



实验课程: 机器学习与数据挖掘

实验名称: 神经网络

专业名称: 计算机科学与技术

学生姓名: 吴臻

学生学号: 21307371

- 实验目的
- 实验环境
- 实验要求
- ▼ 实验内容
 - 实验步骤
 - 线性分类器 (Softmax)

- 多层感知机 (MLP)
- 卷积神经网络 (CNN)
- 总结
- 参考资料

实验目的

探索神经网络在图像分类任务上的应用。在给定数据集 CIFAR-10 的训练集上训练模型，并在测试集上验证其性能。

实验环境

1. 使用pytorch框架
2. 由于使用的是轻薄笔记本，所以没有nvidia显卡，无法使用GPU加速

实验要求

1. 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）
2. 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
3. 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
4. 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
5. 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
6. 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

实验内容

实验步骤

1. 数据预处理(数据维度划分、数据归一化)

- i. 首先使用给定的函数 `load_data` 读入数据，并将其转变为tensor，**X_train和X_test的数据类型为torch.float32，以匹配神经网络模型的输入数据类型**，并且将取值范围从[0,255]变为[0,1]
- ii. 将X_train和X_test划分维度并调整维度，便于后续卷积的操作，线性分类器和MLP无需划分
- iii. 创建数据集，它包含了特征和标签
- iv. 创建数据加载器，它可以在训练和测试过程中批量加载数据。对于训练数据，打乱顺序以提高模型的泛化能力。对于测试数据，则不需要打乱顺序。

```
X_train, Y_train, X_test, Y_test = load_data(r'E:\学习资料\机器学习与数据挖掘\lab2\data
```

```
X_train = torch.tensor(X_train / 255.0, dtype=torch.float32) # 将数据类型转换为float32
X_test = torch.tensor(X_test / 255.0, dtype=torch.float32) # 将数据类型转换为float32
Y_train = torch.tensor(Y_train)
Y_test = torch.tensor(Y_test)
```

```
X_train = X_train.reshape((len(X_train), 3, 32, 32))
X_test = X_test.reshape((len(X_test), 3, 32, 32))
```

```
train_set = TensorDataset(X_train, Y_train)
test_set = TensorDataset(X_test, Y_test)
```

```
train_loader = DataLoader(train_set, batch_size=64, shuffle=True)
test_loader = DataLoader(test_set, batch_size=64, shuffle=False)
```

2. 定义模型结构

3. 定义损失函数(交叉熵) 和优化器 (SGD, SGD Momentum, Adam)

```
# 这个函数会对输入的原始预测值进行softmax操作，然后计算真实标签和预测标签之间的交叉熵
criterion = nn.CrossEntropyLoss()

# SGD优化器
optimizer = optim.SGD(model.parameters(), lr=0.001)
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
# Adam优化器
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

分析：这里采用了三种优化器，这三者大致含义如下：

SGD：基本的梯度下降

SGD Momentum：对标准SGD的改进。它引入了动量的概念，通过在参数更新中添加上一次更新的惯性，通俗的讲，每次梯度计算都包含上一轮的梯度

Adam：Adam = Momentum + Adaptive Learning Rate，不仅存储了过去梯度的平方的指数衰减平均值，还像Momentum一样保持了过去提取的指数衰减平均值。优点是学习率自适应修正，不用手动调整

区别：

1. SGD在更新参数时简单直接，但可能收敛速度较慢，尤其是在出现平坦区域或者局部极小值的情况下。
2. SGD Momentum利用动量来加速收敛，减少参数更新的方差，有助于跳出局部极小值，从而提高收敛速度。
3. Adam算法结合了动量方法和自适应学习率，使得每个参数都有自适应的学习率，因此更适用于训练复杂的深度学习模型，并通常能够更快地收敛到较好的解。

4. 训练模型

```

num_epochs = 50
for epoch in range(num_epochs):
    # 将模型设定为训练模式
    model.train()
    running_loss = 0.0
    for inputs, labels in train_loader:
        # 清空之前的梯度，因为PyTorch的默认行为是在反向传播过程中梯度是累加的
        optimizer.zero_grad()
        outputs = model(inputs)
        # 计算模型输出与真实标签之间的损失值
        loss = criterion(outputs, labels)
        # 进行反向传播，计算模型参数的梯度
        loss.backward()
        # 根据参数的梯度更新模型参数
        optimizer.step()
        running_loss += loss.item() * inputs.size(0)
    epoch_loss = running_loss / len(train_loader.dataset)
    loss_values.append(epoch_loss)
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {epoch_loss:.4f}')
    # 每轮训练都进行测试
    predict()

```

5. 测试模型

