

目 录

引入.....	1
一、张量.....	2
1.1 数组与张量.....	2
1.2 从数组到张量.....	2
1.3 用 GPU 存储张量.....	3
二、DNN 的原理.....	4
2.1 划分数据集.....	4
2.2 训练网络.....	5
2.3 测试网络.....	7
2.4 使用网络.....	7
三、DNN 的实现.....	8
3.1 制作数据集.....	8
3.2 搭建神经网络.....	9
3.3 网络的内部参数.....	10
3.4 网络的外部参数.....	11
3.5 训练网络.....	11
3.6 测试网络.....	12
3.7 保存与导入网络.....	13
四、批量梯度下降.....	14
4.1 制作数据集.....	14
4.2 搭建神经网络.....	14
4.3 训练网络.....	15
4.4 测试网络.....	16
五、小批量梯度下降.....	17
5.1 制作数据集.....	17
5.2 搭建神经网络.....	18
5.3 训练网络.....	19
5.4 测试网络.....	19
六、手写数字识别.....	20
6.1 制作数据集.....	21
6.2 搭建神经网络.....	22
6.3 训练网络.....	23
6.4 测试网络.....	24

引入

0.1 版本需求

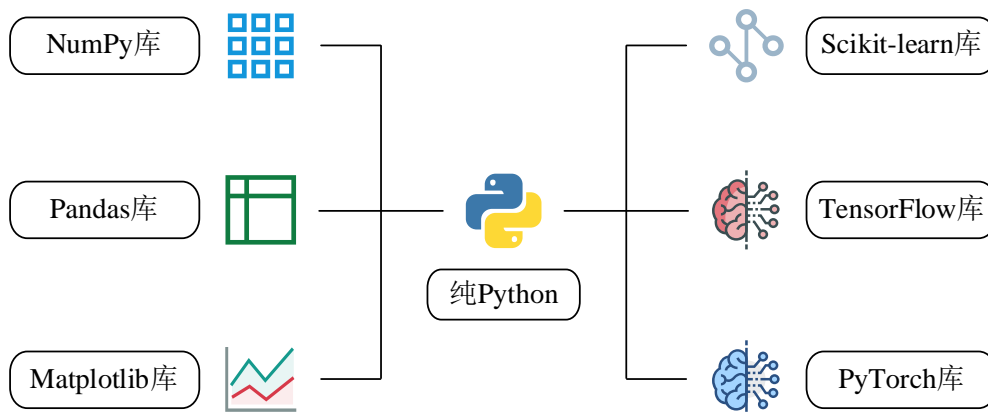
本视频中，使用的 Python 解释器与第三方库的版本如下。

- Python 为 3.9 版本，自 3.4 以来改动的语法可忽略不计；
 - NumPy 为 1.21 版本，Pandas 为 1.2.4 版本，Matplotlib 为 3.5.1 版本；
 - PyTorch 为 1.12.0 版本，此版本相对较新，更高的只有 1.13.0 和 2.0.0。
- 据悉，PyTorch 2.0.0 对性能有极大的提升，语法规则的变动较小。

0.2 视频特点

- **清晰度**：本视频分辨率为 1080P，请调高分辨率；
- **交流群**：关注【布尔艺数】公众号回复“杰哥”，自动弹出助理的二维码；
- **讲义链接**：助理将拉你进微信交流群，讲义 PDF 与代码在群公告中。
- **本课基础**：深度学习环境配置、Python 基础、NumPy 数组库。
- **课程定位**：帮助小白快速掌握 DNN 的基本原理，熟悉 PyTorch 实现方法。

0.3 深度学习的相关库



- ① NumPy 包为 Python 加上了关键的数组变量类型，弥补了 Python 的不足；
- ② Pandas 包在 NumPy 数组的基础上添加了与 Excel 类似的行列标签；
- ③ Matplotlib 库借鉴 Matlab，帮 Python 具备了绘图能力，使其如虎添翼；
- ④ Scikit-learn 库是机器学习库，内含分类、回归、聚类、降维等多种算法；
- ⑤ TensorFlow 库是 Google 公司开发的深度学习框架，于 2015 年问世；
- ⑥ PyTorch 库是 Facebook 公司开发的深度学习框架，于 2017 年问世。

0.4 深度学习的基本常识

- 人工智能是一个很大的概念，其中一个最重要的分支就是机器学习；
- 机器学习的算法多种多样，其中最核心的就是神经网络；
- 神经网络的隐藏层若足够深，就被称为深层神经网络，也即深度学习；
- 深度学习包含深度神经网络、卷积神经网络、循环神经网络等。

一、张量

1.1 数组与张量

本次课属于《Python 深度学习》系列视频，PyTorch 作为当前首屈一指的深度学习库，其将 NumPy 数组的语法尽数吸收，作为自己处理张量的基本语法，且运算速度从使用 CPU 的数组进步到使用 GPU 的张量。

NumPy 和 PyTorch 的基础语法几乎一致，具体表现为：

- np 对应 torch；
- 数组 array 对应张量 tensor；
- NumPy 的 n 维数组对应着 PyTorch 的 n 阶张量。

数组与张量之间可以相互转换：

- 数组 arr 转为张量 ts: `ts = torch.tensor(arr)`；
- 张量 ts 转为数组 arr: `arr = np.array(ts)`。

1.2 从数组到张量

为了找到 NumPy 和 PyTorch 哪些语法不同，UP 对 NumPy 文档进行了替换操作，将 np 改为 torch，将 array 改为 tensor，并重新运行所有代码，得出结论：PyTorch 只是少量修改了 NumPy 的函数或方法，现对其中不同的地方进行罗列。

表 1-1 PyTorch 修正的 NumPy 函数或方法

课件位置	NumPy 的函数	PyTorch 的函数	用法区别
1.1 数据类型	<code>.astype()</code>	<code>.type()</code>	无
2.4 随机数组	<code>np.random.random()</code>	<code>torch.rand()</code>	无
2.4 随机数组	<code>np.random.randint()</code>	<code>torch.randint()</code>	不接纳一维张量
2.4 随机数组	<code>np.random.normal()</code>	<code>torch.normal()</code>	不接纳一维张量
2.4 随机数组	<code>np.random.randn()</code>	<code>torch.randn()</code>	无
3.4 数组切片	<code>.copy()</code>	<code>.clone()</code>	无
4.4 数组拼接	<code>np.concatenate()</code>	<code>torch.cat()</code>	无
4.5 数组分裂	<code>np.split()</code>	<code>torch.split()</code>	参数含义优化
6.1 矩阵乘积	<code>np.dot()</code>	<code>torch.matmul()</code>	无
6.1 矩阵乘积	<code>np.dot(v,v)</code>	<code>torch.dot()</code>	无
6.1 矩阵乘积	<code>np.dot(m,v)</code>	<code>torch.mv()</code>	无
6.1 矩阵乘积	<code>np.dot(m,m)</code>	<code>torch.mm()</code>	无
6.2 数学函数	<code>np.exp()</code>	<code>torch.exp()</code>	必须传入张量
6.2 数学函数	<code>np.log()</code>	<code>torch.log()</code>	必须传入张量
6.3 聚合函数	<code>np.mean()</code>	<code>torch.mean()</code>	必须传入浮点型张量
6.3 聚合函数	<code>np.std()</code>	<code>torch.std()</code>	必须传入浮点型张量

