深度学习部分大作业报告

魏少杭
学号：20373594
学院：人工智能研究院

2022 年 12 月 26 日

说明：所有代码使用的工具不包含任何深度学习框架中的任何模块，只包含 *numpy* 和 *matplotlib* 等基本库模块。

目录

1 第一部分：实现 4 层 MLP、损失函数、训练算法

第一部分的代码文件为：main.ipynb（jupyter notebook 文件）。

1.1 导入 MNIST 数据集

从网络资源 uri 下载 MNIST 的训练和测试集。训练集和测试集分别有 train_images、train_labels 和 test_images、test_labels。由于本次任务是实现 MLP 数字分类，所以输入的 images 需要首先从 (28,28) 的矩阵形式变为 (784,) 格式的矩阵形式；真实标签 labels 直接转化成独热编码，有利于之后利用交叉熵函数作为分类任务的损失函数开展计算。

具体编码如下：

导入 MNIST 数据集后的结果

```
train_images[0].shape    #查看训练的图像的张量形状
>>(784,)

import matplotlib.pyplot as plt
plt.imshow(train_images[0].reshape(28, 28))    #查看灰度图

train_labels[0]    # label独热编码
>>array([0, 0, 0, 0, 0, 1, 0, 0, 0, 0], dtype=uint8)
```

验证数据集导入后的结果如图 1。

1.2 定义 4 层 MLP 模型，并初始化权重

在 main.ipynb 中，我构建了 4 层的 MLP 模型。示意图为图 2：

构建时，每个隐藏层使用的激活函数均为 sigmoid 函数，即 $\sigma(X) = \frac{1}{1+e^{-X}}$，输出层无 sigmoid 函数，而是 SoftMax 操作即 $SoftMax(X_i) = \frac{e^{X_i}}{\sum_{X_j} e^{X_j}}$。具体而言，X 和 H 关系为：$X^{(i+1)} = concat(1, \sigma(H^i)), i = 1, 2, 3$；$H^{(i)} = W^{(i)}X^{(i)}, i = 1, 2, 3, 4$. 输出层的 $\hat{Y}$ 与 $H^{(4)}$ 关系为 $\hat{Y} = SoftMax(H^{(4)})$。

此处，初始化权重使用了一般的正态分布 N(0,0.1)，而偏置项均设为 0。

定义 4 层 MLP 模型，并初始化权重

```
# 定义4层MLP，定义每一个隐层的输出结点个数
H1_len = 512
H2_len = 256
H3_len = 64
H4_len = 10 # 分类为10类

# 获取训练数据的长度、各层输入的长度
X1_len = len(train_images[0]) + 1
X2_len = H1_len + 1
X3_len = H2_len + 1
X4_len = H3_len + 1

# 计算权重大小、初始化权重矩阵中的偏置和权重本身
```

```
1  train_images[0].shape
```

[15] ✓ 0.5s

```
(784,)
```

```
1  import matplotlib.pyplot as plt
2  plt.imshow(train_images[0].reshape(28, 28))
```