```
def predict():
    # 将模型设置为评估模式
    model.eval()
    correct = 0
    total = 0
    # 在评估过程中不计算梯度
    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs)
            # 找到每个样本预测输出中的最大值以及对应的索引，即预测的类别标签
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy_values.append((100 * correct / total))
    print(f'Accuracy of the network on the test images: {100 * correct / total}%')
```

线性分类器 (Softmax)

由于使用pytorch，实现线性分类器 (Softmax) 相当于构建一个单层的全连接神经网络，激活函数采用softmax函数，损失函数采用交叉熵函数 (CrossEntropyLoss)。

Softmax分类器类似逻辑回归，只是不是二分类，而是多分类，虽然引入softmax函数，线性分类器的线性体现在于它的决策边界是线性的，只能处理线性数据

softmax函数如下：（样本向量 \mathbf{x} 属于第 j 个分类的概率为）

$$P(y = j|\mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_j}}{\sum_{k=1}^K e^{\mathbf{x}^\top \mathbf{w}_k}}$$

- 定义线性模型结构

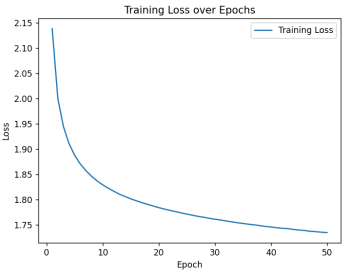
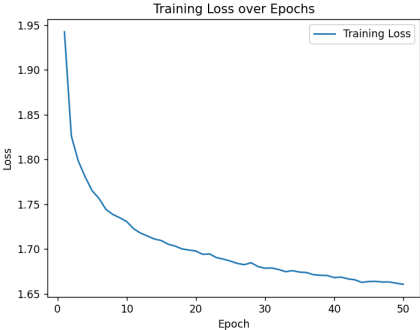
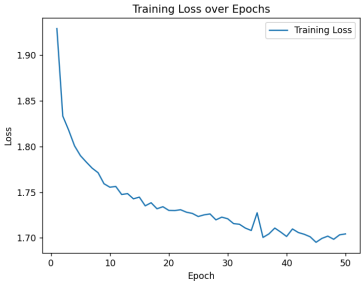
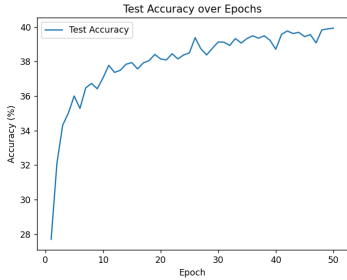
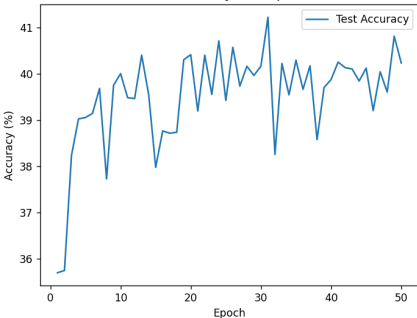
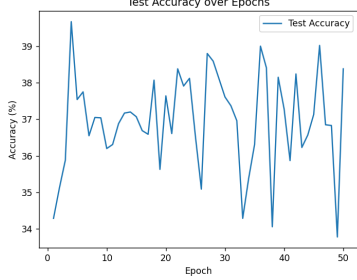
```

class LinearClassifier(nn.Module):
    def __init__(self):
        super(LinearClassifier, self).__init__()
        # 定义一个全连接层
        # 输入的特征维度是32*32*3（对应于32x32的RGB图像），输出的特征维度是10（对应于10个类别）
        self.linear = nn.Linear(32*32*3, 10)

