

机器学习 实验报告

学号

22336180

姓名

马岱

一、实验环境

- OS: Windows 11
- IDE: Visual Studio Code
- programming language: python

二、实验题目

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。数据下载FTP地址：<ftp://172.18.167.164/Assignment2/material>（建议使用FTP客户端链接，用户名与密码均为student）

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

实验报告需包含（但不限于）：

- 1) 采用的模型结构和训练方法（包括数据预处理的方法、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）
- 2) 实验结果，及对观察结果的充分讨论将实验报告（.doc或.pdf）和代码（不要数据）打包成一个文件，文件包的命名规则为：学号+姓名.tar或.zip，并上传到课程FTP：
<ftp://172.18.167.164/Assignment2/>

三、实验内容

1. 模型理论知识&算法原理

【线性分类器（softmax分类器）】

1.1 线性二分类

在上次的实验中我们通过对线性二分类做了一定的讲解，并且基于线性二分类的理论基础实现了SVM模型。线性二分类问题是指将数据划分为两个类别。给定一个输入向量 \mathbf{x} ，任务是找到一个线性分类器来预测其类别（通常标记为0或1）。分类的决策边界是一个线性超平面。

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$$

其中：

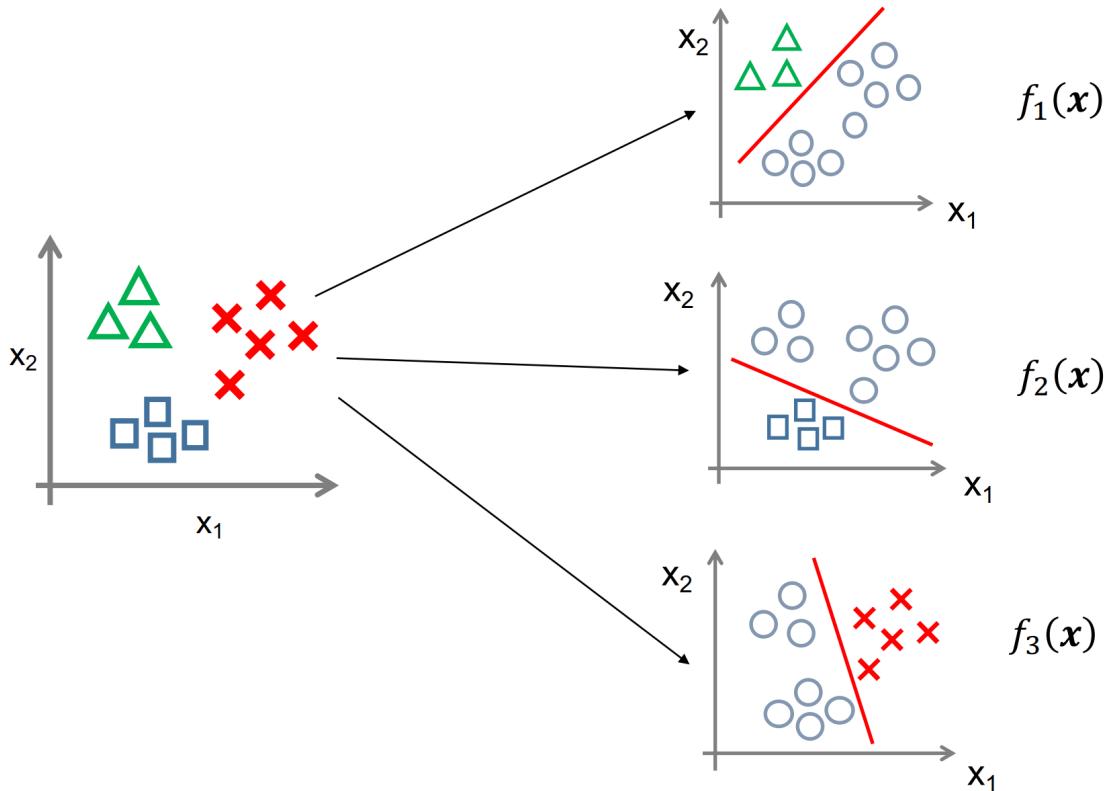
- \mathbf{x} 是输入向量。
- \mathbf{w} 是权重向量，定义了决策超平面的方向。
- b 是偏置项，定义了决策超平面的平移。

$$\text{预测类别} = \begin{cases} 1, & \text{如果 } f(\mathbf{x}) \geq 0 \\ 0, & \text{否则} \end{cases}$$

由此我们使用Sigmoid函数 $\sigma(z) = \frac{1}{1+e^{-z}}$ 将线性分类器的输出转化为概率值，然后通过阈值（如0.5）进行二分类决策。

1.2 线性多分类

对于线性多分类问题，我们最初使用的方法是将K分类问题转换为K个二分类问题，要预测一个新的样本 \mathbf{x} 的类别，最常用的方法之一是选择使得某个得分函数最大的类别。



$$k = \arg \max_j f_j(\mathbf{x})$$

很明显上述方法效率较低，由此我们引入了softmax函数。在数学，特别是概率论和机器学习领域中，**Softmax函数**（也称归一化指数函数）是逻辑斯谛函数（Logistic function）的一种推广。它可以将一个任意实数的 K 维向量 \mathbf{z} ，压缩成一个 K 维的实数向量 $\sigma(\mathbf{z})$ ，使得每个元素都在(0, 1)之间，且所有元素的和为1。可以理解为该向量表示一个概率分布。

Softmax函数通常定义如下：

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \quad \text{for } j = 1, \dots, K$$

其中：

- $\mathbf{z} = [z_1, z_2, \dots, z_K]$ 是输入的 K 维向量。
- $\sigma(\mathbf{z})_j$ 是Softmax输出向量 $\sigma(\mathbf{z})$ 的第 j 个元素。

在多项逻辑回归和线性判别分析中，Softmax函数的输入通常是从 K 个不同线性函数得到的结果。样本向量 \mathbf{x} 属于第 j 个类别的概率为：

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

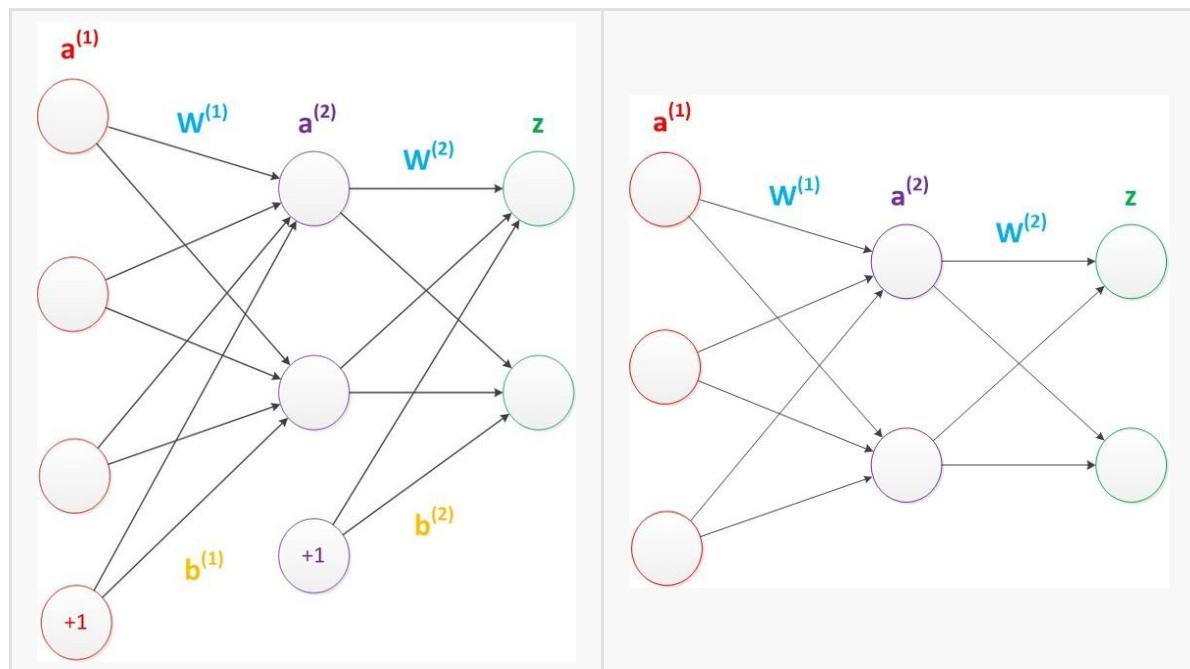
其中：

- \mathbf{x} 是输入样本向量。
- \mathbf{w}_j 是类别 j 对应的参数向量。
- K 是类别的总数。

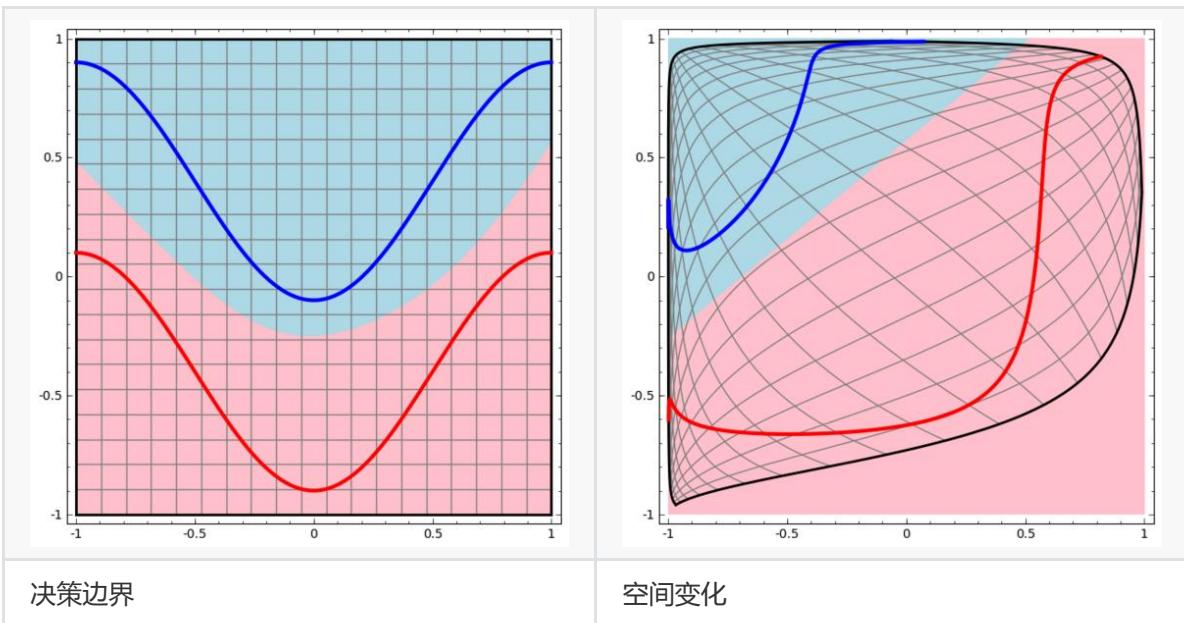
实际上Softmax 分类器可以看作是一个最简单的 MLP，即单层感知机。它只有一个输入层和一个输出层，没有隐藏层。换句话说，Softmax 分类器相当于一个 单层神经网络。MLP 则是在这个基础上加入了隐藏层。因此，当将 MLP 的隐藏层数量设置为 0 时，MLP 就等价于一个 Softmax 分类器。

【多层感知机（MLP）】

单层神经网络无法解决异或问题。但是当增加一个计算层以后，两层神经网络不仅可以解决异或问题，而且具有非常好的非线性分类效果。一个经典的神经网络包含三个层次：红色的是 输入层，绿色的是 输出层，紫色的是 中间层（也叫 隐藏层）- 输入层与输出层的节点数往往是固定的，中间层则可以自由指定- 结构图里的关键不是圆圈（代表“神经元”），而是连接线。每个连接线对应一个不同的权重。

