CSE 252A Computer Vision I Fall 2019 - Homework 5

Instructor: Ben Ochoa

Assignment Published On: Thursday, November 21, 2019

Due On: Saturday, December 7, 2019 11:59 pm

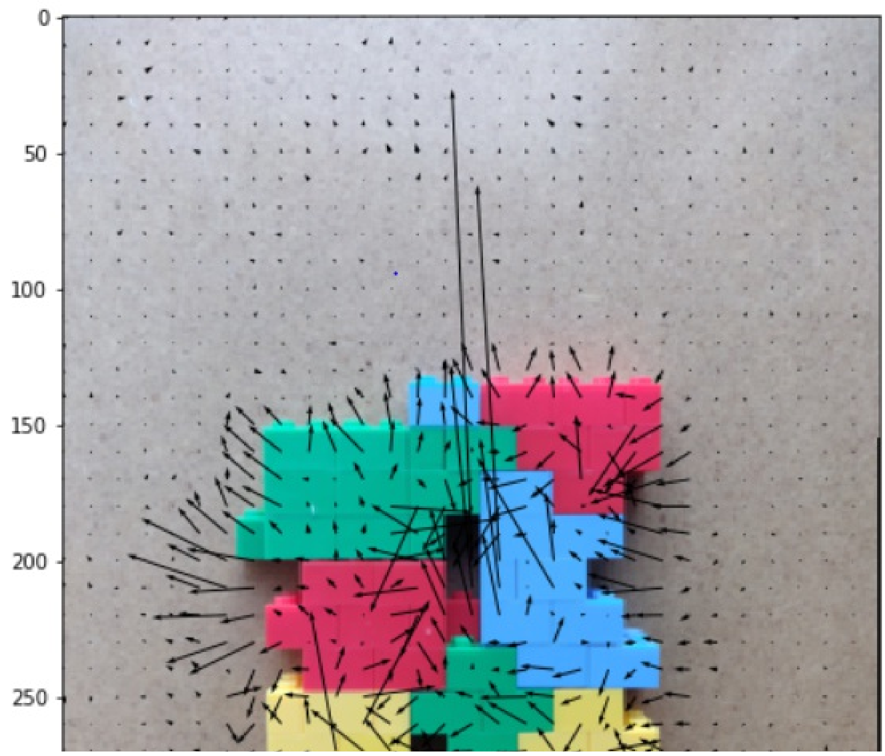
Instructions

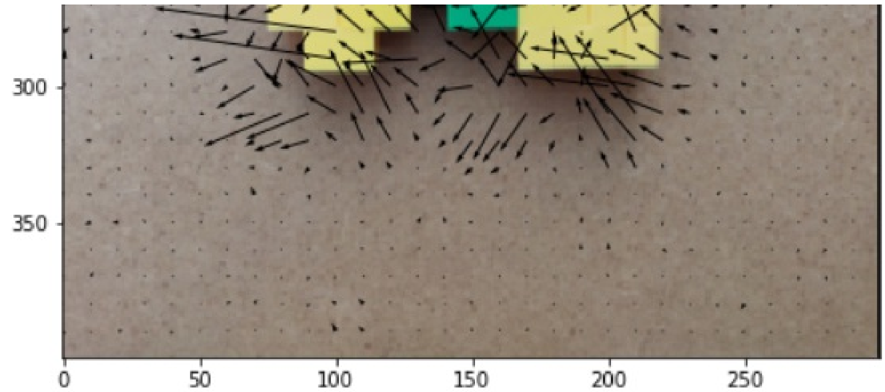
- Review the academic integrity and collaboration policies on the course website.
  - This assignment must be completed individually.
- All solutions must be written in this notebook.
  - Programming aspects of the assignment must be completed using Python in this notebook.
- If you want to modify the skeleton code, you may do so. It has only been provided as a framework for your solution.
- 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. 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: Optical Flow [14 pts]

In this problem, the multi-resolution Lucas-Kanade algorithm for estimating optical flow will be implemented, and the data needed for this problem can be found in the folder 'optical_flow_images'.

An example optical flow output is shown below - this is not a solution, just an example output.





Part 1: Multi-resolution Lucas-Kanade implementation [6 pts]

Implement the Lucas-Kanade method for estimating optical flow. The function 'LucasKanadeMultiScale' needs to be completed. You can implement 'upsample_flow' and 'OpticalFlowRefine' as 2 building blocks in order to complete this.

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import convolve as convolve
from skimage.transform import resize
import math

# from tqdm import tqdm_notebook

def grayscale(img):
    '''
    Converts RGB image to Grayscale
    '''
    gray=np.zeros((img.shape[0],img.shape[1]))
    gray=img[:,:,0]*0.2989+img[:,:,1]*0.5870+img[:,:,2]*0.1140
    return gray

def plot_optical_flow(img,U,V,titleStr):
    '''
    Plots optical flow given U,V and one of the images
    '''

    # Change t if required, affects the number of arrows
    # t should be between 1 and min(U.shape[0],U.shape[1])
    t=10

    # Subsample U and V to get visually pleasing output
    U1 = U[::t,::t]
    V1 = V[::t,::t]

    # Create meshgrid of subsampled coordinates
    r, c = img.shape[0],img.shape[1]
    cols,rows = np.meshgrid(np.linspace(0,c-1,c), np.linspace(0,r-1,r))
    cols = cols[::t,::t]
    rows = rows[::t,::t]

    # Plot optical flow
    plt.figure(figsize=(10,10))
    plt.imshow(img)
    plt.quiver(cols,rows,U1,V1)
    plt.title(titleStr)
    plt.show()


images=[]
for i in range(1,5):
    images.append(plt.imread('optical_flow_images/im'+str(i)+'.png')[:,:288,:])
# each image after converting to gray scale is of size -> 400x288
```

In [2]:

```python
# you can use interpolate from scipy
# You can implement 'upsample_flow' and 'OpticalFlowRefine'
# as 2 building blocks in order to complete this.
def upsample_flow(u_prev, v_prev):
    ''' You may implement this method to upsample optical flow from
    previous level
    u_prev, v_prev -> optical flow from prev level
    u, v -> upsampled optical flow to the current level
    '''
    """ ==========
    YOUR CODE HERE
    ========== """
    #print(u_prev)

    u = resize(u_prev,tuple(i*2 for i in u_prev.shape))
    v = resize(v_prev,tuple(i*2 for i in u_prev.shape))
    u = u*2
    v = v*2

    return u, v

def OpticalFlowRefine(im1, im2, window, u_prev=None, v_prev=None):
    '''
    Inputs: the two images at current level and window size
    u_prev, v_prev - previous levels optical flow
    Return u,v - optical flow at current level
    '''


    u = np.zeros(im1.shape)
    v = np.zeros(im1.shape)


    dx = np.array([[-1/2, 0, 1/2]])
    dy = np.array([[-1/2],[0],[1/2]])

    #If a previous optical flow level exists then this loop runs
    if u_prev is not None and v_prev is not None:
        u_prev, v_prev = upsample_flow(u_prev, v_prev) # upsample flow from previous level
        u = np.zeros(u_prev.shape)
        v = np.zeros(v_prev.shape)
    else:
        u_prev = np.zeros(u.shape)
        v_prev = np.zeros(v.shape)

    im1_rows = im1.shape[0]
    im1_cols = im1.shape[1]
    im1_trim_rows = im1_rows%window
    im1_trim_cols = im1_cols%window

    im2_rows = im2.shape[0]
    im2_cols = im2.shape[1]
    im2_trim_rows = im2_rows%window
    im2_trim_cols = im2_cols%window

    #Cutting off couple extra pixels from rows and cols so window can scan evenly
    im1 = im1[0:im1_rows-im1_trim_rows,0:im1_cols-im1_trim_cols]
    im2 = im2[0:im2_rows-im2_trim_rows,0:im2_cols-im2_trim_cols]


    # RESIZED AGAIN TO FIT WINDOW AS UP_SAMPLE WAS NOT PERFECTLY FITTING SIZE FOR WINDOW SCANNER EVENLY
    resized_im1_rows = im1.shape[0]
    resized_im1_cols = im1.shape[1]

#     print('Image size remainder: ', resized_im1_rows%window , resized_im1_cols%window)
#     print("\n")

    for i in range(int(resized_im1_rows/window)):
            for j in range(int(resized_im1_cols/window)):
                block_start_x = j*window
                block_start_y = i*window
                block_end_x = block_start_x + window
                block_end_y = block_start_y + window

                block = im1[block_start_y:block_end_y,block_start_x:block_end_x]
                I_x = convolve(block,dx,mode='same')
```

```
                I_y = -1*1*convolve(block,dy,mode='same')
                I_t = block - im2[block_start_y:block_end_y,block_start_x:block_end_x]

                A = np.zeros((2,2))
                A[0,0] = np.sum(np.square(I_x))
                A[0,1] = np.sum(I_x*I_y)
                A[1,0] = np.sum(I_x*I_y)
                A[1,1] = np.sum(np.square(I_y))

                A_inv = np.linalg.inv(A)
                b = np.zeros((2,1))
                b[0,0] = -1*np.sum(I_x*I_t)
                b[1,0] = -1*np.sum(I_y*I_t)
                uv = np.matmul(A_inv, b)

                u_value = uv[0,0]
                v_value = uv[1,0]

                u[block_start_y:block_end_y,block_start_x:block_end_x] = u_value
                v[block_start_y:block_end_y,block_start_x:block_end_x] = v_value


    u = u + u_prev
    v = v + v_prev

    return u, v
```

In [3]:

```
def LucasKanadeMultiScale(im1,im2,window, numLevels=2):
    '''
    Implement the multi-resolution Lucas kanade algorithm
    Inputs: the two images, window size and number of levels
    if numLevels = 1, then compute optical flow at only the given image level.
    Returns: u, v - the optical flow
    '''

    """ ==========
    YOUR CODE HERE
    ========== """
    # You can call OpticalFlowRefine iteratively

    #Down sampling for first iteration --- Don't need this anymore, handled in for loop at bottom
#     im1 = im1[::int(math.pow(2,numLevels)),::int(math.pow(2,numLevels))]
#     im2 = im2[::int(math.pow(2,numLevels)),::int(math.pow(2,numLevels))]

    #Used for calculating gradients via convolution
#     dx = np.array([[-1/2, 0, 1/2]])
#     dy = np.array([[-1/2],[0],[1/2]])

    im1_rows = im1.shape[0]
    im1_cols = im1.shape[1]
    im1_trim_rows = im1_rows%window
    im1_trim_cols = im1_cols%window

    im2_rows = im2.shape[0]
    im2_cols = im2.shape[1]
    im2_trim_rows = im2_rows%window
    im2_trim_cols = im2_cols%window

