Deep Learning for Computer Vision: Assignment 4

Computer Science: COMS W 4995 006

Due: March 20, 2018

Problem

In this notebook we provide three networks for classifying handwritten digits from the MNIST dataset. The networks are implemented and tested using the Tensorflow framework. The third and final network is a convolutional neural network (CNN aka ConvNet) which achieves 99.25% accuracy on this dataset.

Your task is to re-implement all three networks using the Keras wrapper around Tensorflow OR re-implement using Pytorch. You will likely find several Keras or Pytorch implementations on the internet. It is ok to study these. However, you must not cut and paste this code into your assignment--you must write this yourself. Furthermore, you need to comment every line of code and succintly explain what it is doing!

Here is what is required:

a) A FULLY commented re-implementation of the ConvNet below using the Keras wrapper on Tensorflow OR Pytorch.

b) your network trained on the same MNIST data as used here.

c) an evaluation of the accuracy on the MNIST test set.

d) plots of 10 randomly selected digits from the test set along with the correct label and the assigned label.

e) have your training record a log of the data using the Keras API and then use Tensorboard (a command line tool) to display plots of the validation loss and validation accuracy. you can zip up a screenshot of this with your notebook before submission.

f) have your training continually save the best model so far (as determined by the validation loss) using the Keras API or Pytorch.

g) after training, load the saved weights using the best model so far. re-run you accuracy evaluation using these saved weights.

Below we include the Tensorflow examples shown in class.

A Simple Convolutional Neural Network in Tensorflow

This notebook covers a python and tensorflow-based solution to the handwritten digits recognition problem. It is based on tensorflow tutorials and Yann LeCun's early work on CNN's. This toturial compares a simple softmax regressor, a multi-layer perceptron (MLP), and a simple convolutional neural network (CNN).

Load in the MNIST digit dataset directly from tensorflow examples.

In [1]:
```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

The MNIST data is split into three parts: 55,000 data points of training data (mnist.train), 10,000 points of test data (mnist.test), and 5,000 points of validation data (mnist.validation).

Let's import tensorflow and begin an interactive session.

In [2]:
```
import tensorflow as tf
sess = tf.InteractiveSession()
```

Softmax Regression Model on the MNIST Digits Data

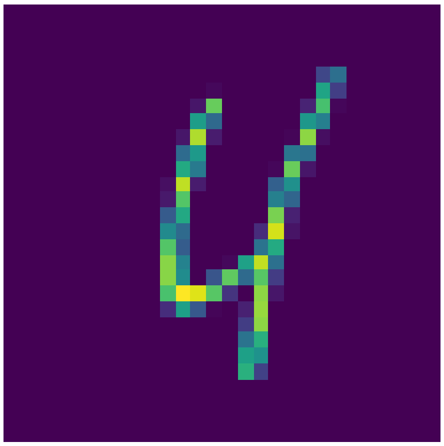
We need to create placeholders for the data. Data will be dumped here when it is batched from the MNIST dataset.

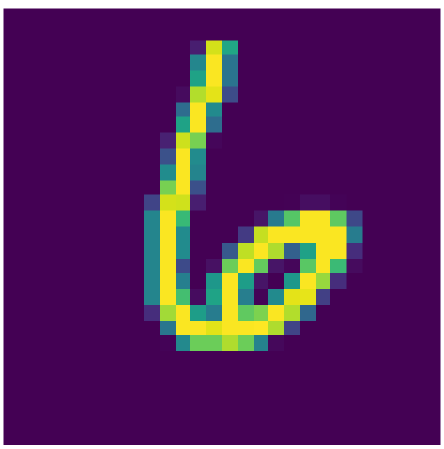
In [3]:
```
x = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10])
```

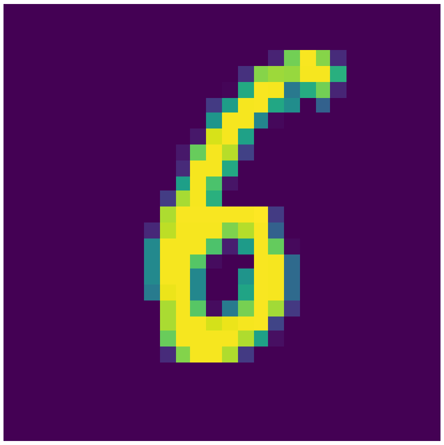
Now let's see what this data looks like.

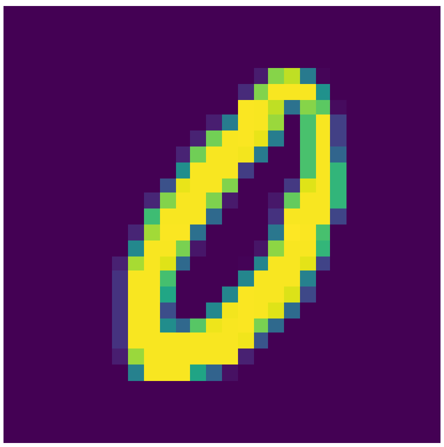
In [4]:

```
import matplotlib.pyplot as plt
import numpy as np

for i in range(4):
    batch = mnist.test.next_batch(1)
    image = np.asarray(batch[0]).reshape((28, 28))
    label = batch[1]

    plt.imshow(image)
    plt.axis("off")
    plt.show()
```









We are first going to do softmax logistic regression. This is a linear layer followed by softmax. Note there are NO hidden layers here. Also note that the digit images (28x28 grayscale images) are reshaped into a 784 element vector.

Below we create the parameters (weights) for our linear layer.

In [5]:

```
W = tf.Variable(tf.zeros([784,10]))
b = tf.Variable(tf.zeros([10]))
```

We then use tensorflows initializer to initialize these weights.

In [6]:

```
sess.run(tf.global_variables_initializer())
```

We create our linear layer as a function of the input and the weights.

In [7]:

```
y_regressor = tf.matmul(x,W) + b
```

Below we create our loss function. Note that the cross entropy is $H_{\hat{y}}(y) = -\sum_i \hat{y}_i \log(y_i)$ where $\hat{y}$ is the true probability distribution and is expressed as a one-hot vector, y is the estimated probability distribution, and i indexes elements of these two vectors. Also note that this reduces to

$H_{\hat{y}}(y) = -\log(y_{i^*})$ where i^* is the correct label. And if we sum this over all of our samples indexed by j, then $H_{\hat{y}}(y) = -\sum_j \log(y_{i^*}^{(j)})$. This is precisely the same loss function as we used before, but we called the MLE loss. They are one and the same.

In [8]:
```
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_regresso
```

Now we tell tf to use gradient descent with a step size of 0.5 and to minimize the cross entropy.

In [9]:
```
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entro
```

We train by grabbing mini-batches with 100 samples each and pushing these through the network to update our weights (W and b).

In [10]:
```
for _ in range(1000):
  batch = mnist.train.next_batch(100)
  train_step.run(feed_dict={x: batch[0], y_: batch[1]})
```

We define how to compute correct predicitions.

In [11]:
```
correct_prediction = tf.equal(tf.argmax(y_regressor,1), tf.argmax(y_,1))
```

And from these correct predictions how to compute the accuracy.

In [12]:
```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

In [13]:
```
print(accuracy.eval(feed_dict={x: mnist.test.images, y_: mnist.test.labe
```

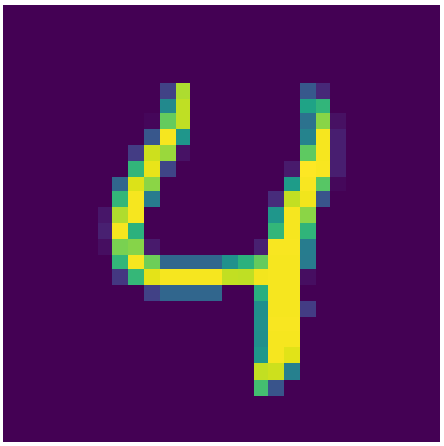
```
0.9192
```

Let's print out some test images and the corresponsing predictions made by the network. But first, let's add an output to the computation graph that computes the softmax probabilities.

