

Lecture 0: PyTorch 总结

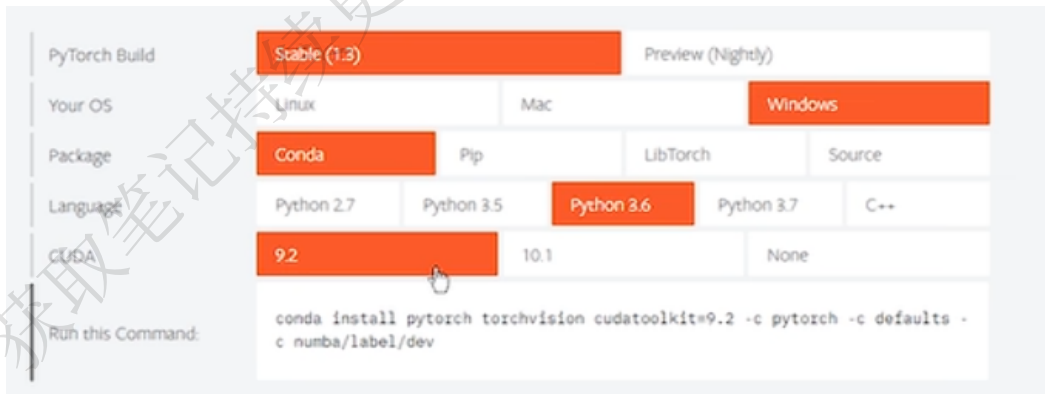
安装

概念：基本概念

- 概念：什么是虚拟环境
 - 环境可以理解成一个房子，比如初始环境为 `Base`，假设当前包版本为 0.4，如果需要 1.0 的版本，就可以再新建一个房子，即新建了一个环境；
 - 在这个新房子里面，可以安装不同的版本的包；
- 方式：创建环境
 - `conda create -n pytorch python=3.6`
 - 以下是需要安装的包：

```
certifi      pkgs/main/win-64::certifi-2019.9.11-py36_0
pip          pkgs/main/win-64::pip-19.3.1-py36_0
python       pkgs/main/win-64::python-3.6.9-h5500b2f_0
setuptools   pkgs/main/win-64::setuptools-41.6.0-py36_0
sqlite       pkgs/main/win-64::sqlite-3.30.1-he774522_0
vc           pkgs/main/win-64::vc-14.1-h0510ff6_4
vs2015_runtime pkgs/main/win-64::vs2015_runtime-14.16.27012-hf0eaf9b_0
wheel        pkgs/main/win-64::wheel-0.33.6-py36_0
wincertstore pkgs/main/win-64::wincertstore-0.2-py36h7fe50ca_0
```

- 方式：激活环境
 - `conda activate pytorch`
- 方式：安装 `Pytorch`
 - (1) 选择版本



- (2) 在环境中安装 `PyTorch`
 - 将粘贴的指令在环境中复制、执行安装；

```
(pytorch) C:\Users\Zhiyao>conda install pytorch torchvision cudatoolkit=9.2 -c pytorch -c defaults -c numba/label/dev
Collecting package metadata (repodata.json): -
```

基础概念

两个重要函数

概念：两个重要函数

- `dir()` 函数：

- 能让我们知道工具箱以及工具箱中的分隔区有什么东西；

```
In[4]: dir(torch)
```

```
Out[4]:
```

```
['AVG',
 'AggregationType',
 'Argument',
 'ArgumentSpec',
 'BFloat16Storage',
 'BFloat16Tensor',
 'BenchmarkConfig',
 • 'BenchmarkExecutionStats',
```

```
In[6]: dir(torch.cuda.is_available)
```

```
Out[6]:
```

```
['__annotations__',
 '__call__',
 '__class__',
 '__closure__',
 '__code__',
 '__defaults__',
 '__delattr__',
 '__dict__',
 • '__dir__',
```

- 前后的双下划线表示是一个变量，也是一个具体的函数；

- `help()` 函数：

- 能让我们知道每个工具是如何使用的，工具的使用方法；

```
In[7]: help(torch.cuda.is_available)
```

```
Help on function is_available in module torch.cuda:
```

```
is_available()
```

- `Returns a bool` indicating if `CUDA` is currently available.

数据加载

概念： `Dataset` 与 `Dataloader`

- 概念： `Dataset`

- 在进行训练时，首先会有大量的数据，这些数据假设目前是垃圾，而 `Dataset` 则是从这些垃圾当中获取数据，并且可以为其进行编号；
- 即：提供一种方式去获取数据以及其 `label`；
- 比如：1. 如何获得每一个数据以及其 `label`；2. 告诉我们总共有多少的数据；

- 概念： `Dataloader`

- 假设当前对数据进行了打包，而 `Dataloader` 主要是可以为后背的网络提供不同的数据形式；
- 方式： `Dataset` 的使用
 - 导入： `from torch.utils.data import Dataset`
 - 使用：

```
class Dataset(builtins.object)
|   An abstract class representing a :class:`Dataset`.
|
|   All datasets that represent a map from keys to data samples should subclass
|   it. All subclasses should override :meth:`~__getitem__`, supporting fetching a
|   data sample for a given key. Subclasses could also optionally overwrite
|   :meth:`~__len__`, which is expected to return the size of the dataset by many
|   :class:`~torch.utils.data.Sampler` implementations and the default options
|   of :class:`~torch.utils.data.DataLoader`.
|
|   .. note::
|       :class:`~torch.utils.data.DataLoader` by default constructs a index
|       sampler that yields integral indices. To make it work with a map-style
|       dataset with non-integral indices/keys, a custom sampler must be provided.
|
|   Methods defined here:
|
|   __add__(self, other)
|
|   __getitem__(self, index)
|
|   -----
|   Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)
```

```
"""An abstract class representing a :class:`Dataset`.

All datasets that represent a map from keys to data samples should subclass
it. All subclasses should override :meth:`~__getitem__`, supporting fetching a
data sample for a given key. Subclasses could also optionally overwrite
:meth:`~__len__`, which is expected to return the size of the dataset by many
:class:`~torch.utils.data.Sampler` implementations and the default options
of :class:`~torch.utils.data.DataLoader`.

.. note::
    :class:`~torch.utils.data.DataLoader` by default constructs a index
    sampler that yields integral indices. To make it work with a map-style
    dataset with non-integral indices/keys, a custom sampler must be provided.
"""
```