    #Cutting off couple extra pixels from rows and cols so window can scan evenly
    im1 = im1[0:im1_rows-im1_trim_rows,0:im1_cols-im1_trim_cols]
    im2 = im2[0:im2_rows-im2_trim_rows,0:im2_cols-im2_trim_cols]

    resized_im1_rows = im1.shape[0]
    resized_im1_cols = im1.shape[1]

    im1_shape = im1.shape
    u_prev = None
    v_prev = None

    for a in range(1, numLevels+1):

        u=np.zeros(im1_shape)
        v=np.zeros(im1_shape)
```

```
        #If numLevels is 1 then next_im1 and next_im2 are just the original image (downsamples by 2^(a-1)) which
        # is 2^0 for numLevels = 1
        next_im1 = im1[::int(math.pow(2,numLevels-a)),::int(math.pow(2,numLevels-a))]
        next_im2 = im2[::int(math.pow(2,numLevels-a)),::int(math.pow(2,numLevels-a))]
        im1_shape = next_im1.shape
        u_prev, v_prev = OpticalFlowRefine(im1=next_im1,im2=next_im2,window=window,u_prev=u_prev, v_prev=v_prev)
        u = u_prev
        v = v_prev
    return u, v
```

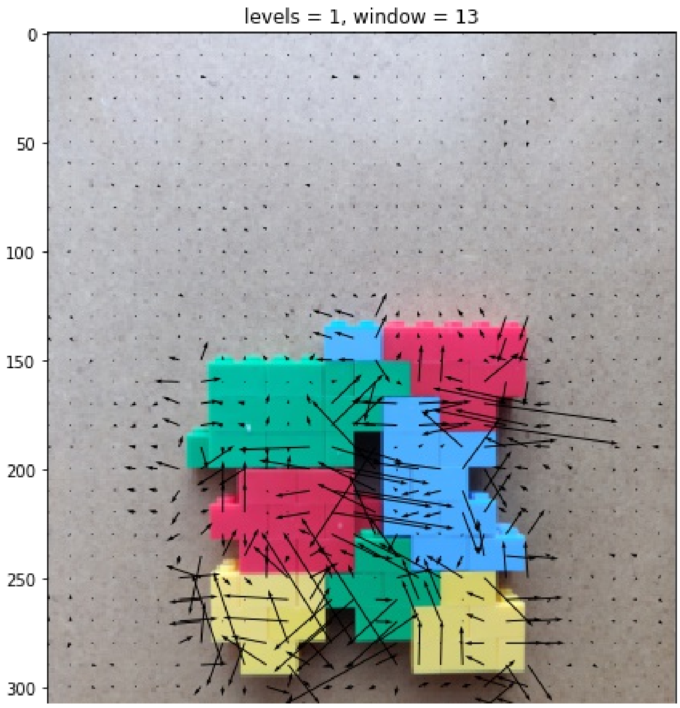
Part 2: Number of levels [2 pts]

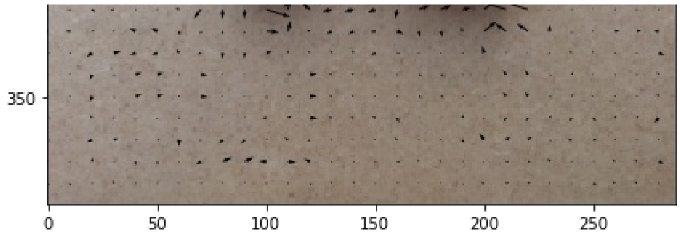
Plot optical flow for the pair of images im1 and im2 for different number of levels mentioned below. Comment on the results and justify.
(i) window size = 13, numLevels = 1
(ii) window size = 13, numLevels = 3
(iii) window size = 13, numLevels = 5
So, you are expected to provide 3 outputs here

Note: if numLevels = 1, then it means the optical flow is only computed at the image resolution i.e. no downsampling

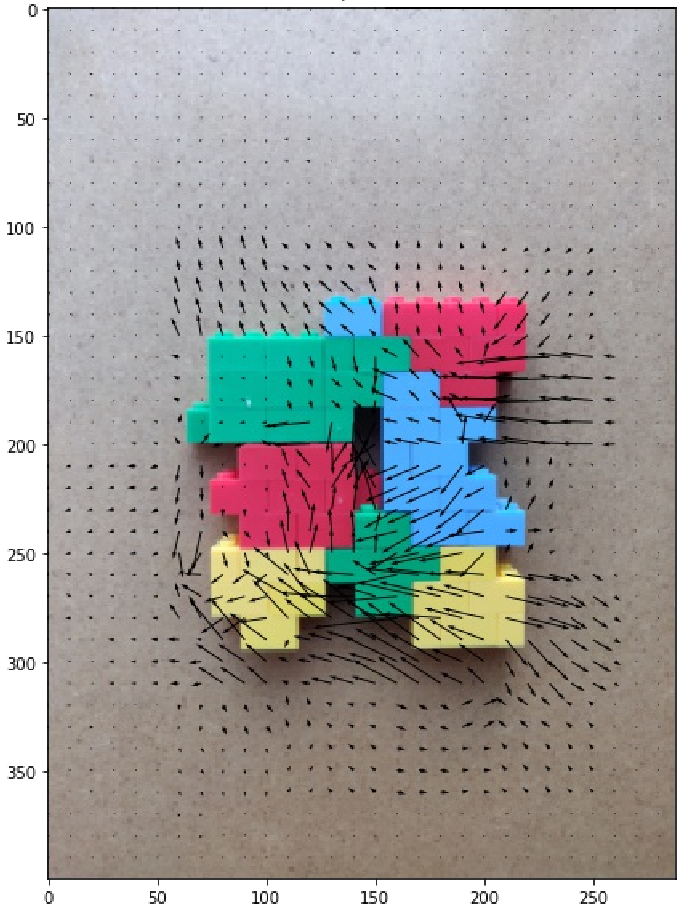
In [4]:

```
# Example code to generate output
window=13
numLevels=1
U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[1]),\
                          window,numLevels)
plot_optical_flow(images[0],U,V, \
                  'levels = ' + str(numLevels) + ', window = '+str(window))

numLevels=3
U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[1]),\
                          window,numLevels)
plot_optical_flow(images[0],U,V, \
                  'levels = ' + str(numLevels) + ', window = '+str(window))

