CSE 252A Computer Vision I Fall 2021 - Assignment 4

Instructor: Ben Ochoa

- Assignment Published On: **Wed, November 17, 2021.**
- Due On: **Wed, December 1, 2021 11:59 PM (Pacific Time).**

Instructions

- This assignment must be completed **individually**. For more details, please follow the Academic Integrity Policy and Collaboration Policy on Canvas.
- All solutions must be written in this notebook.
  - **If** it includes the theoretical problems, you **must** write your answers in Markdown cells (using LaTeX when appropriate).
  - Programming aspects of the assignment must be completed using Python in this notebook.
- You may use Python packages (such as `NumPy` and `SciPy`) for basic linear algebra, but you may not use packages that directly solve the problem.
  - If you are unsure about using a specific package or function, then ask the instructor and/or teaching assistants for clarification.
- You must submit this notebook exported as a PDF that contains separate pages. You must also submit this notebook as `.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:** Assignments submitted late will receive a 15% grade reduction for each 12 hours late (i.e., 30% per day). Assignments will not be accepted 72 hours after the due date. If you require an extension (for personal reasons only) to a due date, you must request one as far in advance as possible. Extensions requested close to or after the due date will only be granted for clear emergencies or clearly unforeseeable circumstances.

Problem 1: Machine Learning [28 pts]

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

Problem 1.1: Initial Setup

We will use Scikit-learn (Sklearn) module in for this problem. It is the most useful and robust library for machine learning in Python. It provides a selection of efficient tools for machine learning and statistical modeling including classification, regression, clustering and dimensionality reduction via a consistence interface in Python. This library, which is largely written in Python, is built upon NumPy, SciPy and Matplotlib.

Get started by installing the Sklearn module.

In [23]:
```
import sklearn
sklearn.__version__
```

Out[23]:
```
'1.0.1'
```

Problem 1.2: Download MNIST data [3 pts]

The MNIST database (Modified National Institute of Standards and Technology database) is a well-known dataset consisting of 28x28 grayscale images of handwritten digits. For this problem, we will use Sklearn to do machine learning classification on the MNIST database.

Sklearn provides a subset of MNIST database with 8x8 pixel images of digits. The `images` attribute of the dataset stores 8x8 arrays of grayscale values for each image. The `target` attribute of the dataset stores the digit each image represents. Complete `plot_mnist_sample()` to plot a 2x5 figure, each grid lies a sample image from a category. The following image gives an example: drawing

In [24]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from collections import defaultdict
```

In [25]:
```
# Download MNIST Dataset from Sklearn
digits = datasets.load_digits()
images = digits.data
labels = digits.target
# Print to show there are 1797 images (8 by 8 images for a dimensionality of 64)
print("Image Data Shape" , digits.data.shape)

# Print to show there are 1797 labels (integers from 0-9)
print("Label Data Shape", digits.target.shape)
```

```
Image Data Shape (1797, 64)
Label Data Shape (1797,)
```

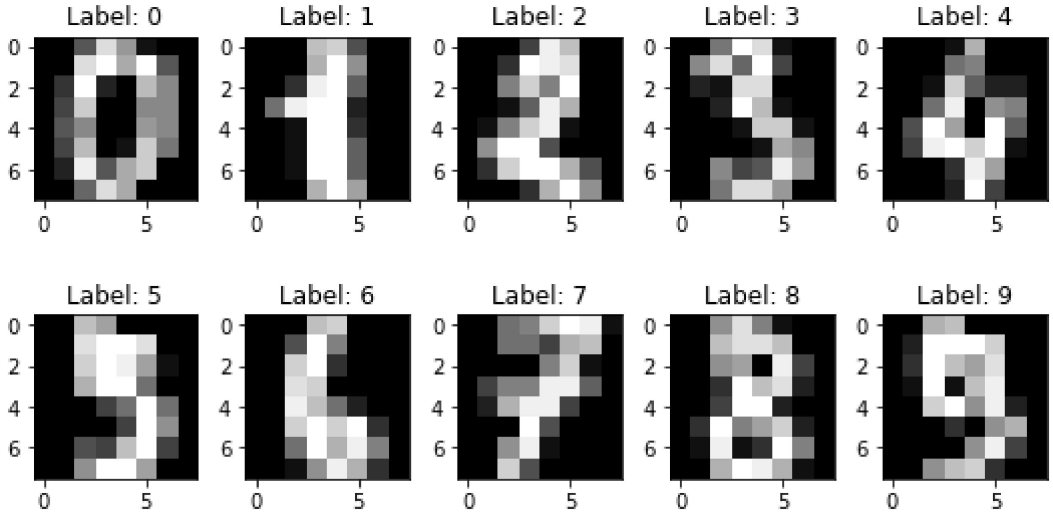
In [26]:
```
def plot_mnist_sample():
    """
    This function plots a sample image for each category,
    The result is a figure with 2x5 grid of images.

    """
    plt.figure()

    """ ==========
    YOUR CODE HERE
    ========== """
    img_dict = {}
    index = 0
    while len(img_dict) < 10:
        l = labels[index]
        if l not in img_dict:
            img_dict[l] = images[index].reshape(8,8)
        index = index + 1

    num_row = 2
```

```
    num_col = 5
    num = 10
    fig, axes = plt.subplots(num_row, num_col, figsize=(1.5*num_col,2*num_row))
    for i in range(num):
        ax = axes[i//num_col, i%num_col]
        ax.imshow(img_dict[i], cmap='gray')
        ax.set_title('Label: {}'.format(i))
    plt.tight_layout()
    plt.show()
```

In [27]:

```
# PLOT CODE: DO NOT CHANGE
# This code is for you to plot the results.

plot_mnist_sample()
```

```
<Figure size 432x288 with 0 Axes>
```



Problem 1.3: Recognizing hand-written digits with Sklearn [5 pts]

One of the most amazing things about Sklearn library is that it provides an easy pattern for you to call different models. In this part, we will get some experience with several classifiers in Sklearn. You will complete `LogisticRegressionClassifier` and `kNNCalssifier`.

In [28]:

```
# DO NOT CHANGE
#### Some helper functions are given below####
def DataBatch(data, label, batchsize, shuffle=True):
    """
    This function provides a generator for batches of data that
    yields data (batchsize, 3, 32, 32) and labels (batchsize)
    if shuffle, it will load batches in a random order
    """
    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]

def test(testData, testLabels, classifier):
    """
    Call this function to test the accuracy of a 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
```

In [29]:

```
# DO NOT CHANGE
# Split data into 50% train and 50% test subsets
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    digits.images.reshape((len(digits.images), -1)), digits.target, test_size=0.5, s
```

In [30]:

```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier

class RandomClassifier():
    """
    This is a sample classifier.
    given an input it outputs a random class
    """
    def __init__(self, classes=10):
        self.classes=classes
    def __call__(self, x):
        return np.random.randint(self.classes, size=x.shape[0])

class LogisticRegressionClassifier():
    def __init__(self, sol='liblinear'):
        """ ==========
        YOUR CODE HERE
        ========== """
        self.classifier = LogisticRegression(multi_class='multinomial')

    def train(self, trainData, trainLabels):
        """ ==========
        YOUR CODE HERE
        ========== """
        self.classifier.fit(trainData, trainLabels)
        return

    def __call__(self, x):
        """ ==========
        YOUR CODE HERE
        ========== """
        return self.classifier.predict(x)


class kNNClassifier():
    def __init__(self, k=3):
        """
         k is the number of neighbors involved in voting
        """
        """ ==========
        YOUR CODE HERE
        ========== """
        self.classifier = KNeighborsClassifier(n_neighbors=k)

    def train(self, trainData, trainLabels):
        """ ==========
```

```
        YOUR CODE HERE
        ========== """
        self.classifier.fit(trainData, trainLabels)
        return

    def __call__(self, x):
        """
        this method should take a batch of images and return a batch of predictions
        """
        """ ==========
        YOUR CODE HERE
        ========== """
        return self.classifier.predict(x)
```

In [31]:

```
# TEST CODE: DO NOT CHANGE
randomClassifierX = RandomClassifier()
print ('Random classifier accuracy: %f'%test(X_test, y_test, randomClassifierX))
```

```
Random classifier accuracy: 10.456062
```

In [59]:

```
# TEST CODE: DO NOT CHANGE
# TEST LogisticRegressionClassifier
import time
from collections import defaultdict

lrClassifierX = LogisticRegressionClassifier()
lrClassifierX.train(X_train, y_train)
start = time.time()
print ('Logistic Regression Classifier classifier accuracy: %f'%test(X_test, y_test,
end = time.time()
ttDict = defaultdict(float)
ttDict['Logistic Regression Classifier(sklearn) testing time:'] = round(end-start,5)
```

```
Logistic Regression Classifier classifier accuracy: 92.992214
C:\Users\Bill\anaconda3\envs\cv\lib\site-packages\sklearn\linear_model\_logistic.py:
818: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG,
```

In [60]:

```
# TEST CODE: DO NOT CHANGE
# TEST kNNClassifier
""" ==========
YOUR CODE HERE
========== """

knnClassifierX = kNNClassifier()
knnClassifierX.train(X_train, y_train)
start = time.time()
print ('kNNClassifier classifier accuracy: %f'%test(X_test, y_test, knnClassifierX))
end = time.time()
ttDict['kNNClassifier classifier(sklearn) testing time:'] = round(end-start,5)
```

```
kNNClassifier classifier accuracy: 96.329255
```

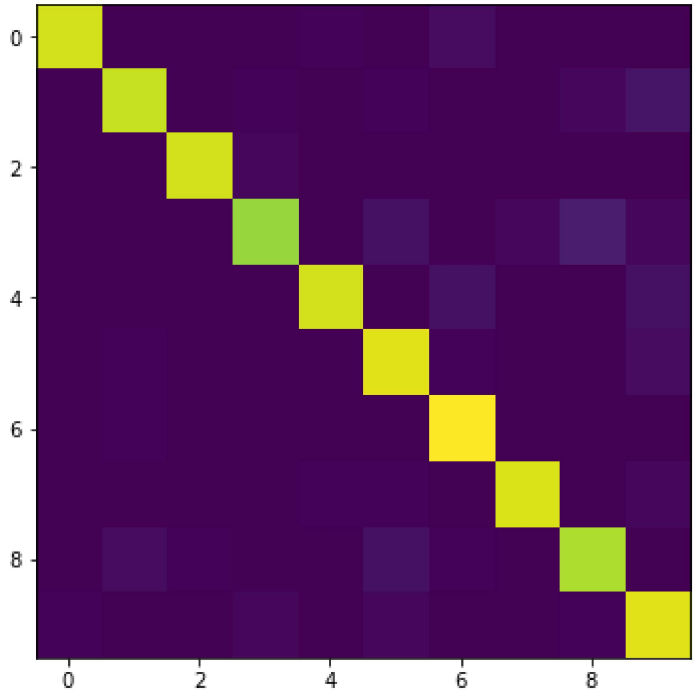
Problem 1.4: Confusion Matrix [5 pts]

A confusion matrix is a table that is often used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known. Here you will implement a function that computes the confusion matrix for a classifier. The matrix (M) should be $n \times n$ 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`. The following example plots confusion matrix for the `RandomClassifier`, your task is to plot the results for `LogisticRegressionClassifier` and `kNNClassifier`.  drawing

In [34]:
```
from tqdm import tqdm

def Confusion(testData, testLabels, classifier):
    batchsize=50
    correct=0
    M=np.zeros((10,10))
    num=testData.shape[0]/batchsize
    count=0
    acc=0

    for data,label in tqdm(DataBatch(testData,testLabels,batchsize,shuffle=False),to
        """ ==========
        YOUR CODE HERE
        ========== """
        results = classifier.classifier.predict(data)
        for i in range(results.shape[0]):
            row = label[i]
            column = results[i]
            M[row,column] = M[row,column] + 1

    correct = sum([M[i,i] for i in range(10)])
    M = M/np.sum(M)
    return M,correct*100.0/len(testData)

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

In [35]:
```
# TEST/PLOT CODE: DO NOT CHANGE
# TEST LogisticRegressionClassifier

M,acc = Confusion(X_test, y_test, lrClassifierX)
VisualizeConfussion(M)
print(acc)
```

```
18it [00:00, 4512.70it/s]
```

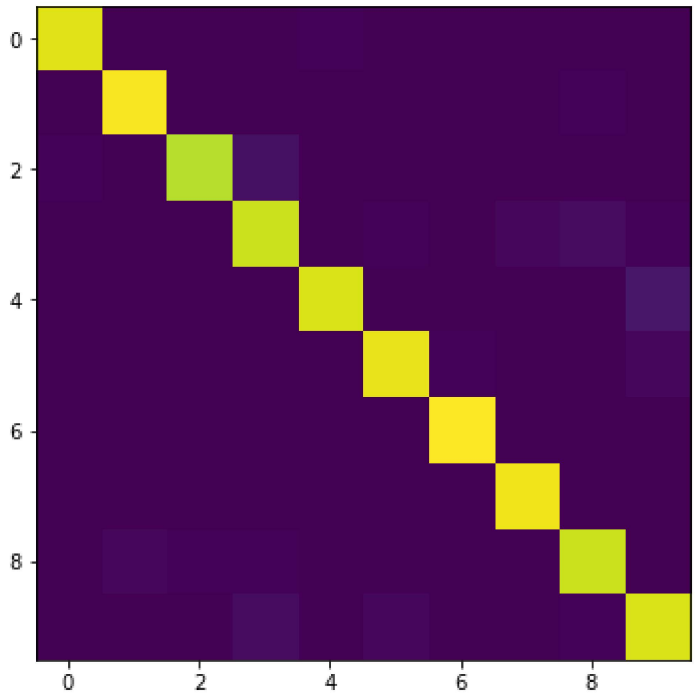



```
[[0.09 0.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.09 0.   0.   0.   0.   0.   0.   0.   0.01]
 [0.   0.   0.09 0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.08 0.   0.   0.   0.   0.01 0.  ]
 [0.   0.   0.   0.   0.09 0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.1  0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.1  0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.09 0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.09 0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.1 ]]
92.99221357063404
```

In [36]:

```
# TEST/PLOT CODE: DO NOT CHANGE
# TEST kNNClassifier

M,acc = Confusion(X_test, y_test, knnClassifierX)
VisualizeConfussion(M)
print(acc)
```

```
18it [00:00, 217.30it/s]
```




```
[[0.1  0.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.1  0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.09 0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.09 0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.1  0.   0.   0.   0.   0.01]
 [0.   0.   0.   0.   0.   0.1  0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.1  0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.1  0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.09 0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.1 ]]
96.32925472747498
```

Problem 1.5: K-Nearest Neighbors (KNN) [7 pts]

For this problem, you will complete a simple kNN classifer without Sklearn. The distance metric is Euclidean distance (L2 norm) in pixel space. k refers to the number of neighbors involved in voting on the class.