- 补充： `Image` 包的使用

```
In[2]: from PIL import Image
In[3]: img_path = "C:\\Users\\Zhiyao\\Desktop\\learn_torch\\dataset\\train\\ants\\0013035.jpg"
In[4]: img = Image.open(img_path)
In[5]: img.show()
```

代码： `Dataset` 实战

- 获得所有图片的地址：

```
dir_path = 'dataset/train/ants'
import os
img_path_list = os.listdir(dir_path)
# 获得图片名称
```

- 代码:

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

class MyData(Dataset):

    # 初始化
    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)

    # 获得图片以及其标签
    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
```

- 使用数据集:

```
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 的使用

代码

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

writer = SummaryWriter("logs")
image_Path = "路径名"
img_PIL = Image.open(image_Path)
img_Array = np.array(img_PIL)
```

Transform

Tensor 的概念

- 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
- 示例: transforms 该如何使用 (python)

```
print(tensor_img)
```

```
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, ..., 0.3216, 0.3216, 0.3216],\n ..., \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, 0.6275, 0.9294]]]])\n> 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, 0.6275, 0.9294]]]])\n  ❶ _backward_hooks = (NoneType) None\n  ❶ _base = (NoneType) None\n  ❶ _cdata = (int) 2501405032288\n  ❶ _grad = (NoneType) None\n  ❶ _grad_fn = (NoneType) None\n  ❶ _version = (int) 0\n> 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, ..., 0.3216, 0.3216, 0.3216],\n ..., \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, 0.6275, 0.9294]]]])\n> device = (device) cpu\n> dtype = (dtype) torch.float32\n  ❶ grad = (NoneType) None\n  ❶ grad_fn = (NoneType) None\n  ❶ is_cuda = (bool) False\n  ❶ is_leaf = (bool) True\n  ❶ is_mkldnn = (bool) False\n  ❶ is_quantized = (bool) False\n  ❶ is_sparse = (bool) False\n> layout = (layout) torch.strided\n  ❶ name = (NoneType) None\n> names = (tuple) (None, None, None)\n  ❶ ndim = (int) 3\n  ❶ output_nr = (int) 0\n  ❶ requires_grad = (bool) False\n> shape = (Size) torch.Size([3, 512, 768])
```

- 由此可知, `tensor` 就是一个包装了反向神经网络的一些基础的参数的对象;
- 在训练时需要将图片转换成 `tensor` 后再进行训练;

常见 Transform 方法

- 概念:

- **Transform** 就是接收一定格式的图片，然后经过一些方法后、得到想要的结果；

- 因此 `Transform` 就是一系列的工具库；
- 重点需要关注其输入、输出以及作用；
- 概念：输入
 - PIL 是 `Image.open` 得到的类型；
- 概念：输出
 - `tensor` 是 `ToTensor` 得到的对象；

概念：常用方法

- 概念： `Compose` 类
 - 将一系列 `Transform` 操作合在一起；
 - 比如：

```
transforms.Compose([
    transforms.CenterCrop(10),
    transforms.ToTensor(),
    ])
```

- 概念： `ToTensor`
 - 这个方法的作用就是将 `PIL Image` or `numpy.ndarray` 转换为 `tensor`；
 - 示例：

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

- 概念： `Normalize`
 - 归一化，将一个 `Tensor` 的数据对象，随着其平均值或者标准差，将其每个信道的输入进行标准化：

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

- 示例：

```
# 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)
```

- 概念： `Resize`
 - 更改尺寸，如果传入的是 `(w,h)`，则将更改其宽高为 `(w,h)`。如果传入的是一个 `int`，则将对 `PIL Image` 进行等比缩放；
 - 示例：

```

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

```

• 概念: Compose

- 将多个变换进行组合;
- Compose() 中的参数需要是一个列表, 在 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, ..])
- 示例:

```

# Compose - resize - 2
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)

```

数据集的使用

torchvision 中数据集的使用

概念: Dataset - 数据集参数介绍

- 概念:
 - root: 数据集的位置
 - train: 如果为真, 则表示当前数据集为训练集。如果为假, 则表示创建了一个测试集;
 - transform: 对数据集中的图片的变换可以写在这里;
 - target_transform: 对目标设置变换;
 - download: 如果使用 torchvision 的数据集, 可以直接下载;
- 图示:

- **root** (string) – Root directory of dataset where directory `cifar-10-batches-py` exists or will be saved to if download is set to True.
- **train** (bool, optional) – If True, creates dataset from training set, otherwise creates from test set.
- **transform** (callable, optional) – A function/transform that takes in an PIL image and returns a transformed version. E.g, `transforms.RandomCrop`
- **target_transform** (callable, optional) – A function/transform that takes in the target and transforms it.
- **download** (bool, optional) – If true, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.

代码

```

import torchvision

# 设置变换
datasets_transform = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor()
])

# 训练集
train_set =
torchvision.datasets.CIFAR10(root="./dataset", train=True, transform=datasets_trans
form, download=True)

# 测试集
test_set =
torchvision.datasets.CIFAR10(root="./dataset", train=False, transform=datasets_tran
sform, download=True)

# 将前10张图片在Tensorboard上显示
writer = SummaryWriter("p10")
for i in range(10):
    img, target = test_set[i]
    writer.add_image("test_set", img, i)

```

Dataloader 的使用

概念： Dataloader

- 概念：
 - Dataset 是指出数据在哪、下载下来；
 - 而 Dataloader 是将数据从数据集中取出来、如何取；
 - 其中参数主要就是设置 dataset，即设置数据集。其他参数都有默认值；
 - shuffle 是打乱的意思， num_workers 表示是否使用多进程，0 表示使用主进程来加载，如果 num_workers>0 则多进程；
- 图示：

- **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.

代码

```
import torchvision

from torch.utils.data import DataLoader

test_set =
torchvision.datasets.CIFAR10(root="./dataset", train=False, transform=torchvision.t
ransforms.ToTensor())

test_loader = DataLoader(dataset=test_set, batch_size = 4, num_work = 0, drop_last =
False)