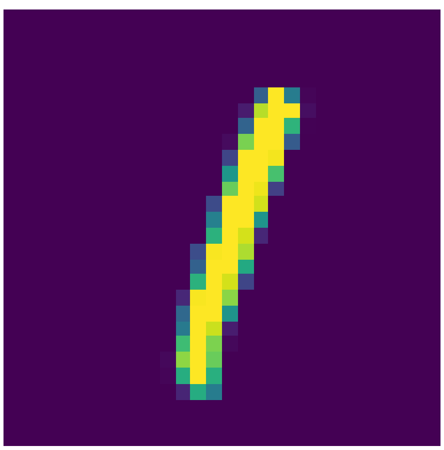
In [14]:
```
y_probs_regressor = tf.nn.softmax(logits=y_regressor, name=None)
```

In [15]:

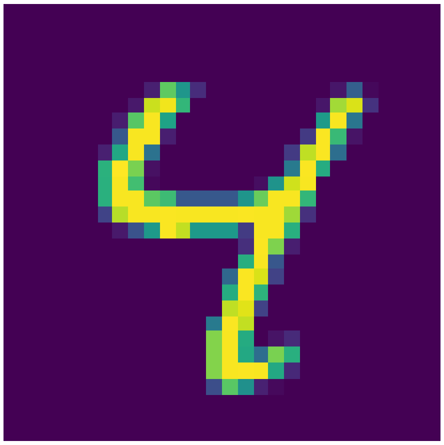
```
for i in range(5):
    batch = mnist.test.next_batch(1)
    image = np.asarray(batch[0]).reshape((28, 28))
    label = batch[1]

    plt.imshow(image)
    plt.axis("off")
    plt.show()
    print "Label = ", label
    print "Class probabilities = ", y_probs_regressor.eval(feed_dict={
        x: batch[0], y_: batch[1]})
```



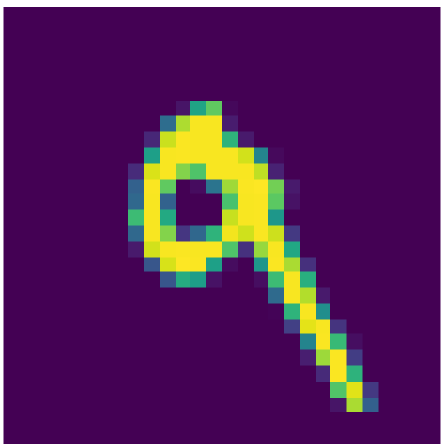
```
Label =  [[ 0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]]
Class probabilities =  [[  1.67203823e-03   6.03468334e-06   6.42732251e-03   1.87764294e-04
    9.26438749e-01   4.59894276e-04   3.28759407e-03   1.33758076e-02
    5.87336766e-03   4.22714762e-02]]
```



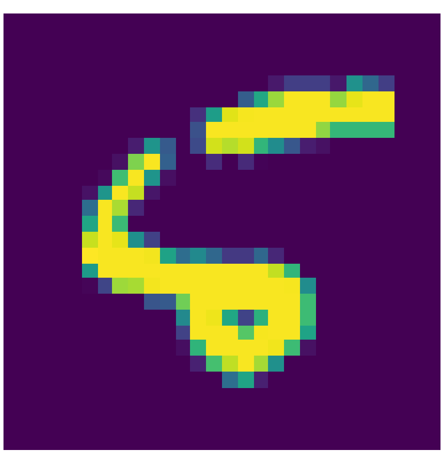
```
Label =  [[ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]]
Class probabilities =  [[  9.20873049e-07   9.87897158e-01   2.22241855e-03   1.86755520e-03
    3.91291178e-05   9.39880556e-05   3.30834628e-05   3.73039884e-03
    3.64026078e-03   4.75243636e-04]]
```



```
Label =  [[ 0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]]
Class probabilities =  [[  5.94127778e-06   1.63856862e-06   2.03961849e-06   2.16156419e-04
    9.74607527e-01   5.81776584e-03   5.42712551e-05   1.18222029e-03
    1.06608802e-02   7.45144626e-03]]
```



```
Label =  [[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]]
Class probabilities =  [[  1.32794867e-06   3.03926598e-03   1.02855719e-03   3.06596723e-03
    1.28692212e-02   9.07216594e-03   2.42878319e-04   1.96316512e-03
    6.66321721e-03   9.62054253e-01]]
```



```
Label =  [[ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]]
Class probabilities =  [[  8.77247099e-03   5.61122533e-06   5.13427751e-
03   1.86741403e-07
    6.87260833e-03   3.76259210e-03   9.73673403e-01   4.59226328e-07
    1.65836595e-03   1.20098288e-04]]
```

Softmax Multi-Layer Perceptron on the MNIST Digits Data

Here we define both weight and bias variables and how they are to be initialized. Note that the weights are are distributed according to a standard normal distribution (mean = 0, std = 0.1). This random initialization helps avoid hidden units get stuck together, as units that start with the same value will be updated identically in the non-convolutional layers. In contrast, the bias variables are set to a small positive number--this is help prevent hidden units from starting out and getting stuck in the zero part of the ReLU.

In [16]:

```
def weight_variable(shape):
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)

def bias_variable(shape):
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial)
```

Next we create placeholders for the training data.

In [17]:

```
x = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10])
```

We create the first and only fully connected hidden layer.

In [18]:

```
W_h = weight_variable([784, 512])
b_h = bias_variable([512])
h = tf.nn.relu(tf.matmul(x, W_h) + b_h)
```

We create the output layer.

In [19]:

```
W_out = weight_variable([512, 10])
b_out = bias_variable([10])
y_MLP = tf.matmul(h, W_out) + b_out
```

We again use cross entropy loss on a softmax distribution on the outputs.

In [20]:

```
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_MLP))
```

For training we choose an Adam learning rate and update rule. We then run this for 20,000 iterations and evaluate our accuracy after training. Note this softmax MLP network does quite a bit bettter than our softmax regressor. The non-linear layer really helps makes sense of the data! But we can do better still...

In [21]:
```
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_MLP,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
sess.run(tf.global_variables_initializer())
for i in range(20000):
  batch = mnist.train.next_batch(50)
  if i%1000 == 0:
    train_accuracy = accuracy.eval(feed_dict={
        x:batch[0], y_: batch[1]})
    print("step %d, training accuracy %g"%(i, train_accuracy))
  train_step.run(feed_dict={x: batch[0], y_: batch[1]})

print("test accuracy %g"%accuracy.eval(feed_dict={
    x: mnist.test.images, y_: mnist.test.labels}))
```

```
step 0, training accuracy 0.2
step 1000, training accuracy 0.9
step 2000, training accuracy 0.9
step 3000, training accuracy 0.98
step 4000, training accuracy 0.98
step 5000, training accuracy 0.94
step 6000, training accuracy 0.96
step 7000, training accuracy 1
step 8000, training accuracy 0.96
step 9000, training accuracy 1
step 10000, training accuracy 0.98
step 11000, training accuracy 1
step 12000, training accuracy 0.94
step 13000, training accuracy 0.98
step 14000, training accuracy 1
step 15000, training accuracy 0.96
step 16000, training accuracy 0.96
step 17000, training accuracy 0.98
step 18000, training accuracy 1
step 19000, training accuracy 0.96
test accuracy 0.9779
```

A Simple Convolutional Neural Network: LeNet

Here we make our first CNN. It's quite simple network, but it's surprisingly good at this handwritten digit recognition task. This a variant on Yann LeCun's CNN network that really helped to move deep learning forward.

We define both weight and bias variables and how they are to be initialized. Note that the weights are are distributed according to a standard normal distribution (mean = 0, std = 0.1). This random initialization helps avoid hidden units get stuck together, as units that start with the same value will be updated identically in the non-convolutional layers. In contrast, the bias variables are set to a small positive number--this is help prevent hidden units from starting out and getting stuck in the zero part of the ReLu.

In [22]:

```
def weight_variable(shape):
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)

def bias_variable(shape):
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial)
```

Next we define how the convolution is to be computed and the extent and type of pooling. The convolution will use a 5x5 kernel and will pad the image with zeros around the edges and use a stride of 1 pixel so that the resulting image (after convolution) has the same size as the original input image. The network will learn the weights for a stack of 32 separate kernels along with 32 bias variables. Finally, after the ReLu is performed the result will be under go 2x2 max pooling, thus halfing both dimensions of the image. The choices for the stride, padding, and pooling are not parameters that the network needs to estimate. Rather these are termed "hyperparamters" that are usually set by the network designer.

In [23]:

```
def conv2d(x, W):
  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')
```

This creates the weight and bias variables for the first convolutional layer as described above. Note the output has depth 32, so there will be 32 feature images after this layer.

In [24]:

```
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
```

Unlike for our softmax regressor above, here we need keep the images as images and not collapse these into vectors; this allows us to perform the 2D convolution.

In [25]:

```
x_image = tf.reshape(x, [-1,28,28,1])
```

Finally, we define are first layer of our CNN!

