ANN_Lab1 仿真实验报告

——李昭阳 2021013445

在 LAB1 的仿真实验中，我整理获得了一个多层感知器模型，并通过联合优化超参数来获得最佳性能。以下是具体步骤与优化分析。

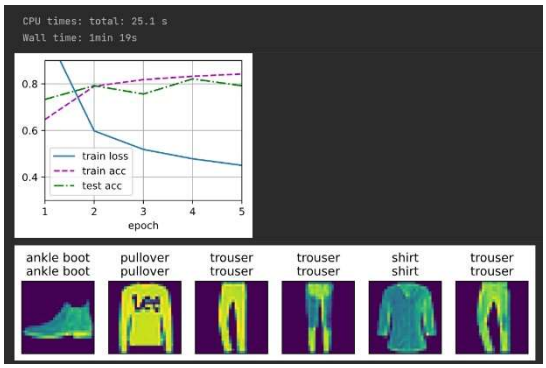
数据集

整理示例文件中的真实数据集 Fashion-Mnist 作为任务输入。

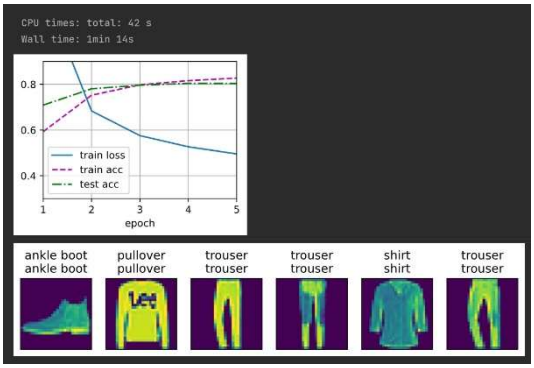
MLP 参数优化过程

在优化的过程中，我选取了四个超参数进行分析，分别为批量大小、隐藏层单元数、学习率、迭代周期数。同时用控制变量法对其优化效果进行了量化实验。

首先分析批量大小（batch_size）对训练的影响，多组超参数实验结果如下：（下文中的所有图注格式均为：批量大小-隐藏层单元数-学习率-迭代周期数）



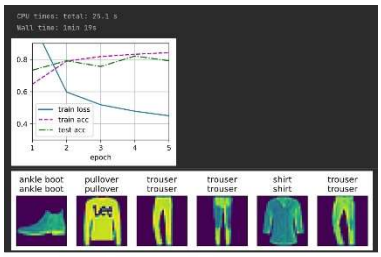
256-256-0.1-5



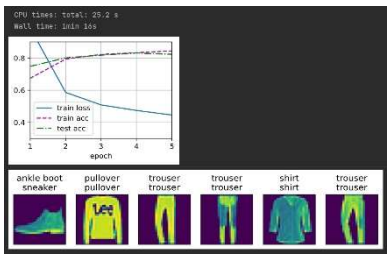
400-256-0.1-5

由图可知，过大的 batch_size 会使得 train_loss 减小速度变慢，即会一定程度上降低训练收敛速度，同时由 CPU 时间可以看出，较大的 batch_size 需要更长的 CPU 训练时间。

