

LAB2: Neural Network

21304219

刘文婧

Requirements

- 在给定的训练数据集上，分别训练一个线性分类器（Softmax 分类器），多层感知机（MLP）和卷积神经网络（CNN）
- 在 MLP 实验中，研究使用不同网络层数和不同神经元数量 对模型性能的影响
- 在 CNN 实验中，以 LeNet 模型为基础，探索不同模型结构因素（如：卷积层数、滤波器数量、Pooling 的使用等）对模型性能的影响
- 分别使用 SGD 算法、SGD Momentum 算法和 Adam 算法训练模型，观察并讨论他们对模型训练速度和性能的影响
- 比较并讨论线性分类器、MLP 和 CNN 模型在 CIFAR-10 图像分类任务上的性能区别
- 学习一种主流的深度学习框架（如：Tensorflow, PyTorch, MindSpore），并用其中一种框架完成上述神经网络模型的实验

Softmax 线性分类器

- 第n个样本记为

$$\mathbf{x}^{(n)} = \begin{pmatrix} x_1^{(n)} \\ x_2^{(n)} \\ \vdots \\ x_m^{(n)} \end{pmatrix}$$

- 权重矩阵 \mathbf{W} 记为：

$$\mathbf{W} = (\mathbf{w}_1 \quad \mathbf{w}_2 \quad \cdots \quad \mathbf{w}_K), \quad \mathbf{w}_i \in \mathbb{R}^m$$

其中 \mathbf{w}_i 是对应给第 i 个类别的权重向量，有 $\mathbf{w}_i \in \mathbb{R}^m$ 。

- Softmax 线性分类器做的就是：

$$\begin{aligned} \mathbf{x}^{(n)\top} \mathbf{W} + \mathbf{b} &= \begin{pmatrix} x_1^{(n)} & x_2^{(n)} & \cdots & x_m^{(n)} \end{pmatrix} (\mathbf{w}_1 \quad \mathbf{w}_2 \quad \cdots \quad \mathbf{w}_K) + (b_1 \quad b_2 \quad \cdots \quad b_K) \\ &= \begin{pmatrix} z_1^{(n)} & z_2^{(n)} & \cdots & z_K^{(n)} \end{pmatrix} \end{aligned}$$

$$\begin{aligned} \text{Softmax}(\mathbf{x}^{(n)\top} \mathbf{W} + \mathbf{b}) &= \text{Softmax}\left(\left(z_1^{(n)} \quad z_2^{(n)} \quad \cdots \quad z_K^{(n)}\right)\right) \\ &= \left(\frac{e^{z_1^{(n)}}}{\sum_{k=1}^K e^{z_k^{(n)}}} \quad \frac{e^{z_2^{(n)}}}{\sum_{k=1}^K e^{z_k^{(n)}}} \quad \cdots \quad \frac{e^{z_K^{(n)}}}{\sum_{k=1}^K e^{z_k^{(n)}}}\right) \end{aligned}$$

- 损失函数：

$$\text{Loss}^{(n)} = - \sum_{k=1}^K y_k^{(n)} \log \left[\text{Softmax}_k \left(\mathbf{x}^{(n)\top} \mathbf{W} + \mathbf{b} \right) \right]$$

其中， $\mathbf{y}^{(n)} = [y_1^{(n)}, y_2^{(n)}, \dots, y_K^{(n)}]$ 是样本标签的 One-Hot 编码， $\text{Softmax}_k(\mathbf{x}^{(n)\top} \mathbf{W} + \mathbf{b})$ 是 Softmax 计算的第 k 个分量，也就是样本被分成该类别的一种“概率值”。

- 然后，梯度下降更新时，使用的是：

$$\mathbf{W} = \mathbf{W} - \alpha \frac{\partial \text{Loss}(\mathbf{W})}{\partial \mathbf{W}}$$

其中偏导公式：

$$\begin{aligned} \frac{\partial \text{Loss}(\mathbf{W})}{\partial \mathbf{W}} &= \frac{1}{N} \sum_{n=1}^N \mathbf{x}^{(n)} \left[\text{Softmax}(\mathbf{x}^{(n)\top} \mathbf{W} + \mathbf{b}) - \mathbf{y}^{(n)} \right] \\ &= \frac{1}{N} (\mathbf{x}^{(1)} \quad \mathbf{x}^{(2)} \quad \cdots \quad \mathbf{x}^{(N)}) \begin{pmatrix} \text{Softmax}(\mathbf{x}^{(1)\top} \mathbf{W} + \mathbf{b}) - \mathbf{y}^{(1)} \\ \text{Softmax}(\mathbf{x}^{(2)\top} \mathbf{W} + \mathbf{b}) - \mathbf{y}^{(2)} \\ \vdots \\ \text{Softmax}(\mathbf{x}^{(N)\top} \mathbf{W} + \mathbf{b}) - \mathbf{y}^{(N)} \end{pmatrix} \end{aligned}$$

其中每个 $x^{(n)} \in \mathbb{R}^m$ ，是列向量；每个 $\text{Softmax}(\mathbf{x}^{(n)\top} \mathbf{W} + \mathbf{b}) - \mathbf{y}^{(n)} \in \mathbb{R}^K$ ，是行向量。

- 一般地，反向传播中，梯度下降的

$$\mathbf{W} = \mathbf{W} - \alpha \frac{\partial \text{Loss}(\mathbf{W})}{\partial \mathbf{W}}$$

公式中，有，

$$\frac{\partial \text{Loss}(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{N} \sum_{n=1}^N x^{(n)} \cdot \text{predict-error}^{(n)}$$

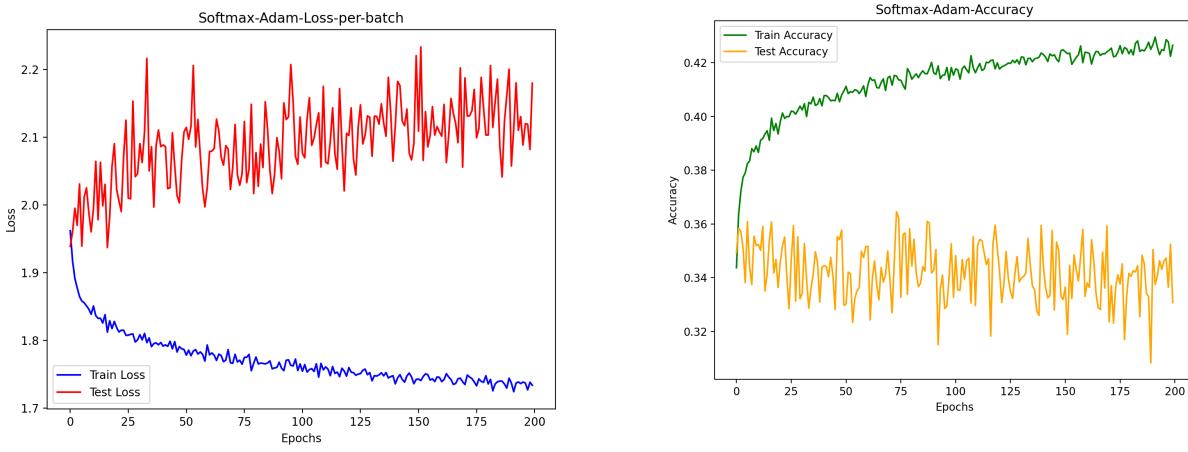
- 使用 pytorch 时，Softmax 其实可以看作就是一个全连接层，激活函数是softmax函数

```
class SoftmaxNN(nn.Module):
    def __init__(self, input_size, output_size):
        super().__init__()
        super(SoftmaxNN, self).__init__()
        # only one fc layer
        self.fc = nn.Linear(input_size, output_size) # xW+b
        # self.softmax = nn.Softmax(dim=1) ---> use log_softmax later

    def forward(self, x):
        x = self.fc(x)
        x = F.log_softmax(x, dim=1)
        return x
```

Observation :

1. 如果选用Adam优化器，Softmax 线性分类器 在图像分类上的性能并不是很好，在测试集上的准确率最高也就 35 % 左右。（后面比较SGD与Adam的实验中，使用了SGD，见下面的实验部分）
2. 如下图可以看到，其实训练大约 50 个epoch就开始过拟合了。
3. Softmax网络结构比较简单，训练较快。



MLP (多层感知机)

研究使用不同网络层数和不同神经元数量对模型性能的影响

- Multilayer Perception , 即多层的全连接神经网络。
- 前向传播 :

$$\begin{aligned}\tilde{\mathbf{h}}_{\ell+1} &= \mathbf{W}_{\ell+1} \mathbf{h}_\ell \\ \mathbf{h}_{\ell+1} &= \alpha(\tilde{\mathbf{h}}_{\ell+1}) = \alpha(\mathbf{W}_{\ell+1} \mathbf{h}_\ell)\end{aligned}$$

- 反向传播 :

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{h}_\ell} &= \mathbf{W}_{\ell+1}^\top \left(\frac{\partial L}{\partial \mathbf{h}_{\ell+1}} \odot \tilde{\mathbf{g}}_{\ell+1} \right) \\ \frac{\partial L}{\partial \mathbf{W}_\ell} &= \left(\frac{\partial L}{\partial \mathbf{h}_\ell} \odot \tilde{\mathbf{g}}_\ell \right) \mathbf{h}_{\ell-1}^\top\end{aligned}$$

其中,

$$\tilde{\mathbf{g}}_\ell = \begin{pmatrix} \frac{\partial \mathbf{h}_{\ell,1}}{\partial \tilde{\mathbf{h}}_{\ell,1}} \\ \frac{\partial \mathbf{h}_{\ell,2}}{\partial \tilde{\mathbf{h}}_{\ell,2}} \\ \vdots \\ \frac{\partial \mathbf{h}_{\ell,m}}{\partial \tilde{\mathbf{h}}_{\ell,m}} \end{pmatrix}$$

Observation1 :

如果train函数中选择的是 `criterion = nn.CrossEntropyLoss()` , 那么网络结构中最后不要用Softmax, 否则Softmax操作是重复的, 可能导致梯度爆炸。

