HW4

March 10, 2021

1 CSE 152A Computer Vision I Winter 2021 - Homework 4

1.1 Instructor: Ben Ochoa

1.1.1 Assignment Published On: Monday, March 1, 2021

1.1.2 Due On: Wednesday, March 10, 2021 11:59 pm

1.2 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.**

1.3 Problem 1: Machine Learning [12 pts]

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

1.3.1 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.

Run the torch import statements below to verify your installation.

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.

[1]:
```
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)
```

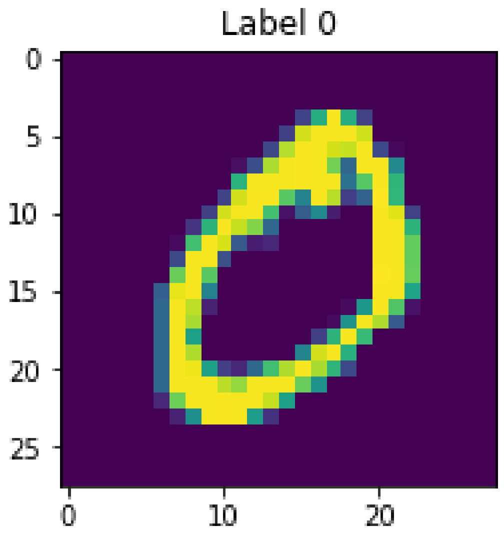
```
tensor([[0.0372, 0.2476, 0.6242],
        [0.2032, 0.7708, 0.7114],
        [0.7448, 0.1704, 0.9760],
        [0.2418, 0.2059, 0.1411],
        [0.5549, 0.6494, 0.4420]])
```

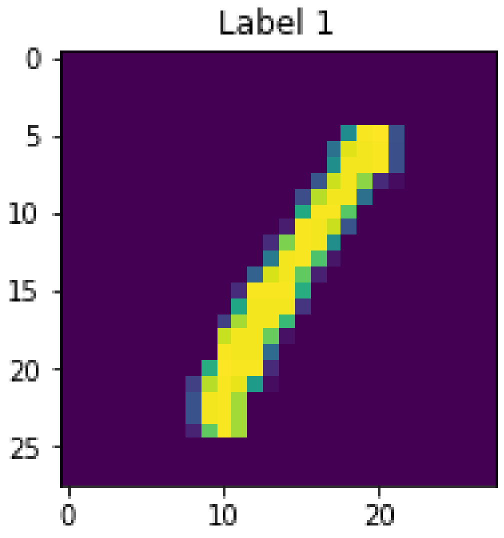
[2]:
```
import os
import struct
import numpy as np

# Change path as required
path = "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)

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')))
```

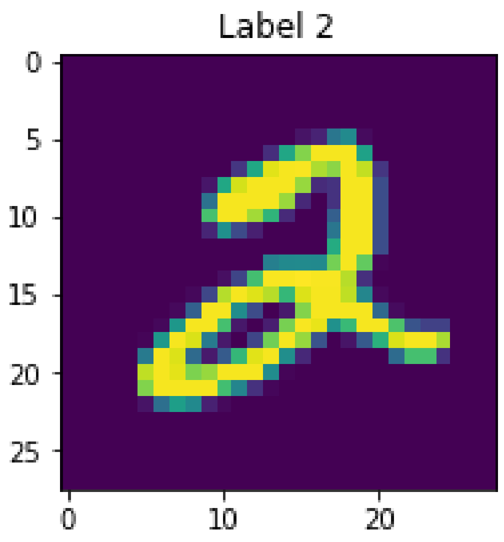
[3]:
```
# Understand the shapes of the each variable carying data
print(trainData.shape, trainLabels.shape)
print(testData.shape, testLabels.shape)
```