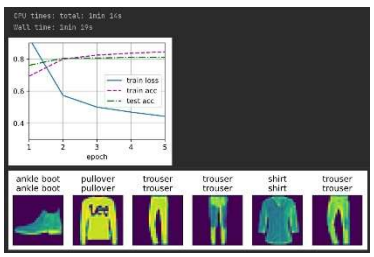
再分析隐藏层单元数对训练的影响，实验结果如下：



256-256-0.1-5



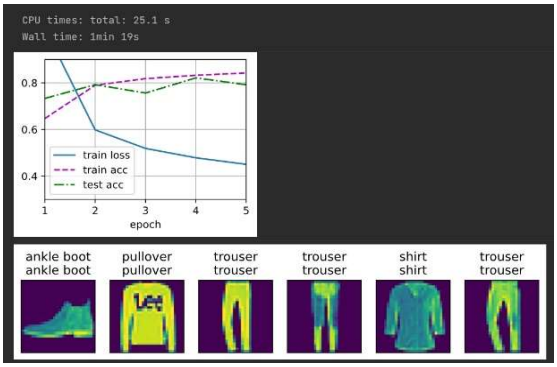
256-512-0.1-5



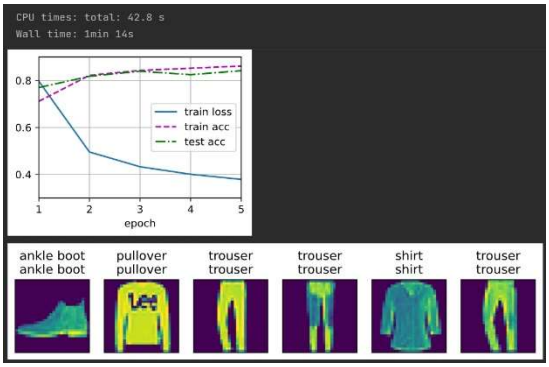
256-1024-0.1-5

由图可知，适当的增加隐藏层数可以略微提高 train_loss 减小速度，使得收敛速度更快，同时还可以略微提升预测准确率。但是过大的隐藏层数会显著增大 CPU 占用时间，使得对设备的要求提高。

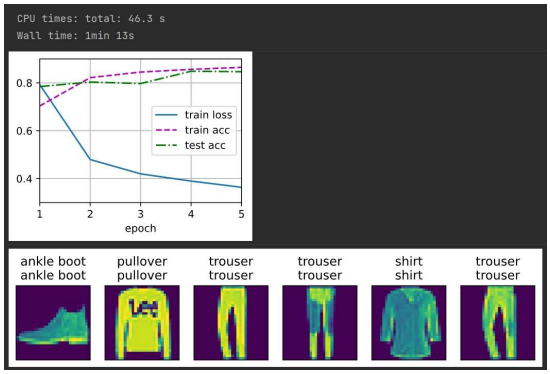
分析学习率对训练的影响，实验结果如下：



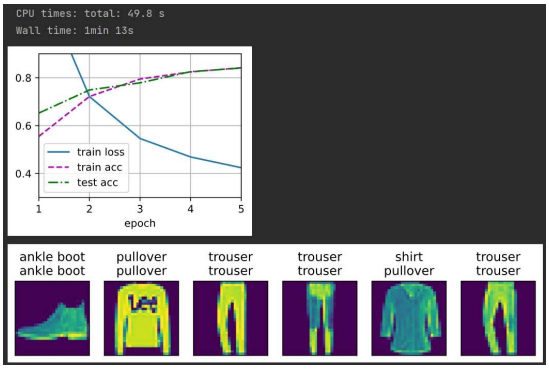
256-256-0.1-5



256-256-0.3-5



256-256-0.5-5

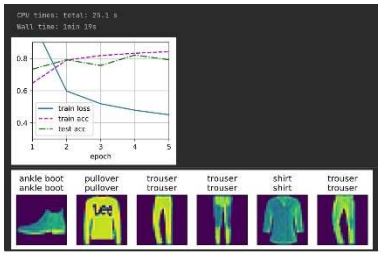


256-256-0.7-5

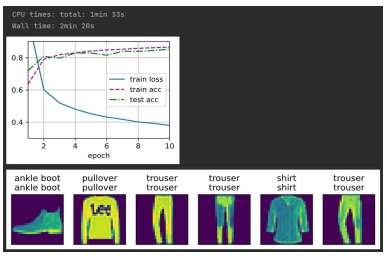
由图可知，适当的增加学习率可以略微提高 train_loss 减小速度，使得收敛速度更快，同时还可以略微提升预测准确率。但是过大的学习率会引起振荡，使得训练精度不足。同时不精确的学习率会增大 CPU 占用时间，使得对设备的要求提高。

值得注意的是，过高的学习率会导致模型训练崩溃，我认为是由于学习率设置得太高，梯度变得异常大，导致权重的更新变得不稳定，使神经网络的权重急剧增加，无法收敛到合适的值，从而破坏了模型的训练。