1.3 用 GPU 存储张量

默认的张量使用 CPU 存储，可将其搬至 GPU 上，如示例所示。

```
In [1]: import torch
```

```
In [2]: # 默认的张量存储在 CPU 上
```

```
ts1 = torch.randn(3,4)
ts1
```

```
Out [2]: tensor([[ 2.2716,  1.2107, -0.0582,  0.5885 ],
                 [-0.5868, -0.6480, -0.2591,  0.1605],
                 [-1.3968,  0.7999,  0.5180,  1.2214 ]])
```

```
In [3]: # 移动到 GPU 上
```

```
ts2 = ts1.to('cuda:0')    # 第一块 GPU 是 cuda:0
ts2
```

```
Out [3]: tensor([[ 2.2716,  1.2107, -0.0582,  0.5885 ],
                 [-0.5868, -0.6480, -0.2591,  0.1605],
                 [-1.3968,  0.7999,  0.5180,  1.2214 ]], device='cuda:0')
```

以上操作可以把数据集搬到 GPU 上，但是神经网络模型也要搬到 GPU 上才可正常运行，使用下面的代码即可。

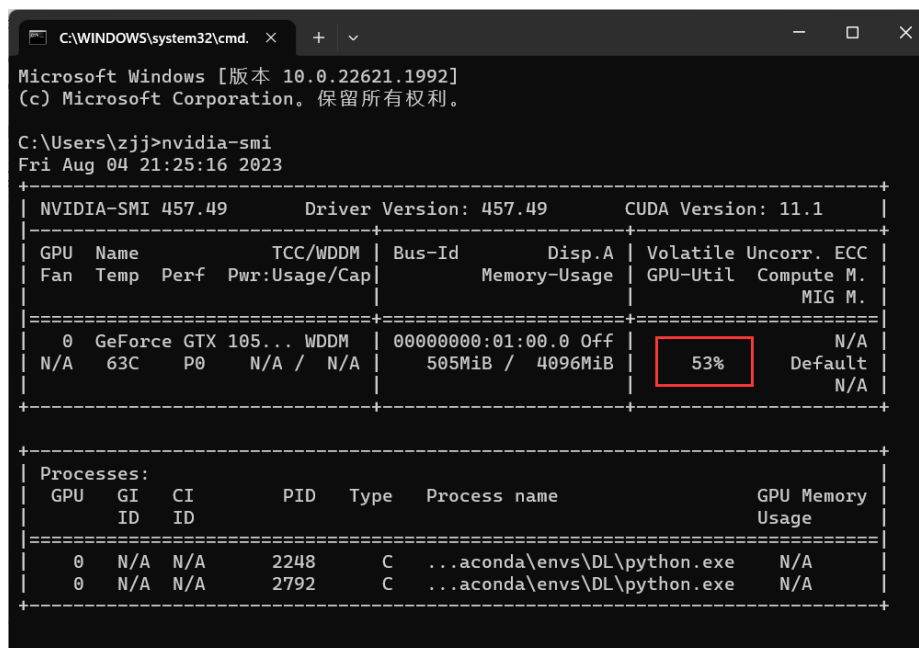
```
In [4]: # 搭建神经网络的类，此处略，详见第三章
```

```
class DNN(torch.nn.Module):
    略
```

```
In [5]: # 根据神经网络的类创建一个网络
```

```
model = DNN().to('cuda:0')    # 把该网络搬到 GPU 上
```

想要查看显卡是否在运作时，在 cmd 中输入：nvidia-smi，如图 1-1 所示。



```
C:\WINDOWS\system32\cmd. x + v
Microsoft Windows [版本 10.0.22621.1992]
(c) Microsoft Corporation. 保留所有权利。

C:\Users\zjj>nvidia-smi
Fri Aug 04 21:25:16 2023

+-----+
| NVIDIA-SMI 457.49      | Driver Version: 457.49      | CUDA Version: 11.1      |
+-----+-----+
| GPU   Name           | TCC/WDDM | Bus-Id  | Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf    Pwr:Usage/Cap|      |          |      |          | GPU-Util  Compute M. |
|=====+=====+
| 0   GeForce GTX 105... | WDDM     | 00000000:01:00.0 Off |          |      N/A             |
| N/A   63C    P0      N/A /  N/A |          | 505MiB / 4096MiB |          | 53%      Default   |
|=====+=====+
+-----+-----+
| Processes: |
| GPU   GI   CI        PID   Type   Process name                  | GPU Memory |
|=====+=====+
| 0     N/A  N/A       2248    C   ...aconda\envs\DL\python.exe |      N/A   |
| 0     N/A  N/A       2792    C   ...aconda\envs\DL\python.exe |      N/A   |
+-----+-----+
```

图 1-1 查看显卡运行情况

二、DNN 的原理

神经网络通过学习大量样本的输入与输出特征之间的关系，以拟合出输入与输出之间的方程，学习完成后，只给它输入特征，它便会可以给出输出特征。神经网络可以分为这么几步：划分数据集、训练网络、测试网络、使用网络。

2.1 划分数据集

数据集里每个样本必须包含输入与输出，将数据集按一定的比例划分为训练集与测试集，分别用于训练网络与测试网络，如表 2-1 所示。

表 2-1 数据集的划分

	样本	输入特征			输出特征		
		In1	In2	In3	Out1	Out2	Out3
训练集	1	*	*	*	*	*	*
	2	*	*	*	*	*	*
	3	*	*	*	*	*	*
	4	*	*	*	*	*	*
	5	*	*	*	*	*	*
	...	*	*	*	*	*	*
	800	*	*	*	*	*	*
测试集	801	*	*	*	*	*	*
	802	*	*	*	*	*	*
	...	*	*	*	*	*	*
	1000	*	*	*	*	*	*

考虑到数据集的输入特征与输出特征都是 3 列，因此神经网络的输入层与输出层也必须都是 3 个神经元，隐藏层可以自行设计，如图 2-1 所示。

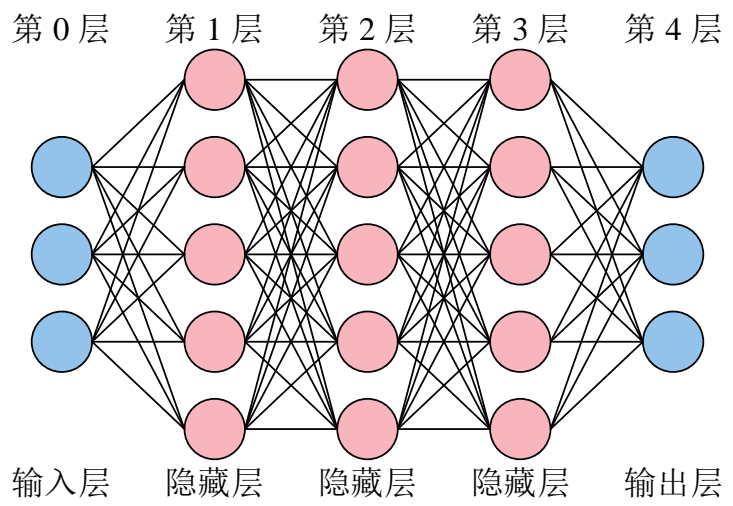


图 2-1 神经网络的结构

考虑到 Python 列表、NumPy 数组以及 PyTorch 张量都是从索引[0]开始，再加之输入层没有内部参数(权重 ω 与偏置 b)，所以习惯将输入层称之为第 0 层。