In [37]:

```
class kNNClassifierManual():
    def __init__(self, k=10):
        self.k=k
        self.x_train = 0
        self.y_train = 0
        self.x_train_1 = 0
        return
    def train(self, trainData, trainLabels):
        self.x_train = trainData
        self.y_train = trainLabels
        self.x_train_1 = np.append(self.x_train,self.y_train.reshape(self.y_train.sh
        return
    def predict(self,x):
        target = np.ones((self.x_train.shape[0], x.shape[0]))*x
        ed_map = np.sqrt((target-self.x_train)**2)
        eds = np.sum(ed_map, axis=1)
        max_indexs = np.argsort(eds,axis=0)[:self.k]
        voter_labels = [self.y_train[i] for i in max_indexs]
        pred = max(set(voter_labels), key=voter_labels.count)
        return pred

    def __call__(self, X):
```

```
        """ ==========
        YOUR CODE HERE
        ========== """
        y_preds = [self.predict(X[i]) for i in range(X.shape[0])]
        return np.array(y_preds)
```

In [68]:

```
# TEST/PLOT CODE: DO NOT CHANGE
# TEST kNNClassifierManual



knnClassifierManualX = kNNClassifierManual(k=3)
start = time.time()
knnClassifierManualX.train(X_train, y_train)
end = time.time()
ttDict['kNNClassifier classifier(Manual) training time:'] = round(end-start,5)
start = time.time()
print ('KNN classifier accuracy: %f'%test(X_test, y_test, knnClassifierManualX))
end = time.time()
ttDict['kNNClassifier classifier(Manual) testing time:'] = round(end-start,5)
```

```
KNN classifier accuracy: 95.216908
```

Problem 1.6: Principal Component Analysis (PCA) K-Nearest Neighbors (KNN) [8 pts]

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 [49]:

```
def Confusion(testData, testLabels, classifier):
    batchsize=50
    correct=0
    M=np.zeros((10,10))
    num=testData.shape[0]/batchsize
    count=0
    acc=0

    for data,label in tqdm(DataBatch(testData,testLabels,batchsize,shuffle=False),to
        """ ==========
        YOUR CODE HERE
        ========== """
        results = classifier.predict(data)
        for i in range(results.shape[0]):
            row = label[i]
            column = results[i]
            M[row,column] = M[row,column] + 1

    correct = sum([M[i,i] for i in range(10)])
    M = M/np.sum(M)
    return M,correct*100.0/len(testData)
class PCAKNNClassifer():
    def __init__(self, components=25, k=3):
        """ ==========
        YOUR CODE HERE
```

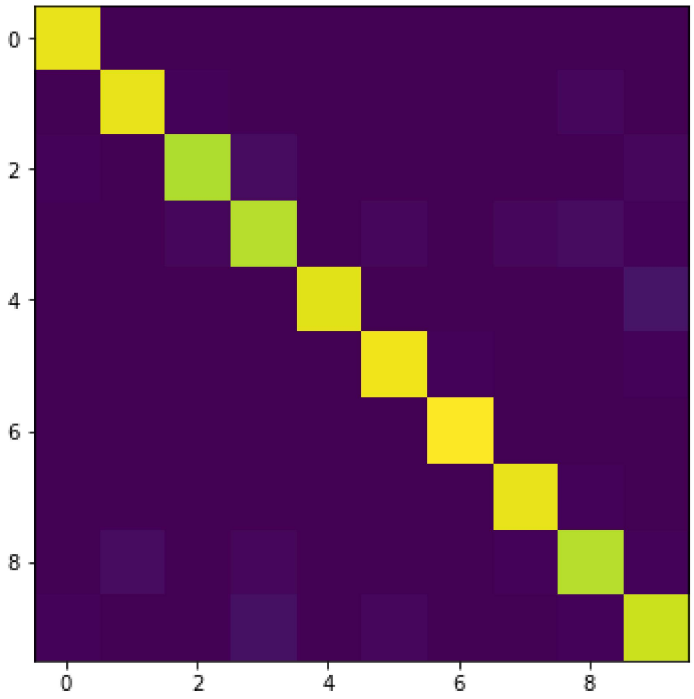
```
        ========== """
        self.components = components
        self.k = k
        self.classifier = kNNClassifierManual(k=k)
        self.proj_w = 0
        self.x_train = 0
        self.y_train = 0
        return

    def train(self, trainData, trainLabels):
        """ ==========
        YOUR CODE HERE
        ========== """
        self.x_train = trainData
        self.y_train = trainLabels
        cov_map = np.cov(self.x_train.T)
        S, U, V = np.linalg.svd(cov_map)
        self.proj_w = V[:self.components].T
        self.classifier.train(self.decompo(self.x_train), self.y_train)
        return
    def decompo(self,x):
        return x.dot(self.proj_w)
    def predict(self, x):
        x_pca = self.decompo(x)
        y_preds = [self.classifier.predict(x_pca[i]) for i in range(x_pca.shape[0])]
        return np.array(y_preds)
    def __call__(self, x):
        """ ==========
        YOUR CODE HERE
        ========== """
        x_pca = self.decompo(x)
        y_preds = [self.classifier.predict(x_pca[i]) for i in range(x_pca.shape[0])]
        return np.array(y_preds)


# test your classifier with only the first 100 training examples (use this
# while debugging)
pcaknnClassiferX = PCAKNNClassifer()
pcaknnClassiferX.train(X_train[:100], y_train[:100])
#print(y_test.shape)
print ('KNN classifier accuracy: %f'%test(X_test, y_test, pcaknnClassiferX))
```

```
KNN classifier accuracy: 82.981090
```

In [69]:

```
# test your classifier with all the training examples
pcaknnClassiferX = PCAKNNClassifer()
start = time.time()
pcaknnClassiferX.train(X_train, y_train)
end = time.time()
ttDict['PCA kNNClassifier classifier(Manual) training time:'] = round(end-start,5)
# display confusion matrix for your PCA KNN classifier with all the training example
""" ==========
YOUR CODE HERE
========== """
#test = pcaknnClassiferX.decompo(X_test)
start = time.time()
print ('PCA KNN classifier accuracy: %f'%test(X_test, y_test, pcaknnClassiferX))
end = time.time()
ttDict['PCA kNNClassifier classifier(Manual) testing time:'] = round(end-start,5)
M,acc_pca = Confusion(X_test, y_test, pcaknnClassiferX)
VisualizeConfussion(M)
#print ('PCA KNN classifier accuracy: %f'%acc_pca)
```

```
PCA KNN classifier accuracy: 95.328142
18it [00:00, 63.89it/s]
```




```
[[0.1  0.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.1  0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.09 0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.09 0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.1  0.   0.   0.   0.   0.01]
 [0.   0.   0.   0.   0.   0.1  0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.1  0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.1  0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.09 0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.09]]
```

In [74]:

```
keys = (list(ttDict.keys())[2:])
for k in keys:
    print(k + str(ttDict[k]))
```

```
kNNClassifier classifier(Manual) testing time:0.33815
PCA kNNClassifier classifier(Manual) testing time:0.26529
kNNClassifier classifier(Manual) training time:0.001
PCA kNNClassifier classifier(Manual) training time:0.00399
```

Time analysis

1. In terms of training time: pure kNN model (without using pca) (t = 0.001) < knn-pca(t = 0.0031) The result makes sense to mesince during the training process, pure kNN model just puts all the training data into memory while knn-pca has to do SVD and project training data into feature domain.
2. In terms of testing time: pure kNN model (without using pca) (t = 0.3382) > knn-pca(t = 0.2653) The result still makes sense to me since knn-pca model only takes 25 features for each data point to predict while pca model takes 28x28 for each data point to predict.

Problem 2: Deep learning [28 pts]

Problem 2.1 Initial setup [1 pts]

Follow the directions on 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. Furthermore, installing with GPU support is often more difficult to configure so it is suggested that you install the CPU only version. TA's will not provide any support related to GPU or CUDA.

Run the torch import statements below to verify your instalation.