numLevels=5
U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[1]),\
                          window,numLevels)
plot_optical_flow(images[0],U,V, \
                  'levels = ' + str(numLevels) + ', window = '+str(window))
```

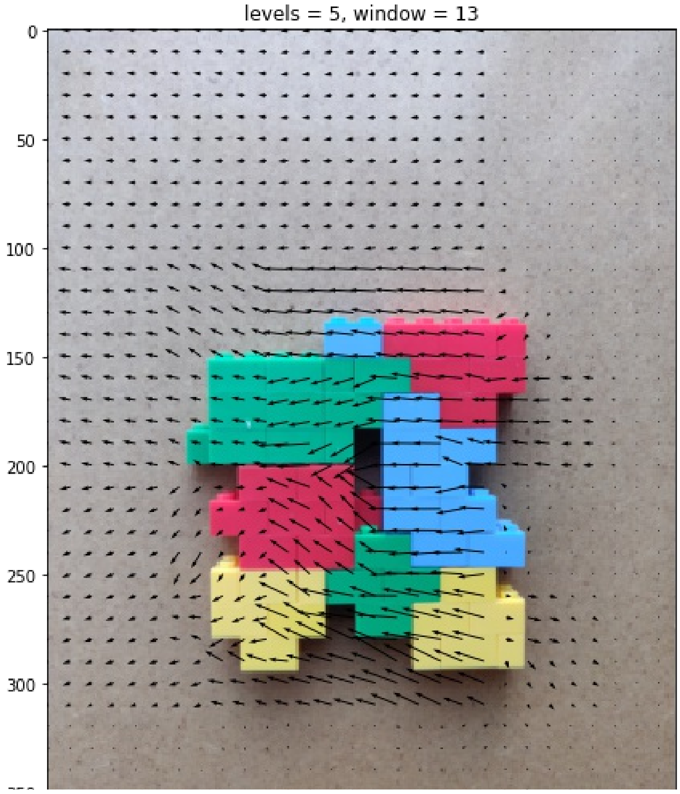


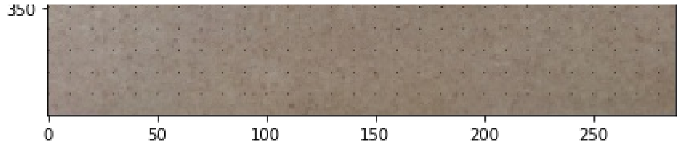


levels = 3, window = 13



levels = 5, window = 13





Your Comments on the results of Part 2:

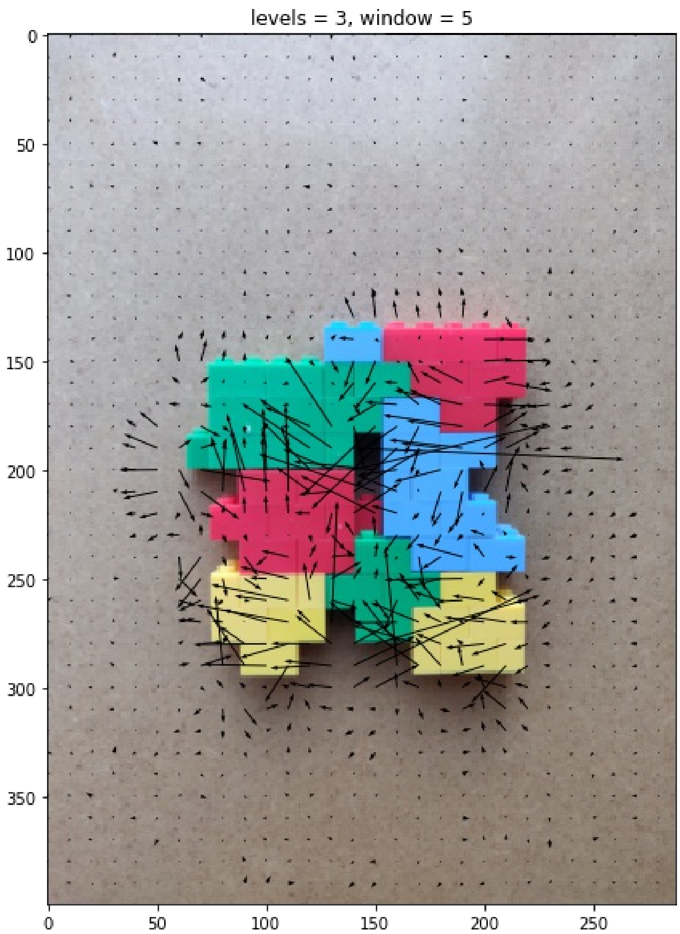
As the number of levels increases the optical flow becomes more smooth - i.e. it represents the motion of the block much better. By using the multi-resolution method one is able to detect the larger movements much better. If the movement of the object is large (the lucas kanade algorithm assumes very small movement) then the single level approach doesn't produce great results.

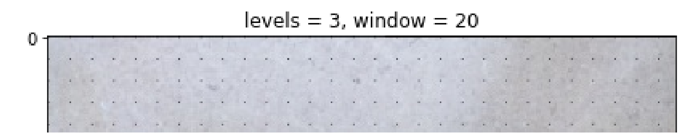
Part 3: Window size [3 pts]

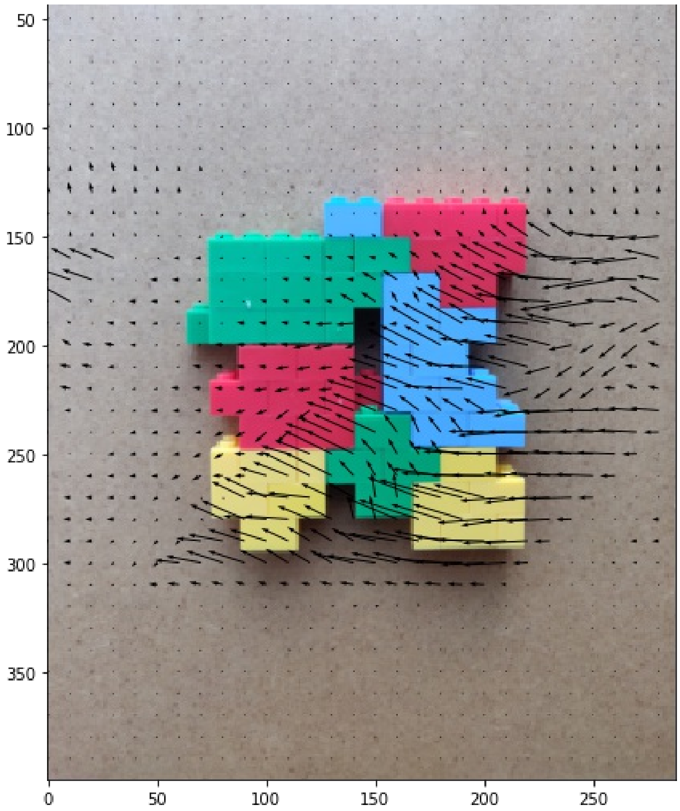
Plot optical flow for the pair of images im1 and im2 for at least 3 different window sizes which leads to observable difference in the results. Comment on the effect of window size on results and justify. For this part fix the number of levels to be 3.

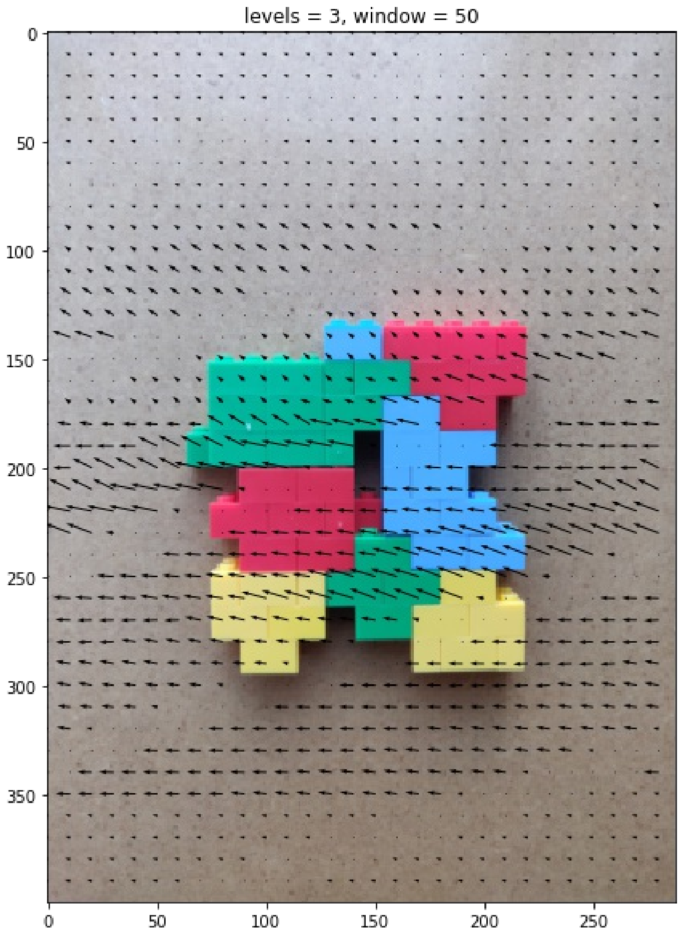
In [16]:

```
# Example code, change as required
numLevels=3

w1, w2, w3 = 5, 20, 50
for window in [w1, w2, w3]:
    U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[1]),\
                              window,numLevels)
    plot_optical_flow(images[0],U,V, \
                      'levels = ' + str(numLevels) + ', window = '+str(window))
```









Your Comments on the results of Part 3:

As the window size increases the optical flow becomes more representative of the lego block as a whole. However as the window size gets substantially large (i.e. the jump from the window size of 20 to 50) the movement of the foreground (lego block) and the background is less distinguished. In reality the foreground is moving and the background isn't - so a window size that encompasses the foreground enough to represent a more uniform motion of the block as a whole while not interpolating the movement of the background to move in the same way is ideal. The window size of 20 is displaying better results than the window size of 5 and 50 - so a bigger window size is not necessarily better and the ideal window size seems to depend on the size of the object one is tracking.

Part 4: All pairs [3 pts]

Find optical flow for the pairs (im1,im2), (im1,im3), (im1,im4) using one good window size and number of levels. Does the optical flow result seem consistent with visual inspection? Comment on the type of motion indicated by results and visual inspection and explain why they might be consistent or inconsistent.

In [12]:

```
# Your code here
# use one fixed window and numLevels for all pairs
window=13
numLevels=5

U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[1]),\
                          window,numLevels)
print("Image 1 to Image 2\n")
plot_optical_flow(images[0],U,V, \
                  'levels = ' + str(numLevels) + ', window = '+str(window))


U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[2]),\
                          window,numLevels)
print("Image 1 to Image 3\n")
plot_optical_flow(images[0],U,V, \
                  'levels = ' + str(numLevels) + ', window = '+str(window))


U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[3]),\
                          window,numLevels)
print("Image 1 to Image 4\n")
plot_optical_flow(images[0],U,V, \
                  'levels = ' + str(numLevels) + ', window = '+str(window))
```