[5] ✓ 0.8s

```
<matplotlib.image.AxesImage at 0x178816e0280>
```

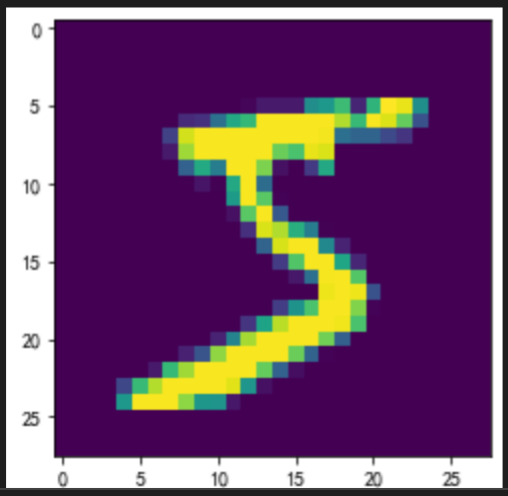

图 1: 验证数据集结果

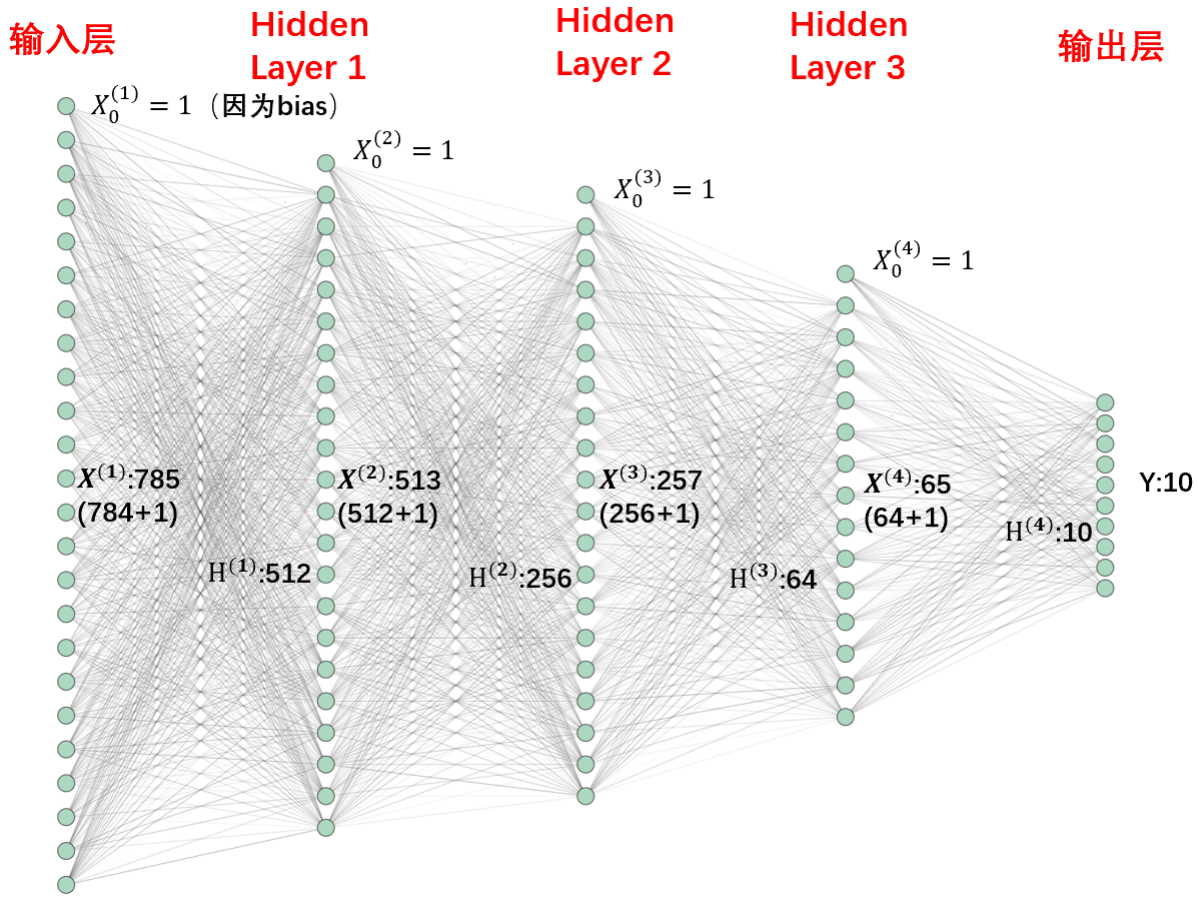

图 2: 4 层 MLP 模型示意图

```
# W1:(512, 785)
W1 = np.random.normal(0, 0.1, (H1_len, X1_len))
W1[:, 0] = 0.0
# W2:(256, 513)
W2 = np.random.normal(0, 0.1, (H2_len, X2_len))
W2[:, 0] = 0.0
# W3:(64, 257)
W3 = np.random.normal(0, 0.1, (H3_len, X3_len))
W3[:, 0] = 0.0
# W4:(10, 65)
W4 = np.random.normal(0, 0.1, (H4_len, X4_len))
W4[:, 0] = 0.0

# 前馈神经网络如下
X1 = np.concatenate((np.array([1]), img)) # X1:(785,)
H1 = np.matmul(W1, X1)  # H1:(512,)
X2 = np.concatenate((np.array([1]), sigmoid(H1)))   # X2:(513,)
H2 = np.matmul(W2, X2)  # H2:(256,)
X3 = np.concatenate((np.array([1]), sigmoid(H2)))   # X3:(257,)
H3 = np.matmul(W3, X3)  # H3:(64,)
X4 = np.concatenate((np.array([1]), sigmoid(H3)))   # X4:(65,)
H4 = np.matmul(W4, X4)  # H4:(10,)
Y_hat = softmax(H4) # Y_hat:(10,)
```

其中 SoftMax 和 sigmoid 设计如下：

SoftMax 和 sigmoid 设计

```
# 定义隐藏层激活函数sigmoid
def sigmoid(X):
    return 1 / (1 + np.exp(-X))

# 定义SoftMax
def softmax(X):
    exp_X = np.exp(X)
    sum_exp_X = np.sum(exp_X)
    return exp_X / sum_exp_X
```