In [4]:
```
import torch.nn as nn
import torch.nn.functional as F
import torch
from torch.autograd import Variable

x = torch.rand(5, 3)
print(x)
```

```
tensor([[0.4398, 0.3692, 0.7112],
        [0.0156, 0.8347, 0.5411],
        [0.8253, 0.8404, 0.7461],
        [0.8623, 0.1548, 0.4987],
        [0.6987, 0.4074, 0.8707]])
```

In this problem, we will use the full dataset of MNIST database with 28x28 pixel images of digits.

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 [5]:
```
import os
import struct
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from collections import defaultdict
from tqdm import tqdm

# Change path as required
path = "C:/Important/FA21/CSE252A/HW4/mnist/"

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)

X_train=np.array(list(read('training','images')))
Y_train=np.array(list(read('training','labels')))
X_test=np.array(list(read('testing','images')))
Y_test=np.array(list(read('testing','labels')))
```

Problem 2.2: Training with PyTorch [8 pts]

Below is some helper code to train your deep networks. Complete the train function for DNN below. You should write down the training operations in this function. That means, for a batch of data you have to initialize the gradients, forward propagate the data, compute error, do back propagation and finally update the parameters. 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 for reference.

In [6]:

```
import torch.nn.init
import torch.optim as optim
from torch.optim import Adam
from torch.autograd import Variable
from torch.nn.parameter import Parameter
from tqdm import tqdm
from scipy.stats import truncnorm
from torch.nn import Sequential, ReLU, Linear, Dropout, Module, CrossEntropyLoss
```

In [7]:

```
# base class for your deep neural networks. It implements the training loop (train_n
def DataBatch(data, label, batchsize, shuffle=True):
    """
    This function provides a generator for batches of data that
    yields data (batchsize, 3, 32, 32) and labels (batchsize)
    if shuffle, it will load batches in a random order
    """
    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]

def test(testData, testLabels, classifier):
    """
    Call this function to test the accuracy of a 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


class LM(Module):
    def __init__(self):
        super(LM, self).__init__()
        self.dense = Linear(28*28, 10)
        return

    def forward(self, x):

        return self.dense(x)

    def train_net(self, x_train, y_train, epochs=1, batchsize=50):
        """ ==========
        YOUR CODE HERE
        ========== """
        print(self.parameters())
        x_train = Variable(torch.FloatTensor(x_train))
        y_train = Variable(torch.LongTensor(y_train))
        #y_train = Variable(y_train)
        optimizer = Adam(self.parameters())
        self.train()
        for epoch in range(epochs):
            for data,label in DataBatch(x_train,y_train,batchsize,shuffle=False):

                X, y = data, label

                optimizer.zero_grad()
                predicted = self.forward(X.view(-1, 28 * 28))

                _,targets = y.max(dim=0)
                targets = Variable(y)
                loss = CrossEntropyLoss()(predicted, targets)
                loss.backward()
                optimizer.step()
            print(f'epoch:{epoch}, loss:{loss}')

        return
    def predict(self,x):
        inputs = Variable(torch.FloatTensor(x))
        prediction = self.forward(inputs.view(-1, 28 * 28))
        return np.argsort(prediction.detach().numpy(),axis=1)[:,-1]

    def __call__(self, x):
        inputs = Variable(torch.FloatTensor(x))
        prediction = self.forward(inputs.view(-1, 28 * 28))
        return np.argmax(prediction.data.cpu().numpy(), 1)
```

In [8]:

```
# example linear classifier - input connected to output
# you can take this as an example to learn how to extend DNN class
X_train=np.array(list(read('training','images')))
Y_train=np.array(list(read('training','labels')))
X_test=np.array(list(read('testing','images')))
Y_test=np.array(list(read('testing','labels')))

x_train=np.float32(np.expand_dims(X_train,-1))/255
x_train=x_train.transpose((0,3,1,2))
```

```
x_test=np.float32(np.expand_dims(X_test,-1))/255
x_test=x_test.transpose((0,3,1,2))
y_train, y_test = Y_train, Y_test
```

In [9]:

```
# test the example linear classifier (note you should get around 90% accuracy
# for 10 epochs and batchsize 50)

linearClassifier = LM()
linearClassifier.train_net(x_train, Y_train, epochs=10)
print(x_train.shape)
print ('Linear classifier accuracy: %f'%test(x_test, Y_test, linearClassifier))
```

```
<generator object Module.parameters at 0x000001EEC55858C8>
epoch:0, loss:0.16040153801441193
epoch:1, loss:0.12389500439167023
epoch:2, loss:0.11205270886421204
epoch:3, loss:0.10579146444797516
epoch:4, loss:0.10193704813718796
epoch:5, loss:0.09939732402563095
epoch:6, loss:0.0976373478770256
epoch:7, loss:0.09635632485151291
epoch:8, loss:0.09537583589553833
epoch:9, loss:0.0945870503783226
(60000, 1, 28, 28)
Linear classifier accuracy: 92.500000
```

In [10]:

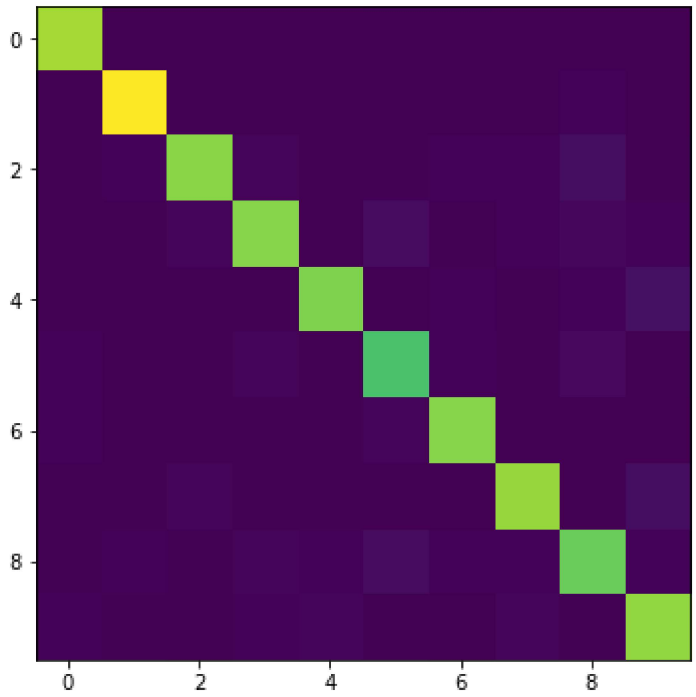
```
def Confusion(testData, testLabels, classifier):
    batchsize=50
    correct=0
    M=np.zeros((10,10))
    num=testData.shape[0]/batchsize
    count=0
    acc=0

    for data,label in tqdm(DataBatch(testData,testLabels,batchsize,shuffle=False),to
        """ ==========
        YOUR CODE HERE
        ========== """
        results = classifier.predict(data)
        #print(results)
        #print(label)
        for i in range(results.shape[0]):
            row = label[i]
            column = results[i]
            M[row,column] = M[row,column] + 1
    correct = sum([M[i,i] for i in range(10)])
    M = M/np.sum(M)
    return M,correct*100.0/len(testData)

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

In [11]:

```
# display confusion matrix
""" ==========
YOUR CODE HERE
========== """
M,acc = Confusion(x_test, y_test, linearClassifier)
```