```
Image 1 to Image 2
```

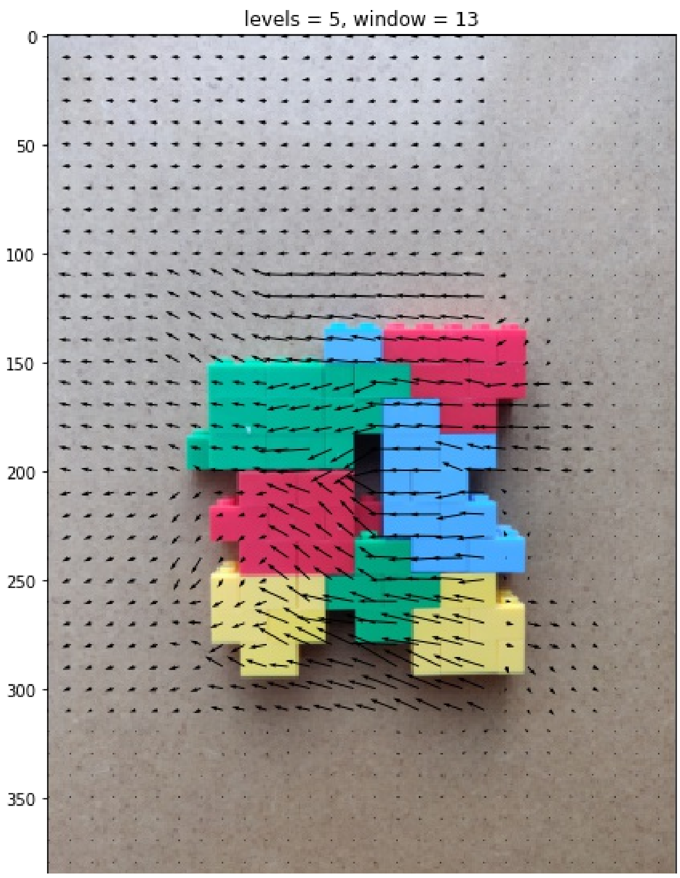


Image 1 to Image 3

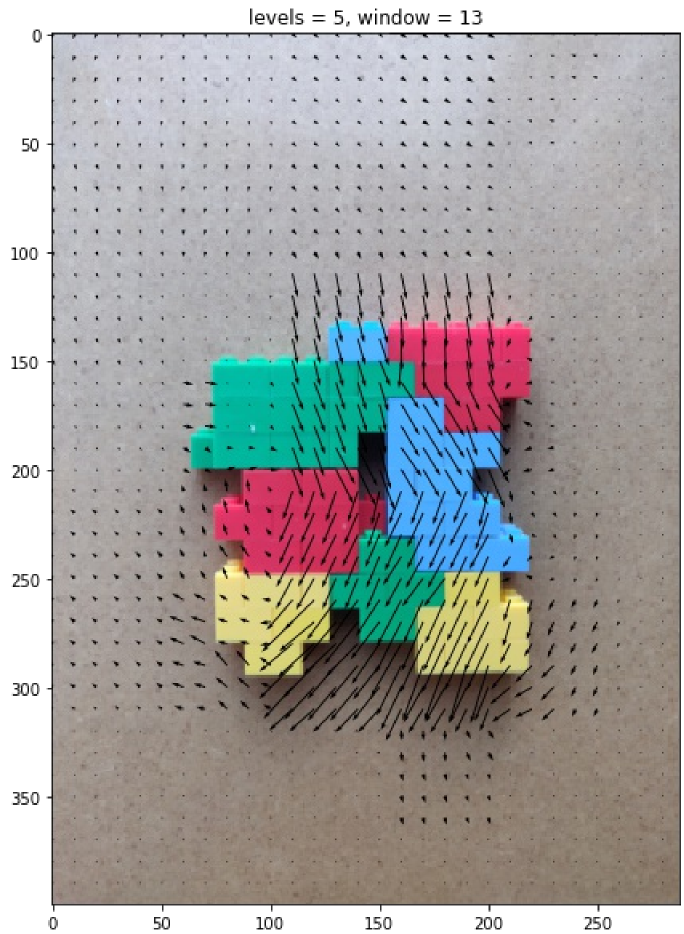
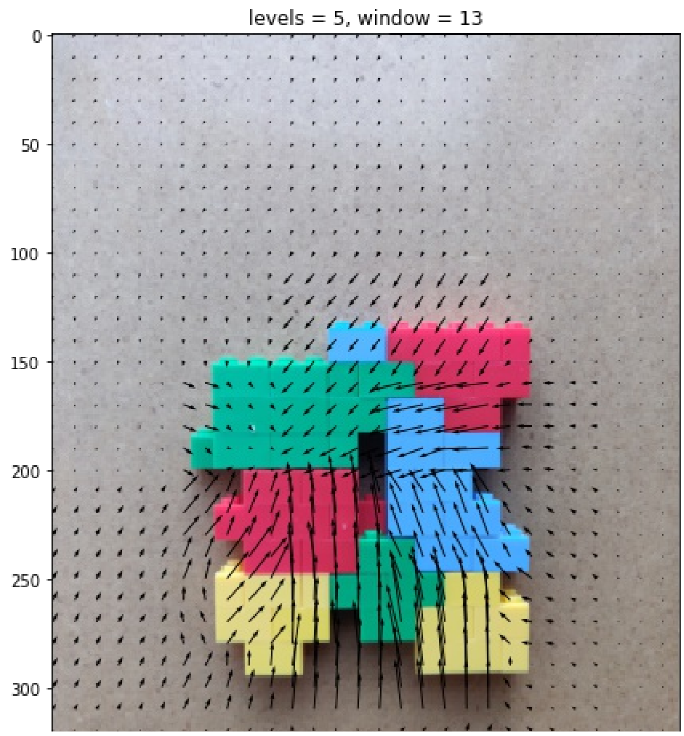
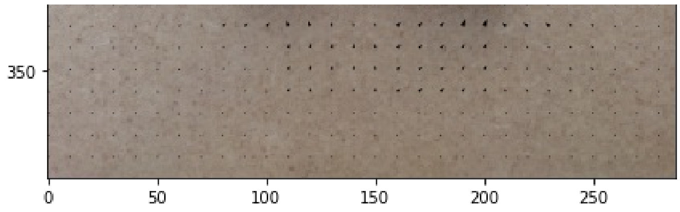


Image 1 to Image 4





Your Comments on the results of Part 4:

Upon visual inspection the movements of the image pairs is as follows:
(im1,im2) = translation to the left
(im1,im3) = clockwise rotation
(im1,im4) = zoom out

The optical flow for all of the pairs matches very well with my visual inspection. The window size of 13 seems to capture the overall movement of the lego block the best out of other window sizes I used. The number of levels was selected to be 5 as this captured the best results of the optical flow - this seems to be a good number of levels to capture the magnitude of movement. As I increased the number of levels the optical flow of the image as a whole was in the correct direction, but the movement of the block was not prioritized (i.e. the flow vectors represented the movement of the whole image not the block). As the number of levels was lower (in the range of 1 to 3) some of the larger movements of the block were missed and the flow field was not as smooth. From reading a paper on optical flow for zoomed in pictures, the optical flow vectors were pointing outward so it makes sense that for zooming out the optical flow vectors will point inward.

Problem 2: Machine Learning [12 pts]

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

Part 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.

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 [6]:

```
import torch
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([[0.1458, 0.7431, 0.8599],
        [0.4026, 0.3029, 0.7598],
        [0.8586, 0.3123, 0.2315],
        [0.6723, 0.4232, 0.7421],
        [0.9590, 0.3104, 0.2311]])
```

In [7]:

```
import os
```

```
import struct

# Change path as required
path = r"C:\Users\josep\OneDrive\Desktop\UCSD_Courses\CSE252A\hw5"

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')))
```

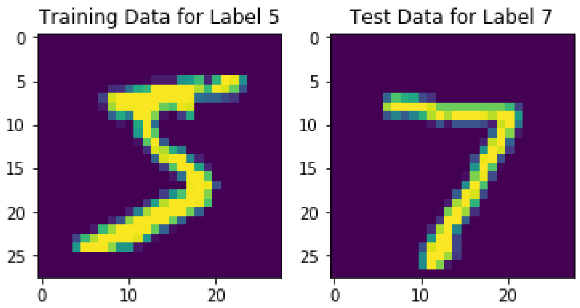
In [8]:

```
# Understand the shapes of the each variable carying data
print(trainData.shape, trainLabels.shape)
print(testData.shape, testLabels.shape)
```

```
(60000, 28, 28) (60000,)
(10000, 28, 28) (10000,)
```

In [9]:

```
# display one image from each label
# """ ==========
# YOUR CODE HERE
# ========== """
trainData_img = trainData[0,:,:]
testData_img = testData[0,:,:]
trainLabel = trainLabels[0]
testLabel = testLabels[0]

f = plt.figure()
f.add_subplot(1,2,1)
plt.imshow(trainData_img.reshape((28,28)))
plt.title('Training Data for Label ' + str(trainLabel))
f.add_subplot(1,2,2)
plt.imshow(testData_img.reshape((28,28)))
plt.title('Test Data for Label ' + str(testLabel))
plt.show()
```



Some helper functions are given below.

In [10]:

```
# a generator for batches of data
# yields data (batchsize, 28, 28) 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: 10.080000
```

Part 2: Confusion Matrix [2 pts]

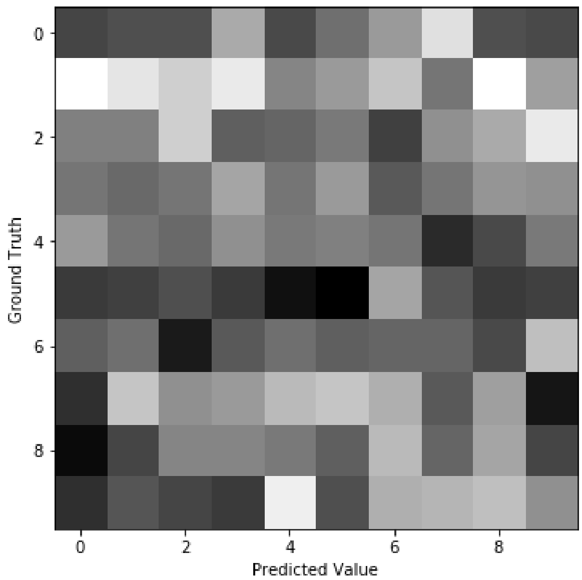
Here you will implement a function that computes the confusion matrix for a classifier. The matrix (M) should be nxn where n is the number of classes. Entry M[i,j] should contain the fraction of images of class i that was classified as class j. Can you justify the accuracy given by the random classifier?

In [11]:

```
# 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):
    M=np.zeros((10,10))
```

```
    acc=0.0
    """ ==========
    YOUR CODE HERE
    ========== """
    acc = test(testData, testLabels, classifier)
    batchsize=50
    correct=0.
    for data,label in DataBatch(testData,testLabels,batchsize,shuffle=False):
        prediction = classifier(data)
        for l in range(len(label)):
            M[label[l],prediction[l]] += 1
    M = M/testData.shape[0]*100
    return M, acc

def VisualizeConfusion(M):
    plt.figure(figsize=(14, 6))
    plt.imshow(M, cmap='gray')
    plt.ylabel('Ground Truth')
    plt.xlabel('Predicted Value')
    plt.show()
    print('Matrix w/ Rounded Values:\n')
    print(np.round(M,2))
    print("\nTrace of Confusion Matrix:", np.trace(M))


M,_ = Confusion(testData, testLabels, randomClassifier)
print('Accuracy: ', _)
print('\nVisualized Confusion Matrix')
VisualizeConfusion(M)
```

```
Accuracy:  10.12

Visualized Confusion Matrix
```




```
Matrix w/ Rounded Values:

[[0.9  0.92 0.92 1.09 0.91 0.98 1.06 1.19 0.92 0.91]
 [1.25 1.2  1.16 1.21 1.02 1.06 1.14 0.99 1.25 1.07]
 [1.01 1.01 1.16 0.95 0.96 1.   0.89 1.04 1.09 1.21]
 [0.99 0.97 0.99 1.08 0.99 1.06 0.94 0.99 1.05 1.04]
 [1.06 0.99 0.97 1.04 1.   1.01 0.99 0.85 0.91 1.  ]
 [0.88 0.89 0.92 0.88 0.8  0.77 1.08 0.93 0.88 0.89]
 [0.95 0.98 0.82 0.94 0.98 0.95 0.96 0.96 0.91 1.13]
 [0.86 1.14 1.04 1.06 1.12 1.14 1.1  0.94 1.07 0.81]
 [0.79 0.9  1.02 1.02 1.   0.95 1.12 0.96 1.08 0.9 ]
 [0.86 0.93 0.9  0.88 1.22 0.92 1.1  1.11 1.13 1.04]]

Trace of Confusion Matrix: 10.129999999999999
```

Your Comments on the accuracy & confusion matrix of random classifier:

The trace of the confusion matrix is equivalent to the total percentage of classification predictions that match the labels - i.e. the same thing as the accuracy of the classifier. The classifier is simply returning a random number between 0 and 9 - so it makes sense that the accuracy is going to be about 10 percent. The accuracy of a random classifier is 1/number_of_classes which in this case is 1/10 = 1 percent (note: the accuracy of each diagonal element is approximately 1 percent). Since the total accuracy is the sum of the accuracy results for each class this is 10 x 1 = 10! However, if we were referring to accuracy as the ability to predict the number 5, for example, then the answer would just be 1 percent.

Part 3: K-Nearest Neighbors (KNN) [4 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 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 '7' is most often predicted to be, other than '7'.

In [12]:

```
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
        ========== """
        self.trainData = trainData
        self.trainLabels = trainLabels

        self.neigh = KNeighborsClassifier(n_neighbors=self.k)

        data_reshaped = np.reshape(self.trainData, (self.trainData.shape[0],self.trainData.shape[1]*sel
f.trainData.shape[2]))
        label_reshaped = self.trainLabels
        self.neigh.fit(data_reshaped, label_reshaped)

    def __call__(self, x):
        # this method should take a batch of images
        # and return a batch of predictions
        """ ==========
        YOUR CODE HERE
        ========== """
        x_reshaped = np.reshape(x, (x.shape[0],x.shape[1]*x.shape[2]))
        prediction = self.neigh.predict(x_reshaped)
        return prediction
# 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 [15]:

```
# test your classifier with all the training examples (This may take a while)
import time
start = time.time()
knnClassifer = KNNClassifer()
knnClassifer.train(trainData, trainLabels)
print ('KNN classifier accuracy: %f'%test(testData, testLabels, knnClassifer))
end = time.time()
elapsed_time = (end - start)/60 #in minutes
```

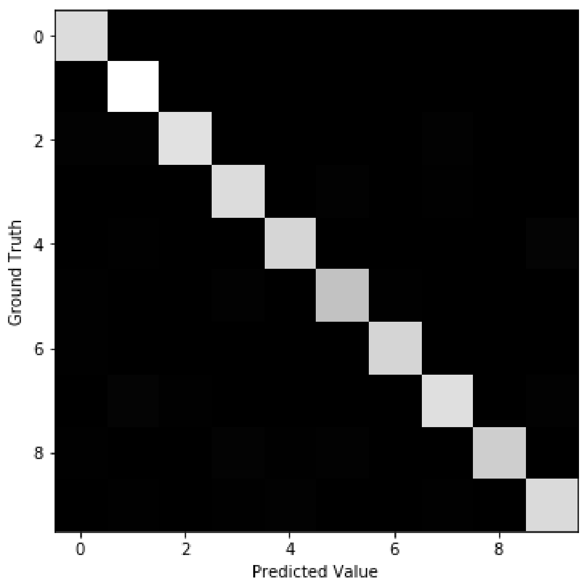
```
print('Computation time in minutes: ' + str(elapsed_time))
```

```
KNN classifier accuracy: 97.050000
Computation time in minutes: 14.77816078265508
```

In [18]:

```
# display confusion matrix for your KNN classifier with all the training examples
# (This may take a while)
""" ==========
YOUR CODE HERE
========== """
M,_ = Confusion(testData, testLabels, knnClassifer)
print('Accuracy: ', _)
print('\nVisualized Confusion Matrix')
VisualizeConfusion(M)



seventh_row = list(M[7,:])
sorted_seventh_row = sorted(seventh_row, reverse=True)
second_most_predicted_val = sorted_seventh_row[1]
index_of_val = seventh_row.index(second_most_predicted_val)
print("\nThe second most predicted value for the number 7 was: " + str(index_of_val))
```

```
Accuracy:  97.05

Visualized Confusion Matrix
```




```
Matrix w/ Rounded Values:

[[9.740e+00 1.000e-02 1.000e-02 0.000e+00 0.000e+00 1.000e-02 2.000e-02
  1.000e-02 0.000e+00 0.000e+00]
 [0.000e+00 1.133e+01 2.000e-02 0.000e+00 0.000e+00 0.000e+00 0.000e+00
  0.000e+00 0.000e+00 0.000e+00]
 [1.000e-01 9.000e-02 9.960e+00 2.000e-02 0.000e+00 0.000e+00 0.000e+00
  1.300e-01 2.000e-02 0.000e+00]
 [0.000e+00 2.000e-02 4.000e-02 9.760e+00 1.000e-02 1.300e-01 1.000e-02
  7.000e-02 3.000e-02 3.000e-02]
 [1.000e-02 6.000e-02 0.000e+00 0.000e+00 9.500e+00 0.000e+00 4.000e-02
  2.000e-02 0.000e+00 1.900e-01]
 [6.000e-02 1.000e-02 0.000e+00 1.100e-01 2.000e-02 8.590e+00 5.000e-02
  1.000e-02 3.000e-02 4.000e-02]
 [5.000e-02 3.000e-02 0.000e+00 0.000e+00 3.000e-02 3.000e-02 9.440e+00
  0.000e+00 0.000e+00 0.000e+00]
 [0.000e+00 2.100e-01 5.000e-02 0.000e+00 1.000e-02 0.000e+00 0.000e+00
  9.910e+00 0.000e+00 1.000e-01]
```

```
 [8.000e-02 2.000e-02 4.000e-02 1.600e-01 8.000e-02 1.100e-01 3.000e-02
  4.000e-02 9.140e+00 4.000e-02]
 [4.000e-02 5.000e-02 2.000e-02 8.000e-02 9.000e-02 2.000e-02 1.000e-02
  8.000e-02 2.000e-02 9.680e+00]]

Trace of Confusion Matrix: 97.04999999999998

The second most predicted value for the number 7 was: 1
```

Answer

The code for finding the value that the number 7 was most often mistaken for is in the cell above. Since ground truth is on the y_axis, the 2nd highest value in the 7th row correlates to the number that the number 7 was most often mistaken for.

7 was most often mistaken to be the number 1

Part 4: Principal Component Analysis (PCA) K-Nearest Neighbors (KNN) [5 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 [54]:

```
def calculate_mean_image(images_vector): #calculates mean of all matrices that represent images
    num_of_images = images_vector.shape[0]
    image_summation = np.zeros(images_vector[0].shape)
    for i in range(num_of_images):
        image_summation = image_summation + images_vector[i,:,:]

    mean_image = image_summation/num_of_images

    return mean_image


def calculate_mean_adjusted_images(images_vector, mean_image):
    mean_adjusted_images = np.zeros(images_vector.shape)
    num_of_images = images_vector.shape[0]
    for i in range(num_of_images):
        mean_adjusted_images[i,:,:] = images_vector[i,:,:] - mean_image

    return mean_adjusted_images


def calculate_covariance_matrix(mean_adjusted_images):
    num_of_images = mean_adjusted_images.shape[0]
    first_image = mean_adjusted_images[0] #used for initializing AAT_image_summation shape
    AAT = np.zeros((first_image.shape[0], first_image.shape[0])) # where A is a mean adjusted image mat
rix

    for i in range(num_of_images):
        image = mean_adjusted_images[i,:,:]
        AAT = AAT + np.matmul(image, image.T) #if image is 193*162 then AAT_image_summation is 193*193

    return AAT

def calculate_eigenpairs(covariance_matrix):
    eigenvalues, eigenvectors = np.linalg.eig(covariance_matrix)

    return eigenvalues, eigenvectors


def select_image_space(eigenvalues, eigenvectors, num_of_images):
    sorted_eigenvalues = sorted(eigenvalues, reverse=True)
    image_space_eigenvectors = np.zeros((eigenvectors.shape[0], num_of_images)) #each column is an eige
nvector
    image_space_eigenvalues = np.zeros(num_of_images)
    for i in range(num_of_images):
        eigenvalue = sorted_eigenvalues[i]
```

```python
        corresponding_eigenvector_index = np.where(eigenvalues == eigenvalue)[0][0]

        corresponding_eigenvector = eigenvectors[:, corresponding_eigenvector_index]

        image_space_eigenvectors[:,i] = corresponding_eigenvector
        image_space_eigenvalues[i] = eigenvalue

    return image_space_eigenvectors, image_space_eigenvalues


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
        ========== """

        self.trainData = trainData
        self.trainLabels = trainLabels
        #print(self.trainData.shape)
        PCA_projections = np.zeros((self.trainData.shape[0], self.components, self.trainData.shape[1]))

        self.neigh = KNeighborsClassifier(n_neighbors=self.k)


        mean_data = calculate_mean_image(self.trainData)
        mean_adjusted_images = calculate_mean_adjusted_images(self.trainData, mean_data)
        covariance_matrix = calculate_covariance_matrix(mean_adjusted_images)
        eigenvalues, eigenvectors = calculate_eigenpairs(covariance_matrix)
        PCA_vectors, PCA_values = select_image_space(eigenvalues, eigenvectors, self.components)

        for i in range(PCA_projections.shape[0]):
            PCA_projection = np.matmul(PCA_vectors.T,self.trainData[i,:,:].T)
            PCA_projections[i,:,:] = PCA_projection

        data_reshaped = np.reshape(PCA_projections, (PCA_projections.shape[0],PCA_projections.shape[1]*
PCA_projections.shape[2]))
        label_reshaped = self.trainLabels

        self.neigh.fit(data_reshaped, label_reshaped)

        return PCA_vectors
    def __call__(self, x):
        # this method should take a batch of images
        # and return a batch of predictions
        """ ==========
        YOUR CODE HERE
        ========== """
        x_reshaped = np.reshape(x, (x.shape[0],x.shape[1]*x.shape[2]))
        prediction = self.neigh.predict(x_reshaped)
        return prediction


# test your classifier with only the first 100 training examples (use this
# while debugging)
pcaknnClassiferX = PCAKNNClassifer()
PCA_vectors = pcaknnClassiferX.train(trainData[:100], trainLabels[:100])
testData_new = np.zeros((testData.shape[0],25,testData.shape[1]))
for i in range(testData.shape[0]):
    PCA_projection = np.matmul(PCA_vectors.T,testData[i,:,:].T)
    testData_new[i,:,:] = PCA_projection

print ('PCAKNN classifier accuracy: %f'%test(testData_new, testLabels, pcaknnClassiferX))
```

```
PCAKNN classifier accuracy: 64.760000
```

In [57]:

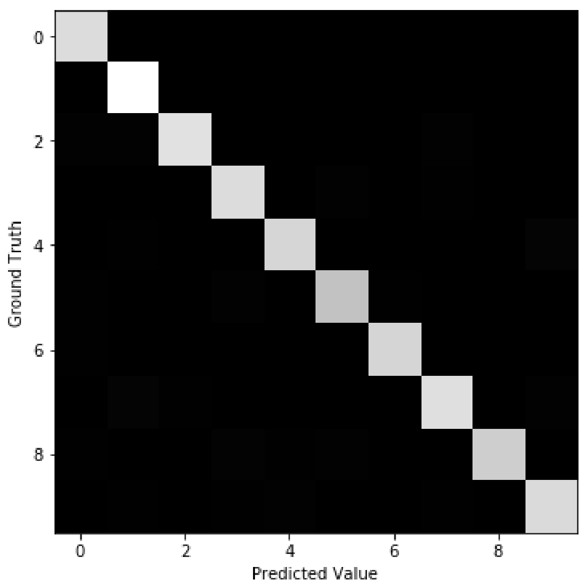
```python
# test your classifier with all the training examples
start = time.time()
pcaknnClassifer = PCAKNNClassifer()
PCA_vectors = pcaknnClassifer.train(trainData, trainLabels)
testData_new = np.zeros((testData.shape[0],25,testData.shape[1]))
for i in range(testData.shape[0]):
    PCA_projection = np.matmul(PCA_vectors.T,testData[i,:,:].T)
    testData_new[i,:,:] = PCA_projection

print ('PCAKNN classifier accuracy: %f'%test(testData_new, testLabels, pcaknnClassifer))
end = time.time()
elapsed_time = (end - start)/60 #in minutes
print('Computation time in minutes: ' + str(elapsed_time))
```

```
PCAKNN classifier accuracy: 97.050000
Computation time in minutes: 12.926141730944316
```

