下面我们首先需要了解 TensorFlow 的基本用法，这样我们才能开始构建神经网络。本小节将从张量与图、常数与变量还有占位符等基本概念出发简要介绍 TensorFlow，熟悉 TensorFlow 的读者可以直接阅读下一节。需要进一步了解 TensorFlow 的读者最好可以阅读谷歌 TensorFlow 的文档，当然也可以阅读其他中文教程或书籍，例如《TensorFlow：实战 Google 深度学习框架》和《TensorFlow 实战》等。

TensorFlow 文档地址：<https://www.tensorflow.org/get_started/>

**1.1 张量和图**

TensorFlow 是一种采用数据流图（data flow graphs），用于数值计算的开源软件库。其中 Tensor 代表传递的数据为张量（多维数组），Flow 代表使用计算图进行运算。数据流图用「结点」（nodes）和「边」（edges）组成的有向图来描述数学运算。「结点」一般用来表示施加的数学操作，但也可以表示数据输入的起点和输出的终点，或者是读取/写入持久变量（persistent variable）的终点。边表示结点之间的输入/输出关系。这些数据边可以传送维度可动态调整的多维数据数组，即张量（tensor）。



下面代码是使用计算图的案例：

a = tf.constant(2, tf.int16)  
b = tf.constant(4, tf.float32)  
  
graph = tf.Graph()  
with graph.as\_default():  
 a = tf.Variable(8, tf.float32)  
 b = tf.Variable(tf.zeros([2,2], tf.float32))  
   
with tf.Session(graph=graph) as session:  
 tf.global\_variables\_initializer().run()  
 print(session.run(a))  
 print(session.run(b))  
  
#输出：  
>>> 8  
>>> [[ 0. 0.]  
>>> [ 0. 0.]]

在 Tensorflow 中，所有不同的变量和运算都是储存在计算图。所以在我们构建完模型所需要的图之后，还需要打开一个会话（Session）来运行整个计算图。在会话中，我们可以将所有计算分配到可用的 CPU 和 GPU 资源中。

如下所示代码，我们声明两个常量 a 和 b，并且定义一个加法运算。但它并不会输出计算结果，因为我们只是定义了一张图，而没有运行它：

1. a=tf.constant([1,2],name="a")
2. b=tf.constant([2,4],name="b")
3. result = a+b
4. print(result)

#输出：Tensor("add:0", shape=(2,), dtype=int32)

下面的代码才会输出计算结果，因为我们需要创建一个会话才能管理 TensorFlow 运行时的所有资源。但计算完毕后需要关闭会话来帮助系统回收资源，不然就会出现资源泄漏的问题。下面提供了使用会话的两种方式：

a=tf.constant([1,2,3,4])  
b=tf.constant([1,2,3,4])  
result=a+b  
sess=tf.Session()  
print(sess.run(result))  
sess.close  
  
#输出 [2 4 6 8]  
  
with tf.Session() as sess:  
    a=tf.constant([1,2,3,4])  
    b=tf.constant([1,2,3,4])  
    result=a+b  
    print(sess.run(result))  
  
#输出 [2 4 6 8]

**1.2 常量和变量**

TensorFlow 中最基本的单位是常量（Constant）、变量（Variable）和占位符（Placeholder）。常量定义后值和维度不可变，变量定义后值可变而维度不可变。在神经网络中，变量一般可作为储存权重和其他信息的矩阵，而常量可作为储存超参数或其他结构信息的变量。下面我们分别定义了常量与变量：

a = tf.constant(2, tf.int16)  
 b = tf.constant(4, tf.float32)  
 c = tf.constant(8, tf.float32)  
  
 d = tf.Variable(2, tf.int16)  
 e = tf.Variable(4, tf.float32)  
 f = tf.Variable(8, tf.float32)  
  
  
  
 g = tf.constant(np.zeros(shape=(2,2), dtype=np.float32))  
  
  
 h = tf.zeros([11], tf.int16)  
 i = tf.ones([2,2], tf.float32)  
 j = tf.zeros([1000,4,3], tf.float64)  
  
 k = tf.Variable(tf.zeros([2,2], tf.float32))  
 l = tf.Variable(tf.zeros([5,6,5], tf.float32))

在上面代码中，我们分别声明了不同的常量（tf.constant()）和变量（tf.Variable()），其中 tf.float 和 tf.int 分别声明了不同的浮点型和整数型数据。而 tf.ones() 和 tf.zeros() 分别产生全是 1、全是 0 的矩阵。我们注意到常量 g，它的声明结合了 TensorFlow 和 Numpy，这也是可执行的。

w1=tf.Variable(tf.random\_normal([2,3],stddev=1,seed=1))

以上语句声明一个 2 行 3 列的变量矩阵，该变量的值服从标准差为 1 的正态分布，并随机生成。TensorFlow 还有 tf.truncated\_normal() 函数，即截断正态分布随机数，它只保留 [mean-2\*stddev,mean+2\*stddev] 范围内的随机数。

现在，我们可以应用变量来定义神经网络中的权重矩阵和偏置项向量：

weights = tf.Variable(tf.truncated\_normal([256 \* 256, 10]))  
biases = tf.Variable(tf.zeros([10]))  
print(weights.get\_shape().as\_list())  
print(biases.get\_shape().as\_list())  
#输出  
>>>[65536, 10]  
>>>[10]

**1.3 占位符和 feed\_dict**

我们已经创建了各种形式的常量和变量，但 TensorFlow 同样还支持占位符。占位符并没有初始值，它只会分配必要的内存。在会话中，占位符可以使用 feed\_dict 馈送数据。

feed\_dict 是一个字典，在字典中需要给出每一个用到的占位符的取值。在训练神经网络时需要每次提供一个批量的训练样本，如果每次迭代选取的数据要通过常量表示，那么 TensorFlow 的计算图会非常大。因为每增加一个常量，TensorFlow 都会在计算图中增加一个结点。所以说拥有几百万次迭代的神经网络会拥有极其庞大的计算图，而占位符却可以解决这一点，它只会拥有占位符这一个结点。

下面一段代码分别展示了使用常量和占位符进行计算：

w1=tf.Variable(tf.random\_normal([1,2],stddev=1,seed=1))  
  
#因为需要重复输入x，而每建一个x就会生成一个结点，计算图的效率会低。所以使用占位符  
x=tf.placeholder(tf.float32,shape=(1,2))  
x1=tf.constant([[0.7,0.9]])  
  
a=x+w1  
b=x1+w1  
  
sess=tf.Session()  
sess.run(tf.global\_variables\_initializer())  
  
#运行y时将占位符填上，feed\_dict为字典，变量名不可变  
y\_1=sess.run(a,feed\_dict={x:[[0.7,0.9]]})  
y\_2=sess.run(b)  
print(y\_1)  
print(y\_2)  
sess.close

其中 y\_1 的计算过程使用占位符，而 y\_2 的计算过程使用常量。

下面是使用占位符的案例：

list\_of\_points1\_ = [[1,2], [3,4], [5,6], [7,8]]  
list\_of\_points2\_ = [[15,16], [13,14], [11,12], [9,10]]  
  
list\_of\_points1 = np.array([np.array(elem).reshape(1,2) for elem in list\_of\_points1\_])  
list\_of\_points2 = np.array([np.array(elem).reshape(1,2) for elem in list\_of\_points2\_])  
  
  
graph = tf.Graph()  
  
with graph.as\_default():     
  
    #我们使用 tf.placeholder() 创建占位符 ，在 session.run() 过程中再投递数据   
    point1 = tf.placeholder(tf.float32, shape=(1, 2))  
    point2 = tf.placeholder(tf.float32, shape=(1, 2))  
  
    def calculate\_eucledian\_distance(point1, point2):  
        difference = tf.subtract(point1, point2)  
        power2 = tf.pow(difference, tf.constant(2.0, shape=(1,2)))  
        add = tf.reduce\_sum(power2)  
        eucledian\_distance = tf.sqrt(add)  
        return eucledian\_distance  
  
    dist = calculate\_eucledian\_distance(point1, point2)  
  
  
with tf.Session(graph=graph) as session:  
    tf.global\_variables\_initializer().run()     
    for ii in range(len(list\_of\_points1)):  
        point1\_ = list\_of\_points1[ii]  
        point2\_ = list\_of\_points2[ii]  
  
        #使用feed\_dict将数据投入到[dist]中  
        feed\_dict = {point1 : point1\_, point2 : point2\_}  
        distance = session.run([dist], feed\_dict=feed\_dict)  
        print("the distance between {} and {} -> {}".format(point1\_, point2\_, distance))  
  
  
#输出：  
>>> the distance between [[1 2]] and [[15 16]] -> [19.79899]  
>>> the distance between [[3 4]] and [[13 14]] -> [14.142136]  
>>> the distance between [[5 6]] and [[11 12]] -> [8.485281]  
>>> the distance between [[7 8]] and [[ 9 10]] -> [2.8284271]

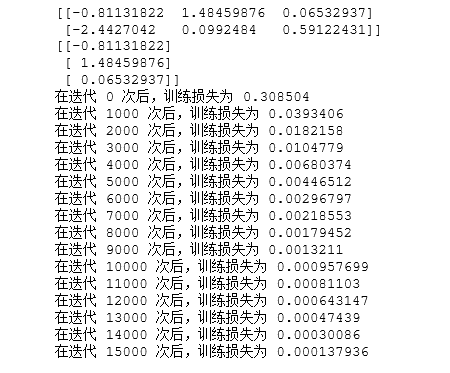
Ahmet Taspinar 在第二部分就直接开始构建深度神经网络了，虽然我们在前一章增加了许多代码段以帮助读者了解 TensorFlow 的基本法则，但上面是远远不够的。所以如果我们能先解析一部分神经网络代码，那么将有助于入门读者巩固以上的 TensorFlow 基本知识。下面，我们将先解析一段构建了三层全连接神经网络的代码。

import tensorflow as tf  
from numpy.random import RandomState  
  
  
batch\_size=10  
w1=tf.Variable(tf.random\_normal([2,3],stddev=1,seed=1))  
w2=tf.Variable(tf.random\_normal([3,1],stddev=1,seed=1))  
  
  
# None 可以根据batch 大小确定维度，在shape的一个维度上使用None  
x=tf.placeholder(tf.float32,shape=(None,2))  
y=tf.placeholder(tf.float32,shape=(None,1))  
  
  
#激活函数使用ReLU  
a=tf.nn.relu(tf.matmul(x,w1))  
yhat=tf.nn.relu(tf.matmul(a,w2))  
  