```
VisualizeConfussion(M)
print(acc)
```

```
100%|██████████████████████████| 200/200 [00:00<00:00, 5141.94it/s]
```




```
[[0.1  0.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.11 0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.09 0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.09 0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.09 0.   0.   0.   0.   0.01]
 [0.   0.   0.   0.   0.   0.08 0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.09 0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.09 0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.09 0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.09]]
92.5
```

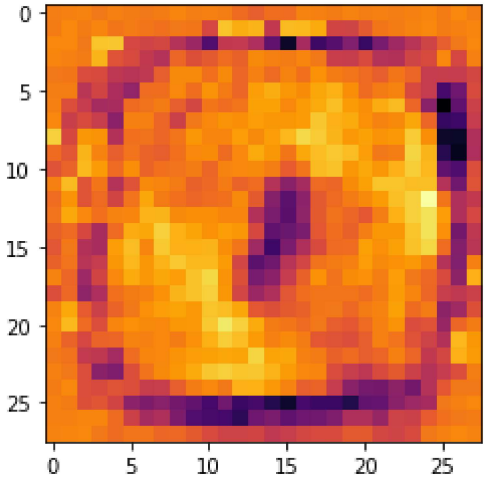
Problem 2.3: Single Layer Perceptron [3 pts]

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.

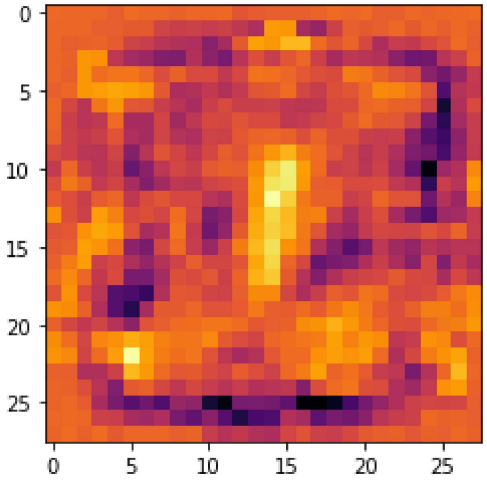
In [12]:

```
# Plot filter weights corresponding to each class, you may have to reshape them to m
# linearClassifier.weight1.data will give you the first layer weights
""" ==========
YOUR CODE HERE
========== """
weights = linearClassifier.dense.weight.detach().numpy()
#weights = (weights - np.min(weights, axis=1).reshape(10,1))/(np.max(weights, axis=1
for i in range(weights.shape[0]):
    print("layer: {}".format(i))
    weights[i] = (weights[i] - np.min(weights[i]))/ (np.max(weights[i])-np.min(weigh
    plt.imshow(weights[i].reshape(28, 28), cmap='inferno')
    plt.show()
#weights.shape
```

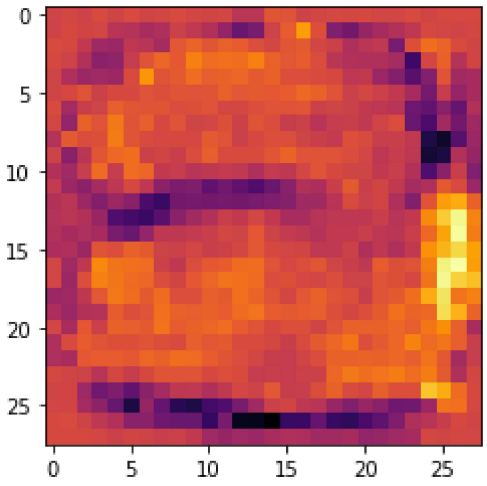
```
layer: 0
```



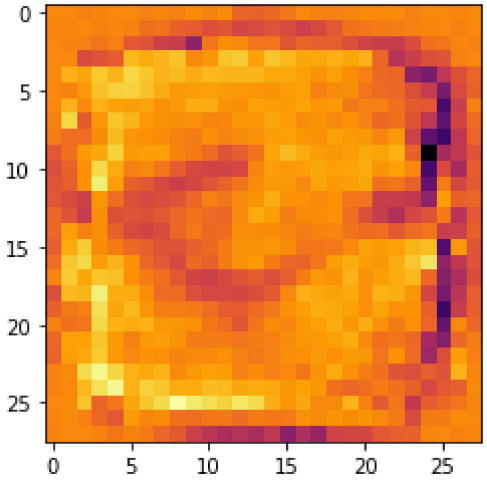

