CSE 152: Intro to Computer Vision - Spring 2019 Assignment 4

Instructor: David Kriegman

Assignment published on Wednesday, May 29, 2019

Due on Friday, June 7, 2019 at 11:59pm

Instructions

- This assignment must be completed individually. Review the academic integrity and collaboration policies on the course website.
- All solutions should be written in this notebook. Show your work for written questions.
- If you want to modify the skeleton code, you may do so. It has been merely been provided as a framework for your solution.
- You may use Python packages for basic linear algebra (e.g. NumPy or SciPy for basic operations), but you may not use packages that directly solve the problem. If you are unsure about using a specific package or function, ask the instructor and/or teaching assistants for clarification.
- You must submit this notebook exported as a PDF. You must also submit this notebook as an `.ipynb` file. Submit both files (`.pdf` and `.ipynb`) on Gradescope. **You must mark the PDF pages associated with each question in Gradescope. If you fail to do so, we may dock points.**
- It is highly recommended that you begin working on this assignment early.
- **Late policy:** a penalty of 10% per day after the due date.

Problem 1: Machine Learning [10 pts]

In this problem, you will implement K-Nearest Neighbors (KNN) algorithm for computer vision problems.

Part 1: Data preparation [1 pts]

Download the MNIST data from http://yann.lecun.com/exdb/mnist/ (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 (https://gist.github.com/akesling/5358964))

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

In [1]:
```
import os
import struct
import numpy as np

# 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')))
```

Some helper functions are given below.

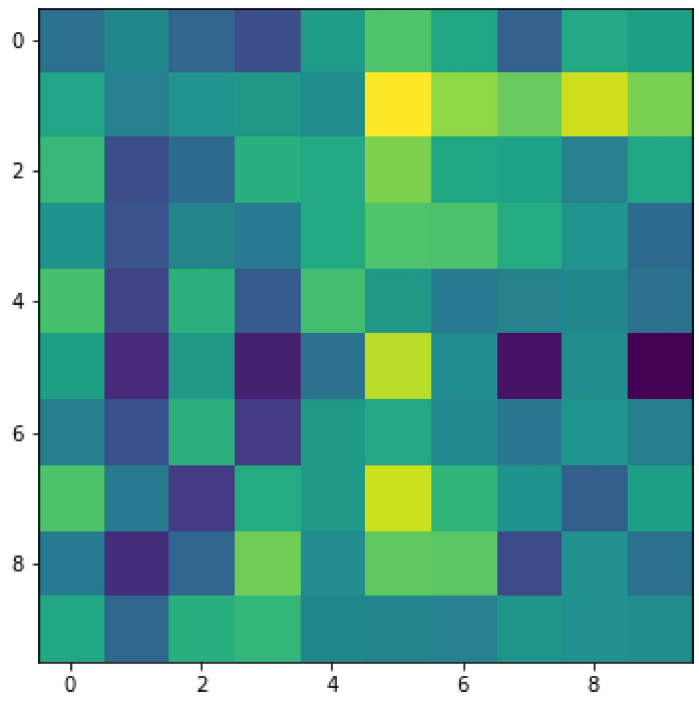
In [2]:
```
# a generator for batches of data
# yields data (batchsize, 3, 32, 32) and labels (batchsize)
# if shuffle, it will load batches in a random order
import matplotlib.pyplot as plt
def DataBatch(data, label, batchsize, shuffle=True):
    n = data.shape[0]
    if shuffle:
        index = np.random.permutation(n)
    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 \n' %
      test(testData, testLabels, randomClassifier))

print('Plot random training images for each class:')
count = 0
check = np.zeros(10)
imgs = np.zeros((10, 28, 28))
for img, lb in DataBatch(trainData, trainLabels, 1, shuffle=True):
    img = np.squeeze(img)
    if check[lb] == 1:
        continue
    else:
        check[lb] += 1
        count += 1
        imgs[lb,:,:] = img
    if count == 10:
        break

fig, ax = plt.subplots(nrows=2, ncols=5)
i = 0
for row in ax:
    for col in row:
        col.imshow(imgs[i,:,:])
        i += 1

plt.show()
```

```
Random classifier accuracy: 9.950000

Plot random training images for each class:

<Figure size 640x480 with 10 Axes>
```

Part 2: Confusion Matrix [3 pts]

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 were classified as class j.

