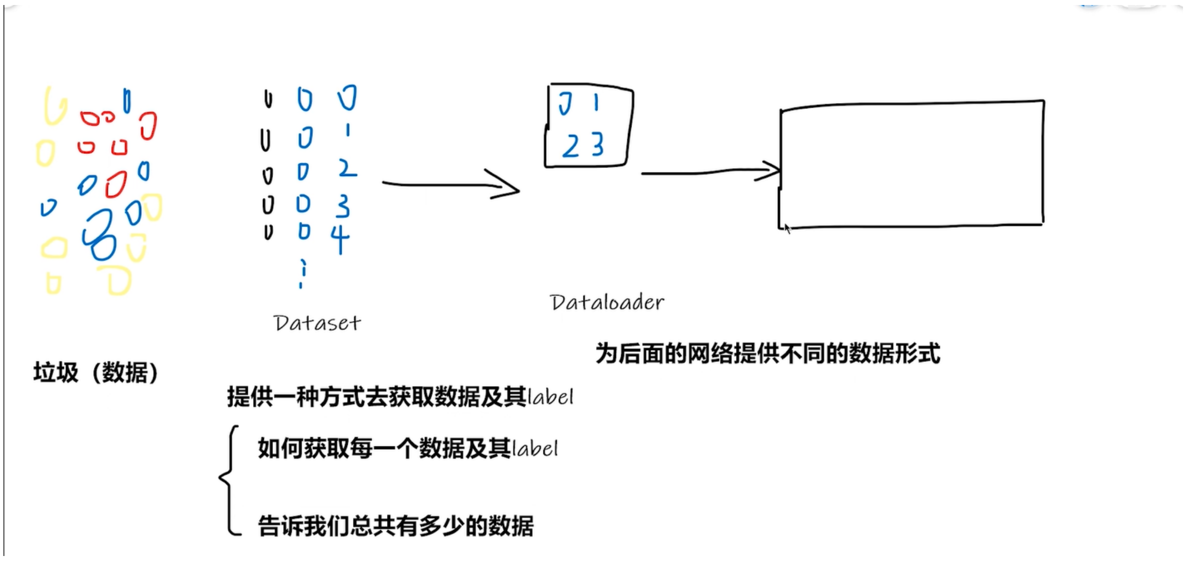
3.Python学习中的两大法宝函数

```
import torch

dir(torch)

//查看使用说明
help(torch.cuda.is_available)
```

4.pytorch加载数据初认识


Dataset
Dataloader
为后面的网络提供不同的数据形式
垃圾（数据）
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

Dataloader相当于对数据进行一个打包，让数据以batchsize的大小进入网络中。

5.Dataset类代码实战

安装opencv的命令：conda install opencv-python

pip install opencv-python

读取一个图片：

```
from PIL import Image
img_path = "图片路径，单斜杠变成双斜杠"
img = Image.open(img_path)
```

获取文件夹下所有图片集合：

```
dir_path="数据文件夹的相对路径"
import os
img_path_list = os.listdir(dir_path)
```

路径拼接：

```
import os
root_dir = ""
label_dir = ""
path = os.path.join(root_dir, label_dir)
```

read_data.py代码：

```
from torch.utils.data import Dataset
from PIL import Image

class MyData(Dataset): //自定义的类
    def __int__(self, root_dir, label_dir):
    //self相当于指定了类中的全局变量
    self.root_dir = root_dir
    self.label_dir = lable_dir
    self.path = os.path.join(self.root_dir, self.lable_dir)
    self.img_path = os.listdir(self.path)//获得文件夹中所有图片的地址，img_path是list
类型

    def __getitem__(self,idx)://idx,索引
        //图片名字
        img_name = self.img_path[idx]
        //图片相对路径
        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) //返回list列表的长度

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//对两个数据集进行拼接

//获取第一张图片
img, label = ants_dataset[0]
img.show()
```

7.TensorBoard使用

TensorBoard是一组用于可视化的工具。

安装TensorBoard：pip install tensorboard

```
from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter("logs") //实体类

#y = x的图像
for i in range（100）
    writer.add_scalar("y=x", i, i)

writer.close()
```

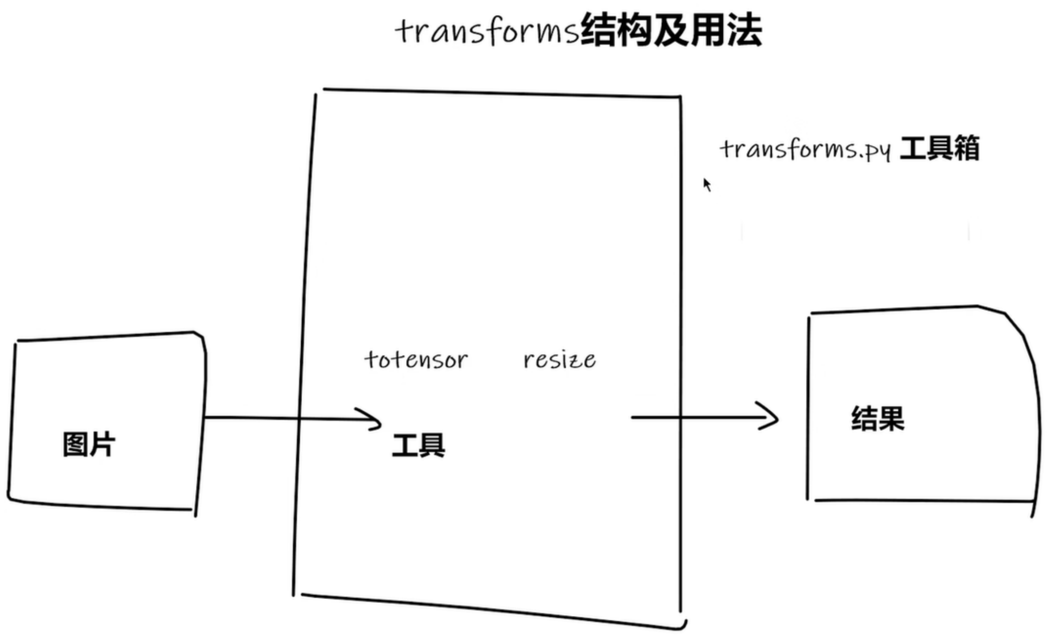
pycharm的console内： **tensorboard --logdir= 事件文件所在文件夹名（logs)**

