size=64,shuffle=True)

# 2自己创建的图像数据
img_input = torch.tensor([[1,2,0,3,1],
                          [0,1,2,3,1],
                          [1,2,1,0,0],
                          [5,2,3,1,1],
                          [2,1,0,1,1]])
# 2图片重定义reshap
img_input = torch.reshape(img_input,(1,1,5,5))

# 池化层 神经网络搭建
class Sguan(nn.Module):
    def __init__(self):
        super(Sguan,self).__init__()
        self.maxpool1 = MaxPool2d(kernel_size=3,ceil_mode=True)

    def forward(self,x):
        x = self.maxpool1(x)
        return x

# 2自己创建数据集的计算
sguan_maxpool = Sguan()
output = sguan_maxpool(img_input)
print(output)

# 1测试数据集的显示
Step = 0
```

```python
print(output)

# 1测试数据集的显示
Step = 0
writer = SummaryWriter("logs")
for data in dataloader:
    imgs,targes = data
    img_out = sguan_maxpool(imgs)
    writer.add_images("Maxpool_in",imgs,Step)
    writer.add_images("Maxpool_out",img_out,Step)
    Step += 1

writer.close()
```

12.神经网络（非线性激活层,增加图片对比度）

RELU

CLASS `torch.nn.ReLU(inplace=False)`

Applies the rectified linear unit function element-wise:

$$\mathrm{ReLU}(x) = (x)^+ = \max(0, x)$$

Parameters

inplace – can optionally do the operation in-place. Default: `False`

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

SIGMOID

CLASS `torch.nn.Sigmoid` [SOURCE]

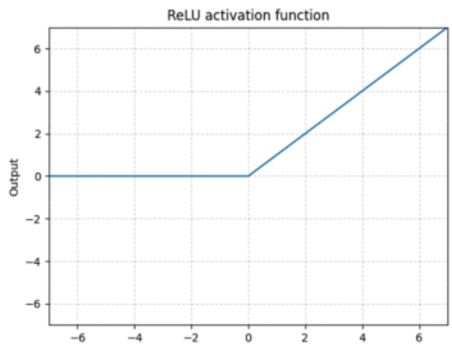
Applies the element-wise function:

$$\mathrm{Sigmoid}(x) = \sigma(x) = \frac{1}{1 + \exp(-x)}$$

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

RELU的折点处为钝角，sigmoid与之不同点在于折点处为光滑圆弧



Input = -1
Relu(input, inplace=True)
Input = 0

Input = -1
Output = Relu(input, inplace=False)
Input = -1
Output = 0

```python
import torch
from torch import nn
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter
# from torch.nn import ReLU
from torch.nn import Sigmoid
import torchvision

dataset = torchvision.datasets.CIFAR10(root=".\\dataset",train=False,
                                       transform=torchvision.transforms.ToTensor(),download=True)
dataloader = DataLoader(dataset,batch_size=64,shuffle=True)

input_relu = torch.tensor([[1,-0.5],
                           [-1,3]])

# 首个reshape参数中-1代表黑1代表白
input_relu = torch.reshape(input_relu,(1,1,2,2))

# 非线性激活层函数的搭建
class Sguan(nn.Module):
    def __init__(self):
        super(Sguan,self).__init__()
        # self.relu1 = ReLU()
        self.sigmoid1 = Sigmoid()

    def forward(self,x):
        x = self.sigmoid1(x)
        return x

sguan_sigmoid = Sguan()
# output = sguan_sigmoid(input_relu)

# 显示图像的训练集中
writer = SummaryWriter("logs")
Step = 0
for data in dataloader:
    imgs,targes = data
    img_out = sguan_sigmoid(imgs)
    writer.add_images("Sigmoid_in",imgs,Step)
    writer.add_images("Sigmoid_out",img_out,Step)
    Step += 1

