基于BP算法的手写数字识别

一. 实验概述

本实验使用BP神经网络，在MNIST数据集上进行训练与识别。我们实现了一个端到端的两层神经网络结构，输入是28*28的图片，输出是模型预测的标签，即0-9这10个数字之一。

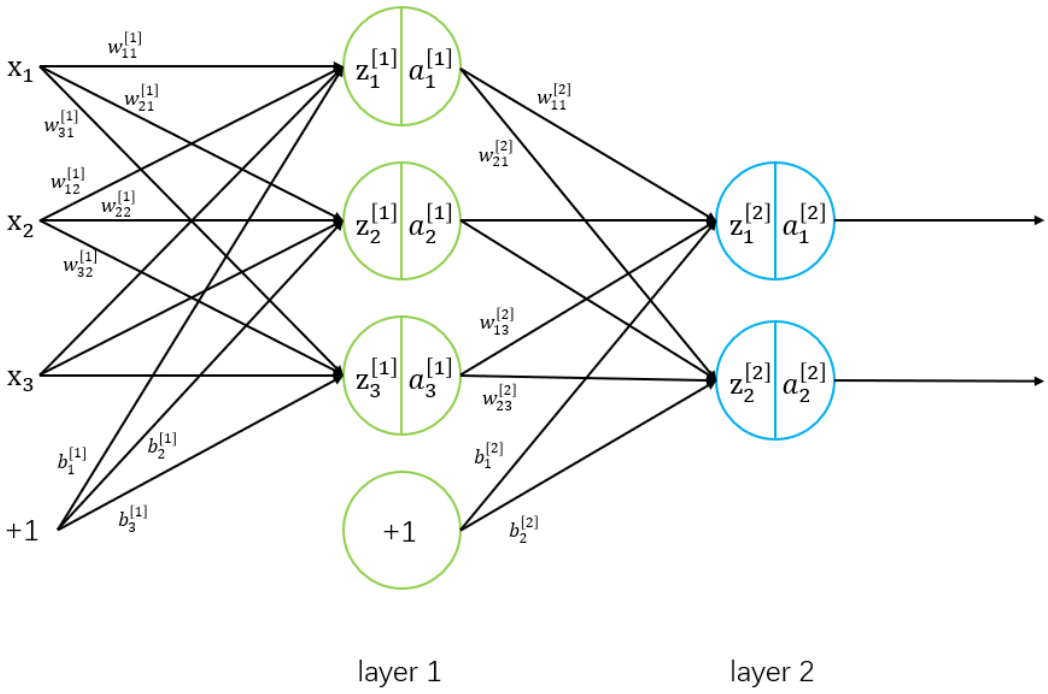
神经网络的架构、训练、测试过程完全由`numpy`实现。

首先我们将标注进行one-hot编码，将数据经过ReLU激活函数以及最后softmax进行训练。训练过程中，为了优化神经网络的表现，我们使用了He initialization等方法提升性能；为了防止过拟合，我们使用了Early Stopping机制。最后我们将全连接神经网络与CNN网络进行了对比，分析了CNN表现更佳的原因。

二. 算法设计

2.1 模型架构

模型是两层神经网络



核心数据结构如下：

- **Layers**：`tuple`类型，分别是输入层到输出层的神经元数量，其中输入层是第0层。
- **Weights**：`numpy.ndarrays`类型，`weights[l]`表示输入到第l层的权重，即$z^{[l]} = W^{[l]}x + b^{[l]}$。
- **Biases**：`numpy.ndarrays`类型，`biases[l]`表示输入到第l层的偏差。
- **Z**：`numpy.ndarrays`类型，`_zs[l]`表示第l层神经元的输入。
- **Activations**：`numpy.ndarrays`类型，`_activations[l]`表示第l层神经元的输出，即$a^{[l]} = ReLU(z^{[l]})$。

2.2 前向传播

前向传播的过程如下：

$$
\begin{aligned}
z^{[1]} &= W^{[1]}x + b^{[1]} \\
a^{[1]} &= ReLU(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= ReLU(z^{[2]})
\end{aligned}
$$