In [3]:
```
# Using the tqdm module to visualize run time is suggested
from tqdm import tqdm
import time

# 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):

    pred = classifier(testData)

    M = np.zeros([10,10])
    for i in range(len(testLabels)):
        M[testLabels[i],pred[i]] += 1


    M = M / M.sum(axis=1)

    accuracy = np.diag(M)

    return M, accuracy

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)
```




```
[[0.09 0.1  0.09 0.08 0.1  0.12 0.11 0.09 0.11 0.11]
 [0.11 0.1  0.1  0.1  0.1  0.14 0.12 0.12 0.13 0.12]
 [0.11 0.08 0.09 0.11 0.11 0.12 0.11 0.11 0.1  0.11]
 [0.1  0.08 0.1  0.09 0.11 0.12 0.12 0.11 0.1  0.09]
 [0.12 0.08 0.11 0.08 0.12 0.1  0.09 0.1  0.1  0.09]
 [0.11 0.07 0.1  0.07 0.09 0.13 0.1  0.07 0.1  0.06]
 [0.09 0.08 0.11 0.08 0.1  0.11 0.1  0.09 0.1  0.1 ]
 [0.12 0.09 0.08 0.11 0.1  0.13 0.11 0.1  0.09 0.11]
 [0.09 0.07 0.09 0.12 0.1  0.12 0.12 0.08 0.1  0.09]
 [0.11 0.09 0.11 0.11 0.1  0.1  0.1  0.1  0.1  0.1 ]]
```

Part 3: K-Nearest Neighbors (KNN) [6 pts]

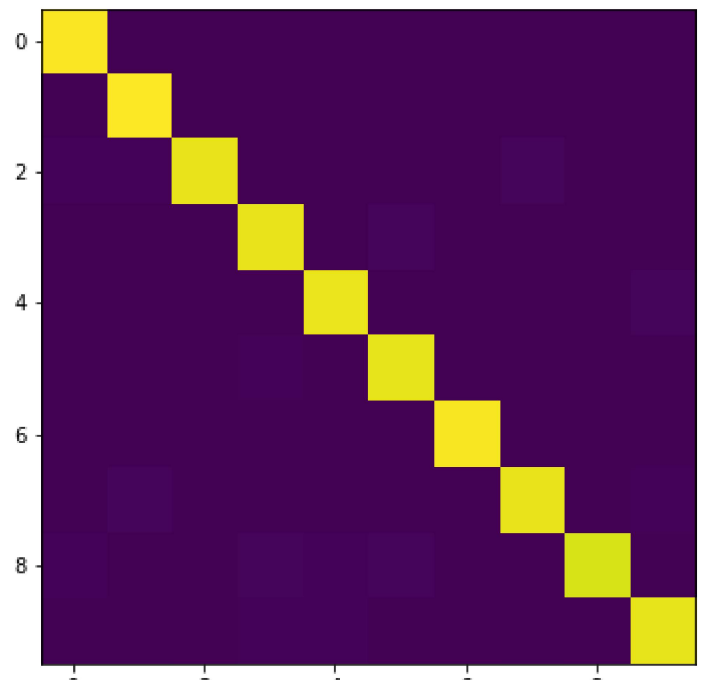
- Here you will implement a simple knn classifier. The distance metric is Euclidean in pixel space. k refers to the number of neighbors involved in voting on the class, and should be 3. You are allowed to use sklearn.neighbors.KNeighborsClassifier.
- Display the confusion matrix and accuracy for your KNN classifier trained on the entire training dataset. (should be ~97%)
- After evaluating the classifier on the test set, based on the confusion matrix, mention the number that the number '4' is most often predicted to be, other than '4'. Write your comment below.