1.3 分类损失函数设计

为使得损失函数具有衡量分类好坏的物理意义，**分类任务最好使用交叉熵损失函数**，在此任务中交叉熵为 $CrossEntropy = -\sum_i P(Y_i)lnP(\hat{Y_i})\ i = 0, 1, ..., 9$。而在之前提到过，MNIST 分类任务是唯一结果的多分类任务，所以 $P(Y_i) = 1$ 当且仅当 label=i 时成立，对于其他 i，则 $P(Y_i) = 0$。所以又可以简化为 $CrossEntropy = -lnP(\hat{Y_i}), where\ \ i = label$。

代码如下：

交叉熵损失函数

```
def CrossEntropy(Y, Y_hat):
    """根据softmax结果Y_hat和Y计算交叉熵损失值"""
    # 通过Y的独热编码找到真正的标签
    true_index = np.where(Y == 1)[0][0]
    return -np.log(Y_hat[true_index])
```

1.4 反向传播和梯度下降训练算法实现

1.4.1 反向传播数学公式推导与理解

首先，反向传播算法是基于链式求导法则的算法，其思路借鉴了动态规划中的迭代思想。

在本任务中，我根据自己设计的激活函数和损失函数进行了进一步的简化，说明如下：

第一，对 sigmoid 激活函数的单层导数进行封装化：因为 sigmoid 函数求导结果与原 sigmoid 函数有关联，$y=\sigma(x)=\frac{1}{1+e^{-x}}$，$\frac{\partial y}{\partial x}=\sigma(x)(1-\sigma(x))$，所以，封装了一个对 sigmoid 求导的函数：

对 sigmoid 进行求导的函数

```
# 定义隐藏层激活函数sigmoid的导数
def dsigmoid(X):
    return sigmoid(X) * (1-sigmoid(X))
```

第二，在输出层对 $H^{(4)}$ 进行 softmax 操作得到 $\hat{Y}$，且损失函数 L 为交叉熵损失的情况下，损失函数 L 对 $H^{(4)}$ 的求导结果十分简单。推导如下：

（1）定义损失函数原始形式 $L(\hat{Y})=-\sum_{i=0,..,9}Y_i ln\hat{Y}_i$；定义 SoftMax 形式为：$\hat{Y}_i=SoftMax(H_i^{(4)})=\frac{e^{H_i^{(4)}}}{\sum_{H_j^{(4)}}e^{H_j^{(4)}}}$。

（2）设 i 为真实标签数字，则如果 j=i 时，

$$
\begin{aligned}
\frac{\partial \hat{Y}_j}{\partial H_j^{(4)}} &= \frac{\partial}{\partial H_j^{(4)}}\left(\frac{e^{H_j^{(4)}}}{\sum_k e^{H_k^{(4)}}}\right) \\
&= \frac{(e^{H_j^{(4)}})' \cdot \sum_k e^{H_k^{(4)}} - e^{H_j^{(4)}} \cdot e^{H_j^{(4)}}}{(\sum_k e^{H_k^{(4)}})^2} \\
&= \frac{e^{H_j^{(4)}}}{\sum_k e^{H_k^{(4)}}} - \frac{e^{H_j^{(4)}}}{\sum_k e^{H_k^{(4)}}} \cdot \frac{e^{H_j^{(4)}}}{\sum_k e^{H_k^{(4)}}} \\
&= \hat{Y}_j(1-\hat{Y}_j)
\end{aligned}
$$

如果 j≠i 时

$$\begin{aligned}\frac{\partial \hat{Y}_j}{\partial H_j^{(4)}} &= \frac{\partial}{\partial H_j^{(4)}}\left(\frac{e^{H_j^{(4)}}}{\sum_k e^{H_k^{(4)}}}\right) \\ &= \frac{0 \cdot \sum_k e^{H_k^{(4)}} - e^{H_j^{(4)}} \cdot e^{H_i^{(4)}}}{(\sum_k e^{H_k^{(4)}})^2} \\ &= -\frac{e^{H_j^{(4)}}}{\sum_k e^{H_k^{(4)}}} \cdot \frac{e^{H_i^{(4)}}}{\sum_k e^{H_k^{(4)}}} \\ &= -\hat{Y}_j \hat{Y}_i\end{aligned}$$