- 做 SGD 和 CNN 实验都不会出现 loss 增加的问题, 但做 MLP 实验发现 loss 越来越大, 搜索后发现问题出在这里 :
- pyTorch中的 `nn.CrossEntropyLoss()` , 做的事情并不是我一开始理解的如下公式

$$\text{CELoss}^{(n)} = - \sum_{k=1}^K \text{labels}_k^{(n)} \cdot \log \left(\text{predicts}^{(n)} \right)$$

- 事实上，`nn.CrossEntropyLoss()` 把 `Softmax` 也做了，即类似于

$$\text{pyTorch-CELoss}^{(n)} = - \sum_{k=1}^K \text{labels}_k^{(n)} \cdot \log \left(\text{Softmax}(\text{predicts}^{(n)}) \right)$$

所以，我一开始在 `criterion = nn.CrossEntropyLoss()` 的同时，如下这样写的MLP实验，梯度爆炸了：

```
class MLP(nn.Module):
    def __init__(self, input_size, output_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, out_features: 1024)
        self.fc2 = nn.Linear(in_features: 1024, out_features: 512)
        self.fc3 = nn.Linear(in_features: 512, out_features: 128)
        self.fc4 = nn.Linear(in_features: 128, out_features: 32)
        self.fc5 = nn.Linear(in_features: 32, output_size)
        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        x = self.relu(x)
        x = self.fc4(x)
        x = self.relu(x)
        x = self.fc5(x)
        out = self.softmax(x)
        return out
```

epoch:2, Average train_loss per batch:2.3178787127685547, train_accuracy:0.14548
epoch:2, Average test_loss per batch:2.3219804794311525, test_accuracy:0.1501
epoch:3, batch:0, loss of one batch:2.320525646209717
epoch:3, batch:100, loss of one batch:2.320525646209717
epoch:3, batch:200, loss of one batch:2.242400646209717
epoch:3, batch:300, loss of one batch:2.398650646209717
epoch:3, batch:400, loss of one batch:2.2736501693725586
epoch:3, batch:500, loss of one batch:2.398650646209717
epoch:3, batch:600, loss of one batch:2.414275646209717
epoch:3, batch:700, loss of one batch:2.2892751693725586
epoch:3, Average train_loss per batch:2.3435906369018555, train_accuracy:0.11974
epoch:3, Average test_loss per batch:2.3729640701293944, test_accuracy:0.1
epoch:4, batch:0, loss of one batch:2.398650646209717
epoch:4, batch:100, loss of one batch:2.3049001693725586
epoch:4, batch:200, loss of one batch:2.383025646209717
epoch:4, batch:300, loss of one batch:2.3205254077911377
epoch:4, batch:400, loss of one batch:2.3830254077911377
epoch:4, batch:500, loss of one batch:2.367400646209717
epoch:4, batch:600, loss of one batch:2.289275646209717
epoch:4, batch:700, loss of one batch:2.3674004077911377
epoch:4, Average train_loss per batch:2.3633932525634767, train_accuracy:0.1
epoch:4, Average test_loss per batch:2.3729640701293944, test_accuracy:0.1
epoch:5, batch:0, loss of one batch:2.3830254077911377
epoch:5, batch:100, loss of one batch:2.2736501693725586
epoch:5, batch:200, loss of one batch:2.398650646209717

最后一层用Softmax()

可以看到，loss不降反升，accuracy直接降到了0.1。

- 然后，pyTorch里有一个 `nn.NLLLoss()`，做的操作大概类似于下面这个公式：

$$\text{CELoss}^{(n)} = - \sum_{k=1}^K \text{labels}_k^{(n)} \cdot \left(\text{predicts}^{(n)} \right)$$

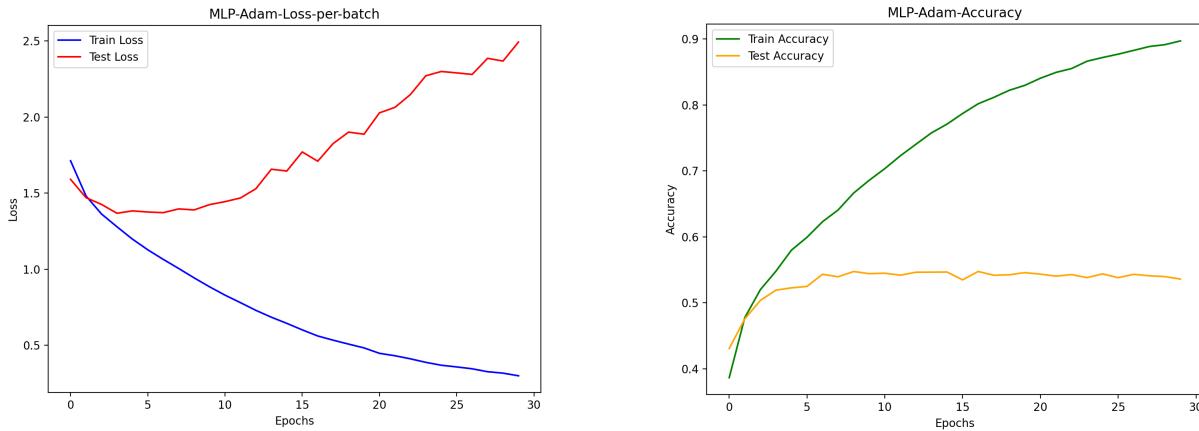
- 所以，考虑到我期望的交叉熵损失，其实是 `log[Softmax()]` 与 `nn.NLLLoss()` 结合的结果，我就改写了代码：（并且 `criterion = nn.NLLLoss()`）

```
class MLP(nn.Module):
    def __init__(self, input_size, output_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, out_features: 1024)
        self.fc2 = nn.Linear(in_features: 1024, out_features: 512)
        self.fc3 = nn.Linear(in_features: 512, out_features: 128)
        self.fc4 = nn.Linear(in_features: 128, out_features: 32)
        self.fc5 = nn.Linear(in_features: 32, output_size)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        x = self.relu(x)
        x = self.fc4(x)
        x = self.relu(x)
        x = self.fc5(x)
        x = F.log_softmax(x, dim=1)
        return x
```

Observation2：如下图MLP

1. 像上面这样，我一开始定义了5个全连接层，网络层数较多，训练起来很慢，但收敛得会比较快，效果也不错，比Softmax要好，在测试集上的准确率可以到达 55% 左右。
2. 当然，最后也过拟合了，5个epochs 就已经能得到一个较好的模型了，大概15个epochs 后测试集上的loss就会越来越大。

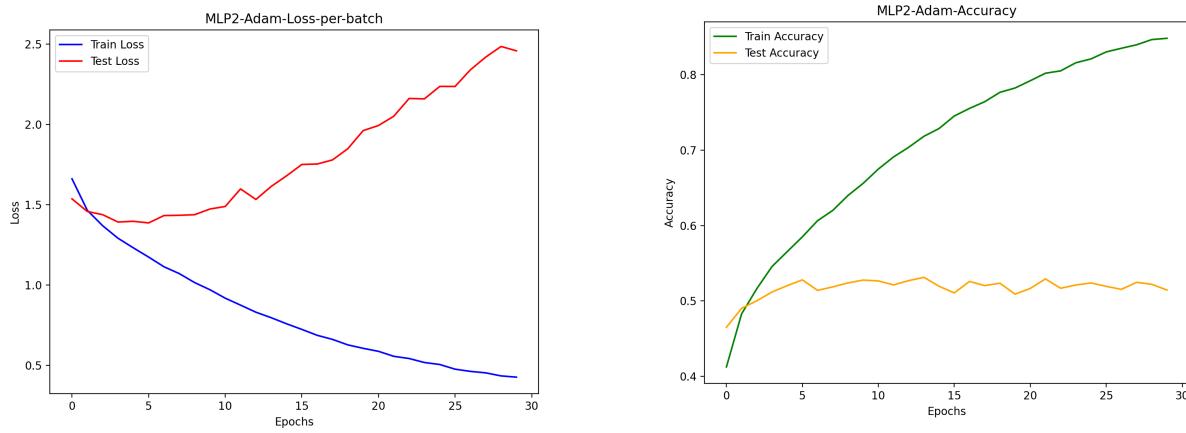


Observation3：减少网络层数，变成只有3层，观察性能变化，如下图MLP2

1. 发现模型训练明显变快了，因为模型变简单了；
2. 同时，30 个epochs后，发现仍然具有和上面5层的网络差不多好的性能，可以在测试集上达到 52% 左右的准确率。
3. 最后也过拟合了。
4. 可见，如果希望训练效率较高，即花更短的时间到达一定的准确率，并不需要太多的网络层数，**3个全连接层就已经有一定的好效果了。**

```
class MLP2(nn.Module):
    def __init__(self, input_size, output_size):
        super(MLP2, self).__init__()
        self.fc1 = nn.Linear(input_size, out_features: 512)
        self.fc3 = nn.Linear(in_features: 512, out_features: 128)
        self.fc5 = nn.Linear(in_features: 128, output_size)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc3(x)
        x = self.relu(x)
        x = self.fc5(x)
        x = F.log_softmax(x, dim=1)
        return x
```

