Exercise3-LeNet5

> 1911533
>
> 朱昱函

实验要求

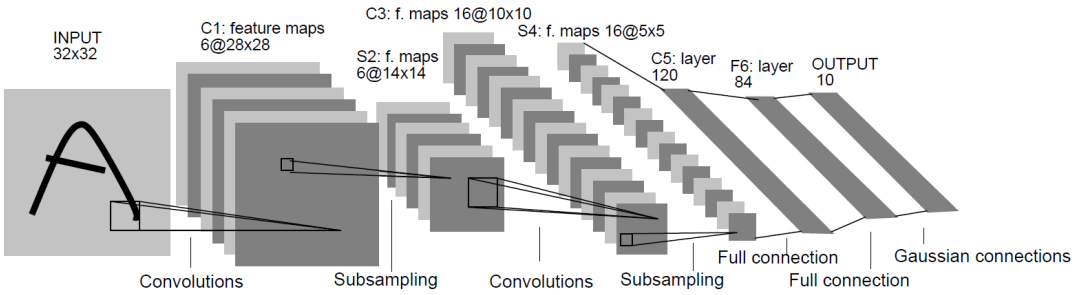
- 在这个练习中，需要用Python实现LeNet5来完成对MNIST数据集中 0-9 10个手写数字的分类。
- 代码只能使用python实现，不能使用PyTorch或TensorFlow框架

实验环境

- Anaconda+Jupyter+numpy
- Windows 10 操作系统

Lenet5网络

LeNet-5模型是Yann LeCun教授1998年在论文**Gradient-Based Learning Applied to Document Recognition**中提出的，它是第一个成功应用于数字识别问题的卷积神经网络。在MNIST数据集上，LeNet-5模型可以达到大约99.2%的正确率。LeNet-5模型总共有7层，下图展示了LeNet-5模型的架构