（3）计算损失函数 L 对 $H_j^{(4)}$ 的导数：设 i 为真实标签数字，则

$$\begin{aligned}\frac{\partial L}{\partial H_j^{(4)}} &= \frac{\partial}{\partial H_j^{(4)}}\left(-\sum_{k=0,..,9} Y_k ln\hat{Y}_k\right) \\ &= \frac{\partial}{\partial \hat{Y}_k}\left(-\sum_{k=0,..,9} Y_k ln\hat{Y}_k\right) \cdot \frac{\partial \hat{Y}_k}{\partial H_j^{(4)}} \\ &= -\sum_{k=0,..,9} Y_k \cdot \frac{1}{\hat{Y}_k} \cdot \frac{\partial \hat{Y}_k}{\partial H_j^{(4)}} \\ &= -Y_j \frac{1}{\hat{Y}_j} \cdot \frac{\partial \hat{Y}_j}{\partial H_j^{(4)}} - \sum_{k!=j} Y_k \frac{1}{\hat{Y}_k} \cdot \frac{\partial \hat{Y}_k}{\partial H_j^{(4)}} \\ &= -Y_j \frac{1}{\hat{Y}_j} \cdot \hat{Y}_j(1-\hat{Y}_j) - \sum_{k!=j} Y_k \frac{1}{\hat{Y}_k} \cdot (-\hat{Y}_j \hat{Y}_k) \\ &= -Y_j + Y_j \hat{Y}_j + \sum_{k!=j} Y_k \hat{Y}_j \\ &= -Y_j + \hat{Y}_j \sum_k Y_k \\ &= \hat{Y}_j - Y_j\end{aligned}$$

对应的代码如下：

计算交叉熵损失关于最后一层隐藏节点向量的导数

```
# 定义交叉熵损失关于输入softmax的最后一层隐藏层的结点向量last_H的导数，需要给定Y和Y_hat
def dCrossEntropy_lastH_softmax(Y, Y_hat):
    """
    基于softmax后loss函数为交叉熵损失的情况下进行计算
    - Y:独热编码：如[0, 1, 0, 0, 0, ..., 0] 共10维 shapelike(10,)
    - Y_hat:SoftMax后的结果：如[0.223, 0.112, 0.511, ..., 0] 共10维且所有维度的值加起来为1, shapelike(10,)
    - 返回值：是直接对最后一层隐藏节点last_H的导数，shapelike：(10,)
    """
```

```
    # 通过Y的独热编码找到真正的标签
    true_index = np.where(Y == 1)[0][0]
    # 由于已经经过数学证明验证得到了在SoftMax和CrossEntropy的双重作用下得到的求导公式
    # 所以直接使用结论
    dCrossEntropy_H = Y_hat
    dCrossEntropy_H[true_index] -= 1
    return dCrossEntropy_H
```

在以上的简化操作后，下面开始正式的 BP 算法流程：

（1）**给定输入，计算各层输出值**，这一部分在之前定义 MLP 的前馈网络已经说明，故省略。

（2）根据链式求导法则，计算**损失函数对每一层的输出结点（激活之前的输出结点）的梯度**：

损失函数对于**最后一层**的输出结点求梯度：$\frac{\partial L}{\partial h^{(4)}} = dCrossEntropy\_lastH\_softmax(Y, \hat{Y})$.

损失函数对于**其他层**的输出结点求梯度 (n=1,2,3)：

$$\begin{aligned}\frac{\partial L}{\partial H^{(n)}} &= \frac{\partial L}{\partial X^{(n+1)}} \cdot \frac{\partial X^{(n+1)}}{\partial H^{(n)}} \\ &= \frac{\partial L}{\partial X^{(n+1)}} \cdot dsigmoid(H^{(n)})\end{aligned}$$

需要说明的是，上面的最后一行是两者进行按元素相乘而非矩阵乘法。

计算**损失函数对于每一层输入结点的梯度** (n=1,2,3,4)：

$$\begin{aligned}\frac{\partial L}{\partial X^{(n)}} &= \frac{\partial L}{\partial H^{(n)}} \cdot \frac{\partial H^{(n)}}{\partial X^{(n)}} \\ &= \frac{\partial L}{\partial H^{(n)}} \cdot W^{(n)}\end{aligned}$$

（3）根据链式求导法则，计算**损失函数对于权重矩阵的梯度**：

$$\frac{\partial L}{\partial W(n)} = (\frac{\partial L}{\partial H^{(n)}})^T \cdot (X^{(n)})^T, n = 1, 2, 3, 4$$

1.4.2 反向传播代码

如下：

反向传播部分

```

        # 根据真实标签Y（独热编码）与预测标签Y_hat计算损失函数值
        loss = CrossEntropy(Y, Y_hat)

        # 计算各个梯度
        # 最后一层的特殊处理，直接计算交叉熵损失关于输入softmax的最后一层隐藏层的结点向量last_H的导数
```

```
        # dL_H4表示损失函数对第四层隐藏节点向量的导数，下面的类似
        dL_H4 = dCrossEntropy_lastH_softmax(Y, Y_hat)   # (10,)
        # dL_X4表示损失函数对第四层输入结点向量的导数，下面的类似
        dL_X4 = np.matmul(dL_H4, W4)    # (65,)
        dL_H3 = dL_X4[1:] * dsigmoid(H3)    # (64,)
        dL_X3 = np.matmul(dL_H3, W3)    # (257,)
        dL_H2 = dL_X3[1:] * dsigmoid(H2)    # (256,)
        dL_X2 = np.matmul(dL_H2, W2)    # (513,)
        dL_H1 = dL_X2[1:] * dsigmoid(H1)    # (512,)
        dL_X1 = np.matmul(dL_H1, W1)    # (785,)

        # 计算权重矩阵的梯度
        dL_W4 = np.matmul(dL_H4.reshape(-1, 1), X4.reshape(1, -1))
        dL_W3 = np.matmul(dL_H3.reshape(-1, 1), X3.reshape(1, -1))
        dL_W2 = np.matmul(dL_H2.reshape(-1, 1), X2.reshape(1, -1))
        dL_W1 = np.matmul(dL_H1.reshape(-1, 1), X1.reshape(1, -1))
```