In [4]:
```
from sklearn.neighbors import KNeighborsClassifier
import numpy as np
import matplotlib.pyplot as plt
class KNNClassifer():
    def __init__(self, k=3):
        # k is the number of neighbors involved in voting
        self.model = KNeighborsClassifier(n_neighbors=k)

    def train(self, trainData, trainLabels):
        trainData = trainData.reshape(trainData.shape[0],-1)
        self.model.fit(trainData,trainLabels)

    def __call__(self, x):
        # this method should take a batch of images
        # and return a batch of predictions
        x = x.reshape(x.shape[0],-1)
        return self.model.predict(x)

# 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: 64.760000
```

In [5]:
```
# test your classifier trained with all the training examples (This may take a while)
knnClassifer = KNNClassifer()
knnClassifer.train(trainData[:-1], trainLabels[:-1])

# display confusion matrix and testing accuracy for your KNN classifier trained with all the training examples
M, accuracy = Confusion(testData, testLabels, knnClassifer)
print ('KNN classifier accuracy: %f'%test(testData, testLabels, knnClassifer))
VisualizeConfusion(M)
```

```
KNN classifier accuracy: 97.040000
```




```
[[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.   0.   0.   0.02]
 [0.01 0.   0.   0.01 0.   0.96 0.01 0.   0.   0.  ]
 [0.01 0.   0.   0.   0.   0.   0.99 0.   0.   0.  ]
 [0.   0.02 0.   0.   0.   0.   0.   0.96 0.   0.01]
 [0.01 0.   0.   0.02 0.01 0.01 0.   0.   0.94 0.  ]
 [0.   0.   0.   0.01 0.01 0.   0.   0.01 0.   0.96]]
```

Comment:

Your comment here.

Problem 2: Deep Learning [18 pts]

Part 1: Initial setup [0 pts]

Follow the directions on https://pytorch.org/get-started/locally/ (https://pytorch.org/get-started/locally/) to install PyTorch on your computer.

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

To ensure that PyTorch was installed correctly, we will now verify the installation by running some sample PyTorch code. Here we construct a randomly initialized tensor.

In [6]:
```
from __future__ import print_function
import torch
x = torch.rand(5, 3)
print(x)
```

```
tensor([[0.2866, 0.6231, 0.7854],
        [0.2735, 0.8132, 0.3834],
        [0.9125, 0.9319, 0.5866],
        [0.9497, 0.6740, 0.1796],
        [0.6000, 0.6716, 0.8926]])
```

Part 2: Training with PyTorch [3 pts]

Below is some helper code to train your deep networks. Complete the train function for PTClassifier below. You should write down the training operations in this function. This function will be used in the following questions with different networks. You can look at https://pytorch.org/tutorials/beginner/pytorch_with_examples.html (https://pytorch.org/tutorials/beginner/pytorch_with_examples.html) for reference.

Reference

torch.nn.parameter vs. Variable: https://stackoverflow.com/questions/50935345/understanding-torch-nn-parameter (https://stackoverflow.com/questions/50935345/understanding-torch-nn-parameter)