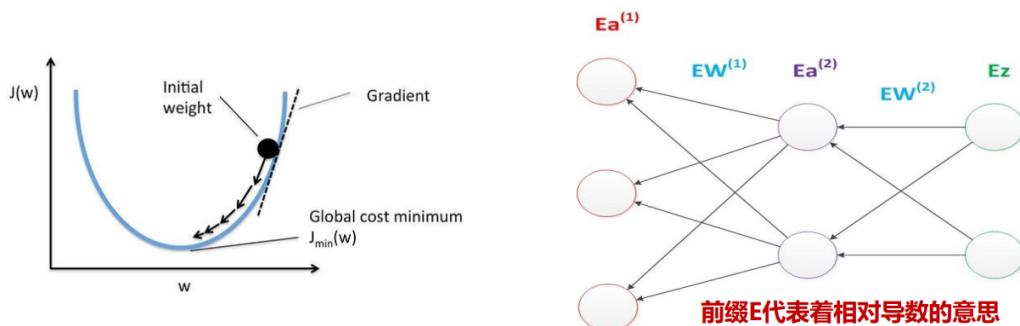


我们可以看到输出层的决策分界仍然是直线。关键是，从输入层到隐藏层时，数据发生了空间变换。也就是说，两层神经网络中，隐藏层对原始的数据进行了一个空间变换，使其可以被线性分类，然后输出层的决策分界划出了一个线性分类分界线，对其进行分类。



多层感知器 - 训练

- 梯度下降 算法：每次计算参数在当前的梯度，然后让参数向着梯度的反方向前进一段距离，不断重复，直到梯度接近零时截止。一般这个时候，所有的参数恰好达到使损失函数达到一个最低值的状态。
- 反向传播 算法：反向传播算法不一次计算所有参数的梯度，而是从后往前。首先计算输出层的梯度，然后是第二个参数矩阵的梯度，接着是中间层的梯度，再然后是第一个参数矩阵的梯度，最后是输入层的梯度。计算结束以后，所要的两个参数矩阵的梯度就都有了。



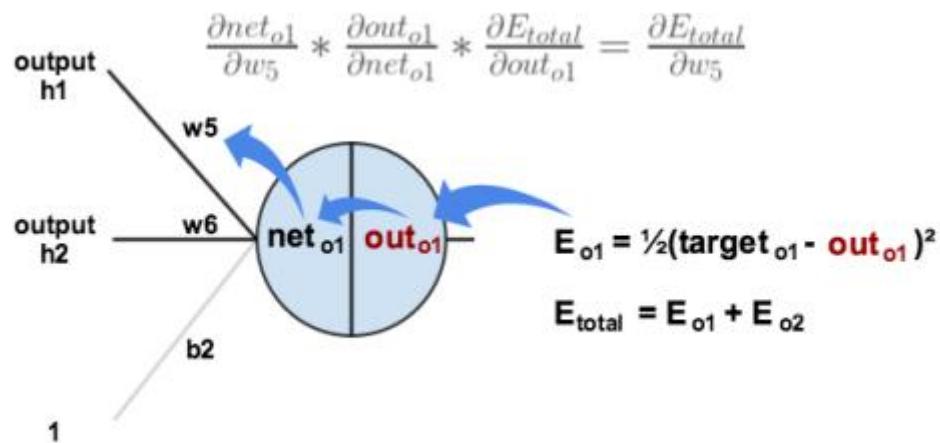
在实际训练过程中，需要先进行正向传播训练，再进行反向传播训练

- 正向传播 (Forward Propagation)
正向传播是从输入数据通过网络层层传递并计算输出的过程。输入数据作为特征向量，进入网络。对于隐藏层，每一层都有权重矩阵 W 和偏置向量 b ；对于每一个神经元，其输入信号是前一层输出的加权和，加上偏置；每一层的输出可以表示为：

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

其中， l 表示当前层数， $a^{(l-1)}$ 是上一层的激活输出， $z^{(l)}$ 是当前层的线性组合输出。然后，对 $z^{(l)}$ 进行非线性激活函数的处理，得到当前层的输出： $a^{(l)} = \sigma(z^{(l)})$ ，其中， σ 是激活函数，如 Sigmoid、ReLU、Tanh 等；最后一层通常会有一个特定的激活函数（如 Softmax）来生成最终的预测结果。通过正向传播，可以得到网络的输出（预测值）。

- 反向传播



反向传播是用于更新权重和偏置的关键算法。目标是通过计算损失函数的梯度，调整网络参数，使损失函数值减小。步骤如下：从输出层开始，计算损失对每一层参数（权重和偏置）的梯度，通过链式法则，将梯度从输出层传递回隐藏层和输入层，对于输出层的梯度：

$$\delta^{(L)} = \nabla_a \mathcal{L} \odot \sigma'(z^{(L)})$$

其中， $\delta^{(L)}$ 是输出层的误差， $\nabla_a \mathcal{L}$ 是损失对输出的导数， $\sigma'(z^{(L)})$ 是激活函数的导数。对于隐藏层的梯度：

$$\delta^{(l)} = (W^{(l+1)})^T \delta^{(l+1)} \odot \sigma'(z^{(l)})$$

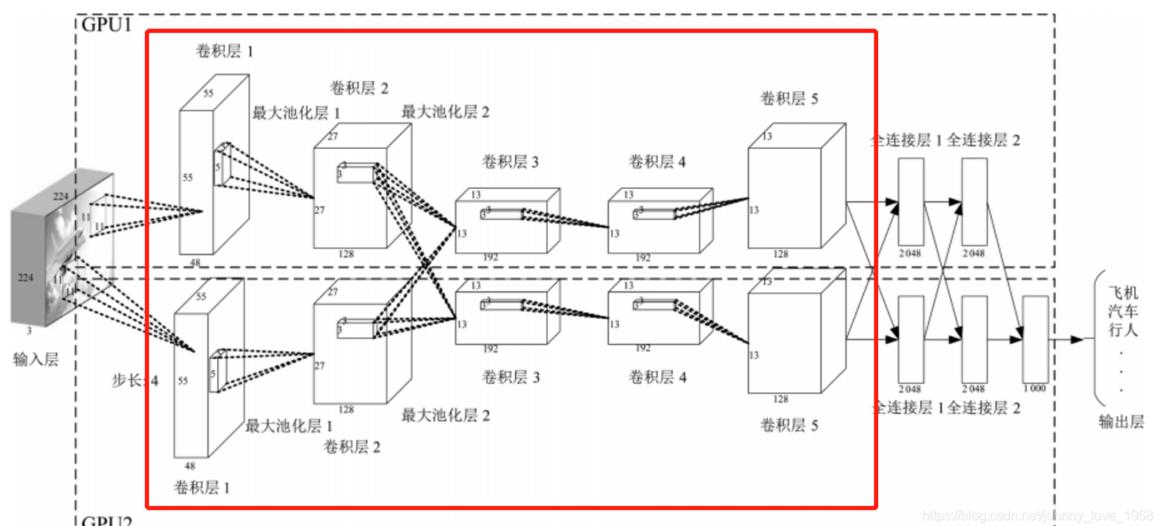
递推公式依次向前层传递，计算各层的误差。最后计算权重和偏置更新：使用计算出的梯度，更新每一层的参数：

$$W^{(l)} = W^{(l)} - \eta \cdot \frac{\partial \mathcal{L}}{\partial W^{(l)}}$$

$$b^{(l)} = b^{(l)} - \eta \cdot \frac{\partial \mathcal{L}}{\partial b^{(l)}}$$

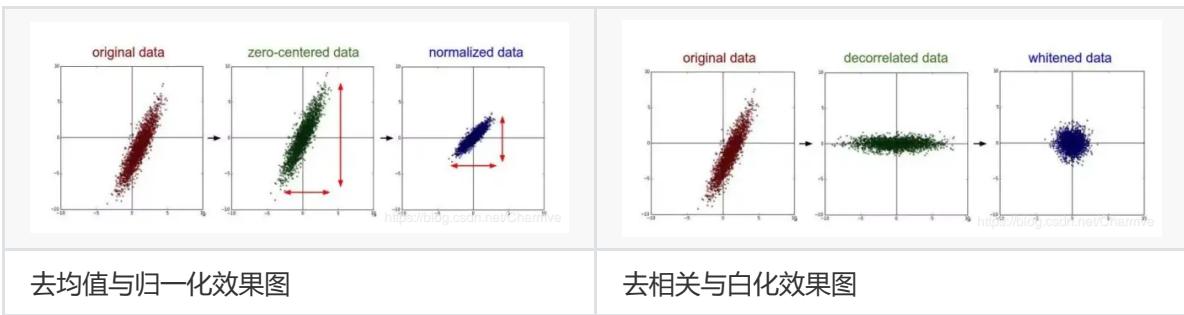
其中， η 是学习率。

【卷积神经网络 (CNN)】



主要由以下5层组成：

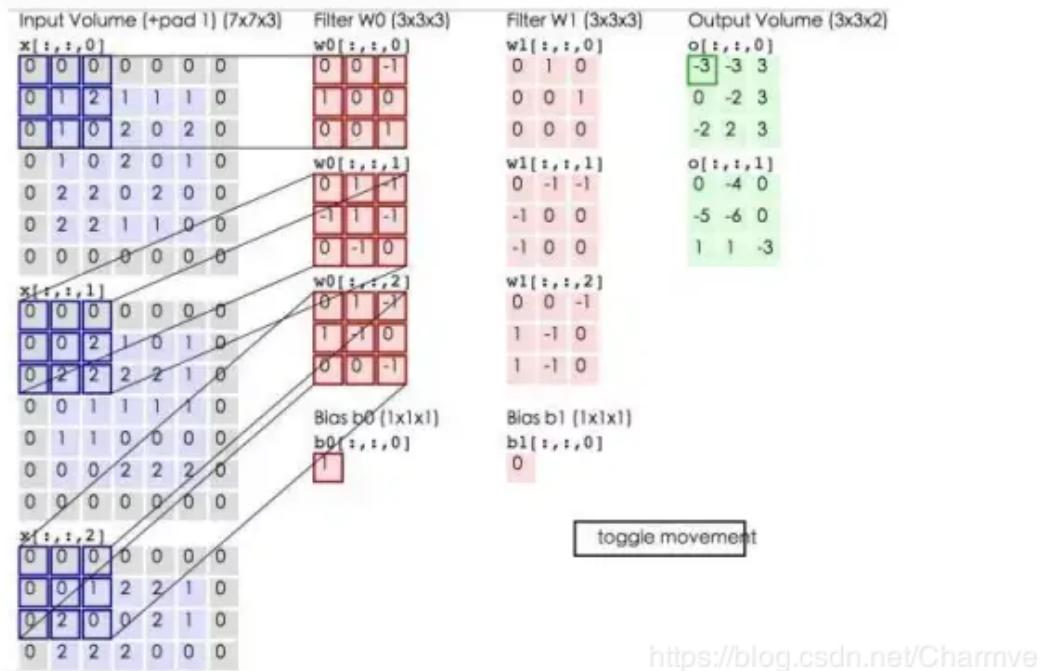
i. 数据输入层 Input layer —— 对原始图像数据进行预处理



去均值：把输入数据各个维度都中心化为0，其目的就是把样本的中心拉回到坐标系原点上。

归一化：幅度归一化到同样的范围，即减少各维度数据取值范围的差异而带来的干扰，比如，我们有两个维度的特征A和B，A范围是0到10，而B范围是0到10000，通过归一化将A和B的数据都变为0到1的范围。(PCA白化：用PCA降维；白化是对数据各个特征轴上的幅度归一化)