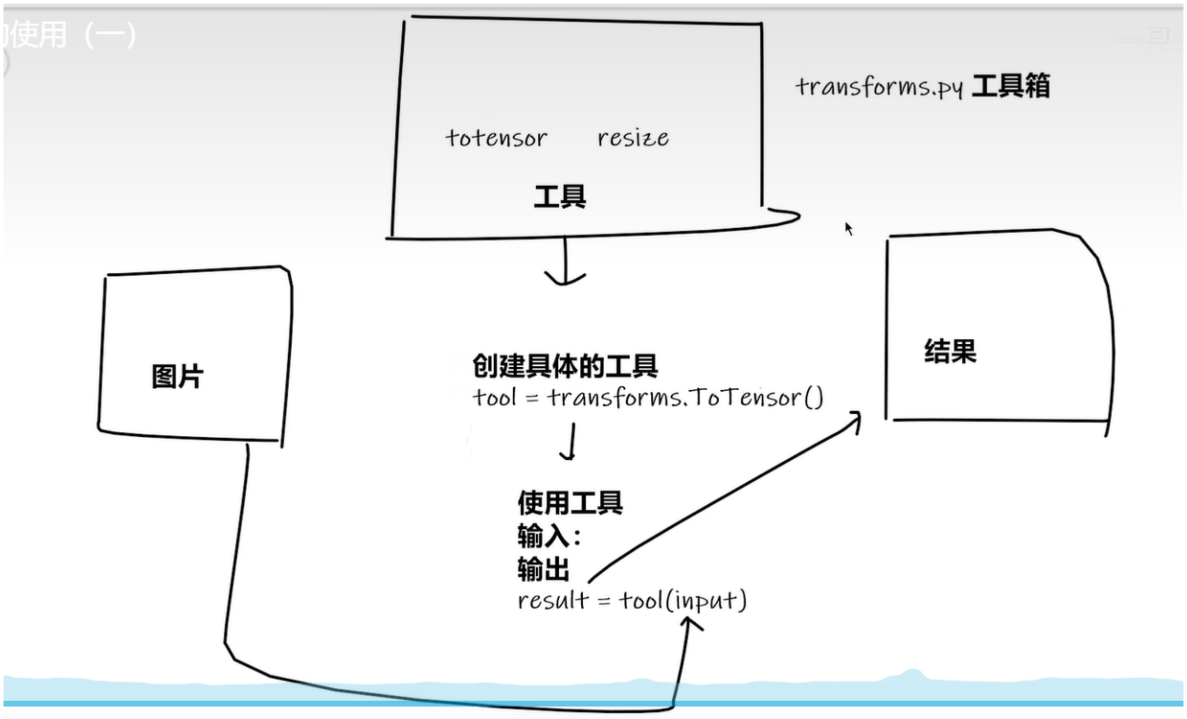
调出网页

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

writer = SummaryWriter("logs") //实体类
image_path="图片的相对路径"
img_PIL = Image.open(image_path)
img_array = np.array(img_PIL) //类型转换

writer.add_image("test", img_array, 1, dataformats='HWC')

writer.close()
```

torchvision中的transforms

transforms的结构和用法





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

# tensor数据类型
# 通过transforms.ToTensor解决两个问题
# 1.transforms该如何使用
# 2.为什么需要tensor数据类型

img_path = "dataset/train/ants_image/5650366_e22b7e1065.jpg"
img = Image.open(img_path)

writer = SummaryWriter("logs")
# 1.transforms该如何使用
tensor_trans = transforms.ToTensor()
tensor_img = tensor_trans(img)

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

```
F:\PycharmProject\learn_pytorch>tensorboard --logdir logs
```

常见的transforms

python中__ call __ 的用法

ToTensor的使用

```
from PIL import Image
from tensorboardX  import SummaryWriter
from torchvision import transforms

writer = SummaryWriter("logs")
img = Image.open("images/test.jpg")
print(img)

#ToTensor的使用
trans_totensor = transforms.ToTensor()
img_tensor = trans_totensor(img)
writer.add_image("ToTensor",img_tensor)
writer.close()
```

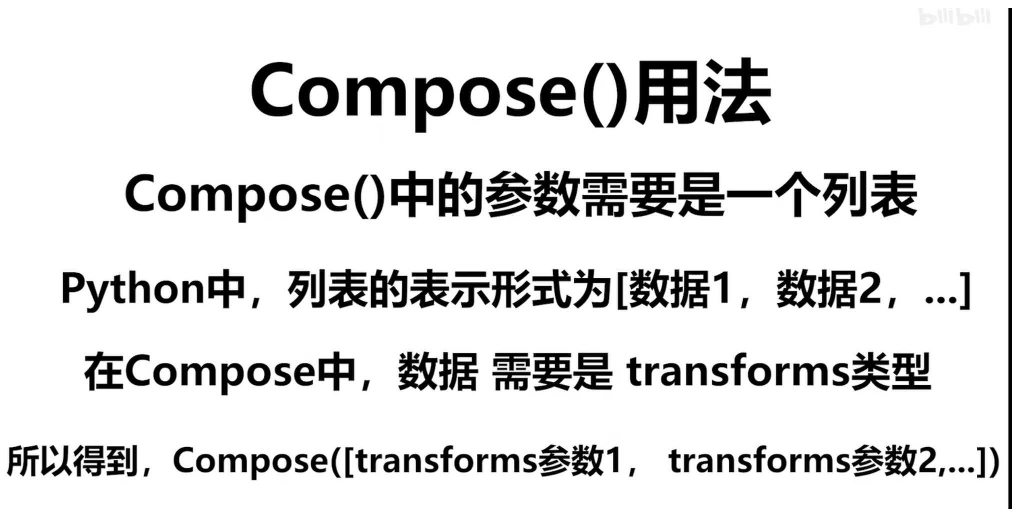
Normalize的使用

```
print(img_tensor[0][0][0]) #0层0行0列
trans_norm = transforms.Normalize([0.5,0.5,0.5] , [0.5,0.5,0.5])
img_norm =trans_norm(img_tensor) #传入tensor数据类型
print(img_norm[0][0][0]) #规一之后的
```

Resize

