基于 Numpy 的手写数字分类

王森 MG21370034

2022 年 1 月 13 日

1 实验目标

在本次实验中，我们要做 MNIST 数据集的分类任务。MNIST 数据集主要由一些手写数字的图片和相应的标签组成，图片一共有 10 类，分别对应从 0～9 ，共 10 个阿拉伯数字。在 MNIST 数据集中有 60000 张训练图片，10000 张测试图片，每张图片大小 28*28，此数据集可以在 Yann LeCun 的个人主页上自由下载。

与之前所做的图片分类任务不同，本次实验不能（不建议）使用像 pytorch 和 tensorflow 这种现有的深度学习框架，而是要自己手写一个多层感知机的神经网络，包括前向传播和后向传播等都需要自己手动实现。在构建完基础的网络之后，还要实现若干优化方法，最后对这些优化方法进行分析和比较。

2 实验内容

2.1 Numpy 构建 MLP

2.1.1 理论基础

本次实验中，我使用了 python 语言中的 numpy 工具包来构建 MLP 神经网络。手动构建全连接神经网络主要包括三个部分：前向传播计算神经网络的输出，反向传播计算每一层的梯度，根据梯度更新神经网络的参数。在清晰了 MLP 的原理之后，代码的书写会变得很简单，因此，我们有必要

复习一下 MLP 的数学推导过程。一个具有单隐层的 MLP 如图1所示。这个神经网络在实际应用时的含义是：输入一个长度为 4 的数据，输出一个长度为 3 的结果。

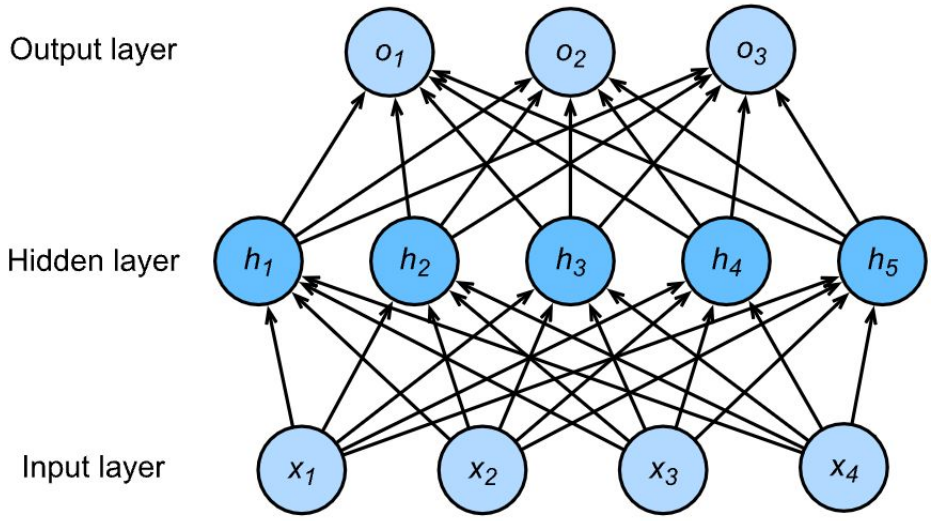

图 1: 单隐层神经网络

从图1中可以看出，此神经网络总共有 3 层，分别是输入层，隐藏层和输出层，并且相邻的两层之间通过全连接的方式联系到一起。下面将以图1为例，对 MLP 的前向过程和后向过程进行数学推导。

假设神经网络的输入为 X，在图1中 X 是一个长度为 4 的向量。因为隐藏层总共有 5 个神经元，所以输入层到隐藏层之间的权重系数 $W^{(1)}$ 是一个 4*5 的矩阵，偏置系数 $b^{(1)}$ 是一个长度为 5 的向量。输入数据由输入层到达隐藏层之后，输出的结果 $h^{(2)}$ 为：

$$h^{(2)} = X \cdot W^{(1)} + b^{(1)} \tag{1}$$

公式1中的 $\cdot$ 为矩阵相乘。根据矩阵乘法的规则，X 的维度是 1*4，$W^{(1)}$ 的维度为 4*5，那么 $h^{(2)}$ 的维度为 1*5。从图1中可以看出，隐藏层神经元的个数为 5，也就是说隐藏层会输出长度为 5 的向量，这与我们根据乘法规则推导出来的结果是一致的。

从隐藏层到输出层的推导与上述过程类似，只不过现在隐藏层的输出成为了下一层的输入。因为隐藏层有 5 个神经元，输出层有 3 个神经元，所

以隐藏层到输出层之间的权重系数 $W^{(2)}$ 是一个 5*3 的矩阵，偏置系数 $b^{(2)}$ 是一个长度为 3 的向量。假设输出层最终输出的结果为 $O^{(3)}$，那么输出的计算过程为：

$$O^{(3)} = h^{(2)} * W^{(2)} + b^{(2)} \tag{2}$$

与公式1相同的分析方法，$O^{(3)}$ 的维度为 1*3。

到目前为止，图1所表示的神经网络的前向过程已经推导完毕了。下面做一个总结，对于三层的神经网络，假如输入数据是 X，那么输出数据 $O^{(3)}$ 为：

$$O^{(3)} = (X \cdot W^{(1)} + b^{(1)}) \cdot W^{(2)} + b^{(2)} \tag{3}$$

为了使问题更具泛化性，我们在推导反向传播过程时，会以图1为例子，但不再完全依赖于图1，而是会将问题做进一步的形式化。

定义损失函数为 $L(y, y^*)$，其中 y^* 为样本的真实标记，y 为样本的预测值。记损失函数 L 关于第 i 层神经元的输出 z^i 的偏导为 $\delta^i = \frac{\partial L}{\partial z^i}$，那么损失函数在最后一层参数上关于权重的偏导为：

$$\begin{aligned} &\frac{\partial L}{\partial W_{i,j}^{n-1}} \\ =&\frac{\partial L}{z_j^n} * \frac{\partial z_j^n}{\partial W_{i,j}^{n-1}} \\ =&\delta_j^n * \frac{\partial (\sum_{k=1}^{|l_{n-1}|} z_k^{n-1} W_{k,j}^{n-1} + b_j^{n-1})}{\partial W_{i,j}^{n-1}} \\ =&\delta_j^n * z_i^{n-1} \end{aligned} \tag{4}$$

公式4的表达形式是对单个权重 $W_{i,j}^{n-1}$ 的导数，而如果写成矩阵的形式，则为：

$$\frac{\partial L}{\partial W^{n-1}} = (z^{n-1})^T \delta_j^n \tag{5}$$

更一般的，损失函数 L 关于第 l 层的权重的偏导为：

$$\frac{\partial L}{\partial W^l} = \frac{\partial L}{\partial z^{l+1}} * \frac{\partial z^{l+1}}{\partial W^l} = (z^l)^T \delta^{l+1} \tag{6}$$

同理可得损失函数 L 关于第 l 层的偏置的偏导为：

$$\frac{\partial L}{\partial b^l} = \delta^{l+1} \tag{7}$$

现在还差最后一个问题没有解决，即 $\delta^l = \frac{\partial L}{\partial z^l}$ 的更新公式，下面进行推导。

$$
\begin{aligned}
&\delta_i^l = \frac{\partial L}{\partial z_i^l} \\
=&\frac{\partial L}{\partial z^{l+1}} * \frac{\partial z^{l+1}}{\partial z_i^l} \\
=&\frac{\partial L}{\partial z^{l+1}} * \frac{\partial (z^l W^l + b^l)}{\partial z_i^l} \\
=&\sum_{j=1}^{|l_{l+1}|} \frac{\partial L}{\partial z_j^{l+1}} * \frac{\partial (\sum_{k=1}^{|l_l|} z_k^l W_{k,j}^l + b_j^l)}{\partial z_i^l} \\
=&\sum_{j=1}^{|l_{l+1}|} \delta_j^{l+1} * W_{i,j}^l \\
=&\delta^{l+1}((W^l)^T)_i
\end{aligned}
\tag{8}
$$

公式8是损失函数 L 对第 l 层的第 i 个神经元的导数，更一般化的表示形式为：

$$
\delta^l = \frac{\partial L}{\partial z^l} = \delta^{l+1}(W^l)^T \tag{9}
$$

至此，神经网络的反向传播过程已经推导完毕了，下面做一个简要地总结。损失函数 L 关于第 l 层神经元的导数，就是第 $l+1$ 层的导数乘上第 l 层权重矩阵的转置，即：

$$
\begin{aligned}
&\delta^l \\
=&\delta^{l+1}(W^l)^T \\
=&\delta^{l+2}(W^{l+1})^T(W^l)^T \\
=&\delta^n(W^{n-1})^T(W^{n-2})^T...(W^l)^T
\end{aligned}
\tag{10}
$$

从公式10中可以看出，我们只要求出损失函数 L 关于最后一层的偏导 δ^n，其它任意层的偏导直接用最后一层的偏导乘上权重的转置即可。有了任意一层的 δ^l 之后，根据公式6和公式7，损失函数 L 关于第 l 层权重 W^l 的偏导为，第 l 层输出的转置乘第 $l+1$ 层的偏导，即：

$$
\frac{\partial L}{\partial W^l} = (z^l)^T \delta^{l+1} \tag{11}
$$

损失函数 L 关于第 l 层偏置 b^l 的偏导就是第 $l+1$ 层的偏导，即：

$$
\frac{\partial L}{\partial b^l} = \delta^{l+1} \tag{12}
$$

到现在为止，多层神经网络的前向传播过程和后向传播过程已经推导完毕，但是在实现的时候，还有几点需要注意的地方。

上述推导过程中，把输入 X 默认的当成是一个向量，即一个输入数据，但是在实际使用过程中，一般不会一次只输入一个数据到神经网络中，一是因为当数据量比较大时，这种训练方式非常耗时，二是因为这样会导致训练过程极其不稳定。所以，一种更合理的做法是，一次输入一批数据，比如一次输入 128 个数据，那么此时神经网络的输入就变成了一个矩阵，其形状为 $128 * m$，其中 128 表示一次把 128 个数据输入到神经网络中，m 表示每个数据的长度为 m。在使用这种方法训练网络时，我们反向传播得到的梯度应该是所有数据梯度的和，因此，将反向传播得到的梯度除以数据的总数得到的才是用于参数更新的梯度。

另一个需要注意的地方是，传统的神经网络的表达力有限，只能处理线性可分的问题，而如果我们想将神经网络应用于更复杂的任务，就必须赋予它非线性的能力。目前一般有两种解决方案，一是做非线性变换，二是使用激活函数。激活函数是神经网络中最常用的增加非线性能力的方法，我在自己的实现中也是选用了激活函数。上述关于前向和后向的推导中，没有使用激活函数，而如果想将激活函数加入到神经网络每一层的输出中，前向过程和后向过程只需要稍微改写一下即可。记激活函数为 $\sigma(x)$，在前向传播过程中，把每一个隐藏层的输出当成激活函数的输入，再把函数值当成新的输出，通过这种方法即可将激活函数加入到前向传播过程中。在后向传播过程中，根据链式法则，现在损失函数 L 对隐藏层 l 求偏导的过程中多了一层激活函数，因此需要将原先的偏导乘上激活函数的导数才是损失函数对隐藏层的偏导，即

$$\delta^l = \frac{\partial L}{\partial z^l} = \delta^{l+1}(W^l)^T \sigma'(z^l) \tag{13}$$

现在的神经网络还差参数更新部分没有完成，基于梯度下降法的参数更新为：

$$\begin{aligned} w &= w - \eta * \frac{\partial L}{\partial w} \\ b &= b - \eta * \frac{\partial L}{\partial b} \end{aligned} \tag{14}$$

其中 η 为学习率。

2.1.2 实验结果

我在实现 MLP 时，没有固定住网络的结构，而是像 pytorch 一样，可以自由地选择使用几层的网络，每层网络有多少个神经元。为了验证自己用 numpy 写的神经网络的正确性，我又用 pytorch 写了一个神经网络（代码文件中的 pytorch_mnist.py 文件），并在相同的超参数设置下，对两个网络的训练效果进行对比，超参数如表1所示。

参数名称	参数值
网络结构	[784, 500, 10]
学习率	0.01
损失函数	MSE
激活函数	sigmoid
epochs	100
batch_size	128
optimizer	SGD

表 1: 神经网络超参设置

为了使实验结果更具说服力，我首先固定了 torch 的随机数种子，然后将网络的初始化参数存储下来，最后用存储下来的初始化参数去初始化用 numpy 实现的神经网络。通过这一系列复杂的操作，可以保证两个网络的设置完全相同，如果我用 numpy 写的神经网络没有 bug，那么训练的效果应该和 pytorch 实现的网络完全相同。

在上述设置下，两个网络的训练效果如图2所示。从图2中可以看出，用 pytorch 框架和我自己用 numpy 实现的神经网络，在前一百个 epochs 上的测试准确率完全相同，这足以说明用 numpy 实现的神经网络的正确性。

2.2 参数初始化

在经过初步的调参尝试之后，选定表2所示的超参数进行不同权重初始化方法的探索。一般在神经网络中，我们将偏置初始化为 0，原因在于该因素对于网络中的梯度流动并无影响。实际上神经网络中，我们需要考虑的

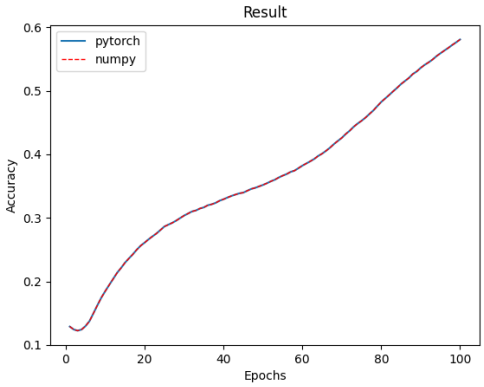

图 2: 两个网络的运行结果

是如何对神经网络中的权重进行初始化。因此，以下所有参数初始化探索都是指对权重的初始化。

参数名称	参数值
网络结构	[784, 500, 10]
学习率	0.1
epochs	20
batch_size	128

表 2: 神经网络超参设置

2.2.1 0 值初始化

0 值初始化，顾名思义，就是将权重的初始值设为 0，训练效果如图3所示。

从图3中可以看出，将网络的权重初始化为 0 会导致网络无法训练。出现这种现象的原因也很好解释，当把权重初始化为 0 时，在第一次前向传播的过程中，0 会一直向前传递，也就是说隐层和输出层的输出都是 0。在反向传播的过程中，根据公式11，如果第 l 层的输出为 0，那么第 l 层的梯度也为 0，既然梯度为 0，那么权重永远不会改变，始终为 0，因此训练效果是图3中的一条直线。

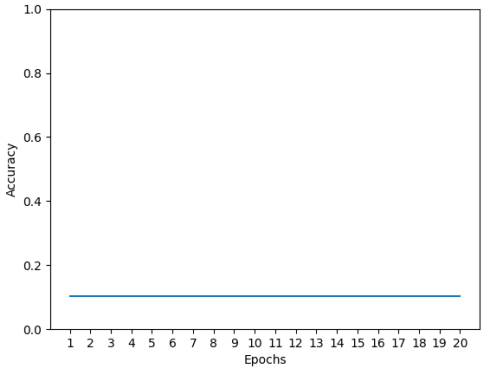

图 3: 0 值初始化

2.2.2 Xavier 初始化

前馈神经网络本质上是一个映射函数，它将原始样本映射成它的类别，即将样本空间映射到类别空间。如果样本空间与类别空间的分布差异很大，比如说类别空间特别稠密，样本空间特别稀疏辽阔，那么在类别空间得到的用于反向传播的误差丢给样本空间后简直变得微不足道，也就是会导致模型的训练非常缓慢。同样，如果类别空间特别稀疏，样本空间特别稠密，那么在类别空间算出来的误差丢给样本空间后简直是爆炸般的存在，即导致模型发散震荡，无法收敛。因此，我们要让样本空间与类别空间的分布差异（密度差别）不要太大，也就是要让它们的方差尽可能相等。Xavier 初始化方法可以达到这个目标。它有两种子方法，分别是从均匀分布中采样和从正态分布中采样，这两种方法的训练效果如图4和图5所示。

从图4和图5中可以看出，基于 uniform 采样的 xavier 方法在最开始的时候训练速度快于基于 normal 采样的 xavier，当然这也不能排除随机性因素，但是在训练到第 20 个 epoch 时，两种初始化方法所能达到的准确率基本相同，都在 0.86 左右。

2.2.3 Kaiming 初始化

在上一小节我们探索了 xavier 初始化方法，并且取得了良好的效果。但是 xavier 假设神经网络使用的激活函数关于 0 对称且在 0 处梯度为 1（如 tanh 函数），如果使用 sigmoid 或者 relu 激活函数，那么就不满足这个假

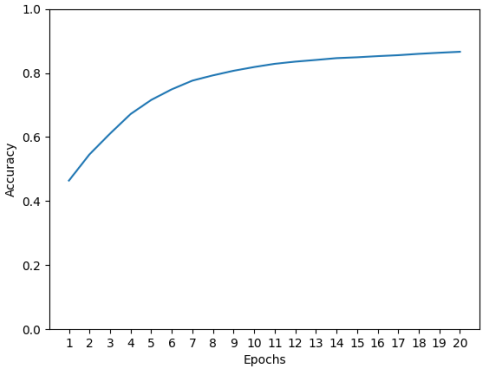

图 4: xavier_uniform 初始化

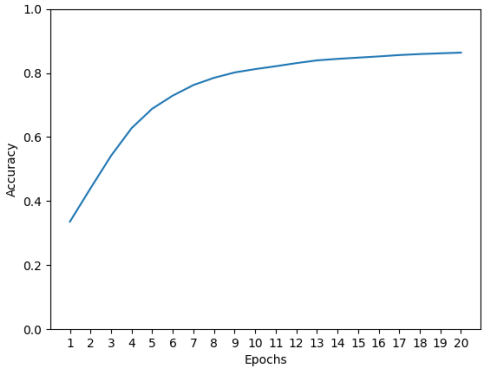

图 5: xavier_normal 初始化

设了。为了找到一种适用于 relu 激活函数的初始化方法，何凯明等人提出了 kaiming 初始化方法。在 Xavier 论文中，作者给出的 Glorot 条件是：正向传播时，激活值的方差保持不变；反向传播时，关于状态值的梯度的方差保持不变。而在 kaiming 初始化方法中则变为：正向传播时，状态值的方差保持不变；反向传播时，关于激活值的梯度的方差保持不变。

同 xavier 初始化方法类似，kaiming 初始化也有两个子方法，分别是从均匀分布中采样和从正态分布中采样，这两种方法的训练效果如图6和图7所示。

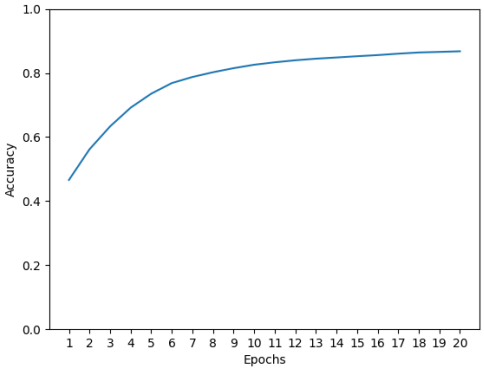

图 6: kaiming_uniform 初始化

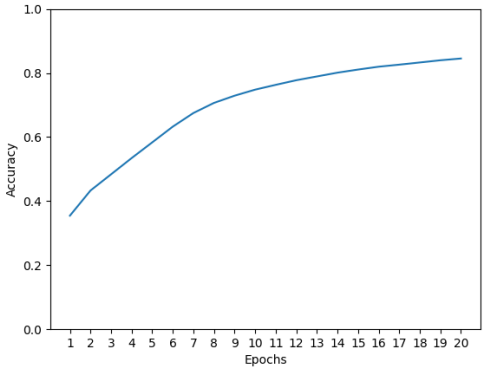

图 7: kaiming_normal 初始化

通过对比 xavier 初始化和 kaiming 初始化发现，两种方法的效果差不多，这貌似并没有体现出 kaiming 初始化方法相较 xavier 初始化方法的优

势。出现这种现象的原因很好解释，kaiming 初始化方法是为 relu 激活函数量身定制的，而现在的网络使用的是 sigmoid 激活函数，因此无法体现 kaiming 初始化的优势之处，为了使这种解释更具说服力，我将网络中的激活函数由 sigmoid 换成 relu，然后重新训练网络，得到的训练效果如图8和图9所示。

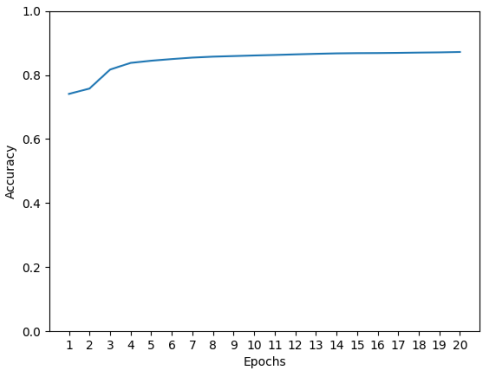

图 8: relu+kaiming_uniform 初始化

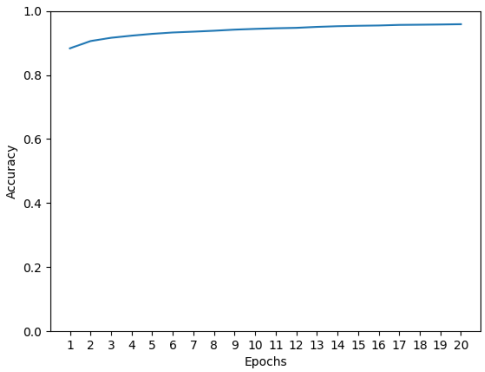

图 9: relu+kaiming_normal 初始化

图8和图9的训练效果非常震撼。当使用 relu 作为激活函数时，基于正态分布的 kaiming 初始化方法仅仅一个 epoch 就可以达到其它方法的训练效果。

2.3 损失函数

在之前的实验中，我们已经使用过了损失函数，比如公式4中就是损失函数对权重求偏导。在这一小节中，我们将介绍 Mean Squared Error(MSE) 损失函数和 Cross Entropy 损失函数，并对它们的性质和特点进行比较与讨论。

2.3.1 Mean Square Error

MSE 损失函数的定义比较简单，就是预测值减去真值的平方，即：

$$L(y, y^*) = (y - y^*)^2 \tag{15}$$

其中 y 是神经网络的预测值，y^* 是数据的真实标记。

在不考虑激活函数的情况下，y 就是神经网络最后一层的输出，即 $y = z^n$，那么此时有：

$$\begin{aligned}\delta_i^n &= \frac{\partial L}{\partial z_i} = \frac{\partial L}{\partial y_i} \\ &= \frac{\partial(\frac{1}{2}(y_i - y_i^*)^2)}{\partial y_i} \\ &= (y_i - y_i^*) * \frac{\partial(y_i - y_i^*)}{\partial y_i} \\ &= (y_i - y_i^*)\end{aligned} \tag{16}$$

更一般的表示为 $\delta^n = y - y^*$，即损失函数 L 关于网络最后一层的导数就是最后一层的输出减去预测值。

MSE 方法常用于回归问题，而本次任务要做的是分类问题。每个数据的标签是一个 0～9 之间的整数，而网络的输出是一个长度为 10 的向量，因此，如果想将 MSE 用于分类问题，需要对标记 y^* 进行 one hot 编码。具体到本次任务，编码之后的标签是一个长度为 10 的向量，这个向量只有在原始标签的位置为 1，其余的位置为 0。神经网络的输出范围是实数域，而编码之后的标签只有 0 和 1，因此我又将神经网络的输出过了一遍 sigmoid 函数，将实数域范围的值映射到 0～1。此时损失函数关于网络最后一层输出的导数为：

$$\delta^n = \frac{\partial L}{\partial z^n} = \frac{\partial L}{\partial y} * \frac{\partial y}{z^n} = (y - y^*) * \sigma(z^n) * (1 - \sigma(z^n)) \tag{17}$$

2.3.2 Cross Entropy

交叉熵损失函数用于度量两个概率分布的差异; 一般使用交叉熵损失前，会对网络输出做 softmax 变换进行概率归一化；所以我在本次任务中使用的交叉熵损失是带 softmax 变换的交叉熵。

softmax 变换的定义为：

$$a_i = e^{y_i} / \sum_k e^{y_k} \tag{18}$$

交叉熵损失函数的定义为：

$$L(y, y^*) = -\sum_i y_i^* \log a_i \tag{19}$$

下面我们推导交叉熵损失函数对神经网络的输出 y 的导数，根据链式法则，我们先推导 a_i 关于 y_i 的导数：

$$\begin{aligned}
&\frac{\partial a_i}{\partial y_j} = \frac{\partial(e^{y_i}/\sum_k e^{y_k})}{\partial y_j} \\
&= \frac{\partial e^{y_i}}{\partial y_j} * \frac{1}{\sum_k e^{y_k}} + e^{y_i} * \frac{-1}{(\sum_k e^{y_k})^2} * \frac{\partial(\sum_k e^{y_k})}{\partial y_j} \\
&= \frac{\partial e^{y_i}}{\partial y_j} * \frac{1}{\sum_k e^{y_k}} - \frac{e^{y_i}}{(\sum_k e^{y_k})^2} * e^{y_j} \\
&= \begin{cases} \frac{e^{y_j}}{\sum_k e^{y_k}} - \frac{(e^{y_j})^2}{(\sum_k e^{y_k})^2} & i = j \\ -\frac{e^{y_i} e^{y_j}}{(\sum_k e^{y_k})^2} & i \neq\ j \end{cases} \\
&= \begin{cases} a_i(1 - a_i) & i = j \\ -a_i a_j & i \neq\ j \end{cases}
\end{aligned} \tag{20}$$

然后再求 L 关于 y_i 的偏导：

$$\begin{aligned}
&\frac{\partial L}{\partial y_j} = -\sum_i \frac{\partial(y_i^* \log a_i)}{\partial a_i} * \frac{\partial a_i}{\partial y_j} \\
&= -\sum_i \frac{y_i^*}{a_i} * \frac{\partial a_i}{\partial y_j} \\
&= -\frac{y_j^*}{a_j} * a_j(1 - a_j) + \sum_{i \neq\ j} \frac{y_i^*}{a_i} * a_i a_j \\
&= -y_j^*(1 - a_j) + \sum_{i \neq\ j} y_i^* a_j \\
&= -y_j^* + \sum_i y_i^* a_j \\
&= a_j - y_j^*
\end{aligned} \tag{21}$$

更一般的表示为：

$$\delta^n = \frac{\partial L}{\partial z^n} = \frac{\partial L}{\partial y} = a - y^* \tag{22}$$

所以使用带 softmax 变换的交叉熵损失函数，损失函数 L 关于网络最后一层的导数就是预测值经 softmax 变换后的值减去真实值。

2.3.3 对比 MSE 和 CE

为了对比 MSE 损失函数和 CE 损失函数的效果，使用和表2相同的超参数对网络进行训练，训练结果如图10和图11所示。

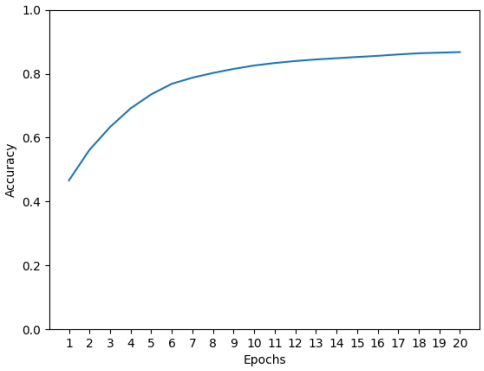

图 10: MSE Loss

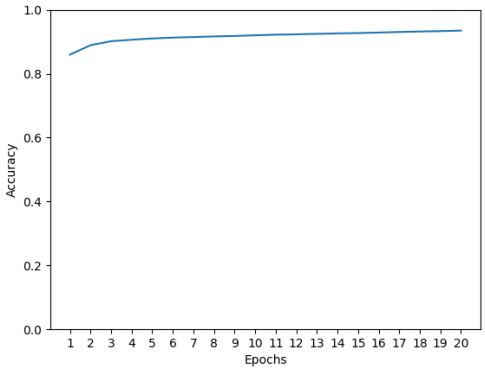

图 11: CE Loss

实验结果与我们的预期是一致的。交叉熵损失函数的训练速度远远快于均方根误差损失函数。MSE 损失函数不适合最后一层含有 Sigmoid 或 Softmax 激活函数的神经网络，平方误差损失函数相对于输出层的导数如公式17所示，激活函数为 sigmoid，如果 z^n 的绝对值较大，函数的梯度会趋于饱和，即 $\sigma'(z^n)$ 的绝对值非常小，导致 δ^n 的取值也非常小，使得基于梯度的学习速度非常缓慢。而交叉熵损失函数相对于输出层的导数如公式22所示，这个导数是线性的，因此不会存在学习速度过慢的问题。

2.4 学习率调整

学习率的调整方法很容易理解，就是随着训练的进行，以某种策略不断的减小学习率，可以理解成先粗调，后微调。为了比较不同学习率调整方法的效果，以下所有学习率调整方法，如无特别说明，都使用表3所示的超参数进行网络训练。

参数名称	参数值
网络结构	[784, 400, 100, 10]
初始学习率	0.3
epochs	20
batch_size	128

表 3: 学习率调整的超参数设置

2.4.1 指数衰减

指数衰减是一种最基本的学习率调整方法，在我的代码中，学习率会在每个 epoch 之后衰减一次。使用表3所示的超参数，设定指数的底数 $gamma = 0.99$，训练效果如图12所示。

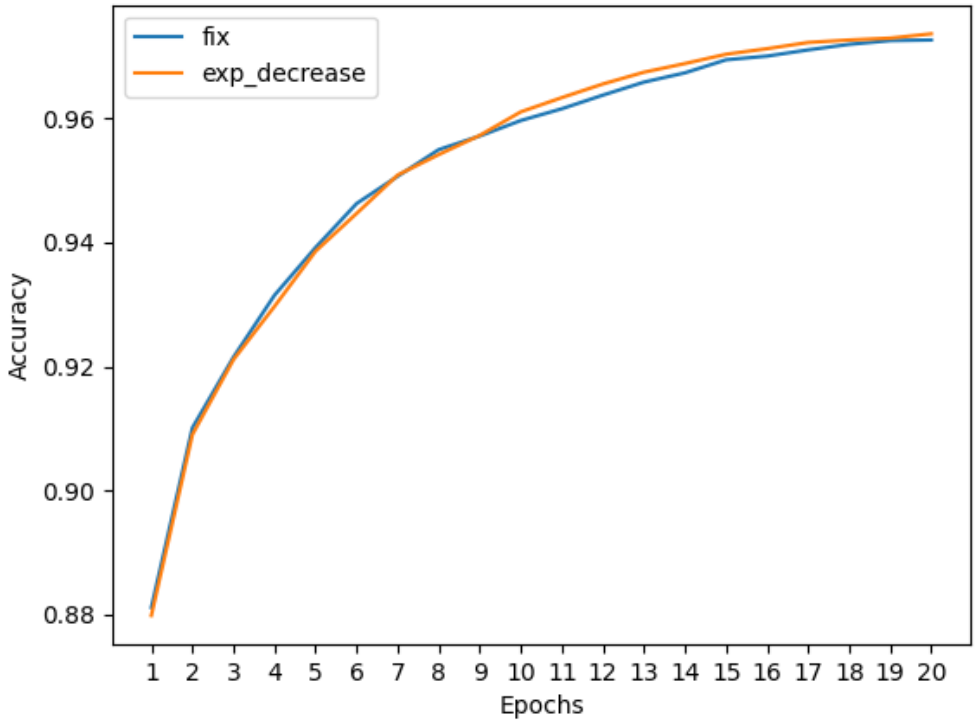

图 12: 学习率指数衰减

从图12中可以看出，采用学习率指数衰减的方法比固定学习率能更快的收敛。这是因为在模型训练的前期，学习率大一点没关系，反正此时离最优解还很远，但是在模型训练训练的后期，如果学习率仍然很大，很可能会导致模型的参数在最优解附近摆动，但就是无法到达最优解。

2.4.2 阶梯式衰减

阶梯式衰减是指每经过固定数量的 epoch，学习率就衰减一次。衰减的方式就是当前的学习率乘以一个衰减因子 $gamma$。使用表3所示的超参数，其中初始学习率改为 1，epochs 改为 100，衰减因子 $gamma = 0.95$，训练效果如图13所示。

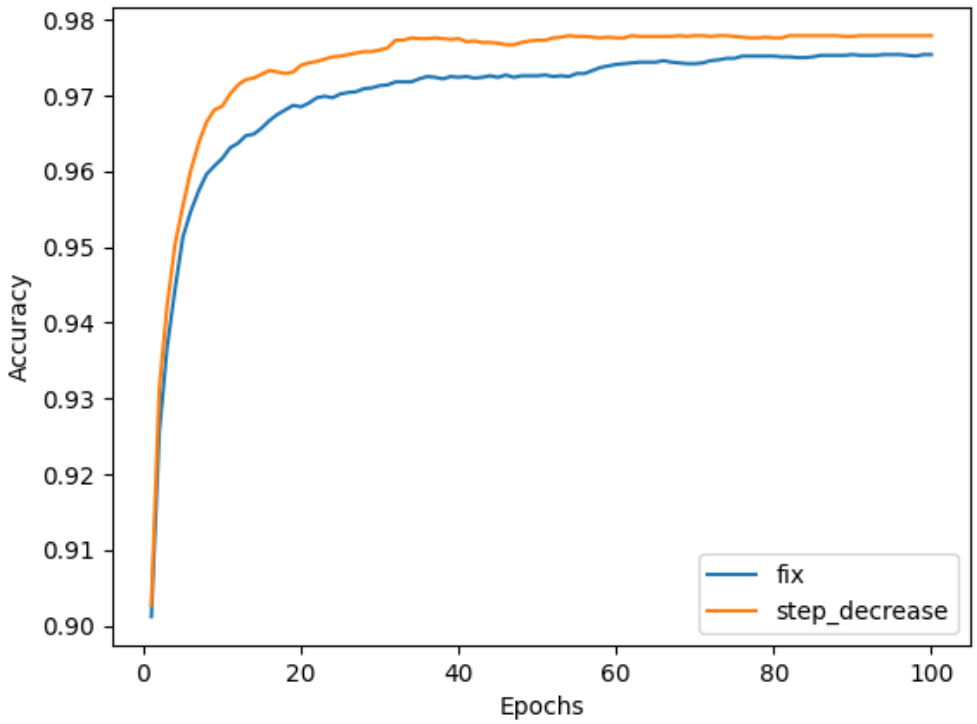

图 13: 学习率阶梯式衰减

从图13中可以看出，学习率阶梯式衰减比固定学习率可以更快的收敛，并且收敛到更好的效果。

2.5 正则化

正则化是一种为了减小测试误差的行为。我们在构造机器学习模型时，最终目的是让模型在面对新数据的时候，可以有很好的表现。当用比较复杂的模型，去拟合数据时，很容易出现过拟合现象，这会导致模型的泛化能力下降，这时候，我们就需要使用正则化，降低模型的复杂度。以下所有实验，如无特别说明，均是在如表4所示的超参数下进行的。

参数名称	参数值
网络结构	[784, 400, 100, 10]
学习率	0.1
epochs	20
batch_size	128

表 4: 神经网络超参设置

2.5.1 L1 正则化

L1 正则化的形式很简单，只需要在原损失函数的基础上加上权重的绝对值之和，即：

$$L_1 = L + \lambda \sum ||W||$$

其中 λ 是一个超参数，需要人为的指定，通过控制 λ 的大小，可以控制前面的损失项和正则化项所占的比例，此时损失函数 L_1 对权重 W 的偏导为：

$$\frac{\partial L_1}{\partial W} = \frac{\partial L}{\partial W} + \lambda sign(W)$$

其中 $sign$ 是符号函数，当 $x > 0$ 时，$sign(x) = 1$，否则为-1。

在表4所示的超参数组合下，设置 $\lambda = 0.1$，实验结果如图14所示。从图14中可以看出，加了 L1 正则化后网络反倒训不起来了，这很可能是参数 λ 设置的过大，或者网络结构过于简单，根本就不会出现过拟合现象导致的。

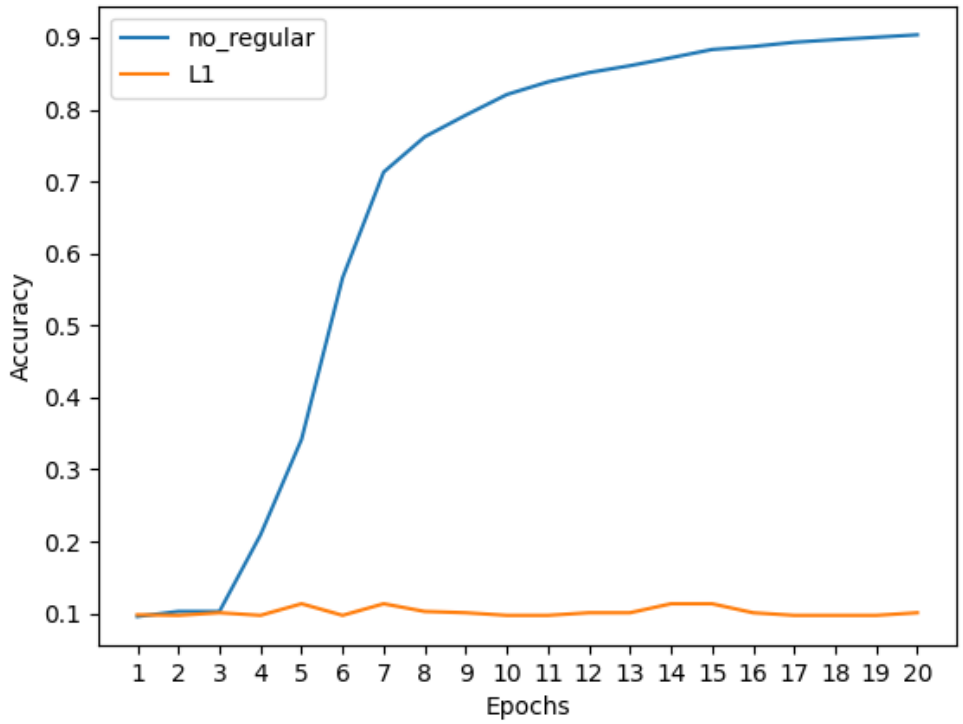

图 14: L1 正则化

2.5.2 L2 正则化

L2 正则化的形式和 L1 类似，它是在原损失函数的基础上加上权重矩阵的平方之和，即：

$$L_2 = L + \lambda \sum ||W||^2$$

此处的 λ 与 L1 正则化中的 λ 相同，损失函数 L_2 对权重 W 的偏导为：

$$\frac{\partial L_2}{\partial W} = \frac{\partial L}{\partial W} + 2\lambda W$$

在表4所示的超参数组合下，设置 $\lambda = 0.1$，实验结果如图15所示，它训不起来的原因与 L1 正则化类似。

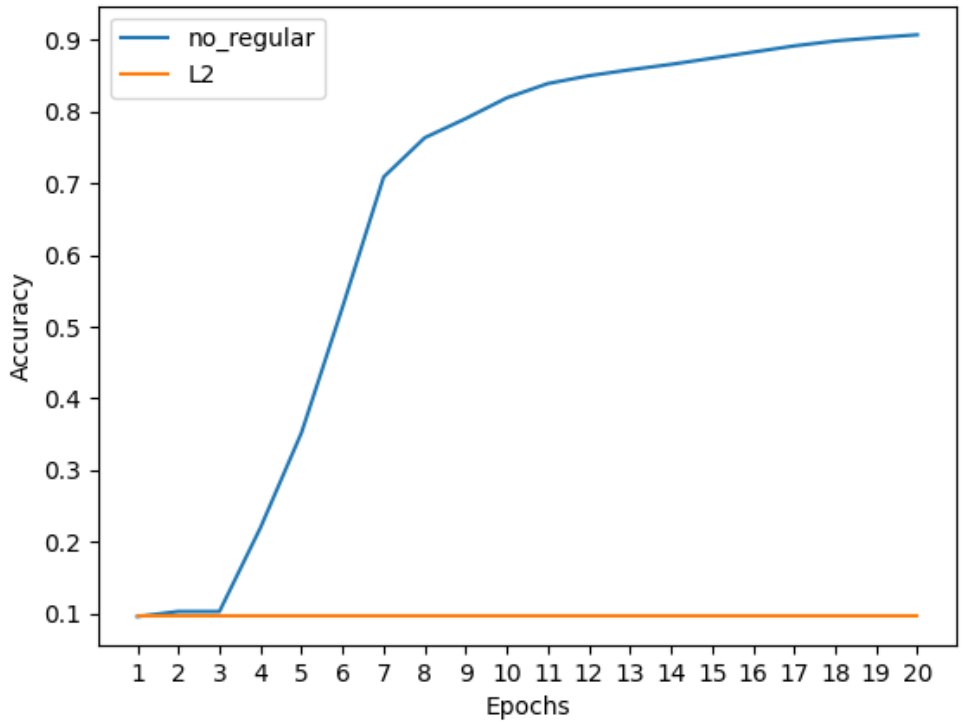

图 15: L2 正则化

2.5.3 Batch Normalization

BN 是由 Google 于 2015 年提出，这是一个深度神经网络训练的技巧，它不仅可以加快了模型的收敛速度，而且更重要的是在一定程度缓解了深层网络中“梯度弥散”的问题，从而使得训练深层网络模型更加容易和稳定。我们知道网络一旦 train 起来，那么参数就要发生更新，除了输入层的数据外 (因为输入层数据，我们已经人为的为每个样本归一化)，后面网络每一层的输入数据分布是一直在发生变化的，因为在训练的时候，前面层训练参数的更新将导致后面层输入数据分布的变化。BN 算法的提出正是为了解决这个问题。

BN 的基本思想其实相当直观：因为深层神经网络在做非线性变换前的激活输入值随着网络深度加深或者在训练过程中，其分布逐渐发生偏移或者变动，之所以训练收敛慢，一般是整体分布逐渐往非线性函数的取值区

间的上下限两端靠近（对于 Sigmoid 函数来说，意味着激活输入值 WU+B 是大的负值或正值），所以这导致反向传播时低层神经网络的梯度消失，这是训练深层神经网络收敛越来越慢的本质原因，而 BN 就是通过一定的规范化手段，把每层神经网络任意神经元这个输入值的分布强行拉回到均值为 0 方差为 1 的标准正态分布，其实就是把越来越偏的分布强制拉回比较标准的分布，这样使得激活输入值落在非线性函数对输入比较敏感的区域，这样输入的小变化就会导致损失函数较大的变化，意思是这样让梯度变大，避免梯度消失问题产生，而且梯度变大意味着学习收敛速度快，能大大加快训练速度。

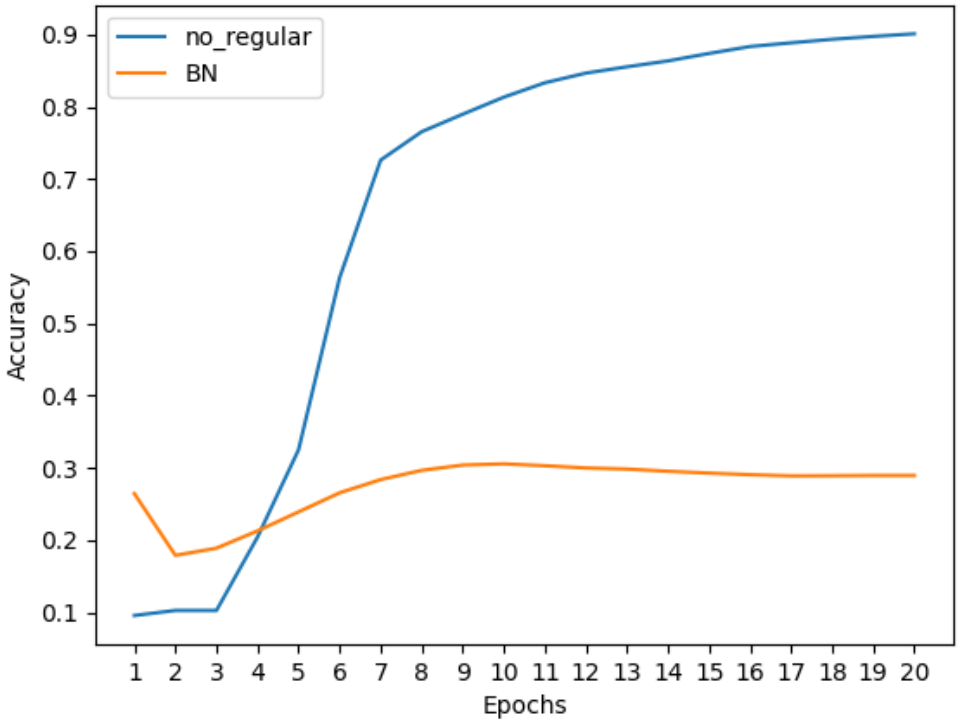

图 16: BN 正则化

上面说过经过正态分布变换后某个神经元的激活 x 形成了均值为 0，方差为 1 的正态分布，目的是把值往后续要进行的非线性变换的线性区拉动，增大导数值，增强反向传播信息流动性，加快训练收敛速度。但是这样会

导致网络表达能力下降，为了防止这一点，每个神经元增加两个调节参数（scale 和 shift），这两个参数是通过训练来学习到的，用来对变换后的激活反变换，使得网络表达能力增强，即对变换后的激活进行如下的 scale 和 shift 操作，这其实是变换的反操作：

$$\hat{z} = \gamma \hat{x} + \beta$$

其中 γ 和 β 也是需要在训练过程中学习的参数。

在我的代码实现中，我采用了滑动平均的方法计算当前批次样本的均值和方差，滑动参数设置为 0.02，即有 0.02 的权重考虑当前的均值和方差，有 0.98 的权重考虑历史的均值和方差。在表4的超参数设置下，训练效果如图16所示。

2.5.4 Dropout

Dropout 是一种能防止模型过拟合的方法，且可以减少训练的时间。它的基本思想是，在前向传播的时候，让某个神经元的激活值以一定的概率 p 停止工作，同样的道理，在反向传播的时候，让那些不再工作的神经元的梯度值为 0，这样可以使模型泛化性更强，因为它不会太依赖某些局部的特征。

使用表4所示的超参数，设置丢弃率为 0.5，训练过程如图17所示。

3 总结

3.1 工作量总结

在本次实验中，我用 numpy 实现了一个多层的神经网络，该网络可以由用户自由的指定层数，以及每层的神经元个数。为了进一步验证我实现的前向过程和后向过程的正确性，我在相同的超参数设置下（包括参数初始值），和 pytorch 实现的网络进行了对比，在测试集的准确率上，两者完全相同。

在完成了基本的网络之后，我又对比了不同的参数初始化方法对训练过程的影响，通过实验验证了一个好的初始化方法可以加快收敛速度，并

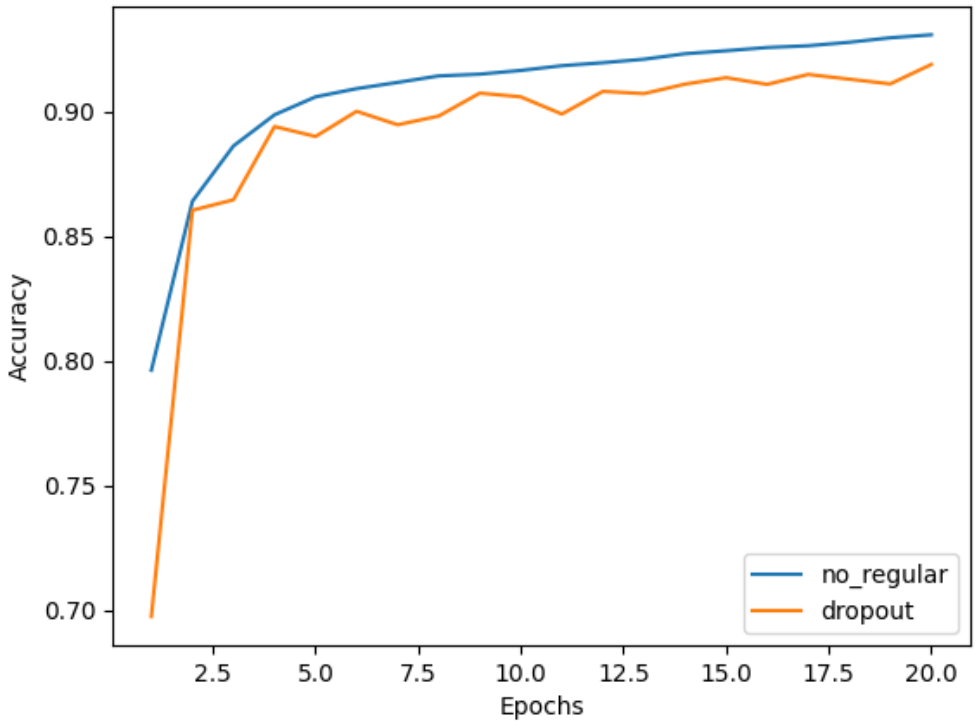

图 17: Dropout 正则化

且取得更好的泛化性能。具体的来说，我探索了 0 值初始化，两种 xavier 初始化和两种 kaiming 初始化。其中 0 值初始化方式不可取，xavier 初始化在 sigmoid 激活函数上可以取得较好的效果，而 kaiming 初始化在 relu 激活函数上可以取得较好的效果。这些初始化方法关于理论上的分析可以参见 2.2 节，此处不再赘述。

一个好的损失函数对于模型的训练来说也是至关重要的，本文实现并对比了均方差损失函数和交叉熵损失函数，并对这两种损失函数所表现出来的效果上的差异做了简要分析。动态调整学习率是神经网络训练过程中一个常用的技巧，本文实现了指数衰减和阶梯式衰减两种学习率调整方法，如果能对初始学习率和学习率衰减系数做更细致的调整，那么固定学习率和动态学习率的对比将会更明显。

正则化是防止模型过拟合，提高泛化能力的一种有效手段，在本文中，

实现了 L1 正则化，L2 正则化，batch normalization 和 dropout，本文将这四种正则化方法和不使用正则化做了对比，发现加了正则化之后并没有给模型泛化性能带来多大的提升，甚至还有可能降低泛化性能，一个可能的原因是 mnist 数据集比较简单，不容易发生过拟合的现象。关于正则化更详细的分析，可以参见 2.5 小节。

3.2 最优准确率

在经过网格搜索之后，使用表5所示的超参数可以在测试集上取得0.9858的准确率。

参数名称	参数值
网络结构	[784, 1024, 64, 10]
学习率	0.3
epochs	100
batch_size	64
gamma	0.5
step_size	30

表 5: 神经网络超参设置

选用的方法为：xavier 初始化 + 交叉熵损失函数 + 阶梯式学习率衰减 + 不正则化 +leaky relu 激活函数。

训练过程如图18所示。

3.3 代码运行方式

进入代码文件的主目录下，在 shell 中输入 python main.py 即可复现最优准确率的效果。

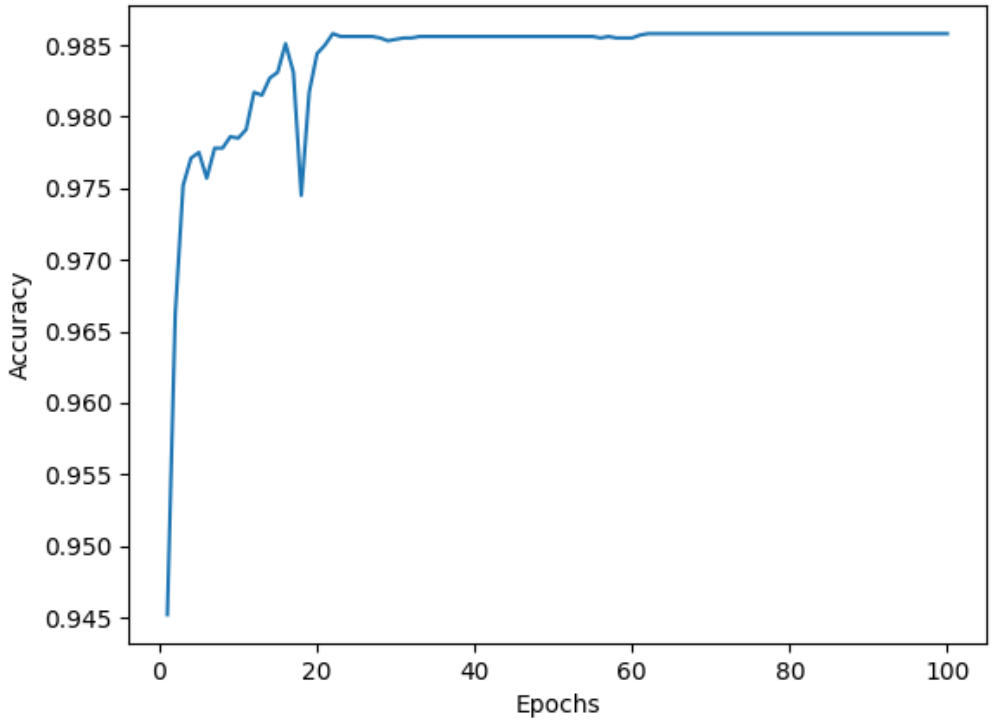
0.985
0.980
0.975
0.970
0.965
0.960
0.955
0.950
0.945
Accuracy
0
20
40
60
80
100
Epochs

图 18: 最优准确率