ii. 卷积计算层CONV layer —— 提取特征



输入矩阵格式：四个维度，依次为：样本数、图像高度、图像宽度、图像通道数

输出矩阵格式：与输出矩阵的维度顺序和含义相同，但是后三个维度（图像高度、图像宽度、图像通道数）的尺寸发生变化。

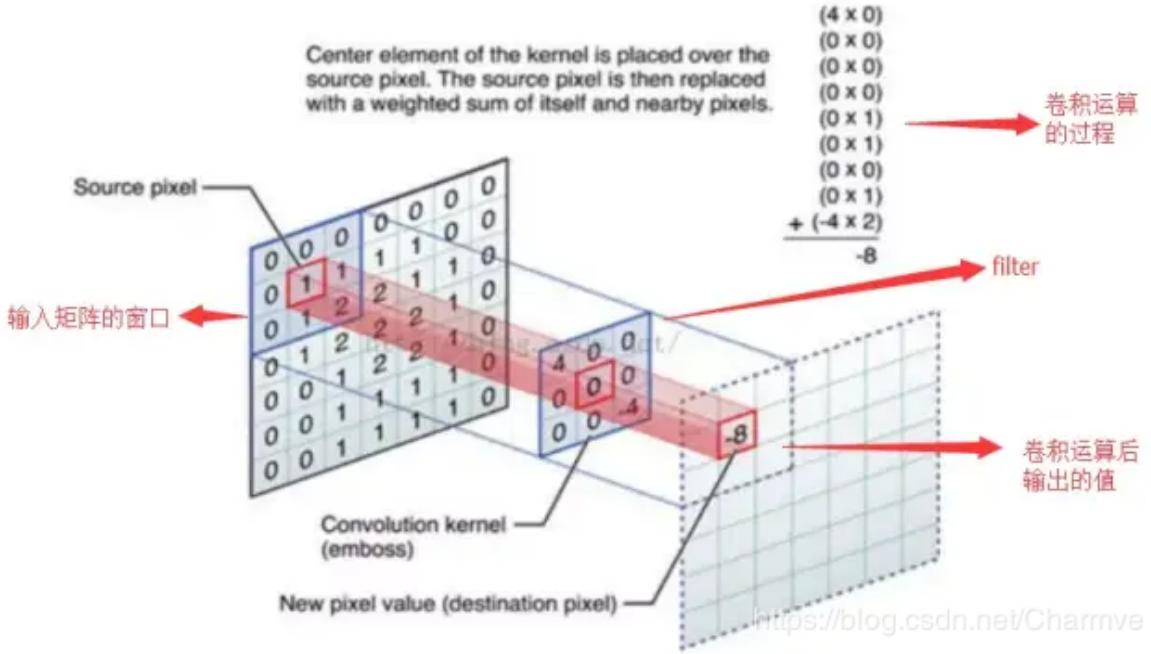
权重矩阵（卷积核）格式：同样是四个维度，但维度的含义与上面两者都不同，为：卷积核高度、卷积核宽度、输入通道数、输出通道数（卷积核个数）

输入矩阵、权重矩阵、输出矩阵这三者之间的相互决定关系

卷积核的输入通道数 (in depth) 由输入矩阵的通道数所决定。

输出矩阵的通道数 (out depth) 由卷积核的输出通道数所决定。

输出矩阵的高度和宽度由输入矩阵、卷积核（过滤器）的尺寸、步幅 (stride) 、填充 (padding) 共同决定。



设：

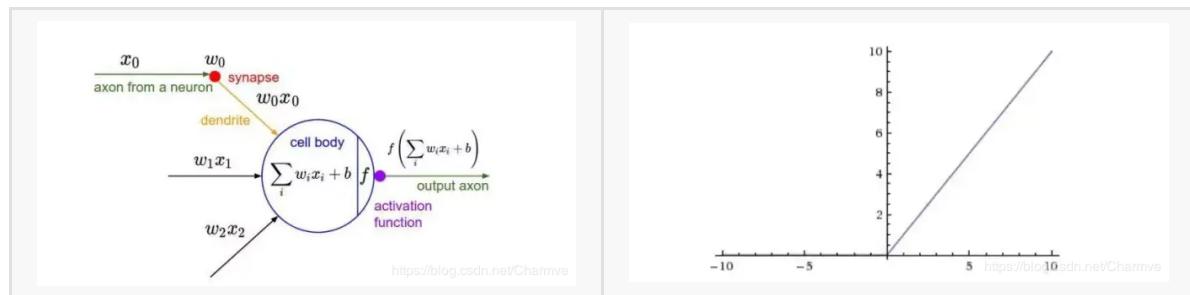
- 输入矩阵的高度为 H_{in} , 宽度为 W_{in}
- 卷积核的高度为 H_f , 宽度为 W_f
- 填充的高度为 P , 宽度为 P (假设填充在高度和宽度方向相同)
- 步幅为 S

则输出矩阵的高度 H_{out} 和宽度 W_{out} 计算公式为：

$$H_{out} = \left\lfloor \frac{H_{in}-H_f+2P}{S} \right\rfloor + 1$$

$$W_{out} = \left\lfloor \frac{W_{in}-W_f+2P}{S} \right\rfloor + 1$$

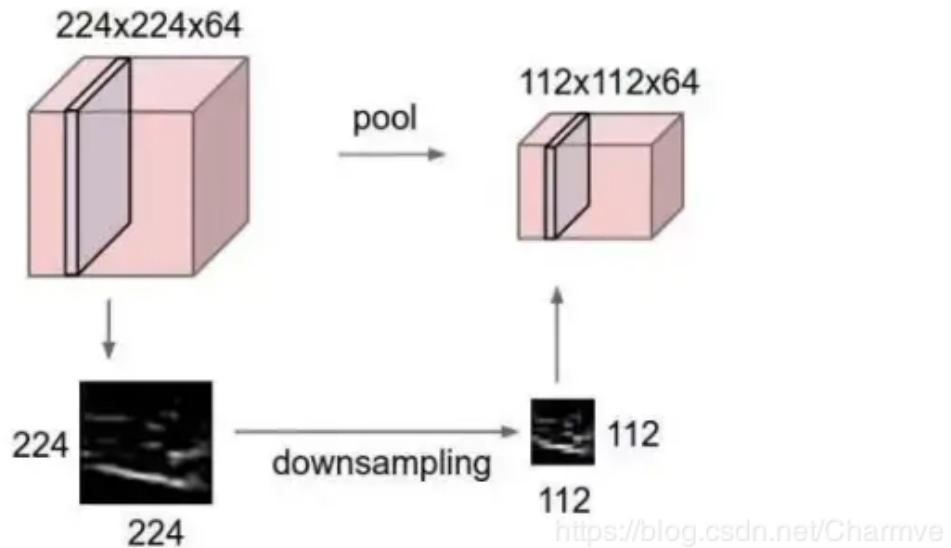
iii. ReLU激励层ReLU layer —— 引入非线性特征



在每个卷积层之后，通常会立即应用一个非线性层（或激活层）。其目的是给一个在卷积层中刚经过线性计算操作（只是数组元素依次（element wise）相乘与求和）的系统引入非线性特征。它同样能帮助减轻梯度消失的问题——由于梯度以指数方式在层中消失，导致网络较底层的训练速度非常慢。

ReLU 层对输入内容的所有值都应用了函数 $f(x) = \max(0, x)$ 。用基本术语来说，这一层把所有的负激活（negative activation）都变为零。这一层会增加模型乃至整个神经网络的非线性特征，而且不会影响卷积层的感受野。由于越靠近1表示与该特征越关联，越靠近-1表示越不关联，而我们进行特征提取时，为了使得数据更少，操作更方便，就直接舍弃掉那些不相关联的数据。 $>=0$ 的值不变，而 <0 的值一律改写为0

iv. 池化层Pooling layer —— 用于对输入特征图进行降维和特征提取



在ReLU层之后，会选择使用一个池化层，用于对输入特征图进行降维和特征提取。

1. 降维：通过减少特征图的空间尺寸（高度和宽度），降低计算复杂度和内存使用，防止过拟合。
2. 保留重要特征：池化操作保留输入特征图中的主要特征，同时丢弃次要的特征。这有助于模型更好地提取重要的信息。
3. 提高不变性：池化层可以增强特征图的平移不变性，即在输入图像发生小的平移时，输出特征图变化不大。这提高了模型对位置变化的鲁棒性。

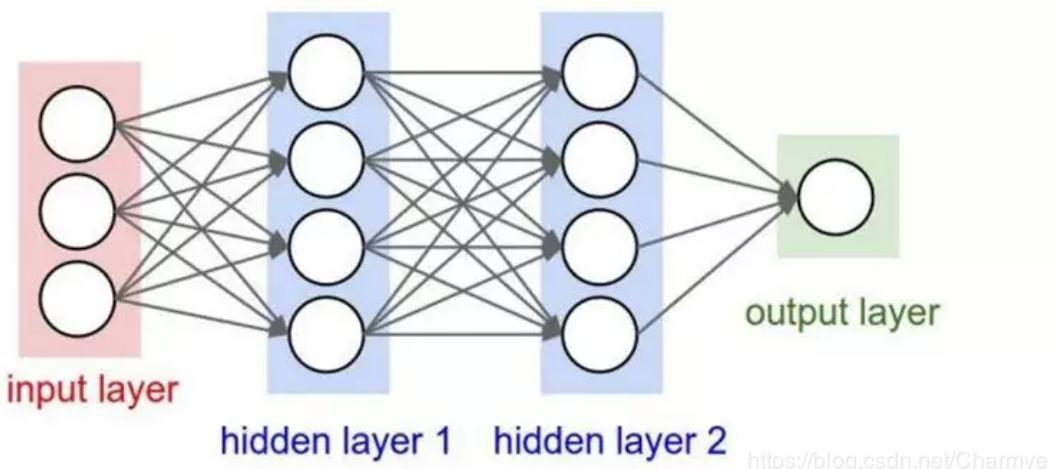
最大池化 (Max Pooling)

通过在特征图的局部区域（通常是一个小的矩形窗口）内取最大值来进行降维。假设窗口大小为 2×2 ，步幅为 2，则每个 2×2 的窗口中的最大值将被保留，其余值被丢弃。

通过以上的池化操作，我们减少了特征图的尺寸，同时保留了输入特征图的主要信息。

到这里就介绍了CNN的基本配置---卷积层、Relu层、池化层。在常见的几种CNN中，这三层都是可以堆叠使用的，将前一层的输入作为后一层的输出。

v.全连接层FC layer —— 综合提取的特征进行分类或回归任务



两层之间所有神经元都有权重连接，通常全连接层在卷积神经网络尾部。也就是跟传统的神经网络神经元的连接方式是一样的。

全连接层中的函数---Softmax，它是一个分类函数，输出的是每个对应类别的概率值。

vi.Dropout —— 正则化技术，旨在防止神经网络过拟合

训练阶段：在每个训练迭代中，对于每一层的每个神经元，以概率 p 临时将其“丢弃”（即将该神经元的输出设为零）。保留下来的神经元的输出除以 $1 - p$ 进行缩放，以保持输入的期望值不变。

测试阶段：在测试过程中，所有神经元都被激活，但不进行任何丢弃操作。

【pytorch中的卷积层、ReLU层、池化层、全连接层和Dropout层】

下面将介绍每种层及其常见参数。

i.卷积层

在 PyTorch 中，卷积层由 `torch.nn.Conv2d` 类实现。其主要参数包括输入通道数、输出通道数、卷积核大小、步幅和填充等。

```
import torch
import torch.nn as nn

# 定义一个卷积层
conv_layer = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, stride=1,
padding=1)
```