  
#定义交叉熵为损失函数，训练过程使用Adam算法最小化交叉熵  
cross\_entropy=-tf.reduce\_mean(y\*tf.log(tf.clip\_by\_value(yhat,1e-10,1.0)))  
train\_step=tf.train.AdamOptimizer(0.001).minimize(cross\_entropy)  
  
rdm=RandomState(1)  
data\_size=516  
  
  
#生成两个特征，共data\_size个样本  
X=rdm.rand(data\_size,2)  
#定义规则给出样本标签，所有x1+x2<1的样本认为是正样本，其他为负样本。Y，1为正样本  
Y = [[int(x1+x2 < 1)] for (x1, x2) in X]  
  
with tf.Session() as sess:  
    sess.run(tf.global\_variables\_initializer())  
    print(sess.run(w1))  
    print(sess.run(w2))  
    steps=11000  
    for i in range(steps):  
  
  
        #选定每一个批量读取的首尾位置，确保在1个epoch内采样训练  
        start = i \* batch\_size % data\_size  
        end = min(start + batch\_size,data\_size)  
        sess.run(train\_step,feed\_dict={x:X[start:end],y:Y[start:end]})  
        if i % 1000 == 0:  
            training\_loss= sess.run(cross\_entropy,feed\_dict={x:X,y:Y})  
            print("在迭代 %d 次后，训练损失为 %g"%(i,training\_loss))

上面的代码定义了一个简单的三层全连接网络（输入层、隐藏层和输出层分别为 2、3 和 1 个神经元），隐藏层和输出层的激活函数使用的是 ReLU 函数。该模型训练的样本总数为 512，每次迭代读取的批量为 10。这个简单的全连接网络以交叉熵为损失函数，并使用 Adam 优化算法进行权重更新。

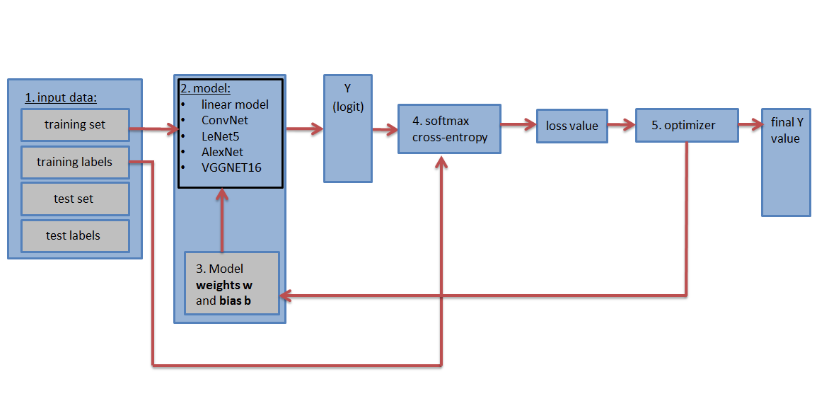
其中需要注意的几个函数如 tf.nn.relu() 代表调用 ReLU 激活函数，tf.matmul() 为矩阵乘法等。tf.clip\_by\_value(yhat,1e-10,1.0) 这一语句代表的是截断 yhat 的值，因为这一语句是嵌套在 tf.log() 函数内的，所以我们需要确保 yhat 的取值不会导致对数无穷大。

tf.train.AdamOptimizer(learning\_rate).minimize(cost\_function) 是进行训练的函数，其中我们采用的是 Adam 优化算法更新权重，并且需要提供学习速率和损失函数这两个参数。后面就是生成训练数据，X=rdm.rand(512,2) 表示随机生成 512 个样本，每个样本有两个特征值。最后就是迭代运行了，这里我们计算出每一次迭代抽取数据的起始位置（start）和结束位置（end），并且每一次抽取的数据量为前面我们定义的批量，如果一个 epoch 最后剩余的数据少于批量大小，那就只是用剩余的数据进行训练。最后两句代码是为了计算训练损失并迭代一些次数后输出训练损失。这一部分代码运行的结果如下：



**TensorFlow 中的神经网络**

**2.1 简介**

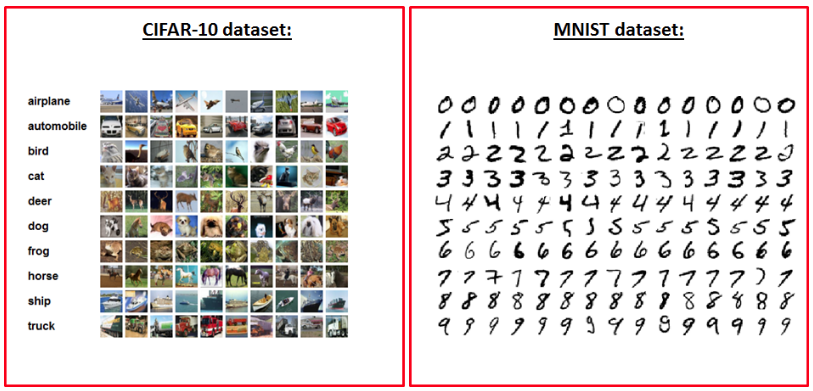


上图所描述的图像识别流程需要包含以下几步：

* 输入数据集，数据集分为训练数据集和标注、测试数据集和标注（包括验证数据集和标注）。测试和验证集能赋值到 tf.constant() 中，而训练集可以导入 tf.placeholder() 中，训练集只有导入占位符我们才能在随机梯度下降中成批量地进行训练。
* 确定神经网络模型，该模型可以是简单的一层全连接网络或 9 层、16 层的复杂卷积网络组成。
* 网络定义的权重矩阵和偏置向量后需要执行初始化，每一层需要一个权重矩阵和一个偏置向量。
* 构建损失函数，并计算训练损失。模型会输出一个预测向量，我们可以比较预测标签和真实标签并使用交叉熵函数和 softmax 回归来确定损失值。训练损失衡量预测值和真实值之间差距，并用于更新权重矩阵。
* 优化器，优化器将使用计算的损失值和反向传播算法更新权重和偏置项参数。

**2.2 加载数据**

首先我们需要加载数据，加载的数据用来训练和测试神经网络。在 Ahmet Taspinar 的博客中，他用的是 MNIST 和 CIFAR-10 数据集。其中 MNIST 数据集包含 6 万张手写数字图片，每一张图片的大小都是 28 x 28 x 1（灰度图）。而 CIFAR-10 数据集包含 6 万张彩色（3 通道）图片，每张图片的大小为 32 x 32 x 3，该数据集有 10 种不同的物体（飞机、摩托车、鸟、猫、狗、青蛙、马、羊和卡车）。



首先，让我们定义一些函数，它们能帮助我们加载和预处理图像数据。

def randomize(dataset, labels):

permutation = np.random.permutation(labels.shape[0])

shuffled\_dataset = dataset[permutation, :, :]

shuffled\_labels = labels[permutation]

return shuffled\_dataset, shuffled\_labels

def one\_hot\_encode(np\_array):

return (np.arange(10) == np\_array[:,None]).astype(np.float32)

def reformat\_data(dataset, labels, image\_width, image\_height, image\_depth):

np\_dataset\_ = np.array([np.array(image\_data).reshape(image\_width, image\_height, image\_depth) for image\_data in dataset])

np\_labels\_ = one\_hot\_encode(np.array(labels, dtype=np.float32))

np\_dataset, np\_labels = randomize(np\_dataset\_, np\_labels\_)

return np\_dataset, np\_labels

def flatten\_tf\_array(array):

shape = array.get\_shape().as\_list()

return tf.reshape(array, [shape[0], shape[1] \* shape[2] \* shape[3]])

def accuracy(predictions, labels):

return (100.0 \* np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1)) / predictions.shape[0])def randomize(dataset, labels):

permutation = np.random.permutation(labels.shape[0])

shuffled\_dataset = dataset[permutation, :, :]

shuffled\_labels = labels[permutation]

return shuffled\_dataset, shuffled\_labels

def one\_hot\_encode(np\_array):

return (np.arange(10) == np\_array[:,None]).astype(np.float32)

def reformat\_data(dataset, labels, image\_width, image\_height, image\_depth):

np\_dataset\_ = np.array([np.array(image\_data).reshape(image\_width, image\_height, image\_depth) for image\_data in dataset])

np\_labels\_ = one\_hot\_encode(np.array(labels, dtype=np.float32))

np\_dataset, np\_labels = randomize(np\_dataset\_, np\_labels\_)

return np\_dataset, np\_labels

def flatten\_tf\_array(array):

shape = array.get\_shape().as\_list()

return tf.reshape(array, [shape[0], shape[1] \* shape[2] \* shape[3]])

def accuracy(predictions, labels):

return (100.0 \* np.sum(np.argmax(predictions, 1) == np.argmax(labels, 1)) / predictions.shape[0])

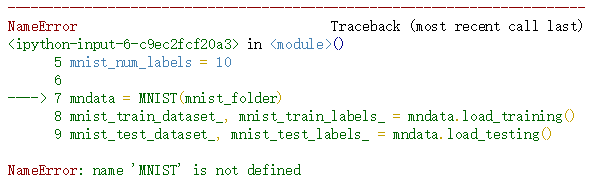
图像的标签使用 one-hot 编码，并且将数据加载到随机数组中。在定义这些函数后，我们可以加载数据：



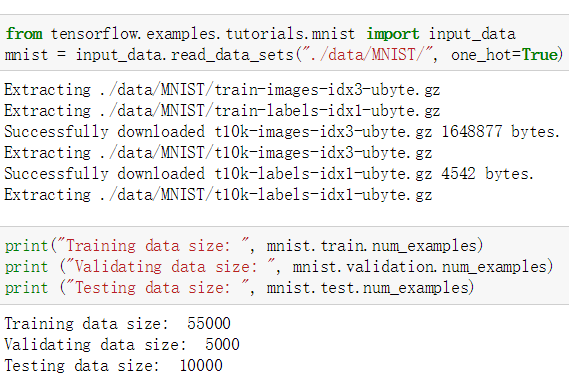
我们能从 Yann LeCun 的网站下载 MNIST 数据集，下载并解压后就能使用 python-mnist 工具加载该数据集。

* MNIST 数据集：http://yann.lecun.com/exdb/mnist/
* python-mnist 工具：https://github.com/sorki/python-mnist
* CIFAR-10 数据集：https://www.cs.toronto.edu/~kriz/cifar.html

