Machine Learning - Neural Network

0.0.1 Yifan Xu

0.0.2 December 7, 2018

0.1 Machine Learning

In this problem, you will implement several machine learning solutions for computer vision problems.

0.1.1 Part 1: Initial setup

Follow the directions on https://www.tensorflow.org/install/ to install Tensorflow on your computer. If you are using the Anaconda distribution for python, you can check out https://www.anaconda.com/blog/developer-blog/tensorflow-in-anaconda/.

Note: You will not need GPU support for this assignment so don't worry if you don't have one. Furthermore, installing with GPU support is often more difficult to configure so it is suggested that you install the CPU only version.

Run the tensorflow hello world snippet below to verify your instalation.

Download the MNIST data from http://yann.lecun.com/exdb/mnist/.

Download the 4 zipped files, extract them into one folder, and change the variable 'path' in the code below. (Code taken from https://gist.github.com/akesling/5358964)

Plot one random example image corresponding to each label from training data.

```
In [7]: import tensorflow as tf
        hello = tf.constant('Hello, TensorFlow!')
        sess = tf.Session()
        print(sess.run(hello))
```

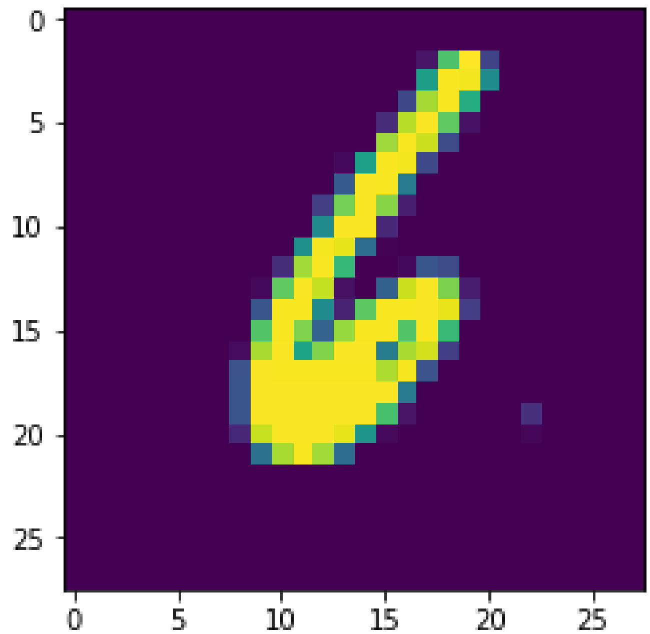
```
b'Hello, TensorFlow!'
```

```
In [8]: import os
        import struct
        import numpy as np
        import matplotlib.pyplot as plt

        # Change path as required
        path = "./mnist_data/"
```

```
def read(dataset = "training", datatype='images'):
    """
    Python function for importing the MNIST data set.  It returns an iterator
    of 2-tuples with the first element being the label and the second element
    being a numpy.uint8 2D array of pixel data for the given image.
    """

    if dataset is "training":
        fname_img = os.path.join(path, 'train-images.idx3-ubyte')
        fname_lbl = os.path.join(path, 'train-labels.idx1-ubyte')
    elif dataset is "testing":
        fname_img = os.path.join(path, 't10k-images.idx3-ubyte')
        fname_lbl = os.path.join(path, 't10k-labels.idx1-ubyte')

    # Load everything in some numpy arrays
    with open(fname_lbl, 'rb') as flbl:
        magic, num = struct.unpack(">II", flbl.read(8))
        lbl = np.fromfile(flbl, dtype=np.int8)

    with open(fname_img, 'rb') as fimg:
        magic, num, rows, cols = struct.unpack(">IIII", fimg.read(16))
        img = np.fromfile(fimg, dtype=np.uint8).reshape(len(lbl), rows, cols)

    if(datatype=='images'):
        get_data = lambda idx: img[idx]
    elif(datatype=='labels'):
        get_data = lambda idx: lbl[idx]

    # Create an iterator which returns each image in turn
    for i in range(len(lbl)):
        yield get_data(i)

trainData=np.array(list(read('training','images')))
trainLabels=np.array(list(read('training','labels')))
testData=np.array(list(read('testing','images')))
testLabels=np.array(list(read('testing','labels')))
```

In [20]:
```
import random
num = random.randint(0,trainData.shape[0]-1)
plt.figure
plt.imshow(trainData[num,...])
plt.show
print('label:',trainLabels[num])
```

```
label: 6
```



Some helper functions are given below.

```
In [21]: # a generator for batches of data
         # yields data (batchsize, 3, 32, 32) and labels (batchsize)
         # if shuffle, it will load batches in a random order
         def DataBatch(data, label, batchsize, shuffle=True):
             n = data.shape[0]
             if shuffle:
                 index = np.random.permutation(n)
             else:
                 index = np.arange(n)
             for i in range(int(np.ceil(n/batchsize))):
                 inds = index[i*batchsize : min(n,(i+1)*batchsize)]
                 yield data[inds], label[inds]

         # tests the accuracy of a classifier
         def test(testData, testLabels, classifier):
             batchsize=50
             correct=0.
             for data,label in DataBatch(testData,testLabels,batchsize,shuffle=False):
                 prediction = classifier(data)
                 correct += np.sum(prediction==label)
             return correct/testData.shape[0]*100

         # a sample classifier
         # given an input it outputs a random class
```

