神经网络 第四章大作业

201300096 杜兴豪 人工智能学院

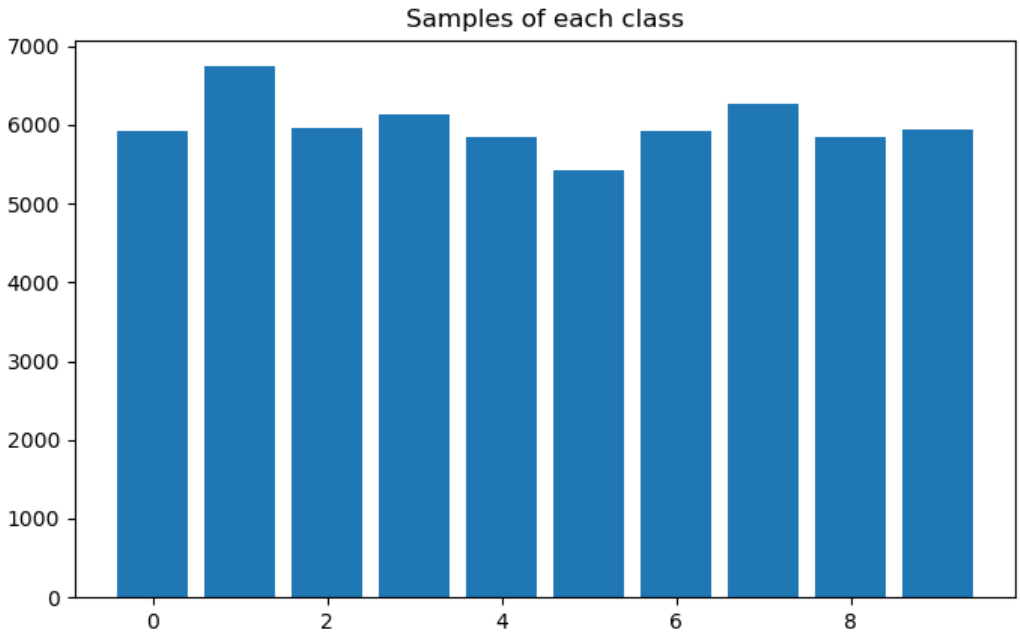
实验总结

在这次实验中，我利用numpy库自己完成了多层感知机中线性层（以及三种权重初始化方法）、激活函数（包含ReLU和Sigmoid）以及相对应的反向传播算法的所有定义和代码实现，**所有部分均为自己实现。**

根据上述内容构建了MLP模型，并用以MNIST手写数字分类，在200个epoch内达到了99.99%的预测集准确率和98%的测试集分类准确率。

数据集分析

MNIST手写数据集包含10个类别，训练集有60000条数据，观察每个类别的数据个数如图：



发现不存在类别不平衡的情况，因此不需要对数据做过多处理。

模型介绍

在对比ReLU和Sigmoid时的我采用的MLP模型的结构大致为：

1. 线性层：784×100
2. 激活函数（ReLU或Sigmoid）
3. 线性层：100×10

参数量为

$$784 \times 100 + 100 + 100 \times 10 = 79500$$

训练最高得分时采用的模型结构中，隐藏层大小做了改变：改为3000维，参数量为

$$784 \times 3000 + 3000 + 3000 \times 10 = 2385000$$

模型的初始化代码定义为

```
class MLP:
    def __init__(self,input_size,               # 输入神经元数
                 hidden_size,                   # 隐藏层神经元数
                 output_size,                   # 输出神经元数
                 learning_rate=1e-2,            # 学习率
                 epochs=20,                     # 最大迭代轮数
                 activate='ReLU',               # 激活函数，默认为ReLU
                 batch_size=20,                 # 批次大小
                 init_method='uniform')         # 权重初始化方法，可选normal、uniform、xavier
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.W1 = np.zeros((input_size, hidden_size))                   # 输入层到隐藏层的权重
        self.b1 = np.zeros(hidden_size)                                 # 隐藏层的偏置
        self.W2 = np.zeros((hidden_size, output_size))                  # 隐藏层到输出层的权重
        self.b2 = np.zeros(output_size)                                 # 输出层的偏置
        self.hidden = None                                              # 隐藏层的输出
        self.output = None                                              # 模型输出
        self.lr = learning_rate                                         # 学习率
        self.MAX_EPOCHS = epochs                                        # 训练轮数
        self.activate = ReLU() if activate == 'ReLU' else Sigmoid()     # 激活函数
        self.batch_size = batch_size                                    # 分批投入数据
        self._init_parameters(init_method)                              # 权重初始化
```