    # 定义前向传播函数
    def forward(self, x):
        # 将输入x的形状变为(batch_size, -1)，其中-1表示自动计算剩余的维度（即32*32*3）
        x = x.view(x.size(0), -1)
        # 将变形后的x传递给全连接层，并得到输出
        x = self.linear(x)
        return x

```

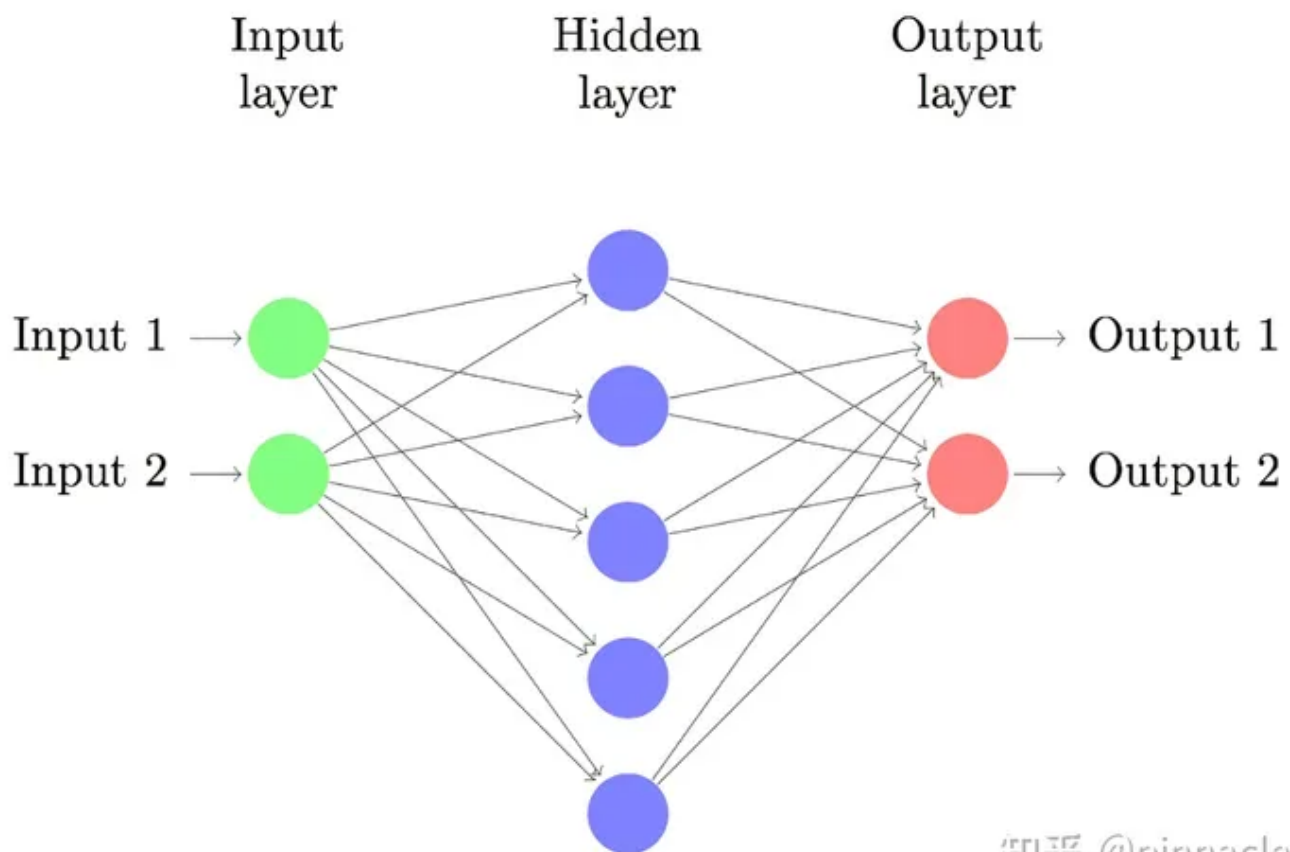
结果展示：

SGD	SGD Momentum	Adam
		
		
<div>Accuracy of the network on the test images: 39.89%</div> <div>Epoch [50/50], Loss: 1.7350</div> <div>Accuracy of the network on the test images: 39.93%</div> <div>It takes 176.152113199234 s</div>	<div>Epoch [50/50], Loss: 1.6607</div> <div>Accuracy of the network on the test images: 40.24%</div> <div>It takes 210.60122632980347 s</div>	<div>Epoch [50/50], Loss: 1.7044</div> <div>Accuracy of the network on the test images: 38.38%</div> <div>It takes 198.99487352371216 s</div>