在 Ahmet Taspinar 提供的上述代码中，我们运行会出错，因为「MNIST」并没有定义，而我们机器之心在安装完 python-mnist，并加上「from mnist import MNIST」语句后，仍然不能导入。所以我们可以修改以上代码，使用 TensorFlow 官方教程中自带的 MNIST 加载工具加载 MNIST。



如下所示，我们可以使用这种方法成功地导入 MNIST 数据集：



我们需要再次导入 CIFAR-10 数据集，这一段代码也会出错，原因是有变量没有定义。下面代码将导入数据集：

cifar10\_folder = './data/cifar10/'  
train\_datasets = ['data\_batch\_1', 'data\_batch\_2', 'data\_batch\_3', 'data\_batch\_4', 'data\_batch\_5', ]  
test\_dataset = ['test\_batch']  
  
c10\_image\_height = 32  
c10\_image\_width = 32  
c10\_image\_depth = 3  
c10\_num\_labels = 10  
c10\_image\_size = 32 #Ahmet Taspinar的代码缺少了这一语句  
  
  
with open(cifar10\_folder + test\_dataset[0], 'rb') as f0:  
    c10\_test\_dict = pickle.load(f0, encoding='bytes')  
  
c10\_test\_dataset, c10\_test\_labels = c10\_test\_dict[b'data'], c10\_test\_dict[b'labels']  
test\_dataset\_cifar10, test\_labels\_cifar10 = reformat\_data(c10\_test\_dataset, c10\_test\_labels, c10\_image\_size, c10\_image\_size, c10\_image\_depth)  
  
c10\_train\_dataset, c10\_train\_labels = [], []  
  
for train\_dataset in train\_datasets:  
    with open(cifar10\_folder + train\_dataset, 'rb') as f0:  
        c10\_train\_dict = pickle.load(f0, encoding='bytes')  
        c10\_train\_dataset\_, c10\_train\_labels\_ = c10\_train\_dict[b'data'], c10\_train\_dict[b'labels']  
  
        c10\_train\_dataset.append(c10\_train\_dataset\_)  
        c10\_train\_labels += c10\_train\_labels\_  
  
c10\_train\_dataset = np.concatenate(c10\_train\_dataset, axis=0)  
train\_dataset\_cifar10, train\_labels\_cifar10 = reformat\_data(c10\_train\_dataset, c10\_train\_labels, c10\_image\_size, c10\_image\_size, c10\_image\_depth)  
del c10\_train\_dataset

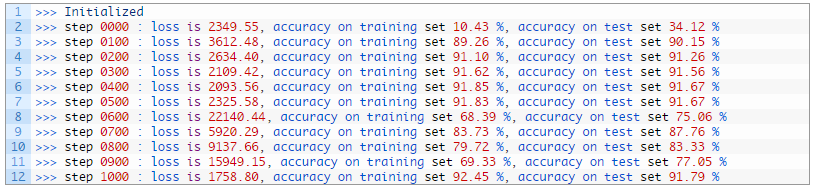
del c10\_train\_labels

print("训练集包含以下标签: {}".format(np.unique(c10\_train\_dict[b'labels'])))  
print('训练集维度', train\_dataset\_cifar10.shape, train\_labels\_cifar10.shape)  
print('测试集维度', test\_dataset\_cifar10.shape, test\_labels\_cifar10.shape)

在试验中，我们需要注意放置数据集的地址。MNIST 可以自动检测指定的目录下是否有数据集，如果没有就自动下载数据集至该目录下。在上面的两段代码中，「./data/MNIST/」就代表着我们放置数据集的地址，它表示在 Python 根目录下「data」文件夹下的「MNIST」文件夹内。CIFAR-10 同样也是这样，只不过它不会自动下载数据集。

**2.3 创建简单的多层全连接神经网络**

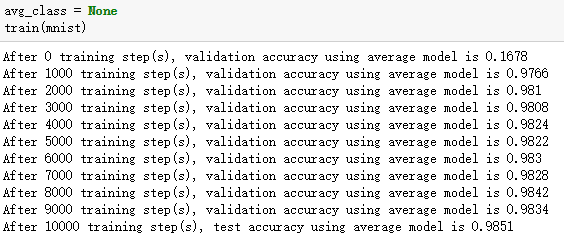
Ahmet Taspinar 后面创建了一个单隐藏层全连接网络，不过我们还是报错了。他在博客中给出了以下训练准确度，我们看到该模型在 MNIST 数据集效果并不是很好。所以我们另外使用一个全连接神经网络来实现这一过程。



下面我们实现的神经网络共有三层，输入层有 784 个神经元，隐藏层与输出层分别有 500 和 10 个神经元。这所以这样设计是因为 MNIST 的像素为 28×28=784，所以每一个输入神经元对应于一个灰度像素点。机器之心执行该模型得到的效果非常好，该模型在批量大小为 100，并使用学习率衰减的情况下迭代 10000 步能得到 98.34% 的测试集准确度，以下是该模型代码：

import tensorflow as tf  
from tensorflow.examples.tutorials.mnist import input\_data  
   
  
  
#加载MNIST数据集  
mnist = input\_data.read\_data\_sets("./data/MNIST/", one\_hot=True)  
  
  
INPUT\_NODE = 784       
OUTPUT\_NODE = 10       
LAYER1\_NODE = 500           
  
BATCH\_SIZE = 100   
   
         
  
# 模型相关的参数  
LEARNING\_RATE\_BASE = 0.8        
LEARNING\_RATE\_DECAY = 0.99      
REGULARAZTION\_RATE = 0.0001     
TRAINING\_STEPS = 10000          
MOVING\_AVERAGE\_DECAY = 0.99   
   
  
  
def inference(input\_tensor, avg\_class, weights1, biases1, weights2, biases2):  
    # 使用滑动平均类  
    if avg\_class == None:  
        layer1 = tf.nn.relu(tf.matmul(input\_tensor, weights1) + biases1)  
        return tf.matmul(layer1, weights2) + biases2  
  
    else:  
  
        layer1 = tf.nn.relu(tf.matmul(input\_tensor, avg\_class.average(weights1)) + avg\_class.average(biases1))  
        return tf.matmul(layer1, avg\_class.average(weights2)) + avg\_class.average(biases2)    
  
  
def train(mnist):  
    x = tf.placeholder(tf.float32, [None, INPUT\_NODE], name='x-input')  
    y\_ = tf.placeholder(tf.float32, [None, OUTPUT\_NODE], name='y-input')  
  
    # 生成隐藏层的参数。  
    weights1 = tf.Variable(tf.truncated\_normal([INPUT\_NODE, LAYER1\_NODE], stddev=0.1))  
    biases1 = tf.Variable(tf.constant(0.1, shape=[LAYER1\_NODE]))  
  
    # 生成输出层的参数。  
    weights2 = tf.Variable(tf.truncated\_normal([LAYER1\_NODE, OUTPUT\_NODE], stddev=0.1))  
    biases2 = tf.Variable(tf.constant(0.1, shape=[OUTPUT\_NODE]))  
  
  
    # 计算不含滑动平均类的前向传播结果  
    y = inference(x, None, weights1, biases1, weights2, biases2)  
  
  
    # 定义训练轮数及相关的滑动平均类   
    global\_step = tf.Variable(0, trainable=False)  
    variable\_averages = tf.train.ExponentialMovingAverage(MOVING\_AVERAGE\_DECAY, global\_step)  
    variables\_averages\_op = variable\_averages.apply(tf.trainable\_variables())  
    average\_y = inference(x, variable\_averages, weights1, biases1, weights2, biases2)  
  
    # 计算交叉熵及其平均值  
    cross\_entropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(logits=y, labels=tf.argmax(y\_, 1))  
    cross\_entropy\_mean = tf.reduce\_mean(cross\_entropy)  
  
    # 定义交叉熵损失函数加上正则项为模型损失函数  
    regularizer = tf.contrib.layers.l2\_regularizer(REGULARAZTION\_RATE)  
    regularaztion = regularizer(weights1) + regularizer(weights2)  
    loss = cross\_entropy\_mean + regularaztion  
  
    # 设置指数衰减的学习率。  
    learning\_rate = tf.train.exponential\_decay(  
        LEARNING\_RATE\_BASE,  
        global\_step,  
        mnist.train.num\_examples / BATCH\_SIZE,  
        LEARNING\_RATE\_DECAY,  
        staircase=True)  
  
    # 随机梯度下降优化器优化损失函数  
    train\_step = tf.train.GradientDescentOptimizer(learning\_rate).minimize(loss, global\_step=global\_step)  
  
    # 反向传播更新参数和更新每一个参数的滑动平均值  
    with tf.control\_dependencies([train\_step, variables\_averages\_op]):  
        train\_op = tf.no\_op(name='train')  
  
    # 计算准确度  
    correct\_prediction = tf.equal(tf.argmax(average\_y, 1), tf.argmax(y\_, 1))  
    accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))  
  
    # 初始化会话并开始训练过程。  
    with tf.Session() as sess:  
        tf.global\_variables\_initializer().run()  
        validate\_feed = {x: mnist.validation.images, y\_: mnist.validation.labels}  
        test\_feed = {x: mnist.test.images, y\_: mnist.test.labels}   
  
        # 循环地训练神经网络。  
        for i in range(TRAINING\_STEPS):  
            if i % 1000 == 0:  
                validate\_acc = sess.run(accuracy, feed\_dict=validate\_feed)  
                print("After %d training step(s), validation accuracy using average model is %g " % (i, validate\_acc))  
  
            xs,ys=mnist.train.next\_batch(BATCH\_SIZE)  
            sess.run(train\_op,feed\_dict={x:xs,y\_:ys})  
  
        test\_acc=sess.run(accuracy,feed\_dict=test\_feed)  
        print(("After %d training step(s), test accuracy using average model is %g" %(TRAINING\_STEPS, test\_acc)))

该模型运行的结果如下：



在上面定义的整个计算图中，我们先加载数据并定义权重矩阵和模型，然后在计算损失值并传递给优化器来优化权重。模型在迭代次数设定之内会一直循环地计算损失函数的梯度以更新权重。在上面的全连接神经网络中，我们使用梯度下降优化器来优化权重。然而，TensorFlow 中还有很多优化器，最常用的是 GradientDescentOptimizer、AdamOptimizer 和 AdaGradOptimizer。

下面我们就需要构建卷积神经网络了，不过在使用 TensorFlow 构建卷积网络之前，我们需要了解一下 TensorFlow 中的函数。

TensorFlow 包含很多操作和函数，很多我们需要花费大量精力完成的过程可以直接调用已封装的函数，比如说「logits = tf.matmul(tf\_train\_dataset, weights) + biases」可以由函数「logits = tf.nn.xw\_plus\_b(train\_dataset, weights, biases)」代替。