# 获得当前四张图片
for data in test_loader:
    imgs, targets = data
    print(imgs.shape)
    print(targets)
```

- 补充:
 - `batch_size=4` 表示的意思就是当前一次取出四张图片，并将其打包；
 - 其将会 `image 0、1、2、3` 打包；
- 注意:
 - 当前 `Dataloader` 默认是随即抓取，比如 `batch_size=4` 时、会随机抓取四张图片；

神经网络搭建

引入

- 神经网络相关的包主要在 `torch.nn` 包当中；

nn.Module 的使用

概念: `nn.Module`

- 图示:

- `Model` 继承自 `Module` 类, 其定义了两个函数:

```
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))
```

- 概念:

- 这是一个所有神经网络模块的基类;
- 你的模型应当继承自此类;

- 概念: `__init__` - 表示初始化的函数

- `super(Model, self).__init__()` 这一步是必须要的, 表示初始化父类的函数;
- 下面的内容是自己定义的初始化行为;

- 概念: `forward` - 表示前向传播

- 定义了每一次计算的计算图;
- 神经网络获得了一个输入后、经过 `forward` 得到输出;
- `self` 指当前的类实例, `x` 表示输入;
- 比如 `x=F.relu(self.conv1(x))` 表示对输入进行一次卷积, 然后再对其进行非线性处理; 然后再对其进行一次卷积、然后进行一次非线性处理, 最后返回;

代码

```
from torch import nn

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

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

test = Test()
x = torch.tensor(1.0)
output = test(x)
print(output)
```

卷积层

概念：卷积层

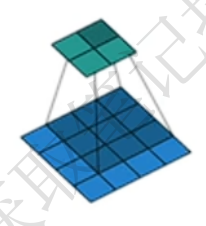
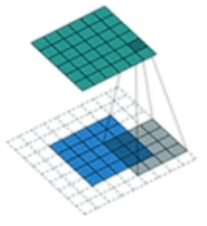
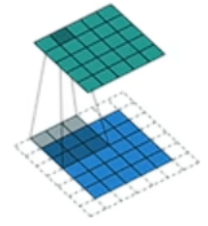
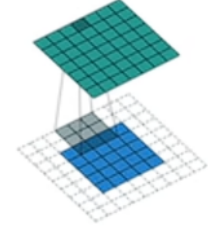
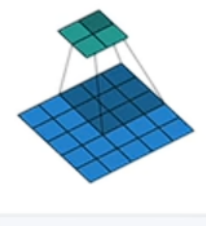
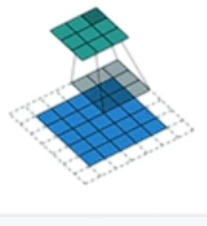
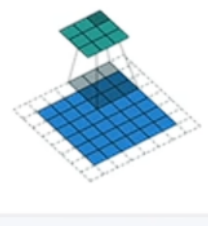
- 图示：一维、二维、三维卷积层

<code>nn.Conv1d</code>	Applies a 1D convolution over an input signal composed of several input planes.
<code>nn.Conv2d</code>	Applies a 2D convolution over an input signal composed of several input planes.
<code>nn.Conv3d</code>	Applies a 3D convolution over an input signal composed of several input planes.
<code>nn.ConvTranspose1d</code>	Applies a 1D transposed convolution operator over an input image composed of several input planes.

- 概念：二维卷积

- 参数：
 - `in_channels`: 输入的通道数
 - `out_channels`: 输出的通道数
 - `kernel_size`: 卷积核大小;
 - `int` 或者 `tuple`;
 - `stride=1`
 - 步径的大小
 - `padding=6`
 - `dilation=1`
 - `bias=True`: 表示偏置值, 一般默认为真即可

- 图示:

			
No padding, no strides	Arbitrary padding, no strides	Half padding, no strides	Full padding, no strides
			
No padding, strides	Padding, strides	Padding, strides (odd)	

- 举例：
 - `in_channel=1`, 因此输入图像只有一通道;
 - `out_channel=2`, 因此输出的图像为两通道, 并且需要使用两个卷积核;

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

输入图像
(5X5)

In_channel=1

1	2	1
0	1	0
2	1	0

卷积核
(3x3)

1	3	1
0	1	0
2	1	0

Stride=1

10	12	12
10	12	12
18	16	16
13	9	3

卷积后的输出

Out_channel=2

代码

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

class Test(nn.Module):
    def __init__(self):
        super(Test, self).__init__()
        self.conv1 = Conv2d(3, 6, 3, stride=1, padding=0)

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

test = Test()

# 使用模型
for data in dataloader:
    imgs, targets = data
    output = test(imgs)
    # 对图片进行reshape
    output = torch.reshape(output, (-1, 3, 30, 30))
```

池化层

概念：池化层

- 引入：
 - 由于卷积层的对位置的敏感性，导致其有时候需要平移不变性；
 - 而比如 `2*2` 的最大池化层，可以容忍一个像素的位移：

```

[ [ 1.  1.  0.  0.  0.
  [ 1.  1.  0.  0.  0.
  [ 1.  1.  0.  0.  0.
  [ 1.  1.  0.  0.  0.

```

•

• 这里面的边缘往左或者往右偏移一个像素、都可以依然得到一样的特征；

• 同样有窗口大小、填充、和步幅作为超参数；

• 池化层中没有需要学习的参数，并且其作用域多输入的通道时、对每一个通道做一次池化层，因此输出的通道数等于输入的通道数；

• 分类：

• 最大池化层：每个窗口中最强的模式信号；

• 平均池化层：将最大池化层中的“最大”操作替换为“平均”；

• 图示：

• `MaxPool`：最大池

• `MaxUnpool`：上采样

• `AvgPool`：平均采样

`nn.MaxPool1d`

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

`nn.MaxPool2d`

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`

Computes a partial inverse of `MaxPool1d`.

`nn.MaxUnpool2d`

Computes a partial inverse of `MaxPool2d`.

`nn.MaxUnpool3d`

Computes a partial inverse of `MaxPool3d`.