```

print(img.size)
trans_resize = transforms.Resize((512,512))
img_resize = trans_resize(img)
print(img_resize)
```

Compose用法



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,...])

RandomCrop用法

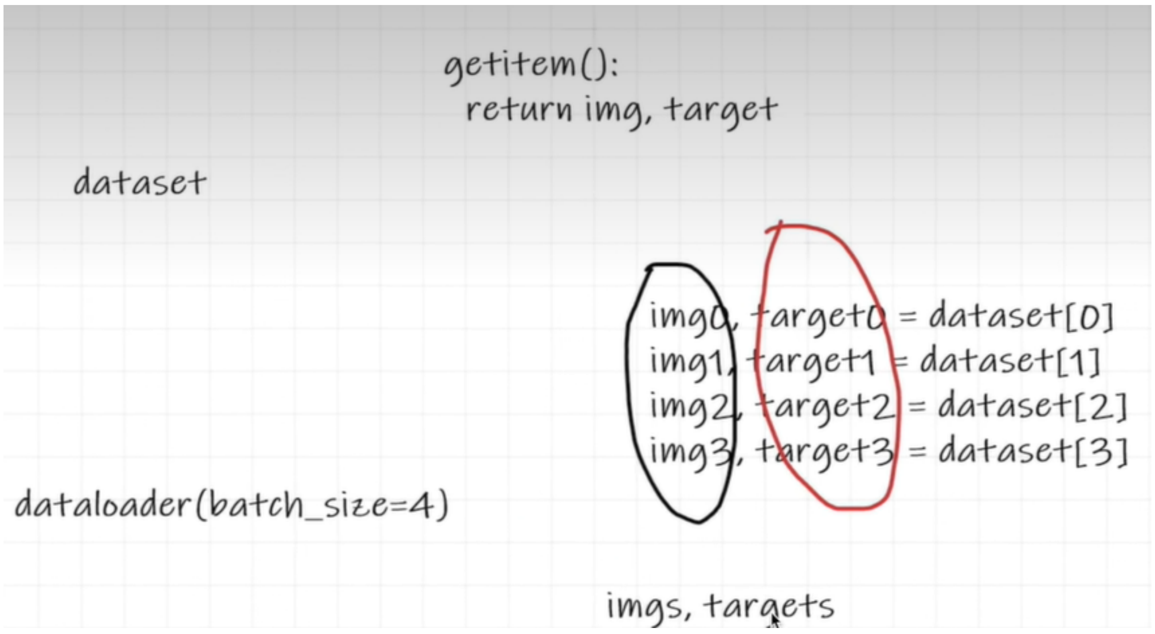
```
#RandomCrop
trans_random = transforms.RandomCrop(512)
trans_compose_2 = transforms.Compose([trans_random, trans_totensor])# 先随机裁剪
然后转换为tensor数据类型
for i in range(10):
    img_crop = trans_compose_2(img)
    writer.add_image("RandomCrop",img_crop,i)
```

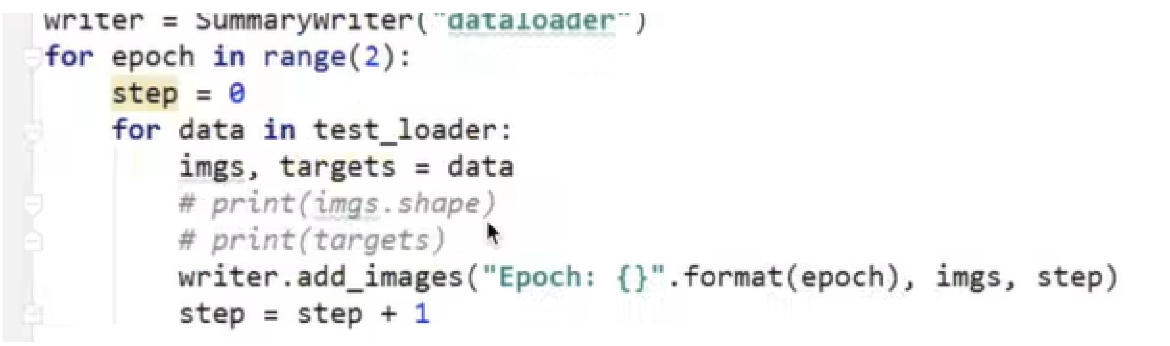
torchvision中的数据集使用

参数中的img_tensor，表示传入tensor类型的图片

P15-DalaLoader的使用

dalaLoader把数据加载到神经网络中，怎么取、取多少数据通过dataloader的参数设置，


getitem():
return img, target
dataset
img0, target0 = dataset[0]
img1, target1 = dataset[1]
img2, target2 = dataset[2]
img3, target3 = dataset[3]
dataloader(batch_size=4)
imgs, targets

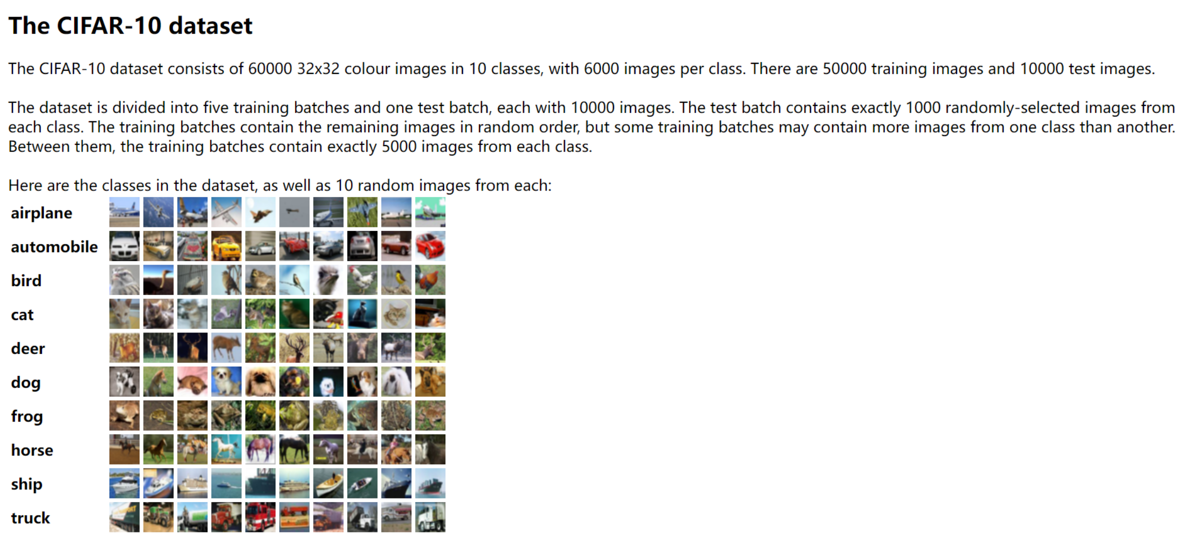


```
writer = SummaryWriter("dataloader")
for epoch in range(2):
    step = 0
    for data in test_loader:
        imgs, targets = data
        # print(imgs.shape)
        # print(targets)
        writer.add_images("Epoch: {}".format(epoch), imgs, step)
        step = step + 1
```

```
import torchvision
from torch.utils.data import DataLoader

test_data = torchvision.datasets.CIFAR10("./dataset2", 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
img, target = test_data[0]
print(img.shape)
print(target)