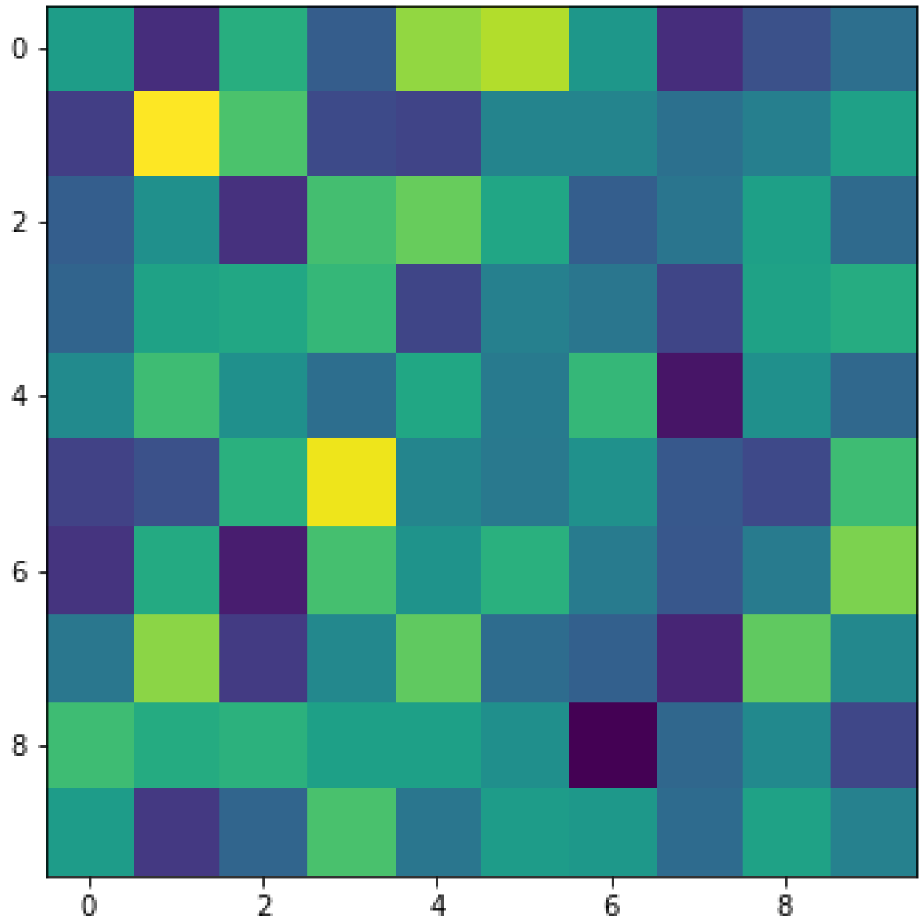
```
class RandomClassifier():
    def __init__(self, classes=10):
        self.classes=classes
    def __call__(self, x):
        return np.random.randint(self.classes, size=x.shape[0])

randomClassifier = RandomClassifier()
print('Random classifier accuracy: %f' %
      test(testData, testLabels, randomClassifier))
```

```
Random classifier accuracy: 10.600000
```

0.1.2 Part 2: Confusion Matrix

Here you will implement a function that computes the confusion matrix for a classifier. The matrix (M) should be nxn where n is the number of classes. Entry M[i,j] should contain the fraction of images of class i that was classified as class j.

```
In [22]: # Using the tqdm module to visualize run time is suggested
         # from tqdm import tqdm

         # It would be a good idea to return the accuracy, along with the confusion
         # matrix, since both can be calculated in one iteration over test data, to
         # save time
         def Confusion(testData, testLabels, classifier):
             '''
             Your code here
             '''
             n = len(set(testLabels.flatten()))
             M = np.zeros((n,n))
             for data,label in DataBatch(testData,testLabels,1):
                 prediction = classifier(data)
                 M[int(label), int(prediction)] += 1

             M = M/np.array([np.sum(M,1)]).T
             return M

         def VisualizeConfusion(M):
             plt.figure(figsize=(14, 6))
             plt.imshow(M)
             plt.show()
             print(np.round(M,2))

         M = Confusion(testData, testLabels, randomClassifier)
         VisualizeConfusion(M)
```




```
[[0.1  0.09 0.11 0.09 0.12 0.12 0.1  0.09 0.09 0.1 ]
 [0.09 0.12 0.11 0.09 0.09 0.1  0.1  0.1  0.1  0.1 ]
 [0.09 0.1  0.09 0.11 0.11 0.11 0.09 0.1  0.1  0.09]
 [0.09 0.1  0.11 0.11 0.09 0.1  0.1  0.09 0.1  0.11]
 [0.1  0.11 0.1  0.1  0.11 0.1  0.11 0.08 0.1  0.09]
 [0.09 0.09 0.11 0.12 0.1  0.1  0.1  0.09 0.09 0.11]
 [0.09 0.11 0.08 0.11 0.1  0.11 0.1  0.09 0.1  0.11]
 [0.1  0.12 0.09 0.1  0.11 0.1  0.09 0.08 0.11 0.1 ]
 [0.11 0.11 0.11 0.1  0.1  0.1  0.08 0.09 0.1  0.09]
 [0.1  0.09 0.09 0.11 0.1  0.1  0.1  0.1  0.11 0.1 ]]
```

0.1.3 Part 3: K-Nearest Neighbors (KNN)

- Here you will implement a simple knn classifier. The distance metric is Euclidean in pixel space. k refers to the number of neighbors involved in voting on the class, and should be 3. You are allowed to use sklearn.neighbors.KNeighborsClassifier.

- Display confusion matrix and accuracy for your KNN classifier trained on the entire train set. (should be ~97 %)
- After evaluating the classifier on the testset, based on the confusion matrix, mention the number that the number '4' is most often predicted to be, other than '4'.

In [23]:
```
from sklearn.neighbors import KNeighborsClassifier
class KNNClassifer():
    def __init__(self, k=3):
        # k is the number of neighbors involved in voting
        '''
        your code here
        '''
        self.k = k

    def train(self, trainData, trainLabels):
        '''
        your code here
        '''
        data = trainData.reshape((trainData.shape[0], trainData.shape[1]*trainData.sha
        self.neigh = KNeighborsClassifier(n_neighbors=self.k,weights='distance')
        self.neigh.fit(data, trainLabels)


    def __call__(self, x):
        # this method should take a batch of images
        # and return a batch of predictions
        '''
        your code here
        '''
        X = x.reshape((x.shape[0], x.shape[1]*x.shape[2]))
        predictions = self.neigh.predict(X)
        return predictions



# test your classifier with only the first 100 training examples (use this
# while debugging)
# note you should get ~ 65 % accuracy
knnClassiferX = KNNClassifer()
knnClassiferX.train(trainData[:100], trainLabels[:100])
print ('KNN classifier accuracy: %f'%test(testData, testLabels, knnClassiferX))
```

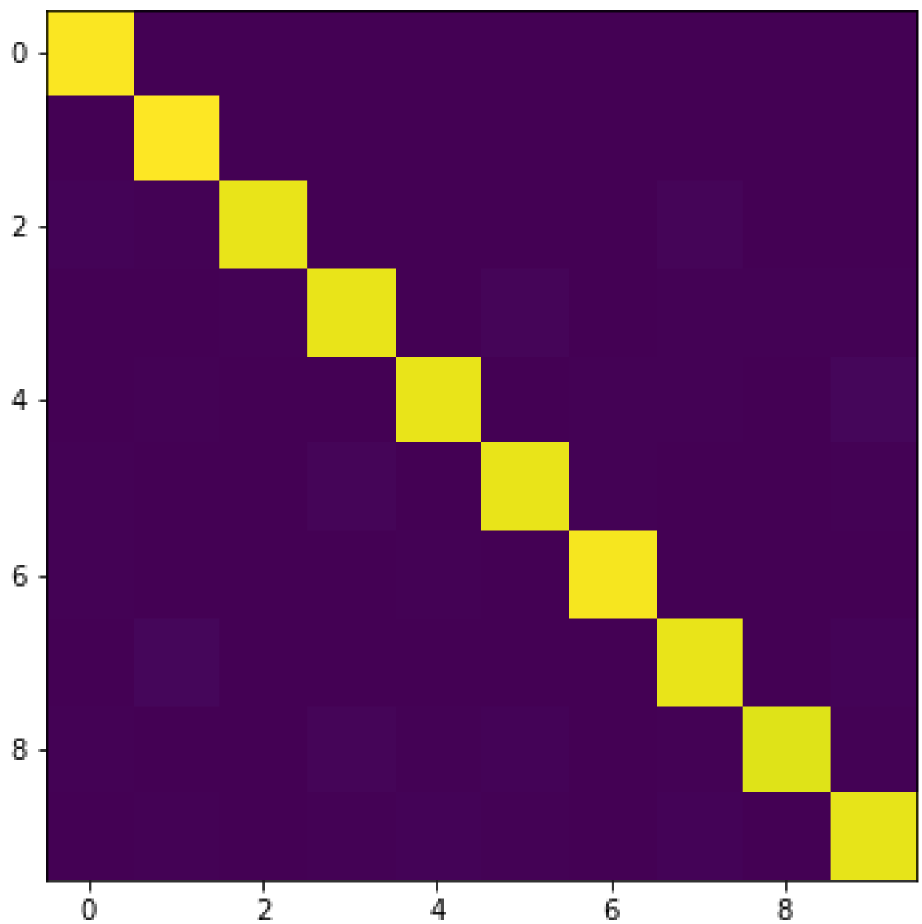
```
KNN classifier accuracy: 66.940000
```

In [24]:
```
import time
# test your classifier with all the training examples (This may take a while)
start = time.time()
```

```
knnClassifer = KNNClassifer()
knnClassifer.train(trainData, trainLabels)
T = time.time() - start
print("Training time: ", T)
```

```
Training time:  18.103584051132202
```

In [25]:
```
# display confusion matrix for your KNN classifier with all the training examples
start = time.time()
M = Confusion(testData, testLabels, knnClassifer)
VisualizeConfusion(M)
T = time.time() - start
print("Testing time: ", T)
```




```
[[0.99 0.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   1.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.01 0.01 0.97 0.   0.   0.   0.   0.01 0.   0.  ]
```

```
 [0.   0.   0.   0.97 0.   0.01 0.   0.01 0.   0.  ]
 [0.   0.01 0.   0.   0.97 0.   0.01 0.   0.   0.02]
 [0.   0.   0.   0.01 0.   0.96 0.01 0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.99 0.   0.   0.  ]
 [0.   0.02 0.   0.   0.   0.   0.   0.97 0.   0.01]
 [0.01 0.   0.   0.01 0.01 0.01 0.   0.   0.95 0.01]
 [0.   0.   0.   0.01 0.01 0.   0.   0.01 0.   0.96]]
Testing time:  691.3477680683136
```

```
In [26]: print ('KNN classifier accuracy: %f'%test(testData, testLabels, knnClassifer))
```

```
KNN classifier accuracy: 97.170000
```

Other than '4', number '4' is most often predicted to be '6' and '8'.

0.1.4 Part 4: Principal Component Analysis (PCA) K-Nearest Neighbors (KNN)

Here you will implement a simple KNN classifer in PCA space (for k=3 and 25 principal components). You should implement PCA yourself using svd (you may not use sklearn.decomposition.PCA or any other package that directly implements PCA transformations

Is the testing time for PCA KNN classifier more or less than that for KNN classifier? Comment on why it differs if it does.

```
In [27]: from scipy.linalg import svd
         class PCAKNNClassifer():
             def __init__(self, components=25, k=3):
                 # components = number of principal components
                 # k is the number of neighbors involved in voting
                 '''
                 your code here
                 '''
                 self.components = components
                 self.k = k

             def train(self, trainData, trainLabels):
                 '''
                 your code here
                 '''
                 data = trainData.reshape((trainData.shape[0], trainData.shape[1]*trainData.sha
                 self.mean = np.mean(data, axis=0)
                 data_ave = data - self.mean

                 dataMatrix = np.zeros((data[0].size, data[0].size), np.float)
                 for i in range(trainData.shape[0]):
                     fai = np.reshape(data_ave[i], (data[0].size, 1))
                     dataMatrix += np.dot(fai, fai.T)/trainData.shape[0]
                 U, S, Vh = np.linalg.svd(dataMatrix)
```

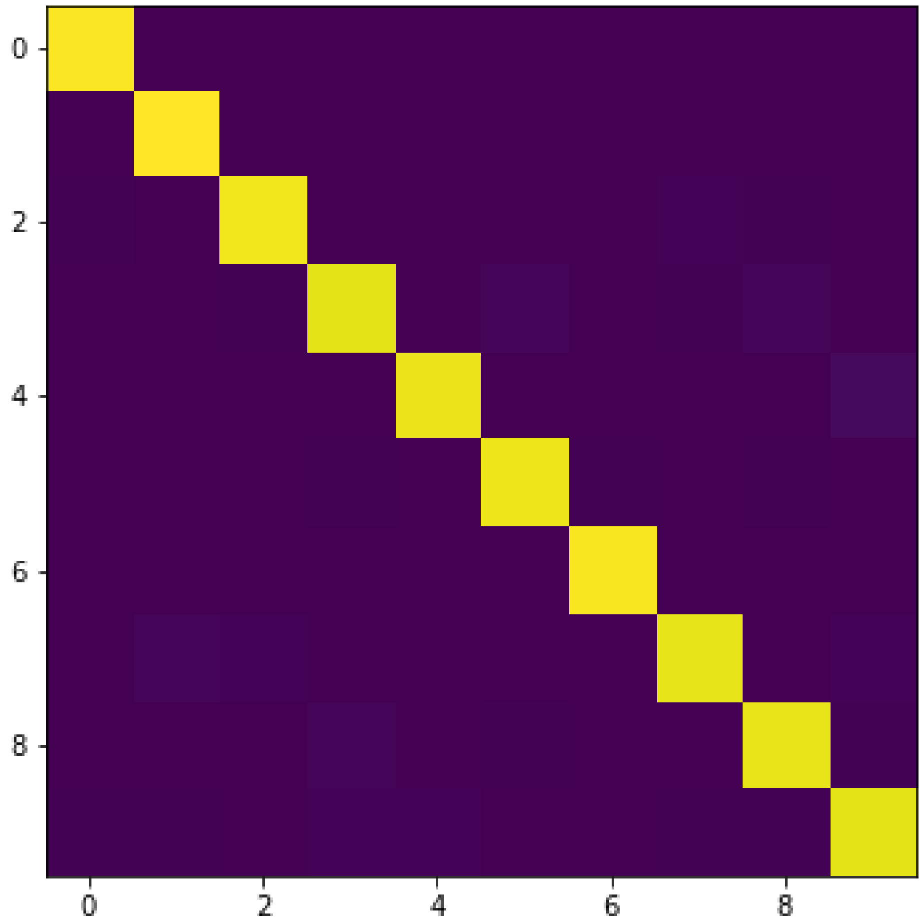
```
            self.pca = U[:, :self.components]

            self.D = np.dot(data, self.pca)
            self.neigh = KNeighborsClassifier(n_neighbors=self.k,weights='distance')
            self.neigh.fit(self.D, trainLabels)

        def __call__(self, x):
            # this method should take a batch of images
            # and return a batch of predictions
            '''
            your code here
            '''

            X = np.reshape(x, (x.shape[0], -1))
            Xpca = np.dot(X, self.pca)
            result = self.neigh.predict(Xpca)
            return result


    # test your classifier with only the first 100 training examples (use this
    # while debugging)
    pcaknnClassiferX = PCAKNNClassifer()
    pcaknnClassiferX.train(trainData[:100], trainLabels[:100])
    print ('pcaKNN classifier accuracy: %f'%test(testData, testLabels, pcaknnClassiferX))
```

```
pcaKNN classifier accuracy: 68.240000
```

In [28]:
```
# test your classifier with all the training examples (This may take a while)
start = time.time()
pcaknnClassifer = PCAKNNClassifer()
pcaknnClassifer.train(trainData, trainLabels)
T = time.time() - start
print("Training time: ", T)
```

```
Training time:  82.35483503341675
```

In [29]:
```
import time
# display confusion matrix for your PCA KNN classifier with all the training examples
start = time.time()
M = Confusion(testData, testLabels, pcaknnClassifer)
VisualizeConfusion(M)
T = time.time() - start
print("Testing time: ", T)
```




```
[[0.99 0.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   1.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.98 0.   0.   0.   0.   0.01 0.   0.  ]
 [0.   0.   0.   0.96 0.   0.02 0.   0.01 0.01 0.  ]
 [0.   0.   0.   0.   0.97 0.   0.   0.   0.   0.03]
 [0.   0.   0.   0.01 0.   0.97 0.01 0.   0.01 0.  ]
 [0.   0.   0.   0.   0.   0.   0.99 0.   0.   0.  ]
 [0.   0.02 0.01 0.   0.   0.   0.   0.96 0.   0.01]
 [0.   0.   0.   0.01 0.   0.01 0.   0.   0.97 0.  ]
 [0.   0.01 0.   0.01 0.01 0.   0.   0.01 0.   0.96]]
Testing time:  20.27758812904358
```

```
In [30]: print ('pcaKNN classifier accuracy: %f'%test(testData, testLabels, pcaknnClassifer))
```

```
pcaKNN classifier accuracy: 97.400000
```

The testing time for PCA KNN classifier is less than that for KNN classifier. That's because the pcaKnn classifier provides a sequence of best linear approximations to a original high-dimensional data and reduce the dimension which reduce some noise and the computing time.

0.2 Problem 3: Deep learning

Below is some helper code to train your deep networks. You can look at https://www.tensorflow.org/get_started/mnist/beginners for reference.

```
In [104]: # base class for your Tensorflow networks. It implements the training loop
          # (train) and prediction(__call__)  for you.
          # You will need to implement the __init__ function to define the networks
          # structures in the following problems.

          class TFClassifier():
              def __init__(self):
                  pass

              def train(self, trainData, trainLabels, epochs=1, batchsize=50):
                  self.prediction = tf.argmax(self.y,1)
                  self.cross_entropy = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_l
                  self.train_step = tf.train.AdamOptimizer(1e-4).minimize(self.cross_entropy)
                  self.correct_prediction = tf.equal(self.prediction, self.y_)
                  self.accuracy = tf.reduce_mean(tf.cast(self.correct_prediction, tf.float32))
                  self.sess.run(tf.global_variables_initializer())


                  for epoch in range(epochs):
                      for i, (data,label) in enumerate(DataBatch(trainData, trainLabels, batchs
                          data=np.expand_dims(data,-1)
                          _, acc = self.sess.run([self.train_step, self.accuracy], feed_dict={s

                      print ('Epoch:%d Accuracy: %f'%(epoch+1, test(testData, testLabels, self)


              def __call__(self, x):
                  return self.sess.run(self.prediction, feed_dict={self.x: np.expand_dims(x,-1)

              def get_first_layer_weights(self):
                  return self.sess.run(self.weights[0])

          # helper function to get weight variable
          def weight_variable(shape):
              initial = tf.truncated_normal(shape, stddev=0.01)
              return tf.Variable(initial)

          # helper function to get bias variable
          def bias_variable(shape):
```

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

        # example linear classifier
        class LinearClassifier(TFClassifier):
            def __init__(self, classes=10):
                self.sess = tf.Session()

                self.x = tf.placeholder(tf.float32, shape=[None,28,28,1]) # input batch of i
                self.y_ = tf.placeholder(tf.int64, shape=[None]) # input labels

                # model variables
                self.weights = [weight_variable([28*28,classes])]
                print(self.weights)
                self.biases = [bias_variable([classes])]

                # linear operation
                self.y = tf.matmul(tf.reshape(self.x,(-1,28*28*1)),self.weights[0]) + self.b



        # test the example linear classifier (note you should get around 90% accuracy
        # for 10 epochs and batchsize 50)
        linearClassifier = LinearClassifier()
        linearClassifier.train(trainData, trainLabels, epochs=10)
```

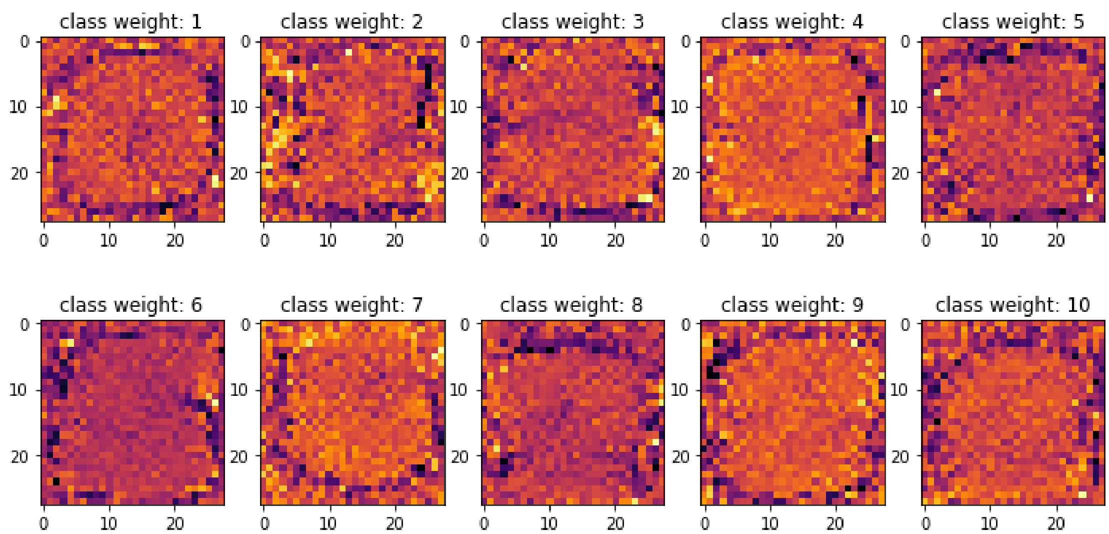
```
[<tf.Variable 'Variable_89:0' shape=(784, 10) dtype=float32_ref>]
Epoch:1 Accuracy: 88.510000
Epoch:2 Accuracy: 89.180000
Epoch:3 Accuracy: 88.610000
Epoch:4 Accuracy: 89.730000
Epoch:5 Accuracy: 90.020000
Epoch:6 Accuracy: 90.210000
Epoch:7 Accuracy: 90.440000
Epoch:8 Accuracy: 87.920000
Epoch:9 Accuracy: 89.960000
Epoch:10 Accuracy: 90.330000
```

0.2.1 Part 1: Single Layer Perceptron

The simple linear classifier implemented in the cell already performs quite well. Plot the filter weights corresponding to each output class (weights, not biases) as images. (Normalize weights to lie between 0 and 1 and use color maps like 'inferno' or 'plasma' for good results). Comment on what the weights look like and why that may be so.

```
In [105]: with linearClassifier.sess as sess:
              weight = linearClassifier.weights[0]
```

```
weight = tf.reshape(weight, (28, 28, 10)).eval()
weight = weight - np.min(weight)
weight = weight / np.max(weight)
plt.figure(figsize=(12, 6))
for i in range(10):
    ax = plt.subplot(2,5,i+1)
    plt.imshow(weight[:,:,i], cmap='inferno',)
    ax.set_title('class weight: {}'.format(i+1))
plt.show()
```



The weights look like that there is a circle around. This is because this is a single layer perceptron which makes its predictions based on a linear predictor function combining the weights with the feature vector. The number is always in the center of the picture. So, the weights look like these pictures.

0.2.2 Part 2: Multi Layer Perceptron (MLP)

Here you will implement an MLP. The MLP shoud consist of 2 layers (matrix multiplication and bias offset) that map to the following feature dimensions:

- 28x28 -> hidden (100)
- hidden -> classes
- The hidden layer should be followed with a ReLU nonlinearity. The final layer should not have a nonlinearity applied as we desire the raw logits output.
- The final output of the computation graph should be stored in self.y as that will be used in the training.

Display the confusion matrix and accuracy after training. Note: You should get ~ 97 % accuracy for 10 epochs and batch size 50.

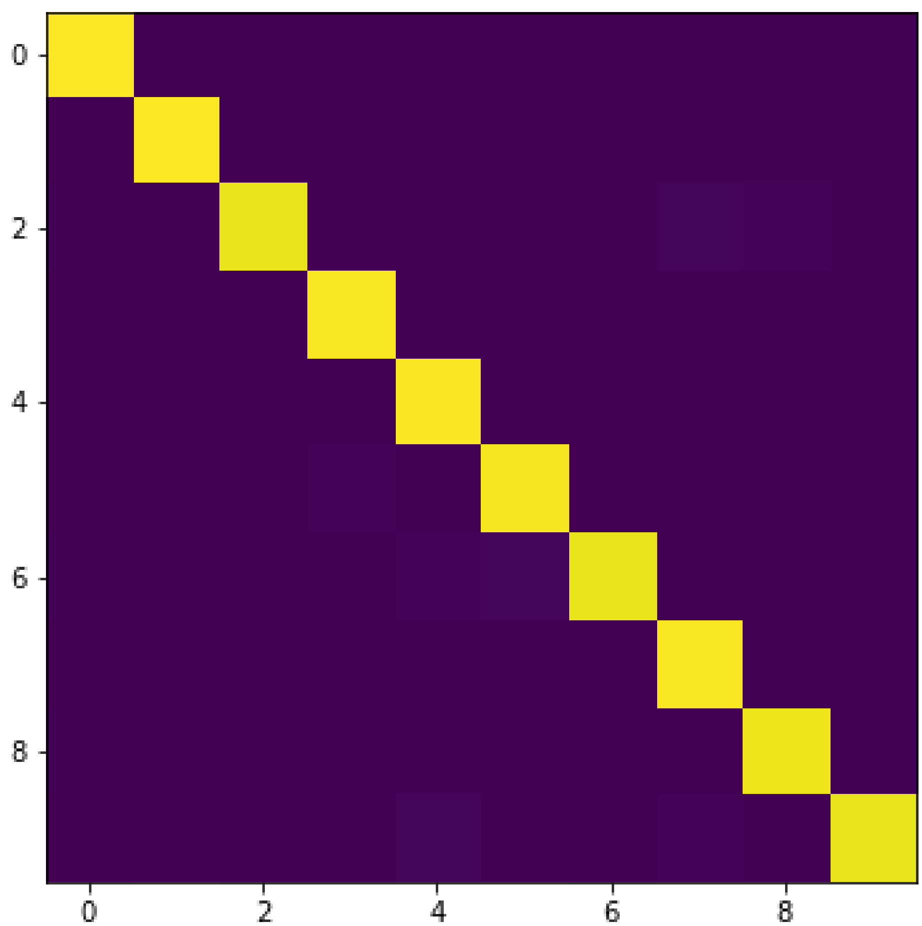
Plot the filter weights corresponding to the mapping from the inputs to the first 10 hidden layer outputs (out of 100). Do the weights look similar to the weights plotted in the previous problem? Why or why not?

```
In [132]: class MLPClassifer(TFClassifier):
              def __init__(self, classes=10, hidden=100):
                  '''
                  your code here
                  '''
                  self.sess = tf.Session()

                  self.x = tf.placeholder(tf.float32, shape=[None,28,28,1]) # input batch of i
                  self.y_ = tf.placeholder(tf.int64, shape=[None]) # input labels

                  # model variables
                  self.weights1 = [weight_variable([28*28,hidden])]
                  self.biases1 = [bias_variable([hidden])]
                  self.h = tf.matmul(tf.reshape(self.x,(-1,28*28*1)),self.weights1[0]) + self.
                  self.hout = tf.nn.relu(self.h)

                  self.weights2 = [weight_variable([hidden,classes])]
                  print(self.weights2)
                  self.biases2 = [bias_variable([classes])]
                  self.y = tf.matmul(self.hout,self.weights2[0]) + self.biases2[0]


              def train(self, trainData, trainLabels, epochs=1, batchsize=50):
                  self.prediction = tf.argmax(self.y,1)
                  self.cross_entropy = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_
                  self.train_step = tf.train.AdamOptimizer(1e-4).minimize(self.cross_entropy)
                  self.correct_prediction = tf.equal(self.prediction, self.y_)
                  self.accuracy = tf.reduce_mean(tf.cast(self.correct_prediction, tf.float32))
                  self.sess.run(tf.global_variables_initializer())


                  for epoch in range(epochs):
                      for i, (data,label) in enumerate(DataBatch(trainData, trainLabels, batch
                          data=np.expand_dims(data,-1)
                          _, acc = self.sess.run([self.train_step, self.accuracy], feed_dict={

                      print ('Epoch:%d Accuracy: %f'%(epoch+1, test(testData, testLabels, self

In [133]: mlpClassifer = MLPClassifer()
          mlpClassifer.train(trainData, trainLabels, epochs=10)
```

```
[<tf.Variable 'Variable_153:0' shape=(100, 10) dtype=float32_ref>]
Epoch:1 Accuracy: 95.330000
```

```
Epoch:2 Accuracy: 96.560000
Epoch:3 Accuracy: 97.040000
Epoch:4 Accuracy: 97.250000
Epoch:5 Accuracy: 97.530000
Epoch:6 Accuracy: 97.610000
Epoch:7 Accuracy: 97.150000
Epoch:8 Accuracy: 97.600000
Epoch:9 Accuracy: 97.740000
Epoch:10 Accuracy: 97.550000
```

In [134]:
```
M = Confusion(testData, testLabels, mlpClassifer)
VisualizeConfusion(M)
```



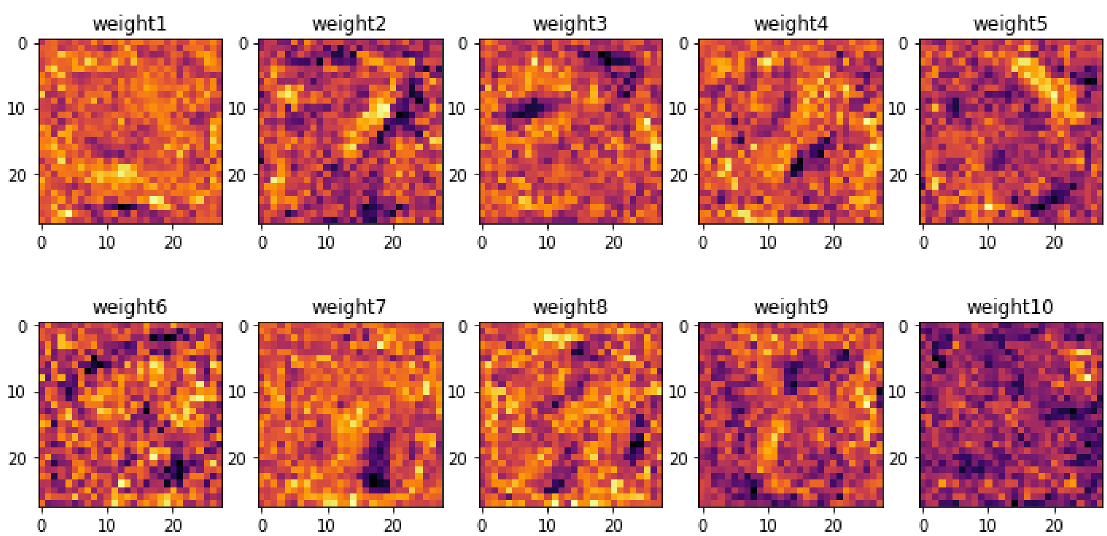

```
[[0.99 0.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.99 0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.01 0.   0.96 0.01 0.   0.   0.   0.01 0.01 0.  ]
 [0.   0.   0.   0.98 0.   0.   0.   0.   0.   0.  ]
```

```
[0.   0.   0.   0.   0.99 0.   0.   0.   0.   0.01]
[0.   0.   0.   0.01 0.   0.98 0.   0.   0.   0.  ]
[0.01 0.   0.   0.   0.01 0.01 0.96 0.   0.   0.  ]
[0.   0.   0.01 0.   0.   0.   0.   0.98 0.   0.  ]
[0.01 0.   0.01 0.   0.01 0.   0.   0.   0.97 0.  ]
[0.   0.   0.   0.   0.02 0.   0.   0.01 0.   0.95]]
```

```
In [135]: print ('mlp Classifer accuracy: %f'%test(testData, testLabels, mlpClassifer))
```

```
mlp Classifer accuracy: 97.550000
```

```
In [136]: with mlpClassifer.sess as sess:
              weight = mlpClassifer.weights1[0]
              weight = tf.reshape(weight, (28, 28, 100)).eval()
              weight = weight - np.min(weight)
              weight = weight / np.max(weight)
              plt.figure(figsize=(12, 6))
              for i in range(10):
                  ax = plt.subplot(2,5,i+1)
                  plt.imshow(weight[:,:,i], cmap='inferno',)
                  ax.set_title('weight{}'.format(i+1))
              plt.show()
```



The weights look different with the weights plotted in the previous problem. Since this is a multi layer perceptron, not like the single layer perceptron, its hidden layer shows features can't be directly observed.

0.2.3 Part 3: Convolutional Neural Network (CNN)

Here you will implement a CNN with the following architecture:

- n=5
- ReLU(Conv(kernel_size=4x4, stride=2, output_features=n))
- ReLU(Conv(kernel_size=4x4, stride=2, output_features=n*2))
- ReLU(Conv(kernel_size=4x4, stride=2, output_features=n*4))
- Linear(output_features=classes)

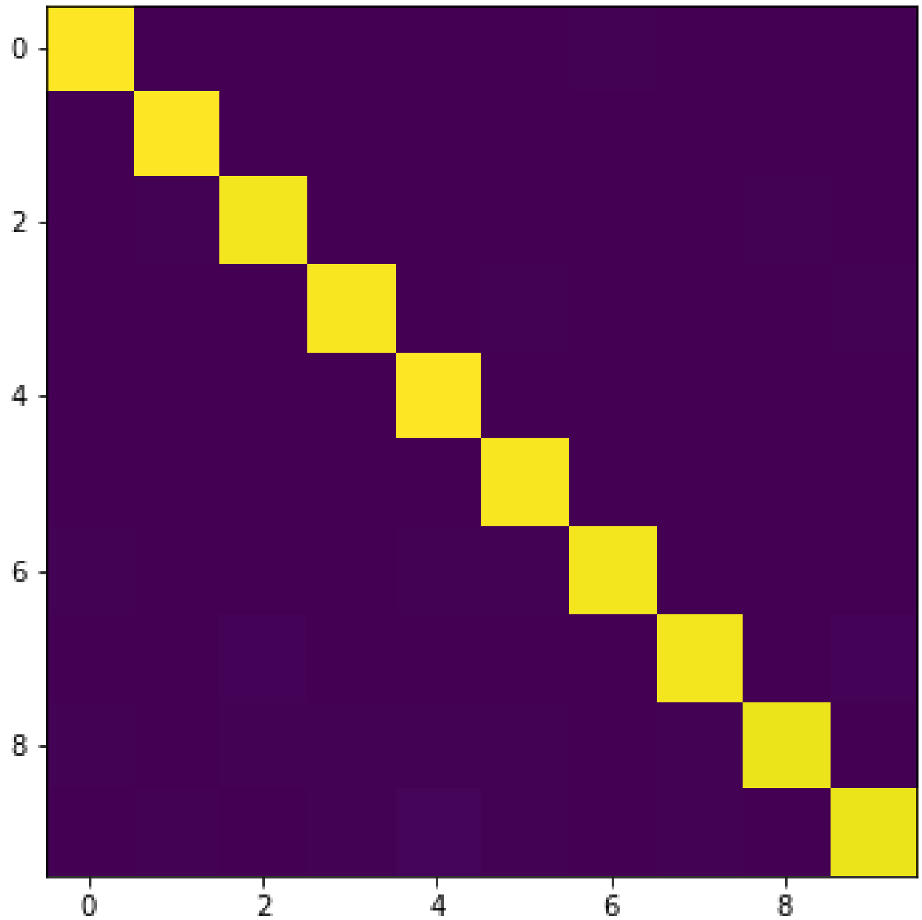
Display the confusion matrix and accuracy after training. You should get around ~ 98 % accuracy for 10 epochs and batch size 50.

```
In [155]: def conv2d(x, W, stride=2):
              return tf.nn.conv2d(x, W, strides=[1, stride, stride, 1], padding='SAME')

          class CNNClassifer(TFClassifier):
              def __init__(self, classes=10, n=5):
                  '''
                  your code here
                  '''
                  self.sess = tf.Session()

                  self.x = tf.placeholder(tf.float32, shape=[None,28,28,1]) # input batch of i
                  self.y_ = tf.placeholder(tf.int64, shape=[None]) # input labels

                  # model variables
                  self.Wconv1 = weight_variable([4, 4, 1, n])
                  self.bconv1 = bias_variable([n])
                  self.hconv1 = tf.nn.relu(conv2d(tf.reshape(self.x,(-1,28,28,1)), self.Wconv1

                  self.Wconv2 = weight_variable([4, 4, n, n*2])
                  self.bconv2 = bias_variable([n*2])
                  self.hconv2 = tf.nn.relu(conv2d(self.hconv1, self.Wconv2) + self.bconv2)

                  self.Wconv3 = weight_variable([4, 4, n*2, n*4])
                  self.bconv3 = bias_variable([n*4])
                  self.hconv3 = tf.nn.relu(conv2d(self.hconv2, self.Wconv3) + self.bconv3)

                  self.weights = [weight_variable([4*4*n*4,classes])]
                  self.biases = [bias_variable([classes])]
                  self.y = tf.matmul(tf.reshape(self.hconv3,(-1,4*4*n*4)),self.weights[0]) + s


              def train(self, trainData, trainLabels, epochs=1, batchsize=50):
                  self.prediction = tf.argmax(self.y,1)
                  self.cross_entropy = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_
                  self.train_step = tf.train.AdamOptimizer(1e-4).minimize(self.cross_entropy)
                  self.correct_prediction = tf.equal(self.prediction, self.y_)
                  self.accuracy = tf.reduce_mean(tf.cast(self.correct_prediction, tf.float32))
                  self.sess.run(tf.global_variables_initializer())
```

```
            for epoch in range(epochs):
                for i, (data,label) in enumerate(DataBatch(trainData, trainLabels, batchs
                    data=np.expand_dims(data,-1)
                    _, acc = self.sess.run([self.train_step, self.accuracy], feed_dict={s

                print ('Epoch:%d Accuracy: %f'%(epoch+1, test(testData, testLabels, self)


cnnClassifer = CNNClassifer()
cnnClassifer.train(trainData, trainLabels, epochs=10)
```

```
Epoch:1 Accuracy: 92.620000
Epoch:2 Accuracy: 95.270000
Epoch:3 Accuracy: 96.540000
Epoch:4 Accuracy: 97.180000
Epoch:5 Accuracy: 97.130000
Epoch:6 Accuracy: 97.550000
Epoch:7 Accuracy: 97.790000
Epoch:8 Accuracy: 97.980000
Epoch:9 Accuracy: 97.710000
Epoch:10 Accuracy: 98.030000
```

```
In [156]: M = Confusion(testData, testLabels, cnnClassifer)
          VisualizeConfusion(M)
```




```
[[0.99 0.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.99 0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.01 0.98 0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.98 0.   0.01 0.   0.   0.   0.01]
 [0.   0.   0.   0.   0.99 0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.99 0.   0.   0.   0.  ]
 [0.01 0.   0.   0.   0.01 0.01 0.97 0.   0.   0.  ]
 [0.   0.   0.01 0.   0.   0.   0.   0.97 0.   0.01]
 [0.01 0.   0.   0.01 0.   0.   0.   0.01 0.97 0.  ]
 [0.   0.   0.   0.   0.01 0.   0.   0.   0.   0.96]]
```

```
In [157]: print ('CNN Classifier accuracy: %f'%test(testData, testLabels, cnnClassifer))
```

```
CNN Classifier accuracy: 98.030000
```

- Note that the MLP/ConvNet approaches lead to an accuracy a little higher than the K-NN approach.

- In general, Neural net approaches lead to significant increase in accuracy, but in this case since the problem is not too hard, the increase in accuracy is not very high.
- However, this is still quite significant considering the fact that the ConvNets we've used are relatively simple while the accuracy achieved using K-NN is with a search over 60,000 training images for every test image.
- You can look at the performance of various machine learning methods on this problem at http://yann.lecun.com/exdb/mnist/
- You can learn more about neural nets/ tensorflow at https://www.tensorflow.org/tutorials/
- You can play with a demo of neural network created by Daniel Smilkov and Shan Carter at https://playground.tensorflow.org/