在初始化部分中保存了之后的过程中需要的变量（如学习率，最大迭代轮数、批次大小等），同时初始化部分中的`self.W1,self.b1,self.W2,self.b2`告诉用户：这个模型采用的结构包含两个线性层。

注意到初始化中提到了另外的权重初始化方法`_init_parameters(method)`，该方法根据传入的参数method来以不同的方法初始化线性层的权重。

```
    def _init_parameters(self, method='uniform'):
        if method == 'normal':
            # 正态分布初始化
            self.W1 = np.random.randn(self.input_size, self.hidden_size)
            self.W2 = np.random.randn(self.hidden_size, self.output_size)
        elif method == 'uniform':
            # 均匀分布初始化
            self.W1 = np.random.uniform(low=-0.1, high=0.1, size=(self.input_size,
self.hidden_size))
            self.W2 = np.random.uniform(low=-0.1, high=0.1, size=(self.hidden_size,
self.output_size))
        elif method == 'xavier':
            # xavier初始化
            std = np.sqrt(2.0 / (self.input_size + self.hidden_size))
```

```
            self.w1 = np.random.normal(loc=0.0, scale=std, size=(self.input_size,
self.hidden_size))
            std = np.sqrt(2.0 / (self.hidden_size + self.output_size))
            self.w2 = np.random.normal(loc=0.0, scale=std, size=(self.hidden_size,
self.output_size))
```

起初的实现是全部采用正态分布的，后来发现采用正态分布的收敛速度很慢，才逐步增加了后面两种初始化方法，收敛速度都很快。

模型的前向传播和反向传播过程仅利用到初始化时存在的变量，便于管理。具体代码如下

```
    def forward(self, x):
        # 前向传播过程
        self.hidden = self.activate(np.dot(x, self.w1) + self.b1)   # o1 = activate(w1x+b1)
        self.output = np.dot(self.hidden, self.w2) + self.b2        # o2 = w2o1+b2
        return self.output

    def train(self, x_train, y_train, x_test, y_test):
        num_classes = self.w2.shape[1]                   # 类别总数
        y_train_encoded = np.eye(num_classes)[y_train]  # 独热编码

        epochs, train_accs, test_accs = [], [], []       # 记录输出轮和准确率，用于绘图
        for epoch in range(self.MAX_EPOCHS):
            start = 0                                    # 定义该批次的起点
            epoch_loss = 0                               # 统计批次损失和
            while start < x_train.shape[0]:
                # 取出该批次数据
                batch_x, batch_y = x_train[start:start + self.batch_size], \
                    y_train_encoded[start:start + self.batch_size]
                # 前向传播
                output = self.forward(batch_x)
                # 计算损失函数
                loss = np.mean((output - batch_y) ** 2)

                # 反向传播
                # grad_output = 2 * (output - batch_y) / x_train.shape[0]
                grad_output = output - batch_y  # 取消系数
                grad_w2 = np.dot(self.hidden.T, grad_output)
                grad_b2 = np.sum(grad_output, axis=0)
                grad_hidden = np.dot(grad_output, self.w2.T) *
self.activate.differential(self.hidden)
                grad_w1 = np.dot(batch_x.T, grad_hidden)
                grad_b1 = np.sum(grad_hidden, axis=0)

                # 参数更新
                self.w2 -= self.lr * grad_w2
                self.b2 -= self.lr * grad_b2
                self.w1 -= self.lr * grad_w1
                self.b1 -= self.lr * grad_b1

                start += self.batch_size                 # 更新起点
                epoch_loss += loss * self.batch_size     # 累加轮次损失和

            if epoch % 1 == 0:
                # 输出观察收敛情况
                train_acc = self.evaluate(x_train, y_train) * 100
                test_acc = self.evaluate(x_test, y_test) * 100
                print(f"Epoch {epoch + 1}: Loss={epoch_loss / x_train.shape[0]:.4f}, "
                      f"train accuracy={train_acc:.2f}% "
                      f"test accuracy={test_acc:.2f}% ")
                epochs.append(epoch)
                train_accs.append(train_acc)
```

```
                test_accs.append(test_acc)
        return epochs, train_accs, test_accs
```