```
layer: 1
```



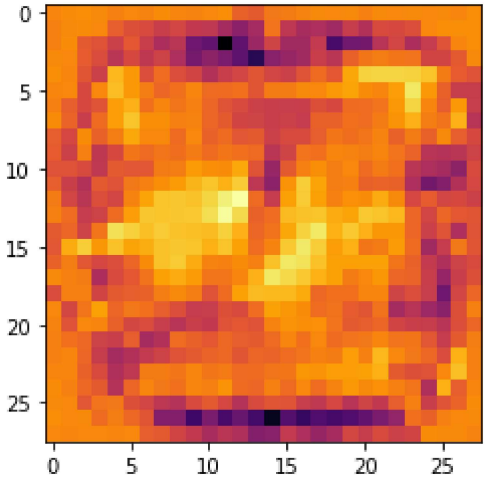

```
layer: 2
```



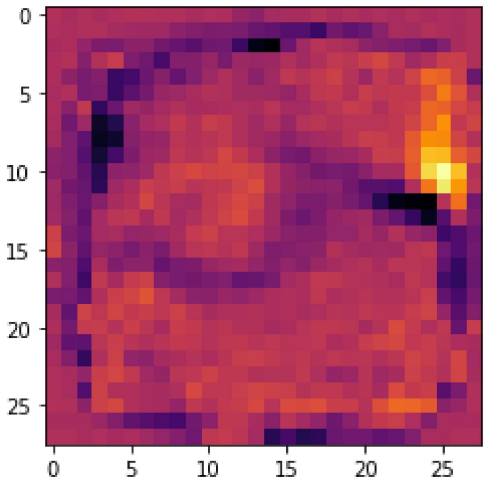

```
layer: 3
```



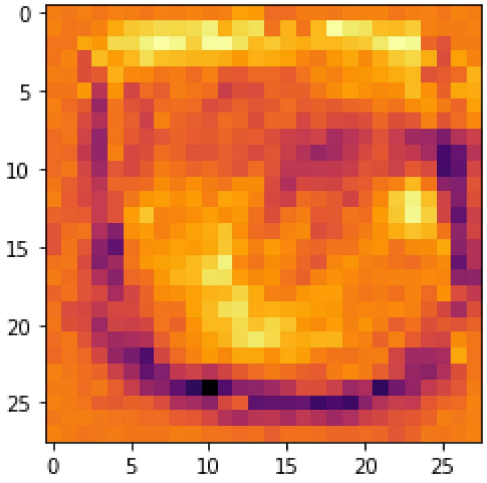

```
layer: 4
```



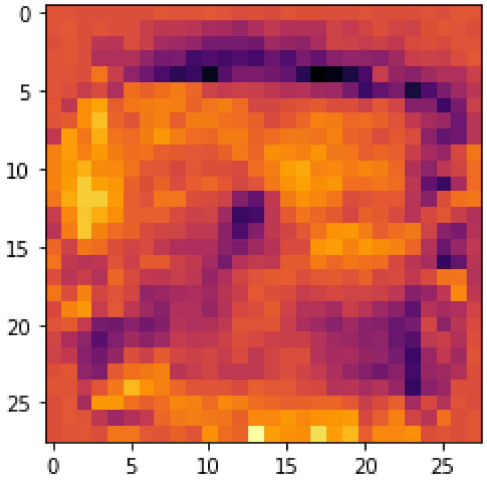

```
layer: 5
```



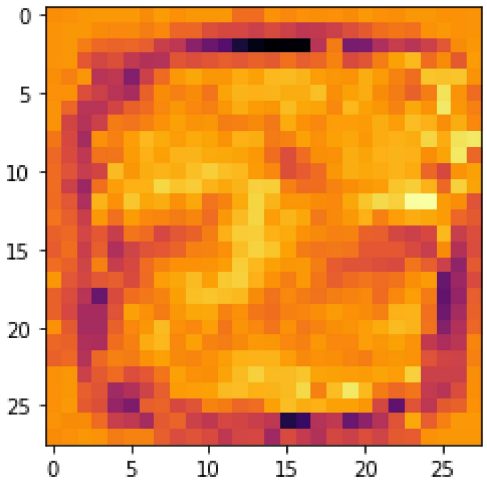

```
layer: 6
```



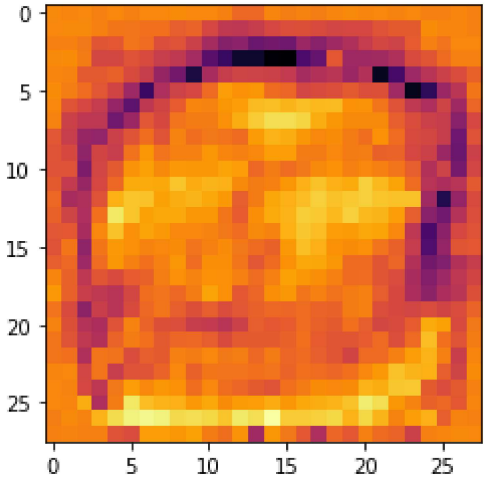
```
layer: 7
```



```
layer: 8
```



```
layer: 9
```



Comments on weights

The pictures show that each weight vector after being plotted more or less possess the contour(or "feature") of the number that it is going to predict since each weight vector has learnt to activate or give a high response when predicting its corresponding number.

Problem 2.4: Multi Layer Perceptron (MLP) [8 pts]

Here you will implement an MLP. The MLP should 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?

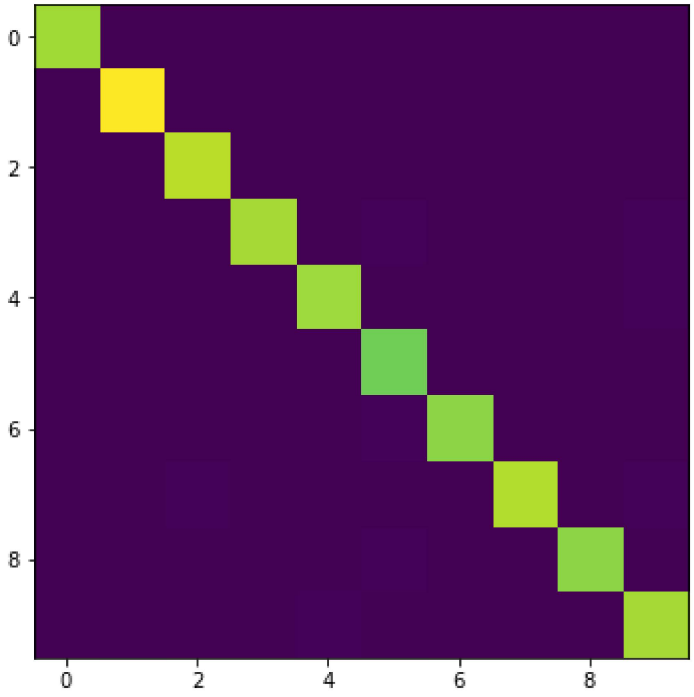
In [13]:
```
class DNN(Module):
    def __init__(self):
        super(DNN, self).__init__()
        self.dense = Linear(28*28, 100)
        self.dense1 = Linear(100, 10)
        return

    def forward(self, x):
        x = F.relu(self.dense(x))
        x = self.dense1(x)
        return x

    def train_net(self, x_train, y_train, epochs=1, batchsize=50):
        """ ==========
```

```
        YOUR CODE HERE
        ========== """
        print(self.parameters())
        x_train = Variable(torch.FloatTensor(x_train))
        y_train = Variable(torch.LongTensor(y_train))
        #y_train = Variable(y_train)
        optimizer = Adam(self.parameters())
        self.train()
        for epoch in range(epochs):
            for data,label in DataBatch(x_train,y_train,batchsize,shuffle=False):

                X, y = data, label

                optimizer.zero_grad()
                predicted = self.forward(X.view(-1, 28 * 28))

                _,targets = y.max(dim=0)
                targets = Variable(y)
                loss = CrossEntropyLoss()(predicted, targets)
                loss.backward()
                optimizer.step()
            print(f'epoch:{epoch}, loss:{loss}')

        return
    def predict(self,x):
        inputs = Variable(torch.FloatTensor(x))
        prediction = self.forward(inputs.view(-1, 28 * 28))
        return np.argsort(prediction.detach().numpy(),axis=1)[:,-1]

    def __call__(self, x):
        inputs = Variable(torch.FloatTensor(x))
        prediction = self.forward(inputs.view(-1, 28 * 28))
        return np.argmax(prediction.data.cpu().numpy(), 1)
```

In [14]:

```
# Plot confusion matrix
dnn = DNN()
dnn.train_net(x_train, y_train, epochs=10, batchsize=50)
M_mlp,acc_mlp = Confusion(x_test, y_test, dnn)
print ('MLP classifier accuracy: %f'%acc_mlp)
VisualizeConfussion(M_mlp)
```

```
<generator object Module.parameters at 0x000001EEC94169C8>
epoch:0, loss:0.06909351050853729
epoch:1, loss:0.04285700246691704
epoch:2, loss:0.04075285419821739
epoch:3, loss:0.03107570856809616
epoch:4, loss:0.020825374871492386
epoch:5, loss:0.012498609721660614
epoch:6, loss:0.007568690925836563
epoch:7, loss:0.006472923327237368
epoch:8, loss:0.004225503653287888
epoch:9, loss:0.002447783015668392
100%|██████████| 200/200 [00:00<00:00, 3518.03it/s]
MLP classifier accuracy: 97.680000
```

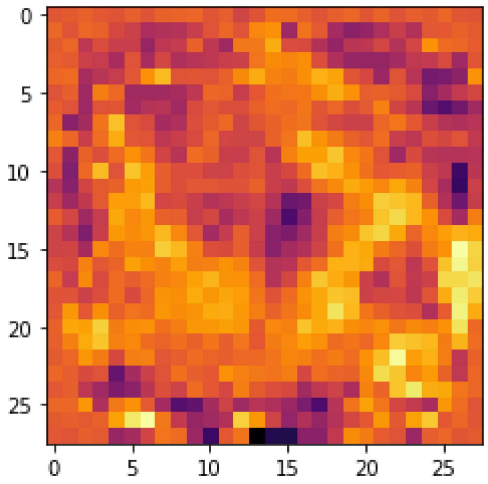


```
[[0.1  0.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.11 0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.1  0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.1  0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.1  0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.09 0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.09 0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.1  0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.09 0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.1 ]]
```

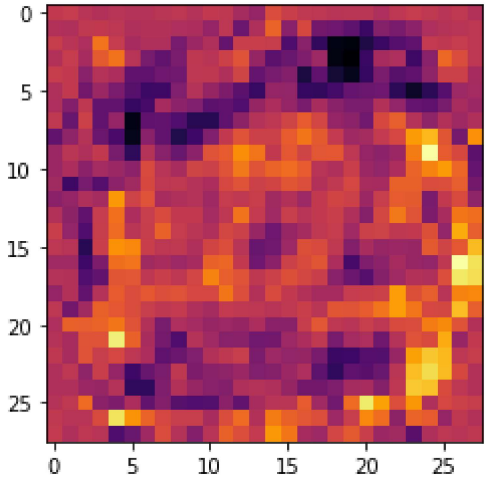
In [15]:

```
# Plot filter weights
""" ==========
YOUR CODE HERE
========== """
weights = dnn.dense.weight.detach().numpy()
#weights = (weights - np.min(weights, axis=1).reshape(10,1))/(np.max(weights, axis=1
for i in range(10):
    print("layer: {}".format(i))
    weights[i] = (weights[i] - np.min(weights[i]))/ (np.max(weights[i])-np.min(weigh
    plt.imshow(weights[i].reshape(28, 28), cmap='inferno')
    plt.show()
#weights.shape
```

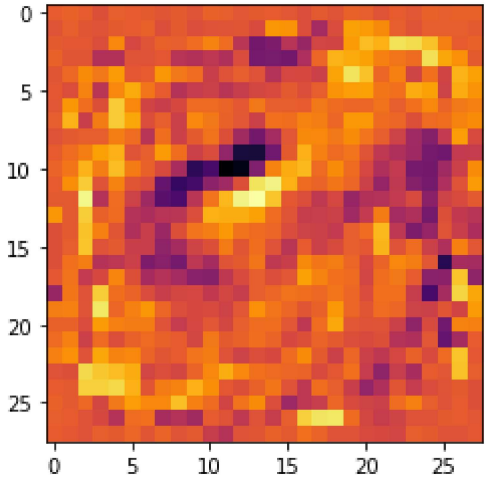
```
layer: 0
```



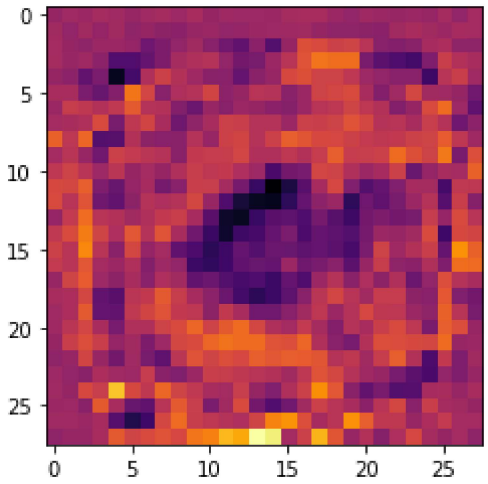
```
layer: 1
```



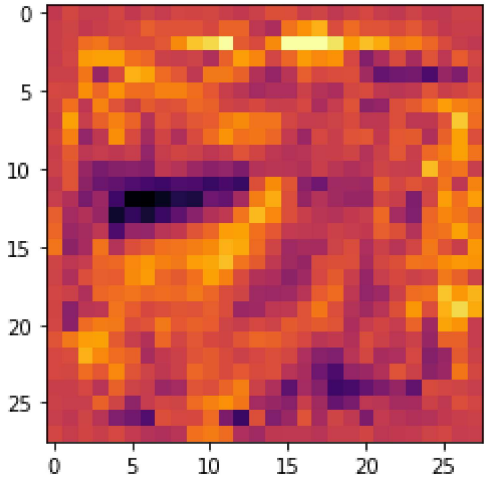