2.2 训练网络

神经网络的训练过程，就是经过很多次前向传播与反向传播的轮回，最终不断调整其内部参数（权重 ω 与偏置 b ），以拟合任意复杂函数的过程。内部参数一开始是随机的（如 Xavier 初始值、He 初始值），最终会不断优化到最佳。

还有一些训练网络前就要设好的外部参数：网络的层数、每个隐藏层的节点数、每个节点的激活函数类型、学习率、轮回次数、每次轮回的样本数等等。

业界习惯把内部参数称为参数，外部参数称为超参数。

(1) 前向传播

将单个样本的 3 个输入特征送入神经网络的输入层后，神经网络会逐层计算到输出层，最终得到神经网络预测的 3 个输出特征。计算过程中所使用的参数就是内部参数，所有的隐藏层与输出层的神经元都有内部参数，以第 1 层的第 1 个神经元，如图 2-2 所示。

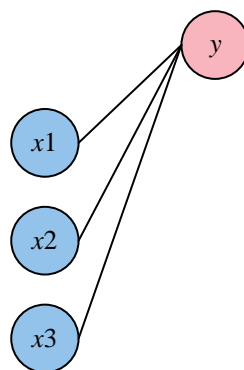


图 2-2 每个神经元节点的计算

该神经元节点的计算过程为 $y = \omega_1 x_1 + \omega_2 x_2 + \omega_3 x_3 + b$ 。你可以理解为，每一根线就是一个权重 ω ，每一个神经元节点也都有它自己的偏置 b 。

当然，每个神经元节点在计算完后，由于这个方程是线性的，因此必须在外面套一个非线性的函数： $y = \sigma(\omega_1 x_1 + \omega_2 x_2 + \omega_3 x_3 + b)$ ， σ 被称为激活函数。如果你不套非线性函数，那么即使 10 层的网络，也可以用 1 层就拟合出同样的方程。

(2) 反向传播

经过前向传播，网络会根据当前的内部参数计算出输出特征的预测值。但是这个预测值与真实值直接肯定有差距，因此需要一个损失函数来计算这个差距。例如，求预测值与真实值之间差的绝对值，就是一个典型的损失函数。

损失函数计算好后，逐层退回求梯度，这个过程很复杂，原理不必掌握，大致意思就是，看每一个内部参数是变大还是变小，才会使得损失函数变小。这样就达到了优化内部参数的目的。

在这个过程中，有一个外部参数叫学习率。学习率越大，内部参数的优化越快，但过大的学习率可能会使损失函数越过最低点，并在谷底反复横跳。因此，在网络的训练开始之前，选择一个合适的学习率很重要。



(3) batch_size

前向传播与反向传播一次时，有三种情况：

- 批量梯度下降 (Batch Gradient Descent, BGD)，把所有样本一次性输入进网络，这种方式计算量开销很大，速度也很慢。
- 随机梯度下降 (Stochastic Gradient Descent, SGD)，每次只把一个样本输入进网络，每计算一个样本就更新参数。这种方式虽然速度比较快，但是收敛性能差，可能会在最优点附近震荡，两次参数的更新也有可能抵消。
- 小批量梯度下降 (Mini-Batch Gradient Decent, MBGD) 是为了中和上面二者而生，这种办法把样本划分为若干个批，按批来更新参数。

所以，batch_size 即一批中的样本数，也是一次喂进网络的样本数。此外，由于 Batch Normalization 层(用于将每次产生的小批量样本进行标准化)的存在，batch_size 一般设置为 2 的幂次方，并且不能为 1。

注：PyTorch 实现时只支持批量与小批量，不支持单个样本的输入方式。PyTorch 里的 torch.optim.SGD 只表示梯度下降，批量与小批量见第四、五章。

(4) epochs

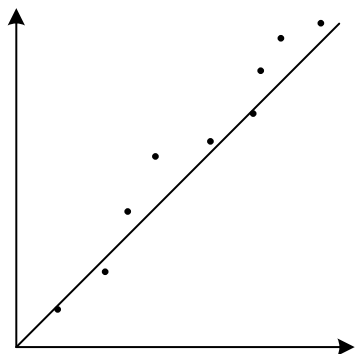
1 个 epoch 就是指全部样本进行 1 次前向传播与反向传播。

假设有 10240 个训练样本，batch_size 是 1024，epochs 是 5。那么：

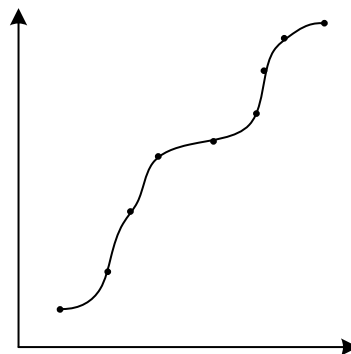
- 全部样本将进行 5 次前向传播与反向传播；
- 1 个 epoch，将发生 10 次 ($10240 \div 1024$) 前向传播与反向传播；
- 一共发生 50 次 (10×5) 前向传播和反向传播。

2.3 测试网络

为了防止训练的网络过拟合，因此需要拿出少量的样本进行测试。过拟合的意思是：网络优化好的内部参数只能对训练样本有效，换成其它就寄。以线性回归为例，过拟合如图 2-3（b）所示。



（a）正确的拟合



（b）过拟合

图 2-3 过拟合的危害

当网络训练好后，拿出测试集的输入，进行 1 次前向传播后，将预测的输出与测试集的真实输出进行对比，查看准确率。

2.4 使用网络

真正使用网络进行预测时，样本只知输入，不知输出。直接将样本的输入进行 1 次前向传播，即可得到预测的输出。



三、DNN 的实现

torch.nn 提供了搭建网络所需的所有组件，nn 即 Neural Network 神经网络。因此，可以单独给 torch.nn 一个别名，即 **import torch.nn as nn**。

```
In [1]: import torch
import torch.nn as nn
import matplotlib.pyplot as plt
%matplotlib inline
```

```
In [2]: # 展示高清图
from matplotlib_inline import backend_inline
backend_inline.set_matplotlib_formats('svg')
```

3.1 制作数据集

在训练之前，要准备好训练集的样本。

这里生成 10000 个样本，设定 3 个输入特征与 3 个输出特征，其中

- 每个输入特征相互独立，均服从均匀分布；
- 当 $(X1+X2+X3) < 1$ 时，Y1 为 1，否则 Y1 为 0；
- 当 $1 < (X1+X2+X3) < 2$ 时，Y2 为 1，否则 Y2 为 0；
- 当 $(X1+X2+X3) > 2$ 时，Y3 为 1，否则 Y3 为 0；
- .float() 将布尔型张量转化为浮点型张量。

```
In [3]: # 生成数据集
X1 = torch.rand(10000,1) # 输入特征 1
X2 = torch.rand(10000,1) # 输入特征 2
X3 = torch.rand(10000,1) # 输入特征 3
Y1 = ((X1+X2+X3) < 1).float() # 输出特征 1
Y2 = ((1 < (X1+X2+X3)) & ((X1+X2+X3) < 2)).float() # 输出特征 2
Y3 = ((X1+X2+X3) > 2).float() # 输出特征 3
Data = torch.cat([X1,X2,X3,Y1,Y2,Y3],axis=1) # 整合数据集
Data = Data.to('cuda:0') # 把数据集搬到 GPU 上
Data.shape
```

```
Out [3]: torch.Size([10000, 6])
```