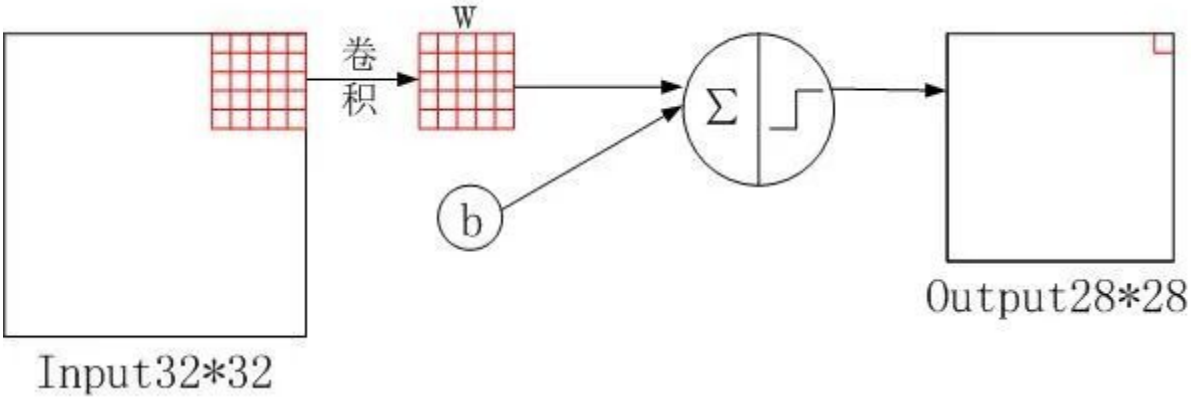


输入层

输入层是 32x32 像素的图像

C1层

C1 层是卷积层，使用 6 个 5×5 大小的卷积核，padding=0，stride=1进行卷积，得到 6 个 28×28 大小的特征图：32-5+1=28

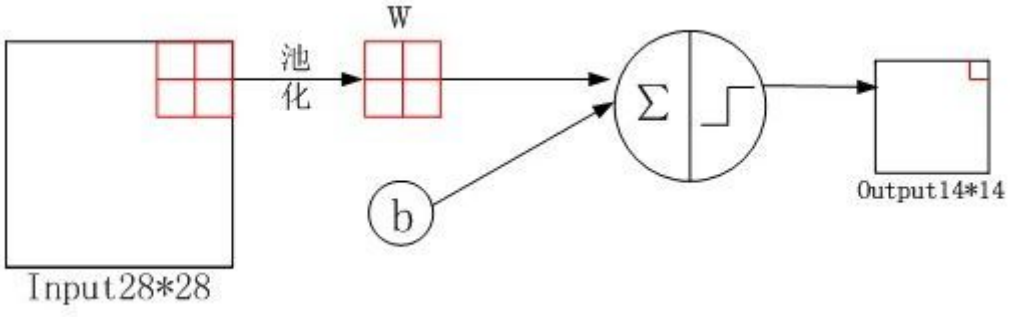


参数个数：(5*5+1)*6=156，其中5*5为卷积核的25个参数w，1为偏置项b。

连接数：156*28*28=122304，其中156为单次卷积过程连线数，28*28为输出特征层，每一个像素都由前面卷积得到，即总共经历28*28次卷积

S2层

S2 层是降采样层，使用 6 个 2×2 大小的卷积核进行池化，padding=0，stride=2，得到 6 个 14×14 大小的特征图：28/2=14



S2 层其实相当于降采样层+激活层。先是降采样，然后激活函数 sigmoid 非线性输出。先对 C1 层 2x2 的视野求和，然后进入激活函数，即：

$$sigmoid(w \cdot \sum_{i=1}^{4} x_i + b)$$

参数个数：(1+1)*6=12，*其中第一个 1 为池化对应的 2*2 感受野中最大的那个数的权重 w，第二个 1 为偏置 b。

连接数：(*2*2+1)*6*14*14= 5880，虽然只选取 *2*2 感受野之和，但也存在 *2*2 的连接数，1 为偏置项的连接，1*414 为输出特征层，每一个像素都由前面卷积得到，即总共经历 1*414 次卷积

