

# 实验 1-深度学习框架熟悉

学号：1190201215

姓名：冯开来

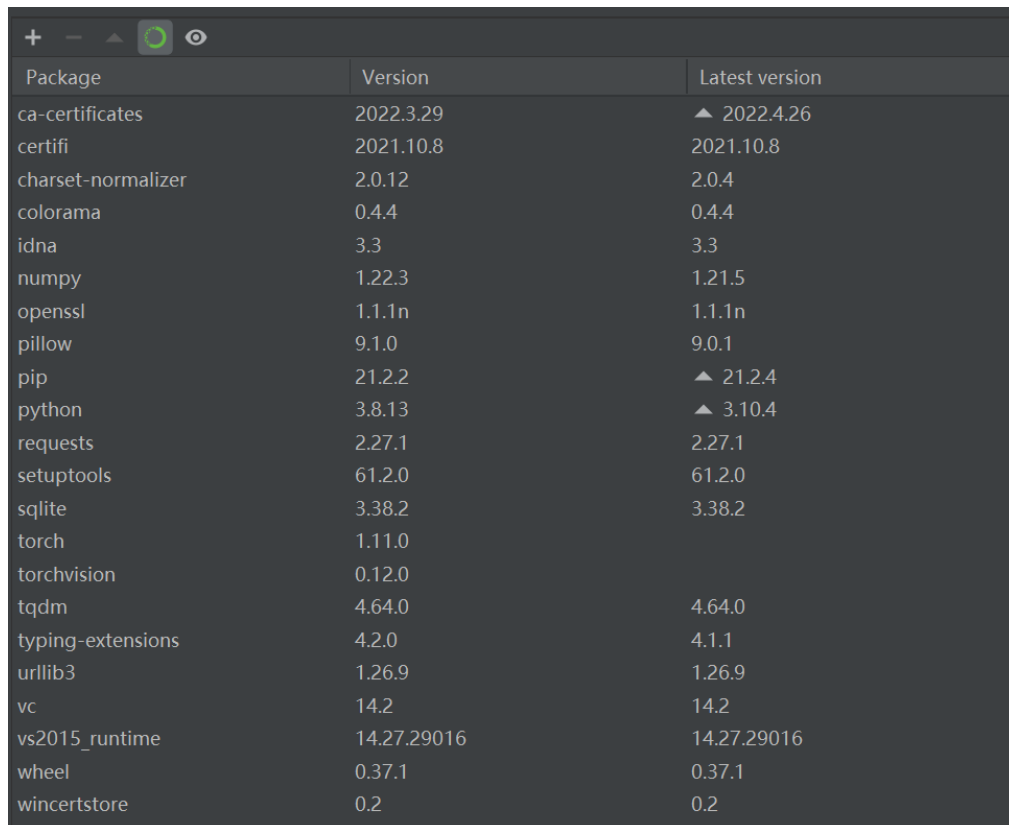
## 一、 实验目的：

使用 PyTorch 实现 MLP，并在 MNIST 数据集上验证。

## 二、 实验环境：

(在文件 requirements.txt 已经体现)

Python=3.8.13; torch=1.11; torchvision=0.12; CPU



| Package            | Version     | Latest version |
|--------------------|-------------|----------------|
| ca-certificates    | 2022.3.29   | ▲ 2022.4.26    |
| certifi            | 2021.10.8   | 2021.10.8      |
| charset-normalizer | 2.0.12      | 2.0.4          |
| colorama           | 0.4.4       | 0.4.4          |
| idna               | 3.3         | 3.3            |
| numpy              | 1.22.3      | 1.21.5         |
| openssl            | 1.1.1n      | 1.1.1n         |
| pillow             | 9.1.0       | 9.0.1          |
| pip                | 21.2.2      | ▲ 21.2.4       |
| python             | 3.8.13      | ▲ 3.10.4       |
| requests           | 2.27.1      | 2.27.1         |
| setuptools         | 61.2.0      | 61.2.0         |
| sqlite             | 3.38.2      | 3.38.2         |
| torch              | 1.11.0      |                |
| torchvision        | 0.12.0      |                |
| tqdm               | 4.64.0      | 4.64.0         |
| typing-extensions  | 4.2.0       | 4.1.1          |
| urllib3            | 1.26.9      | 1.26.9         |
| vc                 | 14.2        | 14.2           |
| vs2015_runtime     | 14.27.29016 | 14.27.29016    |
| wheel              | 0.37.1      | 0.37.1         |
| wincertstore       | 0.2         | 0.2            |