事实上，数据的 3 个输出特征组合起来是一个 One-Hot 编码（独热编码）。

```
In [4]: # 划分训练集与测试集
train_size = int(len(Data) * 0.7) # 训练集的样本数量
test_size = len(Data) - train_size # 测试集的样本数量
Data = Data[torch.randperm( Data.size(0) ),:] # 打乱样本的顺序
train_Data = Data[: train_size ,:] # 训练集样本
test_Data = Data[ train_size : ,:] # 测试集样本
train_Data.shape, test_Data.shape
```

```
Out [4]: (torch.Size([7000, 6]), torch.Size([3000, 6]))
```

In [4] 的代码属于通用型代码，便于我们手动分割训练集与测试集。



3.2 搭建神经网络

搭建神经网络时，以 `nn.Module` 作为父类，我们自己的神经网络可直接继承父类的方法与属性，`nn.Module` 中包含网络各个层的定义。

在定义的神经网络子类中，通常包含 `__init__` 特殊方法与 `forward` 方法。`__init__` 特殊方法用于构造自己的神经网络结构，`forward` 方法用于将输入数据进行前向传播。由于张量可以自动计算梯度，所以不需要出现反向传播方法。

```
In [5]: class DNN(nn.Module):

    def __init__(self):
        ''' 搭建神经网络各层 '''
        super(DNN,self).__init__()
        self.net = nn.Sequential(          # 按顺序搭建各层
            nn.Linear(3, 5), nn.ReLU(),    # 第 1 层：全连接层
            nn.Linear(5, 5), nn.ReLU(),    # 第 2 层：全连接层
            nn.Linear(5, 5), nn.ReLU(),    # 第 3 层：全连接层
            nn.Linear(5, 3)                # 第 4 层：全连接层
        )

    def forward(self, x):
        ''' 前向传播 '''
        y = self.net(x)    # x 即输入数据
        return y           # y 即输出数据
```

```
In [6]: model = DNN().to('cuda:0')    # 创建子类的实例，并搬到 GPU 上
        model                          # 查看该实例的各层
```

```
Out [6]: DNN(
  (net): Sequential(
    (0): Linear(in_features=3, out_features=5, bias=True)
    (1): ReLU()
    (2): Linear(in_features=5, out_features=5, bias=True)
    (3): ReLU()
    (4): Linear(in_features=5, out_features=5, bias=True)
    (5): ReLU()
    (6): Linear(in_features=5, out_features=3, bias=True)
  )
)
```

在上面的 `nn.Sequential()` 函数中，每一个隐藏层后都使用了 `ReLU` 激活函数，各层的神经元节点个数分别是：3、5、5、5、3。

注意，输入层有 3 个神经元、输出层有 3 个神经元，这不是巧合，是有意而为之。输入层的神经元数量必须与每个样本的输入特征数量一致，输出层的神经元数量必须与每个样本的输出特征数量一致。

3.3 网络的内部参数

神经网络的内部参数是权重与偏置，内部参数在神经网络训练之前会被赋予随机数，随着训练的进行，内部参数会逐渐迭代至最佳值，现对参数进行查看。

```
In [7]: # 查看内部参数（非必要）
        for name, param in model.named_parameters():
            print(f"参数:{name}\n 形状:{param.shape}\n 数值:{param}\n")
```

Out [7]: 参数:net.0.weight
形状:torch.Size([5, 3])
数值:Parameter containing:
tensor([[0.0526, -0.3374, -0.0227],
 [0.1673, 0.4338, 0.3040],
 [0.5739, -0.4609, 0.3183],
 [-0.1983, -0.3941, 0.2630],
 [-0.5472, 0.4121, -0.2182]],
 device='cuda:0', requires_grad=True)

参数:net.0.bias
形状:torch.Size([5])
数值:Parameter containing:
tensor([0.5564, -0.0882, -0.4600, -0.2319, 0.2650],
 device='cuda:0', requires_grad=True)

（页面有限，此处仅展示 net.0 的权重与偏置）

代码一共给了我们 8 个参数，其中参数与形状的结果如表 3-1 所示，考虑到其数值初始状态时是随机的（如 Xavier 初始值、He 初始值），此处不讨论。

表 3-1 网络的内部参数及其形状

参数	形状	参数	形状
net.0.weight	torch.Size([5, 3])	net.0.bias	torch.Size([5])
net.2.weight	torch.Size([5, 5])	net.2.bias	torch.Size([5])
net.4.weight	torch.Size([5, 5])	net.4.bias	torch.Size([5])
net.6.weight	torch.Size([3, 5])	net.6.bias	torch.Size([3])

可见，具有权重与偏置的地方只有 net.0、net.2、net.4、net.6，结合 Out [3] 的结果，可知这几个地方其实就是所有的隐藏层与输出层，这符合理论。

- 首先，net.0.weight 的权重形状为[5, 3]，5 表示它自己的节点数是 5，3 表示与之连接的前一层的节点数为 3。
- 其次，由于 In [3]里进行了 model=DNN().to('cuda:0')操作，因此所有的内部参数都自带 device='cuda:0'。
- 最后，注意到 requires_grad=True，说明所有需要进行反向传播的内部参数（即权重与偏置）都打开了张量自带的梯度计算功能。



3.4 网络的外部参数

外部参数即超参数，这是调参师们关注的重点。搭建网络时的超参数有：网络的层数、各隐藏层节点数、各节点激活函数、内部参数的初始值等。训练网络的超参数有：如损失函数、学习率、优化算法、batch_size、epochs 等。

(1) 激活函数

PyTorch 1.12.0 版本进入 <https://pytorch.org/docs/1.12/nn.html> 搜索 Non-linear Activations，即可查看 torch 内置的所有非线性激活函数（以及各种类型的层）。

(2) 损失函数

进入 <https://pytorch.org/docs/1.12/nn.html> 搜索 Loss Functions，即可查看 torch 内置的所有损失函数。

```
In [8]: # 损失函数的选择
        loss_fn = nn.MSELoss()
```

(3) 学习率与优化算法

进入 <https://pytorch.org/docs/1.12/optim.html>，可查看 torch 的所有优化算法。

```
In [9]: # 优化算法的选择
        learning_rate = 0.01    # 设置学习率
        optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

注：PyTorch 实现时只支持 BGD 或 MBGD，不支持单个样本的输入方式。这里的 torch.optim.SGD 只表示梯度下降，具体的批量与小批量见第四、五章。

3.5 训练网络

```
In [10]: # 训练网络
        epochs = 1000
        losses = []           # 记录损失函数变化的列表