1.4.3 梯度下降训练过程

我采用的是**单样本的梯度下降训练方法**。其中学习率设置为从 10e-3 到 10e-6 经过 100000 步线性衰减的可变学习率。每隔 100 步，输出一次这 100 步的平均损失函数值作为参考。由于数据本身不大，训练轮数不需太多，故一共训练 3 个 epoch。

梯度下降训练过程

```
# 定义进行深度学习的学习率和epoch数
# lr = 10e-3
lr = np.linspace(10e-3, 10e-6, 100000)
epoch_num = 3

step = 0
for i in range(epoch_num):
    j = 0
    every_steps = 100
    losses = np.zeros((every_steps,))
    for (img, Y) in zip(train_images, train_labels):
        # 前向传播：给定输入，拼接并计算输出值
        ……（前面已展示过）

        # 根据真实标签Y（独热编码）与预测标签Y_hat计算损失函数值
        loss = CrossEntropy(Y, Y_hat)

        # 反向传播：计算梯度
        ……（前面已展示过）

        # 更新权重矩阵
```

```
        W4 -= lr[step] * dL_W4
        W3 -= lr[step] * dL_W3
        W2 -= lr[step] * dL_W2
        W1 -= lr[step] * dL_W1

        # 每隔every_steps轮，输出一下loss结果
        losses[j % every_steps] = loss
        j += 1
        if step < 100000-1:
            step += 1
        if j % every_steps == 0:
            print('Epoch{} IMG{}to{}, Average Loss:{}'.format(i, j-
                every_steps, j, np.mean(losses)))


>>部分训练过程:
Epoch0 IMG2100to2200, Average Loss:0.18362152379963867
Epoch0 IMG2200to2300, Average Loss:0.1896124135268518
Epoch0 IMG2300to2400, Average Loss:0.20179651754641792
Epoch0 IMG2400to2500, Average Loss:0.2293236266446424
...
Epoch2 IMG59600to59700, Average Loss:0.11401187869244034
Epoch2 IMG59700to59800, Average Loss:0.525565174399886
Epoch2 IMG59800to59900, Average Loss:0.023479666981708586
Epoch2 IMG59900to60000, Average Loss:0.22911379196793016
```

1.5 预测、测试分类结果

预测代码如下:

预测数字

```
def predict(img, printIt=False):
    # 给定输入，拼接并计算输出值
    X1 = np.concatenate((np.array([1]), img)) # X1:(785,)
    H1 = np.matmul(W1, X1)  # H1:(512,)
    X2 = np.concatenate((np.array([1]), sigmoid(H1)))   # X2:(513,)
    H2 = np.matmul(W2, X2)  # H2:(256,)
    X3 = np.concatenate((np.array([1]), sigmoid(H2)))   # X3:(257,)
    H3 = np.matmul(W3, X3)  # H3:(64,)
    X4 = np.concatenate((np.array([1]), sigmoid(H3)))   # X4:(65,)
    H4 = np.matmul(W4, X4)  # H4:(10,)
    Y_hat = softmax(H4) # Y_hat:(10,)
    pred_label = np.argmax(Y_hat)
    if printIt:
        print('预测为数字{}'.format(np.argmax(Y_hat)))
```

```
    return Y_hat, pred_label
```

将测试集中所有图片进行预测后，统计得到平均损失值和平均准确率。测试代码如下：

对测试集进行测试，得到平均损失值和准确率

```
test_losses = []
accuracy = 0
for test_img, test_Y in zip(test_images, test_labels):
    test_Y_hat, pred_label = predict(test_img)
    true_label = np.where(test_Y==1)[0][0]
    test_loss = CrossEntropy(test_Y, test_Y_hat)
    test_losses.append(test_loss)
    accuracy += int(true_label == pred_label)
    print(true_label, pred_label)
accuracy /= len(test_images)
print('测试集平均损失为{}, 测试准确率为{}'.format(np.mean(test_losses),
    accuracy))
```

预测结果如下：

测试分类结果

```
测试集平均损失为0.18925592376580336, 测试准确率为0.9444
```

可以看到这个结果是比较不错的！

2 第二部分：调整权重初始化方法、学习率和激活函数

2.1 调整学习率和激活函数

该部分代码为 main2.ipynb。

相比 main.ipynb，主要调整了激活函数为 ReLU 函数及 ReLU 的导数、可变 learning rate 从 1e-3 开始，经过 15000 步，终止值从 1e-6 变为了 1e-7。

因为 Sigmoid 在远离 0 点的地方，梯度变得非常微小，所以求导以后可能会产生梯度消失的情况。为了避免梯度消失等情况，我选择了调整 ReLU 函数。另一方面，ReLU 函数求导也很方便，对于向量 X 的每一个分量，非负分量的导数为 1，负数分量导数为 0。因此是比较理想的一种激活函数。

结果如下：

测试结果

```
测试集平均损失为0.1666319055472008, 测试准确率为0.9495
```

2.2 再调整权重初始化方法

该部分代码为 main2-调整初始化方法.ipynb、main2-调整初始化方法为 xavier.ipynb、main2-调整初始化方法再调学习率.ipynb。

权重初始化方法选择原则与激活函数密切相关。

首先，我尝试了使用均匀分布初始化权重向量方法，发现并不能收敛，这说明了可能陷入了局部极小值，也说明这种初始化方法对于 ReLU 并不鲁棒。

然后，尝试了 Xavier 初始化方法。它在初始化权重时考虑了网络的大小（输入和输出单元的数量）。这种方法通过使权重与前一层中单元数的平方根成反比来确保权重保持在合理的值范围内。Xavier 初始化适用于使用 tanh、sigmoid 为激活函数的网络。

Xavier 初始化方法

```
W1 = np.random.normal(0, np.sqrt(2/(X1_len+H1_len)), (H1_len, X1_len))
W1[:, 0] = 0.0
# W2:(256, 513)
W2 = np.random.normal(0, np.sqrt(2/(X2_len+H2_len)), (H2_len, X2_len))
W2[:, 0] = 0.0
# W3:(64, 257)
W3 = np.random.normal(0, np.sqrt(2/(X3_len+H3_len)), (H3_len, X3_len))
W3[:, 0] = 0.0
# W4:(10, 65)
W4 = np.random.normal(0, np.sqrt(2/(X4_len+H4_len)), (H4_len, X4_len))
W4[:, 0] = 0.0
```

最后，通过查阅资料，选择了 He Initialization 方法。He Initialization 适用于使用 ReLU、Leaky ReLU 这样的非线性激活函数的网络。

He Initialization

```
W1 = np.random.normal(0, np.sqrt(2/X1_len), (H1_len, X1_len))
W1[:, 0] = 0.0
# W2:(256, 513)
W2 = np.random.normal(0, np.sqrt(2/X2_len), (H2_len, X2_len))
W2[:, 0] = 0.0
# W3:(128, 257)
W3 = np.random.normal(0, np.sqrt(2/X3_len), (H3_len, X3_len))
W3[:, 0] = 0.0
# W4:(64, 129)
W4 = np.random.normal(0, np.sqrt(2/X4_len), (H4_len, X4_len))
W4[:, 0] = 0.0
```

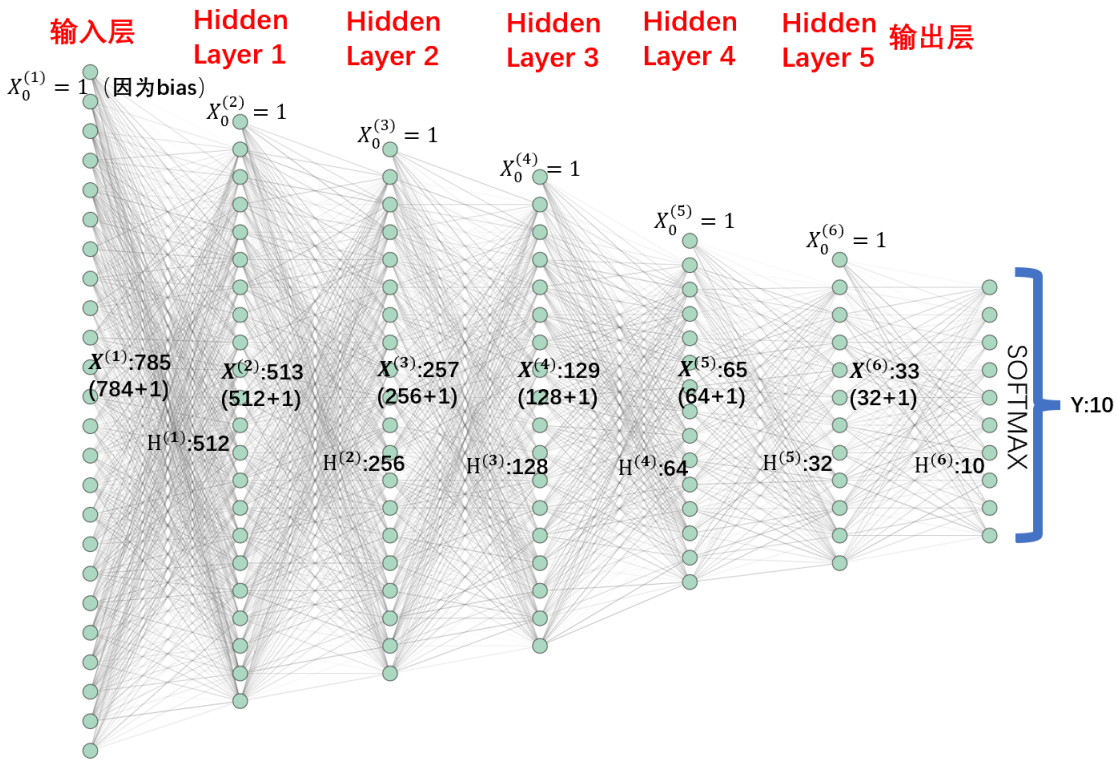

图 3: 6 层 MLP 网络

2.3 对以上调整方法进行测试对比

激活函数	初始化方法	学习率	平均损失	平均准确率
Sigmoid	N(0,0.1)	可变 10 万步 10e-3 到 10e-6	0.1892	0.9444
Sigmoid	Xavier	可变 10 万步 10e-3 到 10e-6	0.3105	0.9084
ReLU	N(0,0.1)	可变 1.5 万步 10e-3 到 10e-7	0.1666	0.9495
ReLU	Xavier	可变 1.5 万步 10e-3 到 10e-7	0.1652	0.9500
ReLU	He Initialization	可变 1.5 万步 10e-3 到 10e-7	0.1553	0.9521
ReLU	He Initialization	恒为 10e-3	0.1041	0.9723

其中，需要指出的是，激活函数为 Sigmoid，初始化方法为 Xavier，学习率为可变 10 万步 10e-3 到 10e-6 时，训练损失下降速度较慢，可能需要进行更多的训练 epoch。

3 第三部分：构建一个 6 层的 MLP 网络

基于之前 4 层的神经网络进行修改，其激活函数为 ReLU，初始化方法使用 He Initialization 方法，学习率为恒定 10e-3，层数为 6 层，其结构为图 3。

前馈网络结构为:

6 层 MLP 网络结构

```
        # 给定输入，拼接并计算输出值
        X1 = np.concatenate((np.array([1]), img)) # X1:(785,)
        H1 = np.matmul(W1, X1)  # H1:(512,)
        X2 = np.concatenate((np.array([1]), ReLU(H1)))   # X2:(513,)
        H2 = np.matmul(W2, X2)  # H2:(256,)
        X3 = np.concatenate((np.array([1]), ReLU(H2)))   # X3:(257,)
        H3 = np.matmul(W3, X3)  # H3:(128,)
        X4 = np.concatenate((np.array([1]), ReLU(H3)))   # X4:(129,)
        H4 = np.matmul(W4, X4)  # H4:(64,)
        X5 = np.concatenate((np.array([1]), ReLU(H4)))   # X5:(65,)
        H5 = np.matmul(W5, X5)  # H5:(32,)
        X6 = np.concatenate((np.array([1]), ReLU(H5)))   # X6:(33,)
        H6 = np.matmul(W6, X6)  # H6:(10,)
        Y_hat = softmax(H6) # Y_hat:(10,)
```

反向传播：

6 层 MLP 网络梯度下降反向传播训练过程

```
step = 0
for i in range(epoch_num):
    j = 0
    every_steps = 100
    losses = np.zeros((every_steps,))
    for (img, Y) in zip(train_images, train_labels):
        # 给定输入，拼接并计算输出值
        X1 = np.concatenate((np.array([1]), img)) # X1:(785,)
        H1 = np.matmul(W1, X1)  # H1:(512,)
        X2 = np.concatenate((np.array([1]), ReLU(H1)))   # X2:(513,)
        H2 = np.matmul(W2, X2)  # H2:(256,)
        X3 = np.concatenate((np.array([1]), ReLU(H2)))   # X3:(257,)
        H3 = np.matmul(W3, X3)  # H3:(128,)
        X4 = np.concatenate((np.array([1]), ReLU(H3)))   # X4:(129,)
        H4 = np.matmul(W4, X4)  # H4:(64,)
        X5 = np.concatenate((np.array([1]), ReLU(H4)))   # X5:(65,)
        H5 = np.matmul(W5, X5)  # H5:(32,)
        X6 = np.concatenate((np.array([1]), ReLU(H5)))   # X6:(33,)
        H6 = np.matmul(W6, X6)  # H6:(10,)
        Y_hat = softmax(H6) # Y_hat:(10,)

        # 根据真实标签Y（独热编码）与预测标签Y_hat计算损失函数值
        loss = CrossEntropy(Y, Y_hat)

        # 计算各个梯度
        # 最后一层的特殊处理，直接计算交叉熵损失关于输入softmax的最后一层隐
            藏层的结点向量last_H的导数
```

```
        dL_H6 = dCrossEntropy_lastH_softmax(Y, Y_hat)   # (10,)
        dL_X6 = np.matmul(dL_H6, W6)    # (33,)
        dL_H5 = dL_X6[1:] * dReLU(H5)    # (32,)
        dL_X5 = np.matmul(dL_H5, W5)    # (65,)
        dL_H4 = dL_X5[1:] * dReLU(H4)    # (64,)
        dL_X4 = np.matmul(dL_H4, W4)    # (129,)
        dL_H3 = dL_X4[1:] * dReLU(H3)    # (128,)
        dL_X3 = np.matmul(dL_H3, W3)    # (257,)
        dL_H2 = dL_X3[1:] * dReLU(H2)    # (256,)
        dL_X2 = np.matmul(dL_H2, W2)    # (513,)
        dL_H1 = dL_X2[1:] * dReLU(H1)    # (512,)
        dL_X1 = np.matmul(dL_H1, W1)    # (785,)

        # 计算权重矩阵的梯度
        dL_W6 = np.matmul(dL_H6.reshape(-1, 1), X6.reshape(1, -1))
        dL_W5 = np.matmul(dL_H5.reshape(-1, 1), X5.reshape(1, -1))
        dL_W4 = np.matmul(dL_H4.reshape(-1, 1), X4.reshape(1, -1))
        dL_W3 = np.matmul(dL_H3.reshape(-1, 1), X3.reshape(1, -1))
        dL_W2 = np.matmul(dL_H2.reshape(-1, 1), X2.reshape(1, -1))
        dL_W1 = np.matmul(dL_H1.reshape(-1, 1), X1.reshape(1, -1))

        # 更新权重矩阵
        W6 -= lr * dL_W6
        W5 -= lr * dL_W5
        W4 -= lr * dL_W4
        W3 -= lr * dL_W3
        W2 -= lr * dL_W2
        W1 -= lr * dL_W1

        # 每隔every_steps轮，输出一下loss结果
        losses[j % every_steps] = loss
        j += 1
        if step < 15000-1:
            step += 1
        if j % every_steps == 0:
            print('Epoch{} IMG{}to{}, Average Loss:{}'.format(i, j-
                every_steps, j, np.mean(losses)))
```

训练方式与前述相同，训练结果如下：

6 层 MLP 训练 3 个 Epoch 的测试结果

```
测试集平均损失为0.15302432550184034，测试准确率为0.9618
```

4 实验心得体会

第一，反向传播过程是整个训练较难理解的部分，其困难主要在于多个参数之间的连接关系比较复杂，因此常常需要使用矩阵进行运算。在推导反向传播原理的时候，数学公式通常需要保证标量对于向量/矩阵的梯度形式满足分子布局形式，所以在写的时候需要保证每一步都清楚矩阵的形式。反向传播过程中，为了简化编程操作，可以先从数学原理上针对性地对激活函数的类型、损失函数的类型进行求导简化。另外，在实际编程过程中，需要注意有时候需要使用矩阵乘法，有的需要按元素相乘，具体如何判断，可以从元素到向量/矩阵进行推广，即主要关注向量/矩阵中的每一个元素值本身如何产生，然后再考虑矩阵之间如何操作得到目标向量/矩阵。

第二，初始化权重对于训练结果是否收敛具有重要影响。通过查阅资料，发现主流的 tanh 和 sigmoid 最好是使用 xavier 初始化方法；何恺明团队提出的 He initialization 方法则对 ReLU 及衍生的若干非线性函数下的权重初始化效果较好。

第三，学习率对结果的影响也比较大。在第一部分和第二部分的前面几个代码中，我训练的时候发现到了 15000 步开始平均损失就下降得很缓慢了，我判断这是因为后面产生了振荡，所以采用了可变学习率，避免振荡，让损失稳定下降。但是最后我尝试保持较大的学习率 10e-3，发现在同样 3 个 epoch 的情况下训练收敛程度更好，这说明 3 个 epoch 可能还不太够，适当增加 epoch 可能能够保证泛化性情况下测试结果会更好。