## 三、 实验内容：

首先定义基本参数，包括 batch\_size, epochs, 优化器, 文件路径等：

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='MLP')
    parser.add_argument('--epochs', default=10, type=int)
    parser.add_argument('--batch_size', default=20, type=int)
    parser.add_argument('--output-dir', default="models")
    parser.add_argument('--seed', default=2, type=int)
    parser.add_argument('--optimizer', default="adam", type=str)
    args = parser.parse_args()
```

## 1. 数据读取

数据读取为 MNIST 官方数据集，分为训练集和测试集。如果文件中没有设定为自行下载，转为张量形式返回。然后创建一个 DataLoader 对象，batch\_size 为 20（相当于一次取 20 张图片），然后循环这个 DataLoader 对象，将其中数据加载到模型中进行训练。

```
# Dataset
dataset_train = datasets.MNIST(root='./dataset/mnist', train=True,
                                download=True, transform=transforms.ToTensor())
dataset_test = datasets.MNIST(root='./data/mnist', train=False,
                                download=True, transform=transforms.ToTensor())
data_loader_train = DataLoader(dataset_train, args.batch_size)
data_loader_test = DataLoader(dataset_test, args.batch_size)
```

## 2. 搭建网络结构

因为 MLP 是全连接神经网络，而 MNIST 数据集中的图片都为 784 个像素先通过 flatten 变为一维，经过两个隐含层，从 784 到 512，从 512 到 128，通过输出层，即 128 到 10。最后输出。

```
class MLP(torch.nn.Module):
    def __init__(self):
        super(MLP, self).__init__() #
        self.Flatten = torch.nn.Flatten()
        # 初始化三层神经网络 两个全连接的隐藏层，一个输出层
        self.fc1 = torch.nn.Linear(784, 512) # 第一个隐含层
        self.fc2 = torch.nn.Linear(512, 128) # 第二个隐含层
        self.fc3 = torch.nn.Linear(128, 10) # 输出层
```

接下来是模型的 forward。前向传播输入值为一维的 Tensor，通过每一层网络的时候使用 ReLU 激活函数。最后一层的输出层选择使用 softmax 激活函数。大致整个流程是  $784 \times 1$  的张量最后输出为  $10 \times 1$  的张量，每个值为 0-9 类别的概率分布，最后选取概率最大的作为预测值输出。

```
def forward(self, x):
    # 前向传播，输入值：x，返回值out
    x = self.Flatten(x) # 将一个多行的Tensor，拼接成一行
    out = F.relu(self.fc1(x)) # 使用 relu 激活函数
    out = F.relu(self.fc2(out))
    out = F.softmax(self.fc3(out), dim=1) # 输出层使用softmax
    # 784×1的张量最后输出为10×1的张量，
    # 每个值为0-9类别的概率分布，最后选取概率最大的作为预测值输出
    return out
```

## 3. 定义优化器

本次实验选用的优化器有 AdamW 和 SGD 两个优化器。根据一开始定义的

参数进行选择。lr 为学习率，momentum 为动量。