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

CIFAR-10



The CIFAR-10 dataset

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

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

Here are the classes in the dataset, as well as 10 random images from each:

P16-神经网络的基本骨架，nn.Module的使用

```
import torch

from torch import nn

class Test1(nn.Module):
    def __init__(self):
        super().__init__()

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

test1 = Test1() #创建的神经网络
x = torch.tensor(1.0) #x是Tensor数据类型
output = test1(x) #把x作为神经网络的输入
print(output)
```

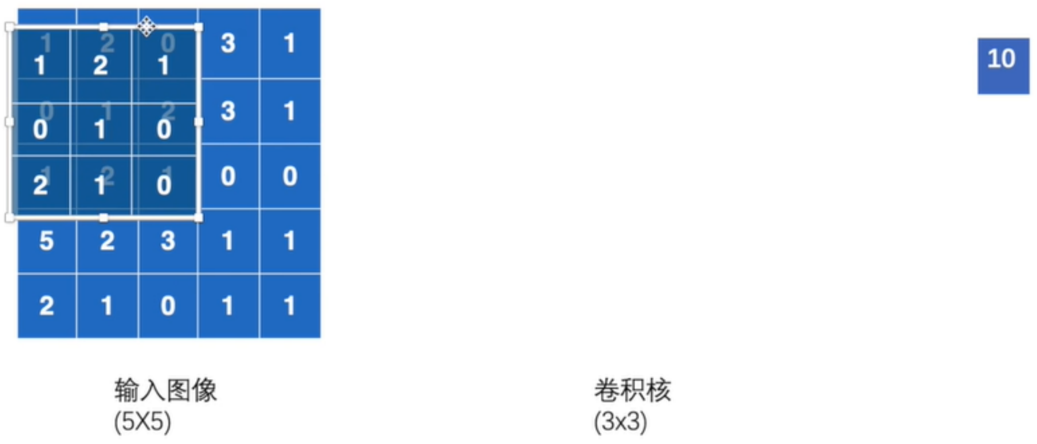
P17-卷积操作

convolution layers卷积层

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

Parameters

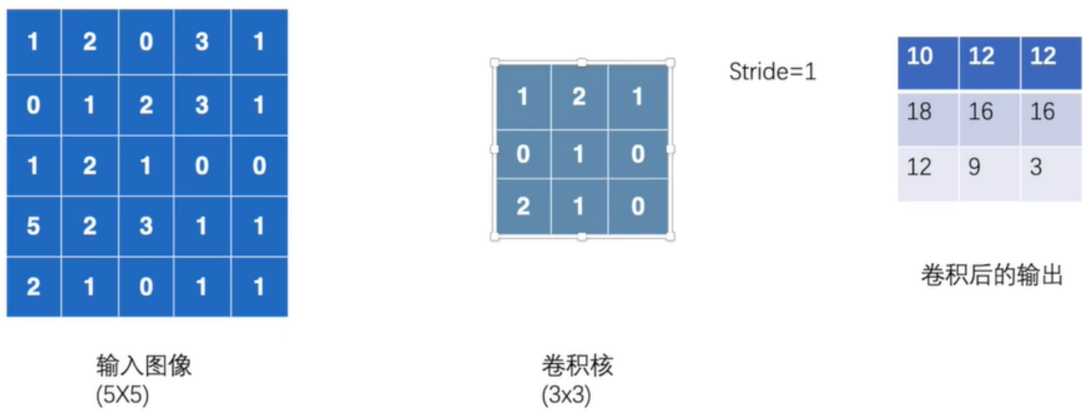
- **input** – input tensor of shape $(\text{minibatch}, \text{in\_channels}, iH, iW)$
- **weight** – filters of shape $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, kH, kW)$
- **bias** – optional bias tensor of shape (out_channels). Default: `None`
- **stride** – the stride of the 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 string {'valid', 'same'}, single number or a tuple (*padH*, *padW*). Default: 0 `padding='valid'` is the same as no padding. `padding='same'` pads the input so the output has the same shape as the input. However, this mode doesn't support any stride values other than 1.