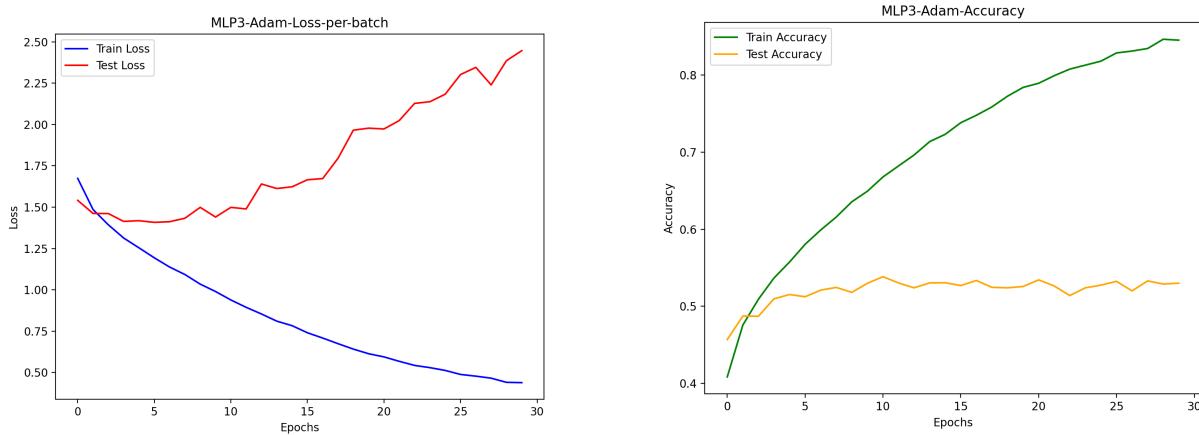


Observation4：改变不同全连接层中的神经元个数，观察性能变化

1. 如果倾向于使用更多的神经元个数，如下图的MLP3；
2. 更多神经元，模型要计算的更多，训练于是更慢；
3. 比起把神经元拆分到更多的网络层（指网络层数更多、但每层神经元个数少一些的情况），这种做法效率要低一些，最终的测试集准确率也就53%左右。

```
class MLP3(nn.Module):
    def __init__(self, input_size, output_size):
        super(MLP3, self).__init__()
        self.fc1 = nn.Linear(input_size, out_features: 1024)
        self.fc2 = nn.Linear(in_features: 1024, out_features: 256)
        self.fc3 = nn.Linear(in_features: 256, output_size)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        x = F.log_softmax(x, dim=1)
        return x
```

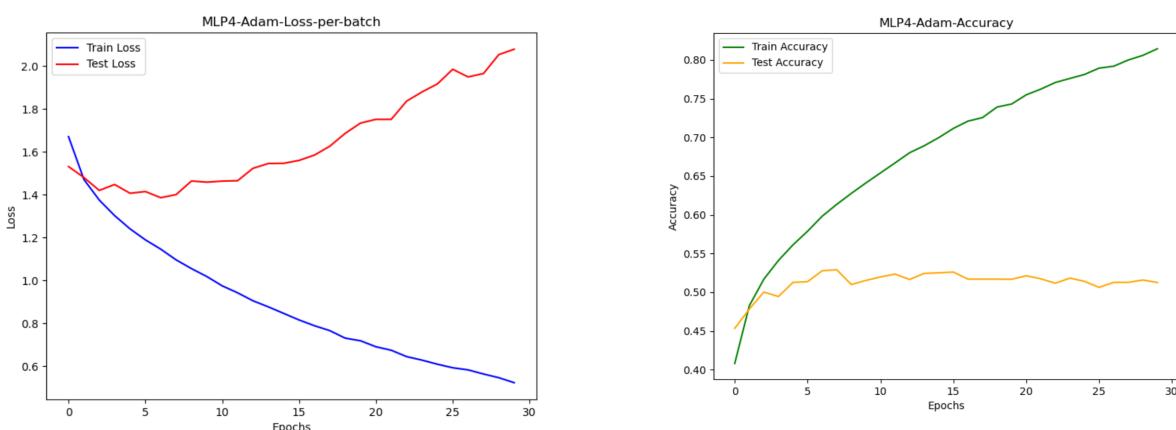


Observation4：改变不同全连接层中的神经元个数，观察性能变化

1. 如果倾向于使用更少的神经元个数，如下图的MLP4
2. 更少神经元，要计算的没那么多，模型于是训练会更快一些，而且相比于MLP3是明显加快的
3. 最终测试集准确率有51%~52%。

```
class MLP4(nn.Module):
    def __init__(self, input_size, output_size):
        super(MLP4, self).__init__()
        self.fc1 = nn.Linear(input_size, out_features: 256)
        self.fc2 = nn.Linear(in_features: 256, out_features: 32)
        self.fc3 = nn.Linear(in_features: 32, output_size)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        x = F.log_softmax(x, dim=1)
        return x
```



Conclusion：

1. 3个全连接层，神经元个数设置成256、128、32等较小的数，效果就已经是MLP模型中能达到的比较好的效果了，测试集上的准确率可以到达 52% 左右；
2. 再增加网络层数、神经元数，准确率没有提升很多 (<5%)，但训练变慢了很多，可见提升效率不佳，这样做的意义不大。
3. 毕竟，只是简单地把图像展平、全连接，就不能很好地对图像进行一小块一小块范围的特征识别了。

CNN（卷积神经网络）

以 LeNet 模型为基础，探索不同模型结构因素（如：卷积层数、滤波器数量、Pooling 的使用等）对模型性能的影响

- LeNet-5:

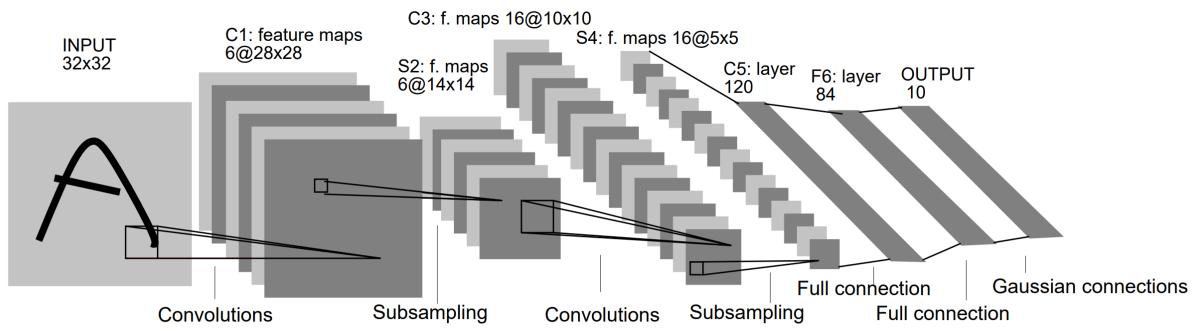
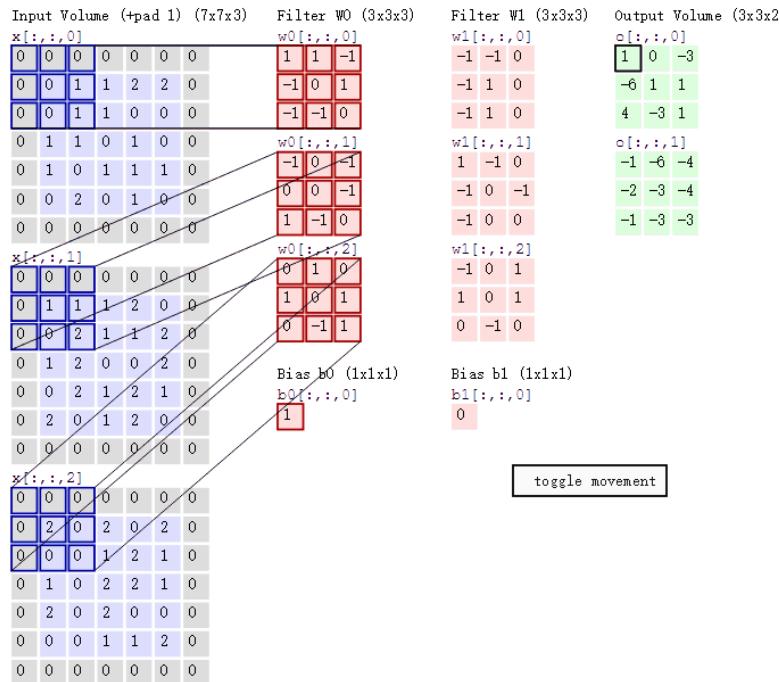


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

- 其中，卷积层：

输入：(number of inputs) × (input height) × (input width) × (input channels)

输出：(number of inputs) × (feature map height) × (feature map width) × (feature map channels)



如上图，3个通道，每个通道都分别用一个filter中对应的卷积核去按位置相乘，而且每个通道各自相乘的结果相加，然后再把三个通道的结果相加再加 bias，就是output 中的1个元素；

输出层需要有多少个通道，这中间就要有多少个 filter。

- 池化层 (subsampling) : 平均池化 或 最大池化。
- 全连接层 (full connection) 。
- 于是，基于LeNet-5，搭建了如下的CNN：

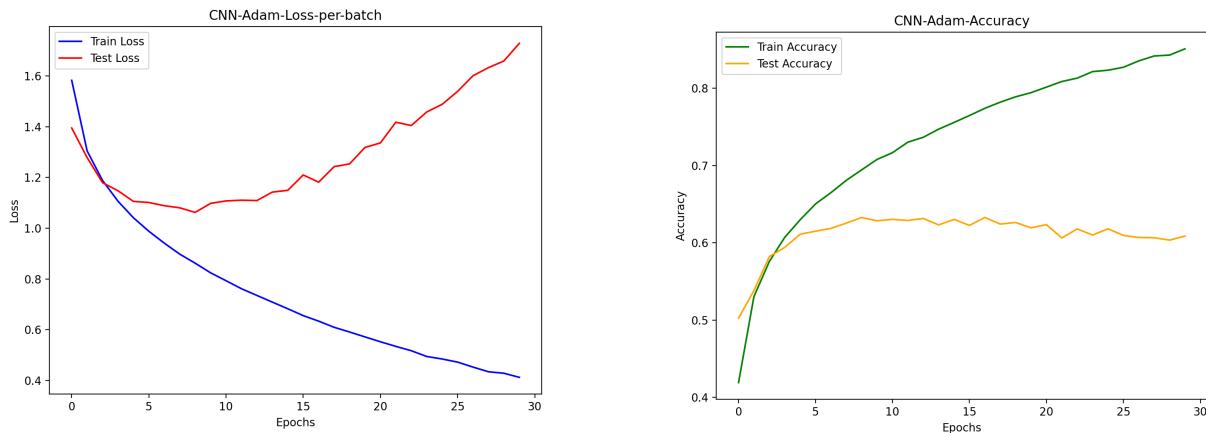
```

class CNN(nn.Module):
    """LeNet-5"""
    def __init__(self, output_size):
        super(CNN, self).__init__()
        # 3x32x32 -> 6x28x28
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=6, kernel_size=5, stride=1, padding=0)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0) # 6x14x14
        # 6x14x14 -> 16x10x10
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1, padding=0)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0) # 16x5x5
        # Note: should flatten 16x5x5 in forward()
        self.fc1 = nn.Linear(16*5*5, out_features=120)
        self.fc2 = nn.Linear(in_features=120, out_features=84)
        self.fc3 = nn.Linear(in_features=84, output_size)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        # flatten
        x = x.view(-1, 16*5*5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        # out = F.relu(self.fc3(x))
        out = F.log_softmax(self.fc3(x), dim=1)
        return out