还有很多函数可以让构建不同层级的神经网络变得十分简单。例如 conv\_2d() 和 fully\_connected() 函数分别构建了卷积层和全连接层。通过这些函数，层级的数量、滤波器的大小/深度、激活函数的类型等都可以明确地作为一个参数。权重矩阵和偏置向量能自动创建，附加激活函数和 dropout 正则化层同样也能轻松构建。

如下所示为定义卷积层网络的代码：

import tensorflow as tf  
   
  
  
w1 = tf.Variable(tf.truncated\_normal([filter\_size, filter\_size, image\_depth, filter\_depth], stddev=0.1))  
b1 = tf.Variable(tf.zeros([filter\_depth]))  
  
  
layer1\_conv = tf.nn.conv2d(data, w1, [1, 1, 1, 1], padding='SAME')  
layer1\_relu = tf.nn.relu(layer1\_conv + b1)  
layer1\_pool = tf.nn.max\_pool(layer1\_pool, [1, 2, 2, 1], [1, 2, 2, 1], padding='SAME')

它们可以使用简单的函数来替代上面的定义：

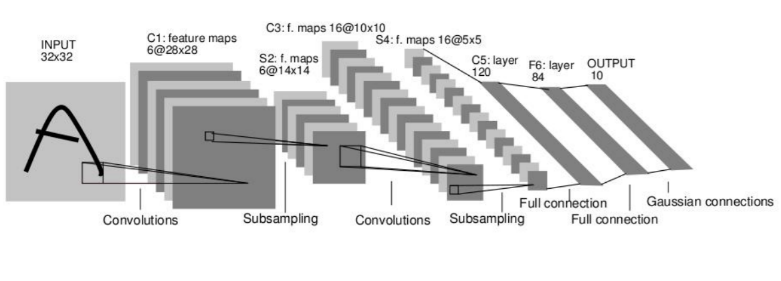
from tflearn.layers.conv import conv\_2d, max\_pool\_2d  
   
  
  
layer1\_conv = conv\_2d(data, filter\_depth, filter\_size, activation='relu')  
layer1\_pool = max\_pool\_2d(layer1\_conv\_relu, 2, strides=2)

正如我们前面所说的，我们并不需要定义权重、偏置和激活函数，特别是在定义多层神经网络的时候，这一点让我们的代码可以看起来十分整洁。

**2.4 创建 LeNet5 卷积网络**

LeNet5 卷积网络架构最早是 Yann LeCun 提出来的，它是早期的一种卷积神经网络，并且可以用来识别手写数字。虽然它在 MNIST 数据集上执行地非常好，但在其它高分辨率和大数据集上性能有所降低。对于这些大数据集，像 AlexNet、VGGNet 或 ResNet 那样的深度卷积网络才执行地十分优秀。

因为 LeNet5 只由 5 层网络，所以它是学习如何构建卷积网络的最佳起点。LeNet5 的架构如下：



LeNet5 包含 5 层网络：

* 第一层：卷积层，该卷积层使用 Sigmoid 激活函数，并且在后面带有平均池化层。
* 第二层：卷积层，该卷积层使用 Sigmoid 激活函数，并且在后面带有平均池化层。
* 第三层：全连接层（使用 Sigmoid 激活函数）。
* 第四层：全连接层（使用 Sigmoid 激活函数）。
* 第五层：输出层。

上面的 LeNet5 架构意味着我们需要构建 5 个权重和偏置项矩阵，我们模型的主体大概需要 12 行代码完成（5 个神经网络层级、2 个池化层、4 个激活函数还有 1 个 flatten 层）。因为代码比较多，所以我们最好在计算图之外就定义好独立的函数：

LENET5\_BATCH\_SIZE = 32

LENET5\_PATCH\_SIZE = 5

LENET5\_PATCH\_DEPTH\_1 = 6

LENET5\_PATCH\_DEPTH\_2 = 16

LENET5\_NUM\_HIDDEN\_1 = 120

LENET5\_NUM\_HIDDEN\_2 = 84

def variables\_lenet5(patch\_size = LENET5\_PATCH\_SIZE, patch\_depth1 = LENET5\_PATCH\_DEPTH\_1,

patch\_depth2 = LENET5\_PATCH\_DEPTH\_2,

num\_hidden1 = LENET5\_NUM\_HIDDEN\_1, num\_hidden2 = LENET5\_NUM\_HIDDEN\_2,

image\_depth = 1, num\_labels = 10):

w1 = tf.Variable(tf.truncated\_normal([patch\_size, patch\_size, image\_depth, patch\_depth1], stddev=0.1))

b1 = tf.Variable(tf.zeros([patch\_depth1]))

w2 = tf.Variable(tf.truncated\_normal([patch\_size, patch\_size, patch\_depth1, patch\_depth2], stddev=0.1))

b2 = tf.Variable(tf.constant(1.0, shape=[patch\_depth2]))

w3 = tf.Variable(tf.truncated\_normal([5\*5\*patch\_depth2, num\_hidden1], stddev=0.1))

b3 = tf.Variable(tf.constant(1.0, shape = [num\_hidden1]))

w4 = tf.Variable(tf.truncated\_normal([num\_hidden1, num\_hidden2], stddev=0.1))

b4 = tf.Variable(tf.constant(1.0, shape = [num\_hidden2]))

w5 = tf.Variable(tf.truncated\_normal([num\_hidden2, num\_labels], stddev=0.1))

b5 = tf.Variable(tf.constant(1.0, shape = [num\_labels]))

variables = {

'w1': w1, 'w2': w2, 'w3': w3, 'w4': w4, 'w5': w5,

'b1': b1, 'b2': b2, 'b3': b3, 'b4': b4, 'b5': b5

}

return variables

def model\_lenet5(data, variables):

layer1\_conv = tf.nn.conv2d(data, variables['w1'], [1, 1, 1, 1], padding='SAME')

layer1\_actv = tf.sigmoid(layer1\_conv + variables['b1'])

layer1\_pool = tf.nn.avg\_pool(layer1\_actv, [1, 2, 2, 1], [1, 2, 2, 1], padding='SAME')

layer2\_conv = tf.nn.conv2d(layer1\_pool, variables['w2'], [1, 1, 1, 1], padding='VALID')

layer2\_actv = tf.sigmoid(layer2\_conv + variables['b2'])

layer2\_pool = tf.nn.avg\_pool(layer2\_actv, [1, 2, 2, 1], [1, 2, 2, 1], padding='SAME')

flat\_layer = flatten\_tf\_array(layer2\_pool)

layer3\_fccd = tf.matmul(flat\_layer, variables['w3']) + variables['b3']

layer3\_actv = tf.nn.sigmoid(layer3\_fccd)

layer4\_fccd = tf.matmul(layer3\_actv, variables['w4']) + variables['b4']

layer4\_actv = tf.nn.sigmoid(layer4\_fccd)

logits = tf.matmul(layer4\_actv, variables['w5']) + variables['b5']

return logits

通过上面独立定义的变量和模型，我们可以一点点调整数据流图而不像前面的全连接网络那样。

#parameters determining the model size

image\_size = mnist\_image\_size

num\_labels = mnist\_num\_labels

#the datasets

train\_dataset = mnist\_train\_dataset

train\_labels = mnist\_train\_labels

test\_dataset = mnist\_test\_dataset

test\_labels = mnist\_test\_labels

#number of iterations and learning rate

num\_steps = 10001

display\_step = 1000

learning\_rate = 0.001

graph = tf.Graph()

with graph.as\_default():

#1) First we put the input data in a tensorflow friendly form.

tf\_train\_dataset = tf.placeholder(tf.float32, shape=(batch\_size, image\_width, image\_height, image\_depth))

tf\_train\_labels = tf.placeholder(tf.float32, shape = (batch\_size, num\_labels))

tf\_test\_dataset = tf.constant(test\_dataset, tf.float32)

#2) Then, the weight matrices and bias vectors are initialized

<strong>variables = variables\_lenet5(image\_depth = image\_depth, num\_labels = num\_labels)</strong>

#3. The model used to calculate the logits (predicted labels)

<strong>model = model\_lenet5</strong>

<strong>logits = model(tf\_train\_dataset, variables)</strong>

#4. then we compute the softmax cross entropy between the logits and the (actual) labels

loss = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits=logits, labels=tf\_train\_labels))

#5. The optimizer is used to calculate the gradients of the loss function

optimizer = tf.train.GradientDescentOptimizer(learning\_rate).minimize(loss)

# Predictions for the training, validation, and test data.

train\_prediction = tf.nn.softmax(logits)

test\_prediction = tf.nn.softmax(model(tf\_test\_dataset, variables))

with tf.Session(graph=graph) as session:

tf.global\_variables\_initializer().run()

print('Initialized with learning\_rate', learning\_rate)

for step in range(num\_steps):

#Since we are using stochastic gradient descent, we are selecting small batches from the training dataset,

#and training the convolutional neural network each time with a batch.

offset = (step \* batch\_size) % (train\_labels.shape[0] - batch\_size)

batch\_data = train\_dataset[offset:(offset + batch\_size), :, :, :]

batch\_labels = train\_labels[offset:(offset + batch\_size), :]

feed\_dict = {tf\_train\_dataset : batch\_data, tf\_train\_labels : batch\_labels}

\_, l, predictions = session.run([optimizer, loss, train\_prediction], feed\_dict=feed\_dict)

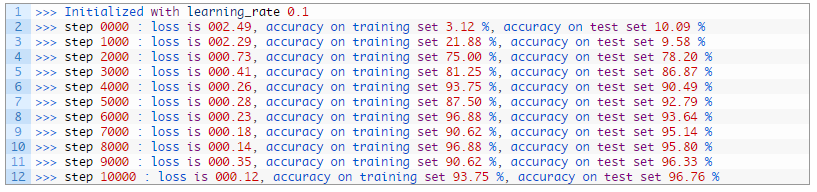
if step % display\_step == 0:

train\_accuracy = accuracy(predictions, batch\_labels)

test\_accuracy = accuracy(test\_prediction.eval(), test\_labels)

message = "step {:04d} : loss is {:06.2f}, accuracy on training set {:02.2f} %, accuracy on test set {:02.2f} %".format(step, l, train\_accuracy, test\_accuracy)

print(message)



我们看到 Ahmet Taspinar 构建的 LeNet5 网络要比他所训练的全连接网络在 MNIST 数据集上有更好的性能。但是在我们所训练的全连接神经网络中，因为使用了 ReLU、学习率指数衰减、滑动平均类和正则化等机制，我们的准确度达到了 98% 以上。

**2.5 超参数如何影响一层网络的输出尺寸**

一般来说，确实是层级越多神经网络的性能就越好。我们可以添加更多的层级、更改激活函数和池化层、改变学习率并查看每一步对性能的影响。因为层级 i 的输出是层级 i+1 的输入，所以我们需要知道第 i 层神经网络的超参数如何影响其输出尺寸。

为了理解这一点我们需要讨论一下 conv2d() 函数。

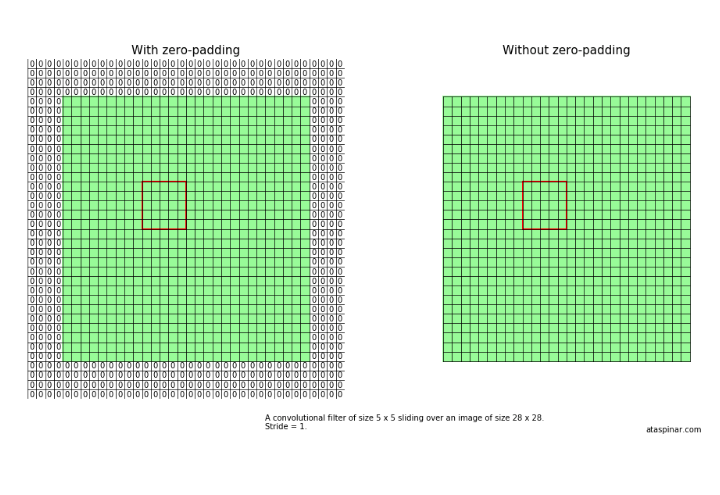
该函数有四个参数：

* 输入图像，即一个四维张量 [batch size, image\_width, image\_height, image\_depth]
* 权重矩阵，即一个四维张量 [filter\_size, filter\_size, image\_depth, filter\_depth]
* 每一个维度的步幅数
* Padding (= 'SAME' / 'VALID')

这四个参数决定了输出图像的尺寸。前面两个参数都是四维张量，其包括了批量输入图像的信息和卷积滤波器的权值。

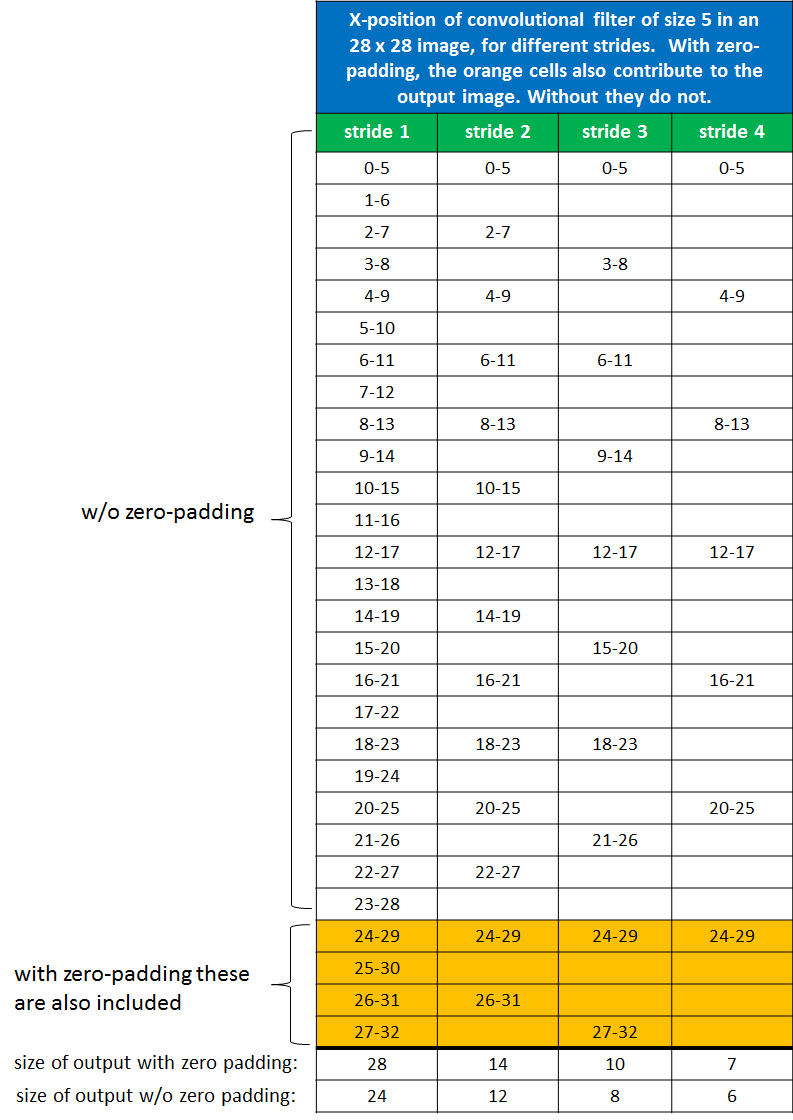
第三个参数为卷积的步幅（stride），即卷积滤波器在 4 个维度中的每一次移动的距离。四个中间的第一个维度代表着图像的批量数，这个维度肯定每次只能移动一张图片。最后一个维度为图片深度（即色彩通道数，1 代表灰度图片，而 3 代表 RGB 图片），因为我们通常并不想跳过任何一个通道，所以这一个值也通常为 1。第二个和第三个维度代表 X 和 Y 方向（图片宽度和高度）的步幅。如果我们希望能应用步幅参数，我们需要设定每个维度的移动步幅。例如设定步幅为 1，那么步幅参数就需要设定为 [1, 1, 1, 1]，如果我们希望在图像上移动的步幅设定为 2，步幅参数为 [1, 2, 2, 1]。

最后一个参数表明 TensorFlow 是否需要使用 0 来填补图像周边，这样以确保图像输出尺寸在步幅参数设定为 1 的情况下保持不变。通过设置 padding = 'SAME'，图像会只使用 0 来填补周边（输出尺寸不变），而 padding = 'VALID'则不会使用 0。在下图中，我们将看到两个使用卷积滤波器在图像上扫描的案例，其中滤波器的大小为 5 x 5、图像的大小为 28 x 28。左边的 Padding 参数设置为'SAME'，并且最后四行/列的信息也会包含在输出图像中。而右边 padding 设置为 'VALID'，最后四行/列是不包括在输出图像内的。



没有 padding 的图片，最后四个像素点是无法包含在内的，因为卷积滤波器已经移动到了图片的边缘。这就意味着输入 28 x 28 尺寸的图片，输出尺寸只有 24 x 24。如果 padding = 'SAME'，那么输出尺寸就是 28 x 28。

如果我们输入图片尺寸是 28 x 28、滤波器尺寸为 5 x 5，步幅分别设置为 1 到 4，那么就能得到下表



对于任意给定的步幅 S、滤波器尺寸 K、图像尺寸 W、padding 尺寸 P，输出的图像尺寸可以总结上表的规则如下：

https://image.jiqizhixin.com/uploads/wangeditor/c50c4f39-faab-4616-a653-97a881aeb905/50215image%20(13).png

**2.6 调整 LeNet5 架构**

LeNet5 架构在原论文中使用的是 Sigmoid 激活函数和平均池化。然而如今神经网络使用 ReLU 激活函数更为常见。所以我们可以修改一下 LeNet5 架构，并看看是否能获得性能上的提升，我们可以称这种修改的架构为类 LeNet5 架构。

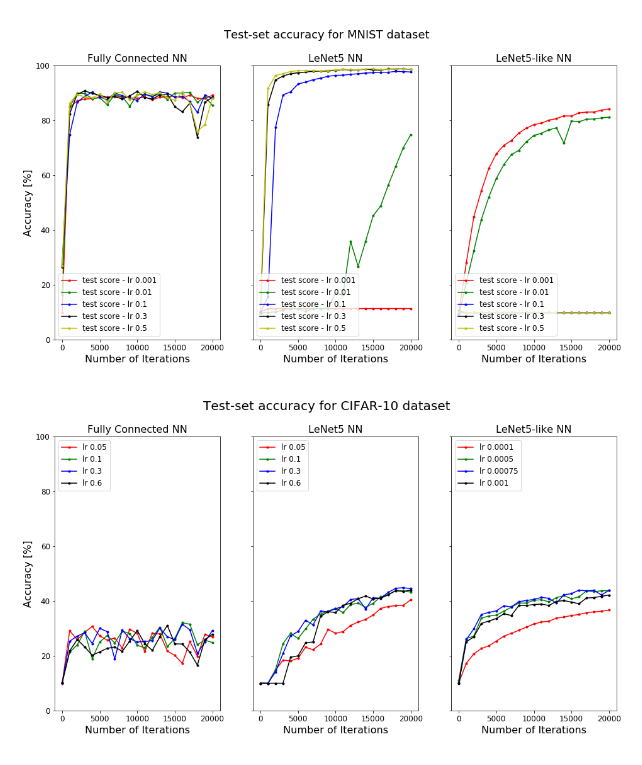


最大的不同是我们使用 ReLU 激活函数代替 Sigmoid 激活函数。除了激活函数意外，我们还修改了优化器，因为我们可以看到不同优化器对识别准确度的影响。在这里，机器之心在 CIFAR-10 上使用该修正的 LeNet 进行了训练，详细代码如下。机器之心训练的准确度并不高，可能是学习率、批量数或者其他设置有些问题，也可能是 LeNet 对于三通道的图太简单了。该运行结果展现在机器之心该项目的 Github 中，感兴趣的读者可以进一步修正该模型以期望达到更好的效果。