分析：从上面表格可以看到对于该测试集，表现最好的是SGD Momentum，后面基本维持在40%左右，SGD的收敛速度最快（训练时间也最短），SGD Momentum稍差一点（这个我也不太理解，理论上应该是SGD Momentum收敛速度最快），至于Adam，波动比较大，Adam逃离鞍点很快，但是在寻找局部最优这方面比较差

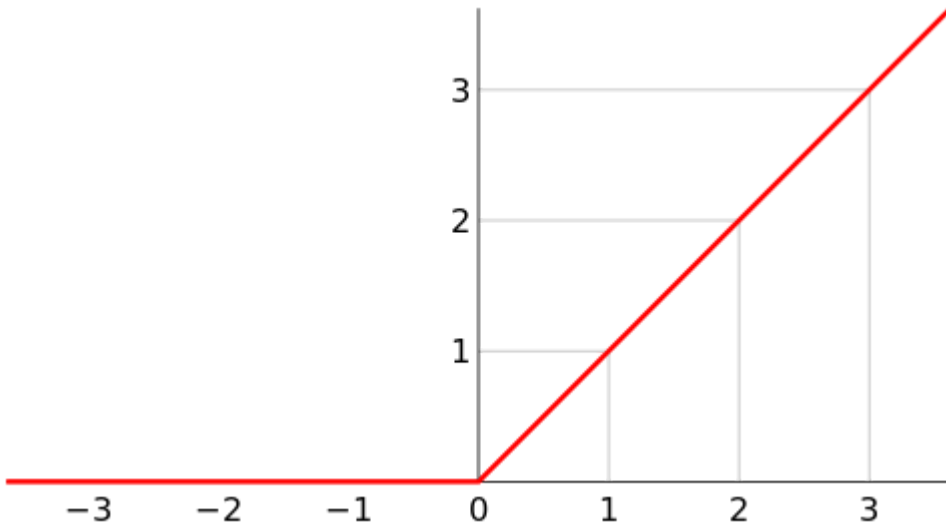
多层感知机（MLP）

多层感知机（英语：Multilayer Perceptron，缩写：MLP）是一种前向结构的人工神经网络，映射一组输入向量到一组输出向量。MLP可以被看作是一个有向图，由多个的节点层所组成，每一层都全连接到下一层。除了输入节点，每个节点都是一个带有非线性激活函数的神经元（或称处理单元），多层感知机的基本结构由三层组成：输入层，中间隐藏层和最后输出层



知乎 @pinnacle

ReLu函数：

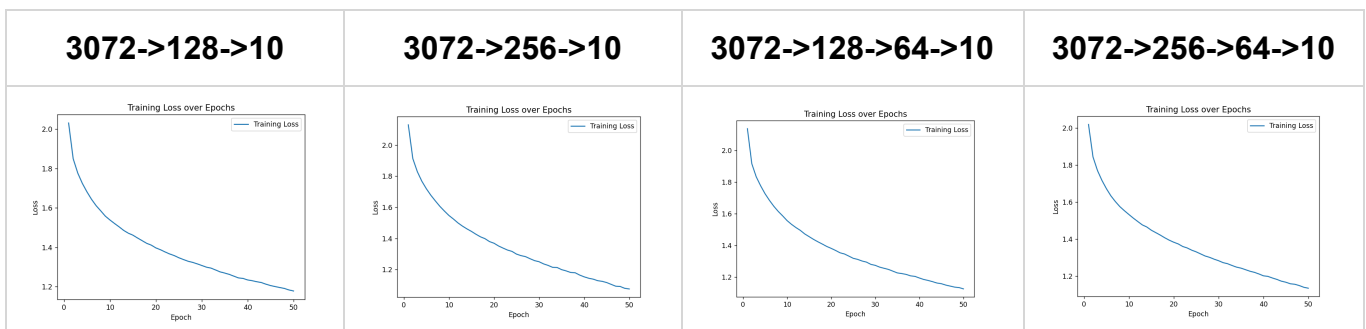


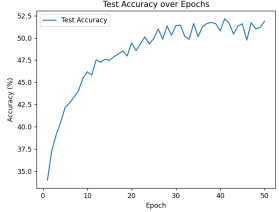
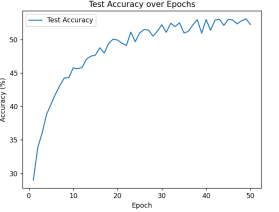
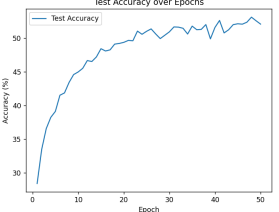
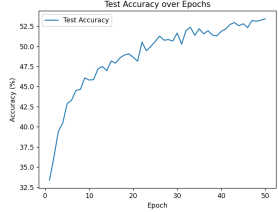
- 构建多层感知机 (MLP) 模型