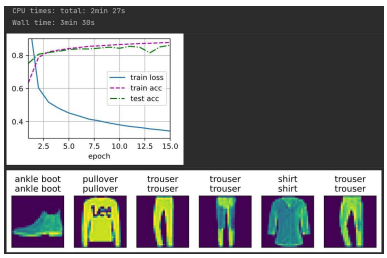
分析迭代周期数对训练的影响，实验结果如下：



256-256-0.1-5



256-256-0.1-10

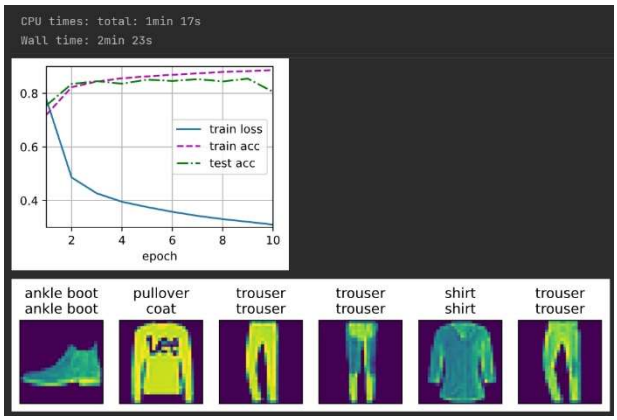


256-1024-0.1-5

由理论推导可知，增加训练 epoch 数目可以使得训练结果收敛，提高模型的预测率。但是就本实验研究所得模型，在 5 epochs 时就已基本收敛，所以 epoch 数目对模型的影响较小。

综上，如下图显示，得到推测最优 MLP 的参数为：

批量大小-隐藏层单元数-学习率-迭代周期数 == 256-512-0.3-10

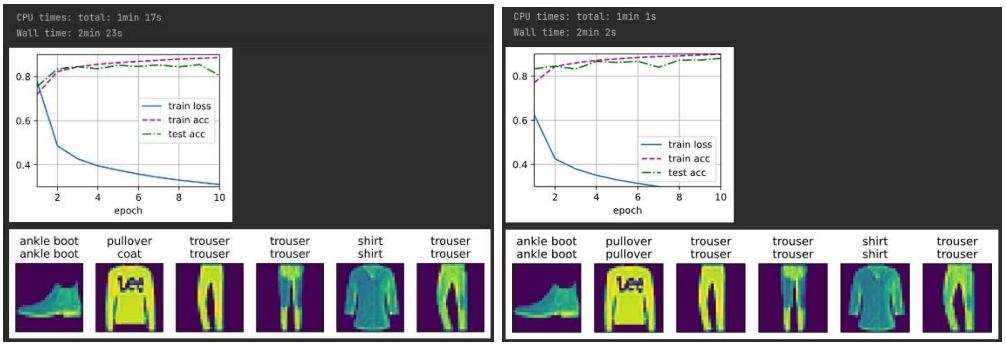


256-512-0.3-10

可以看到 train_loss 迅速下降，收敛较快；同时模型准确率较高，可认为性能较好。

对比自己实现的模块和 PyTorch 库中调用的模块的效率

对比不同自己实现的模块和 PyTorch 库中调用的模块如下，整体对比如下。



256-512-0.3-10　　256-512-0.3-10-PyTorch

训练速度：由 CPU 时间可知，我自己实现的 MLP 比 PyTorch 中的模块在相同任务上的训练速度更慢。

我认为 PyTorch 中的模块通常经过高度优化，能够利用 GPU 等硬件资源，因此在训练速度上具有优势。为了解决这个问题，可能需要更高性能的硬件设备，或者进行底层逻辑的修改。

模型性能：由 train_loss 的减小速率可知，在相同的任务上，PyTorch 的表现更好，训练效率很高；同时由预测准确率可知，PyTorch 所训练出的模型准确率更高，更加佐证了其效率高的事实。

我认为这可能和权重初始化有关，PyTorch 的权重初始化方法可能比我模型更加优化，进而使得其模型的收敛速度和性能更优。为了解决这个问题，我认为合适的初始化方法对模型性能非常重要，需要优化我的初始化方法。