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

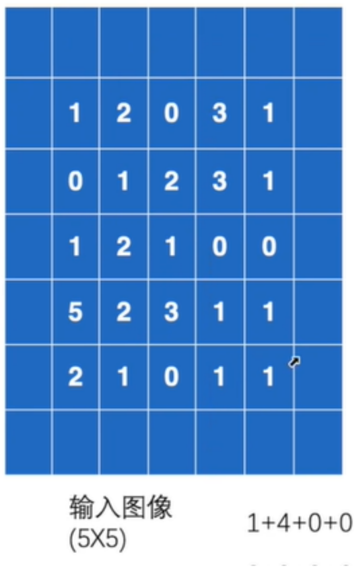
stride就是每次移动的步长

最终结果：



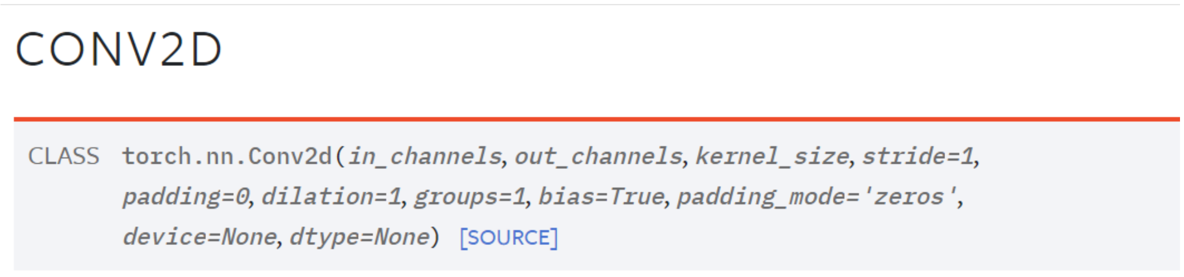
10	12	12
18	16	16
12	9	3

padding=1：



```python
#关于卷积操作
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], #卷积核
                       [0, 1, 0],
                      [2, 1, 0]])

# input – input tensor of shape(minibatch,in_channels,iH,iW)
input = torch.reshape(input, (1, 1, 5, 5)) #改变数组形状
kernel = torch.reshape(kernel, (1, 1, 3, 3))

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

P18-神经网络卷积层

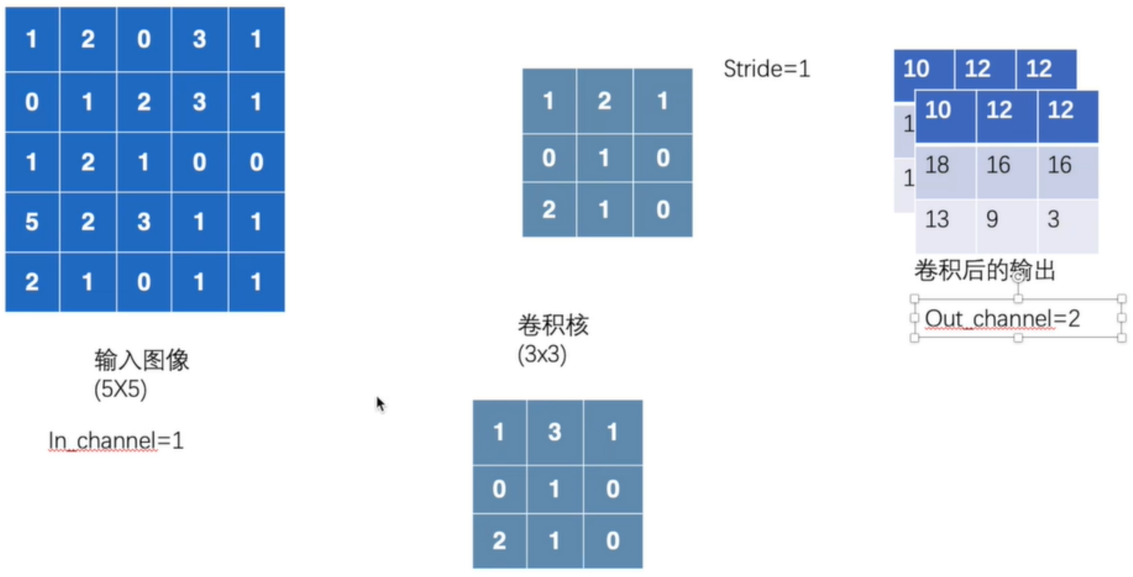


CONV2D

CLASS torch.nn.Conv2d(*in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None*) [SOURCE]

参数介绍：in_channels，输入通道数；out_channels，输出通道数；kernel_size，卷积核大小；

kernel_size，stride，padding这几个参数调整较多。



out_channels=2,两个卷积核，分别对输入图像进行卷积，得到两个输出，把这两个输出进行叠加后最终输出。

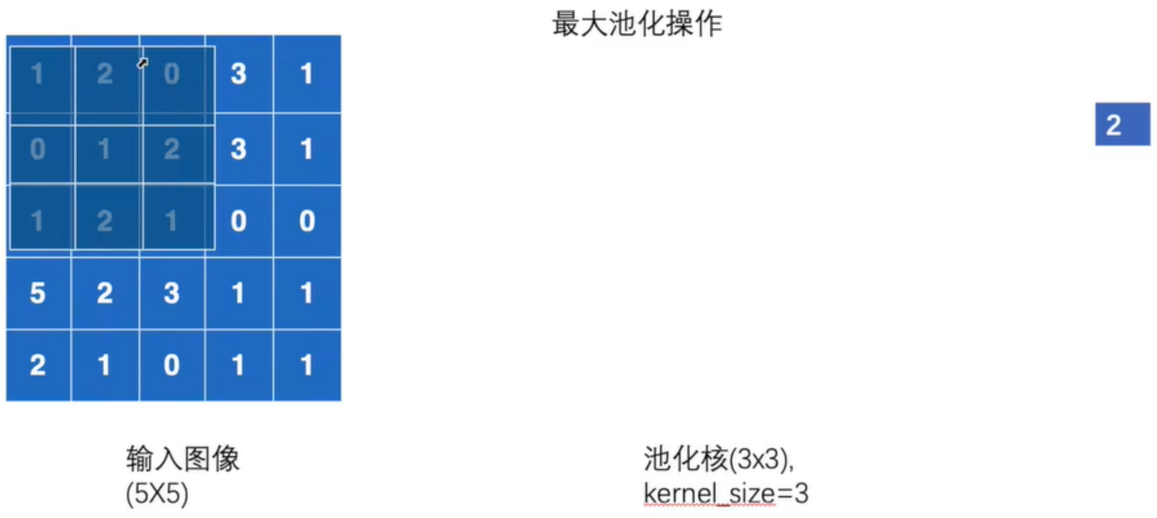
```
import torchvision
from torch import nn
from torch.nn import Conv2d
from torch.utils.data import DataLoader

dataset = torchvision.datasets.CIFAR10("./dataset3", train=False,
transform=torchvision.transforms.ToTensor(),
                                       download=True)

dataloader = DataLoader(dataset, batch_size=64)

class Test1(nn.Module):
    def __init__(self):
        super(Test1, self).__init__()
        self.conv1 = Conv2d(in_channels=3, out_channels=6, kernel_size=3,
stride=1, padding=0) #卷积层

    def forward(self, x): #前向传播
        x = self.conv1(x)
        return x

test1 = Test1()

for data in dataloader:
    imgs, targets = data
    output = test1(imgs)
    print(imgs.shape)
    print(output.shape)
```

输出:

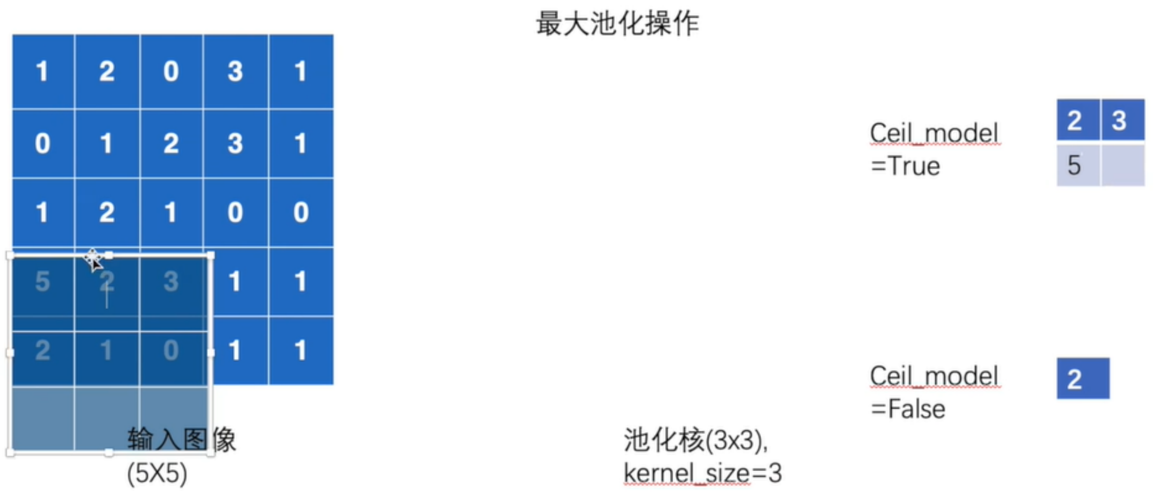
```
D:\Anacoada\anacoada\envs\pytorch\python.exe
F:/PycharmProject/learn_pytorch/P18_nn_conv2d.py
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
./dataset3\cifar-10-python.tar.gz
170499072it [01:37, 1754604.88it/s]
Extracting ./dataset3\cifar-10-python.tar.gz to ./dataset3
torch.Size([64, 3, 32, 32])
torch.Size([64, 6, 30, 30])
torch.Size([64, 3, 32, 32])
torch.Size([64, 6, 30, 30])
```

P19-神经网络最大池化作用

pooling layers



池化核是3*3大小，第一步取的是**覆盖范围内最大的数**2.



最大池化的目的：保留数据的特征，同时减少数据量。