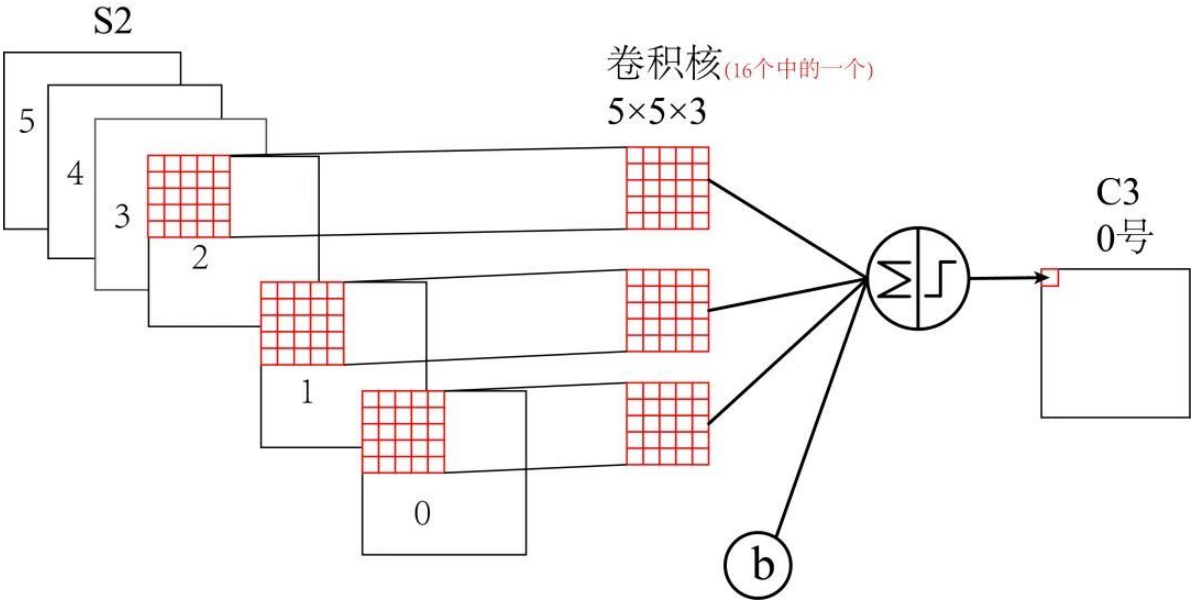
C3层

C3 层是卷积层，使用 16 个 5×5xn 大小的卷积核，padding=0，stride=1 进行卷积，得到 16 个 10×10 大小的特征图：14-5+1=10。

16 个卷积核并不是都与 S2 的 6 个通道层进行卷积操作，如下图所示，C3 的前六个特征图（0,1,2,3,4,5）由 S2 的相邻三个特征图作为输入，对应的卷积核尺寸为：**5x5x3**；接下来的 6 个特征图（6,7,8,9,10,11）由 S2 的相邻四个特征图作为输入对应的卷积核尺寸为：**5x5x4**；接下来的 3 个特征图（12,13,14）号特征图由 S2 间断的四个特征图作为输入对应的卷积核尺寸为：**5x5x4**；最后的 15 号特征图由 S2 全部(6 个)特征图作为输入，对应的卷积核尺寸为：**5x5x6**

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

因卷积核是 5×5 且具有 3 个通道，每个通道各不相同，这也是下面计算时 5*5 后面还要乘以3,4,6的原因。这是多通道卷积的计算方法



参数个数：(5*5*3+1)*6+(5*5*4+1)*6+(5*5*4+1)*3+(5*5*6+1)*1=1516。

连接数：1516*10*10 = 151600。10*10为输出特征层，每一个像素都由前面卷积得到，即总共经历10*10次卷积

S4层

S4 层与 S2 一样也是降采样层，使用 16 个 2×2 大小的卷积核进行池化，padding=0，stride=2，得到 16 个 5×5 大小的特征图：10/2=5。

参数个数：(1+1)*16=32。

连接数：(2*2+1)*16*5*5= 2000

C5层

C5 层是卷积层，使用 120 个 5×5x16 大小的卷积核，padding=0，stride=1进行卷积，得到 120 个 1×1 大小的特征图：5-5+1=1。即相当于 120 个神经元的全连接层。

值得注意的是，与C3层不同，这里120个卷积核都与S4的16个通道层进行卷积操作。

参数个数：(5*5*16+1)*120=48120。

连接数：48120*1*1=48120

F6层

F6 是全连接层，共有 84 个神经元，与 C5 层进行全连接，即每个神经元都与 C5 层的 120 个特征图相连。计算输入向量和权重向量之间的点积，再加上一个偏置，结果通过 sigmoid 函数输出。

F6 层有 84 个节点，对应于一个 7x12 的比特图，-1 表示白色，1 表示黑色，这样每个符号的比特图的黑白色就对应于一个编码。该层的训练参数和连接数是(120 + 1)x84=10164。ASCII 编码图如下：