其中反向传播需要理解。由于我定义的模型有两层，损失函数采用MSE（为了求导方便）因此每一层的输出关系为：

$$
\begin{aligned}
output_1 &= f(xW_1 + b_1) \\
output_2 &= output_1 W_2 + b_2 = f(xW_1 + b_1)W_2 + b_2
\end{aligned}
$$

第二层的输出即为总输出，第一层输出为隐层输出，因此代码中有这样的对应关系：

$$
\begin{aligned}
\text{self.hidden} &= output_1 = f(xW_1 + b_1) \\
\text{self.output} &= output_2 = f(xW_1 + b_1)W_2 + b_2
\end{aligned}
$$

同时由于损失函数为均方误差损失，因此有

$$
\text{loss} = \frac{1}{|D|}\sum_{i\in D}(predict_i - target_i)^2 = \frac{1}{|D|}(\text{self.output} - \text{self.target})^2
$$

梯度反向传播的过程遵循链式法则，则有

$$
\mathrm{d}W_2 = \frac{\partial \text{loss}}{\partial W_2} = \frac{\partial \text{loss}}{\partial \text{self.output}} \cdot \frac{\partial \text{self.output}}{\partial W_2} = \frac{2}{|D|}(\text{self.output} - \text{self.target})^T(\text{self.hidden})
$$

在实际计算中，忽略前面的系数项，用学习率来规定步长，即

$$
\nabla W_2 = (\text{self.output} - \text{self.target})^T(\text{self.hidden})
$$

同理：

$$
\nabla b_2 = \frac{\partial \text{loss}}{\partial \text{self.output}} \cdot \frac{\partial \text{self.output}}{\partial b_2} = \frac{2}{|D|}\|\text{self.output} - \text{self.target}\|
$$

由此可以计算损失对隐藏层的导数情况：

$$
\begin{aligned}
\nabla W_1 &= \frac{\partial \text{loss}}{\partial \text{self.output}} \cdot \frac{\partial \text{self.output}}{\partial \text{self.hidden}} \cdot \frac{\partial \text{self.hidden}}{\partial (xW_1 + b_1)} \cdot \frac{\partial (xW_1 + b_1)}{\partial W_1} = (\text{self.output} - \text{self.target})^T W_2 f'(W_1)^T x \\
\nabla b_1 &= \frac{\partial \text{loss}}{\partial \text{self.output}} \cdot \frac{\partial \text{self.output}}{\partial \text{self.hidden}} \cdot \frac{\partial \text{self.hidden}}{\partial (xW_1 + b_1)} \cdot \frac{\partial (xW_1 + b_1)}{\partial b_1} = (\text{self.output} - \text{self.target})^T W_2 f'(W_1)^T
\end{aligned}
$$

除了激活函数的导函数之外的所有内容都已经获得。Sigmoid函数和ReLU的导函数定义如下：

$$
\begin{aligned}
\sigma'(x) &= \sigma(x)(1 - \sigma(x)) \\
\mathrm{ReLU}'(x) &= \begin{cases} 1 & x > 0 \\ 0 & x \le 0 \end{cases}
\end{aligned}
$$

在activation.py中可以找到激活函数及其导函数的定义：

```
class Sigmoid:
    # def __call__(self, x):
    #     # 出现指数溢出错误，因为-x可能很大
    #     return 1/(1+np.exp(-x))
    def __call__(self, x):
        # 分别处理正值和负值，保证exp(.) < 1
        indices_pos = np.nonzero(x >= 0)
        indices_neg = np.nonzero(x < 0)

        y = np.zeros_like(x)
        y[indices_pos] = 1 / (1 + np.exp(-x[indices_pos]))                    # 对正值计算1 /
(1+exp(-x)
        y[indices_neg] = np.exp(x[indices_neg]) / (1 + np.exp(x[indices_neg]))  # 对负值计算
exp(x) / (1+exp(x))

        return y

    def differential(self, x):
```

```
        # Sigmoid的导数为sgimoid'(x) = sigmoid(x) * (1-sigmoid(x))
        return self(x) * (1-self(x))


class ReLU:
    def __call__(self, x):
        return np.maximum(x, 0)

    def differential(self, x):
        # ReLU的导数：大于0的为1，小于等于0的为0
        pos_x = x + np.abs(x)       # 获得非负矩阵，其中正值翻倍，负值变0
        pos_x[pos_x > 0] = 1        # 正值部分全为1
        return pos_x
```