```

- 效果：



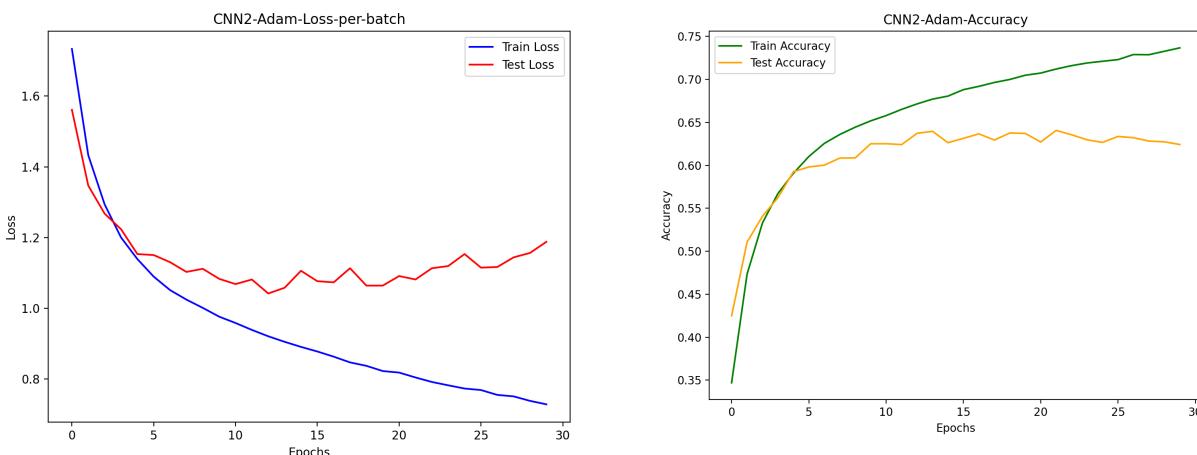
- 与MLP类似地，其实 5个epochs 的训练就能得到1个比较好的模型了，继续训练，会过拟合，在测试集上的loss越来越大；
- 比MLP更好的是，最终在测试集上的准确率可以到 60 % 以上。

Observation1：卷积层数的影响

1. 如下CNN2，我们增加在每个池化层前，多加1个卷积层；
2. 发现对测试集准确率的提升并不大（最终62~63%左右），而且甚至，在训练集上的准确率上升速度反而不如卷积层数更少的模型，在训练集上的loss下降也不如卷积层数更少的模型。也就是增加卷积层数，测试集上的表现变化不大，在训练集上的表现反而下降了；
3. 认为这有可能是我的卷积层加的没有技巧、太过生硬的原因，但可以知道的是，这很明显地反映了图像的识别分类会和图像的内部结构有关，所以盲目地增加卷积层并不会带来什么可见的模型分类效果提升，反而可能使得模型需要更多的训练。

```
class CNN2(nn.Module):  
    def __init__(self, output_size):  
        super(CNN2, self).__init__()  
        # 3x32x32 -> 6x28x28  
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=6, kernel_size=5, stride=1, padding=0)  
        # 6x28x28 -> 8x26x26  
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=8, kernel_size=5, stride=1, padding=1)  
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0) # 8x13x13  
        # 8x13x13 -> 16x9x9  
        self.conv3 = nn.Conv2d(in_channels=8, out_channels=16, kernel_size=5, stride=1, padding=0)  
        # 16x9x9 -> 16x5x5  
        self.conv4 = nn.Conv2d(in_channels=16, out_channels=16, kernel_size=7, stride=1, padding=1)  
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0) # 16x2x2  
  
        self.fc1 = nn.Linear(16*2*2, out_features= 120)  
        self.fc2 = nn.Linear( in_features: 120, out_features: 84)  
        self.fc3 = nn.Linear( in_features: 84, output_size)
```

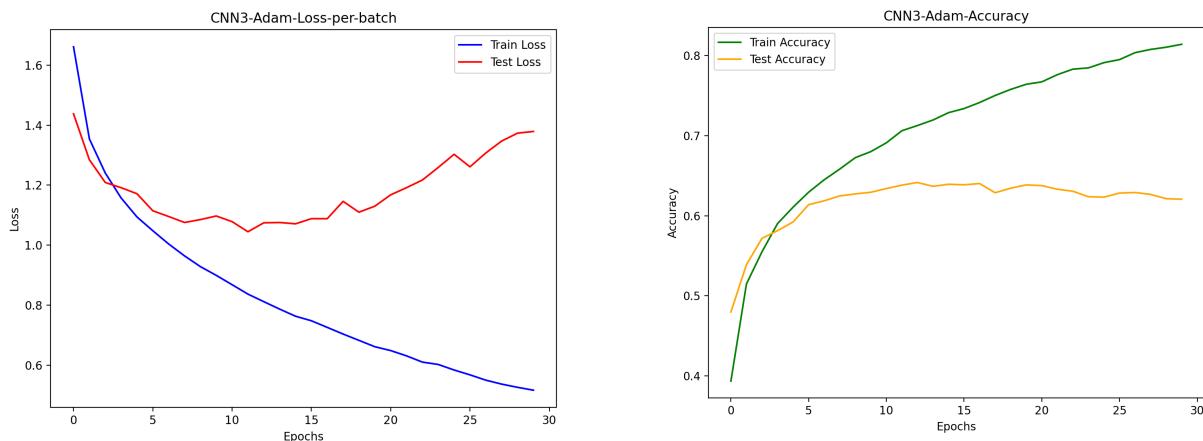
```
def forward(self, x):  
    x = F.relu(self.conv1(x))  
    x = F.relu(self.conv2(x))  
    x = self.pool1(x)  
    x = F.relu(self.conv3(x))  
    x = F.relu(self.conv4(x))  
    x = self.pool2(x)  
    # print(x.size())  
    # flatten  
    x = x.view(-1, 16*2*2)  
    x = F.relu(self.fc1(x))  
    x = F.relu(self.fc2(x))  
    # out = F.relu(self.fc3(x))  
    out = F.log_softmax(self.fc3(x), dim=1)  
    return out
```



Observation2：Pooling的使用

1. 如下CNN3，改用平均池化；
2. 发现平均池化的模型和最大池化的模型表现差不多，最终的测试集准确率都有62%左右；
3. 平均池化在训练集上的loss下降程度、accuracy上升程度没有最大池化那么快
4. 可见，最大池化比平均池化要好一点，这说明图像只需要识别出一小个范围内最明显的表现，也就是提取出最大的值，就能足够代表这个小范围内的图像特征了

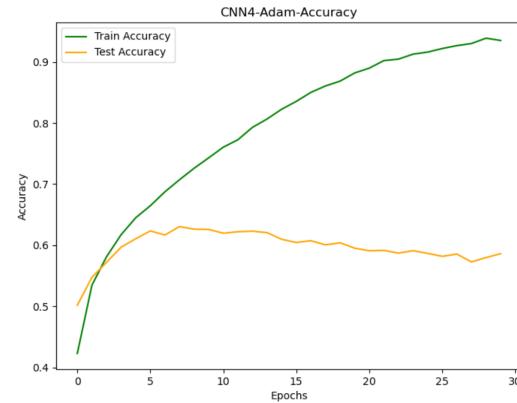
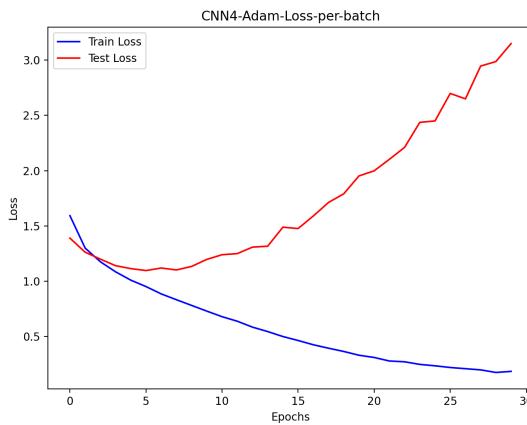
```
class CNN3(nn.Module):  
    def __init__(self, output_size):  
        super(CNN3, self).__init__()  
        # 3x32x32 -> 6x28x28  
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=6, kernel_size=5, stride=1, padding=0)  
        self.pool1 = nn.AvgPool2d(kernel_size=2, stride=2, padding=0) # 6x14x14  
        # 6x14x14 -> 16x10x10  
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1, padding=0)  
        self.pool2 = nn.AvgPool2d(kernel_size=2, stride=2, padding=0) # 16x5x5  
        # Note: should flatten 16x5x5 in forward()  
        self.fc1 = nn.Linear(16*5*5, out_features: 120)  
        self.fc2 = nn.Linear(in_features: 120, out_features: 84)  
        self.fc3 = nn.Linear(in_features: 84, output_size)  
  
    def forward(self, x):  
        x = F.relu(self.conv1(x))  
        x = self.pool1(x)  
        x = F.relu(self.conv2(x))  
        x = self.pool2(x)  
        # flatten  
        x = x.view(-1, 16*5*5)  
        x = F.relu(self.fc1(x))  
        x = F.relu(self.fc2(x))  
        # out = F.relu(self.fc3(x))  
        out = F.log_softmax(self.fc3(x), dim=1)  
        return out
```