```
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(32*32*3, 128) # 输入层到第一个隐藏层
        self.fc2 = nn.Linear(128, 64) # 第一个隐藏层到第二个隐藏层
        self.fc3 = nn.Linear(64, 10) # 第二个隐藏层到输出层

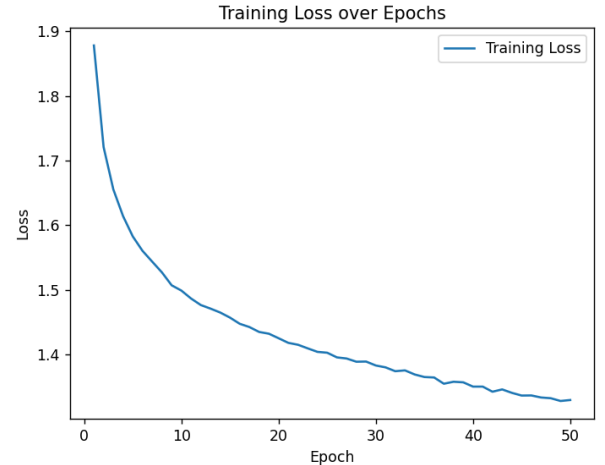
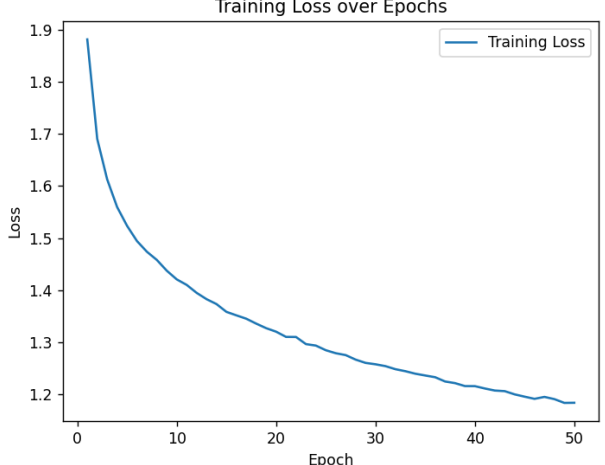
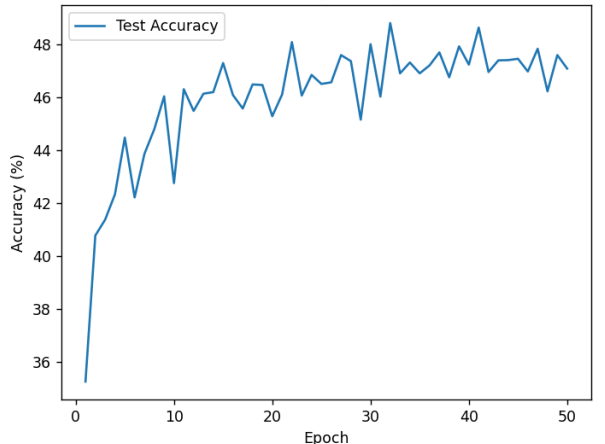
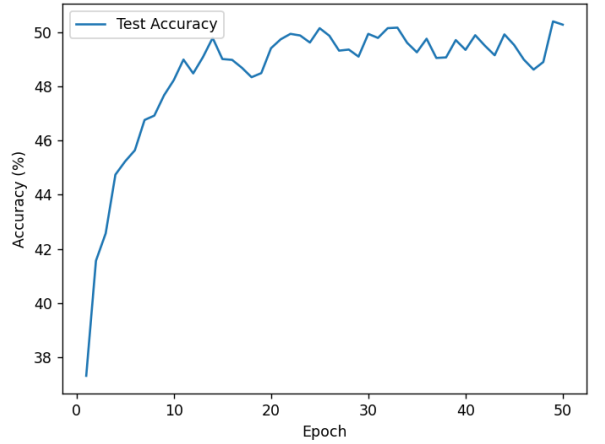
    def forward(self, x):
        x = x.view(-1, 32*32*3) # 将输入数据展开成一维向量
        x = F.relu(self.fc1(x)) # 第一个隐藏层 (使用ReLU激活函数)
        x = F.relu(self.fc2(x)) # 第二个隐藏层 (使用ReLU激活函数)
        x = self.fc3(x) # 输出层
        return x
```

SGD Momentum



3072->128->10	3072->256->10	3072->128->64->10	3072->256->64->10
 <pre> Accuracy of the network on the test images: 51.17% Epoch [50/50], Loss: 1.1786 Accuracy of the network on the test images: 51.89% It takes 1744.8945796489716 s </pre>	 <pre> Epoch [50/50], Loss: 1.0748 Accuracy of the network on the test images: 52.27% The max accuracy of the network on the test images: 53.11% It takes 3481.7445487976874 s </pre>	 <pre> Accuracy of the network on the test images: 53.1% Epoch [49/50], Loss: 1.1339 Accuracy of the network on the test images: 52.57% Epoch [50/50], Loss: 1.1257 Accuracy of the network on the test images: 52.07% It takes 1995.6285259723663 s </pre>	 <pre> Accuracy of the network on the test images: 53.24% Epoch [50/50], Loss: 1.1359 Accuracy of the network on the test images: 53.41% The max accuracy of the network on the test images: 53.41% It takes 3535.8232414722443 s </pre>

Adam

3072->128->10	3072->128->64->10
	
	

3072->128->10	3072->128->64->10
