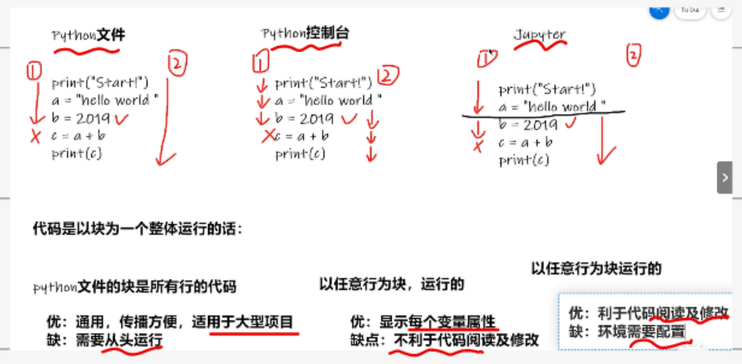
help() 查看方法具体使用

dir() 查看含有什么属性及方法

3种内部化软件对比：



dataset 与 dataloader：

数据集处理：

```python
from torch.utils.data import Dataset
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)
```

```python
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
```

Tensorboard：用于记录训练数据

add_scalar

① 用来记录训练数据

```
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 = x
for i in range(100):
    writer.add_scalar("y=x", i, i)

writer.close()
```

调用：

```
(pytorch) C:\Users\Zhiyao\Desktop\learn_torch>tensorboard --logdir=logs
TensorFlow installation not found - running with reduced feature set.
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass --bind_
TensorBoard 2.1.0 at http://localhost:6006/ (Press CTRL+C to quit)

(pytorch) C:\Users\Zhiyao\Desktop\learn_torch>tensorboard --logdir=logs --port=6007
TensorFlow installation not found - running with reduced feature set.
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass --bind_
TensorBoard 2.1.0 at http://localhost:6007/ (Press CTRL+C to quit)
```

add_image()

② 用来观察训练结果

```
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
```

对于numpy格式，可以用opencv读取，也可以用numpy转换格式

```python
import numpy as np
from PIL import Image

writer = SummaryWriter("logs")
image_path = "data/train/ants_image/0013035.jpg"
img_PIL = Image.open(image_path)
img_array = np.array(img_PIL)
print(type(img_array))
print(img_array.shape)
```

```
writer.add_image("test", img_array, 1, dataformats='HWC')
# y = 2x
for i in range(100):
    writer.add_scalar("y=2x", 3*i, i)

writer.close()
```

HWC先后顺序

指明通道

类似的有 ToPILImage( ), 使用方法一样

```
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 i
              - 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
                ``short``).

    .. _PIL.Image mode: https://pillow.readthedocs.io/en/latest/handbook/concepts.html#co
    """
    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.
```

Transforms : ToTensor() :

```
from PIL import Image
from torchvision import transforms

# python的用法 -》 tensor数据类型
# 通过 transforms.ToTensor去看两个问题

# 2、为什么我们需要Tensor数据类型

# 绝对路径 C:\Users\Zhiyao\Desktop\learn_torch\dataset\train\ants_image\0013035.jpg
# 相对路径 dataset/train/ants_image/0013035.jpg
img_path = "dataset/train/ants_image/0013035.jpg"
img = Image.open(img_path)

# 1、 transforms该如何使用 (python)
tensor_trans = transforms.ToTensor()
tensor_img = tensor_trans(img)

print(tensor_img)
```

返回是PIL格式

```
class ToTensor:
    """Convert a PIL Image or ndarray to tensor and scale the values accordingly.

    This transform does not support torchscript.

    Converts a PIL Image or numpy.ndarray (H x W x C) in the range
    [0, 255] to a torch.FloatTensor of shape (C x H x W) in the range [0.0, 1.0]
    if the PIL Image belongs to one of the modes (L, LA, P, I, F, RGB, YCbCr, RGBA, CMYK, 1)
    or if the numpy.ndarray has dtype = np.uint8

    In the other cases, tensors are returned without scaling.

    .. note::
        Because the input image is scaled to [0.0, 1.0], this transformation should not be used when
        transforming target image masks. See the `references`_ for implementing the transforms for image masks.

    .. _references: https://github.com/pytorch/vision/tree/main/references/segmentation
    """

    def __init__(self) -> None:
        _log_api_usage_once(self)

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

        Returns:
            Tensor: Converted image.
        """
        return F.to_tensor(pic)
```

常见的 Transforms

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

PIL. Image

(Transforms . ToTensor)

或 numpy . array( )

归一化：

Normalize(xxx)

```
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.

    Args:
        mean (sequence): Sequence of means for each channel.
        std (sequence): Sequence of standard deviations for each channel.
        inplace(bool,optional): Bool to make this operation in-place.

    """

    def __init__(self, mean, std, inplace=False):
        self.mean = mean
        self.std = std
        self.inplace = inplace

    def __call__(self, tensor):
        """
```

Tensor 格式才计算

通道

变化计算公式

使用

```
input[channel] = (input[channel] - mean[channel]) / std[channel]
```

(input-0.5)/0.5=2*input - 1

input [0, 1]

result[-1, 1]

变化范围

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

writer = SummaryWriter("logs")
img = Image.open("images/pytorch.png")
print(img)

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

# 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])
writer.add_image("Normalize", img_norm)


writer.close()
```

输入的通道参数

```
C:\Users\Zhiyao\Anaconda3\envs\pytorch\python.exe C:/Users/Zhiyao/De
<PIL.PngImagePlugin.PngImageFile image mode=RGB size=3200x1800 at 0x
tensor(0.8275)
tensor(0.6549)

Process finished with exit code 0
```

效果

Resize :

缩放大小：

```
class Resize(object):
    """Resize the input PIL Image to the given size.