In [8]:
```
# base class for your PyTorch networks. It implements the training loop
# (__init__), (train) and prediction(__call__)  for you.
# You will need to complete the (train) function to define the training operations
# structures in the following problems.
import torch.nn as nn
from torch.nn.parameter import Parameter
import torch.nn.functional as F
import torch.nn.init
import torch.optim as optim
from torch.autograd import Variable
from tqdm import tqdm
from scipy.stats import truncnorm

class PTClassifier():
    def __init__(self, net):
        self.net = net()

    def train(self, trainData, trainLabels, testData, testLabels, epochs=1, batchsize=50):
        criterion = nn.CrossEntropyLoss()
        learning_rate=3e-4
        optimizer = optim.Adam(self.net.parameters(),lr=learning_rate)
        for epoch in range(epochs):
            for i, (data,label) in enumerate(DataBatch(trainData, trainLabels, batchsize, shuffle=True)):
                inputs = Variable(torch.FloatTensor(data))
                targets = Variable(torch.LongTensor(label))

                # YOUR CODE HERE
                # Train the model using the optimizer and the batch data
                y_pred = self.net(inputs)
                loss = criterion(y_pred, targets)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

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

    def __call__(self, x):
        inputs = Variable(torch.FloatTensor(x))
        prediction = self.net(inputs)
        return np.argmax(prediction.data.cpu().numpy(), 1)

    def get_first_layer_weights(self):
        return self.net.weight1.data.cpu().numpy()

# helper function to get weight variable
def weight_variable(shape):
    # truncated normal continuous random variable.
    initial = torch.Tensor(truncnorm.rvs(-1/0.01, 1/0.01, scale=0.01, size=shape))
    return Parameter(initial, requires_grad=True)

# helper function to get bias variable
def bias_variable(shape):
    initial = torch.Tensor(np.ones(shape)*0.1)
    return Parameter(initial, requires_grad=True)

# Define Single Layer Perceptron network
class SLP(nn.Module):
    def __init__(self, in_features=28*28, classes=10):
        super(SLP, self).__init__()
        # model variables
        self.weight1 = weight_variable((classes, in_features))
        self.bias1 = bias_variable((classes))

    def forward(self, x):
        # linear operation
        # torch.addmm(beta=1, mat, alpha=1, mat1, mat2, out=None): returns Tensor beta*mat+alpha*(mat1.dot(mat2))
        # tensor.view(shape): returns a new tensor of a different shape.
        y_pred = torch.addmm(self.bias1, x.view(list(x.size())[0], -1), self.weight1.t())
        return y_pred


# test the example linear classifier (note you should get around 92% accuracy
# for 10 epochs and batchsize 50)
trainData=np.array(list(read('training','images')))
trainData=np.float32(np.expand_dims(trainData,-1))/255
trainData=trainData.transpose((0,3,1,2))
trainLabels=np.int32(np.array(list(read('training','labels'))))

testData=np.array(list(read('testing','images')))
testData=np.float32(np.expand_dims(testData,-1))/255
testData=testData.transpose((0,3,1,2))
testLabels=np.int32(np.array(list(read('testing','labels'))))

linearClassifier = PTClassifier(SLP)
linearClassifier.train(trainData, trainLabels, testData, testLabels, epochs=10)
print ('Linear classifier accuracy: %f'%test(testData, testLabels, linearClassifier))
```

```
Epoch:1 Accuracy: 89.290000
Epoch:2 Accuracy: 90.720000
Epoch:3 Accuracy: 91.320000
Epoch:4 Accuracy: 91.660000
Epoch:5 Accuracy: 91.690000
Epoch:6 Accuracy: 91.930000
Epoch:7 Accuracy: 92.220000
Epoch:8 Accuracy: 92.350000
Epoch:9 Accuracy: 92.440000
Epoch:10 Accuracy: 92.460000
Linear classifier accuracy: 92.460000
```

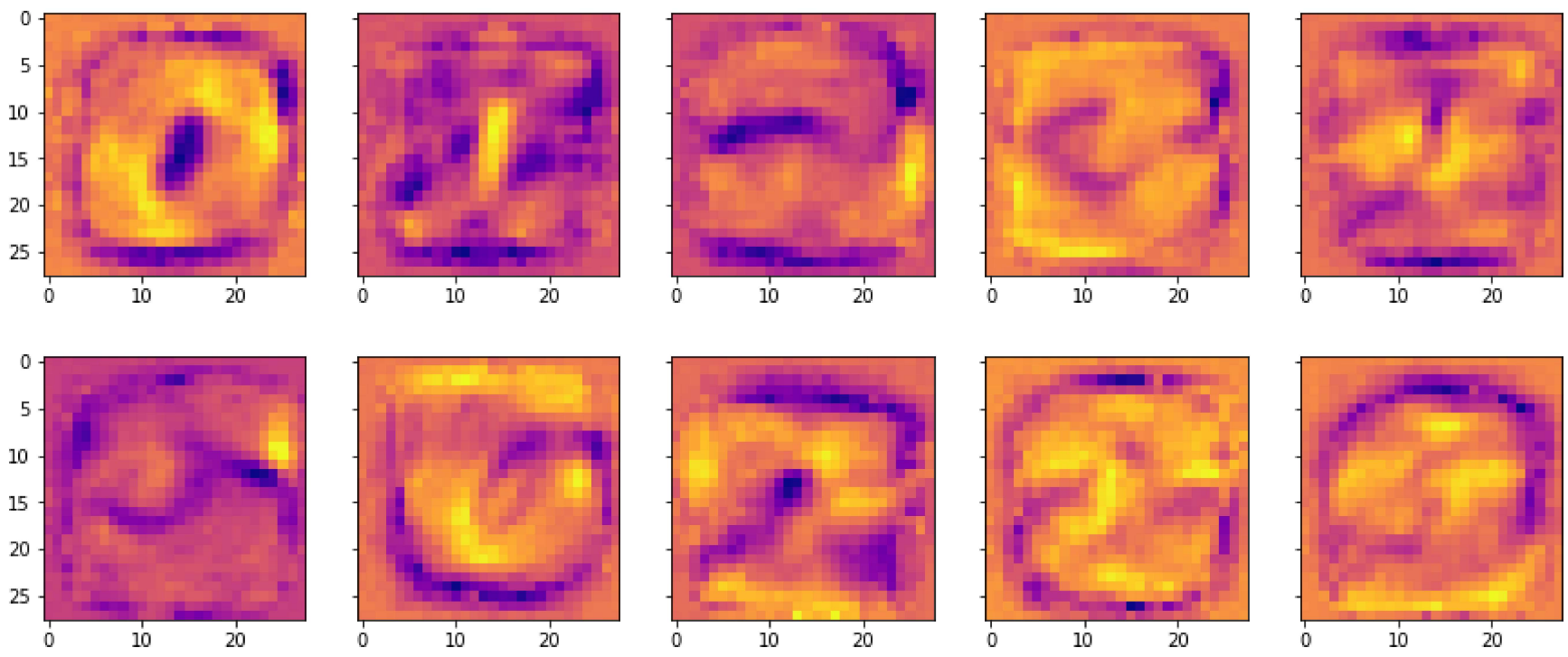
In [35]:
```
print(trainData.shape)
print(trainLabels.shape)
```

```
(60000, 1, 28, 28)
(60000,)
```

Part 3: Single Layer Perceptron [3 pts]

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 [29]:
```
# Your code here.
param = list(linearClassifier.net.parameters())
weights = param[0].detach().numpy()
fig, axes = plt.subplots(ncols=5, sharex='all', sharey='all',figsize=(15,15))
cmap = "plasma"
for i in range(5):
    class_i_weight = weights[i,:]
    class_i_weight = (class_i_weight - class_i_weight.min()) / (class_i_weight.max() - class_i_weight.min())
    class_i_weight = class_i_weight.reshape(28,28)
    axes[i].imshow(class_i_weight,cmap=cmap)

fig, axes = plt.subplots(ncols=5, sharex='all', sharey='all',figsize=(15,15))
cmap = "plasma"
for i in range(5,10):
    class_i_weight = weights[i,:]
    class_i_weight = (class_i_weight - class_i_weight.min()) / (class_i_weight.max() - class_i_weight.min())
    class_i_weight = class_i_weight.reshape(28,28)
    axes[i-5].imshow(class_i_weight,cmap=cmap)
```



Comment

the weights look kind of like the class digit, since the weighs are trying to find the features of the digit class , it represents the high dimensional features of that class which result it looking like the digit

Part 4: Multi Layer Perceptron (MLP) [7 pts]

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 around 97% accuracy for 10 epochs and batch size 50.