参数个数：(120+1)*84=10164

连接数：(120+1)*84=10164

输出层

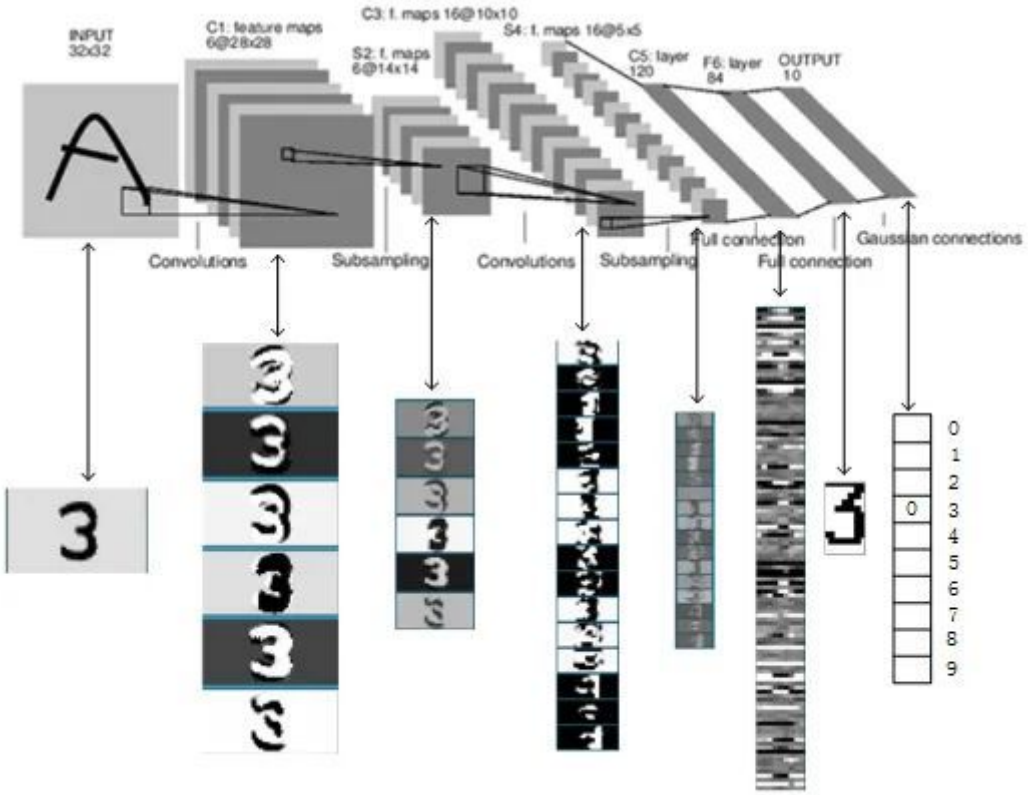
最后的 Output 层也是全连接层，是 Gaussian Connections，采用了 RBF 函数（即径向欧式距离函数），计算输入向量和参数向量之间的欧式距离（目前已经被Softmax 取代）。

Output 层共有 10 个节点，分别代表数字 0 到 9。假设x是上一层的输入，y 是 RBF的输出，则 RBF 输出的计算方式是：

$$y_i = \sum_{j=0}^{83} (x_j - w_{ij})^2$$

上式中 i 取值从 0 到 9，j 取值从 0 到 7*12-1，w 为参数。RBF 输出的值越接近于 0，则越接近于 i，即越接近于 i 的 ASCII 编码图，表示当前网络输入的识别结果是字符 i。

下图是数字 3 的识别过程：



参数个数：84*10=840。

连接数：84*10=840。

代码细节

卷积层

初始化

```
class conv():

    def __init__(self, filter_shape, stride=1, padding='SAME', bias=True,
requires_grad=True):
        self.weight = parameter(np.random.randn(*filter_shape) *
(2/reduce(lambda x,y:x*y, filter_shape[1:]))**0.5)
        self.stride = stride
        self.padding = padding
        self.requires_grad = requires_grad
        self.output_channel = filter_shape[0]
        self.input_channel = filter_shape[1]
        self.filter_size = filter_shape[2]
        if bias:
            self.bias = parameter(np.random.randn(self.output_channel))
        else:
            self.bias =None
```

将图转为向量

```
def image_to_col(self, x, filter_size_x, filter_size_y, stride):
        N, C, H, W = x.shape
        output_H, output_W = (H-filter_size_x)//stride + 1, (W-
filter_size_y)//stride + 1
        out_size = output_H * output_W
        x_cols = np.zeros((out_size*N, filter_size_x*filter_size_y*C))
        for i in range(0, H-filter_size_x+1, stride):
            i_start = i * output_W
            for j in range(0, W-filter_size_y+1, stride):
                temp = x[:,:, i:i+filter_size_x,
j:j+filter_size_y].reshape(N,-1)
                x_cols[i_start+j::out_size, :] = temp
        return x_cols
```

前向传播

```
def forward(self, input):

        # 边缘填充
        if self.padding == "VALID":
            self.x = input
        if self.padding == "SAME":
            p = self.filter_size // 2
            self.x = np.lib.pad(input, ((0,0),(0,0),(p,p),(p,p)), "constant")
        # 处理不能恰好的被卷积核的大小和选定的步长所整除的宽和高
```

```
        x_fit = (self.x.shape[2] - self.filter_size) % self.stride
        y_fit = (self.x.shape[3] - self.filter_size) % self.stride

        if self.stride > 1:
            if x_fit != 0:
                self.x = self.x[:, :, 0:self.x.shape[2] - x_fit, :]
            if y_fit != 0:
                self.x = self.x[:, :, :, 0:self.x.shape[3] - y_fit]

        N, _, H, W = self.x.shape
        O, C, K, K = self.weight.data.shape
        weight_cols = self.weight.data.reshape(O, -1).T
        x_cols = self.image_to_col(self.x, self.filter_size, self.filter_size,
self.stride)
        result = np.dot(x_cols, weight_cols) + self.bias.data
        output_H, output_W = (H-self.filter_size)//self.stride + 1, (W-
self.filter_size)//self.stride + 1
        result = result.reshape((N, result.shape[0]//N, -1)).reshape((N,
output_H, output_W, O))
        return result.transpose((0, 3, 1, 2))
```

反向传播

```
def backward(self, eta, lr):

        if self.stride > 1:
            N, O, output_H, output_W = eta.shape
            inserted_H, inserted_W = output_H + (output_H-1)*(self.stride-1),
output_W + (output_W-1)*(self.stride-1)
            inserted_eta = np.zeros((N, O, inserted_H, inserted_W))
            inserted_eta[:,:,::self.stride, ::self.stride] = eta
            eta = inserted_eta

        N, _, output_H, output_W = eta.shape
        self.b_grad = eta.sum(axis=(0,2,3))
        self.W_grad = np.zeros(self.weight.data.shape)
        for i in range(self.filter_size):
            for j in range(self.filter_size):
                self.W_grad[:,:,i,j] = np.tensordot(eta,
self.x[:,:,i:i+output_H,j:j+output_W], ([0,2,3], [0,2,3]))
        self.weight.data -= lr * self.W_grad / N
        self.bias.data -= lr * self.b_grad / N


        if self.padding == "VALID":
            p = self.filter_size - 1
            pad_eta = np.lib.pad(eta, ((0,0),(0,0),(p,p),(p,p)), "constant",
constant_values=0)
            eta = pad_eta
        elif self.padding == "SAME":
            p = self.filter_size // 2
            pad_eta = np.lib.pad(eta, ((0, 0), (0, 0), (p, p), (p, p)),
"constant", constant_values=0)
            eta = pad_eta
```

```
        _, C, _, _ = self.weight.data.shape
        weight_flip = np.flip(self.weight.data, (2,3))
        weight_flip_swap = np.swapaxes(weight_flip, 0, 1)
        weight_flip = weight_flip_swap.reshape(C, -1).T
        x_cols = self.image_to_col(eta, self.filter_size, self.filter_size,
self.stride)
        result = np.dot(x_cols, weight_flip)
        N, _, H, W = eta.shape
        output_H, output_W = (H - self.filter_size) // self.stride + 1, (W -
self.filter_size) // self.stride + 1
        result = result.reshape((N, result.shape[0] // N, -1)).reshape((N,
output_H, output_W, C))
        self.weight.grad = result.transpose((0, 3, 1, 2))

        return self.weight.grad
```

全连接层

初始化

```
def __init__(self, input_num, output_num, bias=True, requires_grad=True):
        self.input_num = input_num
        self.output_num = output_num
        self.requires_grad = requires_grad
        self.weight = parameter(np.random.randn(self.input_num, self.output_num)
* (2/self.input_num**0.5))
        if bias:
            self.bias = parameter(np.random.randn(self.output_num))
        else:
            self.bias = None
```