代码实现如下：

```
def _forward_prop(self, x):
    self._activations[0] = x
    for i in range(1, self.num_layers):
        self._zs[i] = self.weights[i].dot(self._activations[i - 1]) +
self.biases[i]
        # 最后一层使用softmax
        if i == self.num_layers - 1:
            self._activations[i] = activations.softmax(self._zs[i])
        else:
            self._activations[i] = self.activation_fn(self._zs[i])
```

2.3 反向传播

神经网络的训练使用**小批量梯度下降**（MBGP）。

MBGD是介于BGD和SGD之间的算法，每计算常数b次训练实例，便更新一次参数 。通常我们会令b在2-100之间。这样做的好处在于，我们可以用向量化的方式来循环b个训练实例，能使用科学计算库实现**平行处理**；并且适当Batch Size使得**梯度下降方向更加准确。**

$$
\theta_j := \theta j - \alpha \frac{1}{b} \sum_{k=i}^{i+b-1} \frac{\partial}{\partial \theta_j} cost\left(\theta, (x^{(i)}, y^{(i)})\right)
$$

反向传播代码实现如下：

```
def _back_prop(self, x, y):
    nabla_b = [np.zeros(bias.shape) for bias in self.biases]
    nabla_w = [np.zeros(weight.shape) for weight in self.weights]

    error = (self._activations[-1] - y)
    nabla_b[-1] = error
    nabla_w[-1] = error.dot(self._activations[-2].transpose())

    for i in range(self.num_layers - 2, 0, -1):
        error = np.multiply(
            self.weights[i + 1].transpose().dot(error),
            self.activation_fn_prime(self._zs[i])
        )
        nabla_b[i] = error
        nabla_w[i] = error.dot(self._activations[i - 1].transpose())

    return nabla_b, nabla_w
```

训练过程伪代码如下：

$$
\begin{aligned}
&\text{Ramdomly shuffle training examples;}\\
&\text{Repeat } \{\\
&\quad \text{for i:=1,...,m } \{\\
&\quad\quad \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} cost\left(\theta, (x^{(i)}, y^{(i)})\right)\\
&\quad\quad (\text{for every batch})\\
&\quad \}\\
&\}
\end{aligned}
$$

模型训练代码实现如下：

```
for epoch in range(epochs):
    random.shuffle(training_data)
    mini_batches = [training_data[k:k + self.mini_batch_size] for k in
                    range(0, len(training_data), self.mini_batch_size)]

    for mini_batch in mini_batches:
        nabla_b = [np.zeros(bias.shape) for bias in self.biases]
        nabla_w = [np.zeros(weight.shape) for weight in self.weights]
        for x, y in mini_batch:
            # 前向传播
            self._forward_prop(x)

            # 反向传播
            delta_nabla_b, delta_nabla_w = self._back_prop(x, y)
            nabla_b = [nb + dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw + dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]

        self.weights = [w - (self.lr / self.mini_batch_size) * dw
                        for w, dw in zip(self.weights, nabla_w)]
        self.biases = [b - (self.lr / self.mini_batch_size) * db
                       for b, db in zip(self.biases, nabla_b)]

    if validation_data:
        accuracy = self.validate(validation_data) / 100.0
        print(f"Epoch {epoch + 1}/{epochs}, accuracy {accuracy} %.")
    else:
        print(f"Processed epoch {epoch}.")
```