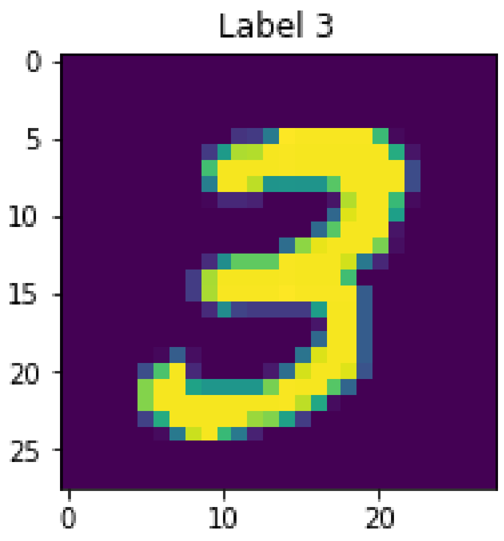
```
(60000, 28, 28) (60000,)
(10000, 28, 28) (10000,)
```

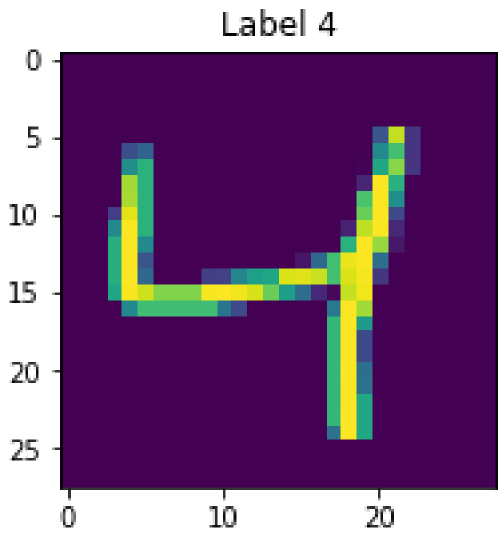
[5]:
```
# display one image from each label
""" ==========
YOUR CODE HERE
========== """
import matplotlib.pyplot as plt

unique_labels = list(set(trainLabels))
for i in unique_labels:
    index = np.where(trainLabels == i)[0][0]
    img = trainData[index]
    plt.figure(figsize=(3,3))
    plt.imshow(img)
    plt.title('Label '+ str(i))
    plt.show()
```