In [59]:

```python
# display confusion matrix for your PCA KNN classifier with all the training examples
""" ==========
YOUR CODE HERE
========== """
M,_ = Confusion(testData_new, testLabels, pcaknnClassifer)
print('Accuracy: ', _)
print('\nVisualized Confusion Matrix')
VisualizeConfusion(M)
```

```
Accuracy:  97.05

Visualized Confusion Matrix
```




```
Matrix w/ Rounded Values:

[[9.740e+00 1.000e-02 1.000e-02 0.000e+00 0.000e+00 1.000e-02 2.000e-02
  1.000e-02 0.000e+00 0.000e+00]
 [0.000e+00 1.133e+01 2.000e-02 0.000e+00 0.000e+00 0.000e+00 0.000e+00
  0.000e+00 0.000e+00 0.000e+00]
 [1.000e-01 9.000e-02 9.960e+00 2.000e-02 0.000e+00 0.000e+00 0.000e+00
  1.300e-01 2.000e-02 0.000e+00]
 [0.000e+00 2.000e-02 4.000e-02 9.760e+00 1.000e-02 1.300e-01 1.000e-02
  7.000e-02 3.000e-02 3.000e-02]
 [1.000e-02 6.000e-02 0.000e+00 0.000e+00 9.500e+00 0.000e+00 4.000e-02
  2.000e-02 0.000e+00 1.900e-01]
 [6.000e-02 1.000e-02 0.000e+00 1.100e-01 2.000e-02 8.500e+00 5.000e-02
```

```
[6.000e-02 1.000e-02 0.000e+00 1.100e-01 2.000e-02 8.590e+00 5.000e-02
 1.000e-02 3.000e-02 4.000e-02]
 [5.000e-02 3.000e-02 0.000e+00 0.000e+00 3.000e-02 3.000e-02 9.440e+00
 0.000e+00 0.000e+00 0.000e+00]
 [0.000e+00 2.100e-01 5.000e-02 0.000e+00 1.000e-02 0.000e+00 0.000e+00
 9.910e+00 0.000e+00 1.000e-01]
 [8.000e-02 2.000e-02 4.000e-02 1.600e-01 8.000e-02 1.100e-01 3.000e-02
 4.000e-02 9.140e+00 4.000e-02]
 [4.000e-02 5.000e-02 2.000e-02 8.000e-02 9.000e-02 2.000e-02 1.000e-02
 8.000e-02 2.000e-02 9.680e+00]]

Trace of Confusion Matrix: 97.04999999999998
```

Comments:

The PCAKNN classifier performs quicker than the KNN which makes sense as the point of PCA is to create a smaller dimension space. A smaller dimension space means less computation required. Although the dimension is smaller the PCAs extract the "beneficial" information (i.e. most distinctive eigenvectors that capture the most variance) so the accuracy is still competitive (in this case it was actually identical).

Problem 3: Deep learning [14 pts]

Below is some helper code to train your deep networks.

Part 1: Training with PyTorch [2 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 [60]:

```
# base class for your deep neural networks. It implements the training loop (train_net).
# You will need to implement the "__init__()" function to define the networks
# structures and "forward()", to propagate your data, in the following problems.

import torch.nn.init
import torch.optim as optim
from torch.autograd import Variable
from torch.nn.parameter import Parameter
from tqdm import tqdm
from scipy.stats import truncnorm
import torch.nn.functional as torch_functional

class DNN(nn.Module):
    def __init__(self):
        super(DNN, self).__init__()
        dtype = torch.float
        device = torch.device("cpu")


    def forward(self, x): # the classes that inherit DNN override this method with their own forward me
thod
        pass

    def train_net(self, trainData, trainLabels, epochs=1, batchSize=50):
        criterion = nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(self.parameters(), lr = 3e-4)

        for epoch in range(epochs):
            self.train()  # set netowrk in training mode
            for i, (data,labels) in enumerate(DataBatch(trainData, trainLabels, batchSize, shuffle=True
)):
                data = Variable(torch.FloatTensor(data))
                labels = Variable(torch.LongTensor(labels))

                # YOUR CODE HERE-------------------------------------------------
                # Train the model using the optimizer and the batch data
```

```
                # Train the model using the optimizer and the batch data
                y_pred = self.forward(data)
                loss = criterion(y_pred, labels)
                optimizer.zero_grad()
                loss.backward()
                optimizer.step()
                #--------------------------------------------------------------
                #-----End of your code, don't change anything else here------------

            self.eval()  # set network in evaluation mode
            print ('Epoch:%d Accuracy: %f'%(epoch+1, test(testData, testLabels, self)))

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

# helper function to get weight variable
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)

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

In [61]:

```
# example linear classifier - input connected to output
# you can take this as an example to learn how to extend DNN class
class LinearClassifier(DNN):
    def __init__(self, in_features=28*28, classes=10):
        super(LinearClassifier, self).__init__()
        # in_features=28*28
        self.weight1 = weight_variable((classes, in_features))
        self.bias1 = bias_variable((classes))

    def forward(self, x):
        # linear operation
        y_pred = torch.addmm(self.bias1, x.view(list(x.size())[0], -1), self.weight1.t())
        return y_pred

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'))))
```

In [62]:

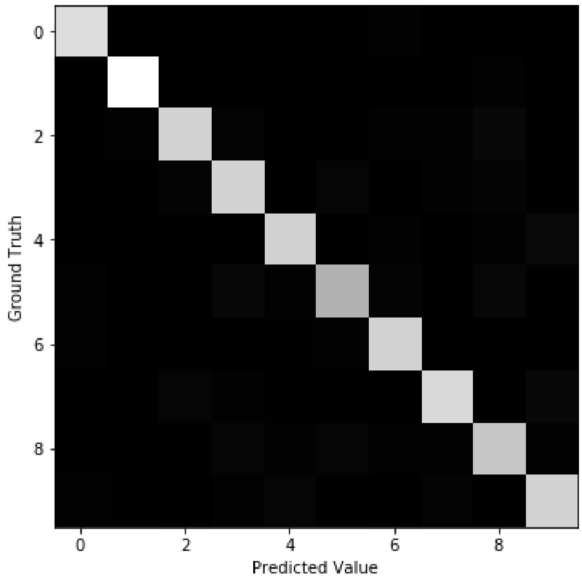
```
# test the example linear classifier (note you should get around 90% accuracy
# for 10 epochs and batchsize 50)
linearClassifier = LinearClassifier()
linearClassifier.train_net(trainData, trainLabels, epochs=10)
```

```
Epoch:1 Accuracy: 89.030000
Epoch:2 Accuracy: 90.800000
Epoch:3 Accuracy: 91.340000
Epoch:4 Accuracy: 91.580000
Epoch:5 Accuracy: 91.740000
Epoch:6 Accuracy: 92.000000
Epoch:7 Accuracy: 92.220000
Epoch:8 Accuracy: 92.260000
Epoch:9 Accuracy: 92.450000
Epoch:10 Accuracy: 92.370000
```

In [63]:

```
# display confusion matrix
""" ==========
YOUR CODE HERE
========== """
M,_ = Confusion(testData, testLabels, linearClassifier)
print('Accuracy: ', _)
print('\nVisualized Confusion Matrix')
VisualizeConfusion(M)
```

```
Accuracy:  92.36999999999999

Visualized Confusion Matrix
```



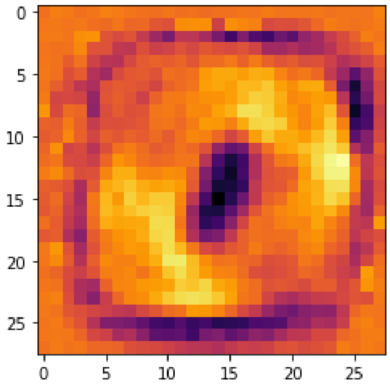

```
Matrix w/ Rounded Values:

[[9.610e+00 0.000e+00 1.000e-02 1.000e-02 0.000e+00 4.000e-02 9.000e-02
  2.000e-02 2.000e-02 0.000e+00]
 [0.000e+00 1.112e+01 2.000e-02 2.000e-02 0.000e+00 2.000e-02 4.000e-02
  2.000e-02 1.100e-01 0.000e+00]
 [7.000e-02 1.000e-01 9.200e+00 1.900e-01 8.000e-02 2.000e-02 1.300e-01
  1.000e-01 3.800e-01 5.000e-02]
 [3.000e-02 0.000e+00 1.800e-01 9.250e+00 0.000e+00 2.300e-01 2.000e-02
  1.000e-01 2.100e-01 8.000e-02]
 [1.000e-02 1.000e-02 5.000e-02 1.000e-02 9.090e+00 0.000e+00 1.300e-01
  2.000e-02 1.000e-01 4.000e-01]
 [9.000e-02 2.000e-02 3.000e-02 3.300e-01 9.000e-02 7.720e+00 1.800e-01
  5.000e-02 3.400e-01 7.000e-02]
 [1.100e-01 3.000e-02 6.000e-02 1.000e-02 8.000e-02 1.000e-01 9.150e+00
  2.000e-02 2.000e-02 0.000e+00]
 [1.000e-02 6.000e-02 2.200e-01 1.000e-01 5.000e-02 0.000e+00 0.000e+00
  9.430e+00 2.000e-02 3.900e-01]
 [7.000e-02 6.000e-02 7.000e-02 2.500e-01 9.000e-02 2.600e-01 1.100e-01
  1.200e-01 8.610e+00 1.000e-01]
 [1.000e-01 7.000e-02 2.000e-02 1.200e-01 2.400e-01 8.000e-02 0.000e+00
  2.100e-01 6.000e-02 9.190e+00]]