    Args:
        size (sequence or int): 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 (int, optional): Desired interpolation. Default is
            ``PIL.Image.BILINEAR``
    """

    def __init__(self, size, interpolation=Image.BILINEAR):
        assert isinstance(size, int) or (isinstance(size, Iterable) and len(size) == 2)
        self.size = size
        self.interpolation = interpolation

    def __call__(self, img):
        """
```

Image.open.

缩放列表对应宽高，给int，以最小边为基准缩放

```
# Resize
print(img.size)
trans_resize = transforms.Resize((512, 512))
# img PIL -> resize -> img_resize PIL
img_resize = trans_resize(img) PIL
# img_resize PIL -> totensor -> img_resize tensor
img_resize = trans_totensor(img_resize)
writer.add_image("Resize", img_resize, 0)
print(img_resize)

# Compose - resize - 2
trans_resize_2 = transforms.Resize(512)
# PIL -> PIL -> tensor
trans_compose = transforms.Compose([trans_resize_2, trans_totensor])
img_resize_2 = trans_compose(img)
writer.add_image("Resize", img_resize_2, 1)


writer.close()
```

使用 compose 组合，前一个输出格式要符合后一个输入格式

调用的是后一个对象的 __call__ 方法

Compose()用法

Compose()中的参数需要是一个列表

Python中，列表的表示形式为[数据1，数据2，...]

在Compose中，数据 需要是 transforms类型

所以得到，Compose([transforms参数1， transforms参数2,...])

使用 Compose 的情况

1. 不用每一步都生成对象

使用 Compose 时，可以将多个变换操作组合成一个对象，直接对图像应用这个组合变换对象，无需为每一步单独创建变量。示例如下：

2. 组合多个

```
from PIL import Image

# 定义组合变换
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# 打开图像
image = Image.open('example.jpg')

# 直接应用组合变换
transformed_image = transform(image)
```

随机裁剪

RandomCrop :

设置宽高：要小于原图片

```
# RandomCrop
trans_random = transforms.RandomCrop((500, 1000))
trans_compose_2 = transforms.Compose([trans_random, trans_totensor])
for i in range(10):
    img_crop = trans_compose_2(img)
    writer.add_image("RandomCropHW", img_crop, i)
```

Dataset 与 Transform 联合使用

```python
dataset_transform = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor()
])

train_set = torchvision.datasets.CIFAR10(root="./dataset", train=True, transform=dataset_transform, download=True)
test_set = torchvision.datasets.CIFAR10(root="./dataset", train=False, transform=dataset_transform, download=True)

# print(test_set[0])
# print(test_set.classes)
#
# img, target = test_set[0]
# print(img)
# print(target)
# print(test_set.classes[target])
# img.show()
#
# print(test_set[0])

writer = SummaryWriter("p10")
for i in range(10):
    img, target = test_set[i]
    writer.add_image("test_set", img, i)

writer.close()
```

路径　是否为训练集　数据集变换　是否下载，这里下载了也可以 true，如果识别已下载就不会再次下载

不同数据集可以用 ctrl+space 查看 要传入参数

返回图片对象，和类别编号

Dataloader

- **dataset** (Dataset) – dataset from which to load the data. ← 从哪抓
- **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). ← 是否打乱
- **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) ← 进程
- **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) ← 剩下的是否
- **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
import torchvision

# 准备的测试数据集
from torch.utils.data import DataLoader

test_data = torchvision.datasets.CIFAR10("./dataset", train=False, transform=torchvision.transforms.ToTensor())

test_loader = DataLoader(dataset=test_data, batch_size=4, shuffle=True, num_workers=0, drop_last=False)

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

for data in test_loader:
    imgs, targets = data
    print(imgs.shape)
    print(targets)
```

最后不舍弃

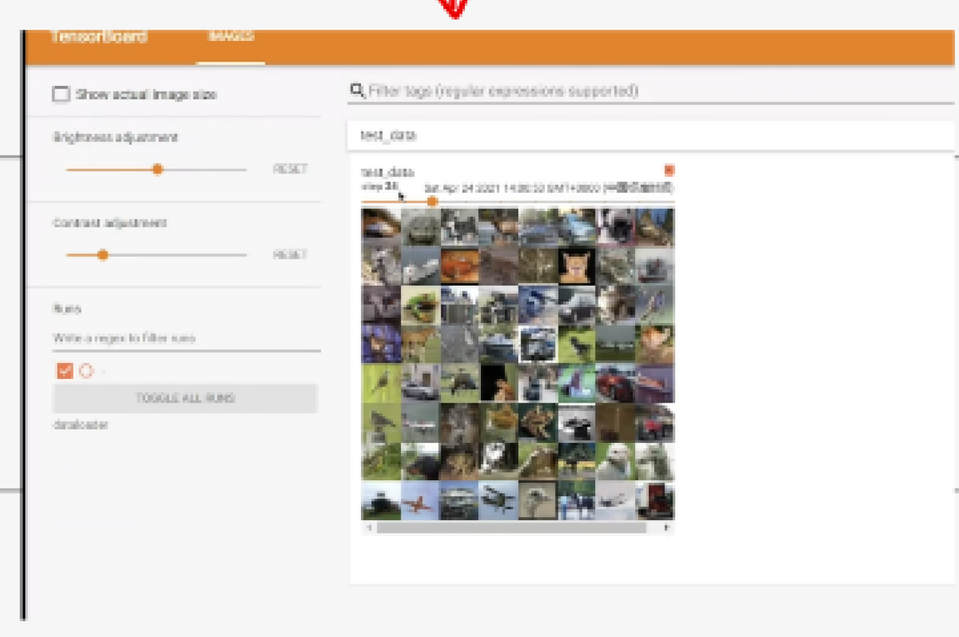
打包返回 batch_size 大小的 imgs 和类别

```
torch.Size([3, 32, 32])
3
torch.Size([4, 3, 32, 32])
tensor([2, 3, 6, 8])
torch.Size([4, 3, 32, 32])
tensor([5, 5, 8, 5])
torch.Size([4, 3, 32, 32])
tensor([6, 6, 3, 8])
torch.Size([4, 3, 32, 32])
tensor([6, 5, 5, 8])
torch.Size([4, 3, 32, 32])
```

类别

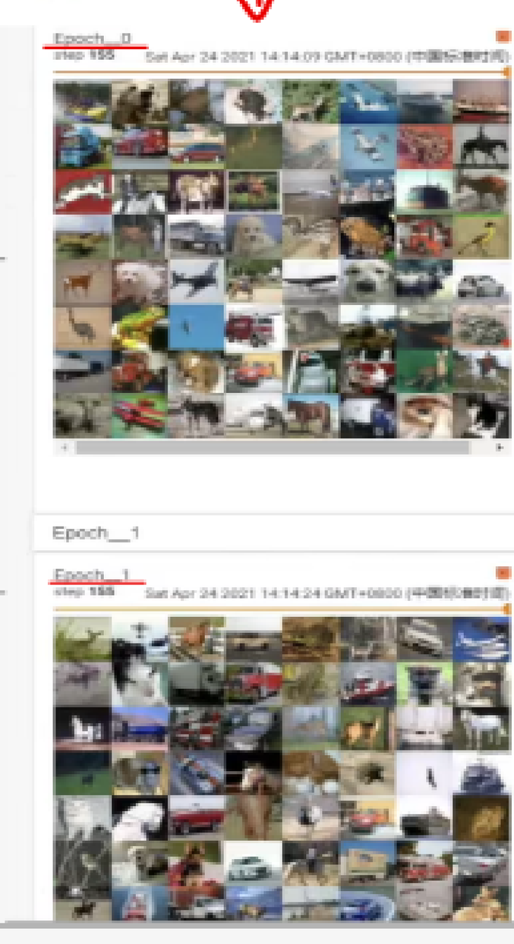
```python
writer = SummaryWriter("dataloader")
step = 0
for data in test_loader:
    imgs, targets = data
    # print(imgs.shape)
    # print(targets)
    writer.add_images("test_data", imgs, step)
    step = step + 1

writer.close()
```

batch_size = 64



两轮

```python
for epoch in range(2):
    step = 0
    for data in test_loader:
        imgs, targets = data
        # print(imgs.shape)
        # print(targets)
        writer.add_images("Epoch: {}".format(epoch), imgs, step)
        step = step + 1

writer.close()
```



神经网络的搭建

- Containers → 神经网络的骨架 →
- 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

★ Module 提供基本类	Base class for all neural network modules.
Sequential	A sequential container.
ModuleList	Holds submodules in a list.
ModuleDict	Holds submodules in a dictionary.
ParameterList	Holds parameters in a list.
ParameterDict	Holds parameters in a dictionary.

→

```python
import torch
import torch.nn as nn

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

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


tudui = Tudui()
x = torch.tensor(1.0)
output = tudui(x)
print(output)
```

这里调用 _call_，_call_ 调用 forward

call 内含完整训练链函数调用

卷积与卷积层

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 $(\text{minibatch}, \text{in\_channels}, iH, iW)$ 需要4维的
- **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

→

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

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]])

input = torch.reshape(input, (1, 1, 5, 5))
kernel = torch.reshape(kernel, (1, 1, 3, 3))

print(input.shape)
print(kernel.shape)

output = F.conv2d(input, kernel, stride=1)
print(output)

output3 = F.conv2d(input, kernel, stride=1, padding=1)
print(output3)
```

```
/Users/xiaotudui/.conda/envs/pytorch1/bin/python /Users/xiaotudui/pytorch-tutorial/src/
torch.Size([1, 1, 5, 5])
torch.Size([1, 1, 3, 3])
tensor([[[[10, 12, 12],
          [18, 16, 16],
          [13,  9,  3]]]])

Process finished with exit code 0
```

reshape 将卷积变为 conv2d 的 input 需要的形式 4维张量

Conv2d

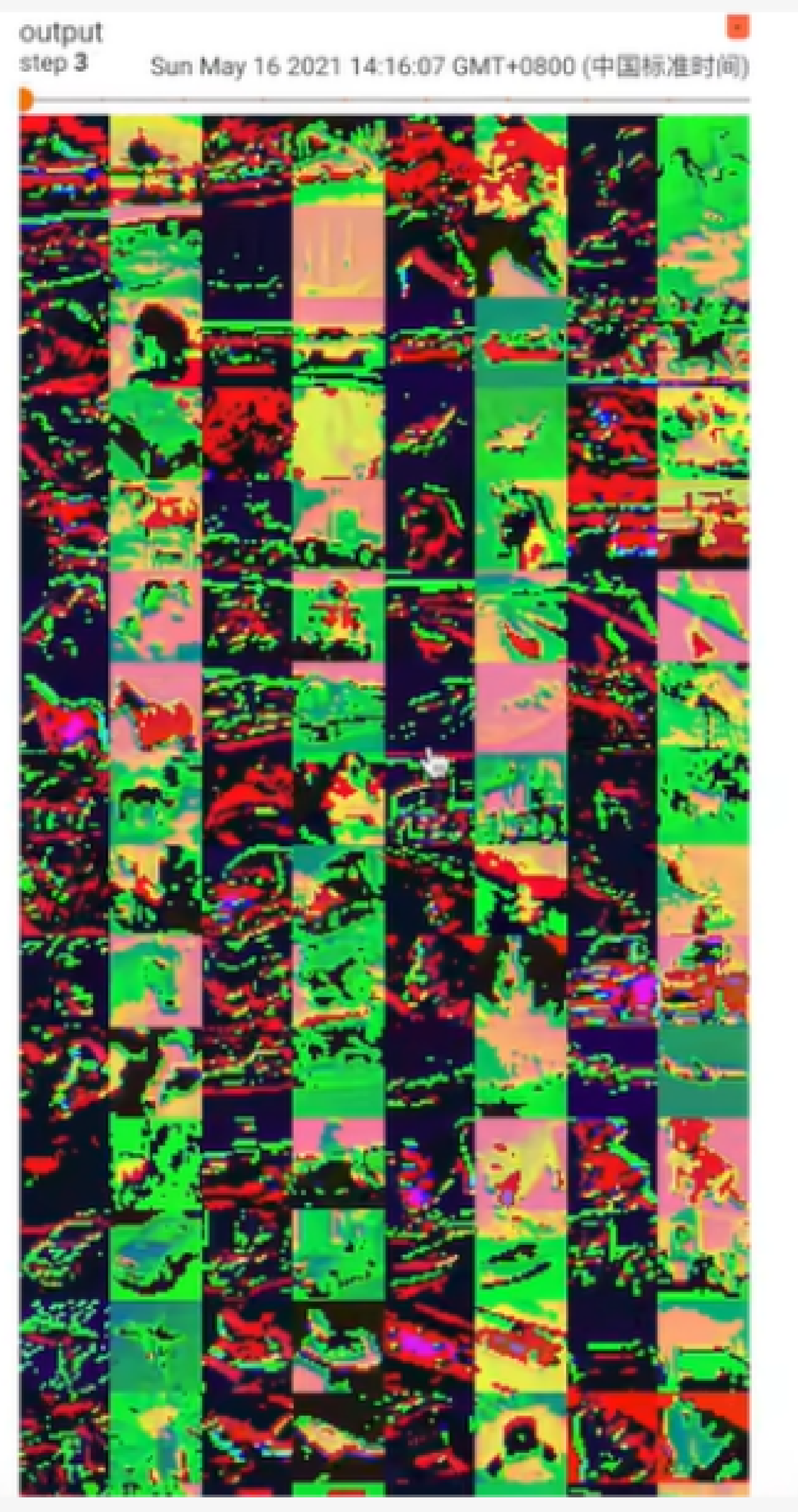
> **NOTE**
> In some circumstances when given tensors on a CUDA device and using CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting `torch.backends.cudnn.deterministic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

Parameters

- **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`

偏置

```python
class Tudui(nn.Module):
    def __init__(self):
        super(Tudui, self).__init__()
        self.conv1 = Conv2d(in_channels=3, out_channels=6, kernel_size=3, stride=1, padding=0)

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

tudui = Tudui()

writer = SummaryWriter("../logs")

step = 0
for data in dataloader:
    imgs, targets = data
    output = tudui(imgs)
    print(imgs.shape)
    print(output.shape)
    # torch.Size([64, 3, 32, 32])
    writer.add_images("input", imgs, step)
    # torch.Size([64, 6, 30, 30])  -> [xxx, 3, 30, 30]

    output = torch.reshape(output, (-1, 3, 30, 30))
    writer.add_images("output", output, step)

    step = step + 1
```



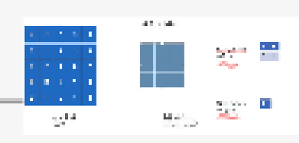
池化层

Pooling layers

nn.MaxPool1d	Applies a 1D max pooling over an input signal composed of several input planes.
nn.MaxPool2d （最大池化下采样）	Applies a 2D max pooling over an input signal composed of several input planes.
nn.MaxPool3d	Applies a 3D max pooling over an input signal composed of several input planes.
nn.MaxUnpool1d （最大池化上采样）	Computes a partial inverse of MaxPool1d.
nn.MaxUnpool2d	Computes a partial inverse of MaxPool2d.
nn.MaxUnpool3d	Computes a partial inverse of MaxPool3d.
nn.AvgPool1d （平均池化）	Applies a 1D average pooling over an input signal composed of several input planes.
nn.AvgPool2d	Applies a 2D average pooling over an input signal composed ...

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

保留/不保



```python
from torch.nn import MaxPool2d

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]], dtype=torch.float32)

input = torch.reshape(input, (-1, 1, 5, 5))
print(input.shape)

class Tudui(nn.Module):
    def __init__(self):
        super(Tudui, self).__init__()
        self.maxpool1 = MaxPool2d(kernel_size=3, ceil_mode=False)

    def forward(self, input):
        output = self.maxpool1(input)
        return output

tudui = Tudui()
output = tudui(input)
```

池化时需要的是浮点类型

```
/Users/xiaotudui/.conda/envs/pytorch/bin/python /Users/xiaotudui/pytorch-tutorial/src/nn_maxpool.py
torch.Size([1, 1, 5, 5])
tensor([[[[2., 3.],
          [5., 1.]]]])

Process finished with exit code 0
```

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

writer = torch.utils.tensorboard.SummaryWriter('logs')
test_data = torchvision.datasets.CIFAR10(root='./dataset', train=False, download=True,transform=torchvision.transforms.ToTensor())
test_data_loader = torch.utils.data.DataLoader(test_data, batch_size=64, shuffle=True, num_workers=0)

class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = Conv2d(in_channels=3, out_channels=6, kernel_size=3,stride=1, padding=0)
        self.maxpool = torch.nn.MaxPool2d(kernel_size=3)
    def forward(self, x):
        x = self.conv1(x)
        x = self.maxpool(x)
        return x

net = Net()
step = 0
for data in test_data_loader:
    imgs,targets = data
    print(imgs.shape)
    output = net(imgs)
    print(output.shape)
    output=torch.reshape(output,[-1,3,10,10])
    writer.add_images('input', imgs, step)
    writer.add_images('output', output, step)
    step += 1
```

这里(10,10)由output.shape查看
"3"通道是可以展现的通道数
"-1"是由电脑自动计算值.

非线性激活

nn.ELU	Applies the element-wise function:
nn.Hardshrink	Applies the hard shrinkage function element-wise:
nn.Hardsigmoid	Applies the element-wise function:
nn.Hardtanh	Applies the HardTanh function element-wise
nn.Hardswish	Applies the hardswish function, element-wise, as described in the paper:
nn.LeakyReLU	Applies the element-wise function:
nn.LogSigmoid	Applies the element-wise function:
nn.MultiheadAttention	Allows the model to jointly attend to information from different representation subspaces.
nn.PReLU	Applies the element-wise function:
nn.ReLU	Applies the rectified linear unit function element-wise:

CLASS torch.nn.ReLU(inplace=False)

Applies the rectified linear unit function element-wise:

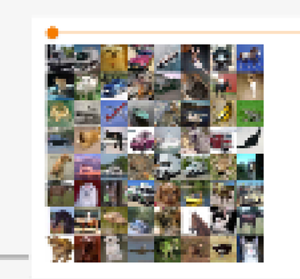
$\text{ReLU}(x) = (x)^+ = \max(0, x)$

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

```python
        self.relu = torch.nn.ReLU()
        self.sigmoid = torch.nn.Sigmoid()
    def forward(self, x):
        x = self.conv1(x)
        x = self.maxpool(x)
        x=self.relu(x)
        x = self.sigmoid(x)
```



线性层

输入与输出维度

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

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

This module supports TensorFloat32.

Parameters

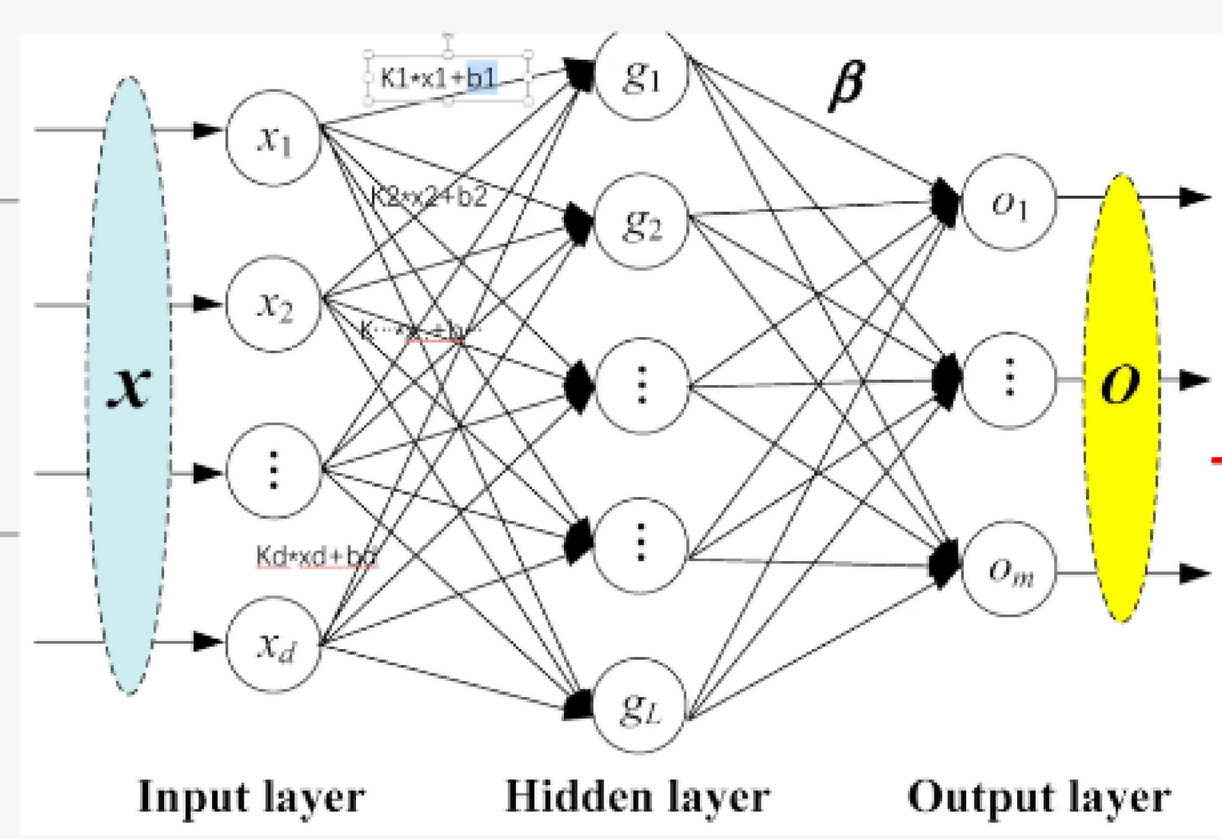
- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to `False`, the layer will not learn an additive bias. Default: `True`

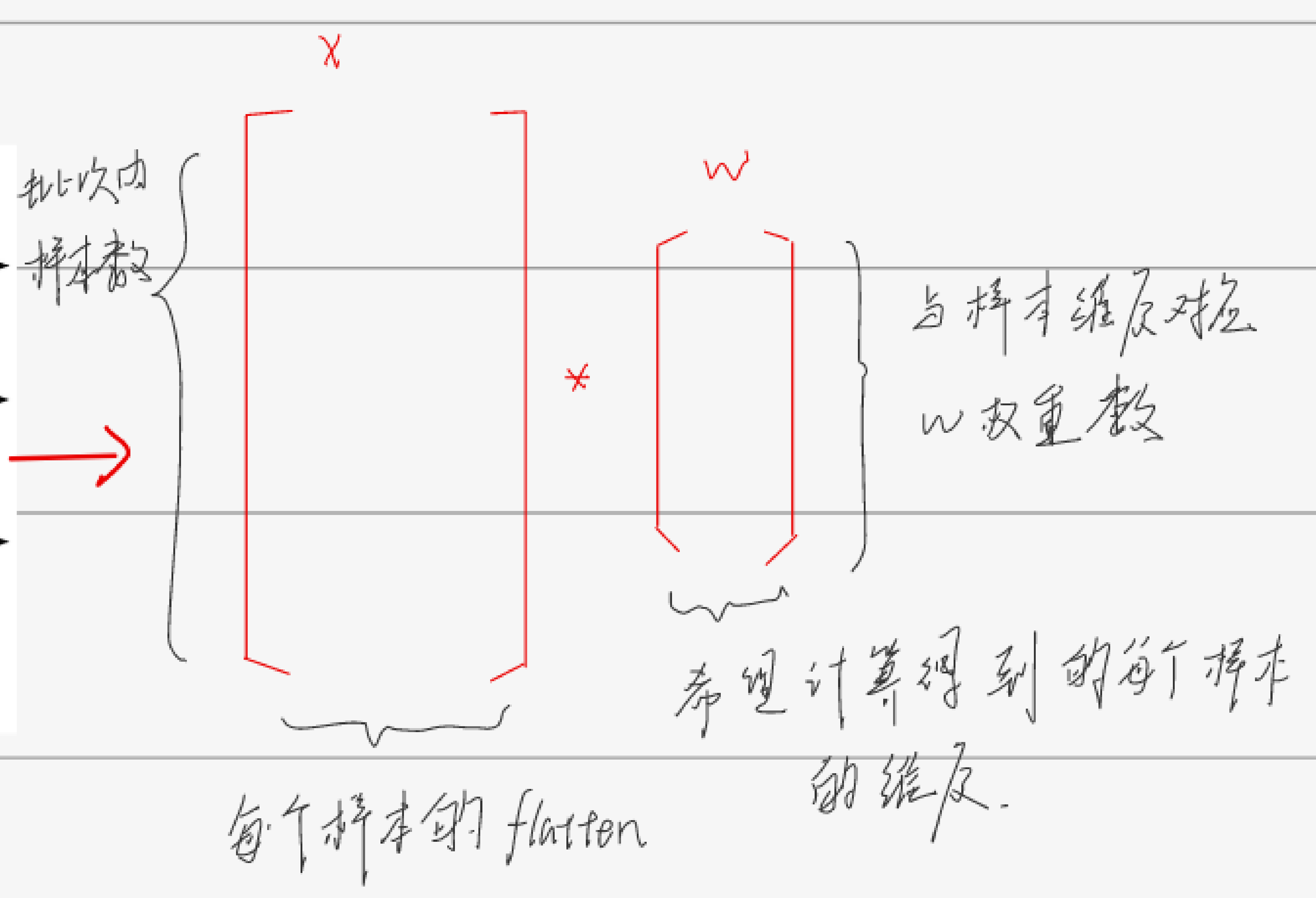
Shape:
- Input: $(N, *, H_{in})$ where $*$ means any number of additional dimensions and $H_{in} = \text{in\_features}$
- Output: $(N, *, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

Variables

- **~Linear.weight** – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$
- **~Linear.bias** – the learnable bias of the module of shape (out_features). If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in\_features}}$

通过分布初始化权重和偏置





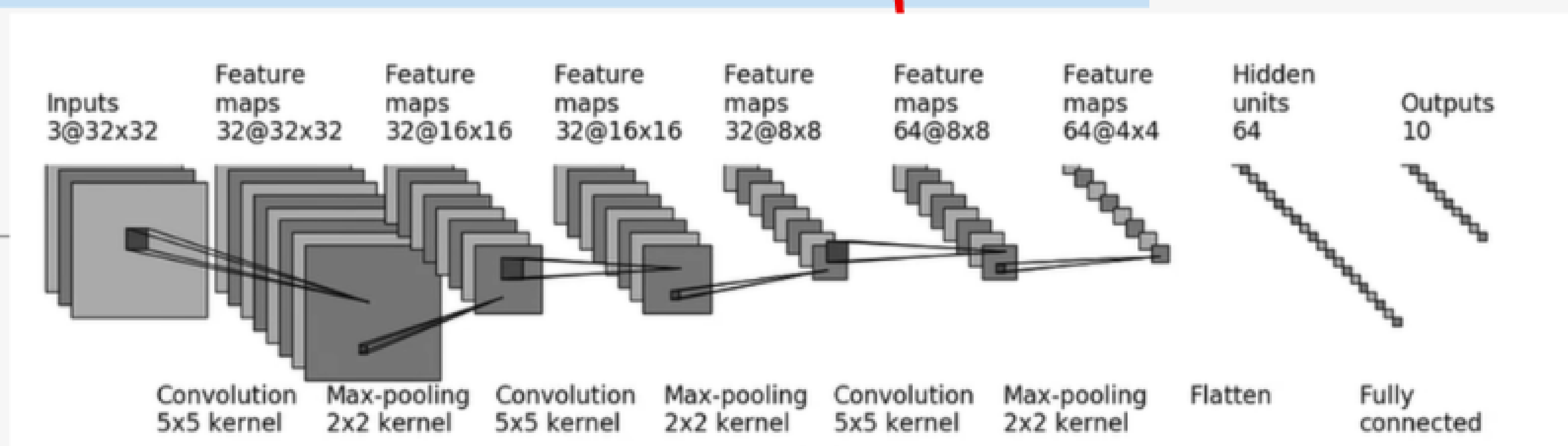

```python
dataloader = DataLoader(dataset, batch_size=64)

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

    def forward(self, input):
        output = self.linear1(input)
        return output

tudui = Tudui()

for data in dataloader:
    imgs, targets = data
    print(imgs.shape)
    output = torch.flatten(imgs)
    print(output.shape)
    output = tudui(output)
    print(output.shape)
```

→ 分别对应输入、输出维度
其中输入维度是每个样本，而不是一个批次展开.

(imgs, start_dim=1)
从哪个维度开始展开，=1时就不会展开批次

→ 这里flatten会自动将每个样本独立展开，不会混淆一个批次样本.

CIFAR -10 模型代码




```python
from torch import nn
from torch.nn import Conv2d, MaxPool2d, Flatten, Linear


class Tudui(nn.Module):
    def __init__(self):
        super(Tudui, self).__init__()
        self.conv1 = Conv2d(3, 32, 5, padding=2)
        self.maxpool1 = MaxPool2d(2)
        self.conv2 = Conv2d(32, 32, 5, padding=2)
        self.maxpool2 = MaxPool2d(2)
        self.conv3 = Conv2d(32, 64, 5, padding=2)
        self.maxpool3 = MaxPool2d(2)
        self.flatten = Flatten()
        self.linear1 = Linear(1024, 64)
        self.linear2 = Linear(64, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.maxpool1(x)
        x = self.conv2(x)
        x = self.maxpool2(x)
        x = self.conv3(x)
        x = self.maxpool3(x)
        x = self.flatten(x)
        x = self.linear1(x)
        x = self.linear2(x)
        return x

tudui = Tudui()
print(tudui)
```

通过 sequential → 简化代码

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

class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.sequential = torch.nn.Sequential(
            Conv2d(in_channels=3, out_channels=32, kernel_size=5,stride=1, padding=2),
            torch.nn.MaxPool2d(kernel_size=2),
            Conv2d(in_channels=32, out_channels=32, kernel_size=5,stride=1, padding=2),
            torch.nn.MaxPool2d(kernel_size=2),
            Conv2d(in_channels=32, out_channels=64, kernel_size=5,stride=1, padding=2),
            torch.nn.MaxPool2d(kernel_size=2),
            torch.nn.Flatten(start_dim=1),
            torch.nn.Linear(64*4*4, 64),
            torch.nn.Linear(64, 10),
        )
    def forward(self, x):
        x = self.sequential(x)
        return x

net=Net()

input=torch.ones((64,3, 32, 32))
output=net(input)
cifar_logs = SummaryWriter('logs_cifar10')
cifar_logs.add_graph(net, input)
cifar_logs.close()
print(output.shape)
```

设张量全为1

绘制模型流程

```python
accuracy += (output.argmax(1) == targets).sum().item()
```

判断类别 argmax(1) 为横向 max
(0) 为纵向 max

损失函数与反向传播

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

- 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

是否对损失值加权,求和等操作

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:

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

求均 / 求和

```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(reduction='sum')
result = loss(inputs, targets)

print(result)
```

Shape:

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

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:

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

```python
loss_mse = nn.MSELoss()
result_mse = loss_mse(inputs, targets)
```

交叉熵 ☆
用于多分类:

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:

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

or in the case of the weight argument being specified:

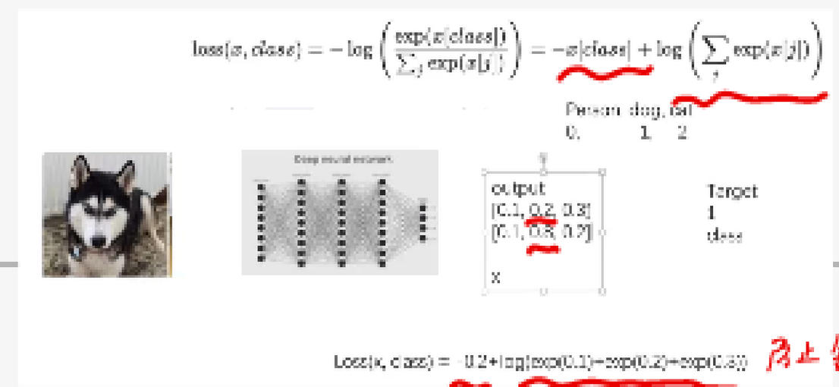
$$\operatorname{loss}(x, class) = weight[class]\left(-x[class] + \log\left(\sum_j \exp(x[j])\right)\right)$$

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

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

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.



防止每个预测都高,难以判断.

```python
x = torch.tensor([0.1, 0.2, 0.3])
y = torch.tensor([1])
x = torch.reshape(x, (1, 3))
loss_cross = nn.CrossEntropyLoss()
result_cross = loss_cross(x, y)
print(result_cross)
```

```python
loss = nn.CrossEntropyLoss()
tudui = Tudui()
for data in dataloader:
    imgs, targets = data
    outputs = tudui(imgs)
    result_loss = loss(outputs, targets)
    print(result_loss)
```

```
/Users/xiaotudui/.conda/envs/pytorch/bin/python /Users/xiaotudui/pytorch-tutorial/
Files already downloaded and verified
tensor(2.3247, grad_fn=<NllLossBackward>)
tensor(2.3107, grad_fn=<NllLossBackward>)
tensor(2.3220, grad_fn=<NllLossBackward>)
tensor(2.3824, grad_fn=<NllLossBackward>)
tensor(2.3001, grad_fn=<NllLossBackward>)
tensor(2.2995, grad_fn=<NllLossBackward>)
tensor(2.2414, grad_fn=<NllLossBackward>)
tensor(2.2937, grad_fn=<NllLossBackward>)
```

→ 反向传播 →

```python
imgs, targets = data
outputs = tudui(imgs)
result_loss = loss(outputs, targets)
result_loss.backward()
print("ok")
```

优化器

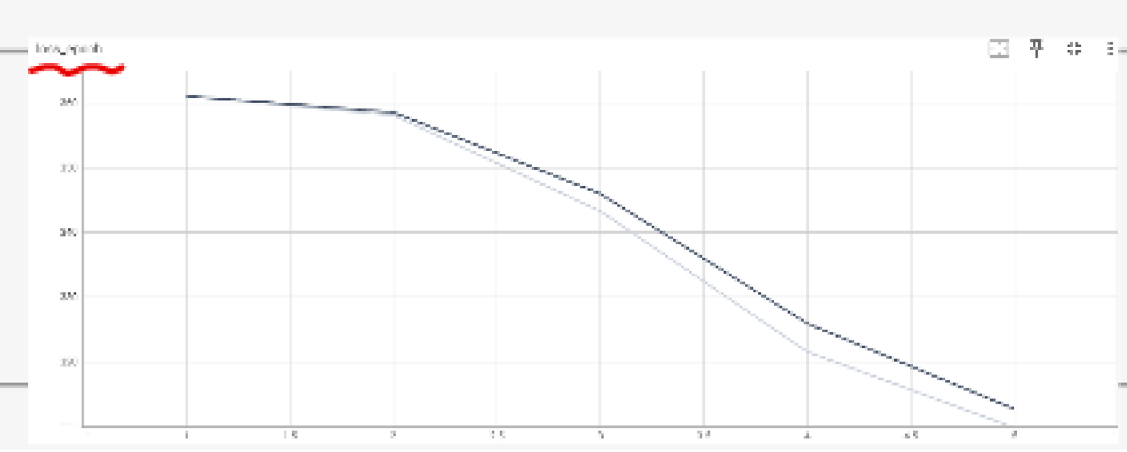
```python
loss = nn.CrossEntropyLoss()
tudui = Tudui()
optim = torch.optim.SGD(tudui.parameters(), lr=0.01)
for epoch in range(20):
    running_loss = 0.0
    for data in dataloader:
        imgs, targets = data
        outputs = tudui(imgs)
        result_loss = loss(outputs, targets)
```

学习率

```python
net=Net()
loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)
cifar_logs = SummaryWriter('logs_cifar10')
epoch = 0
for epoch in range(5):
    running_loss = 0.0
    for data in test_data_loader:
        imgs,targets = data
        output = net(imgs)
        loss_value = loss(output, targets)
        optimizer.zero_grad()
        loss_value.backward()
        optimizer.step()
        running_loss += loss_value
```

→的梯度

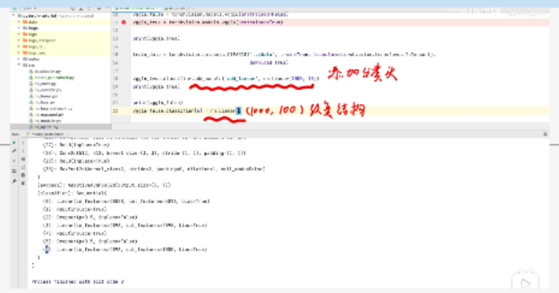
优化器每次求梯度前都要先清零.



```
        optim.zero_grad()
        result_loss.backward()
        optim.step()
        running_loss = running_loss + result_loss
    print(running_loss)
```

```
    epoch += 1
    print(running_loss)
    cifar_logs.add_scalar('loss_epoch', running_loss, epoch)

cifar_logs.close()
```

现有模型的修改使用



```
(28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(29): ReLU(inplace=True)
(30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
(classifier): Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU(inplace=True)
  (2): Dropout(p=0.5, inplace=False)
  (3): Linear(in_features=4096, out_features=4096, bias=True)
  (4): ReLU(inplace=True)
  (5): Dropout(p=0.5, inplace=False)
  (6): Linear(in_features=4096, out_features=1000, bias=True)
```

```
  (5): Dropout(p=0.5, inplace=False)
  (6): Linear(in_features=4096, out_features=100, bias=True)
  (linear): Linear(in_features=100, out_features=10, bias=True)
)
```

现有模型保存读取

```
vgg16 = torchvision.models.vgg16(pretrained=False)
# 保存方式1,模型结构+模型参数
torch.save(vgg16, "vgg16_method1.pth")

# 保存方式2，模型参数（官方推荐）
torch.save(vgg16.state_dict(), "vgg16_method2.pth")
```

用的典型式

```
# 陷阱
class Tudui(nn.Module):
    def __init__(self):
        super(Tudui, self).__init__()
        self.conv1 = nn.Conv2d(3, 64)

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

tudui = Tudui()
torch.save(tudui, "tudui_method1.pth")
```

```
# 方式1-》保存方式1，加载模型
import torchvision

model = torch.load("vgg16_method1.pth")
# print(model)
```

```
# 方式2，加载模型
vgg16 = torchvision.models.vgg16(pretrained=False)
vgg16.load_state_dict(torch.load("vgg16_method2.pth"))
# model = torch.load("vgg16_method2.pth")
print(vgg16)
```

要先 import *

```
model = torch.load('tudui_method1.pth')
print(model)
```

保存自己训练后的模型

```
torch.save(tudui, "tudui_{}.pth".format())
```

模型测试

① with torch.no_grad():

验证前会先置0，∴不用梯度

② (model名).train()

(model名).eval()

利用GPU训练

调.cuda()类型

网络模型 →

```
if torch.cuda.is_available():
    tudui = tudui.cuda()
```

数据（输入，标注）→

```
if torch.cuda.is_available():
    imgs = imgs.cuda()
    targets = targets.cuda()
```

损失函数 →

```
if torch.cuda.is_available():
    loss_fn = loss_fn.cuda()
```

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

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

方法2：

```
tudui = Tudui()
tudui = tudui.to(device)

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

model.train()

model.train() 是用于将模型设定为训练模式的方法。在训练模式下，模型内某些特定层（像 Dropout、BatchNorm 这类）会依据训练的需求进行运作。

- **Dropout 层**：该层在训练期间会随机“丢弃”一部分神经元，以此避免过拟合现象。具体而言，在每次前向传播时，每个神经元都有一定概率被暂时忽略。
- **BatchNorm 层**：此层在训练时会对输入数据进行归一化处理，并且会持续更新自身的统计信息（例如均值和方差）。

model.eval()

model.eval() 用于把模型设置成评估模式。在评估模式下，模型里的 Dropout、BatchNorm 等层会以不同的方式运行。

- **Dropout 层**：在评估阶段，所有神经元都会参与到计算中，不再进行随机丢弃操作。
- **BatchNorm 层**：在评估时，会使用训练阶段所统计的均值和方差，而不会再对这些统计信息进行更新。