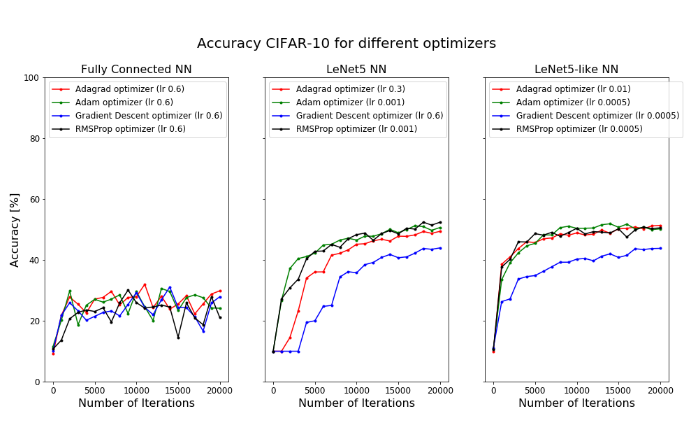
LENET5\_LIKE\_BATCH\_SIZE = 32  
LENET5\_LIKE\_FILTER\_SIZE = 5  
LENET5\_LIKE\_FILTER\_DEPTH = 16  
LENET5\_LIKE\_NUM\_HIDDEN = 120  
  
  
def variables\_lenet5\_like(filter\_size = LENET5\_LIKE\_FILTER\_SIZE,   
                          filter\_depth = LENET5\_LIKE\_FILTER\_DEPTH,   
                          num\_hidden = LENET5\_LIKE\_NUM\_HIDDEN,  
                          image\_width = 32, image\_height = 32, image\_depth = 3, num\_labels = 10):  
  
    w1 = tf.Variable(tf.truncated\_normal([filter\_size, filter\_size, image\_depth, filter\_depth], stddev=0.1))  
    b1 = tf.Variable(tf.zeros([filter\_depth]))  
  
    w2 = tf.Variable(tf.truncated\_normal([filter\_size, filter\_size, filter\_depth, filter\_depth], stddev=0.1))  
    b2 = tf.Variable(tf.constant(1.0, shape=[filter\_depth]))  
  
    w3 = tf.Variable(tf.truncated\_normal([(image\_width // 4)\*(image\_height // 4)\*filter\_depth , num\_hidden], stddev=0.1))  
    b3 = tf.Variable(tf.constant(1.0, shape = [num\_hidden]))  
  
    w4 = tf.Variable(tf.truncated\_normal([num\_hidden, num\_hidden], stddev=0.1))  
    b4 = tf.Variable(tf.constant(1.0, shape = [num\_hidden]))  
  
    w5 = tf.Variable(tf.truncated\_normal([num\_hidden, num\_labels], stddev=0.1))  
    b5 = tf.Variable(tf.constant(1.0, shape = [num\_labels]))  
    variables = {  
        'w1': w1, 'w2': w2, 'w3': w3, 'w4': w4, 'w5': w5,  
        'b1': b1, 'b2': b2, 'b3': b3, 'b4': b4, 'b5': b5  
    }  
    return variables  
   
  
  
def model\_lenet5\_like(data, variables):  
    layer1\_conv = tf.nn.conv2d(data, variables['w1'], [1, 1, 1, 1], padding='SAME')  
    layer1\_actv = tf.nn.relu(layer1\_conv + variables['b1'])  
    layer1\_pool = tf.nn.avg\_pool(layer1\_actv, [1, 2, 2, 1], [1, 2, 2, 1], padding='SAME')  
  
    layer2\_conv = tf.nn.conv2d(layer1\_pool, variables['w2'], [1, 1, 1, 1], padding='SAME')  
    layer2\_actv = tf.nn.relu(layer2\_conv + variables['b2'])  
    layer2\_pool = tf.nn.avg\_pool(layer2\_actv, [1, 2, 2, 1], [1, 2, 2, 1], padding='SAME')  
  
    flat\_layer = flatten\_tf\_array(layer2\_pool)  
    layer3\_fccd = tf.matmul(flat\_layer, variables['w3']) + variables['b3']  
    layer3\_actv = tf.nn.relu(layer3\_fccd)  
    layer3\_drop = tf.nn.dropout(layer3\_actv, 0.5)  
  
    layer4\_fccd = tf.matmul(layer3\_actv, variables['w4']) + variables['b4']  
    layer4\_actv = tf.nn.relu(layer4\_fccd)  
    layer4\_drop = tf.nn.dropout(layer4\_actv, 0.5)  
  
    logits = tf.matmul(layer4\_actv, variables['w5']) + variables['b5']  
    return logits  
  
  
  
num\_steps = 10001  
display\_step = 1000  
learning\_rate = 0.001  
batch\_size = 16  
  
  
#定义数据的基本信息，传入变量  
image\_width = 32  
image\_height = 32  
image\_depth = 3  
num\_labels = 10  
  
  
test\_dataset = test\_dataset\_cifar10  
test\_labels = test\_labels\_cifar10  
train\_dataset = train\_dataset\_cifar10  
train\_labels = train\_labels\_cifar10  
   
  
  
  
graph = tf.Graph()  
  
with graph.as\_default():  
  
    #1 首先使用占位符定义数据变量的维度  
    tf\_train\_dataset = tf.placeholder(tf.float32, shape=(batch\_size, image\_width, image\_height, image\_depth))  
    tf\_train\_labels = tf.placeholder(tf.float32, shape = (batch\_size, num\_labels))  
    tf\_test\_dataset = tf.constant(test\_dataset, tf.float32)  
  
  
    #2 然后初始化权重矩阵和偏置向量  
    variables = variables\_lenet5\_like(image\_width = image\_width, image\_height=image\_height, image\_depth = image\_depth, num\_labels = num\_labels)  
  
  
  
    #3 使用模型计算分类  
    logits = model\_lenet5\_like(tf\_train\_dataset, variables)  
  
  
    #4 使用带softmax的交叉熵函数计算预测标签和真实标签之间的损失函数  
    loss = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits=logits, labels=tf\_train\_labels))  
  
  
    #5  采用Adam优化算法优化上一步定义的损失函数，给定学习率  
    optimizer = tf.train.AdamOptimizer(learning\_rate=learning\_rate).minimize(loss)  
  
  
      
      
# 执行预测推断  
    train\_prediction = tf.nn.softmax(logits)  
    test\_prediction = tf.nn.softmax(model\_lenet5\_like(tf\_test\_dataset, variables))  
  
  
with tf.Session(graph=graph) as session:  
  
    #初始化全部变量  
    tf.global\_variables\_initializer().run()  
    print('Initialized with learning\_rate', learning\_rate)  
    for step in range(num\_steps):  
        offset = (step \* batch\_size) % (train\_labels.shape[0] - batch\_size)  
        batch\_data = train\_dataset[offset:(offset + batch\_size), :, :, :]  
        batch\_labels = train\_labels[offset:(offset + batch\_size), :]  
  
        #在每一次批量中，获取当前的训练数据，并传入feed\_dict以馈送到占位符中  
        feed\_dict = {tf\_train\_dataset : batch\_data, tf\_train\_labels : batch\_labels}  
        \_, l, predictions = session.run([optimizer, loss, train\_prediction], feed\_dict=feed\_dict)  
        train\_accuracy = accuracy(predictions, batch\_labels)  
  
        if step % display\_step == 0:  
            test\_accuracy = accuracy(test\_prediction.eval(), test\_labels)  
            message = "step {:04d} : loss is {:06.2f}, accuracy on training set {:02.2f} %, accuracy on test set {:02.2f} %".format(step, l, train\_accuracy, test\_accuracy)  
            print(message)

**2.7 学习率和优化器的影响**

我们可以在下图看到这些 CNN 在 MNIST 和 CIFAR-10 数据集上的性能。



上图展示了模型在两个测试集上的准确度和迭代次数，其代表的模型从左至右分别为全连接神经网络、LeNet5 和 改进后的 LeNet5。不过由于 MNIST 太简单，全连接网络也能做得挺好。不过在 CIFAR-10 数据集中，全连接网络的性能明显下降了不少。



上图展示了三种神经网络在 CIFAR-10 数据集上使用不同的优化器而得出的性能。可能 L2 正则化和指数衰减学习率能进一步提高模型的性能，不过要获得更大的提升，我们需要使用深度神经网络。

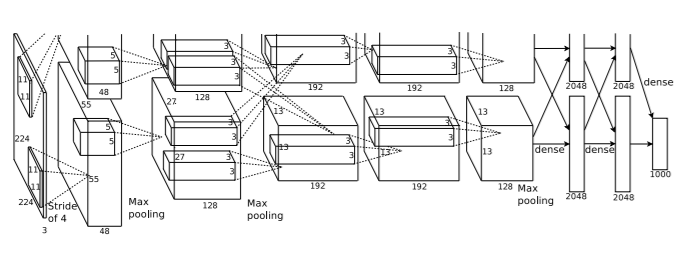
**TensorFlow 中的深度神经网络**

LeNet5 由两个卷积层加上三个全连接层组成，因此它是一种浅层神经网络。下面我们将了解其它卷积神经网络，它们的层级更多，所以可以称为深度神经网络。下面介绍的深度卷积神经网络我们并没有根据 Ahmet Taspinar 提供的代码进行实践，因为我们暂时安装的是 TensorFlow 的 CPU 版，而使用 CPU 训练前面的 LeNet 就已经十分吃力了，所以我们暂时没有实现这几个深度 CNN。我们将会在后面实现它们，并将修改的代码上传到机器之心的 Github 中。

卷积神经网络最出名的就是 2012 年所提出的 AlexNet、2013 年的 7 层 ZF-Net 和 2014 年提出的 16 层 VGGNet。到了 2015 年，谷歌通过 Inception 模块开发出 22 层的卷积神经网络（GoogLeNet），而微软亚洲研究院创造出了 152 层的卷积神经网络：ResNet。下面，我们将学习如何使用 TensofFlow 构建 AlexNet 和 VGGNet16。

**3.1 AlexNet**

AlexNet 是由 Alex Krizhevsky 和 Geoffrey Hinton 等人提出来的，虽然相对于现在的卷积神经网络来说它的架构十分简单，但当时它是十分成功的一个模型。它赢得了当年的 ImageNet 挑战赛，并开启了深度学习和 AI 的变革。下面是 AlexNet 的基本架构：



AlexNet 包含 5 个卷积层（带有 ReLU 激活函数）、3 个最大池化层、3 个全连接层和两个 dropout 层。该神经网络的架构概览如下：

* 层级 0：规格为 224 x 224 x 3 的输入图片。
* 层级 1：带有 96 个滤波器（filter\_depth\_1 = 96）的卷积层，滤波器的尺寸为 11 x 11（filter\_size\_1 = 11）、步幅为 4。该层的神经网络使用 ReLU 激活函数，并且后面带有最大池化层和局部响应归一化层。
* 层级 2：带有 256 个滤波器（filter\_depth\_2 = 256）的卷积层，滤波器的尺寸为 5 x 5（filter\_size\_2 = 5）、步幅为 1。该层的神经网络使用 ReLU 激活函数，并且后面带有最大池化层和局部响应归一化层。
* 层级 3：带有 384 个滤波器（filter\_depth\_3 = 384）的卷积层，滤波器的尺寸为 3 x 3（filter\_size\_3 = 3）、步幅为 1。该层的神经网络使用 ReLU 激活函数。
* 层级 4 和层级 3 的结构是一样的。
* 层级 5：带有 256 个滤波器（filter\_depth\_4 = 256）的卷积层，滤波器的尺寸为 3 x 3（filter\_size\_4 = 3）、步幅为 1。该层的神经网络使用 ReLU 激活函数。
* 层级 6-8：这几个卷积层每一个后面跟着一个全连接层，每一层有 4096 个神经元。在原论文中，他们是为了 1000 个类别的分类，当我们这边并不需要这么多。