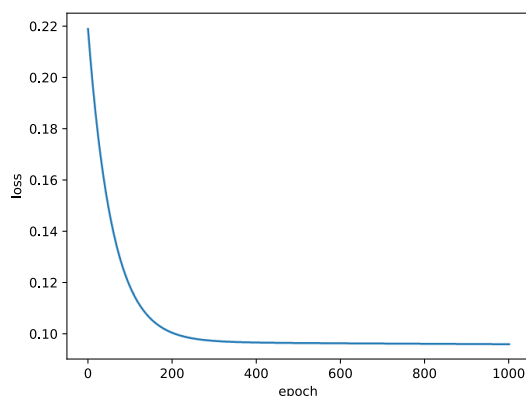
        # 给训练集划分输入与输出
        X = train_Data[:, :3]    # 前 3 列为输入特征
        Y = train_Data[:, -3:]   # 后 3 列为输出特征

        for epoch in range(epochs):
            Pred = model(X)        # 一次前向传播（批量）
            loss = loss_fn(Pred, Y) # 计算损失函数
            losses.append(loss.item()) # 记录损失函数的变化
            optimizer.zero_grad()  # 清理上一轮滞留的梯度
            loss.backward()         # 一次反向传播
            optimizer.step()        # 优化内部参数

        Fig = plt.figure()
        plt.plot(range(epochs), losses)
        plt.ylabel('loss'), plt.xlabel('epoch')
        plt.show()
```



Out [10]:



`losses.append(loss.item())`中, `.append()`是指在列表 `losses` 后再附加 1 个元素, 而`.item()`方法可将 PyTorch 张量退化为普通元素。

3.6 测试网络

测试时, 只需让测试集进行 1 次前向传播即可, 这个过程不需要计算梯度, 因此可以在该局部关闭梯度, 该操作使用 **with torch.no_grad():**命令。

考虑到输出特征是独热编码, 而预测的数据一般都是接近 0 或 1 的小数, 为了能让预测数据与真实数据之间进行比较, 因此要对预测数据进行规整。例如, 使用 `Pred[:,torch.argmax(Pred,axis=1)] = 1` 命令将每行最大的数置 1, 接着再使用 `Pred[Pred!=1] = 0` 将不是 1 的数字置 0, 这就使预测数据与真实数据的格式相同。

In [11]: # 测试网络

```
# 给测试集划分输入与输出
X = test_Data[:, :3]      # 前 3 列为输入特征
Y = test_Data[:, -3:]     # 后 3 列为输出特征

with torch.no_grad():     # 该局部关闭梯度计算功能
    Pred = model(X)        # 一次前向传播 (批量)
    Pred[:,torch.argmax(Pred,axis=1)] = 1
    Pred[Pred!=1] = 0
    correct = torch.sum( (Pred == Y).all(1) )    # 预测正确的样本
    total = Y.size(0)                            # 全部的样本数量
    print(f'测试集精准度: {100*correct/total} %')
```

Out [11]: 测试集精准度: 67.16666412353516 %

在计算 `correct` 时需要动点脑筋。

首先, `(Pred == Y)`计算预测的输出与真实的输出的各个元素是否相等, 返回一个 3000 行、3 列的布尔型张量。

其次, `(Pred == Y).all(1)`检验该布尔型张量每一行的 3 个数据是否都是 **True**, 对于全是 **True** 的样本行, 结果就是 **True**, 否则是 **False**。`all(1)`中的 1 表示按“行”扫描, 最终返回一个形状为 3000 的一阶张量。

最后, `torch.sum((Pred == Y).all(1))`的意思就是看这 3000 个向量相加, **True** 会被当作 1, **False** 会被当作 0, 这样相加刚好就是预测正确的样本数。



3.7 保存与导入网络

现在我们要考虑一件大事，那就是有时候训练一个大网络需要几天，那么必须要把整个网络连同里面的优化好的内部参数给保存下来。

现以本章前面的代码为例，当网络训练好后，将网络以文件的形式保存下来，并通过文件导入给另一个新网络，让新网络去跑测试集，看看测试集的准确率是否也是 67%。

(1) 保存网络

通过 “`torch.save(模型名, '文件名.pth')`” 命令，可将该模型完整的保存至 Jupyter 的工作路径下。

```
In [12]: # 保存网络
         torch.save(model, 'model.pth')
```

(2) 导入网络

通过 “`新网络 = torch.load('文件名.pth')`” 命令，可将该模型完整的导入给新网络。

```
In [13]: # 把模型赋给新网络
         new_model = torch.load('model.pth')
```

现在，`new_model` 就与 `model` 完全一致，可以直接去跑测试集。

(3) 用新模型进行测试

```
In [14]: # 测试网络

         # 给测试集划分输入与输出
         X = test_Data[:, :3]      # 前 3 列为输入特征
         Y = test_Data[:, -3:]     # 后 3 列为输出特征

         with torch.no_grad():      # 该局部关闭梯度计算功能
             Pred = new_model(X)    # 用新模型进行一次前向传播
             Pred[:, torch.argmax(Pred, axis=1)] = 1
             Pred[Pred!=1] = 0
             correct = torch.sum((Pred == Y).all(1)) # 预测正确的样本
             total = Y.size(0)         # 全部的样本数量
             print(f'测试集精准度: {100*correct/total} %')
```

Out [14]: 测试集精准度: 67.16666412353516 %

保存与加载成功，本视频仅演示这么 1 次，后面的章节不再保存网络。



四、批量梯度下降

本小节将完整、快速地再展示一遍批量梯度下降（BGD）的全过程。

```
In [1]: import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
%matplotlib inline

In [2]: # 展示高清图
from matplotlib_inline import backend_inline
backend_inline.set_matplotlib_formats('svg')
```

4.1 制作数据集

这一次的数据集将从 Excel 中导入，需要 Pandas 库中的 `pd.read_csv()` 函数，这在《Pandas 标签库》讲义的第六章中有详细的介绍。

```
In [3]: # 准备数据集
df = pd.read_csv('Data.csv', index_col=0) # 导入数据
arr = df.values # Pandas 对象退化为 NumPy 数组
arr = arr.astype(np.float32) # 转为 float32 类型数组
ts = torch.tensor(arr) # 数组转为张量
ts = ts.to('cuda') # 把训练集搬到 cuda 上
ts.shape
```

Out [3]: torch.Size([759, 9])

在 In [3] 的第 4 行，将 `arr` 数组转为了 `np.float32` 类型这一步必不可少，没有的话计算过程会出现一些数据类型不兼容的情况。

```
In [4]: # 划分训练集与测试集
train_size = int(len(ts) * 0.7) # 训练集的样本数量
test_size = len(ts) - train_size # 测试集的样本数量
ts = ts[torch.randperm(ts.size(0)), :] # 打乱样本的顺序
train_Data = ts[:train_size, :] # 训练集样本
test_Data = ts[train_size:, :] # 测试集样本
train_Data.shape, test_Data.shape
```

Out [4]: (torch.Size([759, 8]), torch.Size([759, 1]))

In [4] 的第 2 行，`0.7` 表示训练集占整个数据集样本量的 70%，可以手动调整。

4.2 搭建神经网络

注意到前面的数据集，输入有 8 个特征，输出有 1 个特征，那么神经网络的输入层必须有 8 个神经元，输出层必须有 1 个神经元。

隐藏层的层数、各隐藏层的节点数属于外部参数（超参数），可以自行设置。



```
In [5]: class DNN(nn.Module):

    def __init__(self):
        ''' 搭建神经网络各层 '''
        super(DNN,self).__init__()
        self.net = nn.Sequential(                # 按顺序搭建各层
            nn.Linear(8,32), nn.Sigmoid(),        # 第 1 层：全连接层
            nn.Linear(32,8), nn.Sigmoid(),        # 第 2 层：全连接层
            nn.Linear(8,4), nn.Sigmoid(),         # 第 3 层：全连接层
            nn.Linear(4,1), nn.Sigmoid()         # 第 4 层：全连接层
        )

    def forward(self, x):
        ''' 前向传播 '''
        y = self.net(x)    # x 即输入数据
        return y           # y 即输出数据
```

```
In [6]: model = DNN().to('cuda:0')    # 创建子类的实例，并搬到 GPU 上
        model                          # 查看该实例的各层
```

```
Out [6]: DNN(
  (net): Sequential(
    (0): Linear(in_features=8, out_features=32, bias=True)
    (1): Sigmoid()
    (2): Linear(in_features=32, out_features=8, bias=True)
    (3): Sigmoid()
    (4): Linear(in_features=8, out_features=4, bias=True)
    (5): Sigmoid()
    (6): Linear(in_features=4, out_features=1, bias=True)
    (7): Sigmoid()
  )
)
```

4.3 训练网络

```
In [7]: # 损失函数的选择
        loss_fn = nn.BCELoss(reduction='mean')
```

```
In [8]: # 优化算法的选择
        learning_rate = 0.005    # 设置学习率
        optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

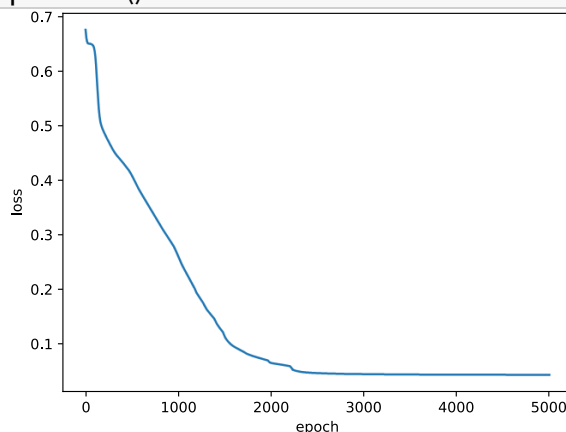
```
In [9]: # 训练网络
        epochs = 5000
        losses = []              # 记录损失函数变化的列表

        # 给训练集划分输入与输出
        X = train_Data[:, :-1]   # 前 8 列为输入特征
        Y = train_Data[:, -1].reshape((-1,1)) # 后 1 列为输出特征
        # 此处的.reshape((-1,1))将一阶张量升级为二阶张量
```

```
for epoch in range(epochs):
    Pred = model(X)                # 一次前向传播（批量）
    loss = loss_fn(Pred, Y)        # 计算损失函数
    losses.append(loss.item())      # 记录损失函数的变化
    optimizer.zero_grad()          # 清理上一轮滞留的梯度
    loss.backward()                 # 一次反向传播
    optimizer.step()                # 优化内部参数

Fig = plt.figure()
plt.plot(range(epochs), losses)
plt.ylabel('loss')
plt.xlabel('epoch')
plt.show()
```

Out [9]:



4.4 测试网络

注意，真实的输出特征都是 0 或 1，因此这里需要对网络预测的输出 Pred 进行处理，Pred 大于 0.5 的部分全部置 1，小于 0.5 的部分全部置 0。

In [10]: # 测试网络

```
# 给测试集划分输入与输出
X = test_Data[:, :-1]          # 前 8 列为输入特征
Y = test_Data[:, -1].reshape((-1,1)) # 后 1 列为输出特征

with torch.no_grad():          # 该局部关闭梯度计算功能
    Pred = model(X)            # 一次前向传播（批量）
    Pred[Pred >= 0.5] = 1
    Pred[Pred < 0.5] = 0
    correct = torch.sum((Pred == Y).all(1)) # 预测正确的样本
    total = Y.size(0)             # 全部的样本数量
    print(f'测试集精准度: {100*correct/total} %')
```

Out [10]: 测试集精准度: 70.6140365600586 %



五、小批量梯度下降

本章将继续使用第四章中的 Excel 与神经网络结构，但使用小批量训练。在使用小批量梯度下降时，必须使用 3 个 PyTorch 内置的实用工具（utils）：

- DataSet 用于封装数据集；
- DataLoader 用于加载数据不同的批次；
- random_split 用于划分训练集与测试集。

```
In [1]: import numpy as np
import pandas as pd
import torch
import torch.nn as nn
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.data import random_split
import matplotlib.pyplot as plt
%matplotlib inline
```

```
In [2]: # 展示高清图
from matplotlib_inline import backend_inline
backend_inline.set_matplotlib_formats('svg')
```

5.1 制作数据集

在封装我们的数据集时，必须继承实用工具（utils）中的 DataSet 的类，这个过程需要重写 `__init__`、`__getitem__`、`__len__` 三个方法，分别是为了加载数据集、获取数据索引、获取数据总量。

```
In [3]: # 制作数据集
class MyData(Dataset):    # 继承 Dataset 类
    def __init__(self, filepath):
        df = pd.read_csv(filepath, index_col=0)    # 导入数据
        arr = df.values    # 对象退化为数组
        arr = arr.astype(np.float32)    # 转为 float32 类型数组
        ts = torch.tensor(arr)    # 数组转为张量
        ts = ts.to('cuda')    # 把训练集搬到 cuda 上
        self.X = ts[:, :-1]    # 前 8 列为输入特征
        self.Y = ts[:, -1].reshape((-1,1))    # 后 1 列为输出特征
        self.len = ts.shape[0]    # 样本的总数

    def __getitem__(self, index):
        return self.X[index], self.Y[index]

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

小批次训练时，输入特征与输出特征的划分必须写在 In [3] 的子类里面。



```
In [4]: # 划分训练集与测试集
Data = MyData('Data.csv')
train_size = int(len(Data) * 0.7)    # 训练集的样本数量
test_size = len(Data) - train_size   # 测试集的样本数量
train_Data, test_Data = random_split(Data, [train_size, test_size])
```

In [4]中，我们利用实用工具（utils）里的 random_split 轻松实现了训练集与测试集数据的划分。

```
In [5]: # 批次加载器
train_loader = DataLoader(dataset=train_Data, shuffle=True, batch_size=128)
test_loader = DataLoader(dataset=test_Data, shuffle=False, batch_size=64)
```

In [5]中，实用工具（utils）里的 DataLoader 可以在接下来的训练中进行小批次的载入数据，shuffle 用于在每一个 epoch 内先洗牌再分批。

5.2 搭建神经网络

```
In [6]: class DNN(nn.Module):

    def __init__(self):
        ''' 搭建神经网络各层 '''
        super(DNN,self).__init__()
        self.net = nn.Sequential(
            nn.Linear(8,32), nn.Sigmoid(), # 第1层：全连接层
            nn.Linear(32,8), nn.Sigmoid(), # 第2层：全连接层
            nn.Linear(8,4), nn.Sigmoid(),  # 第3层：全连接层
            nn.Linear(4,1), nn.Sigmoid()   # 第4层：全连接层
        )

    def forward(self, x):
        ''' 前向传播 '''
        y = self.net(x)    # x 即输入数据
        return y           # y 即输出数据
```

```
In [7]: model = DNN().to('cuda:0')    # 创建子类的实例，并搬到 GPU 上
model                                     # 查看该实例的各层
```

```
Out [7]: DNN(
  (net): Sequential(
    (0): Linear(in_features=8, out_features=32, bias=True)
    (1): Sigmoid()
    (2): Linear(in_features=32, out_features=8, bias=True)
    (3): Sigmoid()
    (4): Linear(in_features=8, out_features=4, bias=True)
    (5): Sigmoid()
    (6): Linear(in_features=4, out_features=1, bias=True)
    (7): Sigmoid()
  )
)
```



5.3 训练网络

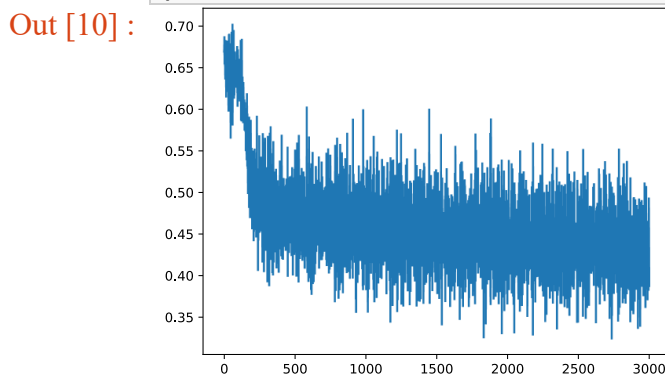
```
In [8]: # 损失函数的选择
loss_fn = nn.BCELoss(reduction='mean')

In [9]: # 优化算法的选择
learning_rate = 0.005 # 设置学习率
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

In [10]: # 训练网络
epochs = 500
losses = [] # 记录损失函数变化的列表

for epoch in range(epochs):
    for (x, y) in train_loader: # 获取小批次的 x 与 y
        Pred = model(x) # 一次前向传播 (小批量)
        loss = loss_fn(Pred, y) # 计算损失函数
        losses.append(loss.item()) # 记录损失函数的变化
        optimizer.zero_grad() # 清理上一轮滞留的梯度
        loss.backward() # 一次反向传播
        optimizer.step() # 优化内部参数

Fig = plt.figure()
plt.plot(range(len(losses)), losses)
plt.show()
```



5.4 测试网络

```
In [11]: # 测试网络
correct = 0
total = 0
with torch.no_grad(): # 该局部关闭梯度计算功能
    for (x, y) in test_loader: # 获取小批次的 x 与 y
        Pred = model(x) # 一次前向传播 (小批量)
        Pred[Pred >= 0.5] = 1
        Pred[Pred < 0.5] = 0
        correct += torch.sum( (Pred == y).all(1) )
        total += y.size(0)
print(f'测试集精准度: {100*correct/total} %')
```

Out [11]: 测试集精准度: 78.50877380371094 %

六、手写数字识别

手写数字识别数据集 (MNIST) 是机器学习领域的标准数据集，它被称为机器学习领域的“Hello World”，只因任何 AI 算法都可以用此标准数据集进行检验。MNIST 内的每一个样本都是一副二维的灰度图像，如图 6-1 所示。

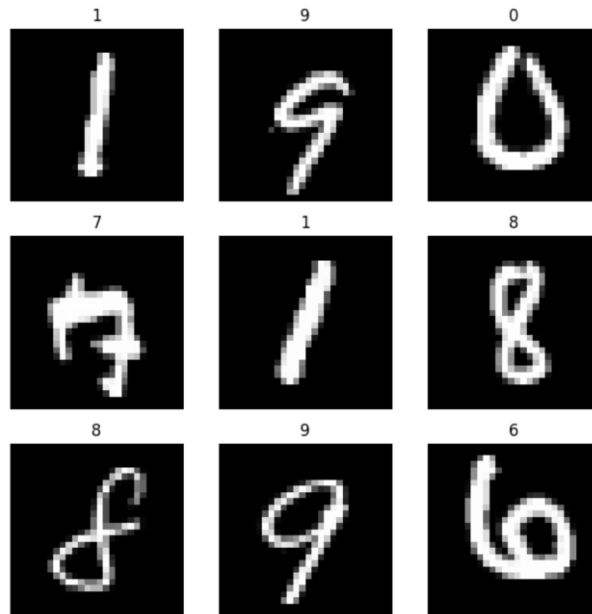
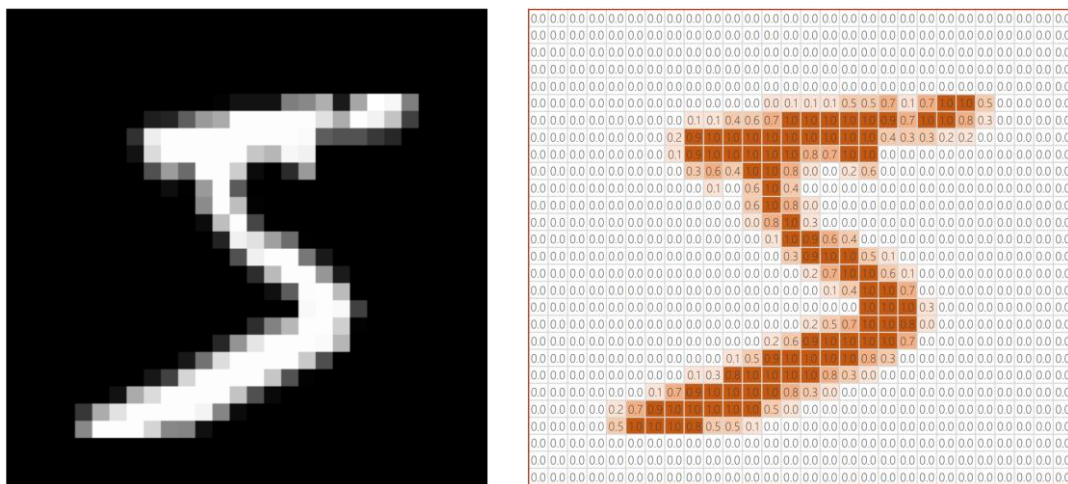


图 6-1 手写数字识别 MNIST 的数据集样本

在 MNIST 中，模型的输入是一副图像，模型的输出就是一个与图像中对应的数字 (0 至 9 之间的一个整数，不是独热编码)。

我们不用手动将输出转换为独热编码，PyTorch 会在整个过程中自动将数据集的输出转换为独热编码。只有在最后测试网络时，我们对比测试集的预测输出与真实输出时，才需要注意一下。

某一个具体的样本如图 6-2 所示，每个图像都是形状为 28×28 的二维数组。



(a) 样本外观

(b) 样本构成

图 6-2 某一个具体的样本

在这种多分类问题中，神经网络的输出层需要一个 softmax 激活函数，它可以把输出层的数据归一化到 0-1 上，且加起来为 1，这样就模拟出了概率的意思。



6.1 制作数据集

这一章我们需要在 `torchvision` 库中分别下载训练集与测试集，因此需要从 `torchvision` 库中导入 `datasets` 以下载数据集，下载前需要借助 `torchvision` 库中的 `transforms` 进行图像转换，将数据集变为张量，并调整数据集的统计分布。

由于不需要手动构建数据集，因此不导入 `utils` 中的 `Dataset`；又由于训练集与测试集是分开下载的，因此不导入 `utils` 中的 `random_split`。

```
In [1]: import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision import datasets
import matplotlib.pyplot as plt
%matplotlib inline
```

```
In [2]: # 展示高清图
from matplotlib_inline import backend_inline
backend_inline.set_matplotlib_formats('svg')
```

在下载数据集之前，要设定转换参数：`transform`，该参数里解决两个问题：

- **ToTensor**：将图像数据转为张量，且调整三个维度的顺序为 $C*W*H$ ； C 表示通道数，二维灰度图像的通道数为 1，三维 RGB 彩图的通道数为 3。
- **Normalize**：将神经网络的输入数据转化为标准正态分布，训练更好；根据统计计算，MNIST 训练集所有像素的均值是 0.1307、标准差是 0.3081。

```
In [3]: # 制作数据集

# 数据集转换参数
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(0.1307, 0.3081)
])

# 下载训练集与测试集
train_Data = datasets.MNIST(
    root = 'D:/Jupyter/dataset/mnist/', # 下载路径
    train = True, # 是 train 集
    download = True, # 如果该路径没有该数据集，就下载
    transform = transform # 数据集转换参数
)
test_Data = datasets.MNIST(
    root = 'D:/Jupyter/dataset/mnist/', # 下载路径
    train = False, # 是 test 集
    download = True, # 如果该路径没有该数据集，就下载
    transform = transform # 数据集转换参数
)
```




```
In [4]: # 批次加载器
train_loader = DataLoader(train_Data, shuffle=True, batch_size=64)
test_loader = DataLoader(test_Data, shuffle=False, batch_size=64)
```

6.2 搭建神经网络

每个样本的输入都是形状为 28×28 的二维数组，那么对于 DNN 来说，输入层的神经元节点就要有 $28 \times 28 = 784$ 个；输出层使用独热编码，需要 10 个节点。

```
In [5]: class DNN(nn.Module):

    def __init__(self):
        ''' 搭建神经网络各层 '''
        super(DNN,self).__init__()
        self.net = nn.Sequential(
            nn.Flatten(),                # 按顺序搭建各层
            # 把图像铺平成一维
            nn.Linear(784, 512), nn.ReLU(), # 第 1 层：全连接层
            nn.Linear(512, 256), nn.ReLU(), # 第 2 层：全连接层
            nn.Linear(256, 128), nn.ReLU(), # 第 3 层：全连接层
            nn.Linear(128, 64), nn.ReLU(),  # 第 4 层：全连接层
            nn.Linear(64, 10)              # 第 5 层：全连接层
        )

    def forward(self, x):
        ''' 前向传播 '''
        y = self.net(x)                  # x 即输入数据
        return y                          # y 即输出数据
```

```
In [6]: model = DNN().to('cuda:0')    # 创建子类的实例，并搬到 GPU 上
model                                       # 查看该实例的各层
```

```
Out [6]: DNN(
  (net): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=784, out_features=512, bias=True)
    (2): ReLU()
    (3): Linear(in_features=512, out_features=256, bias=True)
    (4): ReLU()
    (5): Linear(in_features=256, out_features=128, bias=True)
    (6): ReLU()
    (7): Linear(in_features=128, out_features=64, bias=True)
    (8): ReLU()
    (9): Linear(in_features=64, out_features=10, bias=True)
  )
)
```



6.3 训练网络

```
In [7]: # 损失函数的选择
loss_fn = nn.CrossEntropyLoss()    # 自带 softmax 激活函数
```

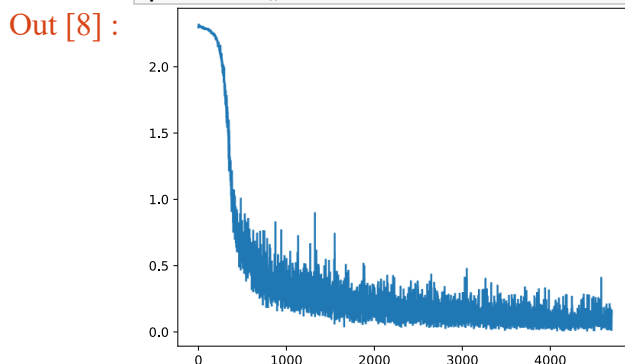
```
In [8]: # 优化算法的选择
learning_rate = 0.01    # 设置学习率
optimizer = torch.optim.SGD(
    model.parameters(),
    lr = learning_rate,
    momentum = 0.5
)
```

在 In [8] 中，给优化器了一个新参数 `momentum`（动量），它使梯度下降算法有了力与惯性，该方法给人的感觉就像是小球在地面上滚动一样。

```
In [9]: # 训练网络
epochs = 5
losses = []    # 记录损失函数变化的列表

for epoch in range(epochs):
    for (x, y) in train_loader:    # 获取小批次的 x 与 y
        x, y = x.to('cuda:0'), y.to('cuda:0')
        Pred = model(x)    # 一次前向传播（小批量）
        loss = loss_fn(Pred, y)    # 计算损失函数
        losses.append(loss.item())    # 记录损失函数的变化
        optimizer.zero_grad()    # 清理上一轮滞留的梯度
        loss.backward()    # 一次反向传播
        optimizer.step()    # 优化内部参数

Fig = plt.figure()
plt.plot(range(len(losses)), losses)
plt.show()
```



注意，由于数据集内部进不去，只能在循环的过程中取出一部分样本，就立即将之搬到 GPU 上。



6.4 测试网络

```
In [10]: # 测试网络
correct = 0
total = 0

with torch.no_grad():          # 该局部关闭梯度计算功能
    for (x, y) in test_loader:  # 获取小批次的 x 与 y
        x, y = x.to('cuda:0'), y.to('cuda:0')
        Pred = model(x)        # 一次前向传播 (小批量)
        _, predicted = torch.max(Pred.data, dim=1)
        correct += torch.sum( (predicted == y) )
        total += y.size(0)

print(f'测试集精准度: {100*correct/total} %')
```

Out [10]: 测试集精准度: 96.65999603271484 %

`a, b = torch.max(Pred.data, dim=1)`的意思是，找出 `Pred` 每一行里的最大值，数值赋给 `a`，所处位置赋给 `b`。因此上述代码里的 `predicted` 就相当于把独热编码转换回了普通的阿拉伯数字，这样一来可以直接与 `y` 进行比较。

由于此处 `predicted` 与 `y` 是一阶张量，因此 `correct` 行的结尾不能加`.all(1)`。