`nn.AvgPool1d`

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 of several input planes.

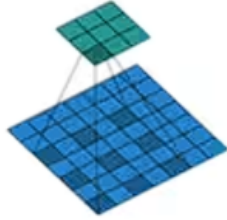
`nn.AvgPool3d`

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

•

• 概念：

- 参数:
 - 1. `kernel_size`: 池化大小;
 - 2. `stride`: 默认为 `kernel_size` 的大小;
 - 3. `padding`: 填充;
 - 4. `dilation`: 空洞卷积
 - 在卷积核和卷积核之间、插入位置



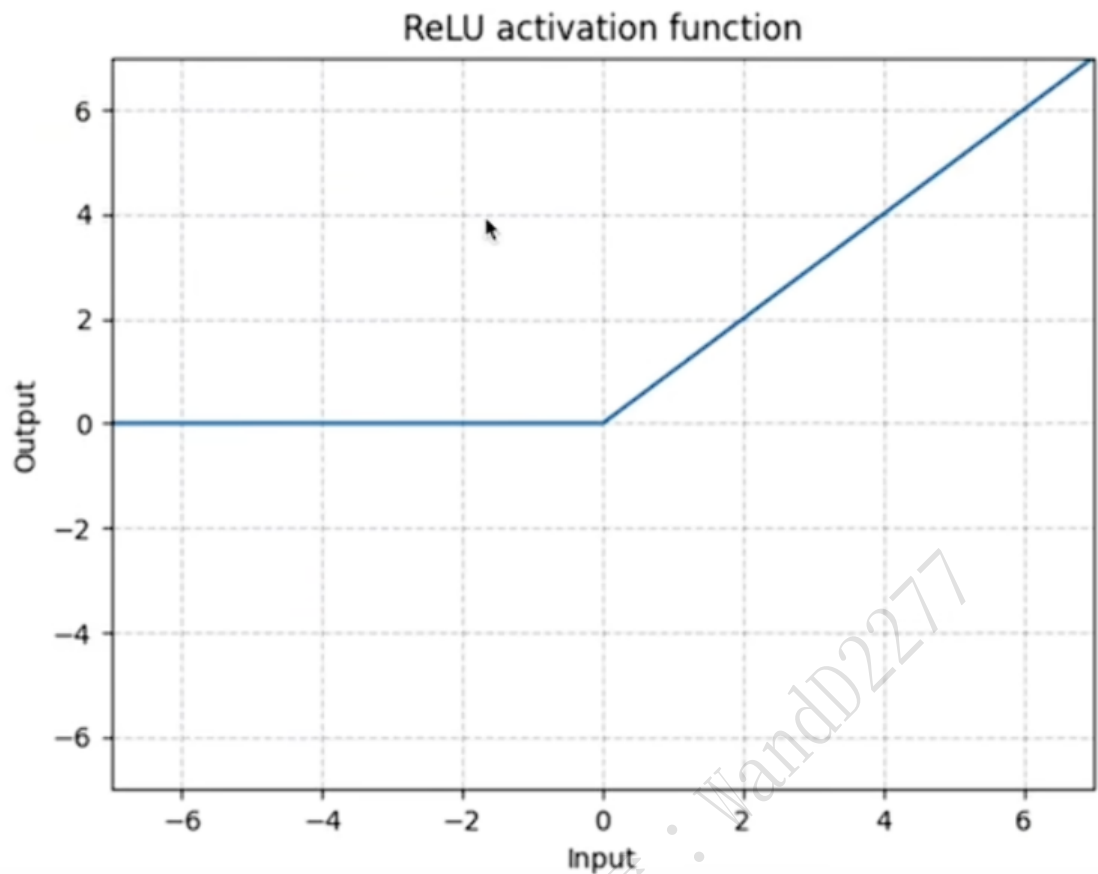
- 5. 模式
 - `ceil` 是小数向上取整;
 - `floor` 是小数向下取整;
- 概念: 最大池化
 - 选出当前池化核覆盖的范围当中、取值最大的一个;
- 示例:

```
>>> # pool of square window of size=3, stride=2
>>> m = nn.MaxPool2d(3, stride=2)
>>> # pool of non-square window
>>> m = nn.MaxPool2d((3, 2), stride=(2, 1))
>>> input = torch.randn(20, 16, 50, 32)
>>> output = m(input)
```