In [26]:

```
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)
```

And wasting no time, we define are second layer. The second layer will have to process 32 feature images coming out of the first layer. Note that the images input to this layer have $\frac{1}{4}$ the number of pixels as the original input images due to the 2x2 pooling in the previous layer. Note that convolution layer NOT fully connected as our previous hidden layers have been. A unit in the output layer has a limited "receptive field." Its connections to the input layer are spatially limited by the kernel (or filter) size. Also, because of weight sharing in convolutional layers, the number of parameters for a convolutional is the size of the kernel x the depth of the input layer x depth of the output layer + depth of the output layer. So for the second layer of our ConvNet, we have 5 x 5 x 32 x 64 + 64 = 51,264 parameters.

In [27]:

```
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
```

After the pooling stage of our second convolutional layer, we have 64 7x7 "feature" images. In one penultimate fully connected hidden layer, we are going to map these feature imges to a 1024 dimensional feature space. Note we need to flatten these feature images to do this.

In [28]:

```
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

Dropout is added here, although it is not really needed for such small network.

In [29]:

```
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

We have a final linear output layer mapping features to scores topped off with a softmax cross entropy loss function, as explained earlier.

In [30]:

```
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
```

In [31]:

```
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_conv))
```

For training we choose an Adam learning rate and update rule. We then run this for 20,000 iterations and evaluate our accuracy after training.

In [32]:
```
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
sess.run(tf.global_variables_initializer())
for i in range(20000):
  batch = mnist.train.next_batch(50)
  if i%1000 == 0:
    train_accuracy = accuracy.eval(feed_dict={
        x:batch[0], y_: batch[1], keep_prob: 1.0})
    print("step %d, training accuracy %g"%(i, train_accuracy))
  train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

print("test accuracy %g"%accuracy.eval(feed_dict={
    x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

```
step 0, training accuracy 0.16
step 1000, training accuracy 0.94
step 2000, training accuracy 1
step 3000, training accuracy 0.98
step 4000, training accuracy 1
step 5000, training accuracy 0.96
step 6000, training accuracy 0.98
step 7000, training accuracy 1
step 8000, training accuracy 1
step 9000, training accuracy 0.98
step 10000, training accuracy 1
step 11000, training accuracy 1
step 12000, training accuracy 1
step 13000, training accuracy 1
step 14000, training accuracy 1
step 15000, training accuracy 1
step 16000, training accuracy 1
step 17000, training accuracy 1
step 18000, training accuracy 1
step 19000, training accuracy 1
test accuracy 0.9925
```

We add an output to compuational graph that computes the label probabilities.

In [33]:
```
y_probs = tf.nn.softmax(logits=y_conv, name=None)
```

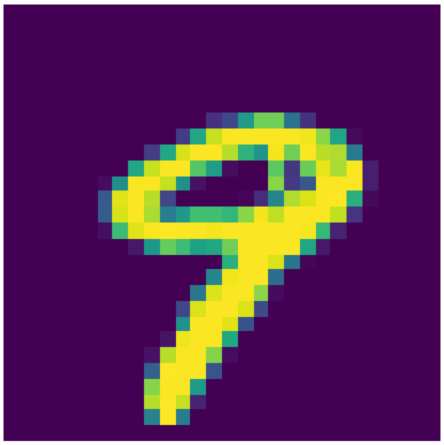
In [34]:
```
print("test accuracy %g"%accuracy.eval(feed_dict={
    x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

```
test accuracy 0.9925
```

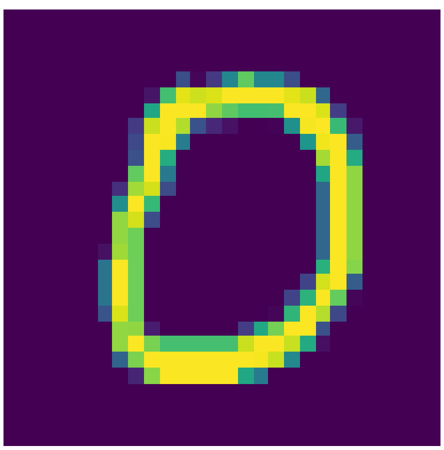
Next we step through some test examples and see how well the network is doing.

In [35]:

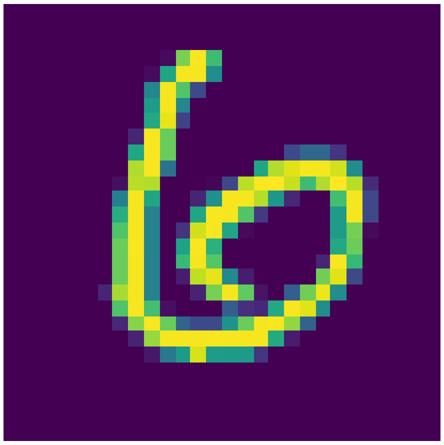
```python
for i in range(5):
    batch = mnist.test.next_batch(1)
    image = np.asarray(batch[0]).reshape((28, 28))
    label = batch[1]

    plt.imshow(image)
    plt.axis("off")
    plt.show()
    print "Label = ", label
    print "Class probabilities = ", y_probs.eval(feed_dict={
        x: batch[0], y_: batch[1], keep_prob: 1.0})
```



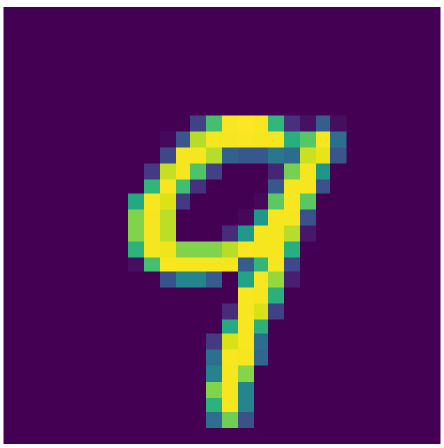
```
Label =  [[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]]
Class probabilities =  [[  5.28820287e-14   1.61565119e-12   1.47112888e-
12   1.99402633e-10
    8.20477112e-07   1.07506337e-09   4.57562201e-15   8.61227306e-07
    3.80472631e-09   9.99998331e-01]]
```



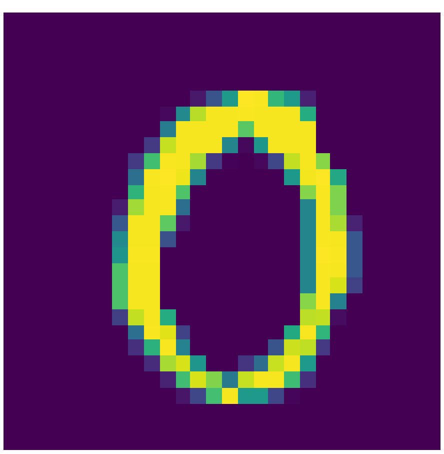
```
Label =  [[ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
Class probabilities =  [[  1.00000000e+00   3.50560683e-12   1.00434594e-
09   1.73949210e-14
    8.60713218e-16   3.75027960e-12   9.13440157e-11   2.25712261e-11
    1.23941948e-12   4.15561162e-11]]
```



```
Label =  [[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  0.]]
Class probabilities =  [[  1.41643053e-09   7.71405238e-14   1.62943230e-
15   5.57100261e-15
    2.53674198e-13   1.26207560e-11   1.00000000e+00   4.33625261e-15
    3.95802037e-11   7.02078399e-16]]
```



```
Label =  [[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]]
Class probabilities =  [[  3.76756785e-12   1.12312442e-11   3.20851019e-
11   2.01345451e-09
    2.59802891e-05   6.02736749e-09   2.61297559e-12   4.99466495e-08
    6.30753760e-09   9.99974012e-01]]
```



```
Label =  [[ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
Class probabilities =  [[  1.00000000e+00   1.98105139e-12   1.07032493e-
12   2.90848210e-14
    8.34232247e-15   6.83068185e-12   1.36369707e-08   4.16134821e-10
    5.86183532e-12   1.50796278e-10]]
```

In [2]:

```
from __future__ import print_function
import numpy as np
import os

import tensorflow as tf
import keras
from keras.layers import Dense, Conv2D
from keras.layers import Activation
from keras.layers import MaxPooling2D, Dropout
from keras.layers import Input, Flatten
from keras.optimizers import SGD, Adam
from keras.callbacks import ModelCheckpoint, TensorBoard
from keras import backend as K
from keras.models import Model, load_model

