实验三实验报告

1.实验介绍：

本实验使用卷及神经网络实现MNIST的手写数字识别功能，主要工作是完成卷积神经网络代码实现。主要部分有：conv_layer卷基层，pooling_layer池化层，reshape_layer层.

2.代码分析：

SGD优化器代码如下：采用了随机梯度下降

```
import numpy as np

class SGD():
    def __init__(self, learningRate, weightDecay):
        self.learningRate = learningRate
        self.weightDecay = weightDecay

    # One backpropagation step, update weights layer by layer
    def step(self, model):
        layers = model.layerList
        for layer in layers:
            if layer.trainable:
                layer.diff_W = - self.learningRate * layer.grad_W - 
self.weightDecay*self.learningRate * layer.W
                layer.diff_b = - self.learningRate * layer.grad_b
                # Weight update
                layer.W += layer.diff_W
                layer.b += layer.diff_b
```

卷基层conv_layer主要代码如下（前馈和反馈）：

```

    def forward(self, Input):
        '''
        forward method: perform convolution operation on the input.
        Agrs:
            Input: A batch of images, shape=(batch_size, channels, height, width)
        '''
        out = None
        self.Input = Input
        N, C, H, W = Input.shape
        F, _, HH, WW = self.W.shape
        P = self.pad
        Ho = 1 + (H + 2 * P - HH)
        Wo = 1 + (W + 2 * P - WW)
        x_pad = np.zeros((N, C, H + 2 * P, W + 2 * P))
        x_pad[:, :, P:P + H, P:P + W] = self.Input
```

```
        out = np.zeros((N, F, Ho, Wo))
        for f in range(F):
            for i in range(Ho):
                for j in range(Wo):
                    out[:, f, i, j] = np.sum(x_pad[:, :, i : i  + HH, j : j  + WW] * self.W[f, :, :, :],axis=(1, 2, 3))
            out[:, f, :, :] += self.b[f]
        return out


    def backward(self, delta):

        N, F, H1, W1 = delta.shape
        x = self.Input
        w = self.W
        b = self.b

        N, C, H, W = x.shape
        HH = w.shape[2]
        WW = w.shape[3]
        P = self.pad

        dx = np.zeros_like(x)
        dw = np.zeros_like(w)
        db = np.zeros_like(b)

        x_pad = np.pad(x, [(0, 0), (0, 0), (P, P), (P, P)], 'constant')
        dx_pad = np.pad(dx, [(0, 0), (0, 0), (P, P), (P, P)], 'constant')
        db = np.sum(delta, axis=(0, 2, 3))

        for n in range(N):
            for i in range(H1):
                for j in range(W1):
                    # Window we want to apply the respective f th filter over (C, HH, WW)
                    x_window = x_pad[n, :, i : i  + HH, j : j  + WW]
                    for f in range(F):
                        dw[f] += x_window * delta[n, f, i, j]  # F,C,HH,WW
                        # C,HH,WW
                        dx_pad[n, :, i : i  + HH, j : j  + WW] += w[f] * delta[n, f, i, j]

        dx = dx_pad[:, :, P:P + H, P:P + W]
        self.grad_W = dw
        self.grad_b = db
        return dx
```

池化层Polling Layer主要代码：

```
    def forward(self, Input):
        '''
        This method performs max pooling operation on the input.
        Args:
            Input: The input need to be pooled.
        Return:
            The tensor after being pooled.
        '''
        self.Input = Input
        input_after_pad = np.pad(Input, ((0,), (0,), (self.pad,), (self.pad,)),
                                 mode='constant', constant_values=0)
        out_len=(Input.shape[3]+2*self.pad)//self.kernel_size
        out_wid=(Input.shape[3]+2*self.pad)//self.kernel_size
        output=np.zeros((Input.shape[0],Input.shape[1],out_len,out_wid))
        self.flag=np.zeros(input_after_pad.shape)

        for i in range(0,Input.shape[0]):
            for j in range(0,Input.shape[1]):
                pic=input_after_pad[i][j]
                row=0
                col=0
                for m in range(out_len):
                    col=0
                    for n in range(out_wid):
                        output[i][j][m]
[n]=np.max(pic[row:row+self.kernel_size,col:col+self.kernel_size])

x=np.argmax(pic[row:row+self.kernel_size,col:col+self.kernel_size])//self.kernel_size

y=np.argmax(pic[row:row+self.kernel_size,col:col+self.kernel_size])% self.kernel_size
                        x+=self.kernel_size*m
                        y+=self.kernel_size*n
                        self.flag[i][j][x][y]=1
                        col+=self.kernel_size
                    row+=self.kernel_size
        return output




    def backward(self, delta):
        '''
        Args:
            delta: Local sensitivity, shape-(batch_size, filters, output_height,
output_width)
        Return:
            delta of previous layer
        '''

        kernel_size=self.kernel_size

output=np.zeros((delta.shape[0],delta.shape[1],delta.shape[2]*self.kernel_size,delta.sh
ape[3]*self.kernel_size))
```

```
        #output=np.zeros(self.flag.shape)
        for i in range(0,output.shape[0]):
            for j in range(0,output.shape[1]):
                pic=output[i][j]
                for m in range(pic.shape[0]):
                    for n in range(pic.shape[1]):
                        output[i][j][m][n]=delta[i][j][m//self.kernel_size]
[n//self.kernel_size]
        output=output*self.flag
        return output
```