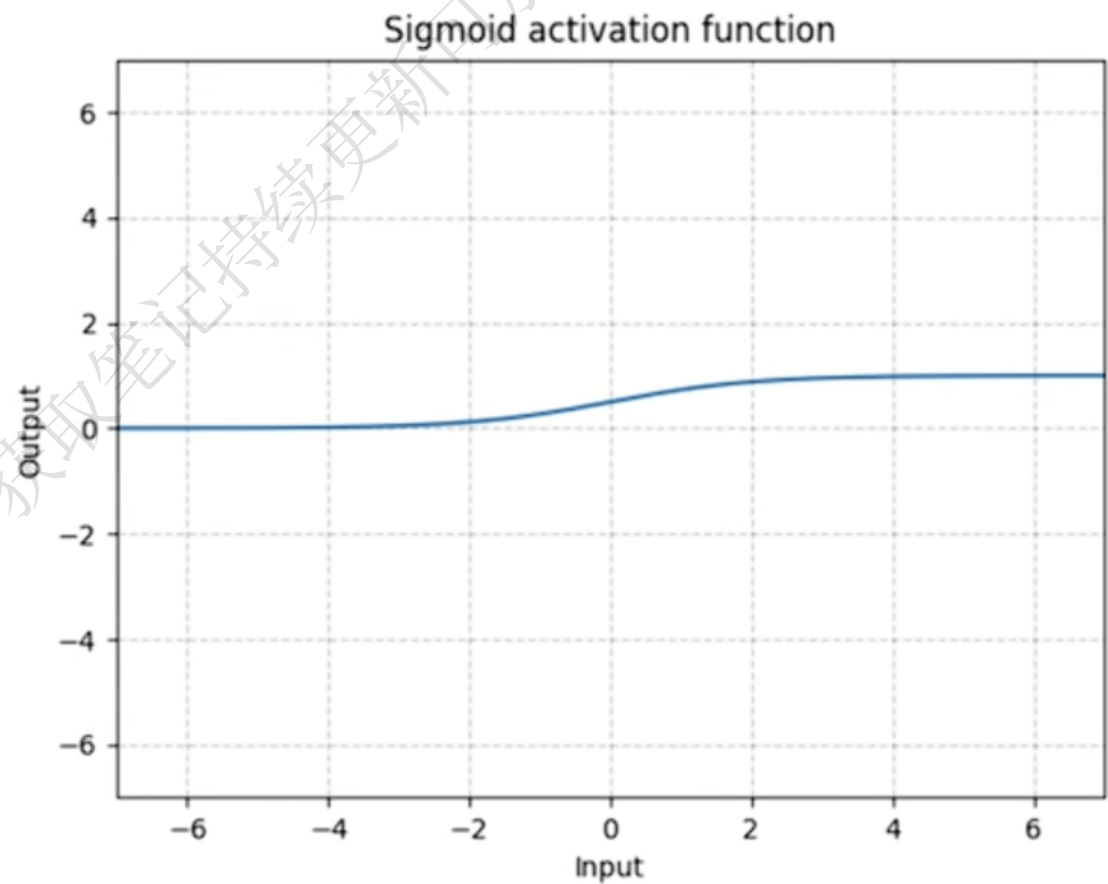
非线性激活

概念: 非线性激活

- 概念:
 - 非线性激活主要是为了给神经网络引入一些非线性的特质;
 - 最常见的就是 `ReLU`;



- 其输入为 N ，其输出的内容不做限制；
- 另一个比较常用的为 Sigmoid：



- 概念：什么是 inplace
 - 比如当前输入是 -1，当经过 ReLU 变换后、对 -1 进行了截断，因此 Input 变成了 0；

```
self.relu = nn.ReLU(inplace=False)
relu1 = nn.ReLU()
```

- 如果 `inplace=False`，则当前 Input 还是-1；
- 因此 `inplace` 就是替换的意思，一般令 `inplace = False`；

- 示例：

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

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

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

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

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

其他线性层

概念： 正则化层

- 图示：

<code>nn.BatchNorm1d</code>	Applies Batch Normalization over a 2D or 3D input (a mini-batch of 1D inputs with optional additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .
<code>nn.BatchNorm2d</code>	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 .
<code>nn.BatchNorm3d</code>	Applies Batch Normalization over a 5D input (a mini-batch of 3D inputs with additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .
<code>nn.GroupNorm</code>	Applies Group Normalization over a mini-batch of inputs as described in the paper Group Normalization .

- 概念：
 - 对输入采用正则化，采用正则化可以加速神经网络的训练速度；
- 示例：

```
>>> # With Learnable Parameters
>>> m = nn.BatchNorm2d(100)
>>> # Without Learnable Parameters
>>> m = nn.BatchNorm2d(100, affine=False)
>>> input = torch.randn(20, 100, 35, 45)
>>> output = m(input)
```

概念：Recurrent Layer

- 图示：

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

- 概念：
 - 文字识别所用到的层次；

概念：Transformer

- 图示：

<code>nn.Transformer</code>	A transformer model.
<code>nn.TransformerEncoder</code>	TransformerEncoder is a stack of N encoder layers
<code>nn.TransformerDecoder</code>	TransformerDecoder is a stack of N decoder layers
<code>nn.TransformerEncoderLayer</code>	TransformerEncoderLayer is made up of self-attn and feedforward network.
<code>nn.TransformerDecoderLayer</code>	TransformerDecoderLayer is made up of self-attn, multi-head-attn and feedforward network.

概念：线性层

- 图示：

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

- 概念：

概念：Dropout Layers

- 概念：
 - 随机的丢弃某些数据，用于防止过拟合；
- 图示：

<code>nn.Dropout</code>	During training, randomly zeroes some of the elements of the input tensor with probability <code>p</code> using samples from a Bernoulli distribution.
<code>nn.Dropout2d</code>	Randomly zero out entire channels (a channel is a 2D feature map, e.g., the j -th channel of the i -th sample in the batched input is a 2D tensor <code>input[i, j]</code>).
<code>nn.Dropout3d</code>	Randomly zero out entire channels (a channel is a 3D feature map, e.g., the j -th channel of the i -th sample in the batched input is a 3D tensor <code>input[i, j]</code>).
<code>nn.AlphaDropout</code>	Applies Alpha Dropout over the input.

概念: Linear - 线性层

- 概念:

- `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`

- 示例:

```
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)
```

输入

将imgs展平为一维

概念: Concat

- 概念: 其用于通道数的增加, 特征图通常具有形状为 `(channels,height,width)`, 如果要在通道维度上连接两个特征图, 可以使用 `concat` 操作, 结果将是形状为 `(channels1+channels2,height,width)`
- 在连接时, 两个张量的维度 (除了连接维度) 必须保持一致, 例如在通道维度上连接时, 其他维度的尺寸必须相同;

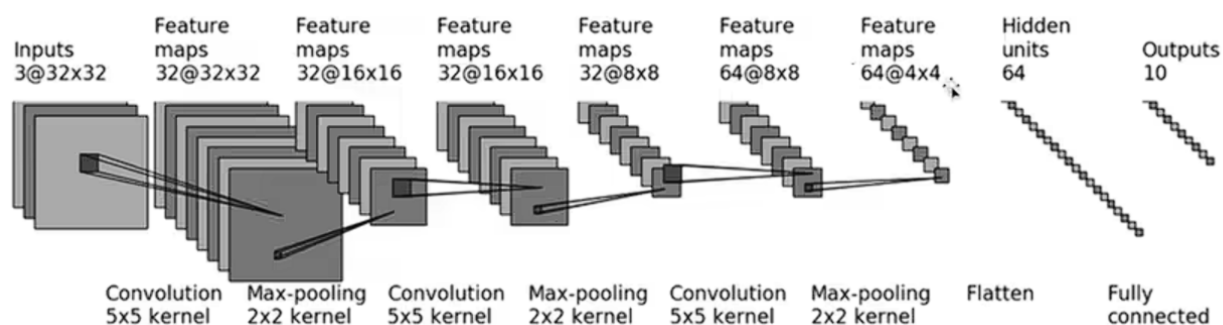
概念: Add

- 简单的相加操作, 相加操作需要具有相同的形状;