```
layer: 2
```



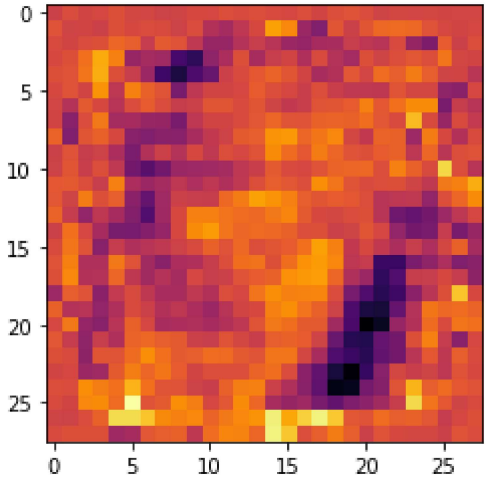

```
layer: 3
```



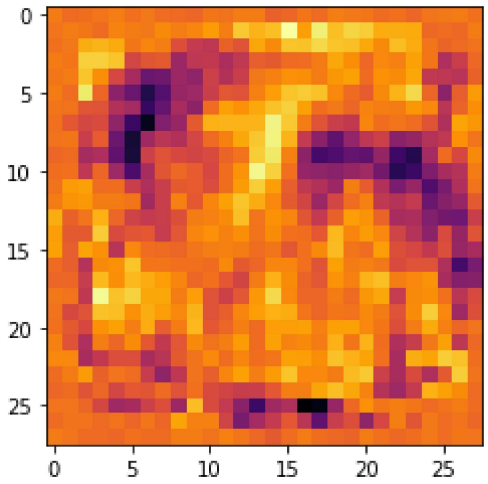

```
layer: 4
```



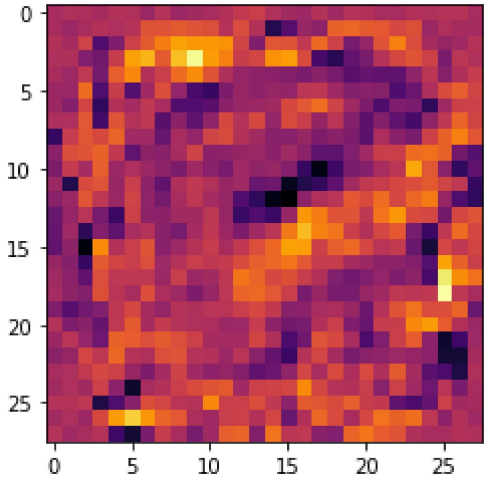

```
layer: 5
```



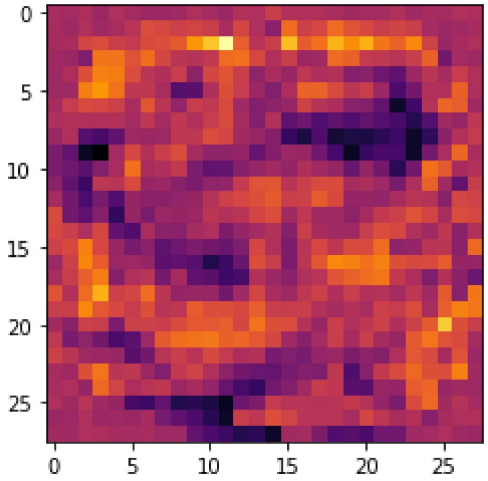

```
layer: 6
```



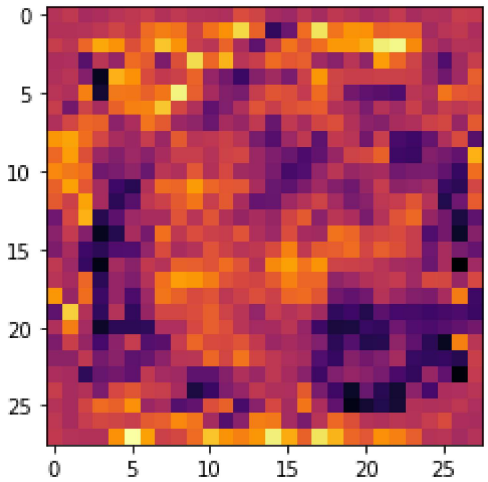

```
layer: 7
```




```
layer: 8
```




```
layer: 9
```



Comments on weights:

The weights do not look like as clearly as numbers like we observed in the previous case. Some of them look like number 2, some look like a combinantion of multiple numbers and some filters do not look like numbers at all.

Linear classifier was a special case where the output neuron is simply dot product with input image plus a bias and hence we had all filters looking like numbers. It may not be so here.

In above figure, that look somewhat like 2 (filter 5, filter 7. filter 8) points that the network is trying to fit different hidden neurons to the same hand-written digits with possibly different

neuron activations for different strokes. But it's unclear what exactly other filters and if they represent anything tangible.

So, eventhough we see few patterns for the filters, we still have 100 hidden neurons and a ReLU non-linearity. It is very difficult to figure out what the neural network actually learns for each filter at the hidden neurons because it has immense flexibility with the 100 units. This aspect essentially reflects in the 'hidden' part of the name 'hidden layer'. In conclusion, as networks grow deep and we keep adding non-linearities, the analysis of what network is doing becomes very difficult.

Problem 2.5: Convolutional Neural Network (CNN) [8 pts]

Here you will implement a CNN with the following architecture:

- n=5
- ReLU(Conv(kernel_size=5x5, stride=2, output_features=n))
- ReLU(Conv(kernel_size=5x5, stride=2, output_features=n*2))
- ReLU(Linear(hidden units = 64))
- Linear(output_features=classes)

So, 2 convolutional layers, followed by 1 fully connected hidden layer and then the output layer

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

Note: You are not allowed to use torch.nn.Conv2d() and torch.nn.Linear(), Using these will lead to deduction of points. Use the declared conv2d(), weight_variable() and bias_variable() functions. Although, in practice, when you move forward after this class you will use torch.nn.Conv2d() which makes life easier and hides all the operations underneath.

In [16]:
```
import numpy as np
import torch.nn.init
from torch import nn as nn
import torch.optim as optim
from torch.optim import Adam
from torch.autograd import Variable
from torch.nn import Parameter as Parameter
from tqdm import tqdm
from scipy.stats import truncnorm
from torch import flatten
from torch.nn import Sequential, ReLU, Linear, Dropout, Module, CrossEntropyLoss
```

In [23]:
```
def DataBatch(data, label, batchsize, shuffle=True):
    """
    This function provides a generator for batches of data that
    yields data (batchsize, 3, 32, 32) and labels (batchsize)
    if shuffle, it will load batches in a random order
    """
    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]

def test(testData, testLabels, classifier):
    """
    Call this function to test the accuracy of a 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
def weight_variable(shape):
    initial = torch.Tensor(truncnorm.rvs(-1/0.01, 1/0.01, scale=0.01, size=shape))
    return Parameter(initial, requires_grad=True)
def bias_variable(shape):
    initial = torch.Tensor(np.ones(shape)*0.1)
    return Parameter(initial, requires_grad=True)

def conv2d(x, W ,stride,bias=None):
    # x: input
    # W: weights (out, in, kH, kW)
    return F.conv2d(x, W, bias, stride=stride, padding=2)

# Defining a Convolutional Neural Network
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.stride = 2
        self.conv_1_weight = weight_variable((5, 1, 5, 5))
        self.conv_1_bias = bias_variable((14,14))
        self.conv_2_weight = weight_variable((10, 5, 5, 5))
        self.conv_2_bias = bias_variable((7,7))
        self.linear_weight1 = weight_variable((64, 490))
        self.bias_weight1 = bias_variable(64)
        self.linear_weight2 = weight_variable((10, 64))
        self.bias_weight2 = bias_variable(10)
        return

    def forward(self, x):
        #x = x.view(x.size(0), -1)
        #print(x.shape)
        out = F.relu(conv2d(x.view(-1,1,28,28), self.conv_1_weight, self.stride) + s
        out = F.relu(conv2d(out, self.conv_2_weight, self.stride) + self.conv_2_bias
        #out = flatten(out, start_dim=1)
        #print(out.shape)
        out = F.relu(torch.addmm(self.bias_weight1, out.view(list(out.size())[0], -1
        #print(out.shape)
        out = F.relu(torch.addmm(self.bias_weight2, out.view(list(out.size())[0], -1
        return out
    def train_net(self, x_train, y_train, epochs=1, batchsize=50):
        """ ==========
        YOUR CODE HERE
        ========== """
        #print(self.parameters())
        x_train = Variable(torch.FloatTensor(x_train))
        y_train = Variable(torch.LongTensor(y_train))
        #y_train = Variable(y_train)
        optimizer = Adam(self.parameters(), lr=0.00025)
        self.train()
        for epoch in range(epochs):
            for data,label in DataBatch(x_train,y_train,batchsize,shuffle=False):
```

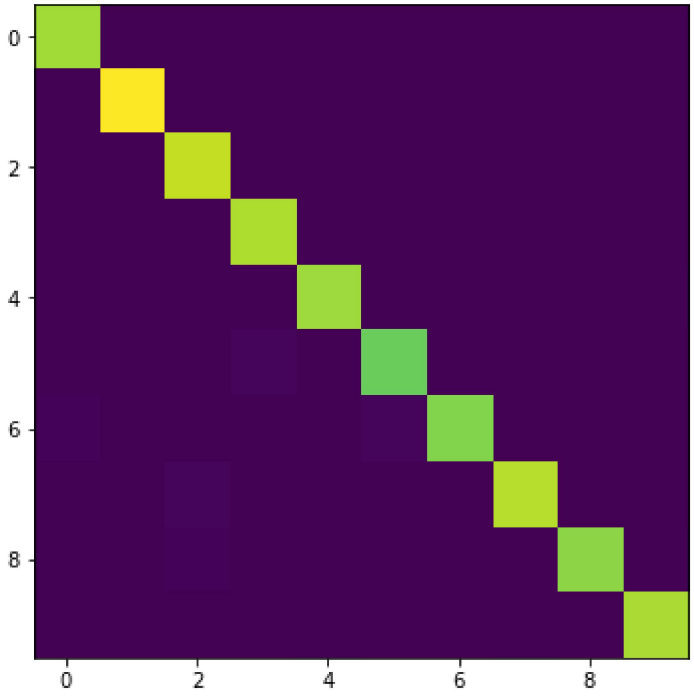
```
                X, y = data, label
                #print(X.shape)
                optimizer.zero_grad()
                predicted = self.forward(X.view(-1,28*28))

                _,targets = y.max(dim=0)
                targets = Variable(y)
                #print(predicted.shape)
                #print(targets.shape)
                loss = CrossEntropyLoss()(predicted, targets)
                loss.backward()
                optimizer.step()
            print(f'epoch:{epoch}, loss:{loss}')

        return
    def predict(self,x):
        inputs = Variable(torch.FloatTensor(x))
        prediction = self.forward(inputs.view(-1,28*28))
        return np.argsort(prediction.detach().numpy(),axis=1)[:,-1]

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

cnn = CNN()
#print(cnn)
X_train = X_train.reshape(60000,1,28,28)
cnn.train_net(X_train, y_train, epochs=10)
```

```
epoch:0, loss:0.023014742881059647
epoch:1, loss:0.016253409907221794
epoch:2, loss:0.015597229823470116
epoch:3, loss:0.00858762115240097
epoch:4, loss:0.0058504752814769745
epoch:5, loss:0.005710897035896778
epoch:6, loss:0.0025857461150735617
epoch:7, loss:0.004232219886034727
epoch:8, loss:0.0010532207088544965
epoch:9, loss:0.00018381547124590725
```

In [24]:

```
# Plot confusion matrix
""" ==========
YOUR CODE HERE
========== """
X_test = X_test.reshape(10000,1,28,28)
M_mlp,acc_mlp = Confusion(X_test, y_test, cnn)
print ('MLP classifier accuracy: %f'%acc_mlp)
VisualizeConfussion(M_mlp)
```

```
100%|████████████████████████| 200/200 [00:00<00:00, 1130.70it/s]
MLP classifier accuracy: 98.070000
```




```
[[0.1  0.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.11 0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.1  0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.1  0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.1  0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.09 0.   0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.09 0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.1  0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.09 0.  ]
 [0.   0.   0.   0.   0.   0.   0.   0.   0.   0.1 ]]
```

- 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/ pytorch at https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
- You can play with a demo of neural network created by Daniel Smilkov and Shan Carter at https://playground.tensorflow.org/

In [ ]: