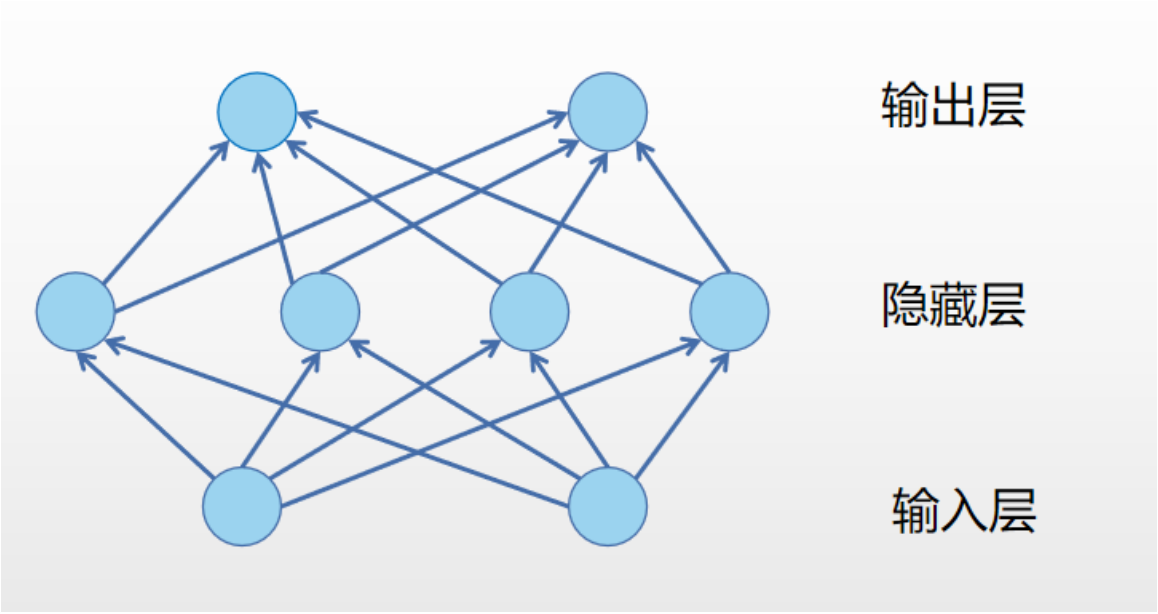
1. 基于MLP的手写数字识别机制

1.1 MLP：多层感知机

多层感知机（MLP，Multilayer Perceptron）也叫人工神经网络（ANN，Artificial Neural Network），它的架构有输入层、输出层，在输入层与输出层的中间可以有多个隐层，最基本的结构就是只有一个隐层的MLP，如下图：



从上图可以看到，多层感知机层与层之间是全连接的，也就是说上一层的任何一个神经元与下一层的所有神经元都有连接。多层感知机最底层是输入层，中间是隐藏层，最后是输出层。

首先，输入层是用来接收的，你输入是一个n维向量，就有n个神经元。

接着，就是隐藏层，它是用来“修饰”的，假设输入层用向量X表示，则隐藏层的输出就是f(W1X+b1)，其中W1是权重（也叫连接系数），b1是偏置。常用激活函数。

最后是输出层，进行输出。

1.2 激活函数

激活函数，就是在人工神经网络的神经元上运行的函数，负责将神经元的输入映射到输出端。使用激活函数，能够给神经元引入非线性因素，使得神经网络可以任意逼近任何非线性函数，这样神经网络就可以利用到更多的非线性模型中。

激活函数需要具备的性质：

1. 连续并可导（允许少数点上不可导）的非线性函数。可导的激活函数可以直接利用数值优化的方法来学习网络参数。
2. 激活函数及其导函数要尽可能的简单，有利于提高网络计算效率。
3. 激活函数的导函数的值域要在一个合适的区间内，不能太大也不能太小，否则会影响训练的效率和稳定性。

常见的激活函数：

sigmoid函数，也叫Logistic函数，用于隐层神经元输出,取值范围为(0,1)，它可以将个实数映射到(0,1)的区间,可以用来做二分类。在特征相差比较复杂或是相差不是特别大时效果比较好。sigmoid缺点:激活函数计算量大，反向传播求误差梯度时，求导涉及除法。反向传播时,很容易就会出现梯度消失的情况，从而无法完成深层网络的训练。Sigmoid函数饱和且kill掉梯度。Sigmoid函数收敛缓慢。

Tanh函数，优点：比Sigmoid收敛速度快，输出以0为中心；缺点：函数饱和，梯度容易消失

ReLU函数，优点：收敛速度比上述激活函数更快，计算简单；缺点：当输入小于0时，训练参数将无法更新。

Leaky-ReLU函数，优点：收敛速度快，解决神经 元死亡现象；缺点：实际应用中发现效果不稳定。

1.3 损失函数

损失函数（loss function）或代价函数（cost function）是将随机事件或其有关随机变量的取值映射为非负实数以表示该随机事件的“风险”或“损失”的函数。在应用中，损失函数通常作为学习准则与优化问题相联系，即通过最小化损失函数求解和评估模型。

1.4 softmax函数

softmax逻辑回归模型是logistic回归模型在多分类问题上的推广，在多分类问题中，类标签y可以取两个以上的值。 Softmax回归模型对于诸如MNIST手写数字分类等问题是很有用的,该问题的目的是辨识10个不同的单个数字。Softmax回归是有监督的，不过后面也会介绍它与深度学习无监督学习方法的结合。

2、设计方法说明

1、激活函数：ReLU函数；
2、损失函数：交叉熵；
3、两个隐层；
4、梯度下降法；

激活函数：

线性整流函数（Rectified Linear Unit, ReLU），又称修正线性单元，是一种人工神经网络中常用的激活函数（activation function），通常指代以斜坡函数及其变种为代表的非线性函数。

$$f(x) = \max(0,x)$$

损失函数：

交叉熵刻画了两个概率分布之间的距离，它是分类问题中使用比较广的一种损失函数。

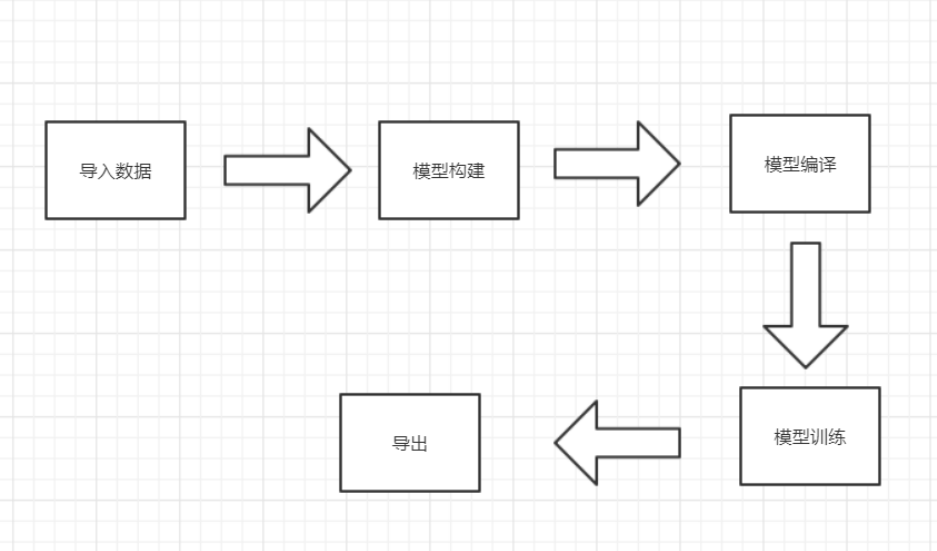
给定两个概率分布 p 和 q，通过 q 来表示 p 的交叉熵为：

$$H(p,q)=-\sum p(x)\log q(x)$$

p 代表正确答案，q 代表的是预测值。交叉熵值越小，两个概率分布越接近。需要注意的是，交叉熵刻画的是两个概率分布之间的距离，然而神经网络的输出却不一定是一个概率分布，很多情况下是实数。如何将神经网络前向传播得到的结果也变成概率分布，Softmax 回归就是一个非常有用的方法。

4、 代码说明

代码流程：



总体代码：

```
from tensorflow.examples.tutorials.mnist import input_data
import matplotlib.pyplot as plt
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

mnist = input_data.read_data_sets('C:/Users/dell/.spyder-py3/MNIST_data',one_hot=True)
#每个批次大小
batch_size=60
learn_rate=0.01#学习率
round=300#迭代次数
#计算有多少批次
n_batch=mnist.train.num_examples //batch_size
#定义占位符
x=tf.placeholder(tf.float32,[None,784])
y=tf.placeholder(tf.float32,[None,10])

#定义隐层，初始化w,b，使用ReLU做激活函数
#W=tf.Variable(tf.random.normal([784,10]))
#b=tf.Variable(tf.random.normal([10]))
#prediction=tf.matmul(x,W)+b
W_L1=tf.Variable(tf.zeros([784,100]))
b_L1=tf.Variable(tf.random.normal([100]))
WL1_plus_b=tf.matmul(x,W_L1)+b_L1
L1=tf.nn.relu(WL1_plus_b)

```

```
W_L2=tf.Variable(tf.random.normal([100,10]))
b_L2=tf.Variable(tf.random.normal([10]))
WL2_plus_b=tf.matmul(L1,W_L2)+b_L2

prediction=WL2_plus_b

#定义损失函数，用交叉熵
cost=tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y,logits=prediction))

epoch_J={"epoch":[],"loss":[]}
#定义梯度下降法
train_step=tf.train.GradientDescentOptimizer(learn_rate).minimize(cost)
#返回预测结果是否正确
correct_prediction=tf.equal(tf.argmax(y,1),tf.argmax(prediction,1))
#求准确率
accuracy=tf.reduce_mean(tf.cast(correct_prediction,tf.float32))
#初始化变量
init=tf.global_variables_initializer()


with tf.Session() as sess:
    sess.run(init)
    for i in range(round):
        for batch in range(n_batch):
            batch_xs,batch_ys=mnist.train.next_batch(batch_size)
```

```
            sess.run(train_step,feed_dict={x:batch_xs,y:batch_ys})
            loss=sess.run(cost,feed_dict={x:batch_xs,y:batch_ys})

        epoch_J["epoch"].append(i+1)
        epoch_J["loss"].append(loss)
        #print("迭代次数：",i+1,"loss值：",epoch_J["loss"][i],"当前W：",sess.run(W_L2),"当前b:",
    #画代价函数图
    plt.plot(epoch_J["epoch"],epoch_J["loss"],'ro',label="epoch_loss")
    plt.legend()
    plt.show()
    acc=sess.run(accuracy,feed_dict={x:mnist.test.images,y:mnist.test.labels})
    print("Iter "+str(i+1)+",testring acc"+str(acc))
```

代码标注：

参数设置：

```
batch_size=60
learn_rate=0.01#学习率
round=300#迭代次数
```

Size为60 迭代次数为300

训练过程：

```
with tf.Session() as sess:
    sess.run(init)
    for i in range(round):
        for batch in range(n_batch):
            batch_xs,batch_ys=mnist.train.next_batch(batch_size)
```

```
            sess.run(train_step,feed_dict={x:batch_xs,y:batch_ys})
            loss=sess.run(cost,feed_dict={x:batch_xs,y:batch_ys})

        epoch_J["epoch"].append(i+1)
        epoch_J["loss"].append(loss)
```

其余代码均有标注。

5、 参数调整优化

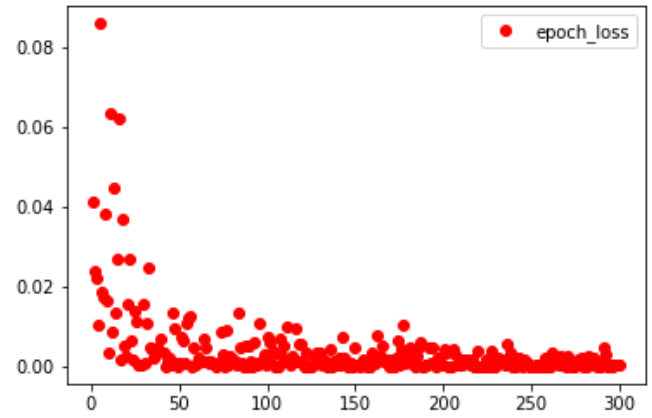
第一层隐层设置Batch_size=100 Learn_rate=0.01 Round=100

发现虽然出现了我们需要的情况，但是精确度不高，所以增加第二层隐层。

第二层隐层：

增加了一层隐层之后，由于增加了一层隐层后，模型训练的效果显著增加，发现精确度确实变高了许多。

```
batch_size=10
learn_rate=0.01#学习率
round=300#迭代次数
```

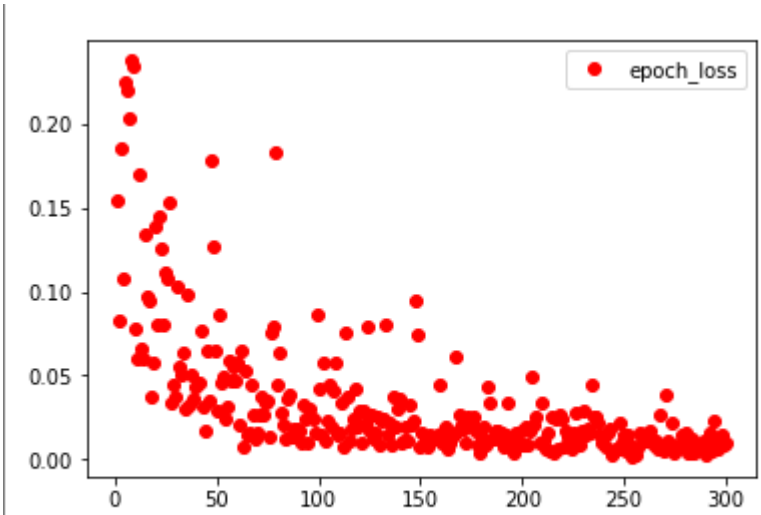


Iter 300,testring acc0.9705

我们继续通过改变batch-size来进行参数优化：

1、batch-size改为60

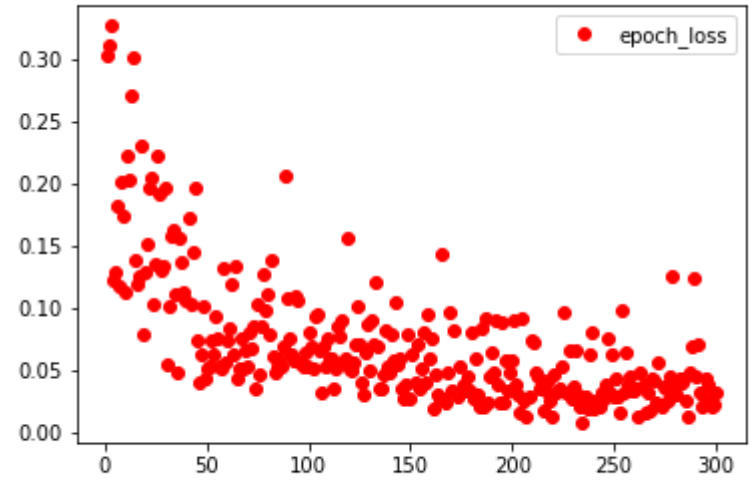
```
batch_size=60
learn_rate=0.01#学习率
round=300#迭代次数
```



Iter 300,testring acc0.9715

2、batch-size改为128

```
batch_size=128
learn_rate=0.01#学习率
round=300#迭代次数
```



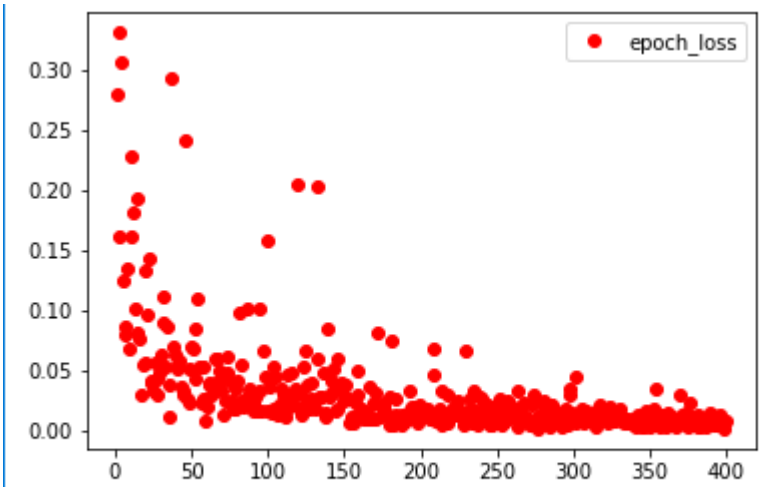
Iter 300,testring acc0.9635

通过这个发现，训练增加，训练效果提高，当batch-size在60左右时精度最高，往高了精确度会下降。

我们接着通过改变迭代次数来进行参数优化：

1、改变迭代次数为400

```
8 batch_size=60
9 learn_rate=0.01#学习率
0 round=400#迭代次数
```



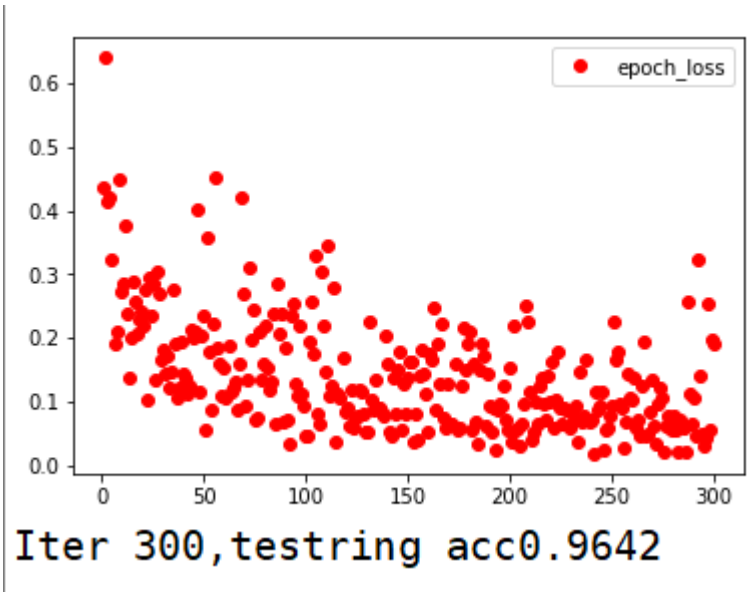
Iter 400,testring acc0.9687

我们发现当迭代次数增加时，精度会下降。

我们最后通过调整学习率来进行参数优化：

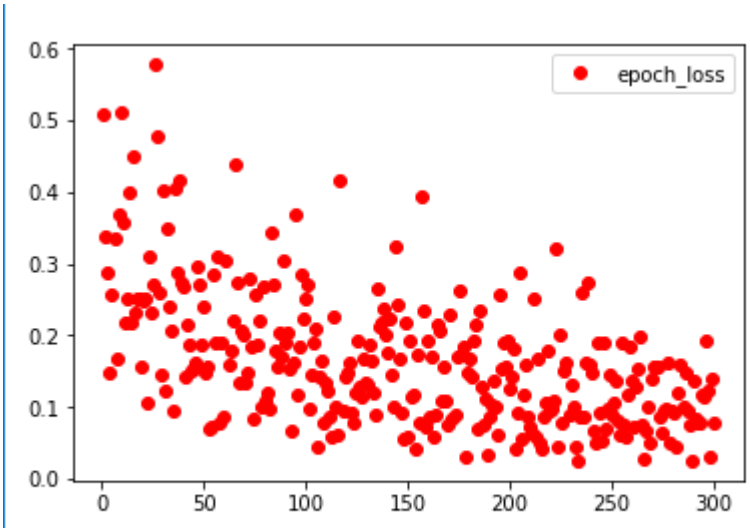
1、改变学习率为0.001

```
batch_size=60
learn_rate=0.001#学习率
round=300#迭代次数
```



2、我们改变学习率为0.0005

```
batch_size=60
learn_rate=0.0005#学习率
round=300#迭代次数
```

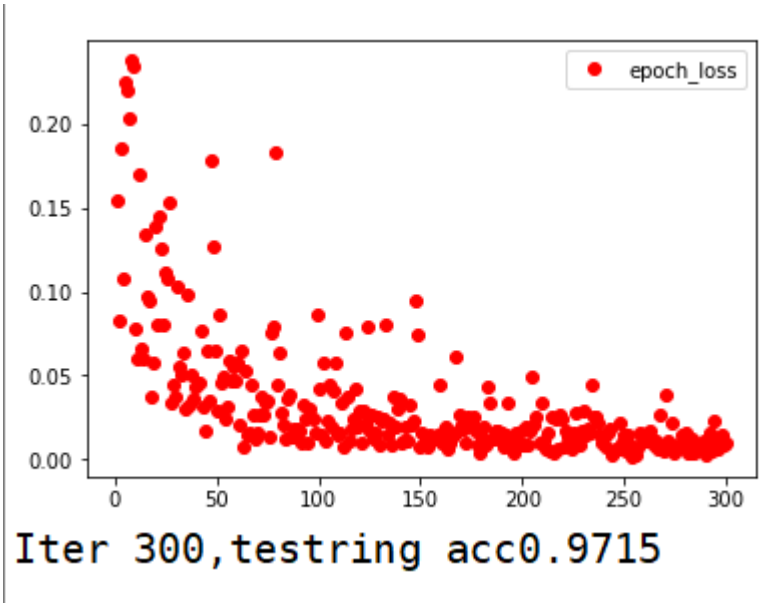


我们可以发现，当学习率不断降低时，他的吻合度也越来越小，精确度也越来越小。

实验结果：

所以取精确度最好的当实验结果。

```
batch_size=60
learn_rate=0.01#学习率
round=300#迭代次数
```



精确度为0.9715

6、 代码安装说明

Anaconda中的spyder
Python：3.5
系统：win10
Tensorflow
第一步配置 Tensorflow环境：
1、 创建 TensorFlow环境 create-name tensorflow python=3.5
2、激活 tensorflow环境：activate tensorflow
第二步，下载安装tensorflow，注意版本。
Python：3.5
系统：win10

7、 基于CNN的实现

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 25 21:44:05 2020

@author: dell
"""

from tensorflow.examples.tutorials.mnist import input_data
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

# 定义神经网络模型的评估部分
def compute_accuracy(test_xs, test_ys):
    # 使用全局变量prediction
    global prediction
    # 获得预测值y_pre
    y_pre = sess.run(prediction, feed_dict = { xs: test_xs, keep_prob: 1})
    # 判断预测值y和真实值y_中最大数的索引是否一致，y_pre的值为1-10概率，返回值为bool序列
    correct_prediction = tf.equal(tf.argmax(y_pre, 1), tf.argmax(test_ys, 1))
    # 定义准确率的计算
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32)) #tf.cast将bool转换为float32
    # 计算准确率
    result = sess.run(accuracy)
    return result
```

```
# 下载mnist数据
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)

# 权重参数初始化
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev = 0.1)  #截断的正态分布，标准差stddev
    return tf.Variable(initial)

# 偏置参数初始化
def bias_variable(shape):
    initial = tf.constant(0.1, shape = shape)
    return tf.Variable(initial)

# 定义卷积层
def conv2d(x, W):
    # stride的四个参数：[batch, height, width, channels], [batch_size, image_rows, image_cols, number_of_c
    # height, width就是图像的高度和宽度，batch和channels在卷积层中通常设为1
    return tf.nn.conv2d(x, W, strides = [1, 1, 1, 1], padding = 'SAME')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize = [1, 2, 2, 1], strides = [1, 2, 2, 1], padding = 'SAME')
    """
    max_pool(x,ksize,strides,padding)参数含义
        x:input
        ksize:filter，滤波器大小2*2
```

```
        strides:步长，2*2，表示filter窗口每次水平移动2格，每次垂直移动2格
        padding:填充方式，补零
    conv2d(x,W,strides=[1,1,1,1],padding='SAME')参数含义与上述类似
        x:input
        W:filter，滤波器大小
        strides:步长，1*1，表示filter窗口每次水平移动1格，每次垂直移动1格
        padding:填充方式，补零('SAME')
    """


# 输入输出数据的placeholder
xs = tf.placeholder(tf.float32, [None, 784])
ys = tf.placeholder(tf.float32, [None, 10])
# dropout的比例
keep_prob = tf.placeholder(tf.float32)

# 对数据进行重新排列，形成图像
x_image = tf.reshape(xs, [-1, 28, 28, 1])#  -1, 28, 28, 1

print(x_image.shape)

# 卷积层一
# patch为5*5，in_size为1，即图像的厚度，如果是彩色，则为3，32是out_size，输出的大小-》32个卷积和（滤波器）
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
```

```
# ReLU操作，输出大小为28*28*32
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
# Pooling操作，输出大小为14*14*32
h_pool1 = max_pool_2x2(h_conv1)

#  卷积层二
# patch为5*5，in_size为32，即图像的厚度，64是out_size，输出的大小
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])
# ReLU操作，输出大小为14*14*64
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
# Pooling操作，输出大小为7*7*64
h_pool2 = max_pool_2x2(h_conv2)

#  全连接层一
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])
# 输入数据变换
h_pool2_flat = tf.reshape(h_pool2, [-1, 7 * 7 * 64])  #整形成m*n，列n为7*7*64
# 进行全连接操作
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)  # tf.matmul
# 防止过拟合，dropout
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

```

```
0.1079
0.9452
0.96
0.9734
0.9733
0.976
0.9819
0.9803
0.9834
0.9832
```

最后的精确度为0.9832