```
# Set up optimizers
if args.optimizer == 'sgd':
    optimizer = torch.optim.SGD(params=model.parameters(), lr=0.01, momentum=0.9)
elif args.optimizer in ["adam", "adamw"]:
    optimizer = torch.optim.AdamW(params=model.parameters(), lr=0.0001)
```

#### 4. 定义损失函数并训练

本次实验采用的是损失函数是交叉熵损失函数。

每一轮训练的流程大致是放入 model 进行训练，然后计算 loss，将 loss 反向传播，更新网络参数，然后继续训练直到收敛。在每次进行过一轮 epoch 后，放入 test 集检验正确率。然后将正确率最高的模型，准确率，优化器，epoch 和参数集写入.pth 作为模型文件进行保存。

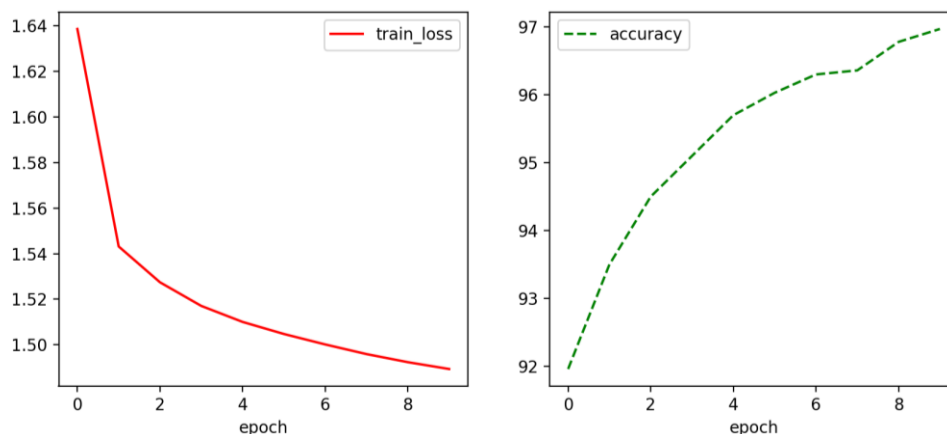
```
if args.output_dir and accuracy > best_metric:
    best_metric = accuracy
    checkpoint_path = output_dir / "BEST_MLP.pth"
    torch.save(
        {
            "model": model.state_dict(),
            "accuracy": best_metric,
            "optimizer": optimizer.state_dict(),
            "epoch": epoch+1,
            "args": args
        },
        checkpoint_path,
    )
```

#### 四、 实验结果与分析：

第一个实验参数作为对照组，依次改变参数的大小判断其对于结果的影响。因为次数太多，我这里就每个参数集就找了其中一次实验作为报告内容。

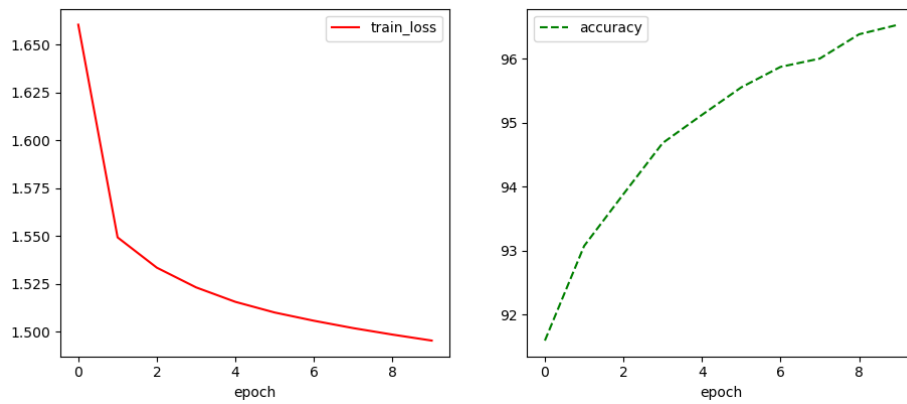
1. 参数：batch\_size=20; epochs=10; optimizier: adam(lr=0.0001);  
最优结果：

正确率：96.67%； 时间：4min59s； 损失：1.489431； epoch: 10



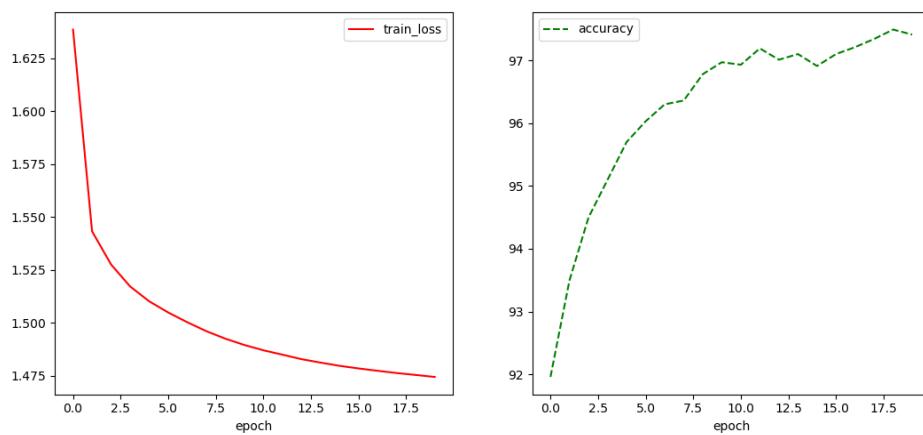