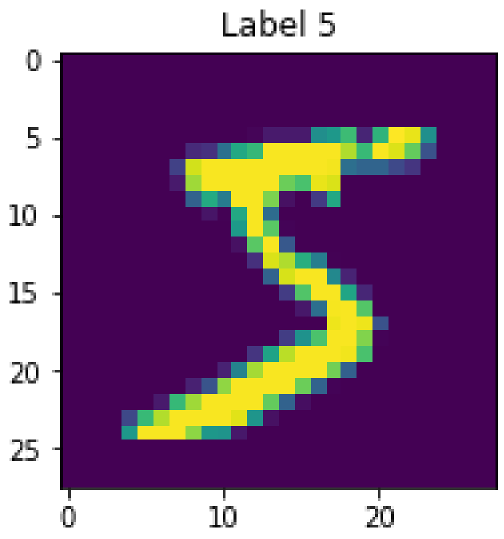




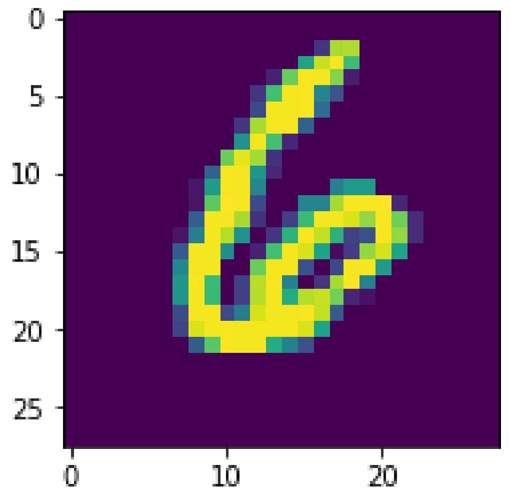

Label 2
0
5
10
15
20
25
0
10
20


Label 3
0
5
10
15
20
25
0
10
20

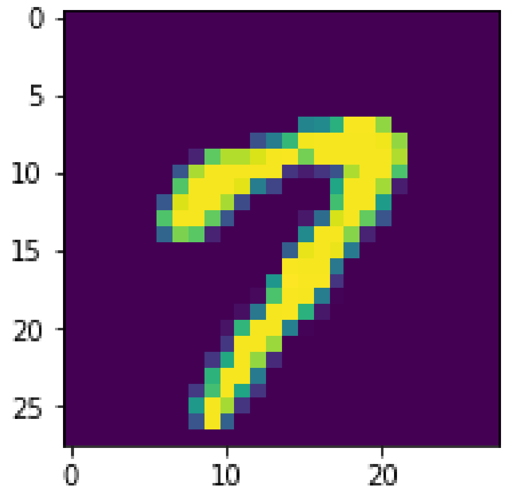

Label 4
0
5
10
15
20
25
0
10
20

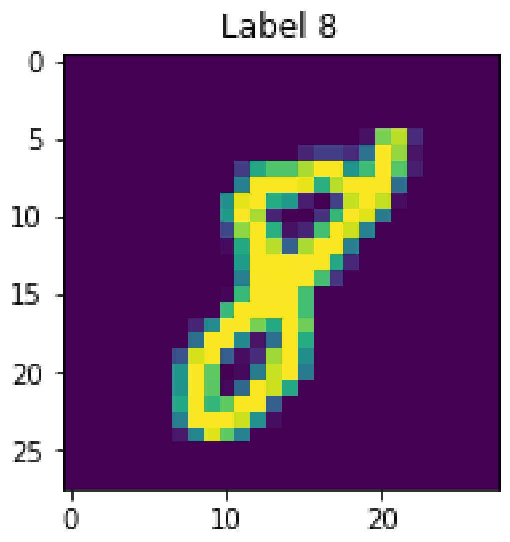

Label 5
0
5
10
15
20
25
0
10
20

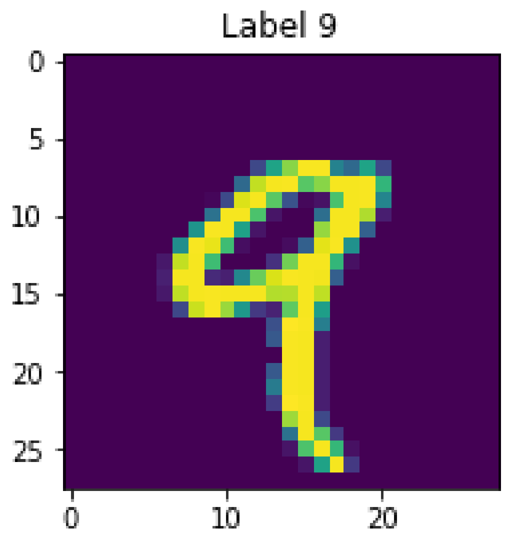
Label 6



Label 7







Some helper functions are given below.

[6]:
```
# a generator for batches of data
# yields data (batchsize, 28, 28) and labels (batchsize)
# if shuffle is True, 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])
```

[7]:

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

```
Random classifier accuracy: 9.780000
```