```

In [3]:

```
num_classes = 10
```

Load Data

In [7]:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
rows = cols = 28
channels = 1
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

In [9]:

```
train_data = mnist.train.images.reshape(-1, rows, cols, 1)
train_labels = mnist.train.labels
val_data = mnist.validation.images.reshape(-1, rows, cols, 1)
val_labels = mnist.validation.labels
test_data = mnist.test.images.reshape(-1, rows, cols, 1)
test_labels = mnist.test.labels
```

(Keras Implementation) Softmax Regression Model on the MNIST Digits Data

In [54]:

```
save_dir = "saved_models/"
model_name = "softmax_regr"
if not os.path.isdir(save_dir):
    os.makedirs(save_dir)
modelpath = os.path.join(save_dir, model_name + "_model.h5")
weightpath = os.path.join(save_dir, model_name + "_weight.h5")

# Network parameters for learning rate, batch size,
# number of epochs
lr = 0.5
batch_size = 100
epochs = 18

# Define expected input shape
input_shape = (rows, cols, channels)
inputs = Input(shape = input_shape)

# Flatten input to a vector
x = Flatten()(inputs)

# Fully connected layer with units = 10
# equivalent to the number of classes
# Apply softmax activation to all
outputs = Dense(10, activation = 'softmax',
                kernel_initializer = 'zeros')(x)

# Define mode
model = Model(inputs = inputs, outputs = outputs)
# Specify categorical crossentropy loss with
# Stochastic gradient descent (learning rate = 0.5)
model.compile(loss = 'categorical_crossentropy',
              optimizer = SGD(lr = 0.5),
              metrics = ['accuracy'])
model.summary()

# Callback to save logs for tensorboard
log_dir = "./logs/" + model_name + "/"
print(log_dir)
tensorboard = TensorBoard(log_dir = log_dir, batch_size = batch_size)

# Callback to save the best model
checkpoint = ModelCheckpoint(filepath = modelpath,
                             verbose = 1,
                             save_best_only = True)

callbacks = [tensorboard, checkpoint]

# Fit training data with batch size of 100
# Save model based on validation loss
model.fit(train_data, train_labels,
          batch_size = batch_size, epochs = epochs,
          validation_data = (val_data, val_labels),
          shuffle = True, callbacks = callbacks)

# Evaluate Test data
score = model.evaluate(test_data, test_labels, verbose = 1)
```

```
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_14 (InputLayer)        (None, 28, 28, 1)         0         
_________________________________________________________________
flatten_14 (Flatten)         (None, 784)               0         
_________________________________________________________________
dense_19 (Dense)             (None, 10)                7850      
=================================================================
Total params: 7,850
Trainable params: 7,850
Non-trainable params: 0
_________________________________________________________________
./logs/softmax_regr/
Train on 55000 samples, validate on 5000 samples
Epoch 1/18
53400/55000 [============================>.] - ETA: 0s - loss: 0.3999 - a
cc: 0.8854Epoch 00000: val_loss improved from inf to 0.29680, saving mode
l to saved_models/softmax_regr_model.h5
55000/55000 [==============================] - 2s - loss: 0.3992 - acc:
0.8856 - val_loss: 0.2968 - val_acc: 0.9182
Epoch 2/18
54300/55000 [============================>.] - ETA: 0s - loss: 0.3110 - a
cc: 0.9111Epoch 00001: val_loss improved from 0.29680 to 0.27905, saving
model to saved_models/softmax_regr_model.h5
55000/55000 [==============================] - 2s - loss: 0.3104 - acc:
0.9113 - val_loss: 0.2790 - val_acc: 0.9206
Epoch 3/18
53900/55000 [============================>.] - ETA: 0s - loss: 0.2945 - a
cc: 0.9167Epoch 00002: val_loss improved from 0.27905 to 0.27102, saving
model to saved_models/softmax_regr_model.h5
55000/55000 [==============================] - 2s - loss: 0.2948 - acc:
0.9167 - val_loss: 0.2710 - val_acc: 0.9266
Epoch 4/18
54500/55000 [============================>.] - ETA: 0s - loss: 0.2860 - a
cc: 0.9194Epoch 00003: val_loss improved from 0.27102 to 0.27017, saving
model to saved_models/softmax_regr_model.h5
55000/55000 [==============================] - 2s - loss: 0.2864 - acc:
0.9195 - val_loss: 0.2702 - val_acc: 0.9224
Epoch 5/18
54600/55000 [============================>.] - ETA: 0s - loss: 0.2813 - a
cc: 0.9206Epoch 00004: val_loss did not improve
55000/55000 [==============================] - 2s - loss: 0.2813 - acc:
0.9206 - val_loss: 0.2705 - val_acc: 0.9250
Epoch 6/18
54100/55000 [============================>.] - ETA: 0s - loss: 0.2777 - a
cc: 0.9224Epoch 00005: val_loss improved from 0.27017 to 0.26663, saving
model to saved_models/softmax_regr_model.h5
55000/55000 [==============================] - 1s - loss: 0.2780 - acc:
0.9224 - val_loss: 0.2666 - val_acc: 0.9260
Epoch 7/18
53800/55000 [============================>.] - ETA: 0s - loss: 0.2751 - a
cc: 0.9236Epoch 00006: val_loss improved from 0.26663 to 0.26292, saving
model to saved_models/softmax_regr_model.h5
```