<pre>Epoch [50/50], Loss: 1.3294 Accuracy of the network on the test images: 47.1% The max accuracy of the network on the test images: 48.82% It takes 1930.6971554756165 s</pre>	<pre>Epoch [50/50], Loss: 1.1837 Accuracy of the network on the test images: 50.28% The max accuracy of the network on the test images: 50.4% It takes 2089.5937542915344 s</pre>

分析：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

网络层数

- 1.增加网络层数可以增强模型的表征能力，使其能够更好地学习复杂的非线性关系和特征，增加网络层数可能导致模型更容易过拟合训练数据，而较浅的网络则可能面临欠拟合问题
- 2.本次实验共尝试了两种网络层数，分别是两层和三层，增加层数之后，准确率有小部分提升，运行时间有所增加，不过增加不明显

神经元数量

- 1.增加神经元数量可以增强模型的表征能力，使其能够更好地拟合复杂的函数关系，增加神经元数量会增加模型的计算复杂度，导致训练时间和资源消耗增加
- 2.本次实验设计了四种神经网络结构，当只有两层时，即3072->128->10和3072->256->10，第一个隐藏层从128设置为256时，原数据的更多信息被保留，准确率提升，但是因为要训练的参数几乎翻了一倍，所以训练时间也翻倍了；而3072->128->64->10和3072->256->64->10相比，准确率提升不明显

综合来看，神经网络机构3072->128->64->10比较理想，综合了准确率和运行时间的优势

分析：由于MLP比较复杂，只采用了两种算法SGD Momentum和Adam进行训练，单从实验结果看，SGD Momentum在准确率和运行时间上的表现都比Adam好

卷积神经网络（CNN）

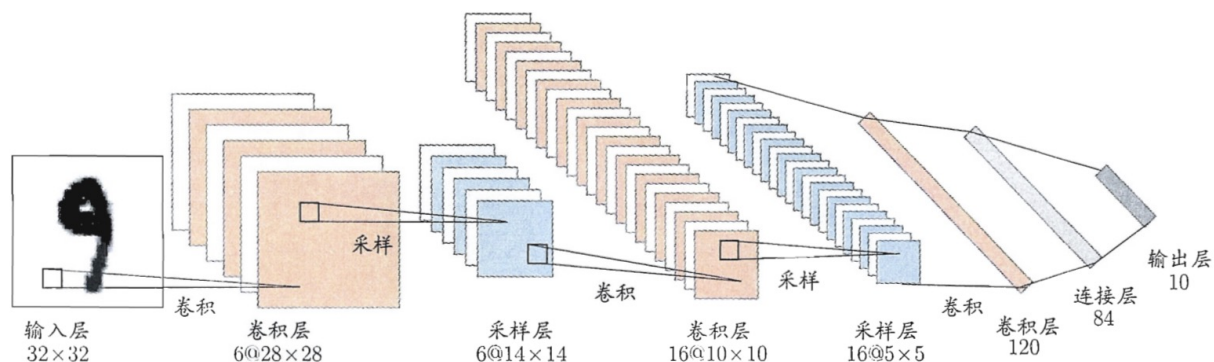
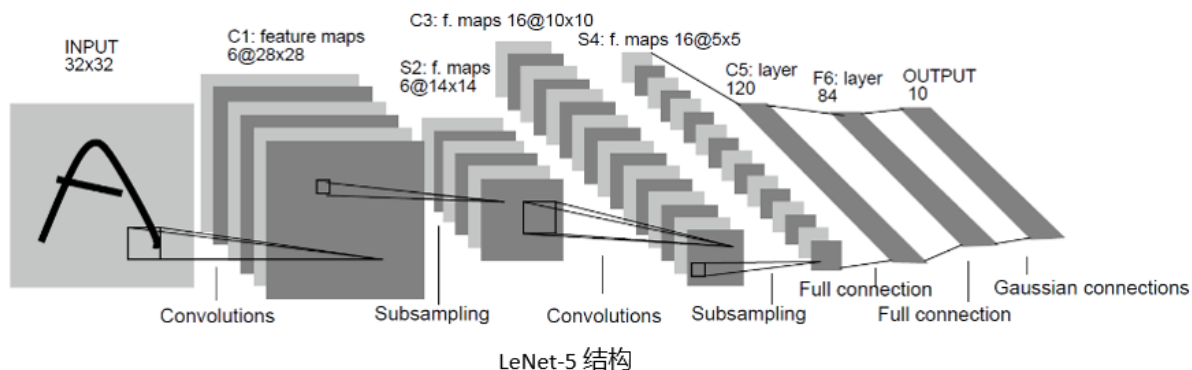
卷积神经网络由一个或多个卷积层和顶端的全连通层（对应经典的神经网络）组成，同时也包括关联权重和池化层（pooling layer）。

卷积核是一种特征发现器，我们通过卷积层可以很容易地发现图像中的各种边缘。但是卷积层发现的特征往往过于精确，我们即使高速连拍拍摄一个物体，照片中的物体的边缘像素位置也不大可能完全一致，通过**池化层**我们可以降低卷积层对边缘的敏感性。

全连接层

最后，在经过几个卷积和最大池化层之后，神经网络中的高级推理通过完全连接层来完成。就和常规的非卷积人工神经网络中一样，完全连接层中的神经元与前一层中的所有激活都有联系。

LeNet网络结构



LeNet结构包括卷积层、池化层和全连接层，具有以下主要特点：

卷积层和池化层交替： LeNet首次将卷积层和池化层结合起来，并采用了交替的方式堆叠，这种结构使得网络能够有效地提取图像特征。

激活函数： LeNet中使用的激活函数是sigmoid函数，这是早期深度学习时代常用的激活函数之一。

全连接层： LeNet在卷积和池化层之后使用了全连接层，最终输出分类结果。

第一张照片最后有三个全连接层，第二张用一个卷积层和两个全连接层代替，本次实验采用第二张图片的结构

- **结构一 (LeNet)**

Conv -> Relu -> MaxPool -> Conv -> Relu -> MaxPool -> Conv -> Relu -> Linear -> Relu -> Linear