writer.close()
```

12.神经网络其他层的介绍（正则化层，RNN循环层，线性层）

BATCHNORM2D

CLASS torch.nn.BatchNorm2d(*num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True*) [SOURCE]

Applies Batch Normalization over a 4D input (a mini-batch of 2D inputs with additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Recurrent Layers

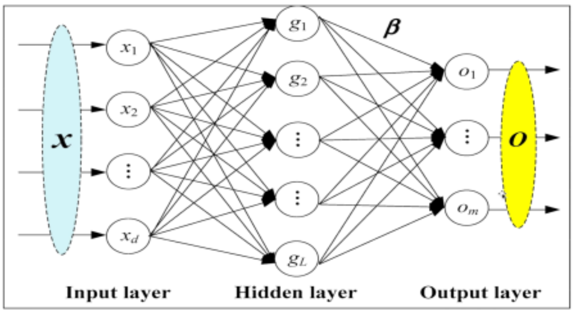
nn.RNNBase	
nn.RNN	Applies a multi-layer Elman RNN with tanh or ReLU non-linearity to an input sequence.
nn.LSTM	Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.
nn.GRU	Applies a multi-layer gated recurrent unit (GRU) RNN to an input sequence.
nn.RNNCell	An Elman RNN cell with tanh or ReLU non-linearity.
nn.LSTMCell	A long short-term memory (LSTM) cell.
nn.GRUCell	A gated recurrent unit (GRU) cell

Linear Layers

nn.Identity	A placeholder identity operator that is argument-insensitive.
nn.Linear	Applies a linear transformation to the incoming data: $y = xA^T + b$
nn.Bilinear	Applies a bilinear transformation to the incoming data: $y = x_1^T A x_2 + b$
nn.LazyLinear	A torch.nn.Linear module with lazy initialization.

- Containers
- Convolution Layers
- Pooling layers
- Padding Layers
- Non-linear Activations (weighted sum, nonlinearity)
- Non-linear Activations (other)
- Normalization Layers
- Recurrent Layers
- Transformer Layers
- Linear Layers
- Dropout Layers
- Sparse Layers

Linear Layers线性连接层（可能全连接）



LINEAR

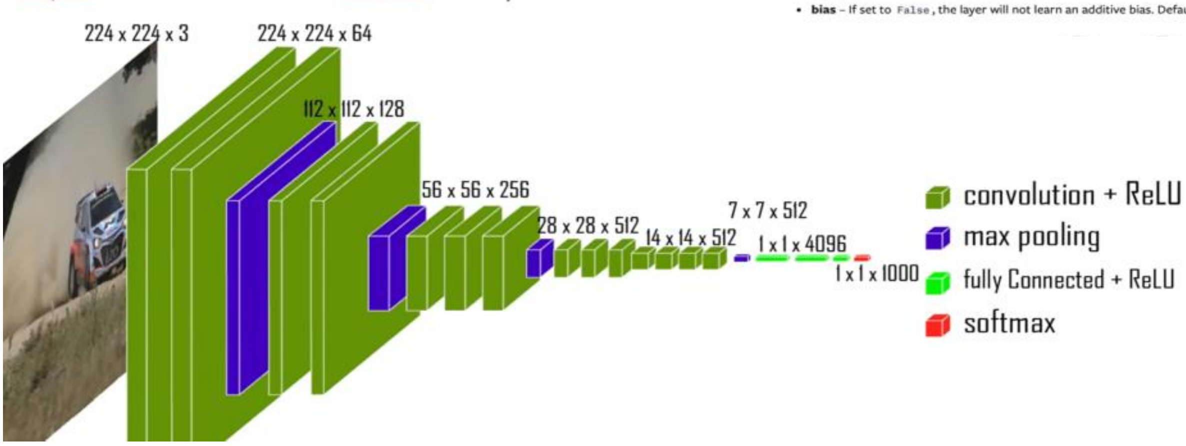
CLASS torch.nn.Linear(*in_features, out_features, bias=True*)

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

This module supports TensorFloat32.

Parameters

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to False , the layer will not learn an additive bias. Default: True



```python
import torch
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter
from torch import nn
from torch.nn import Linear
import torchvision

# 测试的数据集
dataset = torchvision.datasets.CIFAR10(root=".\\dataset",train=False,
                                       transform=torchvision.transforms.ToTensor(),download=True)
dataloader = DataLoader(dataset,batch_size=64,shuffle=True,drop_last=True)


class Sguan(nn.Module):
    def __init__(self):
        super(Sguan,self).__init__()
        self.linear1 = Linear(in_features=196608,out_features=10)

    def forward(self,x):
        x = self.linear1(x)
        return x

sguan_linear = Sguan()

for data in dataloader:
    imgs,targets = data
    # 方法一 使用重定义尺寸函数
    # 此处reshape参数的"-1"是指的"开启自适应模式"
    # img_output = torch.reshape(imgs,(1,1,1,-1))

    # 方法二 直接使用flatten函数展平图片
    img_output = torch.flatten(imgs) # flatten函数可以把图片展平
    print(img_output.shape)
    img_output = sguan_linear(img_output)
    print(img_output.shape)
```

13.神经网络搭建小实战和Sequential的使用

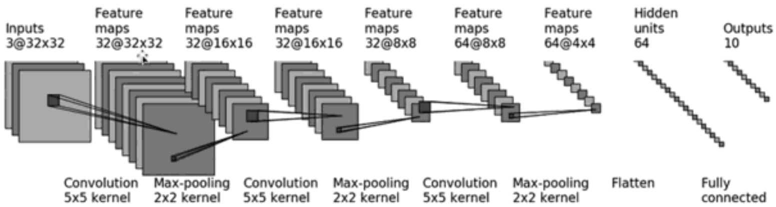
SEQUENTIAL

CLASS `torch.nn.Sequential(*args)` [SOURCE]

A sequential container. Modules will be added to it in the order they are passed in the constructor. Alternatively, an ordered dict of modules can also be passed in.

To make it easier to understand, here is a small example:

```python
# Example of using Sequential
model = nn.Sequential(
          nn.Conv2d(1,20,5),
          nn.ReLU(),
          nn.Conv2d(20,64,5),
          nn.ReLU()
        )

# Example of using Sequential with OrderedDict
model = nn.Sequential(OrderedDict([
          ('conv1', nn.Conv2d(1,20,5)),
          ('relu1', nn.ReLU()),
          ('conv2', nn.Conv2d(20,64,5)),
          ('relu2', nn.ReLU())
        ]))
```



Shape: 32+2*padding-4-1=27+2*padding=31, 2*padding= 4, padding=2 31

stride=1

对实际参数设置的计算（比较简单吧）

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

```python
import torch
from torch import nn
import torchvision
from torch.nn import Conv2d,MaxPool2d,Flatten,Linear,Sequential
from torch.utils.tensorboard import SummaryWriter

class Sguan(nn.Module):
    def __init__(self):
        super().__init__()
        self.module1 = Sequential(
            Conv2d(in_channels=3,out_channels=32,kernel_size=5,padding=2),
            MaxPool2d(kernel_size=2),
            Conv2d(in_channels=32,out_channels=32,kernel_size=5,padding=2),
            MaxPool2d(kernel_size=2),
            Conv2d(in_channels=32,out_channels=64,kernel_size=5,padding=2),
            MaxPool2d(kernel_size=2),
            Flatten(),
            Linear(in_features=1024,out_features=64),
```

```python
import torch
from torch import nn
import torchvision
from torch.nn import Conv2d,MaxPool2d,Flatten,Linear,Sequential
from torch.utils.tensorboard import SummaryWriter

class Sguan(nn.Module):
    def __init__(self):
        super().__init__()
        self.module1 = Sequential(
            Conv2d(in_channels=3,out_channels=32,kernel_size=5,padding=2),
            MaxPool2d(kernel_size=2),
            Conv2d(in_channels=32,out_channels=32,kernel_size=5,padding=2),
            MaxPool2d(kernel_size=2),
            Conv2d(in_channels=32,out_channels=64,kernel_size=5,padding=2),
            MaxPool2d(kernel_size=2),
            Flatten(),
            Linear(in_features=1024,out_features=64),
            Linear(in_features=64,out_features=10)
        )

    def forward(self,input_py):
        output = self.module1(input_py)
        return output

sguan_py = Sguan()
input_x = torch.ones((64,3,32,32))
output = sguan_py(input_x)
print(output.shape)

writer = SummaryWriter("logs")
writer.add_graph(model=sguan_py,input_to_model=input_x)
writer.close()
```

14.损失函数与反向传播（利用梯度下降，更新并减少Loss）

output	target
选择（10）	选择（30）
填空（10）	填空（20）
解答（20）	解答（50）

Loss=(30-10)+(20-10)+(50-10)=70

1. 计算实际输出和目标之间的差距
2. 为我们更新输出提供一定的依据（反向传播）

X:1, 2, 3
Y:1, 2, 5
L1loss = (0+0+2) /3=0.6

Loss Functions

nn.L1Loss	Creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y .
nn.MSELoss	Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y .
nn.CrossEntropyLoss	This criterion combines `LogSoftmax` and `NLLLoss` in one single class.
nn.CTCLoss	The Connectionist Temporal Classification loss.
nn.NLLLoss	The negative log likelihood loss.
nn.PoissonNLLLoss	Negative log likelihood loss with Poisson distribution of target.
nn.GaussianNLLLoss	Gaussian negative log likelihood loss.
nn.KLDivLoss	The Kullback-Leibler divergence loss measure
nn.BCELoss	Creates a criterion that measures the Binary Cross Entropy between the target and the output:

CLASS `torch.nn.L1Loss(size_average=None, reduce=None, reduction='mean')` [SOURCE]

Creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y .

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = |x_n - y_n|,$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then:

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{'mean';} \\ \operatorname{sum}(L), & \text{if reduction} = \text{'sum'.} \end{cases}$$

x and y are tensors of arbitrary shapes with a total of n elements each.

The sum operation still operates over all the elements, and divides by n .

The division by n can be avoided if one sets `reduction = 'sum'` .

Supports real-valued and complex-valued inputs.

Shape:

- Input: $(N, *)$ where $*$ means, any number of additional dimensions
- Target: $(N, *)$, same shape as the input
- Output: scalar. If `reduction` is `'none'` , then $(N, *)$, same shape as the input

