机器学习实验报告

实验名称：CNN 图片分类任务

学生姓名：陆文韬

学生学号：58122231

完成日期：2024/5/9

任务描述

实现卷积神经网络 CNN，并使用 CIFAR-10 数据集进行进行图片分类任务。

CIFAR-10 是计算机视觉领域中的一个重要的数据集[1]。 原始数据集分为训练集和测试集，其中训练集包含 50000 张、测试集包含 300000 张图像。 在测试集中，10000 张图像将被用于评估，而剩下的 290000 张图像将不会被进行评估，包含它们只是为了防止手动标记测试集并提交标记结果。这些图片共涵盖 10 个类别：飞机、汽车、鸟类、猫、鹿、狗、青蛙、马、船和卡车，高度和宽度均为 32 像素并有三个颜色通道（RGB)。图 1 的左上角显示了数据集中飞机、汽车和鸟类的一些图像。本实验使用部分的 CIFAR-10 数据集，其中训练集共包含 5000 张 png 格式的图像，每个类别包含 500 张；测试集共包含 5000 张 jpg 格式的图像。

图 1 CIFAR-10 数据示例

实验原理

CNN 模型

卷积神经网络（Convolutional Neural Network，CNN）是一种前馈神经网络，专门用于处理具有网格结构的数据，如图像和视频。它通过层次化的方式提取数据的特征，从而在图像识别、目标检测等领域取得了巨大成功。

以下是 CNN 的主要组成部分的简要解释：

1. 卷积层：

- 卷积层是 CNN 的核心组件，用于提取输入数据中的特征。

- 它通过在输入数据上滑动一个卷积核（一小块可学习的参数），对数据进行局部感知和特征提取。

[1] https://www.kaggle.com/c/cifar-10

- 卷积操作的输出称为特征图（feature map），它捕捉到了输入数据的局部特征信息。

2. 激活函数：

- 激活函数是用来增加模型的非线性拟合能力的。

- 在卷积层后面通常会添加一个激活函数，如 ReLU（Rectified Linear Unit），用于引入非线性。

- ReLU 函数在正数区间上输出输入值本身，而在负数区间上输出 0，简单高效。

3. 池化层：

- 池化层用于降低特征图的空间维度，减少参数数量，从而加速计算并减少过拟合。

- 最常见的池化操作是最大池化（Max Pooling）和平均池化（Average Pooling），它们分别取池化窗口内的最大值和平均值作为输出。

- 池化操作在特征提取过程中逐渐减小特征图的尺寸，同时保留主要特征。

4. 全连接层：

- 全连接层用于将卷积层和池化层提取的特征映射到最终的输出类别上。

- 在全连接层中，每个神经元与前一层的所有神经元相连，通过学习权重来学习数据的复杂模式和关联。

5. Dropout 层：

- Dropout 层是一种正则化技术，用于减少神经网络的过拟合。

- 在训练过程中，Dropout 会随机地将一些神经元的输出置为 0，以一定的概率（通常为 0.5）丢弃这些神经元，从而减少神经元之间的依赖关系，使网络更加健壮。

- Dropout 通常应用在全连接层之间，但也可用于卷积层。

CIFAR-10 数据集

CIFAR-10（Canadian Institute for Advanced Research - 10）是一个广泛用于图像识别任务的基准数据集。它由加拿大高级研究所（Canadian Institute for Advanced Research）创建，包含了 10 个类别的彩色图像。这些类别包括:

1. 飞机（Airplane）

2. 汽车（Automobile）

3. 鸟类（Bird）

4. 猫（Cat）

5. 鹿（Deer）

6. 狗（Dog）

7. 青蛙（Frog）

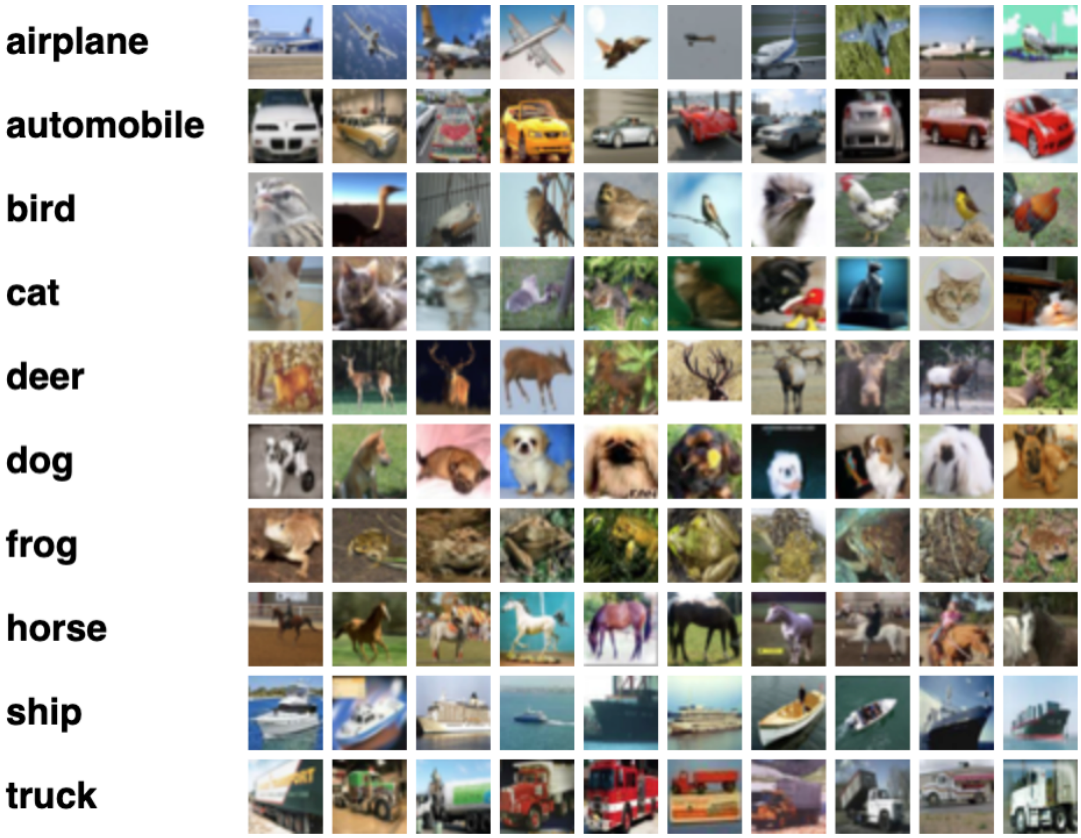
8. 马（Horse）

9. 船（Ship）

10. 卡车（Truck）

每个类别有约 6,000 张图像，总共包含 60,000 张图像。这些图像的分辨率为 32x32 像素，并分为训练集和测试集，每个集合各有 50,000 张图像。CIFAR-10 数据集是一个挑战性的数据集，适合用于测试和比较不同图像分类算法的性能。

由于图像的尺寸较小，且类别丰富多样，CIFAR-10 经常被用来测试卷积神经网络（CNN）等深度学习模型在图像分类任务上的性能。其相对较小的规模使得在普通计算机上进行实验相对容易，同时保持了足够的复杂性以测试模型的泛化能力。



数据预处理

数据读取

数据集位于目录 data/cifar10 中。它包含两个文件夹：train和test，分别包含训练集(5000 张图像)和测试集(1000 张图像)。train文件夹包含 10 个文件夹，每类图像一个。这一目录结构（每个类一个文件夹）被许多计算机视觉数据集使用，并且大多数深度学习库都提供了。
我们可以使用 matplotlib 来查看图像，但是我们需要将张量维度更改为 (32,32,3)。
在 PyTorch 中，图像的默认表示形式是（3，H，W），其中 3 表示通道数（RGB 图像为 3，灰度图像为1），H 表示高度，W 表示宽度。然而，常用的图像显示库（如 matplotlib）期望图像的维度为（H，W，3），其中3 表示RGB 通道。将图像从（3，H，W）转换为（H，W，3）是为了符合常用的图像显示库对图像维度的要求，这样我们可以正确地显示和可视化图像。

数据归一化

数据归一化是将数据按比例缩放，使之落入特定的范围。通常情况下，数据归一化指的是将数据缩放到 0 到 1 或者-1 到 1 的范围内。数据归一化在数据预处理阶段常常被用到，其目的是消除不同特征之间的量纲差异，以及使得数据分布更加符合模型训练的要求。

1. 消除量纲影响：

不同特征的量纲不同，可能会导致在模型训练过程中，由于数值大小差异过大而影响到模型的学习效果。例如，在某个特征的数值范围较大时，其权重更新会更加频繁，而其他特征的影响就会相对较小。数据归一化可以消除这种影响，使各个特征的权重更新更加平衡。

2.加速模型收敛速度：

在梯度下降等优化算法中，数据归一化可以帮助加速模型的收敛速度。因为归一化后的数据分布更接近于均匀分布，优化算法可以更快地找到全局最优解或局部最优解。

3. 提高模型稳定性：

归一化后的数据使得模型更加稳定，减少了参数更新的不确定性。这有助于防止模型在训练过程中出现梯度爆炸或梯度消失的情况，提高了模型的泛化能力。

4. 更好的模型解释性：

归一化后的数据使得特征之间的关系更加清晰，更容易解释模型的结果。因为特征之间的数值范围一致，所以模型对每个特征的重要性更容易理解。

可以使用以下的代码进行数据归一化，但是由于该代码模块不在需要我们自己编写的范围内，因此我们没有在最终的 jupyter notebook 中保留数据归一化的代码。

```
# 定义均值和标准差
mean = [0.5, 0.5, 0.5]  # CIFAR10 数据集的均值
std = [0.5, 0.5, 0.5]   # CIFAR10 数据集的标准差
# 使用 Normalize 进行数据归一化
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])
# 创建数据集时应用变换
dataset = ImageFolder(data_dir+'/train', transform=transform)
```

数据集分割

数据集分割是将一个整体的数据集分成训练集、验证集和测试集等部分的过程。这个过程通常在机器学习和深度学习任务中使用，目的是评估模型的性能和泛化能力。

1. 训练集（Training Set）：用于模型的训练，即通过训练数据集来调整模型的参数，使其能够对数据进行合适的拟合。

2. 验证集（Validation Set）：用于模型的超参数选择和调优，例如调整模型的结构、正则化参数等。验证集通常用来评估模型在训练过程中的性能，并根据性能调整模型的参数。

3. 测试集（Test Set）：用于评估模型的泛化能力，即模型对未见过的数据的表现。测试集在训练和验证阶段都没有用到，只在最终评估模型性能时使用。

在实际应用中，数据集分割的方法和比例通常根据具体问题和数据集的大小来确定。重要的是要确保训练集、验证集和测试集之间的数据没有重叠，并且能够充分地代表原始数据的分布，以保证评估结果的可靠性和泛化能力。