参数说明

- `in_channels`：输入通道数，例如 RGB 图像为 3。
- `out_channels`：输出通道数，表示卷积核的数量。
- `kernel_size`：卷积核大小，可以是单个整数或元组，如 `(3, 3)`。
- `stride`：步幅，可以是单个整数或元组。
- `padding`：填充，可以是单个整数或元组。

ii.ReLU层

在 PyTorch 中，ReLU 层由 `torch.nn.ReLU` 类实现。它没有可训练参数。

```
relu_layer = nn.ReLU()
```

参数说明

- ReLU 层没有参数，它只是将输入中的所有负值置零。

iii.池化层

在 PyTorch 中，`torch.nn.MaxPool2d` (最大池化)

```
# 定义一个最大池化层
max_pool_layer = nn.MaxPool2d(kernel_size=2, stride=2)
```

参数说明

- `kernel_size`：池化窗口的大小，可以是单个整数或元组。
- `stride`：步幅，可以是单个整数或元组。如果未指定，默认为 `kernel_size`。
- `padding`：填充，可以是单个整数或元组，默认为 0。

全连接层

在 PyTorch 中，全连接层由 `torch.nn.Linear` 类实现。

```
# 定义一个全连接层
fc_layer = nn.Linear(in_features=128, out_features=10)
```

参数说明

- `in_features`：输入特征数。
- `out_features`：输出特征数。

Dropout层

在 PyTorch 中，Dropout 层由 `torch.nn.Dropout` 类实现。它在训练过程中以一定的概率随机丢弃输入的部分元素。

```
# 定义一个 Dropout 层
dropout_layer = nn.Dropout(p=0.5)
```

参数说明

- `p`：丢弃的概率，范围在 0 到 1 之间。

【SGD算法、SGD Momentum算法和Adam算法】

SGD算法 (Stochastic Gradient Descent)

SGD是一种最常用的优化算法，它是传统梯度下降算法的变体。标准的梯度下降算法每次计算损失函数关于模型参数的**全部数据**的梯度，然后根据这个梯度更新参数。相对而言，SGD每次仅计算**一个样本**（或一小部分样本，即批量梯度下降中的mini-batch）上的梯度，从而加速了计算过程。

SGD的更新规则：

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} L(\theta_t; x^{(i)}, y^{(i)})$$

其中：

- θ 是模型的参数。
- η 是学习率。
- $\nabla_{\theta} L(\theta_t; x^{(i)}, y^{(i)})$ 是当前样本 $(x^{(i)}, y^{(i)})$ 上的梯度。

虽然SGD计算速度快，每次更新使用单个样本，适合大规模数据集，但是更新过程比较噪声大（因为使用单个样本的梯度），导致收敛速度较慢，甚至可能无法收敛，难以调整到全局最优解，由此引入了SGDM算法。

SGD Momentum算法

Momentum（动量）算法是对标准SGD的改进，它通过引入“动量”的概念来加速收敛并减少振荡。Momentum算法考虑到过去更新的方向，使得当前的更新不仅仅依赖于当前梯度，还会依赖于上次的更新方向，从而使参数更新过程更平滑、更稳定。

SGD Momentum的更新规则：

$$v_{t+1} = \beta v_t + (1 - \beta) \nabla_{\theta} L(\theta_t)$$

$$\theta_{t+1} = \theta_t - \eta v_{t+1}$$

其中：

- v_t 是动量项，表示当前梯度的累积值。
- β 是动量因子（通常设为接近1，比如0.9），控制了过去梯度的影响程度。
- 其余符号和SGD一致。

Adam算法 (Adaptive Moment Estimation)

Adam（自适应矩估计）是一种结合了动量和自适应学习率的优化算法。它不仅考虑梯度的历史信息，还对每个参数的学习率进行自适应调整。Adam对每个参数的梯度进行两种不同的估计：

- 一阶矩（梯度的平均值），类似于动量。
- 二阶矩（梯度的方差），用于调整每个参数的学习率。

Adam的更新规则：

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) \nabla_{\theta} L(\theta_t)$$

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2) (\nabla_{\theta} L(\theta_t))^2$$

$$\hat{m}_{t+1} = \frac{m_{t+1}}{1 - \beta_1^{t+1}}, \quad \hat{v}_{t+1} = \frac{v_{t+1}}{1 - \beta_2^{t+1}}$$

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1}} + \epsilon}$$

其中：

- m_t 是梯度的一阶矩（动量）。
- v_t 是梯度的二阶矩（梯度的方差）。
- β_1 和 β_2 是控制一阶矩和二阶矩的衰减率，通常设置为接近1（例如0.9和0.999）。
- ϵ 是一个很小的常数，用来避免除零错误（通常设为 10^{-8} ）。

特性	SGD	SGD Momentum	Adam
更新规则	仅使用当前梯度进行更新	结合当前梯度和之前的梯度（动量）	使用梯度的一阶和二阶矩，进行自适应调整
收敛速度	相对较慢，容易震荡	加速收敛，减少震荡	快速收敛，特别是在复杂问题中
计算开销	低	稍高，但仍然较低	相对较高，需要维护动量和方差估计
适应性	固定学习率	动量帮助稳定收敛	自适应学习率，更灵活且适用广泛
超参数调整	学习率需要调节	学习率和动量因子需要调节	学习率和 β 值需要调节

- SGD简单但容易陷入局部最优解或收敛慢。
- SGD Momentum通过引入动量来加速收敛并减少震荡，适用于大部分任务。
- Adam结合了动量和自适应学习率，通常在大规模深度学习任务中表现出色，但可能需要更多的超参数调整。

2.关键代码展示

【数据预处理】

```
# 数据处理，加载数据
def load_data(dir):
    X_train = []
    Y_train = []
    # 读取训练数据
    for i in range(1, 6):
        file_path = os.path.join(dir, f'data_batch_{i}')
        print(f"Loading file: {file_path}") # 打印加载的文件路径
        with open(file_path, 'rb') as fo:
```

```

        dict = pickle.load fo, encoding='bytes') # pickle.load()函数可以将文件
中的数据解析为一个python对象
        X_train.append(dict[b'data'])
        Y_train += dict[b'labels']
    X_train = np.concatenate(X_train, axis=0) # 沿着指定轴连接数组, concatenate函数的
第一个参数是一个列表, 表示要连接的数组, 第二个参数axis表示连接的轴

    """
在 CIFAR-10 数据集中 dict[b'data'] 通常是一个已经预处理成 NumPy 数组格式的数据。
因此 直接赋值 X_test = dict[b'data'] 会使得 X_test 成为 NumPy 数组
"""

# 读取测试数据
test_file_path = os.path.join(dir, 'test_batch')
print(f"Loading file: {test_file_path}")
with open(test_file_path, '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

# 数据预处理, 进行归一化

def preprocess(X_train, Y_train, X_test, Y_test):
    X_train = X_train.reshape((-1, 3, 32, 32)).astype(np.float32) / 255.0 # 将数
据转换为3通道的图片格式, 并归一化, astype转换数据类型
    X_test = X_test.reshape((-1, 3, 32, 32)).astype(np.float32) / 255.0 #
reshape参数-1表示自动计算该维度的大小, 3表示通道数, 32表示图片的长宽
    Y_train = np.array(Y_train) # 将list转换为numpy数组
    Y_test = np.array(Y_test)

return X_train, Y_train, X_test, Y_test

# 定义数据集类
class CIFAR10Dataset(Dataset): # pytorch的数据集类需要继承torch.utils.data.Dataset
    def __init__(self, X, Y):
        self.data = torch.from_numpy(X).float() # 将数据转换为 PyT
        self.label = torch.from_numpy(Y).long() # 将标签转换为 PyTorch 的
LongTensor 类型

    def __len__(self):
        return self.data.shape[0] # 返回数据集的大小

    def __getitem__(self, idx):
        return self.data[idx], self.label[idx] # 返回数据和标签

```

【softmax】

```

# softmax分类器

class Softmax(nn.Module):
    def __init__(self):
        super(Softmax, self).__init__() # 调用父类的构造函数
        self.fc = nn.Linear(3 * 32 * 32, 10) # 全连接层, 输入维度是 3 * 32 * 32, 输出
        维度是 10, 对应于 10 个类别
    def forward(self, x):
        x = x.view(x.size(0), -1) # 将输入数据展平成一维向量
        x = self.fc(x) # 全连接层
        x = F.log_softmax(x, dim=1)
        return x

```

【MLP】

```

class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(3 * 32 * 32, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = self.fc4(x)
        x = F.log_softmax(x, dim=1)
        return x

```

【CNN】

```

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5) # 输入通道数为3, 输出通道数为6, 卷积核大小为5
        self.pool = nn.MaxPool2d(2, 2) # 2*2的最大池化层
        self.conv2 = nn.Conv2d(6, 16, 5) # 输入通道数为6, 输出通道数为16, 卷积核大小为5
        self.fc1 = nn.Linear(16 * 5 * 5, 120) # 全连接层
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x))) # 卷积层1 -> 激活函数 -> 池化层
        x = self.pool(F.relu(self.conv2(x))) # 卷积层2 -> 激活函数 -> 池化层
        x = x.view(-1, 16 * 5 * 5) # 展平
        x = F.relu(self.fc1(x)) # 全连接层1 -> 激活函数
        x = F.relu(self.fc2(x)) # 全连接层2 -> 激活函数
        x = self.fc3(x) # 全连接层3
        x = F.log_softmax(x, dim=1) # softmax
        return x

```

【train】

```

def train_model(model, device, train_loader, optimizer, epoch, loss_history,
accuracy_history):
    model.train()
    cot_cnt = 0 # 正确分类的样本数
    total = 0 # 总样本数
    running_loss = 0.0 # 损失值
    for data, target in train_loader:
        data, target = data.to(device), target.to(device) # 将数据和标签转移到GPU上
        训练
        optimizer.zero_grad() # 梯度清零
        forward = model(data) # 前向传播
        loss = F.nll_loss(forward, target) # nll_loss() 函数计算负对数似然损失
        loss.backward() # 反向传播
        optimizer.step() # 更新参数

        running_loss += loss.item() # 累加损失值

        # 计算训练集准确率
        pred = forward.max(1, keepdim=True)[1] # 找到概率最大的下标
        cot_cnt += pred.eq(target.view_as(pred)).sum().item() # 计算正确分类的样本数
        total += target.size(0)

    avg_loss = running_loss / len(train_loader)
    accuracy = cot_cnt / total
    loss_history.append(avg_loss)
    accuracy_history.append(accuracy)

    print(f'Train Epoch: {epoch}\tLoss: {avg_loss:.6f}\tAccuracy: {accuracy * 100:.2f}%')

```

【test】

```

# 测试模型
def test_model(model, device, test_loader, accuracy_history, test_loss_history):
    model.eval()
    cot_cnt = 0
    total = 0
    test_loss = 0
    with torch.no_grad(): # 不计算梯度，加速计算
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            loss = F.nll_loss(output, target, reduction='sum').item()
            test_loss += loss
            pred = output.max(1, keepdim=True)[1]
            cot_cnt += pred.eq(target.view_as(pred)).sum().item()
            total += target.size(0)

    test_loss /= len(test_loader.dataset)
    accuracy = cot_cnt / total
    accuracy_history.append(accuracy)
    test_loss_history.append(test_loss)
    print(f'Test set: Average loss: {test_loss:.4f}, Accuracy: {cot_cnt}/{total}
({100. * accuracy:.2f}%)')