Observation2：Pooling的使用

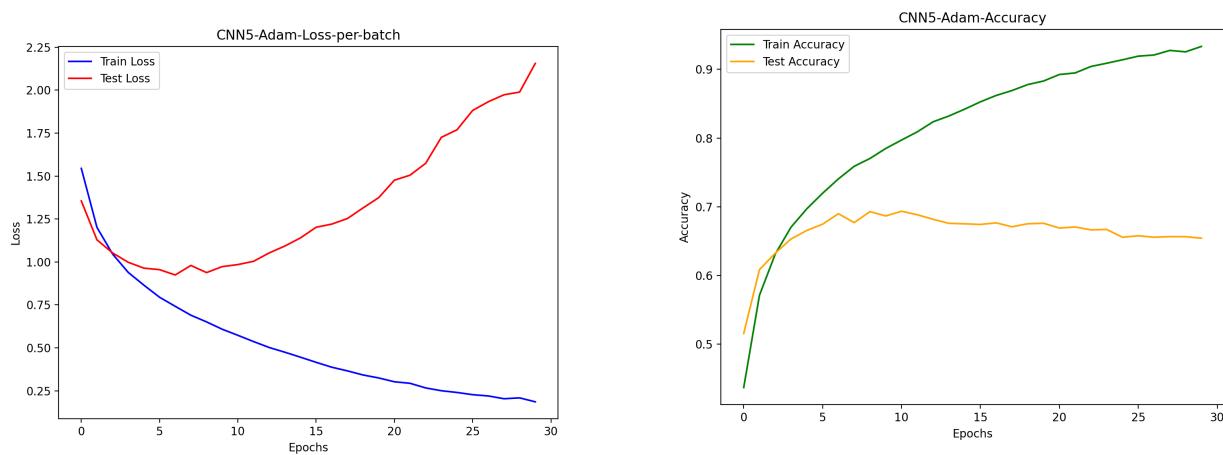
1. 如下CNN4，去掉1个池化层，把更多的神经元数据交给全连接层
2. 测试集准确率变化不大，但发现过拟合现象明显更严重了
3. 可见池化层有助于识别图像的“平均特征”，毕竟为了提升泛化能力，其实并不需要对训练集的一张图片识别得过分仔细。

```
class CNN4(nn.Module):  
    def __init__(self, output_size):  
        super(CNN4, self).__init__()  
        # 3x32x32 -> 6x28x28  
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=6, kernel_size=5, stride=1, padding=0)  
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2, padding=0) # 6x14x14  
        # 6x14x14 -> 16x10x10  
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1, padding=0)  
        # Note: should flatten 16x5x5 in forward()  
        self.fc1 = nn.Linear(16*10*10, out_features=120)  
        self.fc2 = nn.Linear(in_features=120, out_features=84)  
        self.fc3 = nn.Linear(in_features=84, output_size)  
  
    def forward(self, x):  
        x = F.relu(self.conv1(x))  
        x = self.pool1(x)  
        x = F.relu(self.conv2(x))  
        # flatten  
        x = x.view(-1, 16*10*10)  
        x = F.relu(self.fc1(x))  
        x = F.relu(self.fc2(x))  
        # out = F.relu(self.fc3(x))  
        out = F.log_softmax(self.fc3(x), dim=1)  
        return out
```



Observation3：滤波器数量的影响

1. 如下CNN5，增加更多的滤波器
2. 发现此时虽然最后的过拟合现象也很严重，但在测试集上的预测准确率有了明显的提升，最高可以到达69%，比之前的63%有一定的提升；
3. 图像信息有很强的内部结构，而不同图像之间，同个类别的图片背后的内部结构就会是相似的，所以增加滤波通道数 ↔ 增加识别内部结构的分类器改进会相较之下更有效。



```

epoch:10, Average train_loss per batch:0.5730693878173828, train_accuracy:0.79706
epoch:10, Average test_loss per batch:0.9847138034820556, test_accuracy:0.6935
epoch:11, batch:0, loss of one batch:0.32772693037986755
epoch:11, batch:100, loss of one batch:0.4102294147014618
epoch:11, batch:200, loss of one batch:0.4617703258991245
epoch:11, batch:300, loss of one batch:0.5280935168266296
epoch:11, batch:400, loss of one batch:0.6464571952819824
epoch:11, batch:500, loss of one batch:0.6267750859260559
epoch:11, batch:600, loss of one batch:0.7188253998756409
epoch:11, batch:700, loss of one batch:0.546866238117218
epoch:11, Average train_loss per batch:0.5367165937423706, train_accuracy:0.809
epoch:11, Average test_loss per batch:1.0037732414245606, test_accuracy:0.6884

```

Conclusion :

1. 在 CNN 中，在卷积层、池化层、增加滤波通道数 3 个角度的改进上，前二者其实没什么效果，而增加滤波通道数的改进是最明显的，模型的最高准确率可达 69%~70%；
2. 说明图像信息有很强的内部结构，所以增加滤波通道数 \leftrightarrow 增加识别内部结构的分类器改进会最有效。

不同的梯度下降算法

SGD 算法、SGD Momentum 算法和 Adam 算法训练模型，对模型训练速度和性能的影响

- SGD：最简单地随机梯度下降

$$\mathbf{w}_{t+1} = \mathbf{w}_t - lr \cdot \nabla f(\mathbf{w}_t)$$

- SGD + Momentum：不是直接按梯度方向下降，而给梯度加上一个动量

$$\begin{aligned}\mathbf{v}_t &= \rho \mathbf{v}_{t-1} + \nabla f(\mathbf{w}_t) \\ \mathbf{w}_{t+1} &= \mathbf{w}_t - lr \cdot \mathbf{v}_t\end{aligned}$$

其中 $\rho \in (0, 1)$, often 0.9, 0.95, 0.99

- RMSProp：为了防止梯度在一些方向上过于陡峭，而在一些方向上较平缓，引入因子 s_t 放缩到相同的水平

$$\begin{aligned}\mathbf{s}_t &= \rho \cdot \mathbf{s}_{t-1} + (1 - \rho)(\nabla f(\mathbf{w}_t))^2 \\ \mathbf{w}_{t+1} &= \mathbf{w}_t - lr \cdot \nabla f(\mathbf{w}_t) \oslash \sqrt{\mathbf{s}_t}\end{aligned}$$

- Adam：结合 Momentum 和 RMSProp 的思想，既考虑动量，也引入因子 \mathbf{s}_t 放缩，按滑动平均的方式结合过去的梯度与现在的梯度

$$\begin{aligned}\mathbf{m}_t &= \beta_1 \cdot \mathbf{m}_{t-1} + (1 - \beta_1)\nabla f(\mathbf{w}_t) \\ \mathbf{s}_t &= \beta_2 \cdot \mathbf{s}_{t-1} + (1 - \beta_2)(\nabla f(\mathbf{w}_t))^2 \\ \hat{\mathbf{m}}_t &= \frac{\mathbf{m}_t}{1 - \beta_1^t}, \quad \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1 - \beta_2^t} \quad (\text{偏差修正 bias correction}) \\ \mathbf{w}_{t+1} &= \mathbf{w}_t - lr \cdot \hat{\mathbf{m}}_t \oslash \sqrt{\hat{\mathbf{s}}_t}\end{aligned}$$

- 其中，我们一般取 $\beta_1 = 0.9, \beta_2 = 0.999, lr = 10^{-3}$
- bias correction 是为了让在早期迭代中， $\mathbf{m}_t, \mathbf{s}_t$ 的计算更接近实际的 $\nabla f(\mathbf{w}_t), (\nabla f(\mathbf{w}_t))^2$ 值，防止前期的 $\mathbf{m}_t, \mathbf{s}_t$ 因为被初始化为 0 所以滑动平均计算后的值仍较小。
- β_1 是 0.9，不需要 0.999 那么大，是因为 梯度：还是尽量考虑更近的影响，不希望太久远之前的梯度仍在影响；
- β_2 可以有 0.999 那么大，是因为这个放缩因子 \mathbf{s}_t 让更久远一点的数据影响也可以。

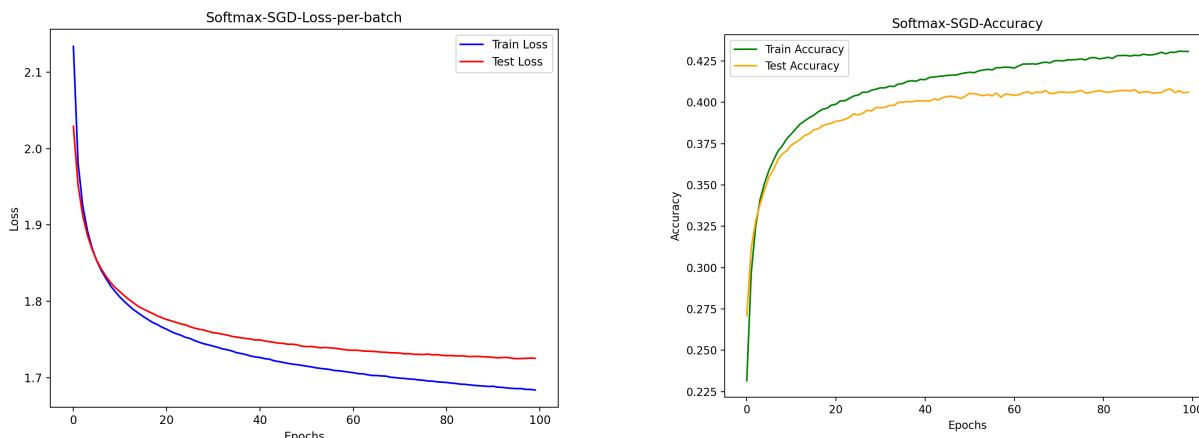
实际实验观察结果如下：

为了方便比较，选择的是 SGD、MLP4、CNN 模型

```
if arguments.optimizer == 'SGD':
    optimizer = optim.SGD(net.parameters(), lr=lr)
elif arguments.optimizer == 'Momentum':
    optimizer = optim.SGD(net.parameters(), lr=lr, momentum=0.9)
elif arguments.optimizer == 'Adam':
    optimizer = optim.Adam(net.parameters()) # default lr=1e-3, betas=(0.9, 0.999)
else:
    raise ValueError("Invalid Optimizer Type")
```