网络搭建示例

示例

- 图示: CIFAR 10 模型结构



代码

```
from torch import nn
from torch.nn import Conv2d

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

        # 输入的channel为3, 输出的channel为32, 卷积核为5*5.
        # 因为没有改变尺寸, 所以利用公式可以计算出来padding为2
        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)

        # 将其展平
        # 因为输入为64*4*4, 如果展平后为1024, 则1024=64*4*4
        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

test = Test()
input = torch.ones((64, 3, 32, 32))
```

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

代码：使用序列

```
from torch import nn
from torch.nn import Conv2d

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

        # 输入的channel为3, 输出的channel为32, 卷积核为5*5
        # 因为没有改变尺寸, 所以利用公式可以计算出来padding为2
        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.model(x)
        return x

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

损失函数与反向传播

损失函数

概念：损失函数

- 概念： `nn.L1Loss` 函数
 - 概念：
 - 比如 $X = 1, 2, 3; Y = 1, 2, 5$
 - 所以 $L1Loss = (0+0+2)/3=0.6$
 - 注意：
 - 一定要注意输入输出的大小;
 - 示例:

```
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)
```

- 概念: `nn.MSELoss` 函数

- 概念:

- 表示的是平方差;

- 比如 $X = 1, 2, 3; Y = 1, 2, 5$, 则 $MSE = (0+0+2^2)/3=4/3=1.333$

- 示例:

- `loss_mes = nn.MSELoss()`

- `result_mse = loss_mse(inputs, targets)`

- 概念: 交叉熵

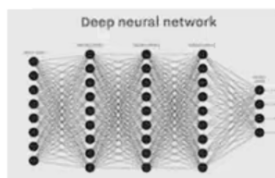
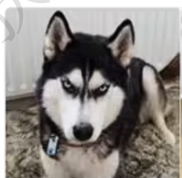
- 公式:

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

- 概念:

- 假设当前有一个三分类的问题:

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



Person, dog, cat
0, 1, 2

output
[0.1, 0.2, 0.3]

Target
1
class

x

$$\text{Loss}(x, \text{class}) = -0.2 + \log(\exp(0.1) + \exp(0.2) + \exp(0.3))$$

优化器

概念: 优化器

- 概念:

- 示例:

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

- 其中 params 是参数, 必须传入;
- 其中, SGD 和 Adam 是两种优化的算法;
- 其中 lr 是学习速率;
- 概念: 调用 step 方法
 - 使用优化器的 step 方法, 进行梯度更新;

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

- 其中 optimizer.zero_grad() 是将上一步计算得到的梯度清零, 以防上一步的影响;

代码

```
from torch import nn
from torch.nn import Conv2d

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

        # 输入的channel为3, 输出的channel为32, 卷积核为5*5
        # 因为没有改变尺寸, 所以利用公式可以计算出来padding为2
        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.model(x)
        return x

loss = nn.CrossEntropyLoss()
test = Test()
optim = torch.optim.SGD(test.parameters(), lr=0.01)

for data in dataloader:
```

```

imgs, targets = data
outputs = test(imgs)
result_loss = loss(outputs, targets)

# 先将上一步的反向传播的值清零
optim.zero_grad()
# 计算反向传播的值
result_loss.backward()
# 计算梯度，然后更新参数
optim.step()

```

模型的使用

模型的保存与加载

代码

```

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

# 保存方式1
# 除了保存模型的结构，还保存了模型的参数
torch.save(vgg16, "vgg16_method.pth")
# 加载模型
model = torch.load("vgg16_method.pth")

# -----
# 保存方式2
# 只保存了模型的参数，官方推荐这一种
torch.save(vgg16.state_dict(), "vgg16_method2.pth")

# 加载模型
vgg16 = torchvision.models.vgg16(pretrained=False)
vgg16.load_state_dict("vgg16_method2.pth")

```

模型训练套路

代码

```

import torchvision
from torch import nn

# 训练数据集
train_data =
torchvision.datasets.CIFAR10(root="./data", train=True, transform=torchvision.trans
forms.ToTensor(), download=True)

# 测试数据集
test_data =

```

```
torchvision.datasets.CIFAR10(root="../data",train=False,transform=torchvision.transforms.ToTensor(),download=True)
```

```
# 看数据集的大小
```

```
train_data_size = len(train_data)
```

```
test_data_size = len(test_data)
```

```
print("训练数据集长度为: {}".format(train_data_size))
```

```
print("测试数据集长度为: {}".format(test_data_size))
```

```
# 加载数据集
```

```
train_dataloader = DataLoader(train_data,batch_size = 64)
```

```
test_dataloader = DataLoader(test_data,batch_size = 64)
```

```
# 搭建神经网络
```

```
class Test(nn.Module):
```

```
    def __init__(self):
```

```
        super(Test,self).__init__()
```

```
        self.model = nn.Sequential(
```

```
            nn.Conv2d(3,32,5,1,2),
```

```
            nn.MaxPool2d(2),
```

```
            nn.Conv2d(32,32,5,1,2),
```

```
            nn.MaxPool2d(2),
```

```
            nn.Conv2d(32,64,5,1,2),
```

```
            nn.MaxPool2d(2),
```

```
            nn.Flatten(),
```

```
            nn.Linear(64*4*4,64),
```

```
            nn.Linear(64,10)
```

```
        )
```

```
    def forward(self,x):
```

```
        x = self.model(x)
```

```
        return x
```

```
if __name__ == '__main__':
```

```
    test = Test()
```

```
    input = torch.ones((64,3,32,32))
```

```
    output = test(input)
```

输出为: torch.size([64, 10]), 表示有64行数据, 每一行有10个数据, 这十个数据代表每一张图片在十个类别当中的概率