```

四、实验结果及分析

1.softmax分类器性能分析+SGD、 SGDM和Adam优化器的对比分析

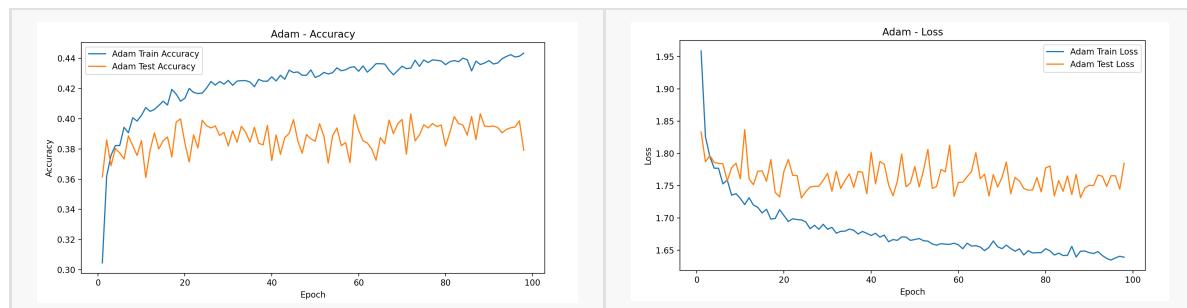
【训练过程】——初始化方法、超参数选择、用到的训练技巧

【初始化方法】softmax分类器可以看成单层感知机，因此我们对softmax函数的定义也是较为简单，我们直接使用单层神经网络进行定义，使用一个input层和一个output层，只是最后在分类的时候使用softmax函数，使用概率进行分类。将原始的四维数据降维以使得其能够对应到全连接层，输入维度是 $3 * 32 * 32$ ，输出维度是10，对应于10个类别。为了减少训练次数并且提高训练效率，将batch设置成256，同时分别使用SGD、 SGDM和Adam优化器来训练模型。

【超参数选择】由于softmax没有什么特定的参数，因为其本身就是一个二层的神经网络，我们这里仅考虑输入的维度和输出维度，这里由于是在CIFAR-10数据集上进行训练，因此我们根据常用的数据维度定义，将输入维度定义成 $3 * 32 * 32$ ，由于有10个分类，因此我们将输出维度定义成10，表示我们最终的输出是10个分类。

【用到的训练技巧】这里我们根据题目要求分别使用了SGD、 SGDM和Adam优化器来训练模型，这里我们先分开放出每一种优化器的训练结果，最后将这几个优化器的训练结果放在一张图中进行对比。

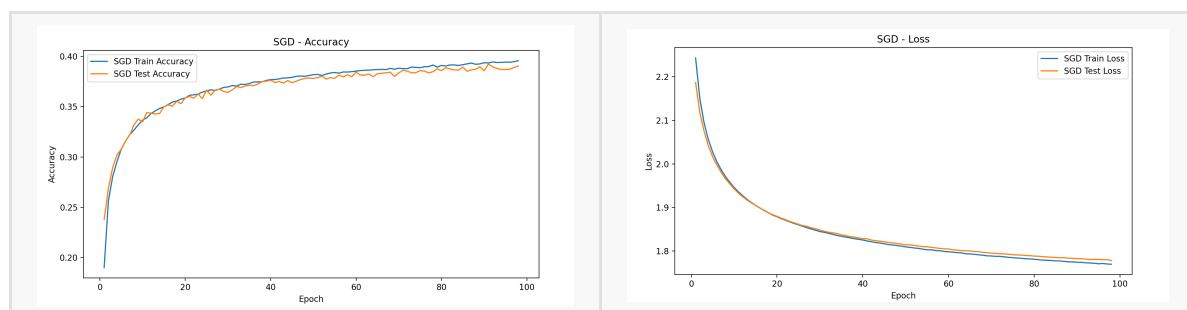
- Adam优化器



【分析】

可以看到Adam的训练集的准确率大致徘徊在45%左右，而其测试集的准确率大致徘徊在40%左右，同时可以看到两者的loss曲线都已经收敛，表明他们都已经达到最优处，而且loss曲线也表现的较好，没有出现过拟合现象，当然可以看到训练结果的准确率较差，以及loss也没有收敛到一个很低的状态，这很大程度上是因为softmax自身的问题，由于其本身可以看成一个二层的神经网络，相比于主流的MLP和CNN来说，其在训练的过程中丢失了很多数据集的细节，由此导致在处理多分类复杂问题的过程中表现较为逊色。

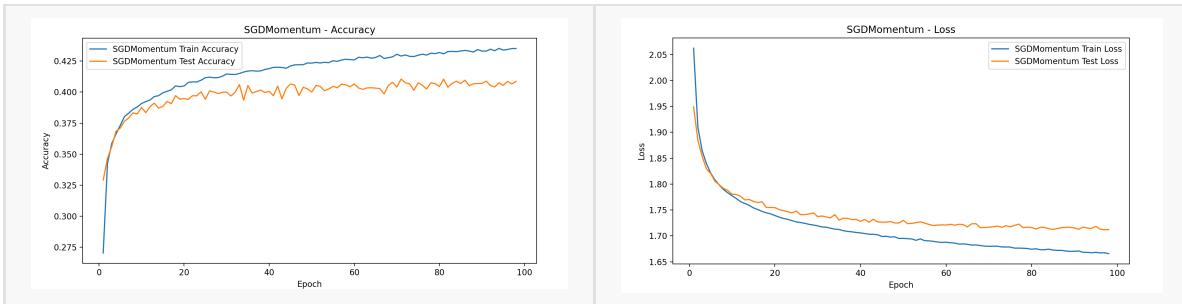
- SGD优化器



【分析】

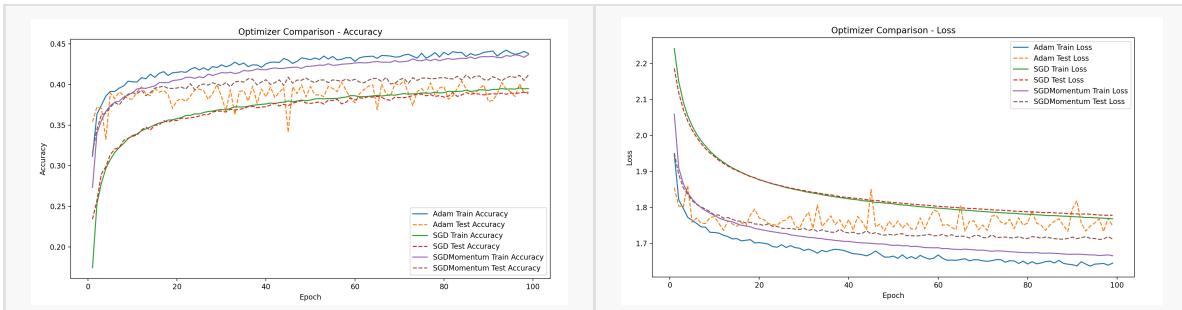
可以看到SGD的训练集的准确率和测试集的准确率都大致徘徊在40%左右，同时可以看到两者的loss曲线都已经收敛，表明他们都已经达到最优处，而且loss曲线也表现的较好，没有出现过拟合现象，相较于上面的Adam曲线来说可以看到SGD的loss曲线更为平滑，几乎没有出现凸起不平，当然可以看到训练结果的准确率较差，以及loss也没有收敛到一个很低的状态，这实际上也是来源于softmax本身的问题。

- SGDM优化器



【分析】

可以看到SGDM的训练集的准确率大致徘徊在43%左右，而其测试集的准确率大致徘徊在40%左右，同时可以看到两者的loss曲线都已经收敛，表明他们都已经达到最优处，而且loss曲线也表现的较好，没有出现过拟合现象，相较于上面的SGD来说，可以明显的看到SGDM的训练效果更好，这是因为SGDM在训练的时候加入了动量的优化，使得数据集在训练的过程中更不易受到局部最优解的影响，当然可以看到其相对于Adam优化器来说，它的性能就表现的相对较差了，很明显Adam的性能更胜一筹。



【分析】

这里我们将三个优化器的测试曲线放在同一张图中进行对比，这里可以更清楚地看到几个优化器训练的性能，很明显Adam的训练效果是最好的，而SGD的训练性能是最差的，SGDM由于在SGD的基础上增加了动量，在一定程度上避免了一些局部最优解。这里出现这样的结果也显得很平凡了，因为对比一下这几个优化器，很明显Adam优化器考虑了更多的训练缺陷，由此在训练过程中表现的最好，而SGD只考虑最普通的梯度下降，很容易trap到一个局部化的最优解，导致准确率降低。Adam优化器的性能最强，它能自动调整每个参数的学习率，通过使用梯度的一阶矩估计（均值）和二阶矩估计（方差），能够避免局部最优问题，表现出色。SGD，作为最基本的梯度下降方法，确实容易陷入局部最优解，因此在面对复杂任务时，通常性能较差，尤其是当数据的特征空间比较复杂时。SGDM（带动量的SGD）通过引入动量，能够在一定程度上加速收敛并减小震荡，较常见地用于加速收敛，避免一些局部最优问题，但相比于Adam，其表现仍然逊色，主要因为Adam考虑了更多的动态因素。

这里也可以看到softmax的训练效果并不好，准确率仅有40%左右，这其实也是较为正常的现象，Softmax作为激活函数在多分类任务中的局限性，尤其是在处理复杂数据集时，容易丢失很多细节。Softmax本质上会将问题转化为二层神经网络，这使得它在面对复杂任务时不如更深层次的网络（如MLP或CNN）表现优秀。实际上，Softmax适合于比较简单的分类问题，或者说特征之间有较强线性可分性的任务。

2. MLP性能分析+SGD、SGDM和Adam优化器的对比分析

【训练过程】——初始化方法、超参数选择、用到的训练技巧

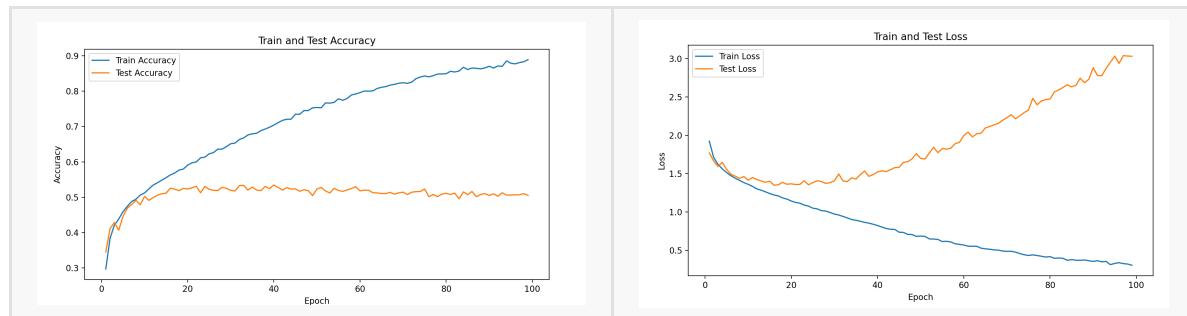
【初始化方法】 MLP是一个多层感知机模型，更详细的讲他是一个多层神经网络模型，相比于上面的softmax分类器，MLP多了隐藏层的使用，并且在模型寻来你的过程中使用了正向传播和反向传播的方法使得训练的准确率得到显著地提高。由此我们在初始化的时候需要在前面的二层神经网络的基础上增加隐藏层使得其转化成多层神经网络模型，在初始的模型上，我定义了四层神经网络，第一层是输入层，中间有两层隐藏层，以及最后的输出层，这里需要根据数据集的定义来改变每一层的参数，即每一层神经网络需要的通道数，最后可以看到吧输出的最后维数是10维，这是由于我们需要讲原始数据集分