```
import torch
from torch import nn
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))

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

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

test = Test()
output = test(input)
print(output)
```

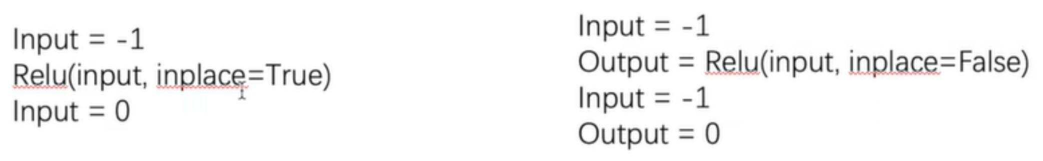
输出：

```
D:\Anacoada\anacoada\envs\pytorch\python.exeF:/PycharmProject/learn_pytorch/P19_
nn_maxpool.py
tensor([[[[2., 3.],
          [5., 1.]]]])
```

P20-神经网络-非线性激活

Non-linear Activations

RELU（）的参数inplace：


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

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

非线性变化最主要目的是给网路中引入一些非线性特征，非线性特征越多，才能训练出符合各种曲线的模型。

```
#非线性激活
import torch
from torch import nn
from torch.nn import ReLU

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

input = torch.reshape(input,(-1,1,2,2))
#print(input.shape)

class Test(nn.Module):
    def __init__(self):
        super(Test, self).__init__()
        self.relu1 = ReLU()

    def forward(self, input):
        output = self.relu1(input)
        return output

test = Test()
output = test(input)
print(output)
```

输出:

```
D:\Anacoada\anacoada\envs\pytorch\python.exeF:/PycharmProject/learn_pytorch/P20_
nn_relu.py
tensor([[[[1., 0.],
          [0., 3.]]]])
```

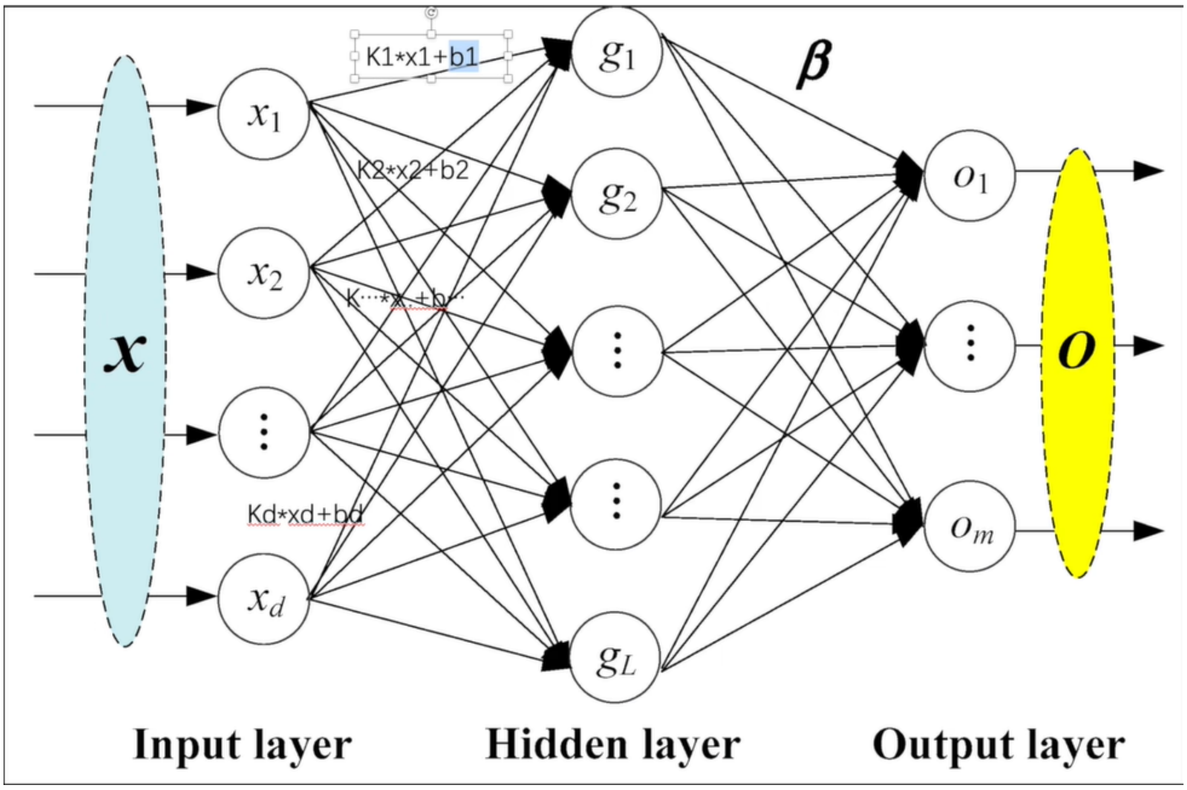
P21-神经网络线性层及其它层介绍

Linear Layers

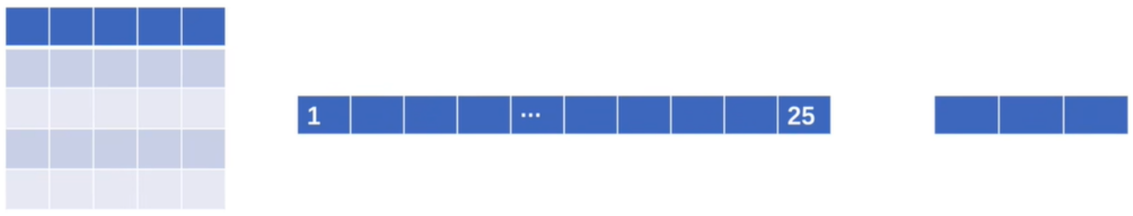
```
torch.nn.Linear(in_features, out_features, bias=True, device=None, dtype=None)
```

线性层又称为**全连接层，其每个神经元与上一层所有神经元相连，实现对前一层的线性组合，线性变换。**

in_features 就是输入x，out_features 是线性层的输出是上图的g



对图片操作

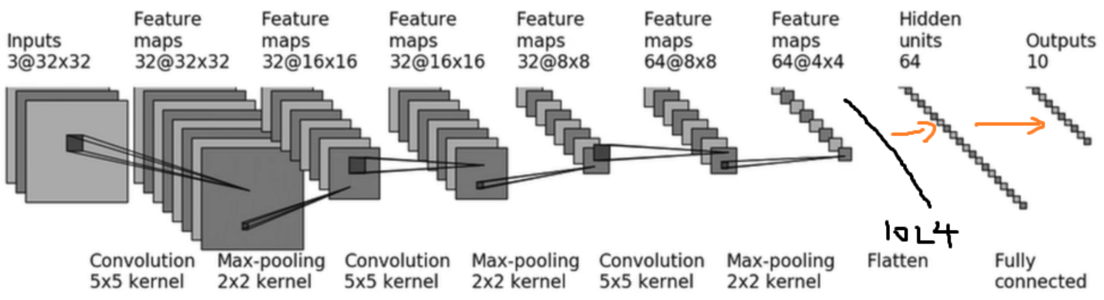