由此梯度计算公式完全可得。代码实现在MLP的`train()`方法中。

模型剩下的部分为预测和评估部分：

```
    def predict(self, x):
        # 将10维输出转化为1维标签
        return np.argmax(self.forward(x), axis=1)

    def evaluate(self, x_test, y_test):
        # 评估模型由输入数据得到的输出标签和真实标签的准确率
        predictions = self.predict(x_test)
        accuracy = np.mean(predictions == y_test)
        return accuracy
```

代码运行逻辑

文件组织

我的实现分别位于四个文件中，框架如下：

- main.py：主文件，运行该文件以获得对MNIST手写数据集的分类
- network.py：定义网络结构、训练过程、预测过程
- activation.py：定义激活函数及其求导后的函数
- utils.py：定义MNIST数据集的读取和预处理函数、以及数据可视化函数

运行逻辑解读

main.py文件的主体如下：

```
if __name__ == '__main__':
    # 载入MNIST数据集
    x_train, y_train, x_test, y_test = load_mnist()

    # 创建MLP模型
    activations = ['ReLU', 'Sigmoid']           # 所有激活函数
    lines = {'ReLU': 'c*-', 'Sigmoid': 'm.-'}   # 用于绘制不同形式的曲线
    for activate in activations:
        print(f"Training MLP using activation [{activate}]")
        lr = 1e-2 if activate == 'ReLU' else 1e-3
        model = MLP(input_size=784,             # 输入神经元数
                    hidden_size=100,            # 隐藏层神经元数
                    output_size=10,             # 输出神经元数
                    learning_rate=lr,           # 学习率
                    epochs=20,                  # 最大迭代轮数
                    activate=activate,          # 激活函数
                    batch_size=20,              # 批次大小
                    init_method='xavier')       # 权重初始化方法，可选normal、uniform、xavier
```

```
        # 训练模型
        epochs, train_accs, test_accs = model.train(x_train, y_train, x_test, y_test)
        # 绘图
        plt.subplot(121)        # 在第一个子图中绘制迭代轮数-训练集准确率图
        plt.plot(epochs, train_accs, lines[activate], label=activate)
        plt.title('Accuracy on Train Set')
        plt.xlabel('Epochs')
        plt.ylabel('Accuracy')
        plt.ylim((70, 100))
        plt.legend()

        plt.subplot(122)        # 在第二个子图中绘制迭代轮数-测试集准确率图
        plt.plot(epochs, test_accs, lines[activate], label=activate)
        plt.title('Accuracy on Test Set')
        plt.xlabel('Epochs')
        plt.ylabel('Accuracy')
        plt.ylim((70, 100))
        plt.legend()
    plt.show()
```

运行main.py文件，可以看到首先从`load_mnist()`函数中获取训练集和测试集。

```
def load_mnist(preprocess=True):
    train_set = torchvision.datasets.MNIST(
        root='MNIST',
        train=True,
        transform=torchvision.transforms.ToTensor(),
        download=True
    )
    test_set = torchvision.datasets.MNIST(
        root='MNIST',
        train=False,
        transform=torchvision.transforms.ToTensor(),
        download=True
    )
    train_x, train_y = train_set.data.numpy(), train_set.targets.numpy()
    test_x, test_y = test_set.data.numpy(), test_set.targets.numpy()    # 转成numpy数据

    if preprocess:  # 对数据进行预处理：将特征变化到一个维度上、对输出数据做归一化、打乱训练集
        train_x, test_x = train_x.reshape(train_x.shape[0], -1),
test_x.reshape(test_x.shape[0], -1)    # 合并后两个维度
        train_x, test_x = train_x / 255.0, test_x / 255.0               # 像素值小于255，做归一
化
        indexes = list(range(train_x.shape[0]))
        random.shuffle(indexes)
        train_x, train_y = train_x[indexes], train_y[indexes]           # 打乱数据集
    return train_x, train_y, test_x, test_y
```