成10类。为了使数据更好被分类，我们在每一层之后使用激活函数来做优化以提高神经网络的非线性拟合能力。前三层激活函数均为 ReLU 函数，最后一层为 log_softmax 函数。

【超参数选择】学习率 (lr)：学习率决定了模型在每次更新权重时的步长。学习率过大可能导致模型发散，过小则收敛速度过慢。正则化参数 (alpha)：正则化项 (L2 范数) 用于防止模型过拟合。训练轮数 (epochs)：训练轮数应足够多以确保模型收敛，但过多的轮次可能导致过拟合。

【用到的训练技巧】这里我们根据题目要求分别使用了SGD、 SGDM和Adam优化器来训练模型，这里我们先分开给出每一种优化器的训练结果，最后将这几个优化器的训练结果放在一张图中进行对比。

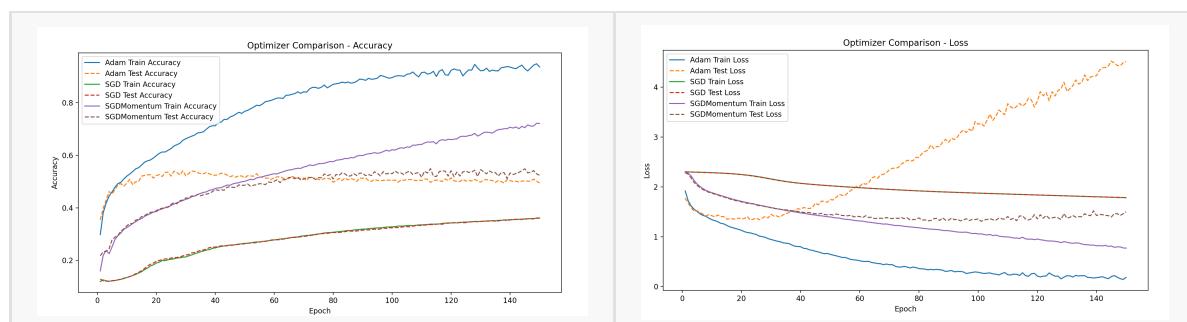
- MLP使用Adam优化器训练的原始模型



【分析】

我们可以看到原始模型通过Adam训练的结果，训练集的准确率处于一个上升的状态，且损失值也一直在降低，表明在训练集的分类中MLP模型的分类情况较好；接着再来观察测试集，可以看到相比于训练集，测试集无论是在准确率还是在损失值上都表现得较差，测试集的准确率只能达到50%左右，同时可以看到损失值最初处于一个下降的趋势，但是在20次迭代之后loss却开始呈现一个上升的趋势，这表明出现了过拟合现象，这也也在某种程度上表明了测试集的准确率在这个参数设计下的mlp达到了上界，最后即便多次迭代，也无法再做到更好。

- SGD、 SGDM和Adam优化器的对比分析



【分析】

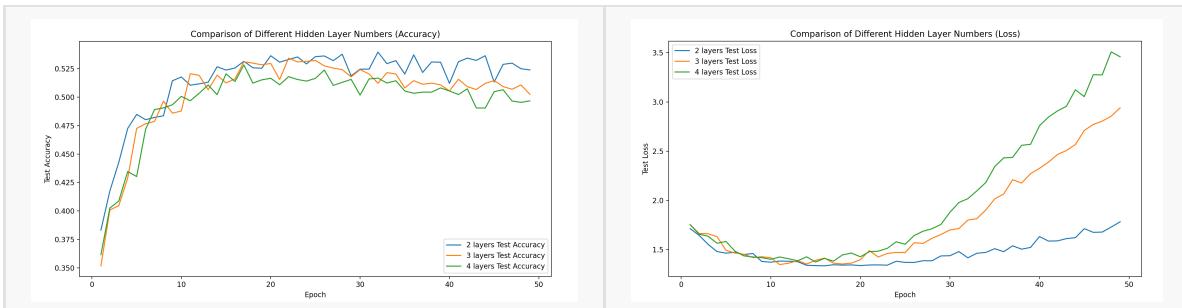
这里不再像上面的softmax分析一样将几个优化器分开来分析，我们将三个优化器的测试曲线放在同一张图中进行对比，这里可以更清楚地看到几个优化器训练的性能，很明显Adam的训练效果是最好的，而SGD的训练性能是最差的， SGDM由于在SGD的基础上增加了动量，在一定程度上避免了一些局部最优解。这里可以看出和上面的softmax分析是一样的，但是再MLP训练的过程中，我们可以很明显地从曲线中看到SGD的优化非常差，在训练SGD的时候出现如下问题，即准确率不发生变化并且后续准确率上升处于一个很慢的速度， loss曲线也处于一个相对平稳的状态。这表明在MLP模型中， SGD会很容易trap到一个局部中，导致准确率很难增长。

```
Test set: Average loss: 1.055, Accuracy: 5892/10000 (58.92%)
```

Training with SGD optimizer...

```
Train Epoch: 1 Loss: 2.305771 Accuracy: 10.00%
Test set: Average loss: 2.3056, Accuracy: 1000/10000 (10.00%)
Train Epoch: 2 Loss: 2.305519 Accuracy: 10.00%
Test set: Average loss: 2.3055, Accuracy: 1000/10000 (10.00%)
Train Epoch: 3 Loss: 2.305388 Accuracy: 10.00%
Test set: Average loss: 2.3053, Accuracy: 1000/10000 (10.00%)
Train Epoch: 4 Loss: 2.305224 Accuracy: 10.00%
Test set: Average loss: 2.3051, Accuracy: 1000/10000 (10.00%)
Train Epoch: 5 Loss: 2.305070 Accuracy: 10.00%
Test set: Average loss: 2.3050, Accuracy: 1000/10000 (10.00%)
Train Epoch: 6 Loss: 2.304881 Accuracy: 10.00%
Test set: Average loss: 2.3048, Accuracy: 1000/10000 (10.00%)
Train Epoch: 7 Loss: 2.304814 Accuracy: 10.00%
Test set: Average loss: 2.3047, Accuracy: 1000/10000 (10.00%)
Train Epoch: 8 Loss: 2.304592 Accuracy: 10.00%
Test set: Average loss: 2.3046, Accuracy: 1000/10000 (10.00%)
Train Epoch: 9 Loss: 2.304454 Accuracy: 10.00%
Test set: Average loss: 2.3044, Accuracy: 1000/10000 (10.00%)
Train Epoch: 10 Loss: 2.304376 Accuracy: 10.00%
Test set: Average loss: 2.3043, Accuracy: 1000/10000 (10.00%)
```

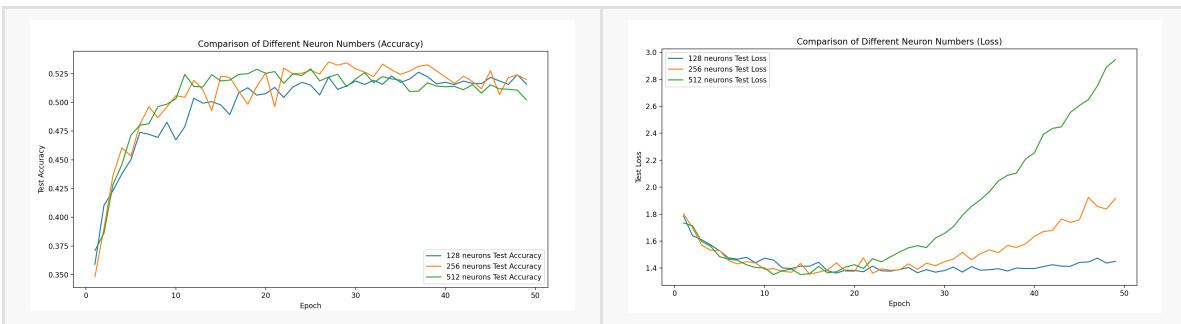
- 使用不同网络层数对模型性能的影响



【分析】

我们改变网络的层数来分析对准确率有什么影响，这里我们基于原始模型来对比，即4层网络模型，同时都使用Adam优化器来训练，这里我省略了训练集的准确率和损失函数的显示，直接观察测试集的准确率和损失函数。这里结果有点反常，正常来说神经网络的层数的增加会导致准确率的增加，但是很反常，在我的测试曲线中可以看到神经网络层数缺少，准确率反而越高。但实际上这三种层数训练出来的准确率都差不多，之所以两层神经网络的在测试集准确率上看着好像比两外两个更高一点，但实际上可能是因为它网络层数少，训练更快，上升更快一点。从Loss曲线中我们呢就可以看出来三层和四层神经网络在过拟合现象中表现得更为明显，而两层的loss曲线过拟合现象则不是特别明显，这在某种程度上也反应了当网络层数更大时，模型的参数数量往往越多，导致模型在相同的迭代次数下训练时间更长、更容易出现过拟合现象。然而在准确率上，更大的网络层数并不一定能带来提升。因此，我们需要根据实际问题选择合理的网络层数。

- 使用不同神经元数量对模型性能的影响



【分析】

在神经元数量方面，其实我们同样可以看到上面在神经元层数上出现的问题，三者的准确率其实差不多，但是多神经元明显过拟合现象更为严重。神经元数量增加会导致模型参数增加，使得模型可以更好地拟合数据，但也更容易出现过拟合现象，导致测试集准确率无法出现提升。

2. CNN性能分析+SGD、SGDM和Adam优化器的对比分析

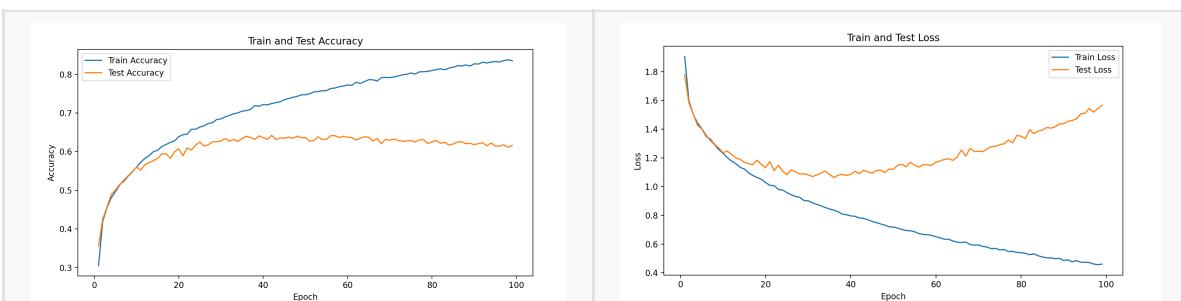
【训练过程】——初始化方法、超参数选择、用到的训练技巧

【初始化方法】 Lenet是一个 7 层的神经网络，包含 3 个卷积层，2 个池化层，1 个全连接层，1 个输出层。其中所有卷积层的卷积核都为 5x5，步长=1，池化方法都为平均池化，激活函数为 Sigmoid（目前使用的Lenet已改为ReLU）。在代码中，CNN 模型的初始化主要通过定义不同的卷积层（Conv Layer）和全连接层来完成。每个模型（如 `LeNet3Conv`, `LeNet4Conv`）都继承自 `torch.nn.Module`，并在 `__init__` 方法中定义了卷积层、池化层、全连接层及激活函数。卷积层（Convolutional Layers）：每个卷积层使用了 `nn.Conv2d`，初始化时会随机生成卷积核权重。卷积核的数量和大小（例如 `5x5`）是超参数，可以根据需要调整。池化层使用了最大池化（`MaxPool2d`）来进行下采样，池化核大小为 `2x2`，步长为 2。全连接层使用 `nn.Linear` 来定义全连接层，每层的输入和输出大小根据卷积层的输出形状来设定。