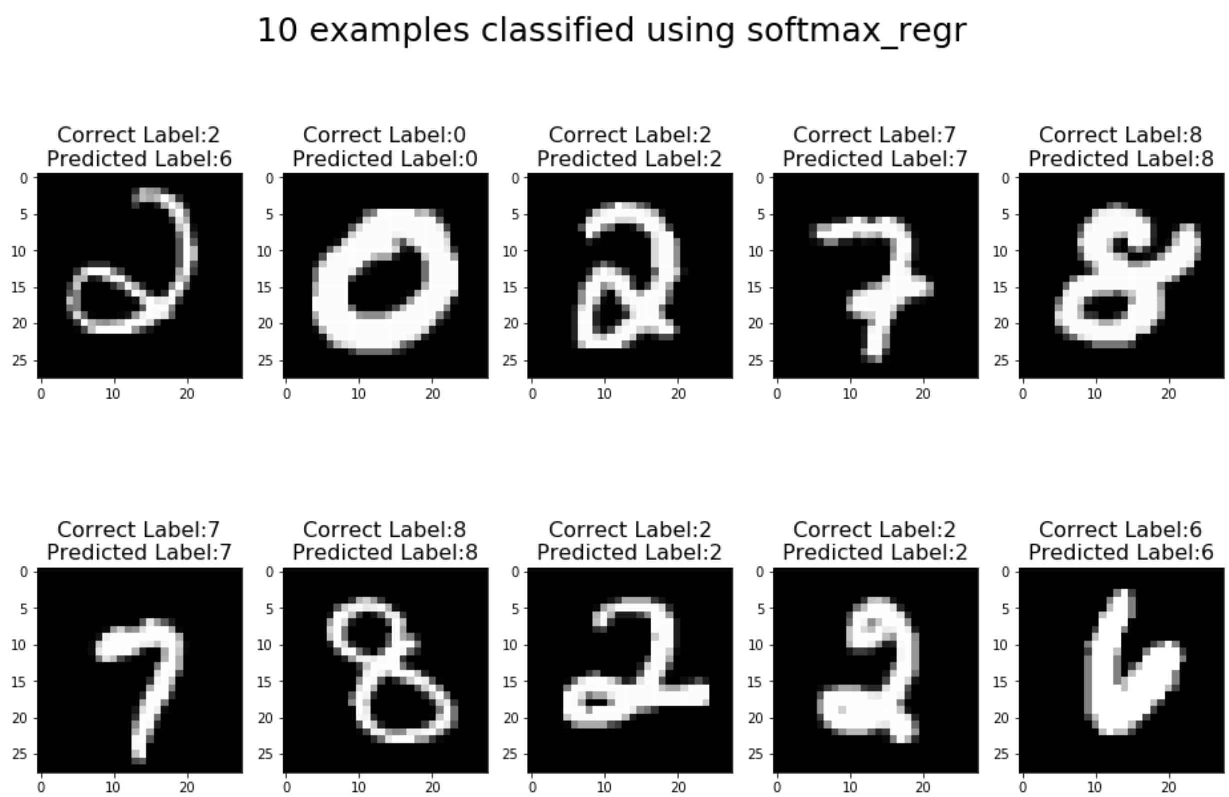
```
55000/55000 [==============================] - 2s - loss: 0.2742 - acc:
0.9238 - val_loss: 0.2629 - val_acc: 0.9276
Epoch 8/18
54800/55000 [============================>.] - ETA: 0s - loss: 0.2710 - a
cc: 0.9243Epoch 00007: val_loss did not improve
55000/55000 [==============================] - 2s - loss: 0.2709 - acc:
0.9244 - val_loss: 0.2696 - val_acc: 0.9252
Epoch 9/18
54400/55000 [============================>.] - ETA: 0s - loss: 0.2685 - a
cc: 0.9249Epoch 00008: val_loss did not improve
55000/55000 [==============================] - 2s - loss: 0.2695 - acc:
0.9247 - val_loss: 0.2701 - val_acc: 0.9254
Epoch 10/18
54700/55000 [============================>.] - ETA: 0s - loss: 0.2677 - a
cc: 0.9253Epoch 00009: val_loss did not improve
55000/55000 [==============================] - 2s - loss: 0.2675 - acc:
0.9253 - val_loss: 0.2680 - val_acc: 0.9256
Epoch 11/18
54500/55000 [============================>.] - ETA: 0s - loss: 0.2660 - a
cc: 0.9266Epoch 00010: val_loss did not improve
55000/55000 [==============================] - 2s - loss: 0.2660 - acc:
0.9265 - val_loss: 0.2675 - val_acc: 0.9276
Epoch 12/18
53700/55000 [============================>.] - ETA: 0s - loss: 0.2636 - a
cc: 0.9266Epoch 00011: val_loss did not improve
55000/55000 [==============================] - 2s - loss: 0.2637 - acc:
0.9267 - val_loss: 0.2701 - val_acc: 0.9256
Epoch 13/18
53300/55000 [============================>.] - ETA: 0s - loss: 0.2640 - a
cc: 0.9263Epoch 00012: val_loss did not improve
55000/55000 [==============================] - 1s - loss: 0.2630 - acc:
0.9266 - val_loss: 0.2641 - val_acc: 0.9272
Epoch 14/18
54500/55000 [============================>.] - ETA: 0s - loss: 0.2612 - a
cc: 0.9266Epoch 00013: val_loss did not improve
55000/55000 [==============================] - 1s - loss: 0.2615 - acc:
0.9265 - val_loss: 0.2718 - val_acc: 0.9232
Epoch 15/18
53900/55000 [============================>.] - ETA: 0s - loss: 0.2601 - a
cc: 0.9271Epoch 00014: val_loss did not improve
55000/55000 [==============================] - 1s - loss: 0.2607 - acc:
0.9269 - val_loss: 0.2689 - val_acc: 0.9284
Epoch 16/18
53900/55000 [============================>.] - ETA: 0s - loss: 0.2592 - a
cc: 0.9272Epoch 00015: val_loss did not improve
55000/55000 [==============================] - 2s - loss: 0.2593 - acc:
0.9273 - val_loss: 0.2647 - val_acc: 0.9282
Epoch 17/18
54700/55000 [============================>.] - ETA: 0s - loss: 0.2589 - a
cc: 0.9273Epoch 00016: val_loss did not improve
55000/55000 [==============================] - 2s - loss: 0.2589 - acc:
0.9273 - val_loss: 0.2653 - val_acc: 0.9288
Epoch 18/18
53700/55000 [============================>.] - ETA: 0s - loss: 0.2579 - a
cc: 0.9280Epoch 00017: val_loss did not improve
55000/55000 [==============================] - 2s - loss: 0.2574 - acc:
0.9282 - val_loss: 0.2765 - val_acc: 0.9214
```

```
 9792/10000 [============================>.] - ETA: 0sTest loss: 0.285547
93046
Test accuracy: 0.9194
```

Softmax Regression: Plot 10 random examples with correct and predicted label

In [61]:

```python
import matplotlib.pyplot as plt

save_dir = "saved_models/"
model_name = "softmax_regr"
modelpath = os.path.join(save_dir, model_name + "_model.h5")

model = load_model(modelpath)

fig, ax = plt.subplots(2, 5, figsize = (16, 10))
plt.suptitle("10 examples classified using " + model_name,
             fontsize = 25)
for i in range(2):
    for j in range(5):
        sample = mnist.test.next_batch(1)
        image = sample[0].reshape(rows, cols)
        label = sample[1]
        pred_label = model.predict(image.reshape(-1, rows, cols, 1))
        label = np.argmax(label)
        pred_label = np.argmax(pred_label)
        ax[i][j].imshow(image, cmap = 'gray')
        ax[i][j].set_title("Correct Label:" + str(label) +
                           "\nPredicted Label:" + str(pred_label),
                           fontsize = 16)

plt.show()
```



(Keras Implementation) Softmax Multi-Layer Perceptron

In [57]:

```
# Directory and filename for saving the best model
save_dir = "saved_models/"
model_name = "softmax_mlp"
if not os.path.isdir(save_dir):
    os.makedirs(save_dir)
modelpath = os.path.join(save_dir, model_name + "_model.h5")

# Optimization parameters for learning like learning rate,
# batch size, epochs
lr = 1e-4
batch_size = 50
epochs = 18
seed = np.random.randint(1000)
print("Seed: %d" %seed)
np.random.seed(seed)

# Weight initializer based on normal distrubution of
# mean 0, stddev 0.1 used for the layers
weight_init = keras.initializers.RandomNormal(stddev = 0.1,
                                              seed = seed)
# Bias initializer to a constant 0.1 value used for the layers
bias_init = keras.initializers.Constant(value = 0.1)

# Specify input shape (doesn't change from before)
inputs = Input(shape = input_shape)

# Flatten input to a vector since this is an MLP
x = Flatten()(inputs)
# 512 fully connected units with ReLU activation
x = Dense(512, activation = 'relu',
          kernel_initializer = weight_init,
          bias_initializer = bias_init)(x)
# Output layer with units equal to the number of classes
# Uses softmax activation for probability distribution
outputs = Dense(num_classes, activation = 'softmax',
          kernel_initializer = weight_init,
          bias_initializer = bias_init)(x)
# Define model
model = Model(inputs = inputs, outputs = outputs)
# Compile model by specifying loss and optimizer
model.compile(loss = 'categorical_crossentropy',
              optimizer = Adam(lr = lr),
              metrics = ['accuracy'])
model.summary()

# Define log callback for tensorboard
log_dir = "./logs/" + model_name + "/"
tensorboard = TensorBoard(log_dir = log_dir, batch_size = batch_size)

# Define callback for saving the best model
checkpoint = ModelCheckpoint(filepath = modelpath,
                             verbose = 1,
                             save_best_only = True)

callbacks = [tensorboard, checkpoint]

```

```python
# Fit training data and use validation loss for checkpoints
model.fit(train_data, train_labels,
          batch_size = batch_size, epochs = epochs,
          validation_data = (val_data, val_labels),
          shuffle = True, callbacks = callbacks)

# Evaluate data on the test set
score = model.evaluate(test_data, test_labels, verbose = 1)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Seed: 321
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_16 (InputLayer)        (None, 28, 28, 1)         0
_________________________________________________________________
flatten_16 (Flatten)         (None, 784)               0
_________________________________________________________________
dense_22 (Dense)             (None, 512)               401920
_________________________________________________________________
dense_23 (Dense)             (None, 10)                5130
=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
_________________________________________________________________
Train on 55000 samples, validate on 5000 samples
Epoch 1/18
54700/55000 [============================>.] - ETA: 0s - loss: 0.5422 - acc: 0.8418Epoch 00000: val_loss improved from inf to 0.26292, saving model to saved_models/softmax_mlp_model.h5
55000/55000 [==============================] - 12s - loss: 0.5406 - acc: 0.8423 - val_loss: 0.2629 - val_acc: 0.9276
Epoch 2/18
54850/55000 [============================>.] - ETA: 0s - loss: 0.2433 - acc: 0.9321Epoch 00001: val_loss improved from 0.26292 to 0.19658, saving model to saved_models/softmax_mlp_model.h5
55000/55000 [==============================] - 11s - loss: 0.2432 - acc: 0.9321 - val_loss: 0.1966 - val_acc: 0.9478
Epoch 3/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.1889 - acc: 0.9479Epoch 00002: val_loss improved from 0.19658 to 0.16356, saving model to saved_models/softmax_mlp_model.h5
55000/55000 [==============================] - 12s - loss: 0.1890 - acc: 0.9479 - val_loss: 0.1636 - val_acc: 0.9576
Epoch 4/18
54750/55000 [============================>.] - ETA: 0s - loss: 0.1562 - acc: 0.9566Epoch 00003: val_loss improved from 0.16356 to 0.14209, saving model to saved_models/softmax_mlp_model.h5
55000/55000 [==============================] - 11s - loss: 0.1561 - acc: 0.9567 - val_loss: 0.1421 - val_acc: 0.9624
Epoch 5/18
54750/55000 [============================>.] - ETA: 0s - loss: 0.1324 - acc: 0.9639Epoch 00004: val_loss improved from 0.14209 to 0.12751, saving model to saved_models/softmax_mlp_model.h5
55000/55000 [==============================] - 10s - loss: 0.1322 - acc:
```