注意 AlexNet 或其他深度 CNN 并不能使用 MNIST 或者 CIFAR-10 数据集，因为这些图片的分辨率太小。正如我们所看到的，池化层（或者步幅为 2 的卷积层）减少了两倍的图像大小。AlexNet 有 3 个最大池化层和一个步幅为 4 的卷积层，这就意味着原图片会被缩小很多倍，而 MNIST 数据集的图像尺寸太小而不能进行着一系列操作。

因此，我们需要加载有更高像素图像的数据集，最好是和原论文一样采用 224 x 224 x 3。aka oxflower17 数据集可能是比较理想的数据集，它含有 17 种花的图片，并且像素正好是我们所需要的：

ox17\_image\_width = 224

ox17\_image\_height = 224

ox17\_image\_depth = 3

ox17\_num\_labels = 17

import tflearn.datasets.oxflower17 as oxflower17

train\_dataset\_, train\_labels\_ = oxflower17.load\_data(one\_hot=True)

train\_dataset\_ox17, train\_labels\_ox17 = train\_dataset\_[:1000,:,:,:], train\_labels\_[:1000,:]

test\_dataset\_ox17, test\_labels\_ox17 = train\_dataset\_[1000:,:,:,:], train\_labels\_[1000:,:]

print('Training set', train\_dataset\_ox17.shape, train\_labels\_ox17.shape)

print('Test set', test\_dataset\_ox17.shape, test\_labels\_ox17.shape)

下面，我们可以定义 AlexNet 中的权重矩阵和不同的层级。正如我们前面所看到的，我们需要定义很多权重矩阵和偏置向量，并且它们还需要和每一层的滤波器尺寸保持一致。

ALEX\_PATCH\_DEPTH\_1, ALEX\_PATCH\_DEPTH\_2, ALEX\_PATCH\_DEPTH\_3, ALEX\_PATCH\_DEPTH\_4 = 96, 256, 384, 256

ALEX\_PATCH\_SIZE\_1, ALEX\_PATCH\_SIZE\_2, ALEX\_PATCH\_SIZE\_3, ALEX\_PATCH\_SIZE\_4 = 11, 5, 3, 3

ALEX\_NUM\_HIDDEN\_1, ALEX\_NUM\_HIDDEN\_2 = 4096, 4096

def variables\_alexnet(patch\_size1 = ALEX\_PATCH\_SIZE\_1, patch\_size2 = ALEX\_PATCH\_SIZE\_2,

patch\_size3 = ALEX\_PATCH\_SIZE\_3, patch\_size4 = ALEX\_PATCH\_SIZE\_4,

patch\_depth1 = ALEX\_PATCH\_DEPTH\_1, patch\_depth2 = ALEX\_PATCH\_DEPTH\_2,

patch\_depth3 = ALEX\_PATCH\_DEPTH\_3, patch\_depth4 = ALEX\_PATCH\_DEPTH\_4,

num\_hidden1 = ALEX\_NUM\_HIDDEN\_1, num\_hidden2 = ALEX\_NUM\_HIDDEN\_2,

image\_width = 224, image\_height = 224, image\_depth = 3, num\_labels = 17):

w1 = tf.Variable(tf.truncated\_normal([patch\_size1, patch\_size1, image\_depth, patch\_depth1], stddev=0.1))

b1 = tf.Variable(tf.zeros([patch\_depth1]))

w2 = tf.Variable(tf.truncated\_normal([patch\_size2, patch\_size2, patch\_depth1, patch\_depth2], stddev=0.1))

b2 = tf.Variable(tf.constant(1.0, shape=[patch\_depth2]))

w3 = tf.Variable(tf.truncated\_normal([patch\_size3, patch\_size3, patch\_depth2, patch\_depth3], stddev=0.1))

b3 = tf.Variable(tf.zeros([patch\_depth3]))

w4 = tf.Variable(tf.truncated\_normal([patch\_size4, patch\_size4, patch\_depth3, patch\_depth3], stddev=0.1))

b4 = tf.Variable(tf.constant(1.0, shape=[patch\_depth3]))

w5 = tf.Variable(tf.truncated\_normal([patch\_size4, patch\_size4, patch\_depth3, patch\_depth3], stddev=0.1))

b5 = tf.Variable(tf.zeros([patch\_depth3]))

pool\_reductions = 3

conv\_reductions = 2

no\_reductions = pool\_reductions + conv\_reductions

w6 = tf.Variable(tf.truncated\_normal([(image\_width // 2\*\*no\_reductions)\*(image\_height // 2\*\*no\_reductions)\*patch\_depth3, num\_hidden1], stddev=0.1))

b6 = tf.Variable(tf.constant(1.0, shape = [num\_hidden1]))

w7 = tf.Variable(tf.truncated\_normal([num\_hidden1, num\_hidden2], stddev=0.1))

b7 = tf.Variable(tf.constant(1.0, shape = [num\_hidden2]))

w8 = tf.Variable(tf.truncated\_normal([num\_hidden2, num\_labels], stddev=0.1))

b8 = tf.Variable(tf.constant(1.0, shape = [num\_labels]))

variables = {

'w1': w1, 'w2': w2, 'w3': w3, 'w4': w4, 'w5': w5, 'w6': w6, 'w7': w7, 'w8': w8,

'b1': b1, 'b2': b2, 'b3': b3, 'b4': b4, 'b5': b5, 'b6': b6, 'b7': b7, 'b8': b8

}

return variables

def model\_alexnet(data, variables):

layer1\_conv = tf.nn.conv2d(data, variables['w1'], [1, 4, 4, 1], padding='SAME')

layer1\_relu = tf.nn.relu(layer1\_conv + variables['b1'])

layer1\_pool = tf.nn.max\_pool(layer1\_relu, [1, 3, 3, 1], [1, 2, 2, 1], padding='SAME')

layer1\_norm = tf.nn.local\_response\_normalization(layer1\_pool)

layer2\_conv = tf.nn.conv2d(layer1\_norm, variables['w2'], [1, 1, 1, 1], padding='SAME')

layer2\_relu = tf.nn.relu(layer2\_conv + variables['b2'])

layer2\_pool = tf.nn.max\_pool(layer2\_relu, [1, 3, 3, 1], [1, 2, 2, 1], padding='SAME')

layer2\_norm = tf.nn.local\_response\_normalization(layer2\_pool)

layer3\_conv = tf.nn.conv2d(layer2\_norm, variables['w3'], [1, 1, 1, 1], padding='SAME')

layer3\_relu = tf.nn.relu(layer3\_conv + variables['b3'])

layer4\_conv = tf.nn.conv2d(layer3\_relu, variables['w4'], [1, 1, 1, 1], padding='SAME')

layer4\_relu = tf.nn.relu(layer4\_conv + variables['b4'])

layer5\_conv = tf.nn.conv2d(layer4\_relu, variables['w5'], [1, 1, 1, 1], padding='SAME')

layer5\_relu = tf.nn.relu(layer5\_conv + variables['b5'])

layer5\_pool = tf.nn.max\_pool(layer4\_relu, [1, 3, 3, 1], [1, 2, 2, 1], padding='SAME')

layer5\_norm = tf.nn.local\_response\_normalization(layer5\_pool)

flat\_layer = flatten\_tf\_array(layer5\_norm)

layer6\_fccd = tf.matmul(flat\_layer, variables['w6']) + variables['b6']

layer6\_tanh = tf.tanh(layer6\_fccd)

layer6\_drop = tf.nn.dropout(layer6\_tanh, 0.5)

layer7\_fccd = tf.matmul(layer6\_drop, variables['w7']) + variables['b7']

layer7\_tanh = tf.tanh(layer7\_fccd)

layer7\_drop = tf.nn.dropout(layer7\_tanh, 0.5)

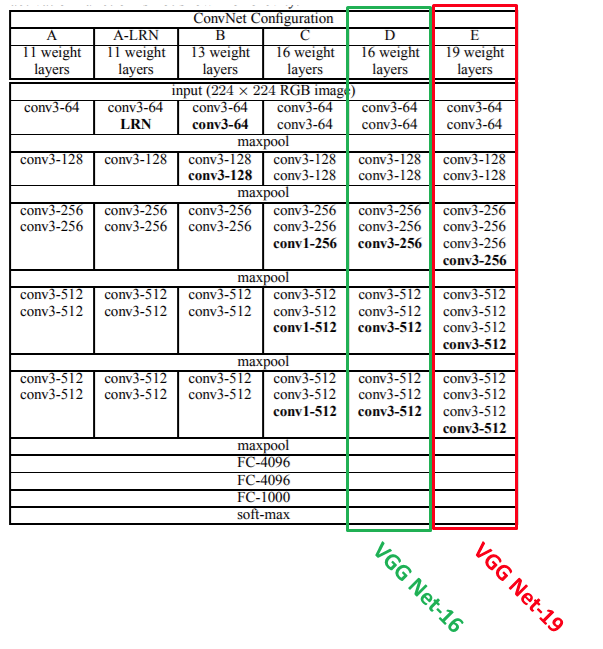
logits = tf.matmul(layer7\_drop, variables['w8']) + variables['b8']

return logits

**3.2 VGGNet-16**

VGGNet 比 AlexNet 拥有的层级更多（16-19 层），但是每一层的设计都简单了许多，所有层的滤波器大小都是 3 x 3、步幅都是 1，而所有的最大池化层的步幅都是 2。所以它虽然是一种深度 CNN，但结构比较简单。