CNN 模型构建

本实验采用的 CNN 结构如下：

1. 卷积层（Convolutional Layers）：

第一个卷积层：输入通道为 3（RGB 图像），输出通道为 32，卷积核大小为 3x3，步长为 1，padding 为 1。

第二个卷积层：输入通道为 32，输出通道为 64，卷积核大小为 3x3，步长为 1，padding 为 1。

第三个卷积层：输入通道为 64，输出通道为 128，卷积核大小为 3x3，步长为 1，padding 为 1。

第四个卷积层：输入通道为 128，输出通道为 128，卷积核大小为 3x3，步长为 1，padding 为 1。

第五个卷积层：输入通道为 128，输出通道为 256，卷积核大小为 3x3，步长为 1，padding 为 1。

第六个卷积层：输入通道为 256，输出通道为 256，卷积核大小为 3x3，步长为 1，padding 为 1。

2. 激活函数（Activation Functions）：

ReLU 激活函数被应用在每个卷积层之后，以增加网络的非线性特性。

3. 池化层（Pooling Layers）：

- Max pooling 层被应用在每个卷积块的末尾，用于降低特征图的空间尺寸，提取出图像的主要特征。

4. 全连接层（Fully Connected Layers）：

- 两个全连接层被用于将卷积层输出的特征图转换成分类结果。

- 第一个全连接层将输入特征维度从 4096（由卷积层输出的特征图展平得到）降维到 1024。

- 第二个全连接层将维度从 1024 降维到 512。

- 最后一个全连接层将输入特征维度降维到 10，对应 CIFAR-10 数据集中的类别数量。

5. Dropout 层：

- 两个 Dropout 层被用于减少过拟合风险。它们会随机将一部分神经元的输出置零，以防止网络过度依赖某些特定的神经元。

实验设置

运行环境

平台：MacBook Air M1

GPU：Apple M1

学习率（Learning rate）

学习率是神经网络训练中的一个重要超参数，决定了模型参数（例如权重和偏

置）的更新速度。它在梯度下降及其变种算法中起着关键作用，因为它控制了梯度在每次迭代中的步长。学习率对神经网络训练有多方面的影响，包括以下几点：

影响模型收敛

收敛速度：较高的学习率可以加快训练过程，因为权重的更新步长较大。这可以帮助模型快速接近目标函数的最小值。

过快收敛或振荡：如果学习率过高，可能会导致模型在最小值附近来回振荡，难以达到稳定的收敛状态。这可能导致模型性能不稳定。

过慢收敛：较低的学习率可能导致模型收敛缓慢，训练时间过长，甚至无法收敛到最佳状态。

影响模型性能

模型精度：适当的学习率可以使模型达到最佳性能。如果学习率过高，可能会错过目标函数的最小值，从而导致模型性能不佳。

过拟合或欠拟合：学习率过低可能导致模型欠拟合，因为更新步长太小，无法充分学习数据特征。学习率过高可能导致模型过拟合，因为权重更新幅度过大，容易受训练数据噪音影响。

影响模型稳定性

梯度爆炸或梯度消失：较高的学习率可能导致梯度爆炸，因更新步长太大，导致梯度不断增大。较低的学习率可能导致梯度消失，因更新步长过小，梯度不断减小，最终趋于零。

训练中断：学习率过高可能导致模型在训练过程中不稳定，甚至出现训练中断的情况。

学习率调整策略

恒定学习率：在整个训练过程中保持固定的学习率。这种方法简单，但可能需要多次尝试才能找到合适的值。

学习率衰减：随着训练的进行，逐渐降低学习率。这样可以帮助模型在训练后期更好地收敛。

自适应学习率：如 Adam、RMSprop 等优化算法会根据梯度的变化调整学习率，确保训练过程更加稳定。

学习率的选择是神经网络训练中的关键环节，需要根据具体任务、数据集和网络结构进行调整。正确的学习率可以显著提高训练速度、模型性能和稳定性，而不合适的学习率可能导致模型训练失败或性能不佳。

学习周期（Epoch）

学习周期（epoch）是深度学习中一个重要的概念，它指的是将整个训练数据集在神经网络上进行一次完整的前向传播和反向传播的过程。在一个 epoch 中，所有的训练样本都被送入网络进行训练，并且根据损失函数计算出梯度，然后使用优化算法（如随机梯度下降）来更新网络的参数。

在每个 epoch 中，训练数据集会被划分为若干个批次（batch），每个批次包含一定数量的样本。神经网络在每个批次上进行一次前向传播和反向传播，然后更新参数。当所有批次都处理完毕后，一个 epoch 完成。

学习周期的选择通常取决于训练数据集的大小和复杂性，以及神经网络的架构和优化算法。通常情况下，一个 epoch 包含的迭代次数越多，模型的训练效果越好，但同时也会增加训练的时间成本。

一般来说，训练模型的过程是通过多个 epoch 来完成的，直到模型收敛到一个满意的状态为止。通常会监控训练过程中的指标（如损失函数的值或者验证集上的性能），并在模型停止改善或者出现过拟合时停止训练。

批大小（Batch size）

批大小（batch size）是深度学习训练过程中一个重要的超参数，它指的是每次训练时所使用的样本数量。在训练过程中，数据集会被分割成若干个批次，每个批次包含一定数量的样本，而批大小就是每个批次中包含的样本数量。

批大小的选择会影响训练的速度和模型的泛化能力，通常来说，批大小越大，训练过程中所需的内存和计算资源就越多，但训练速度通常会更快；而批大小越小，训练过程中的噪声会增加，但模型对于训练数据的梯度估计会更准确，有助于模型的泛化能力。

常见的批大小选择通常在 8 到 256 之间，具体的选择取决于数据集的大小、模型的复杂度、计算资源的限制等因素。在实践中，通常通过尝试不同的批大小，并根据模型的性能来选择最优的批大小。

值得注意的是，在训练过程中，每个批次的梯度更新只是对整个训练集的一个估计，而不是对整个数据集的准确梯度。因此，批大小的选择也会影响梯度更新的稳定性和模型的收敛速度

梯度截断（Gradient clipping）

梯度截断（Gradient Clipping）是一种神经网络训练中的技术，旨在防止梯度爆炸问题。梯度爆炸是指在反向传播过程中，梯度值变得异常大，导致模型训练不稳定，

权重更新过大，从而影响模型性能甚至导致训练失败。

梯度截断的作用

防止梯度爆炸：通过限制梯度的最大值，避免梯度在反向传播过程中变得过大。

确保训练稳定：梯度截断可以防止权重更新幅度过大，确保模型训练过程的稳定性。

提高收敛速度：由于梯度截断可以防止过大的梯度波动，确保模型在训练过程中稳定收敛。

梯度截断的实现

按值截断（Value Clipping）：直接限制梯度的绝对值，如果超过预设阈值，就将其截断至最大值。通常通过限制‘-c <= g <= c’，其中‘g’为梯度，‘c’为设定的阈值。

按范数截断（Norm Clipping）：限制梯度的范数。如果整个梯度向量的范数超过预设阈值，就按比例缩放，使其范数等于阈值。常用的范数是 L2 范数。

梯度截断的应用场景

循环神经网络（RNN）和长短期记忆网络（LSTM）：这些网络在长序列数据的训练过程中容易出现梯度爆炸问题，梯度截断可以帮助解决这一问题。

深度神经网络：深度网络容易在反向传播过程中产生较大的梯度波动，梯度截断可以提高训练稳定性。

不稳定的训练环境：当训练数据或模型结构可能导致梯度过大时，梯度截断是确保训练稳定的重要手段。

注意事项

截断阈值的选择：截断阈值不宜过小，否则可能导致梯度更新不足，影响模型的学习能力；过大则可能无法有效防止梯度爆炸。

结合其他技术：梯度截断通常与其他技术一起使用，如优化算法（如 Adam、RMSprop）和正则化方法，以确保模型训练的稳定和性能。

梯度截断是神经网络训练中的一种重要技术，特别是在处理深度网络和循环网络时，有助于防止梯度爆炸并确保训练的稳定性。正确应用梯度截断可以显著提高模型的收敛速度和性能。在本次实验中，我们选取 0.1 为梯度截断的阈值。

权重衰退（Weight decay）

权重衰退（Weight Decay）是一种常用于神经网络训练中的正则化技术，其目的是通过惩罚模型中的大权重，来防止过拟合。权重衰退在训练过程中通过在损失函数中加入一个基于权重的惩罚项来实现，这一惩罚项通常与 L2 正则化有关。

目的

防止过拟合：权重衰退可以限制权重的大小，减少模型对训练数据的过度拟合，从而提高模型在未见数据上的泛化能力。

控制模型复杂度：通过惩罚过大的权重，权重衰退可以防止模型变得过于复杂。

提高模型稳定性：权重衰退可以帮助稳定训练过程，避免梯度波动过大。

工作原理

权重衰退的核心思想是通过在损失函数中添加一个与权重有关的惩罚项，使得模型在训练过程中尽量保持权重较小。惩罚项的具体形式通常基于权重的范数，例如 L2 范数。常见的权重衰退形式是 L2 正则化。

L2 正则化：在损失函数中添加一个权重的 L2 范数的平方，即 'λ * ||W||^2'，其中 'λ' 是正则化强度，'W' 是模型的权重。这意味着在训练过程中，权重越大，惩罚就越大。

使用权重衰退

超参数调整：权重衰退中的正则化强度 'λ' 是一个超参数，需要根据具体任务和数据集进行调整。如果正则化强度过大，可能导致模型欠拟合；过小则可能无法有效防止过拟合。

结合其他正则化技术：权重衰退通常与其他正则化技术（如 Dropout）结合使用，以确保模型的稳定性和泛化能力。

应用场景

深度神经网络：权重衰退在深度网络中经常使用，帮助防止模型在训练过程中出现过大的权重。

循环神经网络（RNN）：在 RNN 中，权重衰退可以帮助控制网络的复杂度，避免过拟合。

卷积神经网络（CNN）：在 CNN 中，权重衰退有助于防止卷积层中的大权重。

权重衰退是一种强大的正则化工具，可以帮助防止过拟合，提高模型的泛化能力，同时确保训练过程的稳定性。通过在损失函数中加入权重惩罚项，权重衰退可以有效控制模型的复杂度，从而提高模型在实际应用中的表现。

优化器

随机梯度下降（SGD）

SGD 是深度学习中常用的优化算法，全称为随机梯度下降（Stochastic Gradient Descent）。它是梯度下降算法的一种变体，用于更新神经网络参数以最小化损失函数。

在标准的梯度下降算法中，每次更新参数时都需要计算整个训练数据集的损失函数梯度，然后根据该梯度更新参数。这种方法的缺点是计算成本较高，特别是在大规模数据集上训练时。SGD 通过随机选取一个样本或一个小批量样本来估计损失函数的梯度，然后根据该梯度来更新参数，从而降低计算成本。

SGD 的更新规则如下：

1. 从训练集中随机选择一个样本或一个小批量样本。

2. 计算选择样本的损失函数梯度。

3. 使用梯度对参数进行更新，通常是减去梯度乘以学习率的大小。

SGD 算法的优点包括：

- 计算成本较低，特别适用于大规模数据集和大型神经网络。

- 在噪声较多的情况下，随机选择样本可以帮助避免陷入局部最优解。

然而，SGD 也存在一些缺点：

- 更新的方向不一定是最优的，可能会出现震荡现象，影响模型的收敛速度。

- 对学习率敏感，学习率选择不当可能导致训练过程不稳定或者收敛速度过慢。

因此，通常会使用 SGD 的改进版本，如带动量的 SGD、AdaGrad、RMSProp 和 Adam 等，以解决 SGD 存在的一些问题，并提高模型的训练效果和稳定性。

以下是一个简单的 SGD 的实现：

```
def sgd_optimizer(parameters, lr):
    for param in parameters:
        param.data -= lr * param.grad.data
```

Adam

Adam（Adaptive Moment Estimation）是一种常用的自适应优化算法，结合了动量法和自适应学习率的特点，被广泛应用于深度学习中。Adam 在优化过程中动态调整每个参数的学习率，并且考虑了每个参数的一阶矩估计（即梯度的均值）和二阶矩估计（即梯度的方差），以实现更准确的参数更新。

Adam 算法的主要思想如下：

1. 维护两个动态调整的一阶矩估计和二阶矩估计，分别用 m 和 v 表示，它们的初始值通常为 0。

2. 在每个时间步 t，计算当前 mini-batch 的梯度 g_t。

3. 更新一阶矩估计 m_t 和二阶矩估计 v_t，具体为：

   - 计算一阶矩估计的移动平均值：m_t = beta_1 * m_{t-1} + (1 - beta_1) * g_t

   - 计算二阶矩估计的移动平均值：v_t = beta_2 * v_{t-1} + (1 - beta_2) * (g_t)^2

   其中，beta_1 和 beta_2 是控制一阶矩估计和二阶矩估计权重的超参数，通常设置为接近于 1 的值。

4. 根据一阶矩估计和二阶矩估计的修正值，更新参数：

   - 参数更新量：delta_theta = - learning_rate * m_t / (sqrt(v_t) + epsilon)

   - 更新参数：theta_{t+1} = theta_t + delta_theta

   其中，learning_rate 是学习率，epsilon 是为了防止除零错误而添加的小值。

Adam 算法的优点包括：

- 对参数的学习率进行自适应调整，使得不同参数具有不同的学习率，有利于优化过程的收敛。

- 结合了动量的思想，有助于处理优化过程中的梯度变化较大或者损失函数非凸的情况。

Adam 算法已经被广泛地应用于深度学习中，并且在许多任务中表现出良好的性能和收敛速度。

以下是一个 Adam 算法的简单实现：

```
class AdamOptimizer:
    def __init__(self, parameters, lr=0.001, beta1=0.9, beta2=0.999,
epsilon=1e-8):
        self.parameters = list(parameters)
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.t = 0
        self.m = [torch.zeros_like(param) for param in self.parameters]
        self.v = [torch.zeros_like(param) for param in self.parameters]
```

```
    def step(self):
        self.t += 1
        for i, param in enumerate(self.parameters):
            grad = param.grad.data
            self.m[i] = self.beta1 * self.m[i] + (1 - self.beta1) * grad
            self.v[i] = self.beta2 * self.v[i] + (1 - self.beta2) * (grad **
2)
            m_hat = self.m[i] / (1 - self.beta1 ** self.t)
            v_hat = self.v[i] / (1 - self.beta2 ** self.t)
            delta_theta = -self.lr * m_hat / (torch.sqrt(v_hat) +
self.epsilon)
            param.data += delta_theta
```

Adagrad

Adagrad（Adaptive Gradient Algorithm）是一种自适应学习率优化算法，旨在解决传统梯度下降算法中学习率调整不合理的问题。Adagrad 的主要思想是根据参数的历史梯度信息来调整学习率，对频繁出现的参数进行较小的更新，对不经常出现的参数进行较大的更新，从而实现自适应调节。

Adagrad 算法的关键点在于维护一个参数的历史梯度的平方和，并将这个平方和作为学习率的分母。具体来说，Adagrad 算法的步骤如下：

1. 初始化参数的历史梯度平方和为零。
2. 在每个时间步 t，计算当前 mini-batch 的梯度 g_t。
3. 累积参数的历史梯度平方和：
   - 参数历史梯度平方和更新：r_t = r_{t-1} + g_t^2
4. 根据历史梯度平方和来调整学习率：
   - 学习率更新：lr_t = lr / sqrt(r_t + epsilon)
5. 使用调整后的学习率进行参数更新：
   - 参数更新量：delta_theta = - lr_t * g_t
   - 更新参数：theta_{t+1} = theta_t + delta_theta

在 Adagrad 算法中，参数的历史梯度平方和会随着时间的推移而增加，导致学习率不断减小，从而保证了训练过程中学习率的自适应性。这种自适应性使得 Adagrad 在训练初期能够较快地收敛到最优解，但随着训练的进行，学习率的衰减可能会导致模型的学习速度变慢，甚至停滞不前。

Adagrad 算法的一个问题是学习率的衰减可能过快，导致后续训练过程中学习率过

小，使得参数更新速度变慢。为了解决这个问题，后续出现了一些改进算法，如 RMSProp 和 Adam 等，它们对 Adagrad 进行了一些调整，以提高算法的性能和稳定性。

以下是一个 Adagrad 算法的简单实现：

```
class AdagradOptimizer:
    def __init__(self, parameters, lr=0.01, epsilon=1e-8):
        self.parameters = list(parameters)
        self.lr = lr
        self.epsilon = epsilon
        self.gradient_squared_sum = [torch.zeros_like(param) for param in
self.parameters]

    def step(self):
        for i, param in enumerate(self.parameters):
            grad = param.grad.data
            self.gradient_squared_sum[i] += grad ** 2
            lr_t = self.lr / torch.sqrt(self.gradient_squared_sum[i] +
self.epsilon)
            delta_theta = -lr_t * grad
            param.data += delta_theta
```

实验结果

不同批处理大小对模型的影响

在本次实验中，我们固定学习周期 Epoch 为 30，learning rate 为 0.005，优化器为 Adam，采用不同的 batch size 进行训练。我们可以观察到不同的批处理大小（batch size）对卷积神经网络（CNN）训练过程中准确度（accuracy）的影响。具体分析如下：

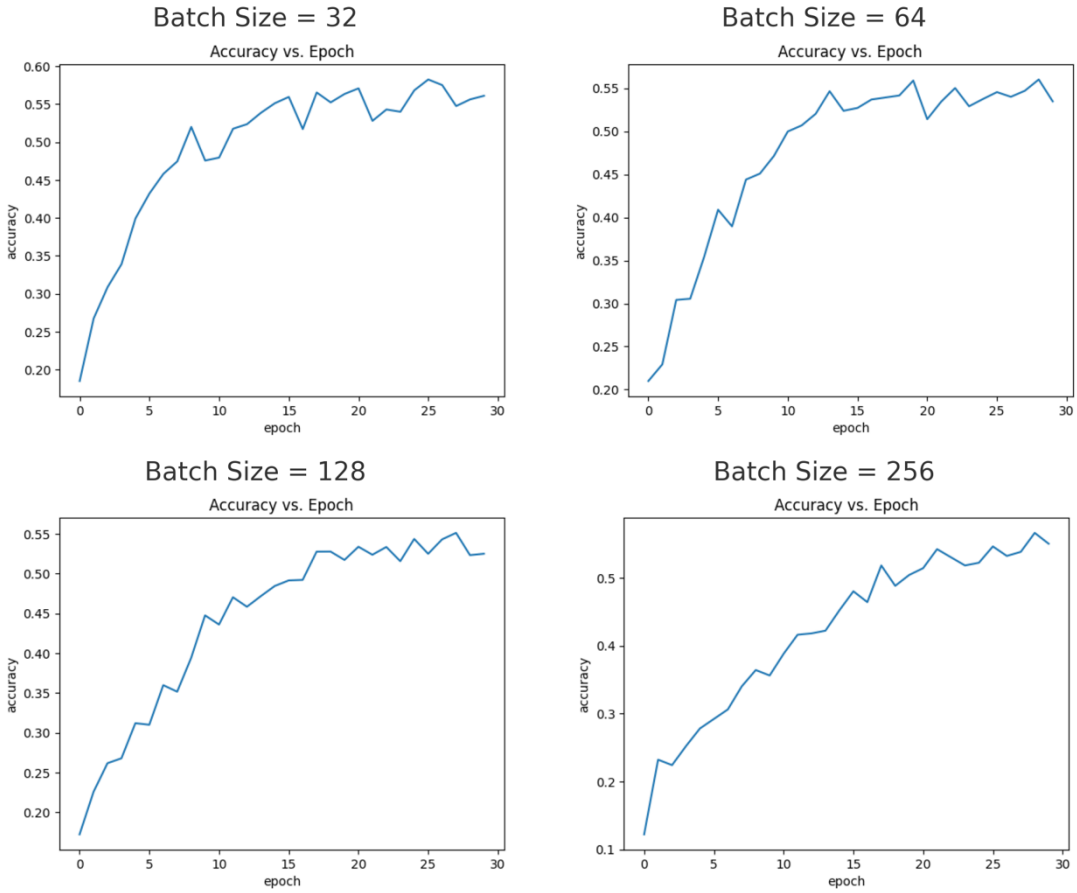
1. Batch Size = 32：这是较小的批处理大小，其准确度提升较快，并在训练过程中比较早地达到较高的稳定性。这表明小批量可以提供更快的收敛速度，并可能因为频繁的权重更新而更快地逼近全局最优解。

2. Batch Size = 64：与 32 相比，批量稍大，准确度提升的速度依然很快，整体趋势平稳，并在训练过程中展现出较好的性能。

3. Batch Size = 128：这个批量大小的准确度提升速度稍慢于前两者，但最终达到的准确度与批量为 64 时相近，显示出较好的稳定性和可靠性。

4. Batch Size = 256：这是最大的批处理大小，其准确度提升的速度最慢，但随着训练的进行，准确度逐渐提升，显示出在后期还有提升空间的可能性。

总结来说，较小的批处理大小能够加快训练的收敛速度，并在训练初期快速提升模型的准确度。但随着批处理大小的增加，每一次迭代更新的稳定性提高，准确度提升的速度可能会减慢。在实际应用中，选择适合的批处理大小需要考虑训练速度、资源消耗和最终模型的性能等多个因素。在本次实验中，我们最终选用了 batch size 为 64 进行训练，来获得较好的效果。



从下面这张图片中，我们可以观察到不同批处理大小（batch size）对卷积神经网络（CNN）训练过程中损失（loss）的影响。蓝色曲线代表训练损失（training loss），而橙色曲线代表验证损失（validation loss）。

1. Batch Size = 32：

- 训练损失较快下降，显示模型能够有效学习训练数据。

- 验证损失在某一点后开始增加，这可能是过拟合的标志，即模型对训练数据过度优化，而在未见数据上的表现较差。

2. Batch Size = 64：

- 训练损失持续稳定地下降。

- 验证损失在下降后趋于平稳，稍后有轻微上升，这同样表明可能出现轻微的过拟合现象。

3. Batch Size = 128：

- 训练损失下降较为平稳，比较慢。

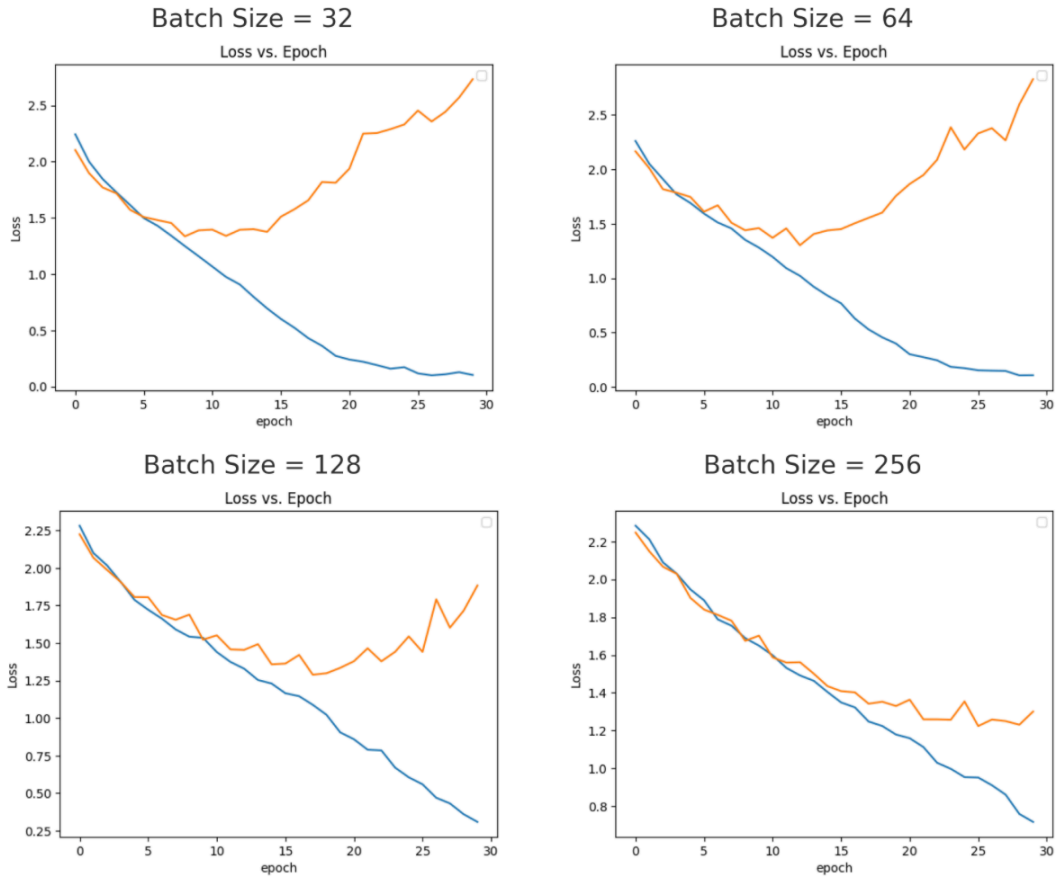
- 验证损失总体趋势也是下降，中途有波动但未显著上升，表明模型在验证集上保持了一定的泛化能力。

4. Batch Size = 256：

- 训练损失缓慢而稳定地下降，显示出良好的收敛性。

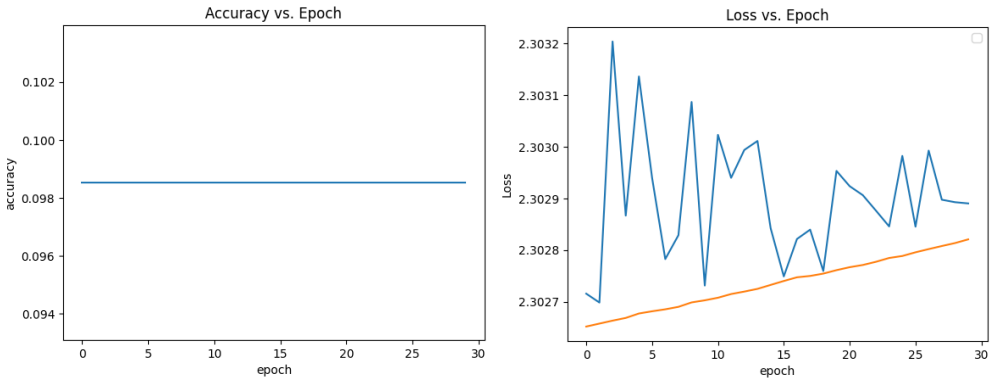
- 验证损失在初期下降后趋于稳定，末端稍有上升，表明模型的泛化能力相对较好。

一般来说，较小的批处理大小可能会导致模型快速学习但容易过拟合，而较大的批处理大小虽然训练速度慢，但更加稳定，泛化能力较强。在实际应用中，合适的批处理大小取决于具体任务的需求和数据的特性，通常需要通过实验来确定最优批处理大小，以平衡训练速度和模型性能。



不同优化器对模型的影响

SGD



从以上的两张图中，我们可以看到使用随机梯度下降（SGD）优化器，学习率为 0.005，在批处理大小为 64，训练 30 个 epoch 的条件下，模型的表现有以下特点：

1、准确率图显示准确率基本不变，呈水平线状态，几乎没有改善。这种情况表明模型在训练过程中几乎没有学到有效信息，或者学习过程陷入了停滞。

2、损失图显示出训练损失（蓝线）波动较大，而验证损失（橙线）则逐渐增加。这种大幅波动可能指示了训练过程的不稳定性，而验证损失的逐步增加则可能表示模型对未见数据的泛化能力较差。

使用 SGD 导致这种问题的可能原因包括：

1. 学习率设置不当：

学习率 0.005 可能对当前任务来说过高或过低。如果学习率过高，模型权重的更新可能太大，导致训练不稳定；如果过低，则可能使得模型学习过程非常缓慢，难以在有限的 epoch 内达到较好的性能。

2. SGD 的固有特性：

SGD 以其高方差闻名，每次仅使用一个批次的数据来更新模型，这可以导致损失曲线波动很大。这种高方差可能使模型难以收敛到最优，特别是在复杂的模型或数据集上。

3. 过拟合：

尽管损失图中验证损失逐渐增加可能表明过拟合，但准确率图显示模型几乎未学到有效信息，这更可能是模型根本就没有正确学习到数据特征。

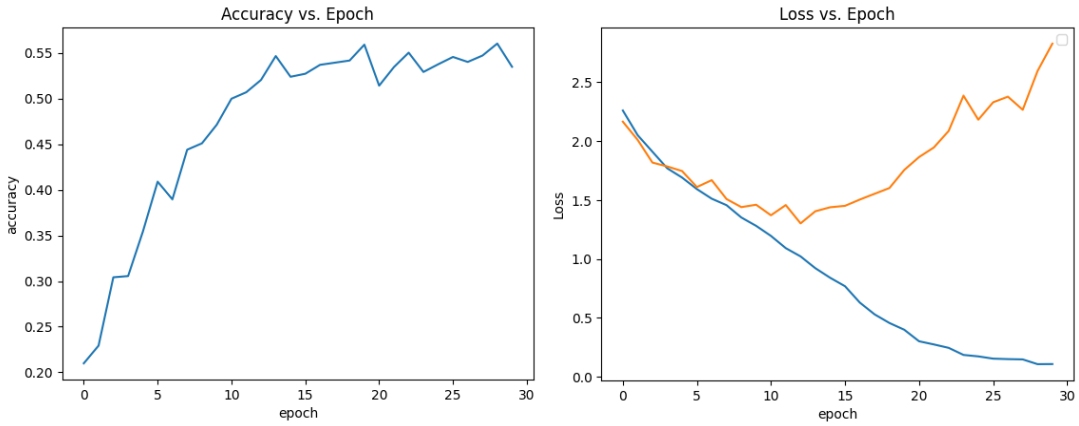
解决这些问题的建议措施包括：

1、调整学习率：尝试减少学习率，观察损失和准确率的变化，或者使用学习率调度器在训练过程中动态调整学习率。

2、使用动量或其他优化器：考虑使用带动量的 SGD 或更高级的优化器如 Adam，这些优化器可以帮助稳定训练过程并加快收敛。

3、增加正则化：如 L2 正则化、dropout 等，以减少模型的过拟合风险。

Adam



我们可以看到采用 Adam 优化器时，在学习率为 0.005、批处理大小为 64，进行 30 个 epoch 训练时的效果：

1. 准确率（Accuracy）：

图中显示了准确率从最初的大约 0.2 逐步提升到超过 0.5 的稳定趋势。模型的性能明显改善，准确率在训练过程中稳步增长，并在后期维持在较高水平，表明模型对训练数据具有良好的学习能力。

2. 损失（Loss）：

训练损失（蓝线）从初始的高值持续下降，表明模型在训练过程中不断优化其预测能力。这种持续的下降趋势显示了模型在逐渐减少误差，增强其预测准确性。

验证损失（橙线）在训练初期也跟随下降，但在中期开始呈现波动，并在训练的后期明显上升，这可能是过拟合的表现，即模型对训练数据过度适应，而在未见数据上表现得不够好。

分析和建议：

模型优化和准确率提升：

模型的准确率稳定提升表明 Adam 优化器在调整权重时较为有效，能够帮助模型捕捉到数据中的复杂模式。Adam 优化器通过结合动量和自适应学习率技术，能有效地

解决 SGD 在非凸优化问题中的一些挑战。

关于损失的波动与上升:

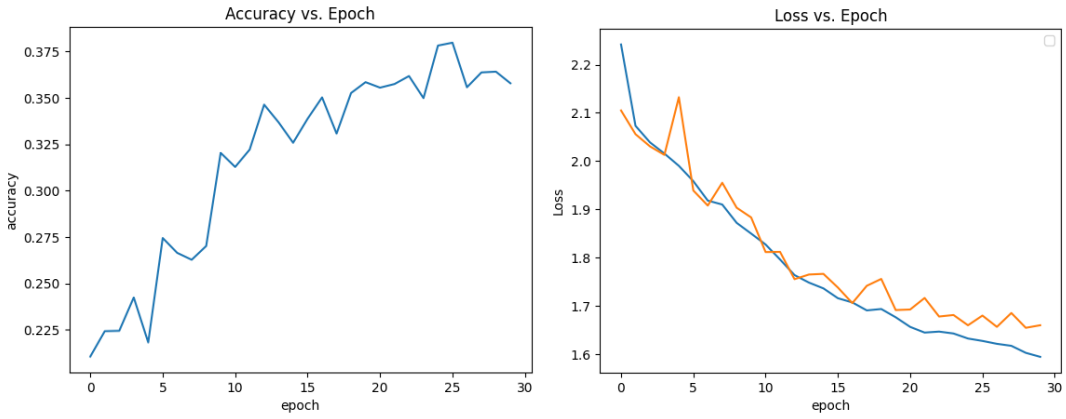
损失的波动与上升可能是因为学习率相对较高所导致的。尽管 Adam 能自适应调整学习率，但基础学习率的设定仍然非常关键。

模型可能在训练数据集上过拟合，导致在验证数据集上损失增加。考虑加入正则化方法（如 L2 正则化、Dropout）或提前停止来控制过拟合。

可以尝试减小学习率或使用学习率衰减策略，以减少训练过程中的损失波动，并尝试提高模型对验证集的泛化能力。

综上所述，虽然模型在准确率上表现良好，但损失的增加提示我们可能需要对模型进行进一步调优，以避免过拟合并改善模型在未见数据上的表现。在本次实验中，我们发现采用更大的 batch size 或者采用比 0.005 更小的学习率能有效防止模型过拟合。

Adagrad



使用 Adagrad 优化器、学习率 0.005、批处理大小 64，并进行 30 个 epoch 的训练的 CNN 模型的准确率和损失图，我们可以得到以下结果:

1. 准确率（Accuracy）:

准确率图显示了从大约 0.225 开始，逐渐上升至大约 0.375 的过程。整个训练过程中，准确率的增长趋势较为稳定，尽管在中后期存在一些波动，总体表现出明显的上升趋势，说明模型在不断改进并学习到有效的特征。

2. 损失（Loss）:

训练损失（蓝线）从较高的初始值开始迅速下降，之后趋势平缓，但整体持续降低，这表明模型在减少预测错误上取得了稳定的进步。

验证损失（橙线）在初始阶段下降后，展示出一些波动，并在中期稍微上升，然后

又开始逐渐下降。这种波动可能指示了模型在某些阶段对训练集的学习已趋于饱和，需要调整以改善泛化能力。

使用 Adagrad 优化器的分析：

Adagrad 的特点：Adagrad 优化器特别适用于处理稀疏数据，它通过对每个参数采用不同的自适应学习率，有效地提高了模型在面对不同特征的灵活性。这种方法可以帮助模型在早期迅速学习重要的特征。

潜在的挑战：Adagrad 的学习率随着训练的进行逐渐降低，这可能导致后期学习速度过慢，从而影响模型在训练结束前达到最优状态的能力。

建议：

调整学习率或优化器：如果模型的学习进展在后期明显减慢，可以考虑增加基线学习率或切换到其他自适应学习率的优化器，如 Adam 或 RMSprop，这些优化器可能更适合持续的性能提升。

增加正则化和早停：以防过拟合，并保持模型对新数据的泛化能力。例如，可以使用 Dropout 或 L2 正则化，并通过监控验证损失来实施早停策略。

数据增强和特征工程：进一步提高模型的稳健性和准确性，特别是在准确率仍有提升空间的情况下。

由以上分析可知，Adagrad 优化器在本例中表现出了稳定的学习能力，尽管在后期可能需要调整以保持学习效率和避免泛化性能下降。