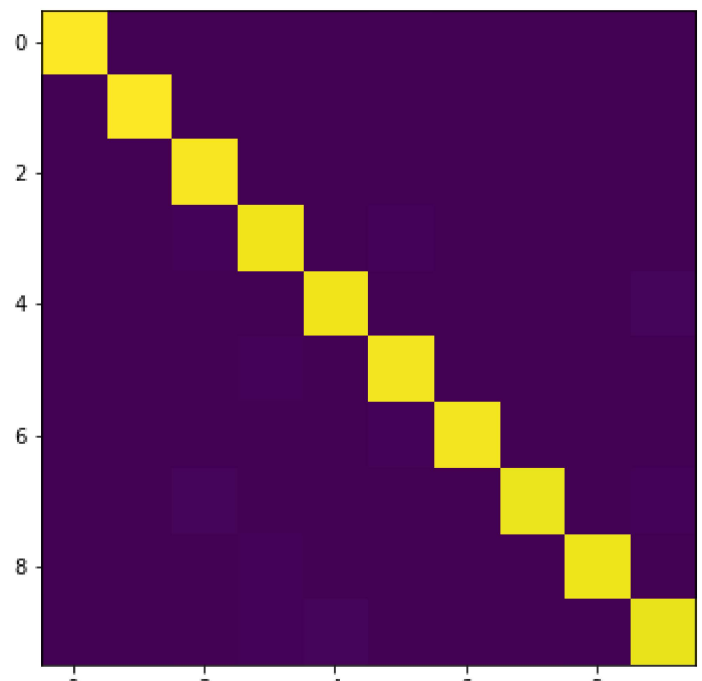
Plot the filter weights corresponding to the mapping from the inputs to the first 10 hidden layer outputs (out of 100). Do the weights look similar to the weights plotted in the previous problem? Why or why not?

In [60]:
```
# Define Multi Layer Perceptron network
class MLP(nn.Module):
    def __init__(self, in_features=28*28, hidden=100, classes=10):
        super(MLP, self).__init__()
        self.weight1 = weight_variable((hidden, in_features))
        self.bias1 = bias_variable((hidden))
        self.weight2 = weight_variable((classes, hidden))
        self.bias2 = bias_variable((classes))

    def forward(self, x):
        hidden1 = torch.addmm(self.bias1, x.view(list(x.size())[0], -1), self.weight1.t())
        relu1 = hidden1.clamp(min=0)
        y_pred = torch.addmm(self.bias2, relu1, self.weight2.t())
        return y_pred

mlpClassifer = PTClassifier(MLP)
mlpClassifer.train(trainData, trainLabels, testData, testLabels, epochs=10)
```

```
Epoch:1 Accuracy: 91.470000
Epoch:2 Accuracy: 92.940000
Epoch:3 Accuracy: 93.840000
Epoch:4 Accuracy: 94.690000
Epoch:5 Accuracy: 95.410000
Epoch:6 Accuracy: 95.770000
Epoch:7 Accuracy: 96.070000
Epoch:8 Accuracy: 96.460000
Epoch:9 Accuracy: 96.590000
Epoch:10 Accuracy: 96.880000
```

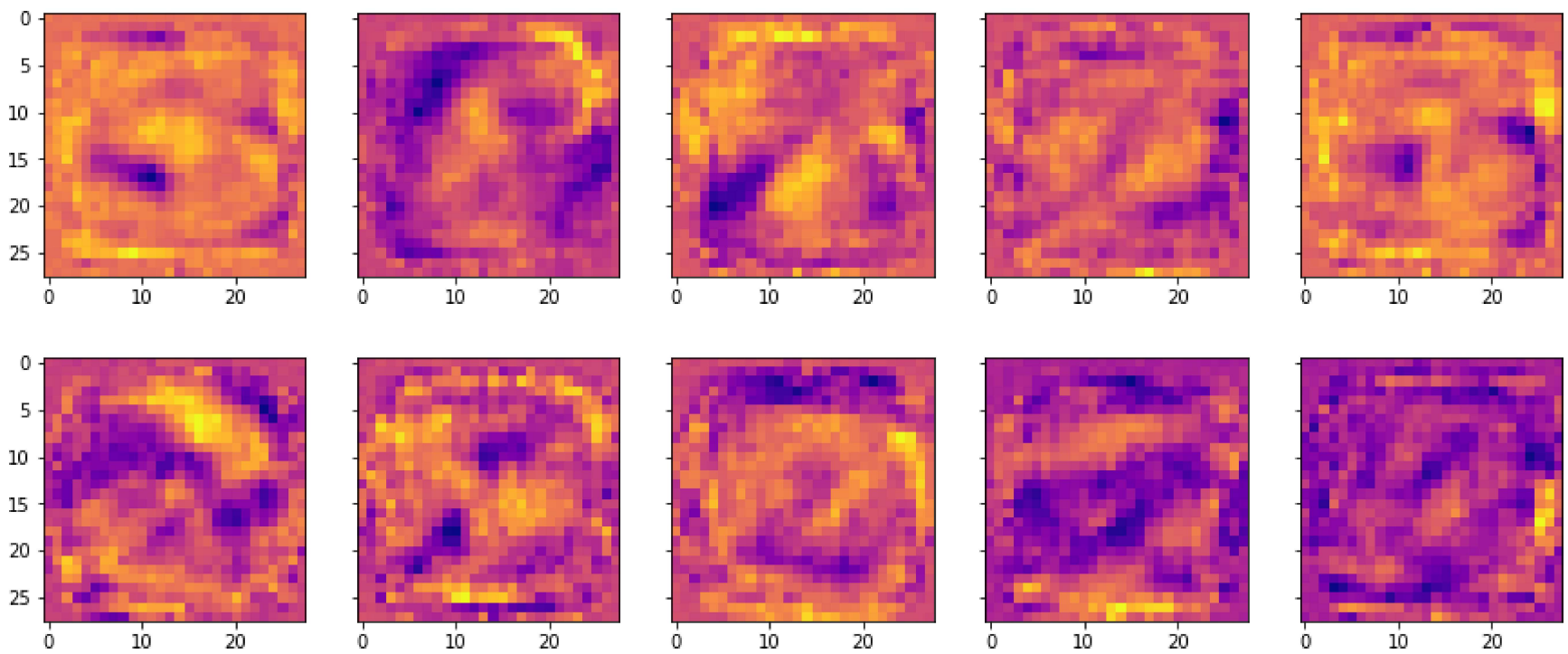
In [37]:
```
# Confusion Matrix and Accuracy
M, accuracy = Confusion(testData, testLabels, mlpClassifer)
print ('mlp classifier accuracy: %f'%test(testData, testLabels, mlpClassifer))
VisualizeConfusion(M)
```

```
mlp classifier accuracy: 96.930000
```




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

In [38]:
```
# plot filter weights
param = list(mlpClassifer.net.parameters())
weights = param[0].detach().numpy()
fig, axes = plt.subplots(ncols=5, sharex='all', sharey='all',figsize=(15,15))
cmap = "plasma"
for i in range(5):
    class_i_weight = weights[i,:]
    class_i_weight = (class_i_weight - class_i_weight.min()) / (class_i_weight.max() - class_i_weight.min())
    class_i_weight = class_i_weight.reshape(28,28)
    axes[i].imshow(class_i_weight,cmap=cmap)

fig, axes = plt.subplots(ncols=5, sharex='all', sharey='all',figsize=(15,15))
cmap = "plasma"
for i in range(5,10):
    class_i_weight = weights[i,:]
    class_i_weight = (class_i_weight - class_i_weight.min()) / (class_i_weight.max() - class_i_weight.min())
    class_i_weight = class_i_weight.reshape(28,28)
    axes[i-5].imshow(class_i_weight,cmap=cmap)
```



Comment

it seems that the more layers the more sophisticated the weights are, it doesn't look similar, there are more blobs on the boundary meaning that the classifier has learn more complicated weights for each pixel

Part 5: Convolutional Neural Network (CNN) [5 pts]

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)

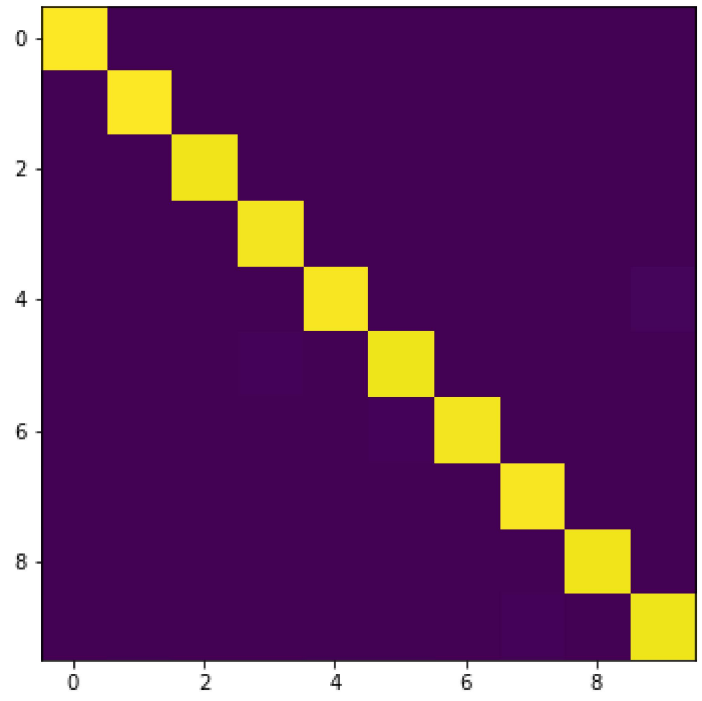
Display the confusion matrix and accuracy after training. You should get around 98% accuracy for 10 epochs and batch size 50.

In [64]:
```
def conv2d(x, W, stride):
    # x: input
    # W: weights (out, in, kH, kW)
    return F.conv2d(x, W, stride=stride, padding=1)

# Define Convolutional Neural Network
class CNN(nn.Module):
    # variables defined in __init__ will be the target of backprop
    def __init__(self, classes=10, n=5):
        super(CNN, self).__init__()
        self.stride = 2
        self.classes = classes
        self.weight1 = weight_variable((n,1,4,4))
        self.weight2 = weight_variable((n*2,n,4,4))
        self.weight3 = weight_variable((n*4,n*2,4,4))
        self.weight4 = weight_variable((classes,180))
        self.bias1 = bias_variable((n,14,14))
        self.bias2 = bias_variable((2*n,7,7))
        self.bias3 = bias_variable((4*n,3,3))
        self.bias4 = bias_variable((classes))

    def forward(self, x):
        hidden1 = conv2d(x,self.weight1,self.stride)
        hidden1.add_(self.bias1)
        relu1 = hidden1.clamp(min=0)

        hidden2 = conv2d(relu1,self.weight2,self.stride)
        hidden2.add_(self.bias2)
        relu2 = hidden2.clamp(min=0)

        hidden3 = conv2d(relu2,self.weight3,self.stride)
        hidden3.add_(self.bias3)
        relu3 = hidden3.clamp(min=0)

        m,_,_,_ = relu3.size()
        vec = relu3.view(m,-1)
        y_pred = torch.addmm(self.bias4, vec, self.weight4.t())
        return y_pred


cnnClassifer = PTClassifier(CNN)
cnnClassifer.train(trainData, trainLabels, testData, testLabels, epochs=10)
```

```
Epoch:1 Accuracy: 90.950000
Epoch:2 Accuracy: 93.480000
Epoch:3 Accuracy: 95.230000
Epoch:4 Accuracy: 96.240000
Epoch:5 Accuracy: 96.690000
Epoch:6 Accuracy: 97.170000
Epoch:7 Accuracy: 97.170000
Epoch:8 Accuracy: 97.420000
Epoch:9 Accuracy: 97.510000
Epoch:10 Accuracy: 97.620000
```

In [65]:
```
# Confusion Matrix and Accuracy
M, accuracy = Confusion(testData, testLabels, cnnClassifer)
print ('mlp classifier accuracy: %f'%test(testData, testLabels, cnnClassifer))
VisualizeConfusion(M)
```

```
mlp classifier accuracy: 97.620000
```




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

- 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 a significant increase in accuracy, but in this case the problem is not too hard, so the increase in accuracy will not be 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/ (http://yann.lecun.com/exdb/mnist/)
- You can learn more about PyTorch at https://pytorch.org/tutorials/index.html (https://pytorch.org/tutorials/index.html)
- You can find another image classifier training example at https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-beginner-blitz-cifar10-tutorial-py (https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-beginner-blitz-cifar10-tutorial-py)
- You can play with a demo of neural network created by Daniel Smilkov and Shan Carter at https://playground.tensorflow.org/ (https://playground.tensorflow.org/)