VGGNet 有 16 层或 19 层两种配置，如下所示，这两种配置的不同之处在于它在第二个、第三个和第四个最大池化层后面到底是采用三个卷积层还是四个卷积层。



#The VGGNET Neural Network

VGG16\_PATCH\_SIZE\_1, VGG16\_PATCH\_SIZE\_2, VGG16\_PATCH\_SIZE\_3, VGG16\_PATCH\_SIZE\_4 = 3, 3, 3, 3

VGG16\_PATCH\_DEPTH\_1, VGG16\_PATCH\_DEPTH\_2, VGG16\_PATCH\_DEPTH\_3, VGG16\_PATCH\_DEPTH\_4 = 64, 128, 256, 512

VGG16\_NUM\_HIDDEN\_1, VGG16\_NUM\_HIDDEN\_2 = 4096, 1000

def variables\_vggnet16(patch\_size1 = VGG16\_PATCH\_SIZE\_1, patch\_size2 = VGG16\_PATCH\_SIZE\_2,

patch\_size3 = VGG16\_PATCH\_SIZE\_3, patch\_size4 = VGG16\_PATCH\_SIZE\_4,

patch\_depth1 = VGG16\_PATCH\_DEPTH\_1, patch\_depth2 = VGG16\_PATCH\_DEPTH\_2,

patch\_depth3 = VGG16\_PATCH\_DEPTH\_3, patch\_depth4 = VGG16\_PATCH\_DEPTH\_4,

num\_hidden1 = VGG16\_NUM\_HIDDEN\_1, num\_hidden2 = VGG16\_NUM\_HIDDEN\_2,

image\_width = 224, image\_height = 224, image\_depth = 3, num\_labels = 17):

w1 = tf.Variable(tf.truncated\_normal([patch\_size1, patch\_size1, image\_depth, patch\_depth1], stddev=0.1))

b1 = tf.Variable(tf.zeros([patch\_depth1]))

w2 = tf.Variable(tf.truncated\_normal([patch\_size1, patch\_size1, patch\_depth1, patch\_depth1], stddev=0.1))

b2 = tf.Variable(tf.constant(1.0, shape=[patch\_depth1]))

w3 = tf.Variable(tf.truncated\_normal([patch\_size2, patch\_size2, patch\_depth1, patch\_depth2], stddev=0.1))

b3 = tf.Variable(tf.constant(1.0, shape = [patch\_depth2]))

w4 = tf.Variable(tf.truncated\_normal([patch\_size2, patch\_size2, patch\_depth2, patch\_depth2], stddev=0.1))

b4 = tf.Variable(tf.constant(1.0, shape = [patch\_depth2]))

w5 = tf.Variable(tf.truncated\_normal([patch\_size3, patch\_size3, patch\_depth2, patch\_depth3], stddev=0.1))

b5 = tf.Variable(tf.constant(1.0, shape = [patch\_depth3]))

w6 = tf.Variable(tf.truncated\_normal([patch\_size3, patch\_size3, patch\_depth3, patch\_depth3], stddev=0.1))

b6 = tf.Variable(tf.constant(1.0, shape = [patch\_depth3]))

w7 = tf.Variable(tf.truncated\_normal([patch\_size3, patch\_size3, patch\_depth3, patch\_depth3], stddev=0.1))

b7 = tf.Variable(tf.constant(1.0, shape=[patch\_depth3]))

w8 = tf.Variable(tf.truncated\_normal([patch\_size4, patch\_size4, patch\_depth3, patch\_depth4], stddev=0.1))

b8 = tf.Variable(tf.constant(1.0, shape = [patch\_depth4]))

w9 = tf.Variable(tf.truncated\_normal([patch\_size4, patch\_size4, patch\_depth4, patch\_depth4], stddev=0.1))

b9 = tf.Variable(tf.constant(1.0, shape = [patch\_depth4]))

w10 = tf.Variable(tf.truncated\_normal([patch\_size4, patch\_size4, patch\_depth4, patch\_depth4], stddev=0.1))

b10 = tf.Variable(tf.constant(1.0, shape = [patch\_depth4]))

w11 = tf.Variable(tf.truncated\_normal([patch\_size4, patch\_size4, patch\_depth4, patch\_depth4], stddev=0.1))

b11 = tf.Variable(tf.constant(1.0, shape = [patch\_depth4]))

w12 = tf.Variable(tf.truncated\_normal([patch\_size4, patch\_size4, patch\_depth4, patch\_depth4], stddev=0.1))

b12 = tf.Variable(tf.constant(1.0, shape=[patch\_depth4]))

w13 = tf.Variable(tf.truncated\_normal([patch\_size4, patch\_size4, patch\_depth4, patch\_depth4], stddev=0.1))

b13 = tf.Variable(tf.constant(1.0, shape = [patch\_depth4]))

no\_pooling\_layers = 5

w14 = tf.Variable(tf.truncated\_normal([(image\_width // (2\*\*no\_pooling\_layers))\*(image\_height // (2\*\*no\_pooling\_layers))\*patch\_depth4 , num\_hidden1], stddev=0.1))

b14 = tf.Variable(tf.constant(1.0, shape = [num\_hidden1]))

w15 = tf.Variable(tf.truncated\_normal([num\_hidden1, num\_hidden2], stddev=0.1))

b15 = tf.Variable(tf.constant(1.0, shape = [num\_hidden2]))

w16 = tf.Variable(tf.truncated\_normal([num\_hidden2, num\_labels], stddev=0.1))

b16 = tf.Variable(tf.constant(1.0, shape = [num\_labels]))

variables = {

'w1': w1, 'w2': w2, 'w3': w3, 'w4': w4, 'w5': w5, 'w6': w6, 'w7': w7, 'w8': w8, 'w9': w9, 'w10': w10,

'w11': w11, 'w12': w12, 'w13': w13, 'w14': w14, 'w15': w15, 'w16': w16,

'b1': b1, 'b2': b2, 'b3': b3, 'b4': b4, 'b5': b5, 'b6': b6, 'b7': b7, 'b8': b8, 'b9': b9, 'b10': b10,

'b11': b11, 'b12': b12, 'b13': b13, 'b14': b14, 'b15': b15, 'b16': b16

}

return variables

def model\_vggnet16(data, variables):

layer1\_conv = tf.nn.conv2d(data, variables['w1'], [1, 1, 1, 1], padding='SAME')

layer1\_actv = tf.nn.relu(layer1\_conv + variables['b1'])

layer2\_conv = tf.nn.conv2d(layer1\_actv, variables['w2'], [1, 1, 1, 1], padding='SAME')

layer2\_actv = tf.nn.relu(layer2\_conv + variables['b2'])

layer2\_pool = tf.nn.max\_pool(layer2\_actv, [1, 2, 2, 1], [1, 2, 2, 1], padding='SAME')

layer3\_conv = tf.nn.conv2d(layer2\_pool, variables['w3'], [1, 1, 1, 1], padding='SAME')

layer3\_actv = tf.nn.relu(layer3\_conv + variables['b3'])

layer4\_conv = tf.nn.conv2d(layer3\_actv, variables['w4'], [1, 1, 1, 1], padding='SAME')

layer4\_actv = tf.nn.relu(layer4\_conv + variables['b4'])

layer4\_pool = tf.nn.max\_pool(layer4\_pool, [1, 2, 2, 1], [1, 2, 2, 1], padding='SAME')

layer5\_conv = tf.nn.conv2d(layer4\_pool, variables['w5'], [1, 1, 1, 1], padding='SAME')

layer5\_actv = tf.nn.relu(layer5\_conv + variables['b5'])

layer6\_conv = tf.nn.conv2d(layer5\_actv, variables['w6'], [1, 1, 1, 1], padding='SAME')

layer6\_actv = tf.nn.relu(layer6\_conv + variables['b6'])

layer7\_conv = tf.nn.conv2d(layer6\_actv, variables['w7'], [1, 1, 1, 1], padding='SAME')

layer7\_actv = tf.nn.relu(layer7\_conv + variables['b7'])

layer7\_pool = tf.nn.max\_pool(layer7\_actv, [1, 2, 2, 1], [1, 2, 2, 1], padding='SAME')

layer8\_conv = tf.nn.conv2d(layer7\_pool, variables['w8'], [1, 1, 1, 1], padding='SAME')

layer8\_actv = tf.nn.relu(layer8\_conv + variables['b8'])

layer9\_conv = tf.nn.conv2d(layer8\_actv, variables['w9'], [1, 1, 1, 1], padding='SAME')

layer9\_actv = tf.nn.relu(layer9\_conv + variables['b9'])

layer10\_conv = tf.nn.conv2d(layer9\_actv, variables['w10'], [1, 1, 1, 1], padding='SAME')

layer10\_actv = tf.nn.relu(layer10\_conv + variables['b10'])

layer10\_pool = tf.nn.max\_pool(layer10\_actv, [1, 2, 2, 1], [1, 2, 2, 1], padding='SAME')

layer11\_conv = tf.nn.conv2d(layer10\_pool, variables['w11'], [1, 1, 1, 1], padding='SAME')

layer11\_actv = tf.nn.relu(layer11\_conv + variables['b11'])

layer12\_conv = tf.nn.conv2d(layer11\_actv, variables['w12'], [1, 1, 1, 1], padding='SAME')

layer12\_actv = tf.nn.relu(layer12\_conv + variables['b12'])

layer13\_conv = tf.nn.conv2d(layer12\_actv, variables['w13'], [1, 1, 1, 1], padding='SAME')

layer13\_actv = tf.nn.relu(layer13\_conv + variables['b13'])

layer13\_pool = tf.nn.max\_pool(layer13\_actv, [1, 2, 2, 1], [1, 2, 2, 1], padding='SAME')

flat\_layer = flatten\_tf\_array(layer13\_pool)

layer14\_fccd = tf.matmul(flat\_layer, variables['w14']) + variables['b14']

layer14\_actv = tf.nn.relu(layer14\_fccd)

layer14\_drop = tf.nn.dropout(layer14\_actv, 0.5)

layer15\_fccd = tf.matmul(layer14\_drop, variables['w15']) + variables['b15']

layer15\_actv = tf.nn.relu(layer15\_fccd)

layer15\_drop = tf.nn.dropout(layer15\_actv, 0.5)

logits = tf.matmul(layer15\_drop, variables['w16']) + variables['b16']

return logits

上面已经为大家介绍了卷积神经网络，我们从 TensorFlow 的安装与基础概念、简单的全连接神经网络、数据的下载与导入、在 MNIST 上训练全连接神经网络、在 CIFAR-10 上训练经过修正的 LeNet 还有深度卷积神经网络等方面向大家介绍了神经网络，机器之心本文所有实验的代码、结果以及代码注释都将在 Github 上开放，这也是机器之心第一次试验性地向大家介绍教程以及实现。我们希望在为读者提供教程的同时也提供实际操作的经验，希望能为大家学习该教程起到积极的作用。