```python
import torch
from torch.nn import L1Loss

inputs = torch.tensor([1,2,3],dtype=torch.float32)
targets = torch.tensor([1,2,5],dtype=torch.float32)

inputs = torch.reshape(inputs,(1,1,1,3))
targets = torch.reshape(targets,(1,1,1,3))

loss = L1Loss()
result = loss(inputs,targets)
print(result)
```

```python
loss = L1Loss(reduce='sum')
result = loss(inputs,targets)
print(result)
```

MSELOSS

CLASS `torch.nn.MSELoss(size_average=None, reduce=None, reduction='mean')` [SOURCE]

Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = (x_n - y_n)^2,$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then:

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \operatorname{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

Shape:

- Input: $(N, *)$ where $*$ means, any number of additional dimensions
- Target: $(N, *)$, same shape as the input

```python
import torch
from torch.nn import MSELoss

inputs = torch.tensor([1,2,3],dtype=torch.float32)
targets = torch.tensor([1,2,5],dtype=torch.float32)

inputs = torch.reshape(inputs,(1,1,1,3))
targets = torch.reshape(targets,(1,1,1,3))

loss = MSELoss(reduce='sum')
result = loss(inputs,targets)
print(result)
```

X:1, 2, 3
Y:1, 2, 5
L1loss = (0+0+2) /3=0.6
MSE = (0+0+2^2)/3=4/3=1.333

CLASS `torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean')` [SOURCE]

This criterion combines `LogSoftmax` and `NLLLoss` in one single class.

It is useful when training a classification problem with C classes. If provided, the optional argument `weight` should be a 1D *Tensor* assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* is expected to contain raw, unnormalized scores for each class.

input has to be a Tensor of size either $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case (described later).

This criterion expects a class index in the range $[0, C-1]$ as the *target* for each value of a 1D tensor of size *minibatch*; if *ignore_index* is specified, this criterion also accepts this class index (this index may not necessarily be in the class range).

The loss can be described as:

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right)$$

or in the case of the `weight` argument being specified:

$$\text{loss}(x, class) = weight[class]\left(-x[class] + \log\left(\sum_j \exp(x[j])\right)\right)$$

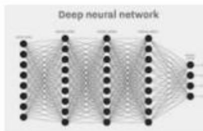
The losses are averaged across observations for each minibatch. If the `weight` argument is specified then this is a weighted average:

$$\text{loss} = \frac{\sum_{i=1}^{N} loss(i, class[i])}{\sum_{i=1}^{N} weight[class[i]]}$$

The loss can be described as:

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right)$$

Person, dog, cat
0, 1, 2



output
[0.1, 0.2, 0.3]

x

Target
1
class

Loss(x, class) = -0.2+log(exp(0.1)+exp(0.2)+exp(0.3))

Shape:

- Input: (N, C) where C = *number of classes*, or $(N, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ in the case of K-dimensional loss.
- Target: (N) where each value is $0 \leq \text{targets}[i] \leq C-1$, or $(N, d_1, d_2, ..., d_K)$ with $K \geq 1$ in the case of K-dimensional loss.
- Output: scalar. If `reduction` is `'none'`, then the same size as the target: (N), or $(N, d_1, d_2, ..., d_K)$ with $K \geq 1$ in the case of K-dimensional loss.

```python
x = torch.tensor([0.1,0.2,0.3])
y = torch.tensor([1])
x = torch.reshape(x,(1,3))

loss = CrossEntropyLoss()
result = loss(x,y)
print(result)
```

```python
x = torch.tensor([0.1,0.2,0.3])
y = torch.tensor([1])
x = torch.reshape(x,(1,3))

loss = CrossEntropyLoss()
result = loss(x,y)
print(result)
```


```python
import torch
from torch import nn
import torchvision
from torch.nn import Conv2d,MaxPool2d,Flatten,Linear,Sequential
from torch.utils.tensorboard import SummaryWriter
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.nn import CrossEntropyLoss

# 测试的数据集
dataset = torchvision.datasets.CIFAR10(root=".\\dataset",train=False,
                                       transform=torchvision.transforms.ToTensor(),download=True)
dataloader = DataLoader(dataset,batch_size=64,shuffle=True)

class Sguan(nn.Module):
    def __init__(self):
        super().__init__()
        self.module1 = Sequential(
            Conv2d(in_channels=3,out_channels=32,kernel_size=5,padding=2),
            MaxPool2d(kernel_size=2),
            Conv2d(in_channels=32,out_channels=32,kernel_size=5,padding=2),
            MaxPool2d(kernel_size=2),
            Conv2d(in_channels=32,out_channels=64,kernel_size=5,padding=2),
            MaxPool2d(kernel_size=2),
            Flatten(),
            Linear(in_features=1024,out_features=64),
            Linear(in_features=64,out_features=10)
        )

    def forward(self,input_py):
        output = self.module1(input_py)
        return output

loss = CrossEntropyLoss()
sguan_loss = Sguan()
for data in dataloader:
    imgs,targets = data
    img_out = sguan_loss(imgs)
    # print(img_out)
    result = loss(img_out,targets)
    # print(result)
    result.backward()
```

15.神经网络（优化器） TORCH.OPTIM

Example:

```python
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
optimizer = optim.Adam([var1, var2], lr=0.0001)
```

Taking an optimization step

All optimizers implement a `step()` method, that updates the parameters. It can be used in two ways:

CLASS `torch.optim.Adadelta(params, lr=1.0, rho=0.9, eps=1e-06, weight_decay=0)` [SOURCE]

Implements Adadelta algorithm.

It has been proposed in ADADELTA: An Adaptive Learning Rate Method.

Parameters

Taking an optimization step

All optimizers implement a `step()` method, that updates the parameters. It can be used in two ways:

```
optimizer.step()
```

This is a simplified version supported by most optimizers. The function can be called once the gradients are computed using e.g. `backward()`.

Example:

```
for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

It has been proposed in ADADELTA: An Adaptive Learning Rate Method.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **rho** (*float, optional*) – coefficient used for computing a running average of squared gradients (default: 0.9)
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-6)
- **lr** (*float, optional*) – coefficient that scale delta before it is applied to the parameters (default: 1.0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)

```
training = {bool} True
transposed = {bool} False
weight = {Parameter: 32} Parameter containing:\ntensor([[[[ 0.0710,  0.0407, -0.0205, -0.0488,  0.0558],\n      [-0.0867, -0.0948, -0.0707, -0.0859, -
    T = {Tensor: 5} tensor([[[[ 0.0710,  0.0747, -0.0959,  ...,  0.0405, -0.0039, -0.0363],\n      [ 0.1137, -0.1059,  0.0557,  ...,  0.0273, -0.0976,  0.0050],
    data = {Tensor: 32} tensor([[[[ 0.0710,  0.0407, -0.0205, -0.0488,  0.0558],\n      [-0.0867, -0.0948, -0.0707, -0.0859, -0.0357],\n      [-0.0697,
    device = {device} cpu
training = {bool} True
transposed = {bool} False
weight = {Parameter: 32} Parameter containing:\ntensor([[[[ 0.0709,  0.0406, -0.0206, -0.0489,  0.0557],\n      [-0.0868, -0.0949, -0.0708, -0.0860, -0.0358],\n      [-0.0698
    T = {Tensor: 5} tensor([[[[ 0.0709,  0.0748, -0.0959,  ...,  0.0404, -0.0039, -0.0364],\n      [ 0.1135, -0.1059,  0.0556,  ...,  0.0272, -0.0976,  0.0049],\n      [ 0.0030, -0.059
    data = {Tensor: 32} tensor([[[[ 0.0709,  0.0406, -0.0206, -0.0489,  0.0557],\n      [-0.0868, -0.0949, -0.0708, -0.0860, -0.0358],\n      [-0.0698,  0.1056, -0.1086,  0.1127,
    device = {device} cpu
    dtype = {dtype} torch.float32
    grad = {Tensor: 32} tensor([[[[ 1.1940e-02,  1.2346e-02,  1.3659e-02,  1.3445e-02,  1.3287e-02],\n      [ 1.0455e-02,  1.0619e-02,  1.1651e-02,  1.3074e-02,  1.3425e-02],\n
        grad_fn = {NoneType} None
        imag = {str} 'Traceback (most recent call last):\n  File "/Applications/PyCharm CE.app/Contents/plugins/python-ce/helpers/pydev/_pydevd_bundle/pydevd_resolver.py", line 178
        is_cuda = {bool} False
```

nn_optim

```
/Users/xiaotudui/.conda/envs/pytorch/bin/python /Users/xiaotudui/pytorch-tutorial/src/nn_optim.py
Files already downloaded and verified
tensor(18709.0840, grad_fn=<AddBackward0>)
tensor(16215.9258, grad_fn=<AddBackward0>)
tensor(15501.3164, grad_fn=<AddBackward0>)
```

```python
import torch
from torch import nn
from torch import optim
import torchvision
from torch.nn import Conv2d,MaxPool2d,Flatten,Linear,Sequential,CrossEntropyLoss
from torch.utils.tensorboard import SummaryWriter
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.optim import SGD

# 测试的数据集
dataset = torchvision.datasets.CIFAR10(root=".\\dataset",train=False,
                                       transform=torchvision.transforms.ToTensor(),download=True)
dataloader = DataLoader(dataset,batch_size=64,shuffle=True)

# 面向于CIFAR10的神经网络训练函数（3, 32x32）->（64, 1x10）
class Sguan(nn.Module):
    def __init__(self):
        super().__init__()
        self.module1 = Sequential(
            Conv2d(in_channels=3,out_channels=32,kernel_size=5,padding=2),
            MaxPool2d(kernel_size=2),
            Conv2d(in_channels=32,out_channels=32,kernel_size=5,padding=2),
            MaxPool2d(kernel_size=2),
            Conv2d(in_channels=32,out_channels=64,kernel_size=5,padding=2),
            MaxPool2d(kernel_size=2),
            Flatten(),
            Linear(in_features=1024,out_features=64),
            Linear(in_features=64,out_features=10)
        )

    def forward(self,input_py):
        output = self.module1(input_py)
        return output

loss = CrossEntropyLoss()
sguan_py = Sguan()
sguan_optim = torch.optim.SGD(sguan_loss.parameters(),lr=0.01,)
for epoch in range(20):
    result_loss = 0.0
    for data in dataloader:
        imgs,targets = data
        # 1.卷积和池化
        img_out = sguan_py(imgs)
        result = loss(img_out,targets)

        # 2.优化器需要把梯度定义为零
        sguan_optim.zero_grad()

        # 3.输出结果需要一个反向传播,用到了backward内置函数
        result.backward()

        # 4.使用优化器优化卷积核参数
        sguan_optim.step()
        result_loss += result
    print(result_loss)
```

16.现有网络模型的修改及使用（本次使用到VGG模型，用于CIFAR10分类）

torchvision.models
- Classification
  - Alexnet
  - VGG
  - ResNet
  - SqueezeNet
  - DenseNet
  - Inception v3
  - GoogLeNet
  - ShuffleNet v2
  - MobileNet v2
  - MobileNet v3
  - ResNext
  - Wide ResNet
  - MNASNet
  - Quantized Models
- Semantic Segmentation
  - Fully Convolutional Networks
  - DeepLabV3
  - LR-ASPP

`torchvision.models.vgg16(pretrained: bool = False, progress: bool = True, **kwargs) → torchvision.models.vgg.VGG` [SOURCE]

VGG 16-layer model (configuration "D") *"Very Deep Convolutional Networks For Large-Scale Image Recognition" <https://arxiv.org/pdf/1409.1556.pdf>_*

Parameters:
- **pretrained** (*bool*) – If True, returns a model pre-trained on ImageNet
- **progress** (*bool*) – If True, displays a progress bar of the download to stderr

```python
import torchvision

# train_data = torchvision.datasets.ImageNet("../data_image_net", split='train', download=True,
#                                            transform=torchvision.transforms.ToTensor())
from torch import nn

vgg16_false = torchvision.models.vgg16(pretrained=False)
vgg16_true = torchvision.models.vgg16(pretrained=True)

print(vgg16_true)

train_data = torchvision.datasets.CIFAR10('../data', train=True, transform=torchvision.transforms.ToTensor(),
```

- Semantic Segmentation
  - Fully Convolutional Networks
  - DeepLabV3
  - LR-ASPP
- Object Detection, Instance Segmentation Keypoint Detection
  - Runtime characteristics
  - Faster R-CNN
  - RetinaNet
  - Mask R-CNN
  - Keypoint R-CNN
+ Video classification

```python
print(vgg16_true)

train_data = torchvision.datasets.CIFAR10('../data', train=True, transform=torchvision.transforms.ToTensor(),
                                          download=True)

vgg16_true.classifier.add_module('add_linear', nn.Linear(1000, 10))
print(vgg16_true)

print(vgg16_false)
vgg16_false.classifier[6] = nn.Linear(4096, 10)
print(vgg16_false)
```

17.网络模型的保存与读取

模型的保存
模型的加载

完整的模型训练套路-GPU训练
完整的模型验证套路

再来看一下GitHub

```python
import torchvision
import torch

vgg16 = torchvision.models.vgg16(pretrained=False)

# 保存方式一
torch.save(vgg16,"vgg16_method1.pth")

# 保存方式二（官方推荐）
torch.save(vgg16.state_dict(),"vgg16_method2.pth")

# 陷阱(不能使用第一种方式的save函数保存自己的模型)
class Sguan(nn.Module):
    def __init__(self):
        super().__init__()
        self.module1 = Sequential(
            Conv2d(in_channels=3,out_channels=32,kernel_size=5,padding=2),
            MaxPool2d(kernel_size=2),
            Conv2d(in_channels=32,out_channels=32,kernel_size=5,padding=2),
            MaxPool2d(kernel_size=2),
            Conv2d(in_channels=32,out_channels=64,kernel_size=5,padding=2),
            MaxPool2d(kernel_size=2),
            Flatten(),
            Linear(in_features=1024,out_features=64),
            Linear(in_features=64,out_features=10)
        )
    def forward(self,input_py):
        output = self.module1(input_py)
        return output
sguan_run = Sguan()
torch.save(sguan_run,"sguan_method1.pth")
```

```python
import torchvision
import torch

# 加载方式一
model1 = torch.load("vgg16_method1.pth",weights_only=False)
print(model1)

# 加载方式二（官方推荐）
model2 = torch.load("vgg16_method2.pth",weights_only=False)
vgg16 = torchvision.models.vgg16(pretrained=False)
vgg16.load_state_dict(model2)
print(vgg16)
```

18.完整的模型训练流程

2 x input

Model(2分类)

Outputs =
[0.1, 0.2]
[0.3, 0.4]

0.   1

```python
import torch

outputs = torch.tensor([[0.1, 0.2],
                        [0.05, 0.4]])

print(outputs.argmax(0))
```

```
tensor([0, 1])
```

```python
import torch

outputs = torch.tensor([[0.1, 0.2],
                        [0.3, 0.4]])

print(outputs.argmax(1))
preds = outputs.argmax(1)
targets = torch.tensor([0, 1])
```

0. 1

tensor([0, 1])

```
print(outputs.argmax(1))
preds = outputs.argmax(1)
targets = torch.tensor([0, 1])
print((preds == targets).sum())
```

Argmax

Preds = [1]
[1]

Inputs target = [0][1]

Preds == inputs target

[false, true].sum() = 1

```
import torch
from torch import nn
import torchvision
from torch.nn import Conv2d,MaxPool2d,Flatten,Linear,Sequential
from torch.utils.tensorboard import SummaryWriter
from torch.utils.data import Dataset
from torch.utils.data import DataLoader

# 1.准备的一个测试集
train_data = torchvision.datasets.CIFAR10(root=".\\dataset",train=True,transform=torchvision.transforms.ToTensor())
test_data = torchvision.datasets.CIFAR10(root=".\\dataset",train=False,transform=torchvision.transforms.ToTensor())

# 2.length长度
train_data_size = len(train_data)
test_data_size = len(test_data)

# 3.利用DataLoader加载数据集
train_loader = DataLoader(dataset=train_data,batch_size=4,shuffle=False,num_workers=0,drop_last=True)
test_loader = DataLoader(dataset=test_data,batch_size=4,shuffle=False,num_workers=0,drop_last=True)

# 4.搭建神经网络
class Sguan(nn.Module):
    def __init__(self):
        super().__init__()
        self.module1 = Sequential(
            Conv2d(in_channels=3,out_channels=32,kernel_size=5,padding=2),
            MaxPool2d(kernel_size=2),
            Conv2d(in_channels=32,out_channels=32,kernel_size=5,padding=2),
            MaxPool2d(kernel_size=2),
            Conv2d(in_channels=32,out_channels=64,kernel_size=5,padding=2),
            MaxPool2d(kernel_size=2),
            Flatten(),
            Linear(in_features=1024,out_features=64),
            Linear(in_features=64,out_features=10)
        )
    def forward(self,input_py):
        output = self.module1(input_py)
        return output

# 5.创建网络模型
sguan_run = Sguan()

# 6.损失函数
loss_fn = nn.CrossEntropyLoss()

# 7.优化器
learning_rate = 0.01
optimizer = torch.optim.SGD(sguan_run.parameters(),lr=learning_rate)

# 8.设置训练网络的参数
total_train_step = 0    # 记录训练的次数
total_test_step = 0     # 记录测试的次数
epoch = 2               # 神经网络训练的轮数

# 9.训练Loss损失函数可视化
writer = SummaryWriter("logs")

# 10.神经网络模型的训练and测试
for i in range(epoch):
    print("-------第 {} 轮训练开始runing...-------".format(i+1))
    for data in train_loader:
        imgs,targets = data
        outputs = sguan_run(imgs)
        # 1运行损失函数
        loss = loss_fn(outputs,targets)
        # 2清零函数梯度
        optimizer.zero_grad()
        # 3反向传播
        loss.backward()
        # 4调用优化器（参数优化）
        optimizer.step()
        total_train_step += 1
        if total_train_step % 1000 == 0:
```

```python
        # 4调用优化器（参数优化）
        optimizer.step()
        total_train_step += 1
        if total_train_step % 1000 == 0:
            print("训练次数：{}次,Loss：{}".format(total_train_step,loss.item()))
            # 5可视化训练参数Loss
            writer.add_scalar("sguan_myTrainLog",loss.item(),total_train_step)

    # 6测试步骤开始
    total_test_loss = 0      # 整体训练的Loss损失函数
    total_accuracy = 0       # 整体训练的准确个数（基于测试集test_loader）
    with torch.no_grad():
        for test_data in test_loader:
            imgs,targets = test_data
            outputs = sguan_run(imgs)
            loss = loss_fn(outputs,targets)
            total_test_loss += loss.item()
            # 7计算训练网络的一个准确率
            accuracy = (outputs.argmax(1) == targets).sum
            total_accuracy += accuracy
    print("整体测试集上的loss：{}".format(total_test_loss))
    print("整体测试集的准确率：{}".format(total_accuracy/test_data_size))
    # 8可视化测试参数Loss
    writer.add_scalar("sguan_myTestLog",loss.item(),total_train_step)
    writer.add_scalar("sguan_myTestMain",total_accuracy/test_data_size,total_train_step)

    # 9整体测试次数记录
    total_train_step += 1
```