【超参数选择】 在 `optim.Adam` 优化器中，学习率没有明确给出，使用了默认的学习率 `lr=0.001`。学习率是影响模型收敛速度和稳定性的关键超参数。在 `DataLoader` 中设置 `batch_size=256`，即每次训练使用 256 个样本。优化器（Optimizer）使用了 `Adam` 优化器。`Adam` 是一种常用的自适应学习率优化器，具有较好的收敛效果。它会自动调整每个参数的学习率，减少手动调整的需要。损失函数（Loss Function）使用 `F.nll_loss` 作为损失函数，这是一个常用于多分类问题的负对数似然损失函数

【用到的训练技巧】 反向传播和梯度更新，使用 `optimizer.zero_grad()` 清除上一轮的梯度信息；使用 `loss.backward()` 进行反向传播，计算梯度；使用 `optimizer.step()` 来更新模型的参数。在训练时，调用 `model.train()` 让模型处于训练模式，这样会启用 Dropout 和 BatchNorm 等训练时特有的操作。在评估时，调用 `model.eval()` 将模型切换为评估模式，这时模型的行为不包括 Dropout 等操作。在每个批次中计算损失和准确率，并在每个 epoch 结束时打印出来。训练准确率和测试准确率分别被存储在 `accuracy_history_train` 和 `accuracy_history_test` 列表中，训练损失和测试损失分别存储在 `loss_history_train` 和 `loss_history_test` 中。

- CNN使用Adam优化器训练的原始模型



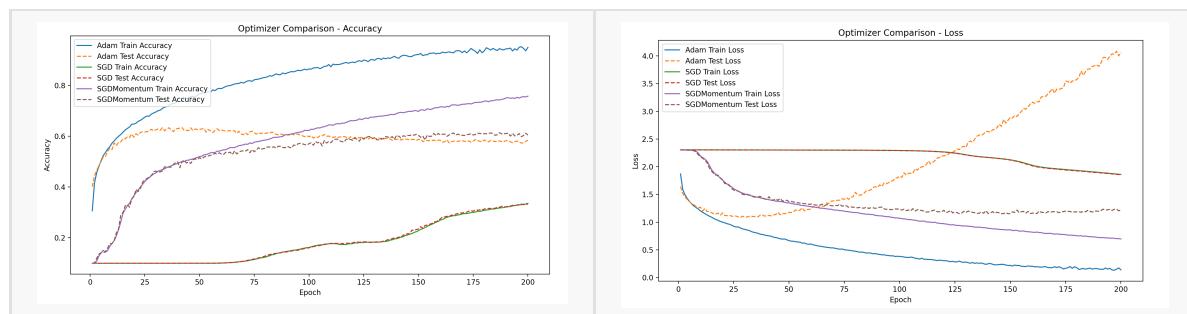
【分析】

我们可以看到训练准确率从0逐步快速上升，最终逼近1.0（接近100%的训练集拟合）。而测试准确率在100个epoch内缓慢上升，最终达到约0.6-0.7，明显低于训练准确率。接着反观训练损失呈现快速下降趋势，在20个epoch内迅速减少，随后逐步趋于稳定，表明模型在训练集上较好地拟合了数据。测试损失在初期下降（约20个epoch后）达到较低值，但随后波动较大甚至略有上升，反映了模型在测试集上存在过拟合。

当然Adam优化器是一种自适应学习率的优化算法，结合了动量法和RMSprop的优势动量法加速了梯度下降，尤其是在损失函数具有高曲率的情况下，能更快地收敛。RMSprop根据不同参数的梯度变化自动调整学习率，避免了传统SGD中可能出现的震荡和不稳定。这些特性使Adam在训练初期表现出快速下降的损失，如右图所示的训练损失曲线的急剧下降。

再详细分析一下过拟合原因，可能由以下几个原因导致：CNN模型参数较多，若数据样本不足或缺乏正则化，会倾向于过拟合训练集；训练集数据的多样性不足可能使模型难以泛化到测试集；尽管Adam优化器能快速拟合训练集，但也可能过早陷入局部最优解或过度拟合训练数据。

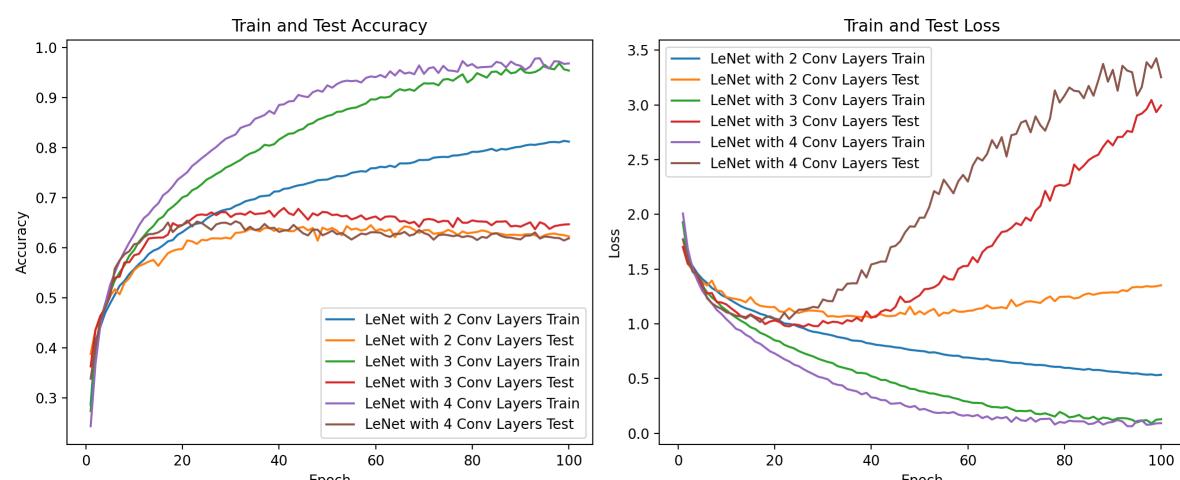
- SGD、SGDM和Adam优化器的对比分析



【分析】

可以明显地看到Adam优化器在初期阶段能快速降低损失和提升准确率，尤其在训练集上拟合速度非常快，其在测试集上的损失曲线后期有波动，表明其可能会在后期表现出过拟合现象。这是因为Adam在训练时对每个参数动态调整学习率，可能导致后期优化不够稳定。SGDM通过引入动量，改善了传统SGD中参数更新过程中的震荡问题，使得优化过程更平滑。从测试损失和准确率曲线可以看出，SGDM在测试集上的表现非常稳定，最终准确率接近Adam，但更稳定。SGD直接使用单次样本梯度更新参数，导致优化速度较慢，训练损失下降曲线非常缓慢。由于更新缺乏动量或动态调整策略，SGD最终的测试准确率和损失都明显低于Adam和SGDM。Adam快速拟合训练集，但可能导致过拟合问题。SGDM优化路径平稳，且更容易找到能泛化到测试集的全局解，因此测试性能更好。SGD优化速度慢，测试性能受影响。

- 使用不同卷积层数对模型性能的影响

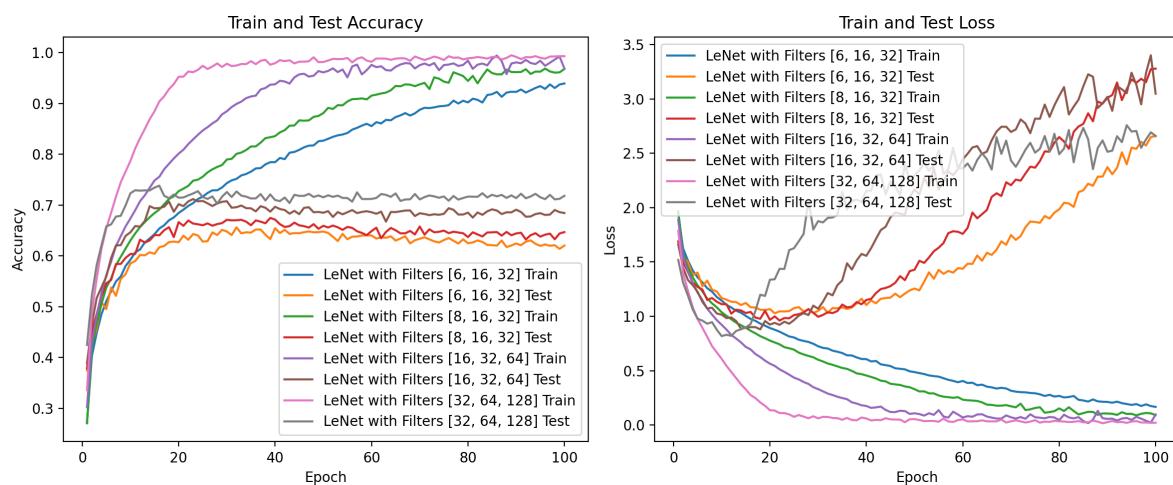


【分析】

对于两层神经网络，训练准确率和测试准确率在初期快速上升，但之后似乎达到了一个平稳期且测试集的准确率不高。对于三层神经网络，训练准确率和测试准确率的趋势相似，但测试准确率略高于LeNet2Conv，说明增加一个卷积层能使模型的表现更好，尤其是在测试集上。测试集的准确性也提升了，可能因为更复杂的特征提取能力。使用四层神经网络的训练和测试准确率继续提高，测试准确率明显高于前两个模型。这表明增加更多的卷积层进一步提高了模型的表现，能够更好地捕捉数据中的复杂模式。此时，随着卷积层数量增加，可能模型能够学习到更多的高级特征，因此在测试集上表现更好。但是可以看到其测试集的过拟合现象较为严重。

增加卷积层数，特别是 LeNet with 4 Conv Layers，虽然模型更复杂，但它的表现却好于前两个模型，并且测试准确率也有所提升。这意味着，虽然深度网络容易导致过拟合，但在一定范围内，增加网络的复杂度（如增加卷积层数）确实能帮助模型学习到更多的特征，进而提高其泛化能力。不过，需要注意的是，随着网络层数的增加，模型的训练时间也会增加，且如果数据集较小或过于简单，可能会面临过拟合风险。因此，除了增加网络深度，还可以考虑正则化技术（如 dropout、数据增强等）来进一步防止过拟合。

- 使用不同滤波器数量对模型性能的影响

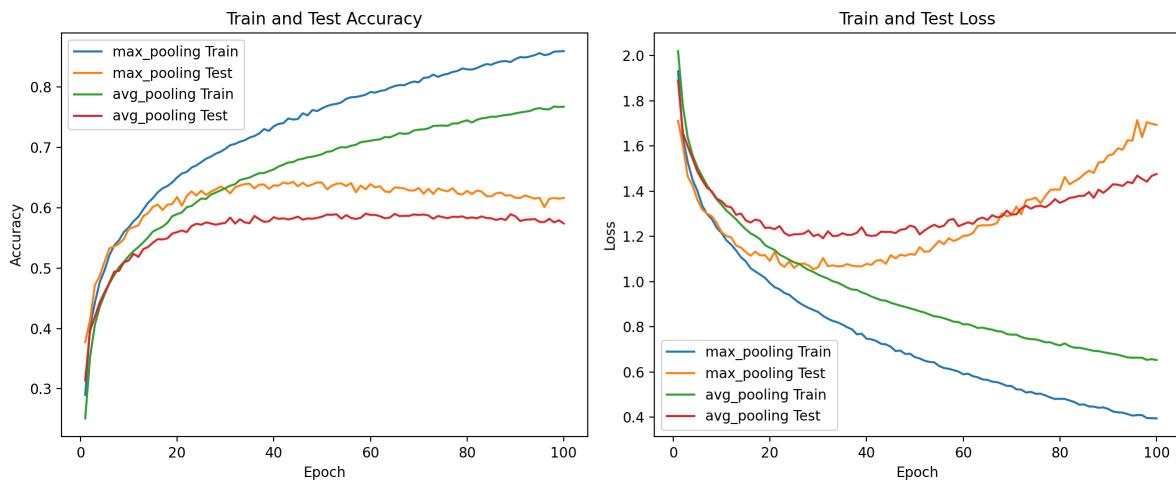


【分析】

滤波器数较少 (6, 16, 32) 的模型准确率较低，但随着epoch增加，性能逐步上升。对于中等滤波器数 (16, 32, 64) 的模型在训练和测试集上的表现相对来说是最好的，准确率快速上升并趋于稳定。滤波器数最多 (32, 64, 128) 的模型训练准确率非常高，但表现出明显的过拟合。再看损失曲线，滤波器数较少的模型损失下降缓慢，但测试损失曲线较平滑。中等滤波器数的模型在训练和测试损失上表现最平衡，测试损失较低且下降趋势平稳。滤波器数较多的模型训练损失迅速下降，但测试损失在后期反而上升，反映出过拟合现象。

由此我们也可以看到滤波器数过少时，模型容量不足，无法捕捉数据中的复杂特征，导致训练和测试表现都较差。滤波器数适中时，模型能够有效学习数据特征，同时不容易过拟合，因此在测试集上表现最佳。滤波器数过多时，模型参数显著增加，训练集上可以完全拟合数据（训练准确率接近100%），但容易学习到噪声和局部模式（过拟合），导致测试集表现较差。

- 使用不同池化层 (max or avg)



【分析】

综合上图可以明显地看到max_pooling好于avg_pooling，在Max Pooling中，池化操作选取池化窗口中的最大值作为该区域的输出。这种方法能保留输入特征图中最显著的特征信息。在Avg Pooling中，池化窗口内的所有值会取平均值作为输出。相比Max Pooling，平均池化对池化区域内的所有信息进行平等的关注。Max Pooling在许多任务中表现更好，尤其是在图像识别中，它能够帮助网络专注于最显著的特征，而忽略不重要的部分。Avg Pooling则更适用于那些需要保留更多全局信息或对于细节较为平滑的任务，但通常在图像任务中效果不如Max Pooling。因此，根据实验结果，Max Pooling应该是更合适的选择。

【总结】

特点	Softmax 分类器	MLP	CNN
结构	输入 → softmax	输入 → 隐层 → 输出层	输入 → 卷积层 → 池化层 → 全连接层 → 输出层
优点	简单直观，适用于简单数据	能处理复杂非线性问题	自动提取特征，适用于图像等空间数据
缺点	无特征提取能力，依赖人工特征	计算复杂，容易过拟合	需要大量数据，训练时间长
适用场景	小规模线性分类问题	中等规模的非线性问题	图像分类、目标检测等空间数据问题

- 训练效率和速度：Softmax分类器在训练时通常最为高效，因为其结构简单，无需进行复杂的特征提取。MLP和CNN都比Softmax分类器计算量更大，训练时需要更多的资源和时间，尤其是在层数较多的情况下。
- 性能：对于具有复杂结构的数据，尤其是图像数据，CNN的表现显著优于MLP和Softmax分类器。在图像分类任务中，CNN可以自动提取图像的空间特征，而MLP和Softmax分类器则无法做到这一点。
- 过拟合和泛化能力：MLP和CNN相比于Softmax分类器更容易出现过拟合，尤其是在数据量不够时。MLP对较小的数据集更容易过拟合，而CNN通过池化等操作可以更好地控制模型的复杂度，从而提高泛化能力。
- 适用性：Softmax分类器适用于简单的分类任务，如线性分类问题。MLP更适合处理复杂的非线性问题，但处理图像等具有空间结构的数据时可能不如CNN。CNN是处理图像和其他空间结构数据的首选模型，能够有效提取空间特征并进行分类。

从实验结果来看，CNN在图像分类任务中展现出了较高的准确率，MLP虽然能够处理非线性任务，但在图像数据上的表现不如CNN。Softmax分类器虽然计算效率高，但对于图像分类等复杂任务，效果有限。因此，对于涉及图像分类或具有空间结构的数据，CNN是最合适的选择，特别是在数据量较大时。如果是处理简单的非线性数据，MLP可以作为替代方案，而Softmax分类器则适合处理非常简单的线性分类问题。

【收敛速度】

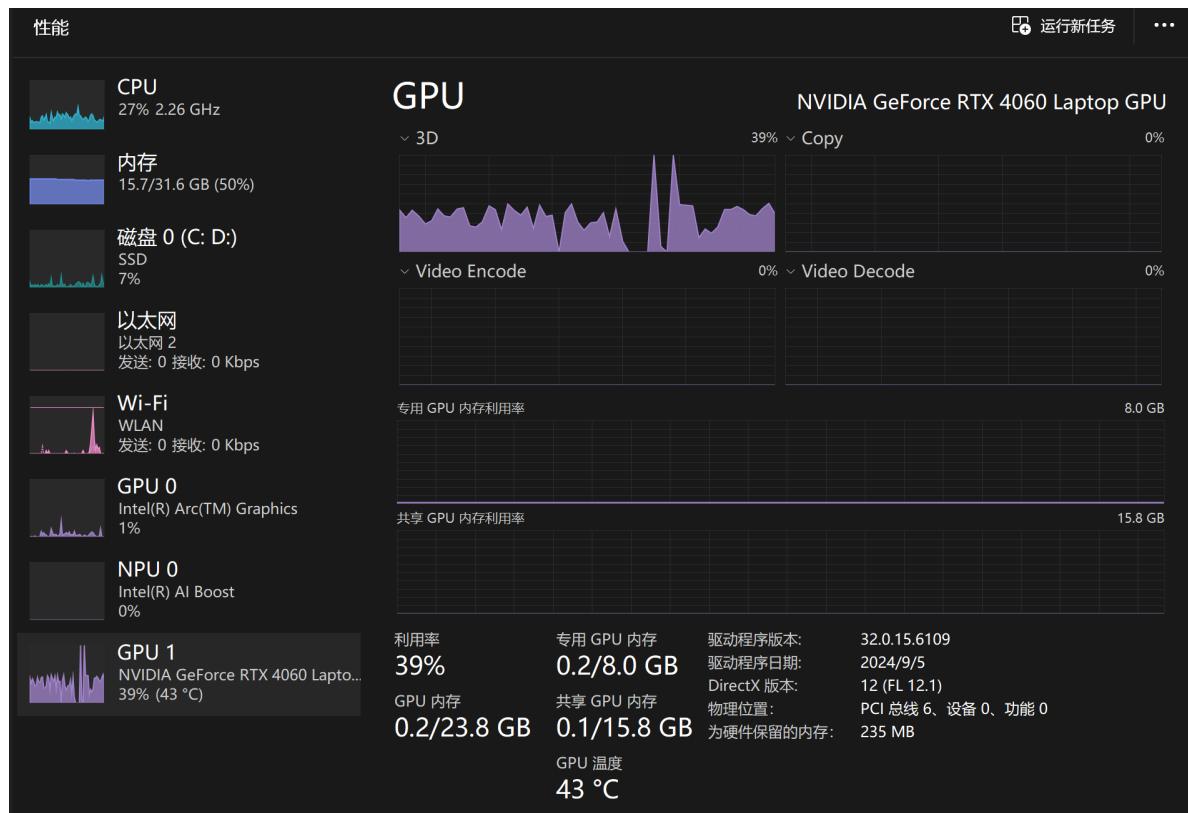
- SGD：收敛速度较慢。由于SGD每次只计算一个小批量数据的梯度更新，且没有考虑历史梯度信息，通常在训练初期会有较大的震荡，导致收敛速度较慢。在复杂的深度网络中，SGD很难在短时间内达到理想的精度，尤其是当梯度变化较大时，收敛过程可能会非常缓慢。因此，训练速度最慢，尤其在复杂的深度网络中收敛困难，训练时间最长。
- SGDM：收敛速度比SGD快。SGDM通过引入动量来加速梯度下降的过程，使得在长时间训练中不会出现震荡。动量帮助优化器更快地走向最优解，尤其是在遇到较深的局部最小值时，能够有效避免陷入不良的局部解。
- Adam：收敛速度最快。由于Adam结合了动量和自适应学习率机制，它能够快速响应参数的梯度变化，并根据不同的梯度方向自适应调整学习率，避免了SGD中常见的梯度震荡问题。Adam在训练深度网络时，通常能够更快地找到最优解，训练过程更加平稳。因此，其整体训练时间最短，尤其在处理复杂任务时，Adam能够有效缩短训练时间，尽管每次迭代计算量较大，但由于较少的训练周期，它的总体训练时间表现最好。

五、实验总结 (remark)

本次实验还是挺有挑战性的，我总共花了一周时间才完成这次实验，从前期对理论知识的研究，到后面面对模型代码的修改与调试，以及各种对比实验的训练，都耗费了大量的时间。但是经过一周的打磨，我也掌握和理解了这几个分类器其内部的原理和训练方法。上次实验完成了SVM模型，它是一个二分类器，本次实验从softmax入手多分类器，其实softmax本质是一个两层神经网络，即单层感知机，然后在其基础上又实现了多层感知机（MLP）和卷积神经网络（CNN）。从小入大，由浅入深。正如同在上面实验结果中最后写到的那样，尽管Softmax分类器在训练时较为迅速，但它的表现受限于其较简单的线性假设，因此在处理更复杂的任务时，性能较弱。相对而言，MLP和CNN可以捕捉到更多的数据特征，尤其是在面对图像这种高维数据时，CNN的表现远超其他模型，因为它能通过卷积层自动提取局部特征，并通过池化层降低维度，显著减少计算复杂度。因此，对于涉及图像数据、时间序列等具有空间或时序结构的数据，CNN通常是最优选择。不过，CNN的训练相对较慢，特别是在大规模数据集上，因为它需要处理更多的参数和更复杂的结构，且需要大量的计算资源。而MLP相较于CNN更加简单，虽然它在图像任务中的表现不如CNN，但在某些非图像任务中，尤其是数据较小或特征较简单的情况下，MLP依然能取得较好的性能。综合来看，选择何种模型不仅要考虑数据的复杂度和特性，还要权衡训练时间和计算资源。对于简单的分类问题，Softmax分类器或MLP可能足够，且能快速得到结果。而对于复杂的视觉任务或更高维度的输入数据，CNN则是更优的选择，尽管它训练较慢，但能够提供更高的准确性。因此我们在选择模型来训练某个数据集的时候需要综合考虑很多东西，最重要的当然是性能了。

我们还对不同优化器的效果进行了分析，主要包括SGD、SGDM和Adam优化器。这些优化器在训练过程中发挥了重要作用，帮助我们加速了收敛并提高了模型的性能。综合三个实验可以看到，Adam优化器结合了SGD和SGDM的优点，同时引入了自适应学习率机制，能够根据每个参数的梯度信息自适应调整学习率。这使得Adam在处理复杂问题时表现得更加稳定和高效。Adam优化器在训练过程中能够更好地应对不同特征尺度的变化，并且相比SGD和SGDM收敛速度更快。我们可以明显看出Adam优化器的表现优于SGD和SGDM。尤其在训练复杂的CNN模型时，Adam能够快速地找到较优解，并且表现出了更好的稳定性，训练过程中的波动较小，精度上升较快。

总而言之，本次实验我学到了很多东西，从理论到实践，再到调试和优化的全过程，不仅提升了自己的编程能力，也深化了对机器学习模型和算法的理解。希望我能在接下来的实验和项目中继续努力。



六、参考文献

[Softmax](#)
[CNN](#)
[MLP](#)
[SGD、SGDM、Adam](#)