```
class LeNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        # 定义卷积层和全连接层
        self.conv1 = nn.Conv2d(3, 6, kernel_size=5, stride=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5, stride=1)
        self.conv3 = nn.Conv2d(16, 120, kernel_size=5, stride=1)
        self.fc1 = nn.Linear(120, 84)
        self.fc2 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = torch.relu(self.conv3(x))
        x = x.view(-1, 120)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

- **结构二**

Conv -> Relu -> MaxPool -> Conv -> Relu -> MaxPool -> Linear

```

class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        # 定义卷积层和全连接层
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.fc = nn.Linear(32 * 8 * 8, 10)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 32 * 8 * 8) # 展开成一维向量
        x = self.fc(x)
        return x

```

- **结构三 (参考网上已有网络结构, 参考资料有具体网站)**

Conv -> BatchNorm -> ReLU -> Conv -> BatchNorm -> ReLU -> MaxPool -> Conv -> BatchNorm -> ReLU -> Conv -> BatchNorm -> ReLU -> Linear

- BatchNorm2d 层对输入应用规范化以获得零均值和单位方差并提高网络精度。

```

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

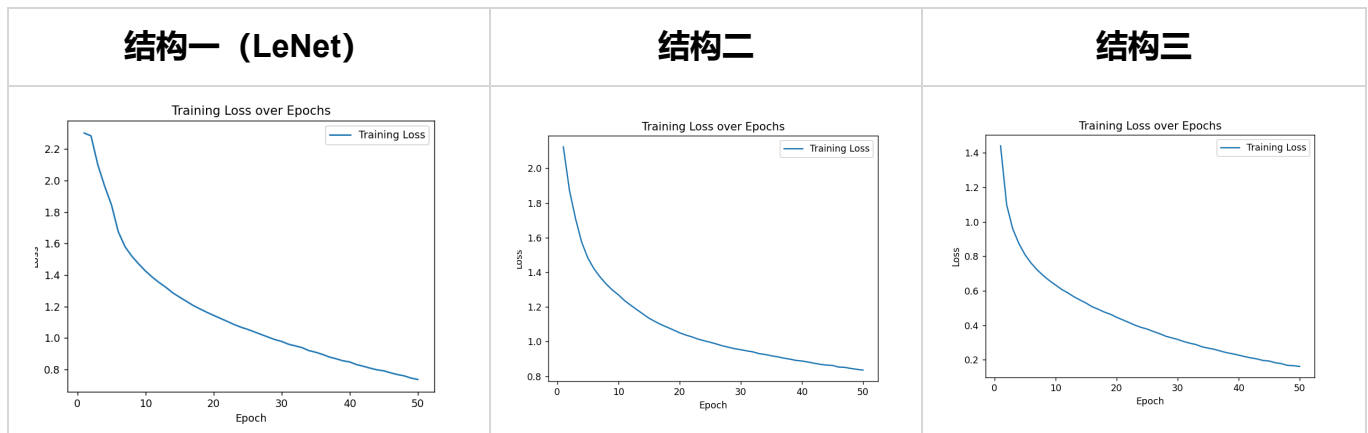
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=12, kernel_size=5, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(12)
        self.conv2 = nn.Conv2d(in_channels=12, out_channels=12, kernel_size=5, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(12)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv4 = nn.Conv2d(in_channels=12, out_channels=24, kernel_size=5, stride=1, padding=1)
        self.bn4 = nn.BatchNorm2d(24)
        self.conv5 = nn.Conv2d(in_channels=24, out_channels=24, kernel_size=5, stride=1, padding=1)
        self.bn5 = nn.BatchNorm2d(24)
        self.fc1 = nn.Linear(24 * 10 * 10, 10)

    def forward(self, input):
        output = F.relu(self.bn1(self.conv1(input)))
        output = F.relu(self.bn2(self.conv2(output)))
        output = self.pool(output)
        output = F.relu(self.bn4(self.conv4(output)))
        output = F.relu(self.bn5(self.conv5(output)))
        output = output.view(-1, 24 * 10 * 10)
        output = self.fc1(output)

    return output