Trace of Confusion Matrix: 92.36999999999999
```

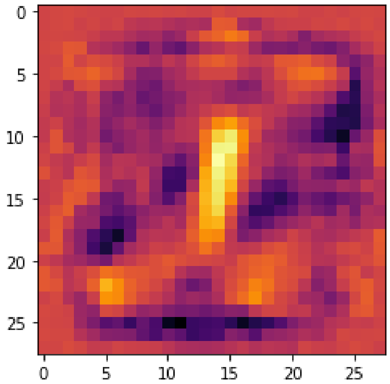
Part 2: Single Layer Perceptron [2 pts]

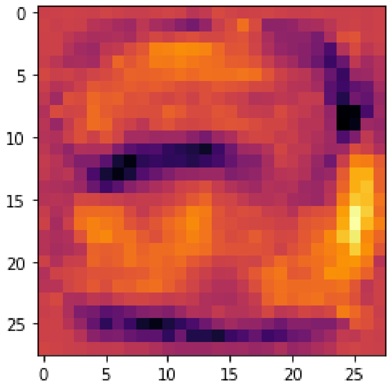
The simple linear classifier implemented in the cell already performs quite well. Plot the filter weights corresponding to each output class (weights, not biases) as images. (Normalize weights to lie between 0 and 1 and use color maps like 'inferno' or 'plasma' for good results). Comment on what the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

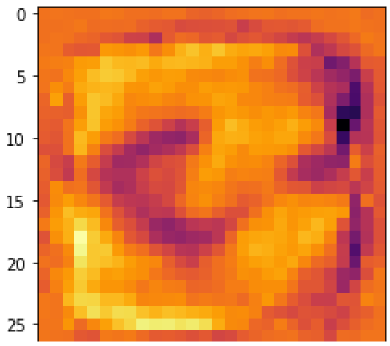
In [64]:

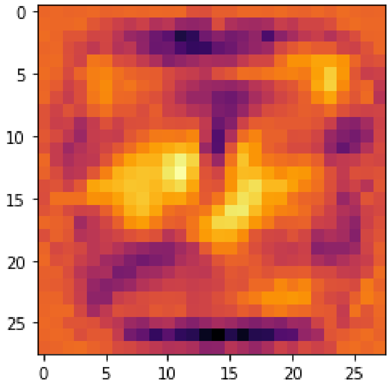
```
# Plot filter weights corresponding to each class, you may have to reshape them to make sense out of th
em
# linearClassifier.weight1.data will give you the first layer weights
weights = linearClassifier.weight1.data

f = plt.figure()

for weight in weights:
    weight = weight.view(28,28)
    plt.imshow(weight, cmap='inferno')
    plt.show()
```

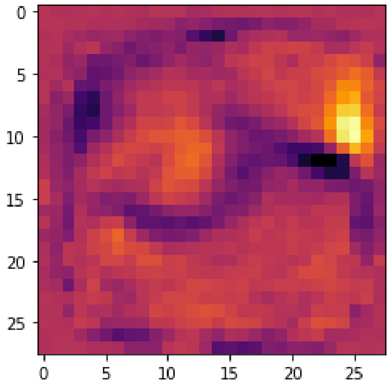


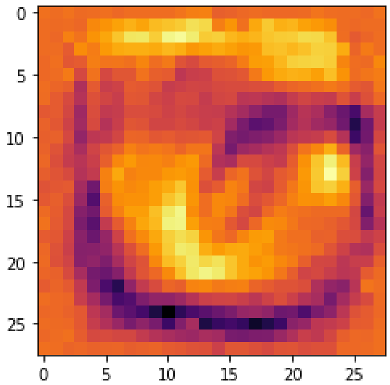


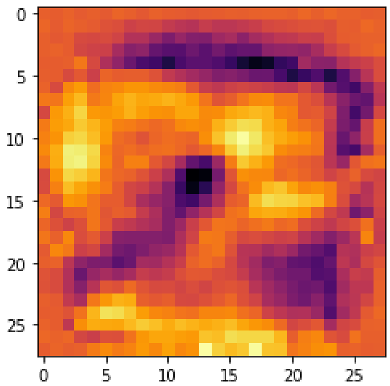


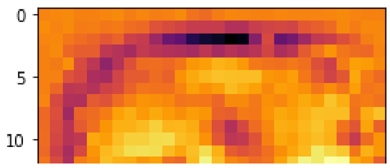


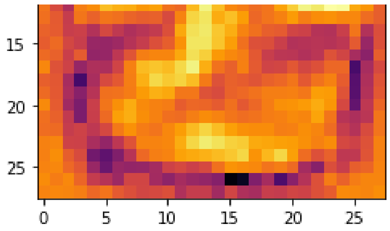


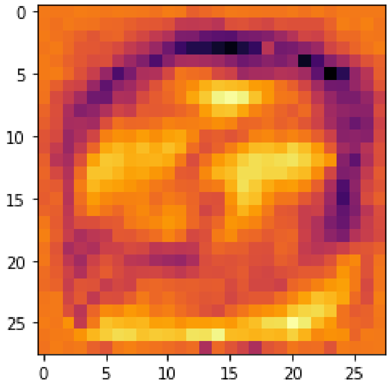












Comments on weights

They look like numbers!!! This makes sense as a weight given to a number that looks like the weight will be given priority over numbers that don't look like the weight.

Part 3: Multi Layer Perceptron (MLP) [5 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 [65]:

```
class MLPClassifer(DNN):
    def __init__(self, in_features=28*28, classes=10, hidden=100):
        """ ==========
        YOUR CODE HERE
        ========== """
        super(MLPClassifer, self).__init__()

        dtype = torch.float
        device = torch.device("cpu")

        ## first layer
        self.weight1 = weight_variable((hidden, in_features))
        self.bias1 = bias_variable((hidden))

        ## second layer
        self.weight2 = weight_variable((classes, hidden))
        self.bias2 = bias_variable((classes))



    def forward(self, x):
        """ ==========
```

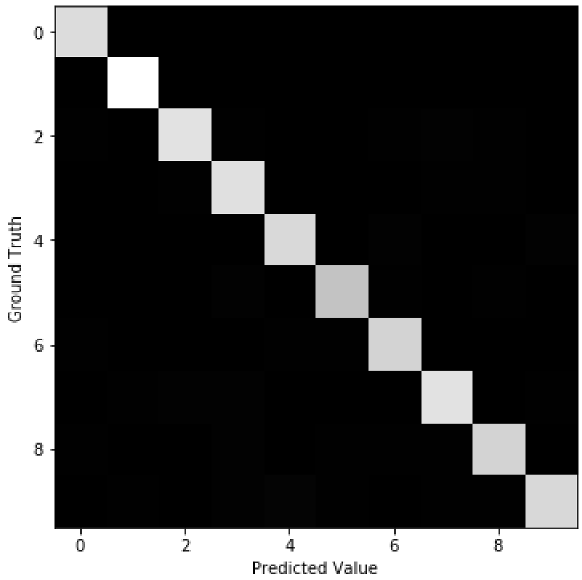
```
        YOUR CODE HERE
        ========== """
        y_pred = torch.addmm(self.bias1, x.view(list(x.size())[0], -1), self.weight1.t())
        y_pred = torch_functional.relu(y_pred)
        y_pred = torch.addmm(self.bias2, y_pred.view(list(x.size())[0], -1), self.weight2.t())
        return y_pred

mlpClassifer = MLPClassifer()
mlpClassifer.train_net(trainData, trainLabels, epochs=10, batchSize=50)
```

```
Epoch:1 Accuracy: 91.520000
Epoch:2 Accuracy: 93.060000
Epoch:3 Accuracy: 94.310000
Epoch:4 Accuracy: 94.970000
Epoch:5 Accuracy: 95.650000
Epoch:6 Accuracy: 96.020000
Epoch:7 Accuracy: 96.260000
Epoch:8 Accuracy: 96.470000
Epoch:9 Accuracy: 96.730000
Epoch:10 Accuracy: 96.830000
```

In [66]:

```
# Plot confusion matrix

M,_ = Confusion(testData, testLabels, mlpClassifer)
print('Accuracy: ', _)
print('\nVisualized Confusion Matrix')
VisualizeConfusion(M)
```

```
Accuracy:  96.83

Visualized Confusion Matrix
```

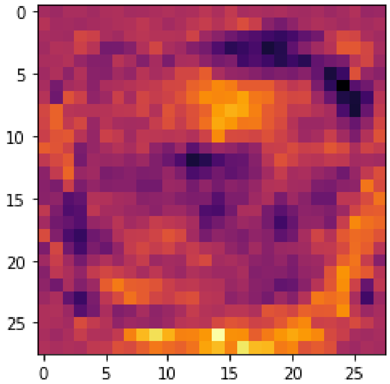


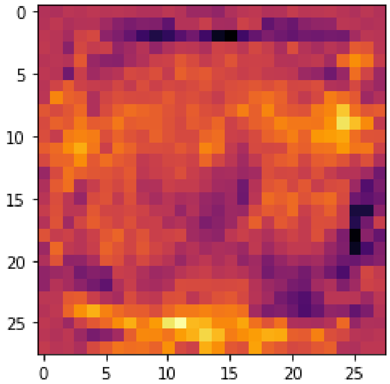

```
Matrix w/ Rounded Values:

[[9.670e+00 0.000e+00 2.000e-02 2.000e-02 0.000e+00 3.000e-02 3.000e-02
  1.000e-02 1.000e-02 1.000e-02]
 [0.000e+00 1.122e+01 4.000e-02 0.000e+00 0.000e+00 1.000e-02 2.000e-02
  2.000e-02 4.000e-02 0.000e+00]
 [7.000e-02 3.000e-02 9.920e+00 6.000e-02 4.000e-02 1.000e-02 5.000e-02
  9.000e-02 5.000e-02 0.000e+00]
 [0.000e+00 0.000e+00 7.000e-02 9.820e+00 1.000e-02 3.000e-02 0.000e+00
  8.000e-02 7.000e-02 2.000e-02]
 [0.000e+00 0.000e+00 4.000e-02 0.000e+00 9.530e+00 0.000e+00 9.000e-02
  4.000e-02 2.000e-02 1.000e-01]
 [2.000e-02 1.000e-02 0.000e+00 1.300e-01 1.000e-02 8.570e+00 6.000e-02
  2.000e-02 7.000e-02 3.000e-02]
```

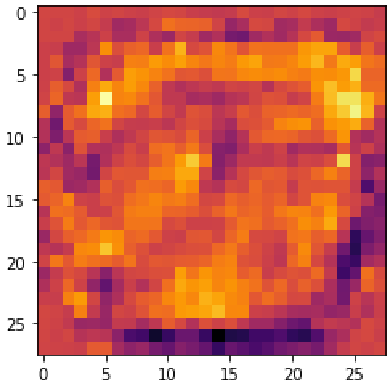
```
 [5.000e-02 3.000e-02 2.000e-02 1.000e-02 5.000e-02 5.000e-02 9.330e+00
  0.000e+00 4.000e-02 0.000e+00]
 [1.000e-02 5.000e-02 9.000e-02 9.000e-02 2.000e-02 1.000e-02 0.000e+00
  9.940e+00 1.000e-02 6.000e-02]
 [5.000e-02 3.000e-02 2.000e-02 9.000e-02 4.000e-02 5.000e-02 5.000e-02
  6.000e-02 9.330e+00 2.000e-02]
 [4.000e-02 5.000e-02 1.000e-02 1.100e-01 1.900e-01 5.000e-02 1.000e-02
  7.000e-02 6.000e-02 9.500e+00]]