- 若不对数据做预处理，则得到原始数据集，可以用来做数据可视化。预处理后的数据传入模型中用于训练。

得到数据后，可以看到主要框架为for循环：遍历activation列表中的每一种激活函数，分别利用这些激活函数构造MLP，在相同的其他超参数下训练（这里在之后的训练中改为不同的学习率，因为Sigmoid所需要的学习率低于ReLU）。训练函数的返回值为每次输出中的轮数、训练集准确率和测试集准确率，得到这些数据后绘制迭代轮数-准确率图。

环境依赖及运行方法

- 环境：python3+numpy+matplotlib（均可使用pip或conda命令直接安装）
- 运行方法：命令行输入`python main.py`或IDE中运行main.py文件即可

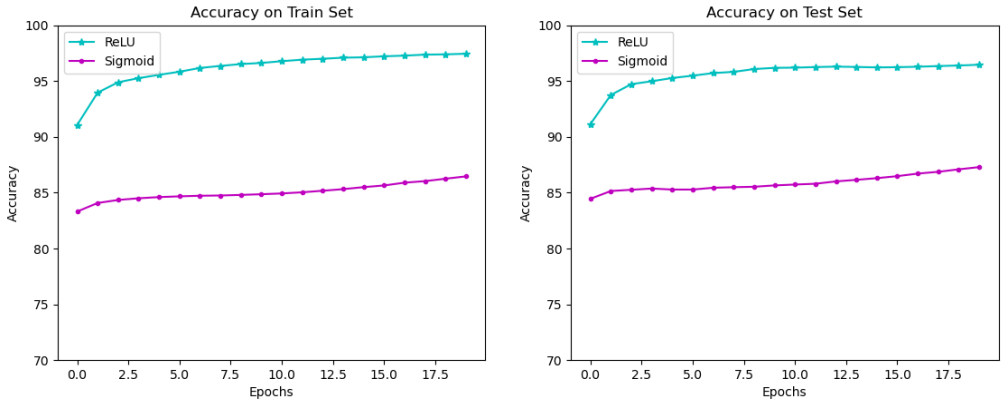
实验结果

结果展示

对比时采用的模型参数如下：

- 隐层大小：100
- 最大训练轮数：20
- 激活函数：ReLU和Sigomid
- 学习率：0.01（ReLU），0.001（Sigmoid）
- 权重初始化方法：xavier