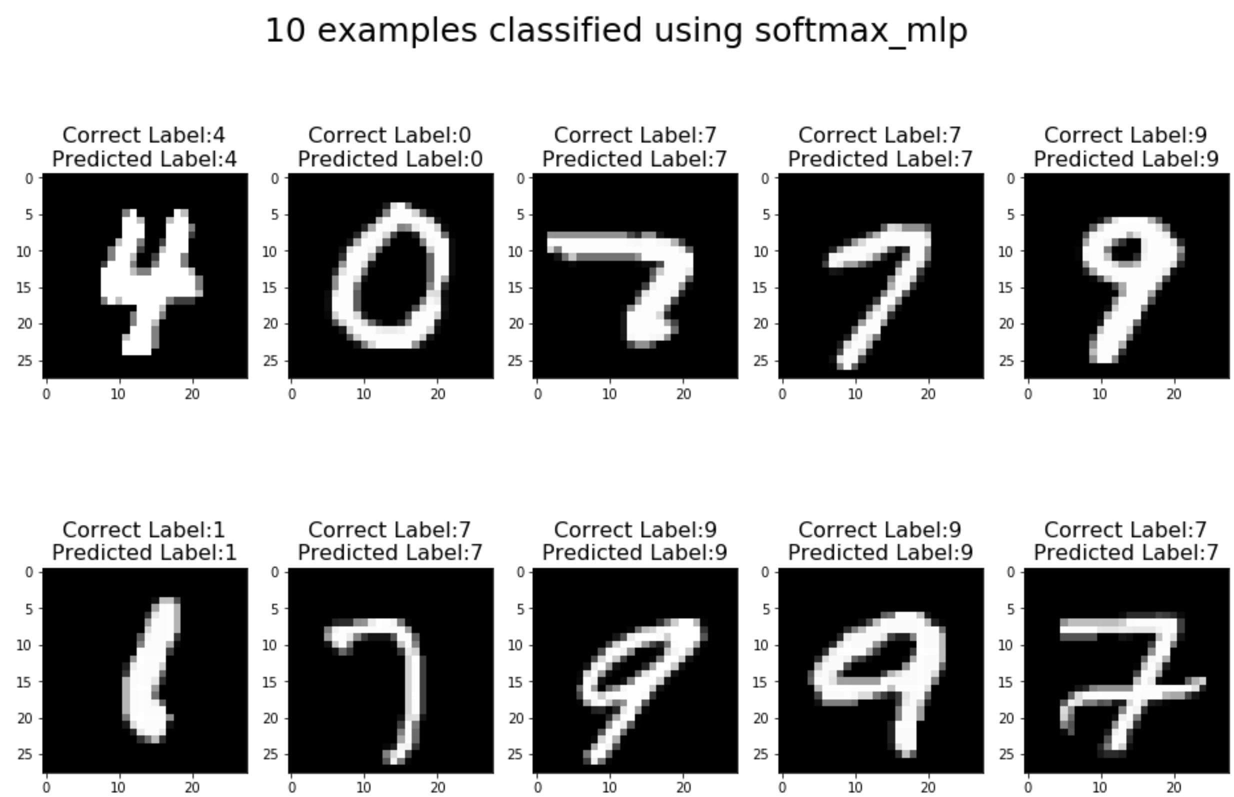
```
0.9639 - val_loss: 0.1275 - val_acc: 0.9654
Epoch 6/18
54900/55000 [============================>.] - ETA: 0s - loss: 0.1143 - a
cc: 0.9688Epoch 00005: val_loss improved from 0.12751 to 0.11684, saving
model to saved_models/softmax_mlp_model.h5
55000/55000 [==============================] - 10s - loss: 0.1143 - acc:
0.9687 - val_loss: 0.1168 - val_acc: 0.9680
Epoch 7/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.0999 - a
cc: 0.9730Epoch 00006: val_loss improved from 0.11684 to 0.10623, saving
model to saved_models/softmax_mlp_model.h5
55000/55000 [==============================] - 10s - loss: 0.0999 - acc:
0.9730 - val_loss: 0.1062 - val_acc: 0.9706
Epoch 8/18
54900/55000 [============================>.] - ETA: 0s - loss: 0.0878 - a
cc: 0.9764Epoch 00007: val_loss improved from 0.10623 to 0.09835, saving
model to saved_models/softmax_mlp_model.h5
55000/55000 [==============================] - 10s - loss: 0.0878 - acc:
0.9764 - val_loss: 0.0984 - val_acc: 0.9728
Epoch 9/18
54800/55000 [============================>.] - ETA: 0s - loss: 0.0777 - a
cc: 0.9797Epoch 00008: val_loss improved from 0.09835 to 0.09445, saving
model to saved_models/softmax_mlp_model.h5
55000/55000 [==============================] - 10s - loss: 0.0776 - acc:
0.9797 - val_loss: 0.0945 - val_acc: 0.9732
Epoch 10/18
54900/55000 [============================>.] - ETA: 0s - loss: 0.0693 - a
cc: 0.9816Epoch 00009: val_loss improved from 0.09445 to 0.09094, saving
model to saved_models/softmax_mlp_model.h5
55000/55000 [==============================] - 10s - loss: 0.0693 - acc:
0.9816 - val_loss: 0.0909 - val_acc: 0.9750
Epoch 11/18
54800/55000 [============================>.] - ETA: 0s - loss: 0.0617 - a
cc: 0.9842Epoch 00010: val_loss improved from 0.09094 to 0.08582, saving
model to saved_models/softmax_mlp_model.h5
55000/55000 [==============================] - 11s - loss: 0.0618 - acc:
0.9841 - val_loss: 0.0858 - val_acc: 0.9760
Epoch 12/18
54800/55000 [============================>.] - ETA: 0s - loss: 0.0554 - a
cc: 0.9862Epoch 00011: val_loss improved from 0.08582 to 0.08305, saving
model to saved_models/softmax_mlp_model.h5
55000/55000 [==============================] - 11s - loss: 0.0555 - acc:
0.9861 - val_loss: 0.0830 - val_acc: 0.9754
Epoch 13/18
54800/55000 [============================>.] - ETA: 0s - loss: 0.0497 - a
cc: 0.9880Epoch 00012: val_loss improved from 0.08305 to 0.07948, saving
model to saved_models/softmax_mlp_model.h5
55000/55000 [==============================] - 12s - loss: 0.0497 - acc:
0.9880 - val_loss: 0.0795 - val_acc: 0.9766
Epoch 14/18
54750/55000 [============================>.] - ETA: 0s - loss: 0.0446 - a
cc: 0.9896Epoch 00013: val_loss improved from 0.07948 to 0.07715, saving
model to saved_models/softmax_mlp_model.h5
55000/55000 [==============================] - 12s - loss: 0.0447 - acc:
0.9896 - val_loss: 0.0771 - val_acc: 0.9770
Epoch 15/18
54850/55000 [============================>.] - ETA: 0s - loss: 0.0403 - a
```

```
cc: 0.9906Epoch 00014: val_loss improved from 0.07715 to 0.07471, saving
model to saved_models/softmax_mlp_model.h5
55000/55000 [==============================] - 11s - loss: 0.0403 - acc:
0.9906 - val_loss: 0.0747 - val_acc: 0.9786
Epoch 16/18
54700/55000 [============================>.] - ETA: 0s - loss: 0.0363 - a
cc: 0.9919Epoch 00015: val_loss improved from 0.07471 to 0.07189, saving
model to saved_models/softmax_mlp_model.h5
55000/55000 [==============================] - 12s - loss: 0.0363 - acc:
0.9919 - val_loss: 0.0719 - val_acc: 0.9786
Epoch 17/18
54750/55000 [============================>.] - ETA: 0s - loss: 0.0328 - a
cc: 0.9931Epoch 00016: val_loss did not improve
55000/55000 [==============================] - 12s - loss: 0.0327 - acc:
0.9930 - val_loss: 0.0735 - val_acc: 0.9774
Epoch 18/18
54750/55000 [============================>.] - ETA: 0s - loss: 0.0293 - a
cc: 0.9941Epoch 00017: val_loss improved from 0.07189 to 0.07110, saving
model to saved_models/softmax_mlp_model.h5
55000/55000 [==============================] - 11s - loss: 0.0294 - acc:
0.9941 - val_loss: 0.0711 - val_acc: 0.9794
 9632/10000 [===========================>..] - ETA: 0sTest loss: 0.073113
9129666
Test accuracy: 0.9781
```