```

SGD Momentum

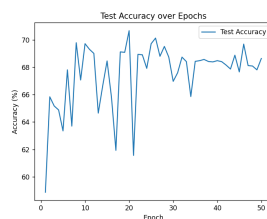
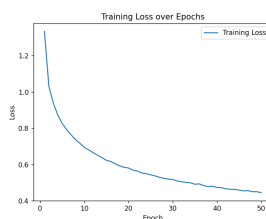


结构一 (LeNet)	结构二	结构三
<pre>Epoch [50/50], Loss: 0.7374 Accuracy of the network on the test images: 62.36% The max accuracy of the network on the test images: 63.2% It takes 411.7941817150879 s</pre>	<pre>Epoch [50/50], Loss: 0.8359 Accuracy of the network on the test images: 65.81% The max accuracy of the network on the test images: 66.22% It takes 934.1491838799286 s</pre>	<pre>Epoch [50/50], Loss: 0.1617 Accuracy of the network on the test images: 69.31% The max accuracy of the network on the test images: 72.24% It takes 1896.3592150211334 s</pre>

Adam

结构一 (LeNet)	结构二	结构三
<pre>Epoch [50/50], Loss: 0.6248 Accuracy of the network on the test images: 62.16% The max accuracy of the network on the test images: 63.46% It takes 486.7385537624359 s</pre>	<pre>Epoch [50/50], Loss: 0.5544 Accuracy of the network on the test images: 68.55% The max accuracy of the network on the test images: 69.25% It takes 938.6080770492554 s</pre>	<pre>Epoch [50/50], Loss: 0.0937 Accuracy of the network on the test images: 69.14% The max accuracy of the network on the test images: 73.4% It takes 1857.9851262569427 s</pre>

结构二 (使用BatchNorm2d) Adam



```
Epoch (50/50), Loss: 0.4444
Accuracy of the network on the test images: 68.66%
The max accuracy of the network on the test images: 70.67%
It takes 1227.4023864269257 s
```

分析：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

1.本次CNN实验采用了三种网络结构，分别是两个、三个和四个卷积层，三个卷积层的网络结构（LeNet）准确率最低，不过运行最快，结构二（两个卷积层）的准确率比预想的好，经过分析，我认为应该是卷积核大小设置得比较小（3*3），使得原图像在卷积的过程保留了比较多的特征

2.结构三是在网上搜索到的，它采用了比较复杂的网络结构，四个卷积层，还使用了BatchNorm2d 层对输入应用规范化(后面也用了BatchNorm2d 层对结构二进行改进)，并且前面的两个结构都是卷积层后跟着一个maxpool层，而结构三虽然有四个卷积层，但是只设置了一个maxpool层，maxpool层可以提取显著的特征，极大地减少需要训练的参数，但同时不可避免的减少了特征。结构三虽然耗时远远大于结构一和二，但是准确率也达到了73%，由于结构三比较复杂，训练到十代左右就达到最大值，后面的训练就有些过拟合了

3.CNN实验也只采用了SGD Momentum 算法和 Adam 算法，在CNN中，Adam 算法可以说优于SGD Momentum 算法，两者耗时差不多，但是使用了Adam 算法之后，准确率有小部分提升

总结

本次实验共实现了线性分类器、MLP 和CNN 模型三个模型。

1.线性分类器结构最简单，它只包括一个线性变换（全连接层）和一个 softmax 函数用于分类，无法有效地捕获图像中的复杂特征，训练时间最短，不过准确率也最低，最多只到40%

2.MLP包括多个全连接层，每个隐藏层都使用非线性激活函数（ReLU）来引入非线性特征变换。MLP可以学习到更丰富和复杂的特征，相比于线性分类器有更好的性能，准确率最高达到了53%，但是它忽略了图像的空间结构，对于像素之间的相关性并没有进行利用，并且由于是全连接，引入了大量参数，所以耗时最长

3.CNN包括卷积层、池化层和全连接层等部分，能够有效地捕获图像中的局部模式和空间结构。卷积层能够学习到图像中的特征，池化层可以提取显著特征，减少训练参数，全连接层负责神经网络中的高级推理。CNN表现最佳，准确率最高达到73%，并且耗时介于线性分类器和MLP之间

4.本次实验共使用了 SGD 算法、SGD Momentum 算法和 Adam 算法训练模型，SGD Momentum 算法在线性分类器、MLP表现最佳，Adam 算法在CNN表现最佳

参考资料

softmax: <https://zhuanlan.zhihu.com/p/105722023>

MLP:<https://zh.wikipedia.org/wiki/多层感知器#>

CNN:

1. <https://learn.microsoft.com/zh-cn/windows/ai/windows-ml/tutorials/pytorch-train-model>
2. <https://zh.wikipedia.org/wiki/卷积神经网络>

LeNet: https://blog.csdn.net/qq_55433305/article/details/127599789

Adam: <https://www.zhihu.com/question/323747423>