19.利用GPU训练网络模型

网络模型

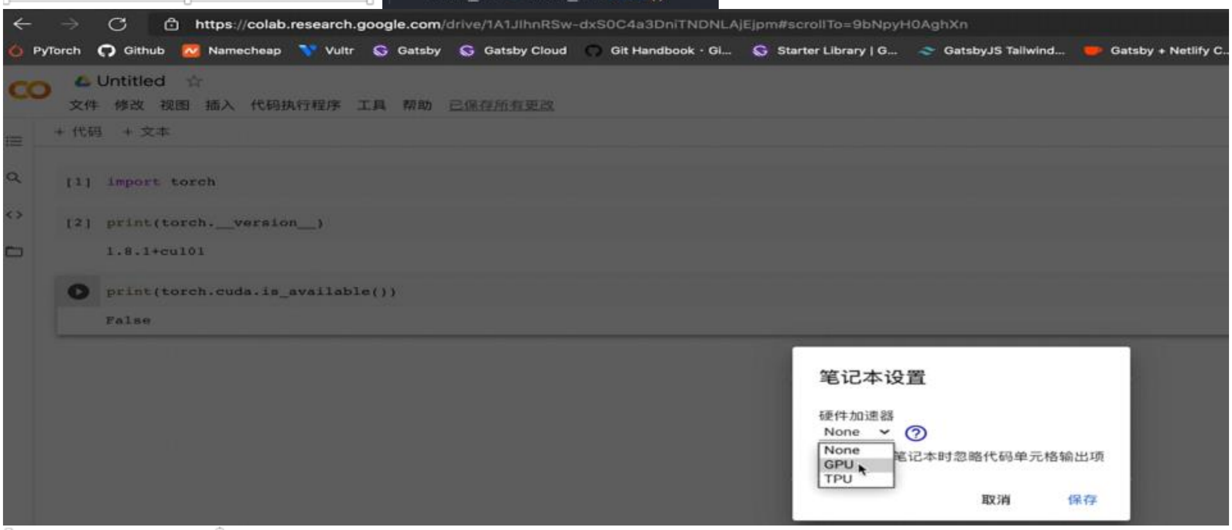
数据（输入，标注）

损失函数

.cuda()

```python
# 5.创建网络模型
sguan_run = Sguan()
if torch.cuda.is_available():
    sguan_run = sguan_run.cuda()

# 6.损失函数
loss_fn = nn.CrossEntropyLoss()
if torch.cuda.is_available():
    loss_fn = loss_fn.cuda()
```

```python
for data in train_loader:
    imgs,targets = data
    if torch.cuda.is_available():
        imgs = imgs.cuda()
        targets = targets.cuda()
```



.to(device)

Device = torch.device(“cpu”)

Torch.device(“cuda”)

Torch.device(“cuda:0”)

Torch.device(“cuda:1”)

```python
# 定义训练的设备
device = torch.device("cpu")
tudui = Tudui()
tudui = tudui.to(device)

# 损失函数
loss_fn = nn.CrossEntropyLoss()
loss_fn = loss_fn.to(device)

for data in test_dataloader:
    imgs, targets = data
    imgs = imgs.to(device)
    targets = targets.to(device)
```

```python
# 定义训练的设备
device = torch.device("cuda:0")
```

```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

20.完整模型验证套路（测试自己图片使用）


```python
import torchvision
from PIL import Image

image_path = "../imgs/dog.png"
image = Image.open(image_path)
print(image)

transform = torchvision.transforms.Compose([torchvision.transforms.Resize((32, 32)),
                                            torchvision.transforms.ToTensor()])

image = transform(image)
print(image.shape)

model = torch.load("tudui_0.pth")
print(model)
image = torch.reshape(image, (1, 3, 32, 32))
model.eval()
with torch.no_grad():
    output = model(image)
print(output)
```

```
class_to_idx = {dict: 10} {'airplane': 0, 'automobile': 1, 'bird': 2, 'cat': 3, 'deer': 4, 'dog': 5, 'frog': 6, 'horse': 7,
01 'airplane' = {int} 0
01 'automobile' = {int} 1
01 'bird' = {int} 2
01 'cat' = {int} 3
01 'deer' = {int} 4
01 'dog' = {int} 5
01 'frog' = {int} 6
01 'horse' = {int} 7
01 'ship' = {int} 8
01 'truck' = {int} 9
01 __len__ = {int} 10
```

```python
model = torch.load("tudui_29_gpu.pth", map_location=torch.device('cpu'))
print(model)
image = torch.reshape(image, (1, 3, 32, 32))
model.eval()
with torch.no_grad():
    output = model(image)
print(output)
```