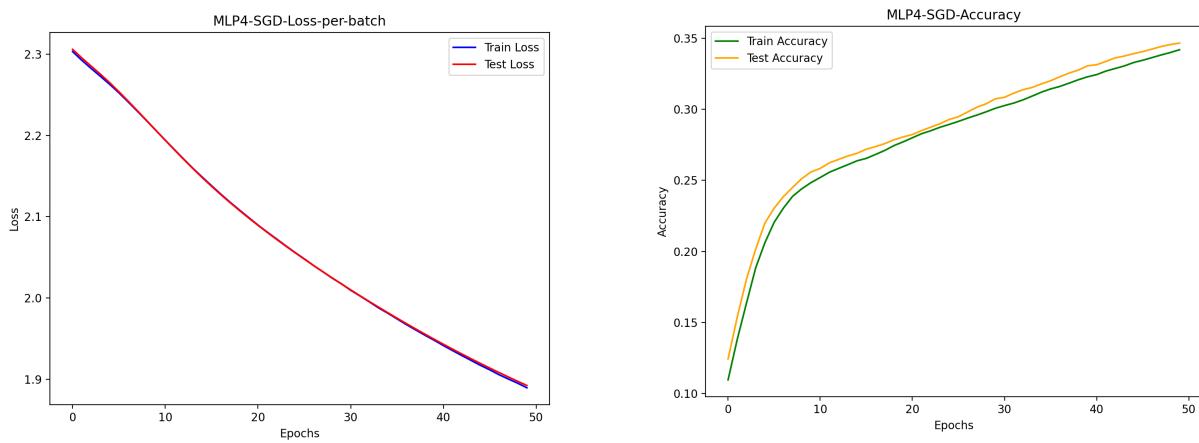
• 使用 SGD 算法进行更新的 3 种模型效果：

1. 在 Softmax 中使用：

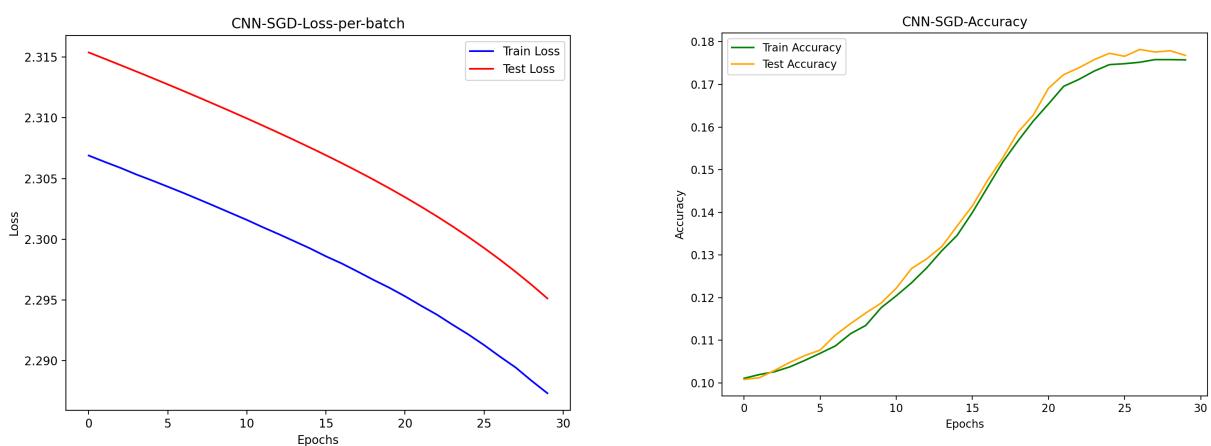


发现 SGD 在 Softmax 上的表现比 Adam 更好。

2. 在 MLP 中使用，效果不好：

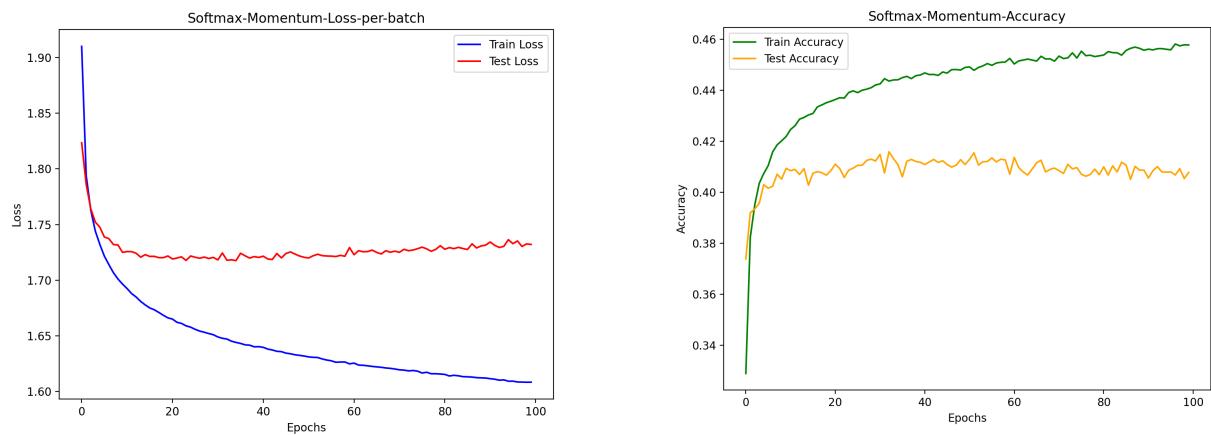


3. 在CNN中使用，效果不好：

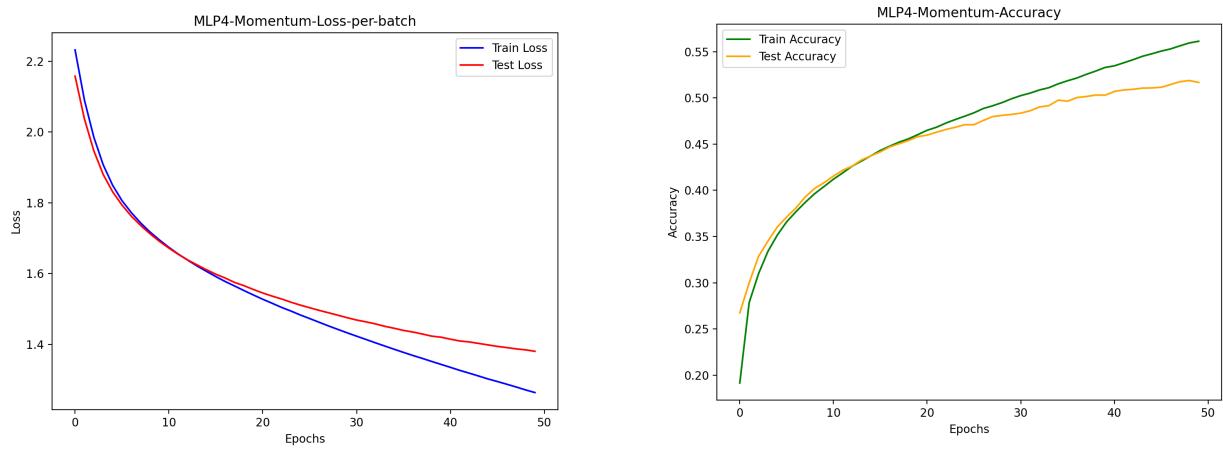


- 使用 SGD+Momentum 算法进行更新的 3 种模型效果：**

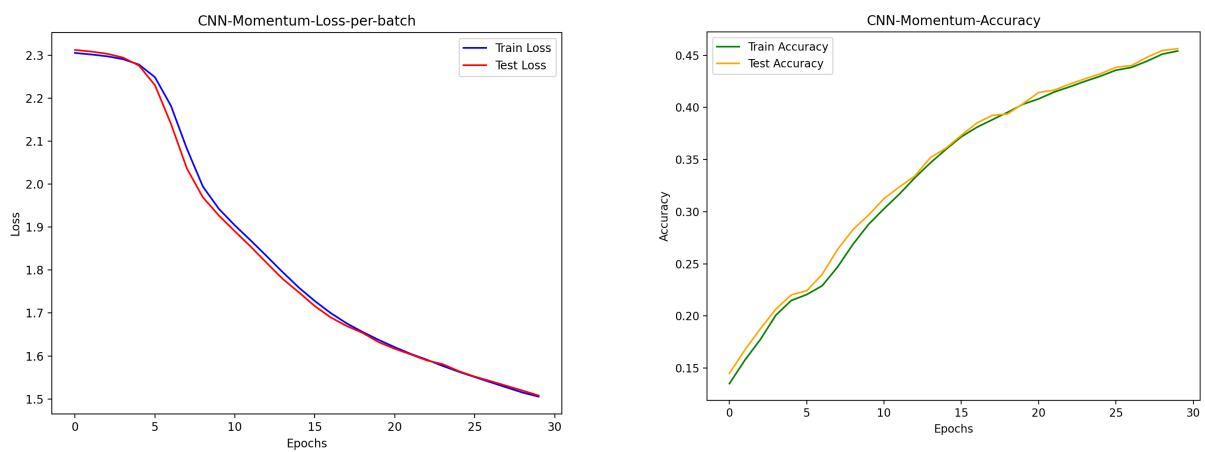
1. 在Softmax中使用：



2. 在 MLP 中使用：

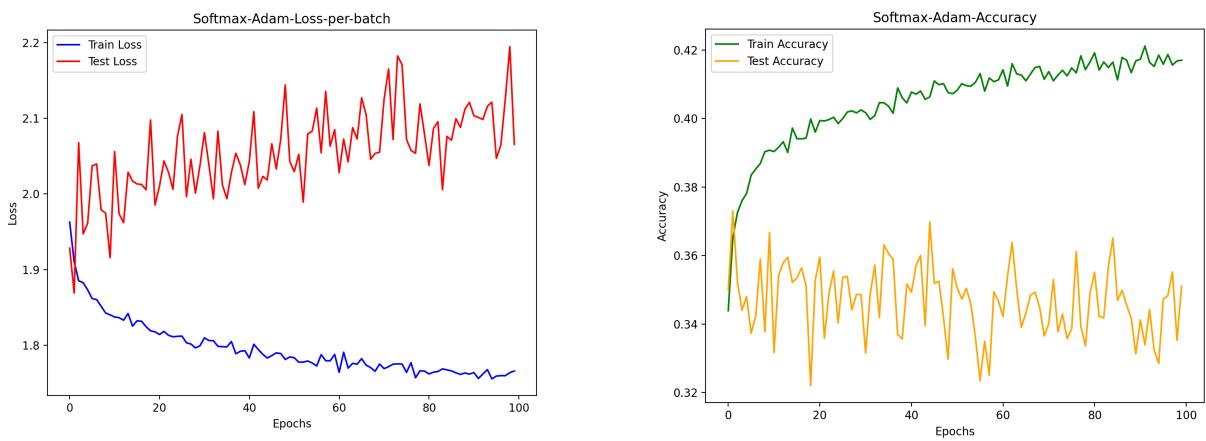


3. 在CNN中使用：

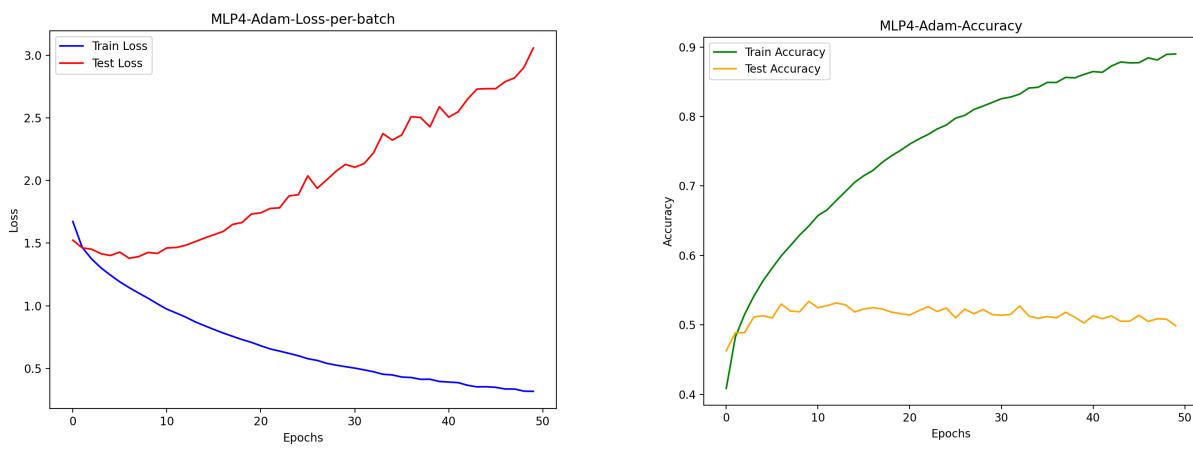


- 使用 Adam 算法进行更新的 3 种模型效果：**

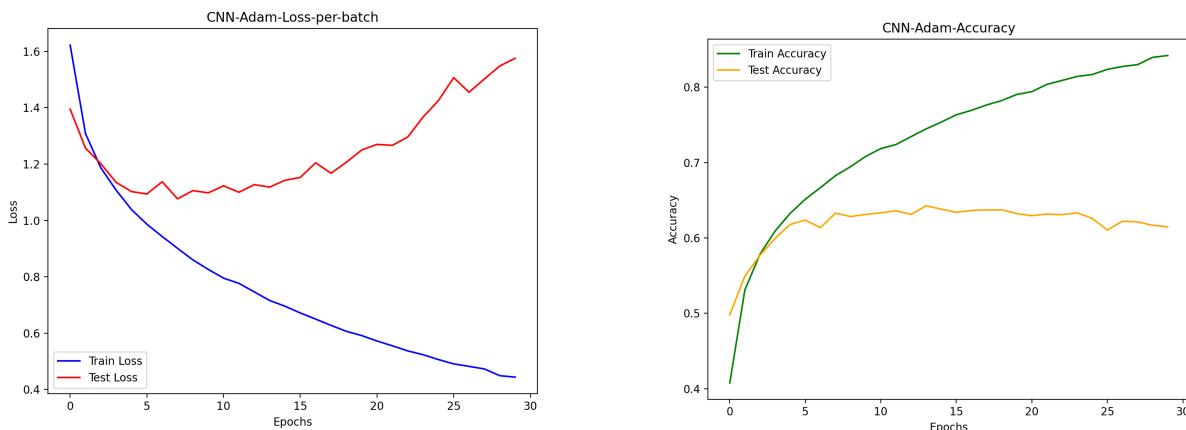
1. 在Softmax中使用：



2. 在 MLP 中使用：



3. 在CNN中使用：



Observation :

1. 首先观察到的是，更换不同的梯度下降算法，训练时间没有太大的差别，Adam会慢一点，因为计算更复杂一点；
2. 在Softmax分类器上，最简单的SGD反而是表现最好的，Adam反而表现得过于不稳定 + 过拟合；
3. 发现Adam易导致明显的严重过拟合；
4. 越深层、复杂的网络结构，比如CNN结构，SGD和Momentum就会逐渐不适合，Adam才有好的效果。

分析原因：

1. **SGD不适合深层网络**：SGD更新较慢，难以在深层网络中有效传播梯度。所以面对MLP和CNN，反而可能导致梯度爆炸、梯度消失；
2. 除此之外，在深层网络模型中，SGD容易陷入局部最小值或鞍点，容易振荡；
3. **简单网络（比如这里的Softmax）不适合用Adam，而适合用简单的SGD**：Softmax由于网络结构并不复杂，所以简单的梯度下降就能达到平稳的更新效果了，Adam有自适应机制（滑动平均等）可能导致过度细致调整而振荡；
4. **Adam容易过拟合**：Adam在每个参数的学习率上都进行动态的调整，那么某些情况下，参数可能会调整得过快或过细，这就可能导致网络过度拟合训练数据集。

5. 复杂网络适合用Adam：越深层、复杂的网络结构，Adam通过使用动量和自适应调整学习率，有助于更快地摆脱局部最小值点、鞍点等问题。

Conclusion :

1. Softmax 用SGD比较合适，用momentum有一定的过拟合，用Adam不仅会过拟合，还会十分不稳定；
2. MLP 适合用 momemtum或Adam， momentum收敛到一个较好的模型会比Adam慢，但过拟合没有Adam那么严重；
3. CNN 用 Adam。
4. SGD适合简单的分类回归结构，此时能有较好的效率和稳定性；
5. 复杂深层网络，要用Adam。

模型训练方法

数据预处理的方法、模型参数初始化方法、超参数选择、优化方法及其它用到的训练技巧

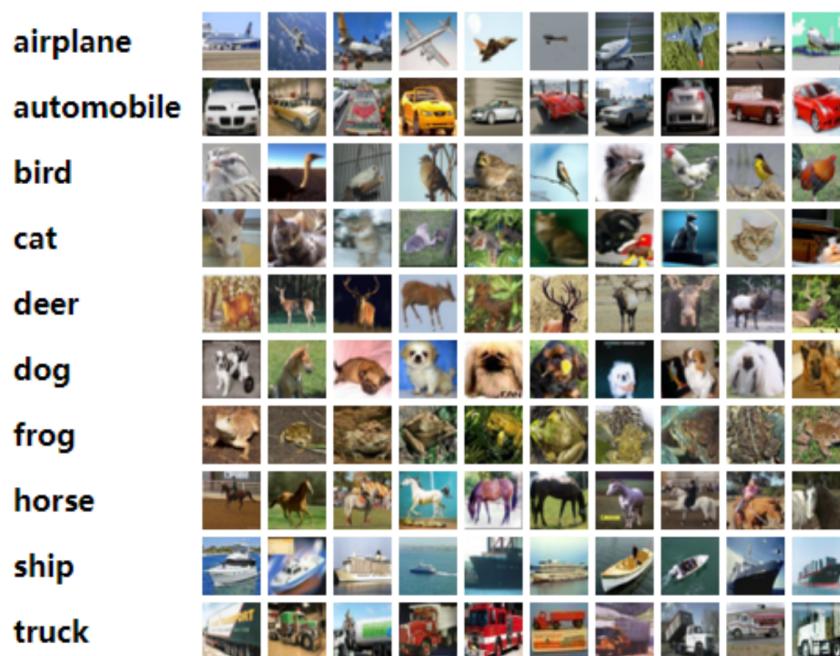
数据预处理

- 关于 CIFAR-10 数据集：

CIFAR-10数据集由60000张32x32的彩色图片组成，包含10类图片，每一类图片均包含6000张图片。采用随机划分的方式，将60000个样本划分成了50000个训练样本和10000个测试样本。

图片为RGB彩色图片，即图片数据有三个颜色通道，所以每一张图片由 $32 \times 32 \times 3$ （长x宽x通道）个像素点组成，在数据集中是将多维矩阵展开存放，即 $60000 \times 32 \times 32 \times 3 \rightarrow 60000 \times 3072$ ，需要同学们自行reshape成想要的形状。

下面是数据的十个类别：



- 读取数据：（实验材料中给出了相应代码，如下）

```

def load_data(dir):
    X_train = []
    Y_train = []
    for i in range(1, 6):
        with open(dir + r'/data_batch_' + str(i), 'rb') as fo:
            dict = pickle.load(fo, encoding='bytes')
            X_train.append(dict[b'data'])
            Y_train += dict[b'labels']
    X_train = np.concatenate(X_train, axis=0)

    with open(dir + r'/test_batch', 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
        X_test = dict[b'data']
        Y_test = dict[b'labels']

    return X_train, Y_train, X_test, Y_test