2. 参数: batch\_size=30; epochs=10; optimizer: adam(lr=0.0001);  
最优结果:

正确率: 96.53%; 时间: 3min53s; 损失: 1.495285; epoch: 10



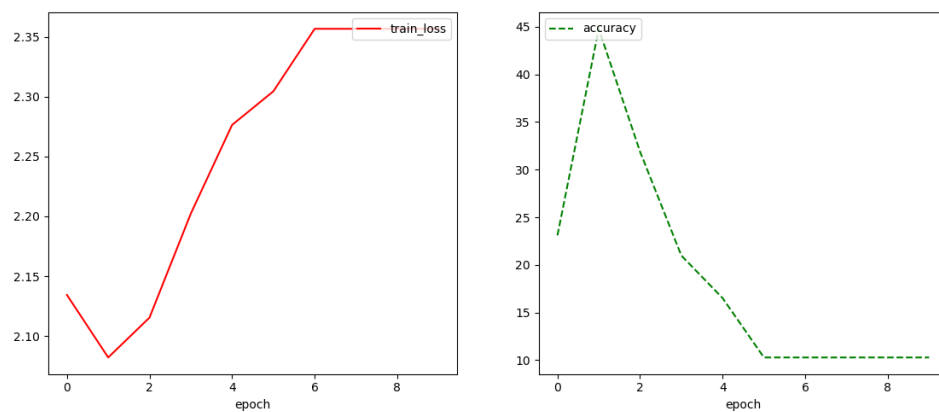
3. 参数: batch\_size=20; epochs=20; optimizer: adam(lr=0.0001);  
最优结果:

正确率: 97.49%; 时间: 7min54s; 损失: 1.475271; epoch: 19



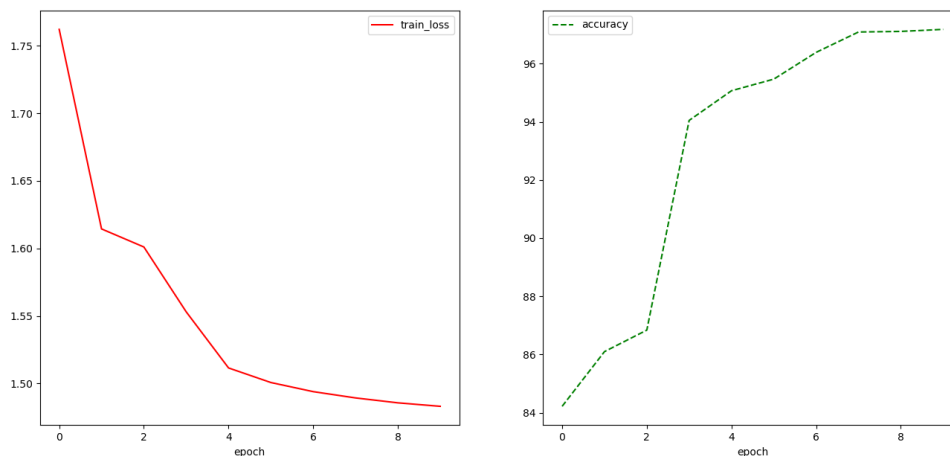
4. 参数: batch\_size=20; epochs=10; optimizer: adam(lr=0.01);  
最优结果:

正确率: 44.79%; 时间: 5min04s; 损失: 2.082089; epoch: 2



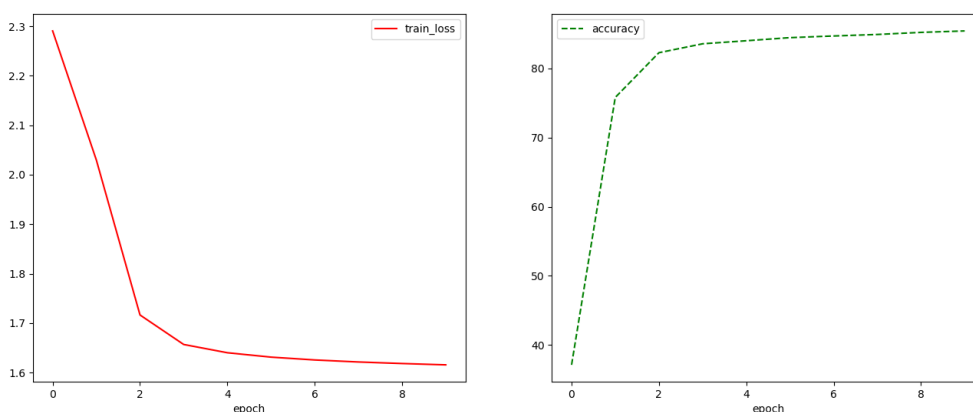
5. 参数: `batch_size=20`; `epochs=10`; `optimizer: sgd(lr=0.01)`;  
最优结果:

正确率: 97.18%; 时间: 4min44s; 损失: 1.483129; epoch: 10



4. 参数: `batch_size=20`; `epochs=10`; `optimizer: sgd(lr=0.0001)`;  
最优结果:

正确率: 85.42%; 时间: 4min25s; 损失: 1.615741; epoch: 10



纵观以上实验（当然现实不止做了这么多），可以看到：`batch_size`的提升虽然不能显著提升效果，但能减少训练时间；而`epochs`的提高可以显著提升效果；而当学习率调整的时候，结果就很有意思了，很容易出现已经达到了最低点但是迈过去的情况，而且最终的结果也是不如小学习率的情况；当换了优化器后发现，同样的学习率，`sgd`表现得更好，时间也更快一些；而在最后一次实验中，我们降低了`sgd`的`lr`，但是准确率上不来的原因在于`epoch`数太小了，如果调大`epoch`，那一定会得到一个很好的结果。

因为电脑没有 `gpu`，每一次等了挺久，所以没有尝试 100 以上的 `epoch`。想着尝试用 `colab`，而因为第一次写深度学习框架，报告写了一半才了解怎么把模型和参数放到 `cuda` 中训练，下次实验争取写出 `cuda` 版本的。