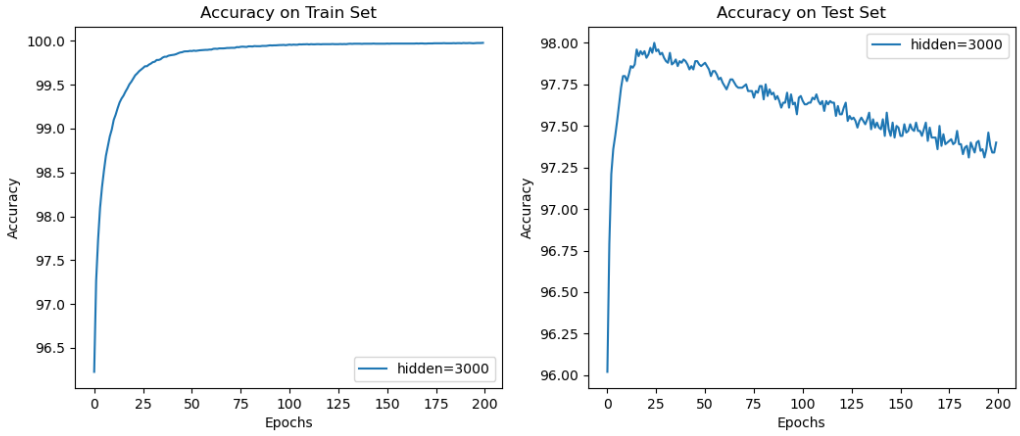
训练过程中的训练集和测试集准确率对比如图：



最终结果时采用的模型参数如下：

- 隐层大小：3000
- 最大训练轮数：200
- 激活函数：ReLU
- 学习率：0.01
- 权重初始化方法：xavier

训练结果如图：

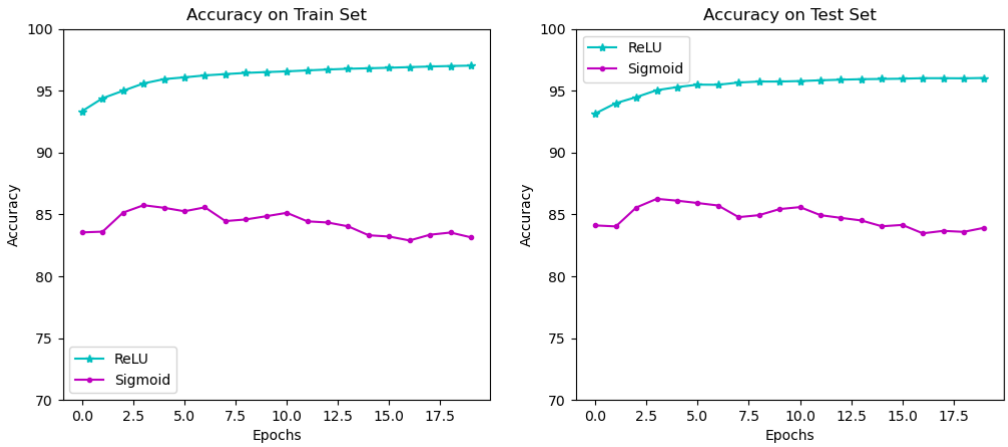


训练集上的准确率随着轮数增加不断上升，到最后一轮时达到99.99%；测试集上的准确率随着轮数增加而先上升后下降，最高时达到98.0%，判断在后期出现了过拟合，模型过度关注训练集上的数据，导致泛化能力下降，因此在测试集上表现不好。

结果分析

激活函数上的分析

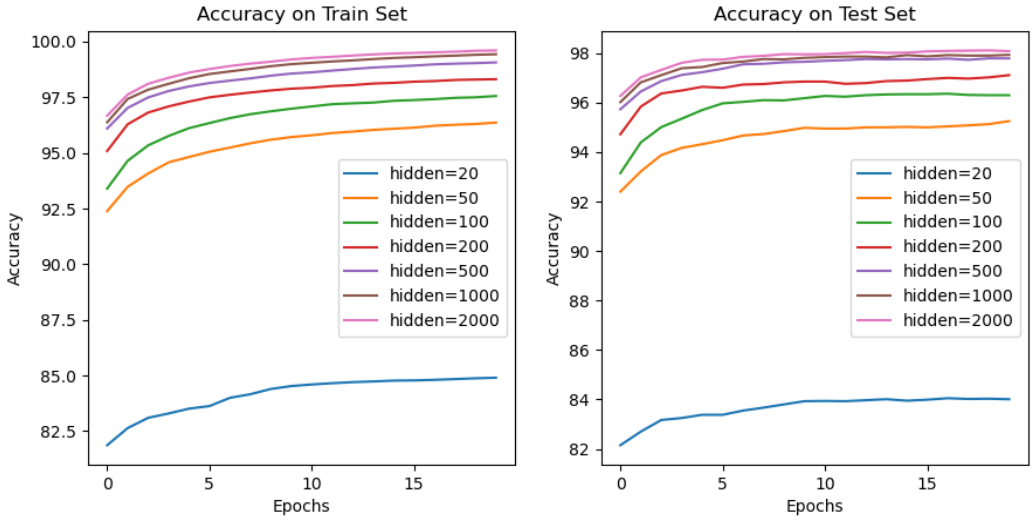
观察到ReLU作为激活函数时，比Sigmoid更快达到收敛，效果更好。如果要采用控制变量法，强行让Sigmoid的学习率也为0.01，则会造成这样的情况：



观察到Sigmoid作为激活函数时出现了准确率下降的趋势，判断为模型过拟合（此时Sigmoid的分类准确率仍然低于ReLU）得到结论：选择适当的激活函数对于模型的收敛非常关键，ReLU比Sigmoid更适合在MNIST数据集上分类时作激活函数。

模型参数的影响

首先分析隐藏神经元的个数（宽度）的影响：

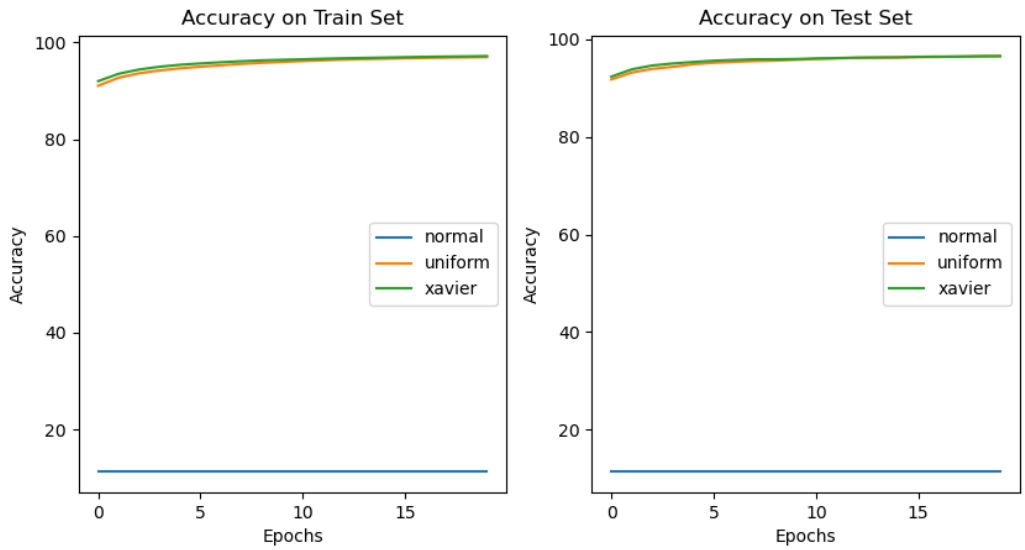


观察到

- 当隐藏神经元个数较小时，模型过于简单，无法较好地学习到MNIST数据集中的隐含特征，导致在训练集和测试集上的准确率都偏低。
- 当隐藏神经元个数达到一定值后，模型都可以达到很好的分类准确率，此时需要综合考虑训练时间的影响：当隐层较小时，训练非常快，但隐层很大时，每训练一轮都需要大量的时间开销。

总体上来说，增大网络宽度对提升精度有帮助，但计算开销和时间开销也随之增加，需要综合考量。

再分析权重初始化方法的影响：



观察到均匀分布初始化和xavier初始化在这个问题上效果类似，而正态分布初始化的效果很差。在选择初始化方法时，一定要多加尝试，选择最适合特定问题的初始化方法。

如何判断过拟合还是欠拟合

由于框架代码中包含绘制训练集上准确率的图和测试集上准确率的图，因此很容易分辨是过拟合还是欠拟合。

- 当在训练集上和测试集上的准确率都较低时，模型还未达到收敛，此时是欠拟合的
- 当在训练集上的准确率较高，而测试集的准确率先升高后下降时，判断模型发生了过拟合现象。

相关的图可以参考结果展示部分：epoch=20的那张图中模型还未收敛，此时训练集和测试集上的准确率都在不断升高；epoch=200的那张图中，测试集准确率有明显的先上升后下降过程，是非常经典的过拟合样例。

当模型发生欠拟合时，可以通过增大学习率的方法提高拟合速度；当模型发生过拟合时，可以检查模型结构，调大隐藏层或减小学习率，来避免模型迅速达到过拟合状态。

遇到的问题及改进

Sigmoid实现问题

Sigmoid的公式很简单：

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

但在实现过程中容易发现：当输入的x为一个很大的负数时，e^{-x}取值将非常大，造成数值溢出错误。因此必须要处理负数情况，考虑到Sigmoid函数的等价变形：

$$\sigma(x) = \frac{e^{x}}{e^{x}+1}$$

可以完美去除e^{-x}的问题，因此我们只要考虑分别对正负输入带入不同的公式即可。但如果我们对输入矩阵逐元素进行判断，则时间开销将特别大，因此需要考虑使用`numpy.array`的特性，即广播和数组索引特性：利用广播特性分别找出所有正值和负值，存在不同的索引数组中，再直接利用数组索引将输入分成两部分，最后再利用数组索引和广播功能将运算结果赋值给返回值，即可完美完成这部分内容。（代码在模型结构部分的最后）

心得体会

这次大作业让我们从零开始构建了整个多层感知机网络，让我体验到了第一批造轮子的人的艰难和不易，同时也让我感受到Pytorch框架帮我们把训练深度学习模型的难度降低到了何种程度，以后我会更加珍惜这些工具，并且更加刻苦地学习知识，希望在未来自己也有机会制造出造福大家的工具。