Reshape层主要代码：

```
    def forward(self, Input):
        return Input.reshape(self.output_shape)


    def backward(self, delta):
        return delta.reshape(self.input_shape)
```

3.结果分析：

本实验中超参数的选择如下：

batch_size = 100 总epoch次数为max_epoch = 10 init_std = 0.01 SGD学习率为 learning_rate_SGD = 0.001 weight_decay = 0.05 disp_freq = 10

```
训练时间约两小时

Epoch [0][10]    Batch [0][550]      Training Loss 10.0045   Accuracy 0.0900
Epoch [0][10]    Batch [10][550]     Training Loss 4.1416    Accuracy 0.1445
Epoch [0][10]    Batch [20][550]     Training Loss 3.1800    Accuracy 0.2029
Epoch [0][10]    Batch [30][550]     Training Loss 2.7522    Accuracy 0.2490
Epoch [0][10]    Batch [40][550]     Training Loss 2.4760    Accuracy 0.2961
Epoch [0][10]    Batch [50][550]     Training Loss 2.2812    Accuracy 0.3304
Epoch [0][10]    Batch [60][550]     Training Loss 2.1101    Accuracy 0.3711
Epoch [0][10]    Batch [70][550]     Training Loss 1.9916    Accuracy 0.4014
Epoch [0][10]    Batch [80][550]     Training Loss 1.8791    Accuracy 0.4326
Epoch [0][10]    Batch [90][550]     Training Loss 1.7768    Accuracy 0.4629
Epoch [0][10]    Batch [100][550]    Training Loss 1.6919    Accuracy 0.4878
Epoch [0][10]    Batch [110][550]    Training Loss 1.6183    Accuracy 0.5095
Epoch [0][10]    Batch [120][550]    Training Loss 1.5526    Accuracy 0.5291
Epoch [0][10]    Batch [130][550]    Training Loss 1.4922    Accuracy 0.5476
Epoch [0][10]    Batch [140][550]    Training Loss 1.4419    Accuracy 0.5623
Epoch [0][10]    Batch [150][550]    Training Loss 1.3947    Accuracy 0.5763
Epoch [0][10]    Batch [160][550]    Training Loss 1.3516    Accuracy 0.5889
```

```
Epoch [0][10]    Batch [170][550]    Training Loss 1.3112    Accuracy 0.6008
Epoch [0][10]    Batch [180][550]    Training Loss 1.2725    Accuracy 0.6122
Epoch [0][10]    Batch [190][550]    Training Loss 1.2405    Accuracy 0.6220
Epoch [0][10]    Batch [200][550]    Training Loss 1.2087    Accuracy 0.6315
Epoch [0][10]    Batch [210][550]    Training Loss 1.1775    Accuracy 0.6413
Epoch [0][10]    Batch [220][550]    Training Loss 1.1494    Accuracy 0.6499
Epoch [0][10]    Batch [230][550]    Training Loss 1.1240    Accuracy 0.6577
Epoch [0][10]    Batch [240][550]    Training Loss 1.0971    Accuracy 0.6658
Epoch [0][10]    Batch [250][550]    Training Loss 1.0742    Accuracy 0.6728
Epoch [0][10]    Batch [260][550]    Training Loss 1.0552    Accuracy 0.6784
Epoch [0][10]    Batch [270][550]    Training Loss 1.0361    Accuracy 0.6841
Epoch [0][10]    Batch [280][550]    Training Loss 1.0169    Accuracy 0.6893
Epoch [0][10]    Batch [290][550]    Training Loss 0.9989    Accuracy 0.6948
Epoch [0][10]    Batch [300][550]    Training Loss 0.9812    Accuracy 0.7003
Epoch [0][10]    Batch [310][550]    Training Loss 0.9636    Accuracy 0.7055
Epoch [0][10]    Batch [320][550]    Training Loss 0.9498    Accuracy 0.7098
Epoch [0][10]    Batch [330][550]    Training Loss 0.9334    Accuracy 0.7149
Epoch [0][10]    Batch [340][550]    Training Loss 0.9195    Accuracy 0.7193
```

由于机器性能和模型运算量的问题，本次试验想要完全跑完时间成本过高，因此只跑了两个小时左右，可以从结果看见此时网络还未收敛，loss仍在下降中，从训练数据上来看，训练准确率上升较快，经过约半个epoch的训练此时准确率已经达到了0.7左右 。由于卷积层的存在，与普通的MLP相比卷及神经网络在没有GPU加速时训练速度较普通神经网络慢很多，由于没有达到收敛，因此没有办法从目前的网络上得出准确率的比较，但是通过之前使用keras搭建的两层的卷及神经网络来看，CNN在MNIST数据集上可以达到99%以上的准确率，而由实验二，两层的采用ReLu激活函数交叉熵评价函数的两层神经网络能达到95%的准确率，可以看到卷及神经网络的效果更好。由于dropout层和局部权值的特性，CNN也未出现过拟合的情况。

从这次实验也可以看到，卷积神经网络对与计算机性能的要求较高，而在大规模使用GPU进行运算加速之前，这种模型所需要的时间成本和硬件成本都是不可接受的，因此即使在多年前已经被提出，也由于一系列原因并没有得到大规模的应用，直到近些年由于硬件的提升和随机梯度下降方法的使用而大放异彩。因此，在学科发展的历史上，我们可以看到，硬件与软件理论算法的发展都很重要，没有合适的硬件的支撑，很多理论也得不到很好的发展与应用。