```

1. 使用 `pickle.load()` 方法，把存储在文件中的字节流（bytes格式）反序列化为 Python 对象，恢复原始数据结构。
2. 在 CIFAR-10数据集中，每个batch文件包含一个字典，该字典有两个键：b'data' 和 b'labels'（CIFAR-10的key名是按bytes格式存储的）
 - 其中，**data 对应的值是一个10000x3072的数组**，每一行（每一个样本）对应的3072个值是：前32x32个是红色通道的值，中间32x32是绿色，最后32x32是蓝色；且是按行优先地存储，也就是，红色通道的32x32中的前32个值，对应的是图片第一行。
 - 然后，**labels 对应的值是一个 list，由10000个在0-9范围内的数组成**，第 i 个就是第 i 个图片的标签。
3. 于是，使用 `X_train.append(dict[b'data'])`，得到的列表有5个元素，每个元素是一个 10000x3072的Numpy数组；使用 `Y_train += (dict[b'labels'])`，得到的列表就直接是 50000 个labels；
4. 所以，还要对 `X_train` 处理一下，合并成50000x3072的Numpy数组。于是使用 `np.concatenate()` 方法，按axis = 0 也就是在最外层的维度上合并。
5. 由此，得到了 `X_train, Y_train, X_test, Y_test`：

`X_train` : 50000×3072 的Numpy数组；
`Y_train` : 长 50000 的list；
`X_test` : 10000×3072 的Numpy数组；
`Y_test` : 长 10000 的list。

- **根据不同的模型进行不同的reshape :**

1. 首先，为了方便后续使用 pytorch 进行模型训练，三种模型都需要先把数据转换成 `torch.tensor` 类型。
2. Softmax 线性分类：直接保留这样被展平的一维向量格式就好。
3. MLP：也是一维向量直接输入输入层。
4. CNN：RGB图像，每一种颜色直接作为一个输入通道，而且恢复 32×32的矩阵格式，也就是把3072的一行重新拆成 32×32×3。

```

def reshape(model, X_train, Y_train, X_test, Y_test):
    """
    :param model: Softmax or CNN or MLP
    :param X_train: Training Samples
    :param Y_train: Training Labels
    :param X_test: Test Samples
    :param Y_test: Test Labels
    :return: Tensor Type, reshaped for specific model
    """

    X_train_tensor = torch.from_numpy(X_train).float()
    Y_train_tensor = torch.tensor(Y_train, dtype=torch.int64)
    X_test_tensor = torch.from_numpy(X_test).float()
    Y_test_tensor = torch.tensor(Y_test, dtype=torch.int64)

    if model in {'CNN', 'CNN2', 'CNN3', 'CNN4', 'CNN5'}:
        X_train_tensor = X_train_tensor.view(-1, 3, 32, 32)
        X_test_tensor = X_test_tensor.view(-1, 3, 32, 32)
    else: # for MLP or Softmax
        X_train_tensor = X_train_tensor.view(-1, 3072)
        X_test_tensor = X_test_tensor.view(-1, 3072)

    return X_train_tensor, Y_train_tensor, X_test_tensor, Y_test_tensor

```

- 另外，像上次实验一样，我还先对数据进行了标准化（standardization）处理。

打包数据

- 在pyTorch里，可以把数据打包好放到 Loader 里，就能方便后续根据 Loader 设置的 batch_size 等参数进行批量数据训练模型

```

# packaging data into TensorDataset for training
train_dataset = TensorDataset(*tensors: X_train_tensor, Y_train_tensor)
test_dataset = TensorDataset(*tensors: X_test_tensor, Y_test_tensor)

# DataLoader for batching and shuffling data
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

```

训练过程

- 首先，要根据我们定义好的网络模型，实例化一个net：

```

if model == 'Softmax':
    net = SoftmaxNN(input_size: 3072, output_size: 10)
elif model == 'MLP':
    net = MLP(input_size: 3072, output_size: 10)
elif model == 'CNN':
    net = CNN(10)
else:
    raise ValueError("Invalid Model Type")

```

- 然后，训练 epochs 轮，在每轮训练中，首先用训练数据集去训练模型，记录损失和准确率，然后，用测试集去评估一下这一轮训练完的模型。
- 训练方法train()如下：

```

def train(net, epoch, train_loader, criterion, optimizer, device):
    net.train()
    training_loss = 0.0
    correct = 0
    total = 0

    # for inputs, labels in train_loader:
    for batch_idx, (inputs, labels) in enumerate(train_loader):
        # inputs: [batch_size, input_size]
        # labels: [batch_size]
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad() # zero the grad
        outputs = net(inputs) # forward
        loss = criterion(outputs, labels) # calculate loss
        loss.backward() # backward, calculate the grads
        optimizer.step() # update model's parameters using the grads

        # now, we have outputs: [batch_size, 10], dim=1: a Tensor with length=10
        max_values, indices = torch.max(outputs, 1)
        predicts = indices
        training_loss += loss.item() # item(): Tensor(loss_value) -> loss_value
        correct_elements = torch.eq(labels, predicts) # get bool Tensor
        correct += torch.sum(correct_elements).item() # use item(): Tensor(num) -> num
        total += labels.size(0)
        if batch_idx % 100 == 0:
            print(f"epoch:{epoch}, batch:{batch_idx}, loss of one batch:{loss.item()}")

    accuracy = correct / total
    return training_loss, accuracy

```

- 评估/测试方法evaluate()如下：

```

def evaluate(net, test_loader, criterion, device):
    net.eval()
    test_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            outputs = net(inputs)
            loss = criterion(outputs, labels)

            test_loss += loss.item()

            max_values, predicts = torch.max(outputs, 1)
            # correct_elements = torch.eq(predicts, labels)
            correct_elements = (predicts == labels)
            correct += torch.sum(correct_elements).item()
            total += labels.size(0)

    accuracy = correct / total
    return test_loss, accuracy

```

- 然后，在每个epoch里使用他们，来观察我们的模型效果：

```

train_loss_list = []
train_accuracy_list = []
test_loss_list = []
test_accuracy_list = []

for epoch in range(epochs):
    # training the model
    train_loss, train_accuracy = train(net, epoch, train_loader, criterion, optimizer, device)
    train_loss = train_loss / (X_train_tensor.size(0) / batch_size)
    print(f"epoch:{epoch}, Average train_loss per batch:{train_loss}, train_accuracy:{train_accuracy}")
    train_loss_list.append(train_loss)
    train_accuracy_list.append(train_accuracy)

    # evaluating the model
    test_loss, test_accuracy = evaluate(net, test_loader, criterion, device)
    test_loss = test_loss / (X_test_tensor.size(0) / batch_size)
    print(f"epoch:{epoch}, Average test_loss per batch:{test_loss}, test_accuracy:{test_accuracy}")
    test_loss_list.append(test_loss)
    test_accuracy_list.append(test_accuracy)

plot(train_loss_list, train_accuracy_list, test_loss_list, test_accuracy_list, model, arguments.optimizer)

```

- 其中，为了loss观察方便，对loss进行了除以 batch的个数 的平均化处理，否则直接输出会形如 train_loss: 1523, test_loss : 322这样，与输入数据量有关，不方便观察。

参数的初始化及选择

- 下面给出的是进行完一系列比较之后认为还不错的参数设置，更多的相关比较在上面已经说明了：

```
python main.py --learning_rate 1e-4 --model "Softmax" --epochs 200 --optimizer "SGD"
```

```
python main.py --model "MLP" --epochs 50 --optimizer "Adam"
```

```
python main.py --model "CNN5" --epochs 30 --optimizer "Adam"
```

优化方法

- 选用更好的梯度下降算法，即Adam，具体见“SGD、SGD+Monmentum、 Adam比较”部分。

在 CIFAR-10 图像分类任务上的性能区别

- 更多的相关讨论在上面的实验中已经说明了，这里给出他们主要的区别与总结。
- Softmax 性能：
 - 训练较快，但准确率不是很好，在测试集上最终的准确率最高只有 40%左右。
 - Softmax+SGD，过拟合不严重，比较稳定；
 - 我认为原因是，Softmax只是线性分类器，也没有深层的网络结构，所以，**没办法提取图像的共同内部结构特征**。
- MLP 性能：
 - 总体而言，好于Softmax，在测试集上最终的准确率可以在 50% 以上；
 - 通过增加神经元数量、网络层数，可以稍微继续地增加测试集上的准确率，但代价是要大大增加训练时长，可见这种做法效率并不高。
 - 5个epoch后，就能逐渐开始收敛到一个较好的模型了，继续训练会过拟合。

4. 可以一定程度地因为自己有多层网络结构的特性，从而提取到一定的图像内部结构。

- CNN 性能

1. 是最适合用来进行图像分类的，在测试集上最终的准确率最高可以到达 69 %；

2. 增加滤波通道数，最有助于性能提升；

3. 有过拟合问题：5个epoch后，就能逐渐开始收敛到一个较好的模型了，继续训练会过拟合。

4. 进行小区域的卷积，确实是提取图像特征的好方法。

实验结果讨论

1. CNN确实效果最好，可见关注图像内部的结构特征是非常重要的，并且，设置多个滤波通道，最能提取更多不同角度的图像结构特征、也就是同类图像共有的内部结构（详见上面CNN实验分析）。

2. 所以，要针对特定的分类任务使用特别的模型，更重要地，要关注分类对象的独有特征。

CNN就是很好地关注到了图像的结构特征，CNN会比MLP效果好，就是因为MLP直接把图像展平了，没有去关注图像上的像素点是小范围内有相关性的，那么CNN进行2D的卷积，就能很好地提取图像特征。

2. 本次实验让我很有收获，除了对CNN为什么在CIFAR-10分类上表现得最好的分析，还有：
在这之前我对 pytorch 应用不多，这次实验让我更加熟悉了pyTorch进行神经网络训练的方法。尤其是在MLP实验中，我发现pyTorch的 CrossEntropyLoss() 方法是包括了Softmax的这一点，以及相关的pyTorch方法探究（详见上面MLP实验分析）。