代码复杂性：我自己实现的模块的代码复杂性非常高，而 PyTorch 中的模块的代码复杂性较低。从编码效率上看，我认为 PyTorch 更优。

深度学习框架文档提供的函数：

- **激活函数：**
  - Sigmoid 激活函数：将输入值映射到 0 到 1 的范围；
  - ReLU 激活函数：将输入值映射为非负值，函数形式为 f(x) = max(0, x)；
  - Leaky ReLU 激活函数： ReLU 的一种变种；
  - ELU 激活函数：ELU 是另一种修正线性单元，它在输入为负数时有一个非零斜率，而且具有更平滑的曲线；
  - Tanh 激活函数：tanh（双曲正切）激活函数将输入值映射到 -1 到 1 的范围；
  - Softmax 激活函数：将一组输入值映射到 0 到 1 之间，并且所有输出值的总和等于 1，表示各个类别的概率；
  - Swish 激活函数：输入值为正时类似于 ReLU，在输入值为负时类似于 sigmoid。

- **损失函数：**
  - 均方误差损失函数：用于回归问题，计算预测值与目标值之间的平均平方误差。
  - 二元交叉熵损失函数：用于二元分类问题，衡量模型输出与实际标签之间的差异。
  - 多类别交叉熵损失函数：用于多类别分类问题，衡量模型对多个类别的预测与实际标签的差异。
  - 对数似然损失函数：用于训练生成模型，如变分自动编码器和生成对抗网络。
  - 汉明损失函数：用于多标签分类问题，衡量预测标签与实际标签之间的汉明距离。
  - 背景误差损失函数：通常用于支持向量机和某些形式的分类任务。
  - KL 散度损失函数：通常用于训练变分自动编码器（VAE）和其他生成模型。
  - Huber 损失函数：类似于均方误差，但对离群值的敏感性较小。

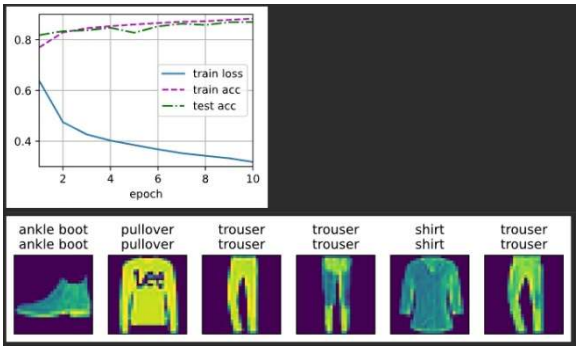
- **初始化方法**
  - 随机初始化：在这种初始化方法中，权重和偏差是随机生成的。通常使用均匀分布或正态分布的随机值来初始化。这种方法在一开始为模型提供了随机的起点，有助于模型避免陷入局部最小值。
  - 零初始化：在零初始化中，所有的权重和偏差都被设置为零。虽然这是一个简单的初始化方法，但在某些情况下可能不够有效，因为它可能导致梯度消失问题。
  - Xavier 初始化：Xavier 初始化是一种常用于激活函数是 S 型函数（如 sigmoid 和 tanh）的前馈神经网络的初始化方法。它通过根据权重数量和输入、输出单元的数量来计算适当的缩放因子，以保持梯度的方差稳定。
  - He 初始化：He 初始化适用于使用 ReLU 激活函数的神经网络。它通过将权重初始化为均匀或正态分布的随机值，并缩放以保持方差稳定。这有助于解决 ReLU 函数可能导致的梯度消失问题。
  - Lecun 初始化：Lecun 初始化是一种针对 tanh 激活函数的特殊初始化方法，旨在保持梯度的方差稳定。它通常使用均匀或正态分布的随机值，并根据输入单元的数量进行缩放。

- **优化方法：**
  - 随机梯度下降：SGD 是一种基本的优化方法，它在每次迭代中使用单个样本或小批量样本来计算梯度并更新模型参数。虽然它是一种简单的方法，但可以在训练大型神经网络时表现良好。

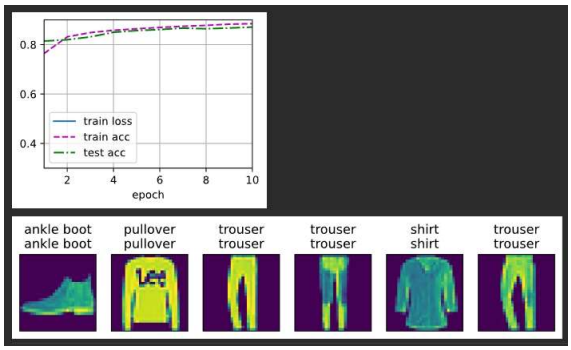
- 小批量梯度下降：与 SGD 不同，小批量梯度下降在每次迭代中使用小批量样本来计算梯度和更新参数。这通常比纯 SGD 更稳定，且在训练过程中收敛更快。
- 动量优化：动量优化引入了一个动量项，模拟了物理中的动量概念。它有助于平稳化梯度下降，减少训练过程中的震荡，并有助于快速收敛。
- 自适应学习率方法：这类方法根据梯度更新的情况来自适应地调整学习率。常见的方法包括 AdaGrad、RMSprop 和 Adam。它们通常能够有效地处理非均匀的梯度分布，以及自动调整学习率以提高训练效率。
- 带动态学习率的随机梯度下降：这种方法通过在训练过程中逐渐减小学习率，有助于提高训练的稳定性和性能。常见的学习率调度包括指数衰减、时间衰减和余弦退火。
- L-BFGS：L-BFGS 是一种拟牛顿法，通常用于优化较小规模的神经网络。它具有较低的内存消耗，但通常不适用于大型神经网络。
- Nesterov 动量：Nesterov 动量是动量优化的一种变种，它在计算梯度时使用了预测的下一个位置。这有助于减少振荡和提高收敛速度。

每个类别尝试选择一种新的方法：

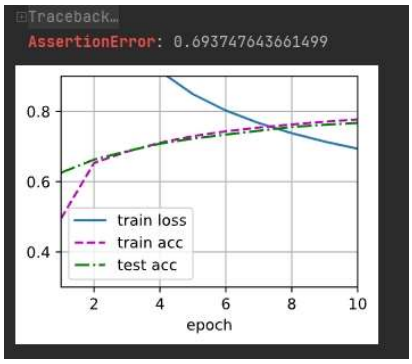
分别尝试了 Tanh 激活函数、MultiMarginLoss 损失函数、Xavier_normal_初始化方法、Adam 优化方法，实验结果如下。



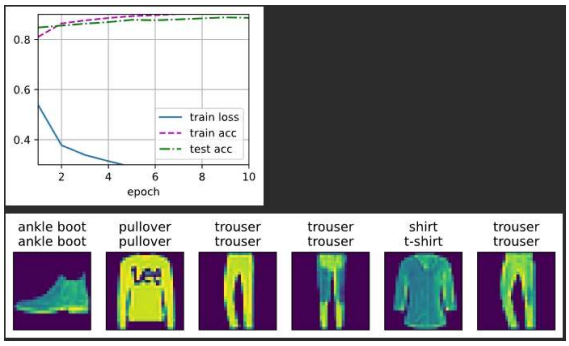
Tanh



MultiMarginLoss



Xavier_normal_



Adam

实验结果显示，在当前任务下，Tanh 激活函数与原 ReLU 激活函数的训练差异不大；MultiMarginLoss 会改变 train_loss，但是最终学习效果与原模型近似；Xavier_normal_初始化方法会显著影响模型效果，不合适的方法会导致灾难性的后果；Adam 优化函数会使得模型性能达到显著提高，准确率和收敛速度均有所提升。

总结

这次实验为我积累了深度学习领域的宝贵经验，我有很多心得体会。

首先我认为模型优化是一门艺术，正如老师上课所言“炼丹”，MLP 的性能优化不仅仅依赖于模型结构，还受到超参数的影响。联合优化超参数可以找到最佳组合，但这也需要耐心和大量的计算资源。我需要不断思考尝试，而不仅仅是机械地调整超参数。

同时在库模块 vs. 自己实现中，虽然 PyTorch 提供了许多高效的模块，但有时我们需要自己实现特定的功能。在实际项目中，选择何时使用库模块和何时自己实现是至关重要的。在性能和时间要求紧迫的情况下，应该更多使用库模块，但自己实现的模块提供了更大的灵活性，可以满足特定需求。

总的来说，这次实验不仅让我了解了有关 MLP 模型优化的具体步骤，还强调了深度学习领域的复杂性和多样性。通过不断实践，我才可以不断提高自己的学习技能，以应对不断变化的挑战。谢谢老师和助教的耐心讲解与准备！