```
print(output.shape)
```

```
# 创建网络模型
```

```
test = Test()
```

```
# 损失函数
```

```
loss_fn = nn.CrossEntropyLoss()
```

```
# 优化器
```

```
## 随机梯度下降
```

```
learning_rate = 0.01
```

```
optimizer = torch.optim.SGD(test.parameters(),lr = learning_rate)
```

```

# 设置训练网络的一些参数
## 记录训练的步数
total_train_step = 0
# 记录测试的次数
total_test_step = 0
# 训练的轮数
epoch = 10

# 添加tensorBorad
writer = SummaryWriter("../logs_train")

for i in range(epoch):
    print("第{}论训练开始".format(i+1))

    # 训练步骤开始
    for data in train_dataloader:
        imgs,targets = data
        outputs = test(imgs)
        loss = loss_fn(outputs,targets)

        # 进行了一次：优化 -> 反向传播 -> 更新
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_train_step = total_train_step + 1

    # 每训练100次打印一次
    if total_train_step % 100 == 0:
        print("训练次数: {}, Loss: {}".format(total_train_step, loss))
        writer.add_scalar("train_loss",loss.item(),total_train_step)

    # 在进行一次训练后，可以将其在测试数据集上跑一次、测试其准确率
    # 在测试过程当中就不需要调优了
    total_test_loss = 0
    with torch.no_grad():
        for data in test_dataloader:
            imgs,targets = data
            outputs = test(imgs)
            loss = loss_fn(outputs,targets)
            total_test_loss = total_test_loss + loss.item()
    print("整体测试集上的Loss: {}".format(total_test_loss))
    writer.add_scalar("test_loss",total_test_loss,total_test_step)

    # 保存每一轮训练结束后的结果
    torch.save(test, "test_{}.pth".format(i))
    print("模型已保存")

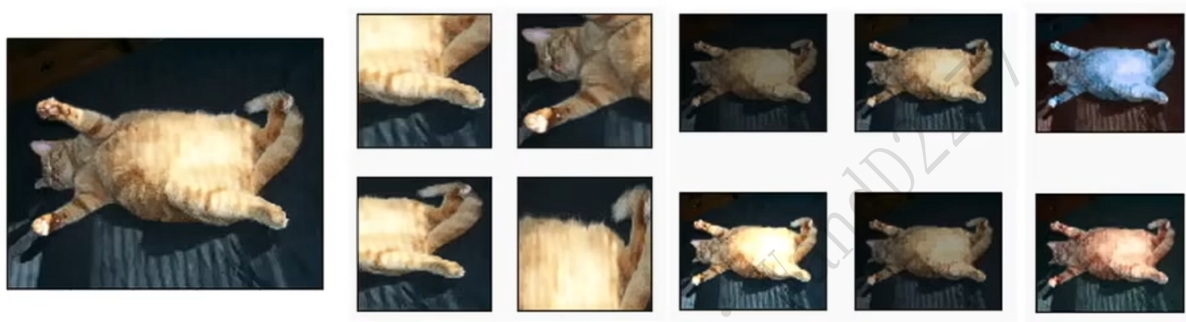
writer.close()

```

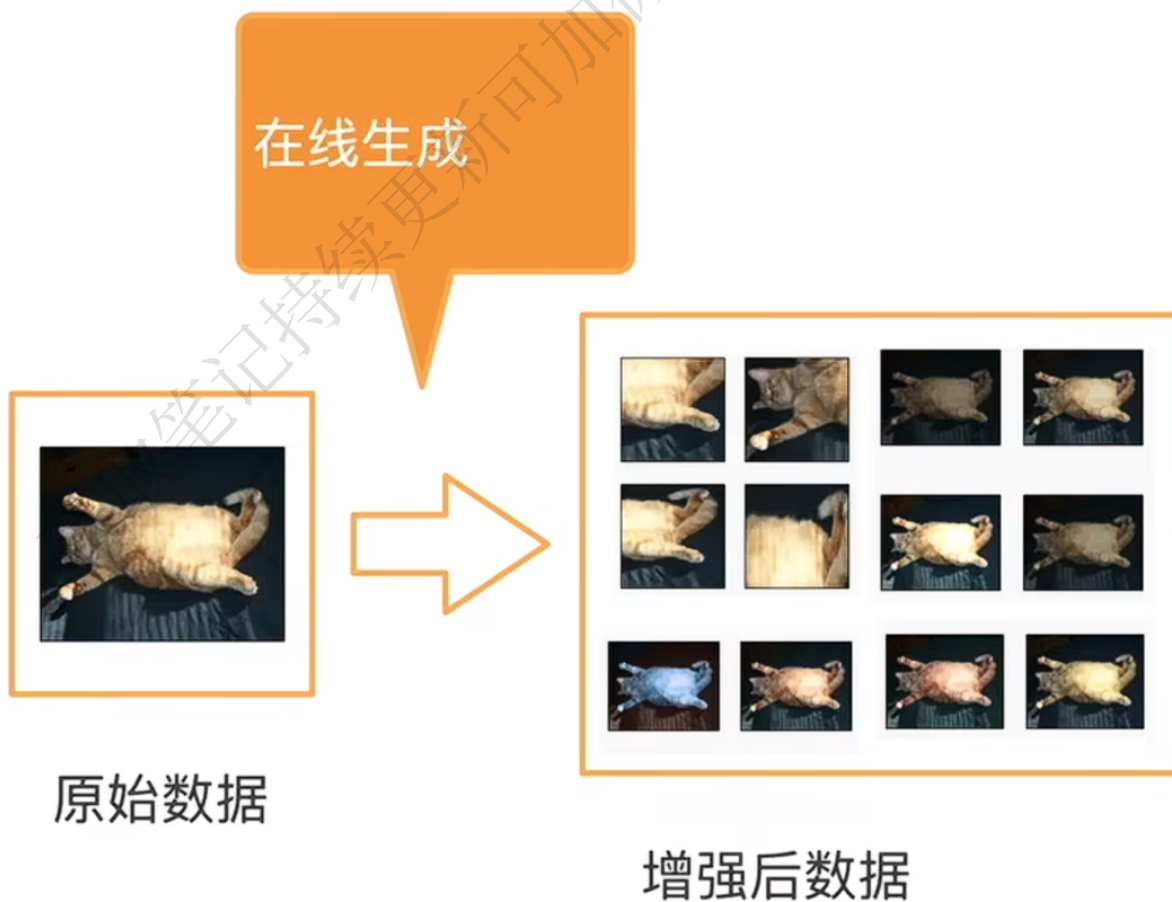
数据增强

概念：数据增强

- 概念：
 - 增加一个已有数据集，使得有更多的多样性；
 - 在语言里面：加入各种不同的背景噪音；
 - 在图片里面：改变图片的颜色和形状；
 - 数据增广通过变形数据来获取多样性，从而使得模型泛化性能更好；
 - 常见图片增广包括翻转、切割、变色；
- 图示：



- 实现：
 - 一般是在线、随机的生成；



- 分类：
 - 左右翻转；
 - 上下翻转；

- 注意：上下翻转并不一定一直都是可行的，比如建筑的图像、就不能上下翻转；
- 切割：
 - 从图片中切割一块，然后变形到固定形状；
 - 随机高宽比；
 - 随机大小；
 - 随机位置；
- 颜色：
 - 改变色调、饱和度、明亮度；
 - 一般是基于当前值进行变换；
- 代码：
 - 随机裁剪：`shape aug = torchvision.transforms.RandomResizedCrop((200, 200), scale=(0.1,1), ratio=(0.5, 2))`
 - 第一个参数是输出的大小，`scale` 是指的是在保存多大的区域，`ratio` 就是高宽比；
 - 随机改变图片的亮度：`apply(img, torchvision.transforms.ColorJitter(brightness=0.5, contrast=0, saturation=0, hue=0))`
 - 随机改变图像的亮度、对比度、饱和度、色调：
 - `color aug`
`=torchvision.transforms.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5)`

代码

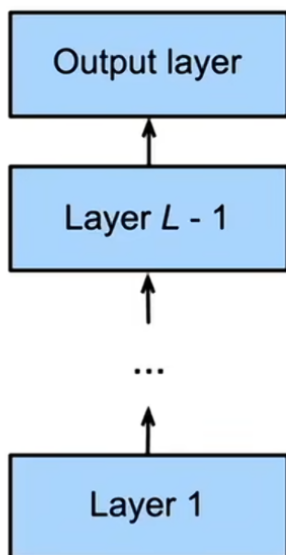
```
train_augs = torchvision.transforms.Compose([
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ToTensor()])