前向传播

```
 def forward(self, input):
        self.input_shape = input.shape
        if input.ndim > 2:
            N, C, H, W = input.shape
            self.x = input.reshape((N, -1))
        elif input.ndim == 2:
            self.x = input
        else:
            print("fc.forward error")
        result = np.dot(self.x, self.weight.data)
        if self.bias is not None:
            result = result + self.bias.data
        return result
```

反向传播

```
    def backward(self, eta, lr):
        N, _ = eta.shape
        next_eta = np.dot(eta, self.weight.data.T)
        self.weight.grad = np.reshape(next_eta, self.input_shape)
```

```
        x = self.x.repeat(self.output_num, axis=0).reshape((N, self.output_num,
-1))
        self.W_grad = x * eta.reshape((N, -1, 1))
        self.W_grad = np.sum(self.W_grad, axis=0) / N
        self.b_grad = np.sum(eta, axis=0) / N


        self.weight.data -= lr * self.W_grad.T
        self.bias.data -= lr * self.b_grad

        return self.weight.grad
```

池化层（下采样层）

初始化

```
class Pooling():
    def __init__(self, kernel_size=(2, 2), stride=2, ):
        self.ksize = kernel_size
        self.stride = stride
```

正向传播

```
def forward(self, input):
        N, C, H, W = input.shape
        out = input.reshape(N, C, H//self.stride, self.stride, W//self.stride,
self.stride)
        out = out.max(axis=(3,5))
        self.mask = out.repeat(self.ksize[0], axis=2).repeat(self.ksize[1],
axis=3) != input
        return out
```

反向传播

```
def backward(self, eta):
        result = eta.repeat(self.ksize[0], axis=2).repeat(self.ksize[1], axis=3)
        result[self.mask] = 0
        return result
```

加载数据

```
import glob
import struct
import time
import os
def load_mnist(file_dir, is_images='True'):
    bin_file = open(file_dir, 'rb')
    bin_data = bin_file.read()
    bin_file.close()
    if is_images:
```

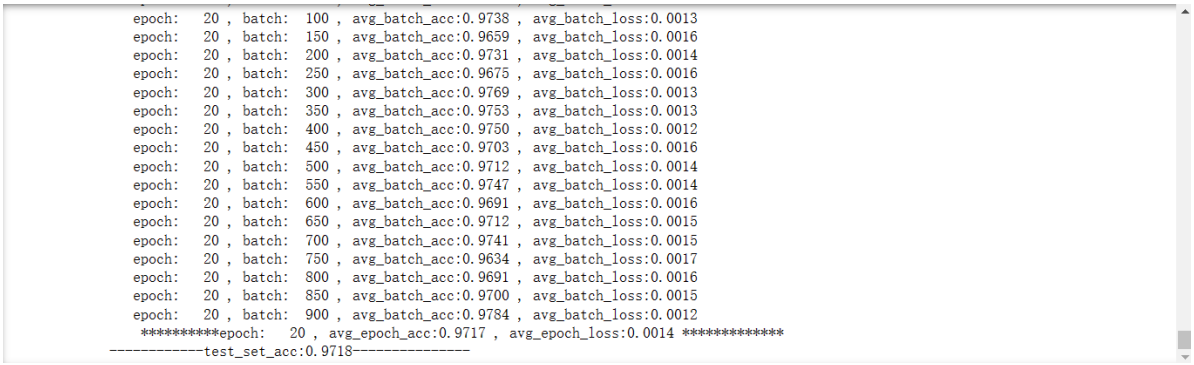
```
        fmt_header = '>iiii'
        magic, num_images, num_rows, num_cols = struct.unpack_from(fmt_header,
bin_data, 0)
    else:
        fmt_header = '>ii'
        magic, num_images = struct.unpack_from(fmt_header, bin_data, 0)
        num_rows, num_cols = 1, 1
    data_size = num_images * num_rows * num_cols
    mat_data = struct.unpack_from('>' + str(data_size) + 'B', bin_data,
struct.calcsize(fmt_header))
    mat_data = np.reshape(mat_data, [num_images, num_rows * num_cols])
    print('Load images from %s, number: %d, data shape: %s' % (file_dir,
num_images, str(mat_data.shape)))
    return mat_data


def data_convert(x, y, m, k):
    x[x<=40]=0
    x[x>40]=1
    ont_hot_y = np.zeros((m,k))
    for t in np.arange(0,m):
        ont_hot_y[t,y[t]]=1
    ont_hot_y=ont_hot_y.T
    return x, ont_hot_y


def load_data(mnist_dir, train_data_dir, train_label_dir, test_data_dir,
test_label_dir):
    print('Loading MNIST data from files...')
    train_images = load_mnist(os.path.join(mnist_dir, train_data_dir), True)
    train_labels = load_mnist(os.path.join(mnist_dir, train_label_dir), False)
    test_images = load_mnist(os.path.join(mnist_dir, test_data_dir), True)
    test_labels = load_mnist(os.path.join(mnist_dir, test_label_dir), False)
    return train_images, train_labels, test_images, test_labels

mnist_dir = "mnist_data/"
train_data_dir = "train-images.idx3-ubyte"
train_label_dir = "train-labels.idx1-ubyte"
test_data_dir = "t10k-images.idx3-ubyte"
test_label_dir = "t10k-labels.idx1-ubyte"

train_images, train_labels, test_images, test_labels = load_data(mnist_dir,
train_data_dir, train_label_dir, test_data_dir, test_label_dir)
print("Got data. ")
```

训练和测试

```
batch_size = 64
test_batch = 50
epoch = 20
learning_rate = 1e-3

iterations_num = 0

net = LeNet5()
```

```
for E in range(epoch):
    batch_loss = 0
    batch_acc = 0

    epoch_loss = 0
    epoch_acc = 0

    for i in range(train_images.shape[0] // batch_size):
        img = train_images[i*batch_size:(i+1)*batch_size].reshape(batch_size, 1,
28, 28)
        img = normalization(img)
        label = train_labels[i*batch_size:(i+1)*batch_size]
        loss, prediction = net.forward(img, label, is_train=True)

        epoch_loss += loss
        batch_loss += loss
        for j in range(prediction.shape[0]):
            if np.argmax(prediction[j]) == label[j]:
                epoch_acc += 1
                batch_acc += 1

        net.backward(learning_rate)

        if (i+1)%50 == 0:
            print("   epoch:%5d , batch:%5d , acc:%.4f , loss:%.4f "
                  % (E+1, i+1, batch_acc/(batch_size*50),
batch_loss/(batch_size*50)))


            batch_loss = 0
            batch_acc = 0



    print("--------------epoch:%5d , avg_acc:%.4f , avg_loss:%.4f---------------
"
          % (E+1, epoch_acc/train_images.shape[0],
epoch_loss/train_images.shape[0]))
    test_acc = 0
    for k in range(test_images.shape[0] // test_batch):
        img = test_images[k*test_batch:(k+1)*test_batch].reshape(test_batch, 1
,28, 28)
        img = normalization(img)
        label = test_labels[k*test_batch:(k+1)*test_batch]
        _, prediction = net.forward(img, label, is_train=False)

        for j in range(prediction.shape[0]):
            if np.argmax(prediction[j]) == label[j]:
                test_acc += 1

    print("------------total_acc:%.4f---------------" % (test_acc /
test_images.shape[0]))
```

运行结果



在20轮训练后，模型基本收敛，可以得到**97.18%**的准确率

参考文献

网络解析（一）：LeNet-5详解 - 枫飞飞 - 博客园 (cnblogs.com)

这可能是神经网络 LeNet-5 最详细的解释了！ - 腾讯云开发者社区-腾讯云 (tencent.com)