Trace of Confusion Matrix: 96.83
```

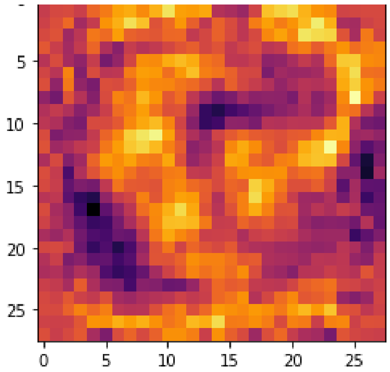
In [70]:

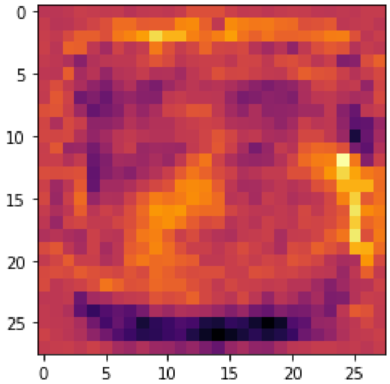
```python
# Plot filter weights
weights = mlpClassifer.weight1.data
first_ten_weights = weights[:10,:]
f = plt.figure()

for weight in first_ten_weights:
    weight = weight.view(28,28)
    plt.imshow(weight, cmap='inferno')
    plt.show()
```

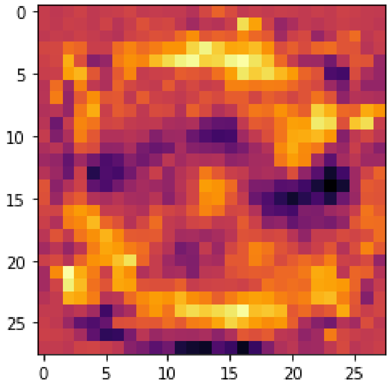


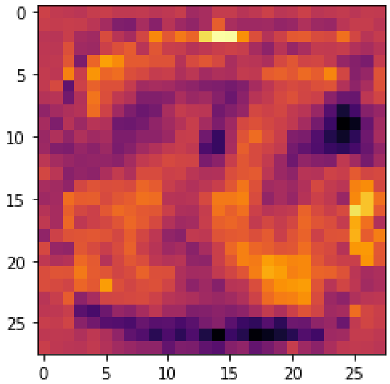


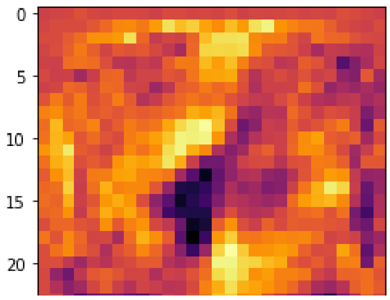


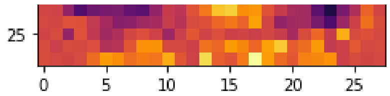

5
10
15
20
25
0 5 10 15 20 25

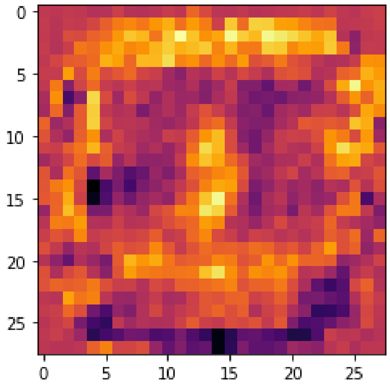

0
5
10
15
20
25
0 5 10 15 20 25

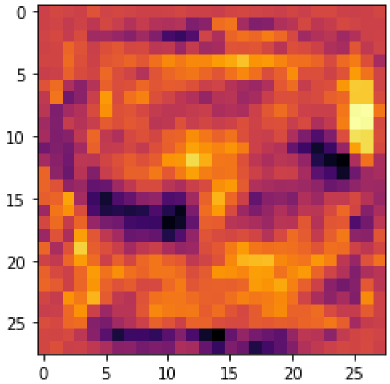

0
5
10
15
20
25
0 5 10 15 20 25


0
5
10
15
20
25
0 5 10 15 20 25


0
5
10
15
20







Comments on weights:

They still slightly resemble numbers, but look more blurry than the single layer perceptron weights and some look like combinations of numbers. Non-linearity is introduced in this perceptron as opposed to the solely linear previous perceptron. Also there is an extra layer which means an extra modification of weights on backpropagation. Both of these play a role in modifying the weights.

Part 3: Convolutional Neural Network (CNN) [5 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 [99]:

```
def conv2d(x, W, stride):
    # x: input
    # W: weights (out, in, kH, kW)

    return F.conv2d(x, W, stride=stride, padding=2)

# Defining a Convolutional Neural Network
```

```python
class CNNClassifer(DNN):
    def __init__(self, classes=10, n=5):
        super(CNNClassifer, self).__init__()
        """ ==========
        YOUR CODE HERE
        ========== """

        # dimensions for convolution were calculated by formula in lecture 18 notes
        # dimensions for everything else were found by guess and check and just filling in
        # the required dimension to match previous layer output

        ## first layer
        self.weight1 = weight_variable((n, 1, n, n))
        self.bias1 = bias_variable((50,5,14,14))

        ## second layer
        self.weight2 = weight_variable((n*n, 5, n, n))
        self.bias2 = bias_variable((50,25,7,7))

        ## third layer
        self.weight3 = weight_variable((50,1225))
        self.bias3 = bias_variable((50))

        ## fourth layer
        self.weight4 = weight_variable((10,50))
        self.bias4 = bias_variable((10))

        dtype = torch.float
        device = torch.device("cpu")


    def forward(self, x):
        """ ==========
        YOUR CODE HERE
        ========== """

        y_pred = conv2d(x,self.weight1,2)

        y_pred = torch_functional.relu(y_pred) + self.bias1

        y_pred = conv2d(y_pred,self.weight2,2)

        y_pred = torch_functional.relu(y_pred) + self.bias2

        y_pred = y_pred.view(list(y_pred.size())[0], -1)

        y_pred = torch.addmm(self.bias3, y_pred, self.weight3.t())

        y_pred = torch_functional.relu(y_pred)


        y_pred = torch.addmm(self.bias4, y_pred.view(list(y_pred.size())[0], -1), self.weight4.t())

        return y_pred

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

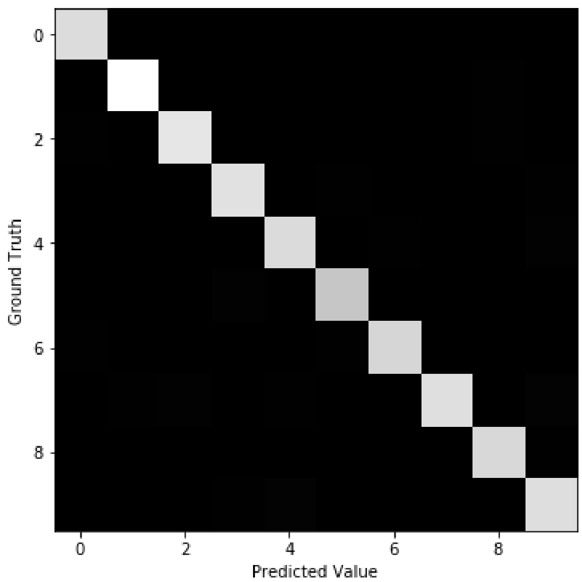
```
Epoch:1 Accuracy: 90.680000
Epoch:2 Accuracy: 93.290000
Epoch:3 Accuracy: 94.730000
Epoch:4 Accuracy: 95.620000
Epoch:5 Accuracy: 96.450000
Epoch:6 Accuracy: 96.890000
Epoch:7 Accuracy: 97.130000
Epoch:8 Accuracy: 97.350000
Epoch:9 Accuracy: 97.400000
Epoch:10 Accuracy: 97.800000
```

In [100]:

```python
# Plot Confusion matrix
M,_ = Confusion(testData, testLabels, cnnClassifer)
print('Accuracy: ', )
```

```
print('\nVisualized Confusion Matrix')
VisualizeConfusion(M)
```

```
Accuracy:  97.8

Visualized Confusion Matrix
```




```
Matrix w/ Rounded Values:

[[9.690e+00 0.000e+00 2.000e-02 0.000e+00 1.000e-02 1.000e-02 3.000e-02
  0.000e+00 3.000e-02 1.000e-02]
 [0.000e+00 1.123e+01 3.000e-02 0.000e+00 0.000e+00 0.000e+00 2.000e-02
  0.000e+00 7.000e-02 0.000e+00]
 [6.000e-02 2.000e-02 1.013e+01 1.000e-02 1.000e-02 0.000e+00 0.000e+00
  3.000e-02 6.000e-02 0.000e+00]
 [0.000e+00 0.000e+00 2.000e-02 9.910e+00 0.000e+00 6.000e-02 0.000e+00
  1.000e-02 4.000e-02 6.000e-02]
 [1.000e-02 0.000e+00 0.000e+00 0.000e+00 9.640e+00 0.000e+00 5.000e-02
  0.000e+00 1.000e-02 1.100e-01]
 [2.000e-02 0.000e+00 0.000e+00 1.000e-01 0.000e+00 8.700e+00 4.000e-02
  0.000e+00 4.000e-02 2.000e-02]
 [5.000e-02 3.000e-02 0.000e+00 0.000e+00 2.000e-02 5.000e-02 9.400e+00
  0.000e+00 3.000e-02 0.000e+00]
 [0.000e+00 5.000e-02 1.100e-01 4.000e-02 6.000e-02 0.000e+00 0.000e+00
  9.820e+00 4.000e-02 1.600e-01]
 [4.000e-02 0.000e+00 2.000e-02 4.000e-02 3.000e-02 2.000e-02 2.000e-02
  1.000e-02 9.510e+00 5.000e-02]
 [4.000e-02 4.000e-02 0.000e+00 5.000e-02 1.400e-01 3.000e-02 0.000e+00
  2.000e-02 0.000e+00 9.770e+00]]

Trace of Confusion Matrix: 97.80000000000001
```

- 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/