三. 遇到的问题及解决方法

3.1 标注编码

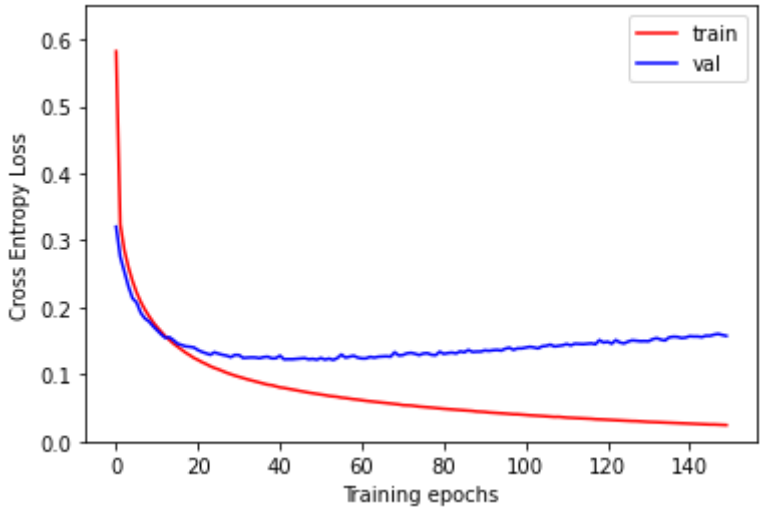
首先是在训练过程中，我们选用交叉熵函数作为损失函数。但是刚开始我们训练时发现反向传播过程中出现了异常。

$$L(y,t) = -\sum_{i=1}^{n} t_i \, log \, yi$$

后来经过研究发现在反向传播时会涉及预测值和真实值相减的一个过程，若只是把使用int类型作为标注，那么不同数字之间的顺序信息就会反映到损失函数中，这当然是我们不需要的，所以我们使用了**one-hot**编码来避免这样的情况。

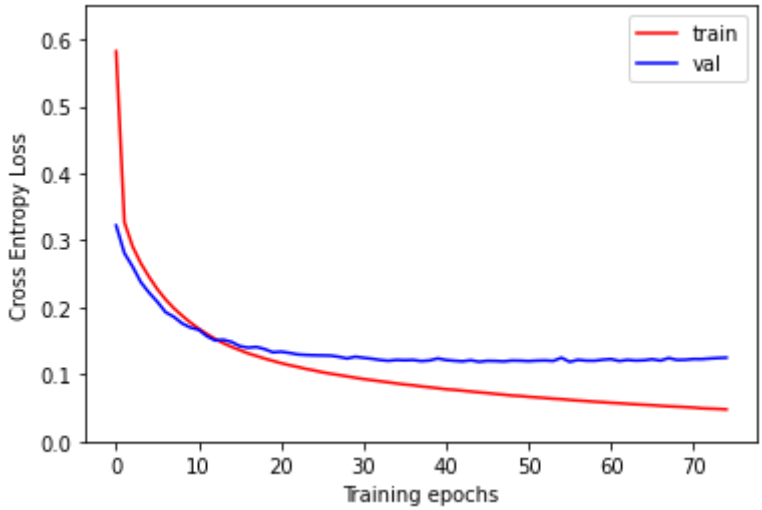
3.2 防止过拟合

训练集与测试集的交叉熵损失函数随着训练的epoch增加曲线如下：



随着训练的进行，模型在训练集上的误差逐渐降低，但是在验证集上的误差却有走高的趋势，说明模型发生了过拟合。

我们使用了**early stopping**机制来解决这一问题。其原理是在某个节点之前，更好地拟合训练集使得模型在训练集之外的数据上表现得更好；但在该节点之后，更好地拟合训练集会增大泛化误差。提前停止相关规则给出停止迭代的条件（这里我们允许模型的表现最多变差2次），以便在模型开始过拟合之前停止迭代优化。使用earlystop后，训练停止与第74个epoch。



四. 创新方法

4.1 初始化权重

最初，我们初始化权重时将其初始化为标准正态分布，但是训练效果欠佳，其训练之初指标如下：

```
Epoch 1/100, accuracy 68.27 %.
Epoch 2/100, accuracy 74.66 %.
Epoch 3/100, accuracy 78.61 %.
Epoch 4/100, accuracy 81.5 %.
Epoch 5/100, accuracy 83.28 %.
```

训练结束也仅有93.6的指标，经过分析后我们判断是层输出过大，产生了梯度爆炸。

查阅了很多资料后，我们使用**He initialization**，其基本思想是保持输入和输出的方差一致，这是一个针对ReLU的初始化方法，最终达到了不错的效果。

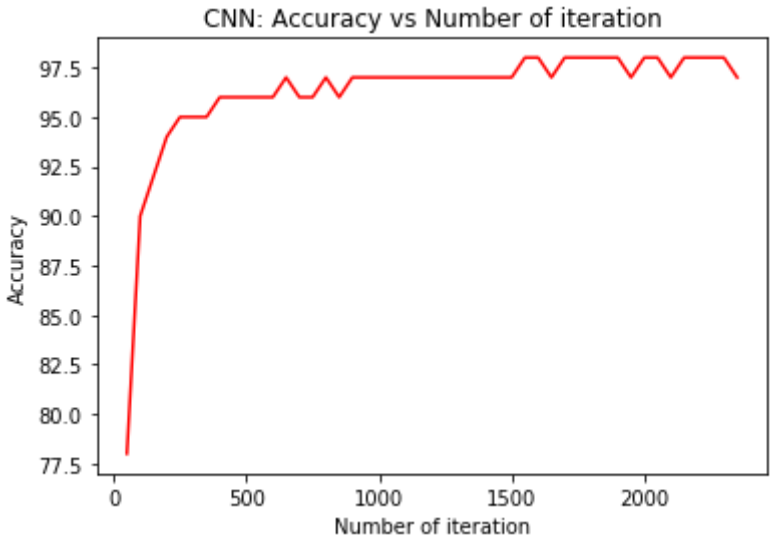
```
Epoch 1/100, accuracy 91.02 %.
Epoch 2/100, accuracy 92.0 %.
Epoch 3/100, accuracy 92.63 %.
Epoch 4/100, accuracy 92.93 %.
Epoch 5/100, accuracy 93.75 %.
```

4.2 与CNN比较

在Kaggle竞赛页面，我们找到了大量优秀的模型，它们大多使用卷积神经网络实现。卷积可以有效提高模型的表现。

使用卷积的理由主要有两个：首先是允许**参数共享**，一个卷积核可以在图像的任意位置检测到某个特定的特征，这使得参数的数量打大减少。其次是**稀疏连接**，输出图像的每个值只与少数被扫描的输入值有关，其他值对这个输出毫无影响。这两种机制使得网络参数减少，以便我们用更小的训练集来训练它，从而预防过度拟合。

我们也使用pytorch框架实现了一个卷积神经网络，其最终正确率达到了99%。



五. 结果分析

借助`sklearn`的`confusion_matrix`模块完成分类算法的基本评价指标，代码如下：

```
def validate(self, validation_data):
    """
    测试模型在给定数据集上的表现
    """
    y_true = [y for _, y in validation_data]
    y_pred = [self.predict(x) for x, _ in validation_data]
    conf_matrix = confusion_matrix(y_true, y_pred)

    # (TP+TN)/(TP+TN+FP+FN)
    accuracy = np.diag(conf_matrix).sum() / conf_matrix.sum()
```

```
    # TP/(TP+FP)
    precision = np.diag(conf_matrix) / conf_matrix.sum(axis=0)
    precision = np.nanmean(precision)

    # TP/(TP+FN)
    recall = np.diag(conf_matrix) / conf_matrix.sum(axis=1)
    recall = np.nanmean(recall)

    # 2PR/(P+R)
    F1 = 2 * precision * recall / (precision + recall)

    return accuracy, precision, recall, F1
```

最终模型在测试集上的指标如下：

```
TEST DATA accuracy:0.9645, precision:0.9645, recall:0.9641, F1:0.9643
```

混淆矩阵如下，总体来讲模型在每个数字上的表现均衡，但是在数字3上的表现稍差一点。

```
[[ 964    0    5    2    1    2    3    2    1    0]
 [   0 1122    1    2    0    1    3    1    5    0]
 [   5    3  985   16    5    1    8    3    6    0]
 [   2    2    6  982    1    5    1    5    6    0]
 [   1    1    3    2  949    2    4    3    1   16]
 [   5    1    0   22    2  841    7    2    8    4]
 [   9    2    2    1    5    6  932    0    1    0]
 [   0    8   14    6    4    0    0  984    1   11]
 [   7    1    5   13    3    8    2    0  931    4]
 [   3    4    1    9   18    4    1    9    5  955]]
```