test_augs = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor()])
```

迁移学习

概念：迁移学习

- 引入：网络架构
 - 一个神经网络一般可以分成两块，特征抽取将原始像素变成容易线性分割的特征，线性分类器来做分类；
 - 可以理解为：除了最后一块在做 `Softmax` 回归外，其他所有块都在做特征提取。通过学习的手段把原始的图片信息，将其转换为可以线性分类的语义空间里；
- 概念：微调
 - 假设在原本的大的数据集上，假设这个大的数据集对原数据集可以做特征提取，那么可以假设、其可能仍然可以对我数据集做特征抽取；
 - 但是最后一层不能直接用：



不能直接使用，因为标号可能变了



可能仍然可以对我数据集做特征抽取



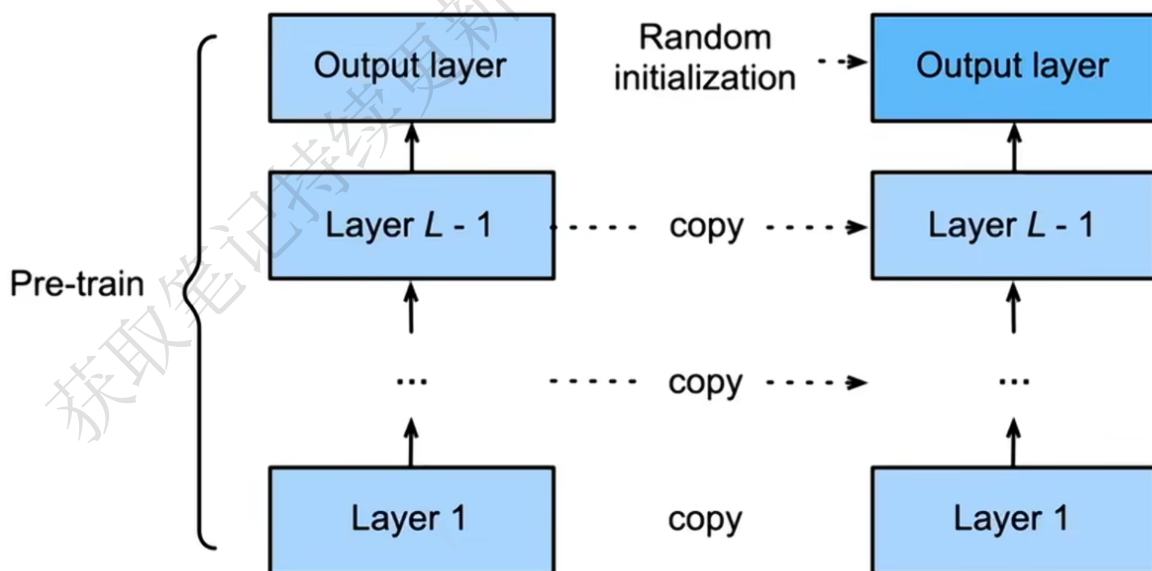
源数据集



目标数据集

实现:

- 假设在原数据上训练好了 Pre-Train，训练好后、使用一个一样的模型，从这个训练好的模型作为初始化的状态，那么这个新的数据集一开始状态就不错，然后更改其最后一层；



源数据集



目标数据集



概念：微调的训练

- 是一个目标数据集上的正常训练任务，但使用更强的正则化；
- 使用更小的学习率；

- 使用更少数据迭代；
- 源数据集远复杂于目标数据时，通常微调效果更好；
- 并且一般来讲，神经网络中：
 - 低层次的特征更加通用，高层次的特征则更跟数据集相关；
 - 因此可以**固定底部一些层的参数，不参与更新**；
 - 这是一种正则的方法；
- 总结：
 - 微调通过使用在大数据上得到的预训练好的模型，来初始化模型权重来完成提升精度；
 - 预训练模型质量很重要，微调通常速度更快、精度更高；

获取笔记持续更新可加微信：WandD2277