机器学习实验报告

实验名称：建立全连接神经网络

学生姓名：陆文韬

学生学号：58122231

完成日期：2024/4/23

任务描述

通过两种方式实现全连接神经网络，并对图片分类任务行进行测试与实验。

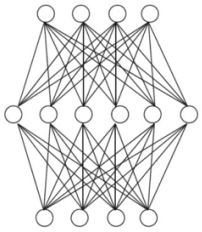
1. 手动实现简单的全连接神经网络
2. 使用 Pytorch 库简洁实现全连接神经网络

Fashion-MNIST 图片分类数据集包含 10 个类别的时装图像，训练集有 60,000 张图片，测试集中有 10,000 张图片。图片为灰度图片，高度（h）和宽度（w）均为 28 像素，通道数（channel）为 1。10 个类别分别为：t-shirt(T 恤), trouser（裤子）, pullover（套衫）, dress（连衣裙）, coat（外套）， sandal（凉鞋），shirt（衬衫），sneaker（运动鞋），bag（包），ankle boot（短靴）。使用训练集数据进行训练，测试集数据进行测试。

实验原理

单隐层全连接神经网络

采用单隐层全连接神经网络进行本次试验，神经网络包括三层结构：输入层，节点个数要与输入图片的像素数量大小匹配，因为 Fashion-MNIST 数据集中的图片均采用 28*28 的格式，所以输入层应该有 784 个节点构成。隐藏层，节点个数可以根据自己的需要和实际训练得到的结果进行判断，选择一个合理的隐藏层节点个数，既可以获得较好的训练效果又可以节约训练所需的时间成本。输出层，由于我们采用神经网络来执行一个分类任务，因此我们要求输出层的节点个数与数据集中所含类别的数量相等，在输出时，每个节点的输出在 softmax 之后对应于该样本从属于该类别的概



率，从而去对样本进行分类。

Fashion-MNIST 数据集

内容

Fashion-MNIST 是一个经典的图像分类数据集，包含 了 10 个类别共 7000 张 28*28 像素的灰度图像，每个类别包含了 6000 张训练样本和 1000 张测试样本，这个数据集的目的是代替 MNIST 数据集，以便更好地评估图像分类算法的性能。具体内容如下：

·t-shirt(T 恤)

·trouser（裤子）

·pullover（套衫）

·dress（连衣裙）

·coat（外套）

·sandal（凉鞋）

·shirt（衬衫）

·sneaker（运动鞋）

·bag（包）

·ankle boot（短靴）

图像特征

Fashion-MNIST 数据集中的每个图像的尺寸都是 28*28 像素，而且和传统的具有 RGB 三通道的彩色图像相比，灰度图像只有一个通道，无疑极大简化了要训练的模型的复杂度，每个图像可以非常简单的表示为一个 28*28 的矩阵，一共 784 个特征。

训练集和测试集

Fashion-MNIST 数据集已经被划分为了训练集和测试集。训练集包含 60000 张图

像，其中每个类别有 6000 张图像。测试集包含 10000 张 taxing，用于评估算法在未见过的数据上的性能。

数据预处理

数据增强

神经网络不管是卷积神经网络还是全连接层神经网络，在执行计算机视觉的相关问题时都会出现的共同问题是，在原先的图像核心不变的情况下，模型会误以为翻转，旋转裁剪等变换操作之后的图像与原先的不同。而如果简单的对原来的图像进行训练拟合的话，很容易出现严重的过拟合现象。因此，通过数据增强，对原始的训练数据进行一系列的变换操作，从而产生更多的训练样本，来增加数据的多样性。数据增强可以有效地扩展训练集规模，减轻过拟合问题，并提高模型的泛化能力。

常见的数据增强技术包括:

图像翻转：将图像水平或垂直翻转。

图像旋转：将图像按照指定角度进行旋转。

图像裁剪：随机或固定位置裁剪图像的部分区域。

图像缩放：改变图像的大小，可能是缩小或放大。

图像平移：将图像沿某一方向移动。

添加噪声：在图像上添加随机噪声。

颜色调整：改变图像的亮度、对比度、饱和度等。

模糊和锐化：通过滤镜应用模糊或锐化效果。

随机擦除：在图像上随机擦除某些部分。

混合图像：将两幅图像合成，以创建新样本。

数据归一化

数据归一化是数据预处理中非常重要的一步，它通过缩放和标准化，将数据调整到相似的范围和统计特性。归一化有助于确保模型训练的稳定性和一致性。以下是归一化的几个关键原因:

1. 梯度下降算法：很多机器学习模型在训练过程中使用梯度下降算法来调整参数。这个算法对数据的规模和范围非常敏感。如果没有进行归一化，可能会导致梯度更新过快或过慢，从而影响模型的收敛速度和稳定性。

2. 特征权重：在某些模型中，如支持向量机（SVM）和 K 近邻（KNN），特征的尺度差异会影响模型的学习和预测。若某些特征的尺度较大，它们可能会对模型产生过度影响，忽略了其他较小特征。通过归一化，可以确保每个特征的权重更加均衡。

3. 模型收敛和稳定性：对于神经网络等深度学习模型，数据归一化有助于加快收敛速度。归一化使每个特征的范围相似，减少了梯度的波动，从而提高模型的稳定性。

4. 避免数值计算问题：在进行数值计算时，如果数据具有不同的尺度，可能会出现数值溢出或下溢的情况。归一化将数据控制在合理的范围内，可以避免这些数值计算问题。

5. 提高模型的泛化能力：数据归一化有助于降低特征间的相关性，提高特征的独立性，从而提高模型在未见数据上的泛化能力。

这些因素使得数据归一化成为数据预处理中不可或缺的步骤。在训练模型之前，通常会对输入数据进行归一化，以确保模型的训练效果、稳定性和泛化能力。不同数据集有不同的统计特性，所以需要单独计算它们的平均值和标准差。在 Fashion-MNIST 数据集上，平均值和标准差分别是 0.1307 和 0.3081。实践中，可以使用 torchvision.transforms 中的 Normalize 进行归一化。

模型原理

单隐层全连接神经网络是一种最基本的神经网络结构，包括一个输入层，一个隐藏层和一个输出层，其中输入层的节点数对应于图像的像素数量，为 784，隐藏层的节点数设置为 256，但他是一个可调节的参数，可以根据实际训练的需求和训练的效果进行灵活的调整，同时，隐藏层节点数一般采用 2 的整数次幂，因为考虑到计算机的结构问题，这样的设置可以最大限度利用计算机的结构特性，在取得良好训练效果的同时，还能加快训练的速度。输出层节点数为 10，对应数据集中的 10 个类别。

在全连接神经网络中，每个节点都与前一层中的所有节点相连接，每个节点都有一个权重(weight)和一个偏置(bias)，这两个参数分别决定了节点的重要性和激活的阈值。隐藏层节点通过激活函数（在本次实验中采用了 ReLU 激活函数）将输入的信号进行加权求和之后，输出一个非线性的结果，输出层的节点也通过激活函数（在本次实验中采用了 softmax 激活函数）将输入信号加权求和之后，输出一个概率分布，表示输入图像属于每个类别的概率。

任务一：手动实现单隐层全连接神经网络

正向传播

输出的计算过程如下：

- 首先，输入数据 x 转换为 $z^{(0)}$，然后计算第一个隐藏层的激活值 $h^{(1)}$：
  - $z^{(1)} = W^{(1)}x + b^{(1)}$,
  - $h^{(1)} = \text{relu}(z^{(1)})$,
- 接着计算输出层的激活值 $h^{(2)}$：
  - $z^{(2)} = W^{(2)}h^{(1)} + b^{(2)}$,
  - $h^{(2)} = \text{softmax}(z^{(2)})$.

反向传播

输出层误差的计算方式为：

- $\delta^{(2)} = h^{(2)} - y$,

而隐藏层误差的计算过程为：

- $\delta^{(1)} = \text{relu}'(z^{(1)}) \circ (\delta^{(2)} \cdot (W^{(2)})^T)$,

其中 ReLU 的导数为：

- $\text{relu}'(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{if } x \leq 0. \end{cases}$,

梯度的计算方法如下：

- $\nabla_{W^{(2)}} L = (h^{(1)})^T \cdot \delta^{(2)}$,
- $\nabla_{b^{(2)}} L = \delta^{(2)}$,
- $\nabla_{W^{(1)}} L = (x)^T \cdot \delta^{(1)}$,
- $\nabla_{b^{(1)}} L = \delta^{(1)}$.

更新各层参数

- $W^{(t)} \leftarrow W^{(t)} - \eta \nabla_{W^{(t)}} L$,
- $b^{(t)} \leftarrow b^{(t)} - \eta \nabla_{b^{(t)}} L$,

其中 η 是学习率。

任务二：使用 PyTorch 库简洁实现全连接神经网络

由三层神经网络组成：

1、**展平层**：将 28*28 的图像展平成 784 个神经元输入
2、**全连接层**：784 个节点，ReLU 激活函数
3、**输出层**：10 个节点，softmax 激活函数

实验设置

运行环境

平台：MacBook Air M1

GPU：Apple M1

Pytorch 的 dev 版本中新增了对 Apple M 系列芯片的 GPU 的支持，类似于将模型和数据装载到 CUDA 的步骤，将其装载到 mps 上用 GPU 进行计算可以大幅加快训练速度。同时 M 系列芯片采用统一内存架构，将传统内存和显存合并统一，省去了数据 IO 的步骤，进一步加快了训练速度。

隐藏层节点数

对于单隐层全连接神经网络来说，隐藏层节点数是一个可以调节的参数，并且会对最终的预测结果产生影响，隐藏层的节点数指定了神经网络中隐藏层的神经元数量。

神经网络隐藏层节点数对模型性能有多方面的影响，包括以下几点：

模型复杂度：隐藏层的节点数越多，神经网络的容量就越大。这可能会提高模型的表达能力，帮助其更好地捕捉复杂的数据模式。

过拟合风险：节点数增加可能会导致过拟合，特别是在数据量相对较小的情况下。过拟合时，模型可能会过于专注于训练数据中的噪音，而不是通用的模式。

训练时间和计算资源：节点数增多会增加训练和推理的计算成本，因为更多的节点需要计算和存储。这可能需要更长的训练时间和更高的硬件性能。

模型收敛：过多的隐藏层节点可能会导致训练收敛不稳定，或者需要更多的正则化措施。节点数过少则可能导致模型难以充分学习数据特征。

泛化能力：隐藏层节点数适中时，模型更可能具有较好的泛化能力，即在未见数据上表现出色。如果节点数过多，模型可能仅适用于训练数据而难以泛化。

参数初始化

在进行参数初始化的实验中，容易出现梯度消失和梯度爆炸的情况，导致大部分反向传播得到的梯度不起作用或者起反作用。因此，找到一种好的权重初始化方法，让网络前向传播或者反向传播的时侯，卷积的输出和梯度比较稳定，是非常重要的。

下面介绍几种常见的权重初始化的方法:

1、 Zero Initialization

零初始化是一种简单且直观的权重初始化方法，其中神经网络的所有权重在初始阶段被设定为零。这意味着在训练开始时，每个神经元的输出都是相同的，因为没有权重差异导致不同的输出。尽管这种方法易于理解，但在实际应用中，通常不被推荐。原因如下:

对称性问题：零初始化会导致对称性问题。在神经网络的每一层中，所有神经元在初始阶段都会产生相同的输出。这意味着在梯度下降的过程中，所有权重的更新方式完全相同，导致无法区分各个神经元的作用。结果是，网络在整个训练过程中会保持这种对称性，无法学到有效的特征。

无效的训练：由于上述对称性问题，神经网络在训练过程中无法有效学习。这会导致网络在多层结构中无法展开，最终只会形成一组等效于单个神经元的权重，而不是实际多层网络的复杂功能。

训练收敛困难：零初始化容易导致网络在训练过程中陷入困境，因为梯度更新无法引入足够的随机性和多样性。这可能使得训练时间过长，甚至无法达到期望的收敛效果。

尽管零初始化在一些特殊情况下可能会被用于特定用途，例如某些正则化或测试用例，但在通常情况下，它并不适合用于实际的神经网络训练。

2、 Constant Initialization

常数初始化是一种权重初始化方法，其中神经网络的所有权重被设定为一个固定的常数值。这种初始化方法非常简单，但通常在神经网络训练中不被推荐，原因与零初始化有相似之处。

固定权重：在常数初始化中，每个权重被赋予一个相同的常数值。这意味着在训练开始时，神经网络的所有权重具有相同的值。

易于理解：由于所有权重都是相同的固定值，常数初始化方法简单易懂，且实现起来

容易。

对称性问题：与零初始化类似，常数初始化容易导致对称性问题。在神经网络的每一层中，所有神经元的输出都是相同的，因为权重没有任何多样性。这意味着在梯度下降的过程中，所有权重的更新方式完全相同，导致网络无法学习复杂的特征。

训练收敛缓慢或失败：由于对称性问题，常数初始化可能导致网络在训练过程中无法收敛或收敛缓慢。这是因为梯度更新没有足够的随机性来打破对称性，使得模型难以学习。

无法学习有效特征：由于所有神经元的输出相同，网络在整个训练过程中无法学到有效特征。这会导致模型在实际应用中性能不佳。

常数初始化在某些特定情况下可能会被使用，例如：

偏置项的初始化：在一些情况下，常数初始化可用于初始化神经网络的偏置项，通常是用一个小的正数值。这样做是为了确保神经元在初始阶段有一定的激活水平。

用于特定测试：在实验环境中，常数初始化可以用于测试模型对权重初始化的敏感性，或者作为一种控制条件。

3、 Random Initialization

随机初始化是一种神经网络权重初始化的方法，通过随机生成权重以赋予网络一定的随机性。这种方法通常用于避免神经网络训练过程中的对称性问题，并确保梯度在整个网络中有效传播。

随机初始化的核心在于权重的随机性。通过给每个权重赋予随机值，网络在训练过程中可以有效地学习多样化的特征，而不会因对称性而陷入困境。随机初始化通常使用均匀分布或正态分布来生成权重。

随机初始化的实现方式可以根据权重的分布类型和范围来选择：

均匀分布：权重从一个指定范围内的均匀分布中采样，通常范围可以是 '[-a, a]'，其中 'a' 是一个预定义的值。这个范围的选择影响网络的初始随机性。

正态分布：权重从一个正态分布中采样，通常以零为中心，并设定标准差。标准差的选择会影响权重的离散程度。

打破对称性：随机初始化的最大优点是打破了神经网络中的对称性。每个神经元的权重不同，使得网络能够学习多样化的特征。

增加随机性：随机初始化引入了一定的随机性，有助于模型训练过程中避免局部最优，促进模型收敛。

广泛适用：随机初始化可以用于各种神经网络，包括前馈神经网络、卷积神经网络和

循环神经网络。

无法确保稳定性：如果权重初始化范围过大，可能导致梯度爆炸。如果范围过小，可能导致梯度消失。这需要根据网络结构和激活函数的特性来调整。

需要优化：随机初始化通常不是一次性设定，而是需要经过实验和调整，以确保训练的稳定和效率。

初始化范围：选择合适的初始化范围是关键。如果过大，可能导致网络收敛困难；如果过小，可能导致网络无法有效学习。

激活函数：根据网络中使用的激活函数，选择适当的随机初始化方法。如果使用 Sigmoid 或 Tanh，范围应适中；如果使用 ReLU 或其变种，范围可以稍大一些。

结合正则化：随机初始化可能需要与正则化方法结合使用，以确保网络的稳定性和防止过拟合。

随机初始化在神经网络训练中广泛使用，因为它可以提供初始随机性，打破对称性，并确保模型在训练过程中能够有效学习。不过，选择合适的随机初始化范围和分布类型是关键，通常需要根据具体任务和网络结构进行调整。在本次实验中，我们也采用了随机初始化的策略。

4、 Xavier/Glorot Initialization

Xavier/Glorot 初始化是一种广泛应用于神经网络中的权重初始化方法，其目标是确保网络在训练过程中能够稳定地传播信息，而不会发生梯度消失或梯度爆炸。它由 Xavier Glorot 提出，通常称为 Xavier 初始化或 Glorot 初始化。

Xavier 初始化的核心理念是平衡神经网络每一层的输入和输出的方差，以确保在前向传播和反向传播过程中梯度和激活值保持稳定。它的关键点在于根据每一层的输入和输出节点数来计算权重的初始值。

Xavier 初始化可以使用均匀分布或正态分布进行权重初始化，具体取决于网络结构和激活函数的选择：

均匀分布：权重从一个以零为中心、范围为 ‘[-a, a]’ 的均匀分布中采样，其中 ‘a = sqrt(6 / (n_in + n_out))’，’n_in’ 是当前层的输入节点数，’n_out’ 是当前层的输出节点数。

正态分布：权重从一个以零为中心、标准差为 ‘sqrt(2 / (n_in + n_out))’ 的正态分布中采样。

平衡输入输出方差：Xavier 初始化的公式旨在使每一层的输入和输出方差保持平衡，从而在训练过程中保持梯度的稳定。

避免梯度消失和梯度爆炸：通过控制权重的初始范围，Xavier 初始化可以有效避免梯度消失和梯度爆炸，这两者常常是神经网络训练失败的主要原因。

Xavier 初始化通常适用于使用 Sigmoid、Tanh 等对称激活函数的网络。对于 ReLU 及其变种，He 初始化可能更为适用，因为这些激活函数的特性要求更大的权重范围以确保梯度的稳定。

激活函数的选择：在选择 Xavier 初始化时，要考虑网络中使用的激活函数。如果是非对称激活函数，如 ReLU，则可能需要考虑 He 初始化。

网络规模：对于较深或复杂的网络，Xavier 初始化可能需要调整权重初始化方式，以确保训练的稳定性。

总体而言，Xavier/Glorot 初始化是一种非常有效的权重初始化方法，尤其适用于避免梯度消失和梯度爆炸等问题，从而确保神经网络训练的稳定和收敛。

5、 LeCun Initialization

LeCun 初始化是一种权重初始化方法，通常用于神经网络，特别是使用 Sigmoid 或 Tanh 激活函数的网络。它的目标是确保网络在前向传播和反向传播过程中保持稳定，以避免梯度消失或梯度爆炸等问题。LeCun 初始化以 Yann LeCun 的研究为基础，专注于根据输入节点数来设置权重的初始值。

LeCun 初始化的核心理念是确保每一层的输入和输出之间的方差保持平衡，以避免在训练过程中出现梯度消失或梯度爆炸的问题。它的主要思路是根据每一层的输入节点数（即特征数）来计算权重的初始值，以确保网络能够稳定训练。

LeCun 初始化通常采用正态分布来初始化权重，具体的标准差取决于每一层的输入节点数：

正态分布：权重从一个正态分布中采样，平均值为零，标准差为 ‘1 / sqrt(n_in)’，其中 ‘n_in’ 是当前层的输入节点数。

避免梯度消失：LeCun 初始化通过设置权重的标准差，确保在训练过程中，梯度不会过于迅速地减小，避免了梯度消失问题。

适用于 Sigmoid 和 Tanh：这种初始化方法专门针对 Sigmoid 和 Tanh 激活函数设计，这两者在较大的权重范围下容易出现梯度消失的情况。

收敛稳定：由于权重初始化考虑了输入节点数，网络在训练过程中收敛更稳定。

LeCun 初始化通常用于以下场景：

1、使用 Sigmoid 或 Tanh 激活函数的神经网络。

2、希望确保训练过程稳定且避免梯度消失的情况。

激活函数的选择：LeCun 初始化适用于 Sigmoid 和 Tanh 激活函数，但对于 ReLU 及其变种，可能需要考虑其他初始化方法，如 He 初始化，因为 ReLU 通常需要更大的权重范围。

网络规模：在较深或复杂的网络中，可能需要进一步调整权重的初始化方式，以确保训练的稳定性。

LeCun 初始化提供了一种确保网络在训练过程中保持稳定的方法，特别适用于那些容易出现梯度消失的激活函数。通过这种方法，权重初始化能够更好地支持模型在训练过程中收敛，并提高模型的性能。

激活函数

激活函数是神经网络中的一个关键组件，决定了神经元的输出值。它通过对输入信号进行非线性变换，使神经网络能够学习和表达复杂的特征和模式。在深度学习中，激活函数是确保网络具有非线性特征的关键，因为纯线性变换无法实现复杂的任务。

激活函数的作用

引入非线性：激活函数将线性输入转换为非线性输出，赋予神经网络处理复杂数据的能力。

确定神经元的激活状态：激活函数决定每个神经元的输出值，通常在特定范围内，例如 0 到 1，或 -1 到 1。

支持反向传播：激活函数通常是可微的，以便在反向传播过程中计算梯度。

常见的激活函数

Sigmoid：将输入映射到 0 和 1 之间，是阶跃函数的近似，相比于阶跃函数，sigmoid 函数的数学性质更好，全局可导的性质方便采用梯度下降法去计算梯度并反向传播，适用于二分类任务，但在深度网络中容易导致梯度消失。

Tanh：将输入映射到 -1 和 1 之间，类似于 Sigmoid，但中心对称，梯度消失的风险稍小。

ReLU (Rectified Linear Unit)：如果输入大于 0，则返回输入，否则返回 0。ReLU 是深度学习中最常用的激活函数，具有简单、高效、不容易出现梯度消失的问题。

Leaky ReLU：ReLU 的变种，如果输入小于 0，返回输入乘以一个小的常数（通常是 0.01）。它缓解了 ReLU 的“死亡神经元”问题，防止计算所得的梯度位于 ReLU 的梯度为零的区域而造成神经元失效的情况发生。。

Softmax：通常用于神经网络的输出层，将一组数值转换为概率分布，适用于多分类

任务。

ELU（Exponential Linear Unit）：与 Leaky ReLU 类似，但对负值有更平滑的处理方式。

Swish：由 Google 提出的激活函数，具有非线性特征，同时在某些任务上可能比 ReLU 更有效。

选择激活函数的考虑因素

任务类型：对于二分类任务，Sigmoid 是常用的选择；对于多分类任务，Softmax 通常用于输出层。

梯度稳定性：激活函数不应引起梯度消失或梯度爆炸。

计算效率：ReLU 等函数具有计算简单、高效的特性。

非线性特征：激活函数应具有足够的非线性，以使神经网络能够学习复杂的特征。

激活函数在神经网络中发挥着关键作用，影响着模型的性能和稳定性。选择适当的激活函数是构建有效神经网络的重要步骤。在本次实验中，对于隐藏层的神经元节点，我们采用了 ReLU 激活函数，对于输出层的神经元节点，我们采用了 softmax 激活函数，将输出值转换为相应的概率分布。

学习率

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。

梯度截断

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

梯度截断通常通过以下两种方式实现：

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

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。

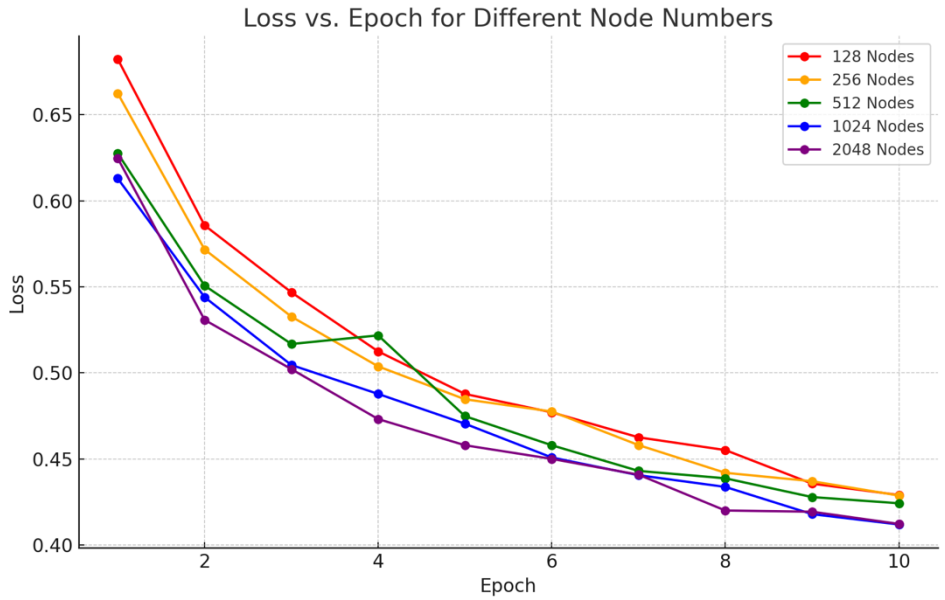
权重衰退是一种强大的正则化工具，可以帮助防止过拟合，提高模型的泛化能力，同时确保训练过程的稳定性。通过在损失函数中加入权重惩罚项，权重衰退可以有效控制模型的复杂度，从而提高模型在实际应用中的表现。在本次实验中，我们采用了$\lambda = 10^{-4}$的阈值。

实验结果

参数实验

隐藏层节点数

在本次实验中，测试了隐藏层节点数为 128、256、512、1024、2048 时的情况，并绘制了测试误差-轮数的图像，从图中可以看出，最佳的隐藏层节点数应该为 1024，然而较大的神经网络会消耗更多的训练时间，并且训练效果的差异并不显著，所以仍然选择节点数为 256。



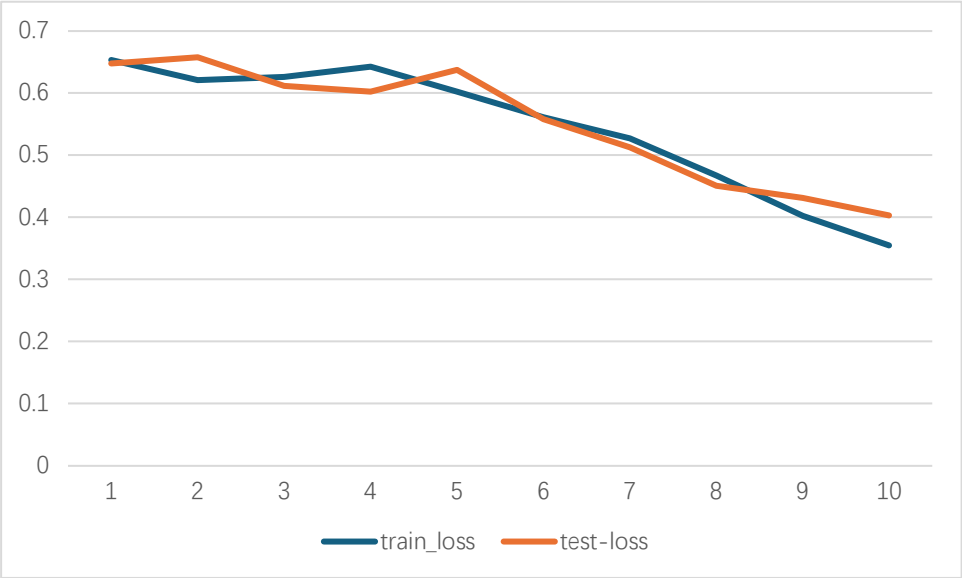
学习率

在本次实验中测试了多种学习率，得到最佳的学习率约为 0.1

模型训练

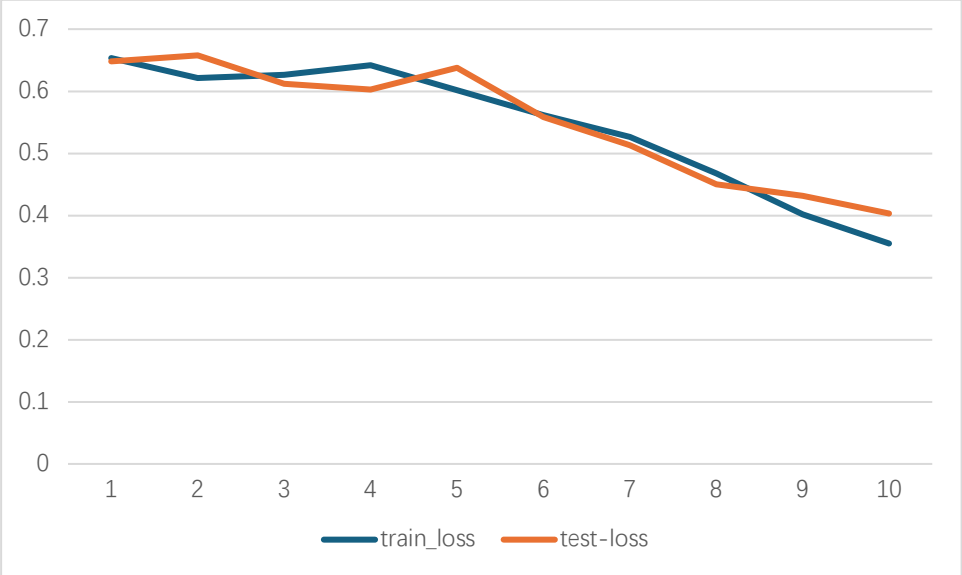
任务一

采用 256 个隐藏层节点，学习率 0.1 的参数条件，得到训练的结果



在测试集上，模型预测准确率达到了 87.11%

任务二



在测试集上，模型预测准确率达到了 86.29%

代码附录

utils.py

```
import torch
from torch.utils.data import DataLoader
from torchvision import transforms


def to_device(data, device):
    if isinstance(data, (list, tuple)):
        return [to_device(x, device) for x in data]
    return data.to(device, non_blocking=True)


def get_default_device():
    return torch.device("mps" if torch.backends.mps.is_available()
else 'cpu')


class MpsDataLoader(DataLoader):
    def __init__(self,
                 dataset,
                 batch_size=1,
                 shuffle=False,
                 sampler=None,
                 batch_sampler=None,
                 num_workers=0,
                 collate_fn=None,
                 pin_memory=False,
                 drop_last=False,
                 timeout=0,
                 worker_init_fn=None,
                 multiprocessing_context=None):
        super(MpsDataLoader,
              self).__init__(dataset, batch_size, shuffle, sampler,
                             batch_sampler, num_workers, collate_fn,
                             pin_memory, drop_last, timeout,
worker_init_fn,
                             multiprocessing_context)
        self.device = get_default_device()

    def __iter__(self):
        base_iterator = super().__iter__()
        for batch in base_iterator:
            yield to_device(batch, self.device)
```

```
train_tfms = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize((0.1307, ), (0.3081, ))
])

test_tfms = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.1307, ), (0.3081, ))])


def relu(x):
    return torch.max(x, torch.zeros_like(x))


def fit_and_test(model, epochs, train_dl, test_dl, train_step):
    history = []
    for epoch in range(epochs):
        if hasattr(model, 'train'):
            model.train()
        train_losses = []
        for images, labels in train_dl:
            train_losses.append(train_step(model, images, labels))
        if hasattr(model, 'eval'):
            model.eval()
        test_losses = []
        test_accs = []
        for images, labels in test_dl:
            outputs = model(images)
            test_losses.append(
                torch.nn.functional.cross_entropy(outputs, labels))
            _, preds = torch.max(outputs, dim=1)
            correct = torch.sum((preds == labels).int())
            total = len(preds)
            accuracy = correct.item() / total
            test_accs.append(torch.tensor(accuracy))

        history.append({
            'train_loss': torch.stack(train_losses).mean().item(),
            'test_loss': torch.stack(test_losses).mean().item(),
            'test_acc': torch.stack(test_accs).mean()
        })
        print(
```

```
            f"Epoch[{epoch + 1:d}]: train_loss: {history[-
1]['train_loss']:.4f}, "
            f"test_loss: {history[-1]['test_loss']:.4f}, test_acc:
{history[-1]['test_acc']:.4f}"
        )
    return history
```

config.py

```
import argparse

parser = argparse.ArgumentParser(description='Hyper-parameters
management')

# Hardware options
parser.add_argument('--n_threads',
                    type=int,
                    default=6,
                    help='number of threads for data loading')
parser.add_argument('--cpu', action='store_true', help='use cpu
only')
parser.add_argument('--seed', type=int, default=42, help='random
seed')

# data in/out and dataset

parser.add_argument('--save',
                    default='./',
                    help='save path of trained model')

parser.add_argument('--predict',
                    default='./',
                    help='save path of predict model')

parser.add_argument('--batch_size',
                    type=int,
                    default=64,
                    help='batch size of trainset')

# train
parser.add_argument('--epochs',
                    type=int,
                    default=10,
                    metavar='N',
```

```
                help='number of epochs to train (default: 10)')

parser.add_argument('--lr',
                type=float,
                default=0.01,
                metavar='LR',
                help='learning rate (default: 0.01)')

parser.add_argument('--momentum',
                type=float,
                default=0.5,
                metavar='M',
                help='SGD momentum (default: 0.5)')

parser.add_argument('--weight_decay',
                type=float,
                default=1e-4,
                metavar='W',
                help='SGD weight_decay (default: 1e-4)')

parser.add_argument('--nesterov',
                type=bool,
                default=False,
                help='SGD nesterov (default: False)')

parser.add_argument('--early-stop',
                default=20,
                type=int,
                help='early stopping (default: 20)')

args, unknown = parser.parse_known_args()
```

train_man.py

```
import torch
from torch import nn
from utils import *
from torchvision import datasets
from config import args as arg
device = torch.device(get_default_device())

batch_size = arg.batch_size
lr = arg.lr
grad_clip = 0.1
```

```
epochs = arg.epochs
seed = arg.seed

torch.manual_seed(seed)

train_data = datasets.FashionMNIST(root="data",
                                   train=True,
                                   download=True,
                                   transform=train_tfms)

test_data = datasets.FashionMNIST(root="data",
                                  train=False,
                                  download=True,
                                  transform=test_tfms)

train_dl = MpsDataLoader(train_data,
                         batch_size=batch_size,
                         shuffle=True,
                         pin_memory=True)
test_dl = MpsDataLoader(test_data,
                        batch_size=batch_size,
                        pin_memory=True)


class SingleLayerNN:
    def __init__(self,
                 input_size,
                 hidden_size,
                 output_size,
                 learning_rate=0.1):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate
        self.device = torch.device(get_default_device())

        self.w1 = torch.randn(input_size, hidden_size) * 0.01
        self.b1 = torch.zeros(hidden_size)
        self.w2 = torch.randn(hidden_size, output_size) * 0.01
        self.b2 = torch.zeros(output_size)

    def __call__(self, inputs):
        return self.forward(inputs)
```

```
    def forward(self, x):
        x = x.view(-1, self.w1.shape[0])
        h1 = relu(x @ self.w1 + self.b1)
        return h1 @ self.w2 + self.b2

    def parameters(self):
        yield self.w1
        yield self.b1
        yield self.w2
        yield self.b2

    def named_parameters(self):
        params = {'w1': self.w1, 'b1': self.b1, 'w2': self.w2, 'b2':
self.b2}
        return params.items()

    def to(self, my_device, non_blocking=False):
        self.device = my_device
        for name, param in self.named_parameters():
            setattr(self, name, param.to(device,
non_blocking=non_blocking))
        return self


class TrainStep:
    def __call__(self, model, images, labels):
        # forward
        outputs = model(images)
        z = images.view(-1, model.w1.shape[0]) @ model.w1 + model.b1

        # calculate loss
        loss = torch.nn.functional.cross_entropy(outputs, labels)

        # backpropagation
        delta_2 = torch.nn.functional.softmax(outputs, dim=1) -
torch.nn.functional.one_hot(labels,

num_classes=10)
        grad_w2 = relu(z).T @ delta_2 / batch_size
        grad_b2 = torch.sum(delta_2, dim=0) / batch_size

        delta_1 = delta_2 @ model.w2.T
        delta_1[z <= 0] = 0
        grad_w1 = images.view(-1, model.w1.shape[0]).T @ delta_1 /
```

```
batch_size
        grad_b1 = torch.sum(delta_1, dim=0) / batch_size

        # parameters update
        model.w2 -= model.learning_rate * grad_w2
        model.b2 -= model.learning_rate * grad_b2
        model.w1 -= model.learning_rate * grad_w1
        model.b1 -= model.learning_rate * grad_b1
        for param in model.parameters():
            if param.grad is not None and param.grad.nelement() > 0:
                nn.utils.clip_grad_value_([param], grad_clip=1.0)

        return loss


if __name__ == "__main__":
    my_model = to_device(
        SingleLayerNN(input_size=784,
                      hidden_size=256,
                      output_size=10,
                      learning_rate=lr), get_default_device())
    history = fit_and_test(my_model, epochs, train_dl, test_dl,
TrainStep())
```

train_torch.py

```
import torch
from torch import nn
from torchvision import datasets
from config import args as arg
from utils import *
device = torch.device(get_default_device())

batch_size = arg.batch_size
my_lr = arg.lr
my_max_lr = 10 * my_lr
grad_clip = 0.1
my_epochs = arg.epochs
my_momentum = arg.momentum
my_weight_decay = arg.weight_decay
seed = arg.seed

torch.manual_seed(seed)
```

```
train_data = datasets.FashionMNIST(root="data",
                                   train=True,
                                   download=True,
                                   transform=train_tfms)

test_data = datasets.FashionMNIST(root="data",
                                  train=False,
                                  download=True,
                                  transform=test_tfms)

train_dl = MpsDataLoader(train_data,
                         batch_size=batch_size,
                         shuffle=True,
                         pin_memory=True)
test_dl = MpsDataLoader(test_data,
                        batch_size=batch_size,
                        pin_memory=True)


class SingleLayerNN(nn.Module):
    def __init__(self, num_inputs, num_outputs, num_hidden,
initializer=None):
        super().__init__()
        self.num_inputs, self.num_outputs, self.num_hidden =
num_inputs, num_outputs, num_hidden
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(num_inputs, num_hidden)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(num_hidden, num_outputs)
        if initializer:
            initializer(self.fc1.weight)
            initializer(self.fc2.weight)

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x


class TrainStep:
    def __init__(self,
                 model,
```

```
                lr=my_lr,
                max_lr=my_max_lr,
                epochs=my_epochs,
                momentum=my_momentum,
                weight_decay=my_weight_decay):
        self.loss_fn = nn.CrossEntropyLoss()
        self.optimizer = torch.optim.SGD(model.parameters(),
                                         lr=lr,
                                         momentum=momentum,
                                         weight_decay=weight_decay)
        self.scheduler = torch.optim.lr_scheduler.OneCycleLR(
            self.optimizer,
            max_lr,
            epochs=epochs,
            steps_per_epoch=len(train_dl))

    def __call__(self, model, images, labels):
        self.optimizer.zero_grad()
        loss = self.loss_fn(model(images), labels)
        loss.backward()
        nn.utils.clip_grad_value_(model.parameters(), grad_clip)
        self.optimizer.step()
        self.scheduler.step()
        return loss


if __name__ == "__main__":
    my_model = to_device(SingleLayerNN(784, 10, 256),
get_default_device())

    history = fit_and_test(my_model, my_epochs, train_dl, test_dl,
TrainStep(my_model))
    torch.save(my_model.state_dict(), 'single_layer_nn.pth')
```