Softmax MLP: Plot 10 random examples with correct and predicted label

In [62]:

```
import matplotlib.pyplot as plt

save_dir = "saved_models/"
model_name = "softmax_mlp"
modelpath = os.path.join(save_dir, model_name + "_model.h5")

model = load_model(modelpath)

fig, ax = plt.subplots(2, 5, figsize = (16, 10))
plt.suptitle("10 examples classified using " + model_name,
             fontsize = 25)
for i in range(2):
    for j in range(5):
        sample = mnist.test.next_batch(1)
        image = sample[0].reshape(rows, cols)
        label = sample[1]
        pred_label = model.predict(image.reshape(-1, rows, cols, 1))
        label = np.argmax(label)
        pred_label = np.argmax(pred_label)
        ax[i][j].imshow(image, cmap = 'gray')
        ax[i][j].set_title("Correct Label:" + str(label) +
                           "\nPredicted Label:" + str(pred_label),
                           fontsize = 16)

plt.show()
```



(Keras Implementation) Simple CNN: LeNet

In [63]:
```
# Directory and filename for saving the best model
save_dir = "saved_models/"
model_name = "simple_cnn"
if not os.path.isdir(save_dir):
    os.makedirs(save_dir)
modelpath = os.path.join(save_dir, model_name + "_model.h5")

# Specify parameters like learning rate, batch size
lr = 1e-4
batch_size = 50

seed = np.random.randint(1000)
print("Seed: %d" %seed)
np.random.seed(seed)

# Initializer for kernel from normal distribution with
# mean 0 and stddev 0.1 for all layers
weight_init = keras.initializers.RandomNormal(stddev = 0.1,
                                               seed = seed)
# Initiliazer for bias to constant value of 0.1 for all layers
bias_init = keras.initializers.Constant(value = 0.1)

# Input layer with input shape same as before
inputs = Input(shape = input_shape)

# LAYER 1

# 32 2D Convolutional layers of size 5x5 each,
# padded such that the convolutional output remains
# of the same size as input (28x28)
# Outputs a 28x28x32 tensor
x = Conv2D(32, kernel_size = 5, padding = 'same',
           kernel_initializer = weight_init,
           bias_initializer = bias_init)(inputs)
# Apply ReLU activation
x = Activation('relu')(x)
# Maxpooling within a 2x2 grid. Stride specifies the
# no overlap constraint
x = MaxPooling2D(pool_size = 2, strides = 2,
                 padding='same')(x)

# LAYER 2

# 2D Covolutional layer with 64 kernels of size 5x5
# padded such that output dimension is the same as input
# Outputs a 28x28x64 tensor for each image
x = Conv2D(64, kernel_size = 5, padding = 'same',
           kernel_initializer = weight_init,
           bias_initializer = bias_init)(x)
# Apply ReLU activation
x = Activation('relu')(x)
# Maxpool same as in the first layer
x = MaxPooling2D(pool_size = 2, strides = 2,
                 padding='same')(x)

# LAYER 3
```

```

# Flatten to a network before using the fully connected layer
x = Flatten()(x)
# Fully connected layer of 1024 units with ReLU activation
x = Dense(1024, activation = 'relu',
          kernel_initializer = weight_init,
          bias_initializer = bias_init)(x)
# Add a dropout layer with 0.5 prob
x = Dropout(0.5)(x)

# OUTPU LAYER
# Final fully connected layer with units equal to
# number of classes. Output probability distribution
# using softmax
outputs = Dense(num_classes, activation = 'softmax',
          kernel_initializer = weight_init,
          bias_initializer = bias_init)(x)

# Define model
model = Model(inputs = inputs, outputs = outputs)
# Compile model with the loss and optimizer
model.compile(loss = 'categorical_crossentropy',
              optimizer = Adam(lr = lr),
              metrics = ['accuracy'])
model.summary()

# Define log callback for tensorboard
log_dir = "./logs/" + model_name + "/"
tensorboard = TensorBoard(log_dir = log_dir, batch_size = batch_size)

# Define callback to save the best model
checkpoint = ModelCheckpoint(filepath = modelpath,
                             verbose = 1,
                             save_best_only = True)

callbacks = [tensorboard, checkpoint]

# Fit using training data and use validation loss for checkpoint
model.fit(train_data, train_labels,
          batch_size = batch_size, epochs = epochs,
          validation_data = (val_data, val_labels),
          shuffle = True, callbacks = callbacks)

# Evaluate on test data
score = model.evaluate(test_data, test_labels, verbose = 1)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Seed: 478
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_18 (InputLayer)        (None, 28, 28, 1)         0         
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 28, 28, 32)        832       
_________________________________________________________________
activation_7 (Activation)    (None, 28, 28, 32)        0         
```

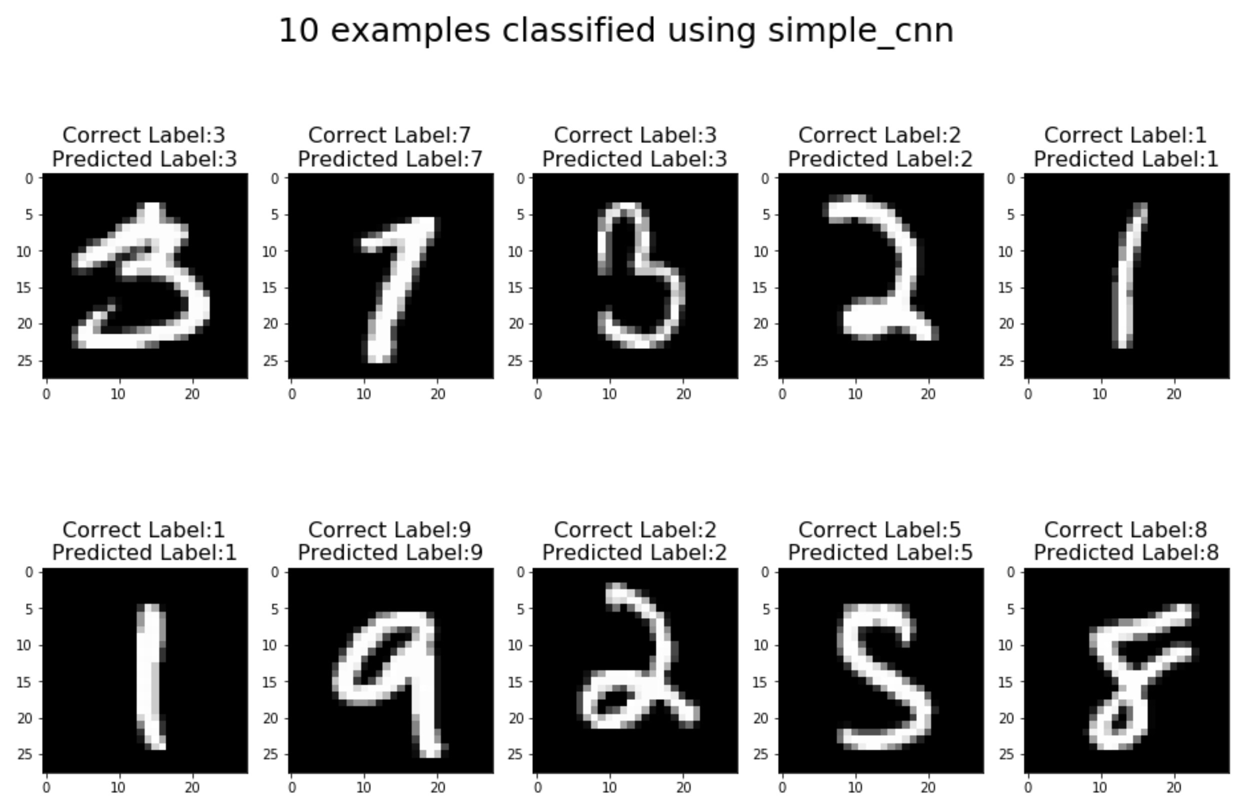
```
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 14, 14, 32)        0         
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 14, 14, 64)        51264     
_________________________________________________________________
activation_8 (Activation)    (None, 14, 14, 64)        0         
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 7, 7, 64)          0         
_________________________________________________________________
flatten_18 (Flatten)         (None, 3136)              0         
_________________________________________________________________
dense_26 (Dense)             (None, 1024)              3212288   
_________________________________________________________________
dropout_4 (Dropout)          (None, 1024)              0         
_________________________________________________________________
dense_27 (Dense)             (None, 10)                10250     
=================================================================
Total params: 3,274,634
Trainable params: 3,274,634
Non-trainable params: 0
_________________________________________________________________
Train on 55000 samples, validate on 5000 samples
Epoch 1/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.9371 - a
cc: 0.8252Epoch 00000: val_loss improved from inf to 0.11519, saving mode
l to saved_models/simple_cnn_model.h5
55000/55000 [==============================] - 357s - loss: 0.9363 - acc:
0.8254 - val_loss: 0.1152 - val_acc: 0.9672
Epoch 2/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.1615 - a
cc: 0.9508Epoch 00001: val_loss improved from 0.11519 to 0.07632, saving
model to saved_models/simple_cnn_model.h5
55000/55000 [==============================] - 359s - loss: 0.1616 - acc:
0.9508 - val_loss: 0.0763 - val_acc: 0.9776
Epoch 3/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.1050 - a
cc: 0.9671Epoch 00002: val_loss improved from 0.07632 to 0.06012, saving
model to saved_models/simple_cnn_model.h5
55000/55000 [==============================] - 412s - loss: 0.1050 - acc:
0.9671 - val_loss: 0.0601 - val_acc: 0.9824
Epoch 4/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.0823 - a
cc: 0.9741Epoch 00003: val_loss improved from 0.06012 to 0.05446, saving
model to saved_models/simple_cnn_model.h5
55000/55000 [==============================] - 360s - loss: 0.0823 - acc:
0.9741 - val_loss: 0.0545 - val_acc: 0.9846
Epoch 5/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.0660 - a
cc: 0.9796Epoch 00004: val_loss improved from 0.05446 to 0.04658, saving
model to saved_models/simple_cnn_model.h5
55000/55000 [==============================] - 332s - loss: 0.0660 - acc:
0.9796 - val_loss: 0.0466 - val_acc: 0.9856
Epoch 6/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.0541 - a
cc: 0.9831Epoch 00005: val_loss improved from 0.04658 to 0.04459, saving
model to saved_models/simple_cnn_model.h5
55000/55000 [==============================] - 333s - loss: 0.0541 - acc:
```

```
0.9831 - val_loss: 0.0446 - val_acc: 0.9864
Epoch 7/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.0430 - a
cc: 0.9859Epoch 00006: val_loss improved from 0.04459 to 0.03979, saving
model to saved_models/simple_cnn_model.h5
55000/55000 [==============================] - 338s - loss: 0.0430 - acc:
0.9859 - val_loss: 0.0398 - val_acc: 0.9882
Epoch 8/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.0378 - a
cc: 0.9877Epoch 00007: val_loss did not improve
55000/55000 [==============================] - 337s - loss: 0.0378 - acc:
0.9877 - val_loss: 0.0408 - val_acc: 0.9878
Epoch 9/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.0308 - a
cc: 0.9902Epoch 00008: val_loss improved from 0.03979 to 0.03794, saving
model to saved_models/simple_cnn_model.h5
55000/55000 [==============================] - 329s - loss: 0.0308 - acc:
0.9902 - val_loss: 0.0379 - val_acc: 0.9898
Epoch 10/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.0263 - a
cc: 0.9916Epoch 00009: val_loss improved from 0.03794 to 0.03487, saving
model to saved_models/simple_cnn_model.h5
55000/55000 [==============================] - 335s - loss: 0.0263 - acc:
0.9916 - val_loss: 0.0349 - val_acc: 0.9904
Epoch 11/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.0236 - a
cc: 0.9924Epoch 00010: val_loss did not improve
55000/55000 [==============================] - 375s - loss: 0.0236 - acc:
0.9924 - val_loss: 0.0371 - val_acc: 0.9902
Epoch 12/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.0202 - a
cc: 0.9934Epoch 00011: val_loss did not improve
55000/55000 [==============================] - 327s - loss: 0.0202 - acc:
0.9935 - val_loss: 0.0350 - val_acc: 0.9922
Epoch 13/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.0168 - a
cc: 0.9942Epoch 00012: val_loss did not improve
55000/55000 [==============================] - 328s - loss: 0.0168 - acc:
0.9942 - val_loss: 0.0381 - val_acc: 0.9902
Epoch 14/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.0163 - a
cc: 0.9944Epoch 00013: val_loss improved from 0.03487 to 0.03430, saving
model to saved_models/simple_cnn_model.h5
55000/55000 [==============================] - 329s - loss: 0.0162 - acc:
0.9944 - val_loss: 0.0343 - val_acc: 0.9910
Epoch 15/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.0126 - a
cc: 0.9957Epoch 00014: val_loss improved from 0.03430 to 0.03344, saving
model to saved_models/simple_cnn_model.h5
55000/55000 [==============================] - 331s - loss: 0.0126 - acc:
0.9957 - val_loss: 0.0334 - val_acc: 0.9914
Epoch 16/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.0123 - a
cc: 0.9961Epoch 00015: val_loss did not improve
55000/55000 [==============================] - 332s - loss: 0.0124 - acc:
0.9961 - val_loss: 0.0338 - val_acc: 0.9920
Epoch 17/18
```

```
54950/55000 [============================>.] - ETA: 0s - loss: 0.0106 - a
cc: 0.9966Epoch 00016: val_loss did not improve
55000/55000 [==============================] - 328s - loss: 0.0106 - acc:
0.9966 - val_loss: 0.0358 - val_acc: 0.9912
Epoch 18/18
54950/55000 [============================>.] - ETA: 0s - loss: 0.0096 - a
cc: 0.9970Epoch 00017: val_loss did not improve
55000/55000 [==============================] - 330s - loss: 0.0096 - acc:
0.9970 - val_loss: 0.0379 - val_acc: 0.9918
 9984/10000 [============================>.] - ETA: 0sTest loss: 0.026654
3549486
Test accuracy: 0.9916
```

Simple CNN: Plot 10 random examples with correct and predicted label

In [64]:
```
import matplotlib.pyplot as plt

save_dir = "saved_models/"
model_name = "simple_cnn"
modelpath = os.path.join(save_dir, model_name + "_model.h5")

model = load_model(modelpath)

fig, ax = plt.subplots(2, 5, figsize = (16, 10))
plt.suptitle("10 examples classified using " + model_name,
             fontsize = 25)
for i in range(2):
    for j in range(5):
        sample = mnist.test.next_batch(1)
        image = sample[0].reshape(rows, cols)
        label = sample[1]
        pred_label = model.predict(image.reshape(-1, rows, cols, 1))
        label = np.argmax(label)
        pred_label = np.argmax(pred_label)
        ax[i][j].imshow(image, cmap = 'gray')
        ax[i][j].set_title("Correct Label:" + str(label) +
                           "\nPredicted Label:" + str(pred_label),
                           fontsize = 16)

plt.show()
```



Load Softmax Regression Model and evaluate

In [65]:
```
save_dir = "saved_models/"
model_name = "softmax_regr"
modelpath = os.path.join(save_dir, model_name + "_model.h5")

model = load_model(modelpath)
score = model.evaluate(test_data, test_labels)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
10000/10000 [==============================] - 0s
Test loss: 0.2704395448
Test accuracy: 0.9244
```

Load Softmax Multilayer Perceptron Model and evaluate

In [66]:
```
save_dir = "saved_models/"
model_name = "softmax_mlp"
modelpath = os.path.join(save_dir, model_name + "_model.h5")

model = load_model(modelpath)
score = model.evaluate(test_data, test_labels)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
 9728/10000 [============================>.] - ETA: 0sTest loss: 0.073113
9129666
Test accuracy: 0.9781
```

Load Simple CNN (LeNet) model and evaluate

In [67]:
```
save_dir = "saved_models/"
model_name = "simple_cnn"
modelpath = os.path.join(save_dir, model_name + "_model.h5")

model = load_model(modelpath)
score = model.evaluate(test_data, test_labels)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
 9984/10000 [============================>.] - ETA: 0s ETest loss: 0.0256
009992089
Test accuracy: 0.991
```