```python
import torch

import torchvision
from torch import nn
from torch.nn import Linear
from torch.utils.data import DataLoader

dataset = torchvision.datasets.CIFAR10("/dataset3", train=False,
transform=torchvision.transforms.ToTensor(),
                                       download=True)

dataloader = DataLoader(dataset, batch_size=64)

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

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

test = Test()

for data in dataloader:
    imgs, targets = data
    print(imgs.shape)
    output = torch.reshape(imgs, (1,1,1,-1))
    print(output.shape)
    output = test(output)
    print(output.shape)
```

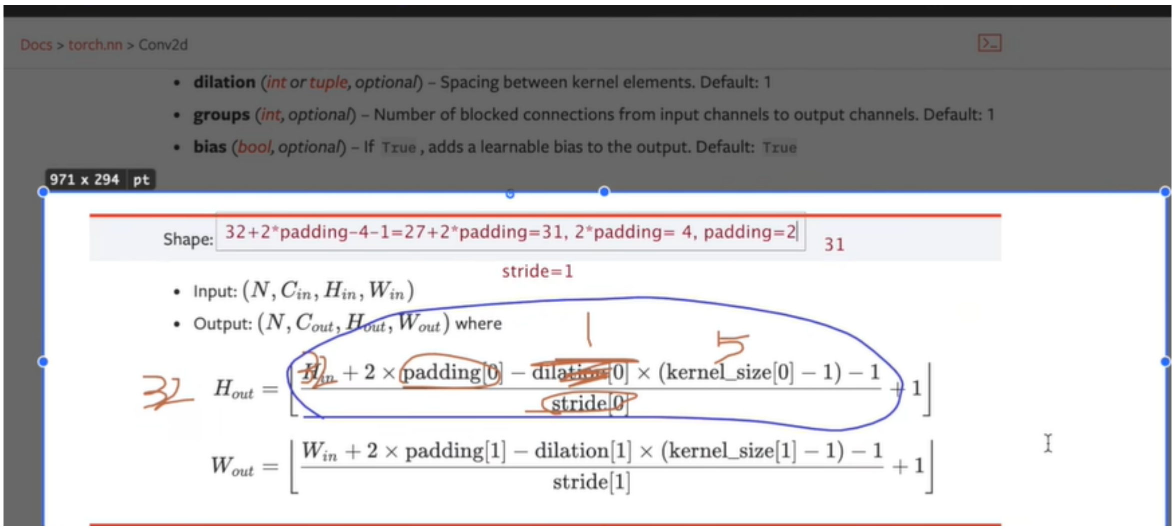
输出:

```
Files already downloaded and verified
torch.Size([64, 3, 32, 32])
torch.Size([1, 1, 1, 196608])
torch.Size([1, 1, 1, 10])
```

P22-神经网络-搭建小实战与Sequential的使用



CIFAR 10 model 结构



Docs > torch.nn > Conv2d

- **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`

971 x 294 pt

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

stride=1

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

$$32 \quad H_{out} = \left\lfloor \frac{\overset{32}{H_{in}} + 2 \times \text{padding}[0] - \overset{1}{\text{dilation}}[0] \times (\overset{5}{\text{kernel\_size}}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

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

```
import torch

from torch import nn
from torch.nn import Conv2d, MaxPool2d, Flatten, Linear, Sequential


class Test(nn.Module):
    def __init__(self):
        super(Test, 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)
        self.model1 = Sequential(
            Conv2d(3, 32, 5, padding=2),
            MaxPool2d(2),
            Conv2d(32, 32, 5, padding=2),
            MaxPool2d(2),
            Conv2d(32, 64, 5, padding=2),
            MaxPool2d(2),
            Linear(1024, 64),
            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)
        x = self.model1(x)
        return x

test = Test()
print(test)

input = torch.ones((64, 3, 32, 32))
output = test(input)
print(output.shape)
```

输出：

```
Test(
  (model1): Sequential(
    (0): Conv2d(3, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (2): Conv2d(32, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (4): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (6): Flatten(start_dim=1, end_dim=-1)
    (7): Linear(in_features=1024, out_features=64, bias=True)
    (8): Linear(in_features=64, out_features=10, bias=True)
  )
)
torch.Size([64, 10])
```

P23-损失函数与反向传播

Loss Function

Loss是为了衡量神经网络实际输出与预测输出的差距，

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

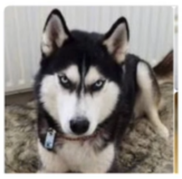
1. 计算实际输出和目标之间的差距
2. 为我们更新输出提供一定的依据（反向传播）

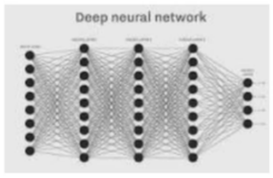
CROSSENTROPYLOSS:

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))

L1Loss:

```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)
```

实际神经网络中的loss:

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

dataset = torchvision.datasets.CIFAR10("/dataset4", train=False,
transform=torchvision.transforms.ToTensor(),
                                       download=True)

dataloader = DataLoader(dataset, batch_size=1)

class Test(nn.Module):
    def __init__(self):
        super(Test, self).__init__()
        self.model1 = Sequential(
            Conv2d(3, 32, 5, padding=2),
            MaxPool2d(2),
            Conv2d(32, 32, 5, padding=2),
            MaxPool2d(2),
            Conv2d(32, 64, 5, padding=2),
            MaxPool2d(2),
            Flatten(),
            Linear(1024, 64),
            Linear(64, 10)
        )

    def forward(self, x):
        x = self.model1(x)
        return x

loss = nn.CrossEntropyLoss()
test = Test()
for data in dataloader:
    imgs, targets = data
    outputs = test(imgs)
    # #输出一下outputs和targets看选什么损失函数
    # print(outputs) #输出当前图片是不同class的概率
    # print(targets) #图片所属正确class
    result_loss = loss(outputs, targets)
    print(result_loss)
```

输出结果:

```
D:\Anacoada\anacoada\envs\pytorch\python.exe
F:/PycharmProject/learn_pytorch/P24_nn_network.py
Files already downloaded and verified
tensor(2.1993, grad_fn=<NllLossBackward>)
tensor(2.1683, grad_fn=<NllLossBackward>)
tensor(2.1712, grad_fn=<NllLossBackward>)
tensor(2.3200, grad_fn=<NllLossBackward>)
tensor(2.3357, grad_fn=<NllLossBackward>)
tensor(2.3579, grad_fn=<NllLossBackward>)
tensor(2.4193, grad_fn=<NllLossBackward>)
tensor(2.3555, grad_fn=<NllLossBackward>)
tensor(2.1963, grad_fn=<NllLossBackward>)
tensor(2.3833, grad_fn=<NllLossBackward>)
tensor(2.3300, grad_fn=<NllLossBackward>)
tensor(2.3775, grad_fn=<NllLossBackward>)
tensor(2.3010, grad_fn=<NllLossBackward>)
....................
```

P24-优化器

调用损失函数的反向传播函数，利用反向传播可以求出每个需要调节的参数对应的梯度，优化器可以通过这个梯度，对参数进行调整，实现误差降低的目的。

学习率：

http://www.pointborn.com/article/2020/10/6/989.html

例子：

```python
import torch

import torchvision
from torch import nn
from torch.nn import Conv2d, MaxPool2d, Flatten, Linear, Sequential
from torch.utils.data import DataLoader

dataset = torchvision.datasets.CIFAR10("/dataset4", train=False,
transform=torchvision.transforms.ToTensor(),
                                       download=True)

dataloader = DataLoader(dataset, batch_size=1)

class Test(nn.Module):
    def __init__(self):
        super(Test, self).__init__()
        self.model1 = Sequential(
            Conv2d(3, 32, 5, padding=2),
            MaxPool2d(2),
            Conv2d(32, 32, 5, padding=2),
            MaxPool2d(2),
            Conv2d(32, 64, 5, padding=2),
            MaxPool2d(2),
            Flatten(),
            Linear(1024, 64),
            Linear(64, 10)
        )

    def forward(self, x):
        x = self.model1(x)
        return x

loss = nn.CrossEntropyLoss()
test = Test()

optim = torch.optim.SGD(test.parameters(),lr=0.005) #设置优化器

for epoch in range(5): #所有数据作为一轮，进行20轮
    running_loss = 0.0
    for data in dataloader:
        imgs, targets = data
        outputs = test(imgs)
        result_loss = loss(outputs, targets)
        optim.zero_grad()#把网络模型中每个可调节参数的梯度都设置为0
        result_loss.backward() # 求出每个参数的梯度
        optim.step() # 调优
        running_loss = running_loss + result_loss
    print(running_loss)
```

输出结果:

```
D:\Anacoada\anacoada\envs\pytorch\python.exe
F:/PycharmProject/learn_pytorch/P24_nn_loss_optim.py
Files already downloaded and verified
tensor(18800.0195, grad_fn=<AddBackward0>)
tensor(15474.7842, grad_fn=<AddBackward0>)
tensor(13704.0908, grad_fn=<AddBackward0>)
tensor(12341.2627, grad_fn=<AddBackward0>)
tensor(11256.3564, grad_fn=<AddBackward0>)

Process finished with exit code 0
```

P25-现有网络模型的使用和修改

P26-模型的保存与读取

保存:

```
import torch
import torchvision

vgg16 = torchvision.models.vgg16(pretrained=False)
# 保存方式1
# 保存模型结构和参数
torch.save(vgg16, "vgg16_method1.pth") # 参数是要保存的路径

# 保存方式2,官方推荐
# 把模型中的参数保存成python中的字典格式
torch.save(vgg16.state_dict(), "vgg16_method2.pth")
```

读取:

```python
import torch
import torchvision

# 方式1-》保存方式1，加载模型

model = torch.load("vgg16_method1.pth") # 参数是模型保存的路径

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

P27-完整的模型训练套路1

Q：什么是梯度下降

A:

梯度下降拆解为梯度+下降，那么梯度可以理解为导数（对于多维可以理解为偏导），那么合起来变成了：导数下降，那问题来了，导数下降是干什么的？答案：梯度下降就是用来求某个函数最小值时自变量对应取值

其中这句话中的某个函数是指：损失函数（cost/loss function），直接点就是误差函数。

一个算法不同参数会产生不同拟合曲线，也意味着有不同的误差。

损失函数就是一个自变量为算法的参数，函数值为误差值的函数。所以梯度下降就是找让误差值最小时候算法取的参数。

一般用损失函数的值来衡量这个误差，所以损失函数的误差值越小说明拟合效果越好。

Q：什么是反向传播

A:

反向传播算法是目前用来训练人工神经网络的最常用且最有效的算法。

反向传播工作原理就是：

（1）前向传播：将训练集数据输入到ANN的输入层，经过隐藏层，最后到达输出层并输出结果。【输入层—隐藏层–输出层】
（2）反向传播：由于ANN的输入结果与输出结果有误差，则计算估计值与实际值之间的误差，并将该误差从输出层向隐藏层反向传播，直至传播到输入层。【输出层–隐藏层–输入层】
（3）权重更新：在反向传播的过程中，根据误差调整各种参数的值；不断迭代上述过程，直至收敛。（使得损失函数越来越小，整体模型也越来越接近于真实值）

Q：为什么要进行梯度清零

A:

因为grad在反向传播的过程中是==累加==的，也就是说上一次反向传播的结果会对下一次的反向传播的结果造成影响，则意味着每一次运行反向传播，梯度都会累加之前的梯度，所以一般在反向传播之前需要把梯度清零。

清零使用的方法是.grad.data.zero_()