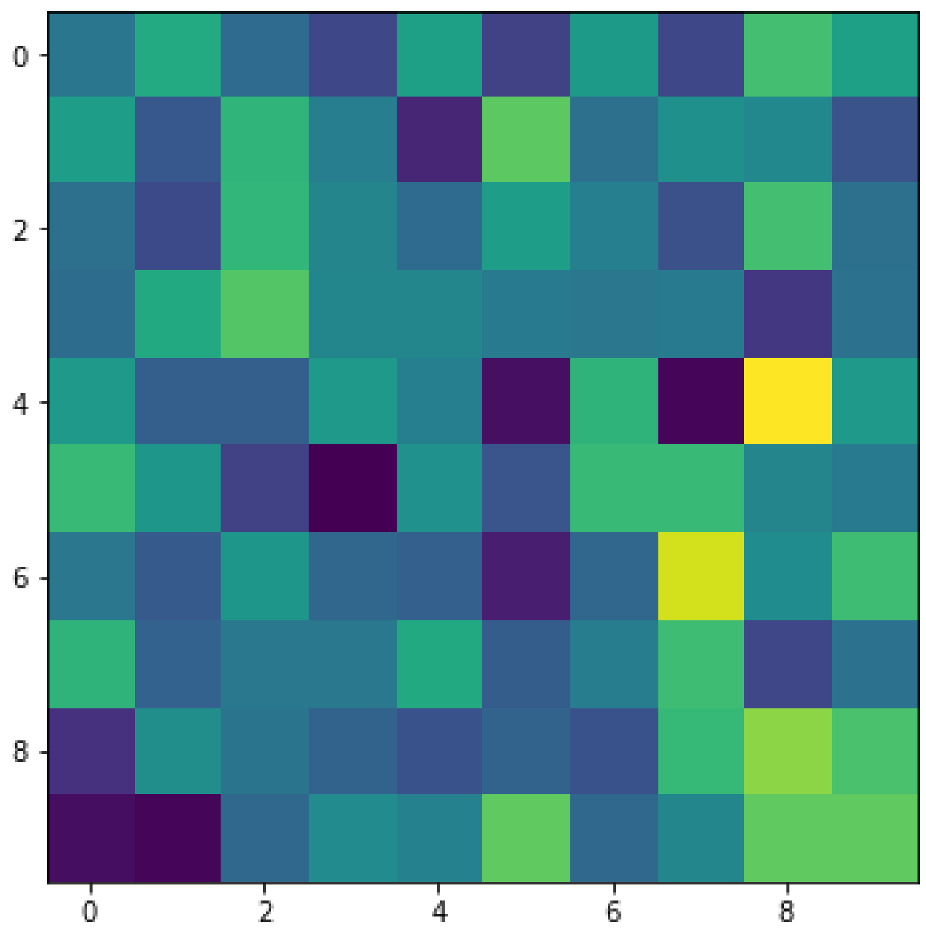
1.3.2 Part 2: Confusion Matrix [2 pts]

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?

[8]:

```
# Using the tqdm module to visualize run time is suggested
# from tqdm import tqdm
# from tqdm import tqdm_notebook

# 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
    ========== """
    batchsize=50
    correct=0.
    for data,label in DataBatch(testData,testLabels,batchsize,shuffle=False):
        prediction = classifier(data)
        correct += np.sum(prediction==label)
        for i in range(len(label)):
            M[label[i],prediction[i]] += 1
    for i in range(10):
        M[i,:] /= np.sum(M[i,:])
    acc = correct/testData.shape[0]*100

    return M, acc

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

[10]:
```
M, acc = Confusion(testData, testLabels, randomClassifier)
VisualizeConfusion(M)
print("Accuracy =", acc)
```



```
[[0.1  0.11 0.1  0.09 0.11 0.09 0.11 0.09 0.11 0.11]
 [0.11 0.09 0.11 0.1  0.08 0.11 0.1  0.1  0.1  0.09]
 [0.1  0.09 0.11 0.1  0.1  0.11 0.1  0.09 0.11 0.1 ]
 [0.1  0.11 0.11 0.1  0.1  0.1  0.1  0.1  0.09 0.1 ]
 [0.1  0.09 0.09 0.1  0.1  0.08 0.11 0.08 0.13 0.1 ]
 [0.11 0.1  0.09 0.08 0.1  0.09 0.11 0.11 0.1  0.1 ]
 [0.1  0.09 0.1  0.09 0.09 0.08 0.09 0.12 0.1  0.11]
 [0.11 0.09 0.1  0.1  0.11 0.09 0.1  0.11 0.09 0.1 ]
 [0.09 0.1  0.1  0.09 0.09 0.09 0.09 0.11 0.12 0.11]
 [0.08 0.08 0.1  0.1  0.1  0.11 0.1  0.1  0.11 0.11]]
Accuracy = 10.34
```

Justify the accuracy & values of the confusion matrix of random classifier: Because the random classifier randomly predicts for any input, the probability that the predicted class is the same as the true class for an input is 1 / #classes or in this case 1 / 10. This explains why every entry of the confusion matrix is roughly 0.1 (the probability that images of class i predicted as class i is 0.1) and the accuracy of the random classifier is 0.1.

1.3.3 Part 3: K-Nearest Neighbors (KNN) [4 pts]

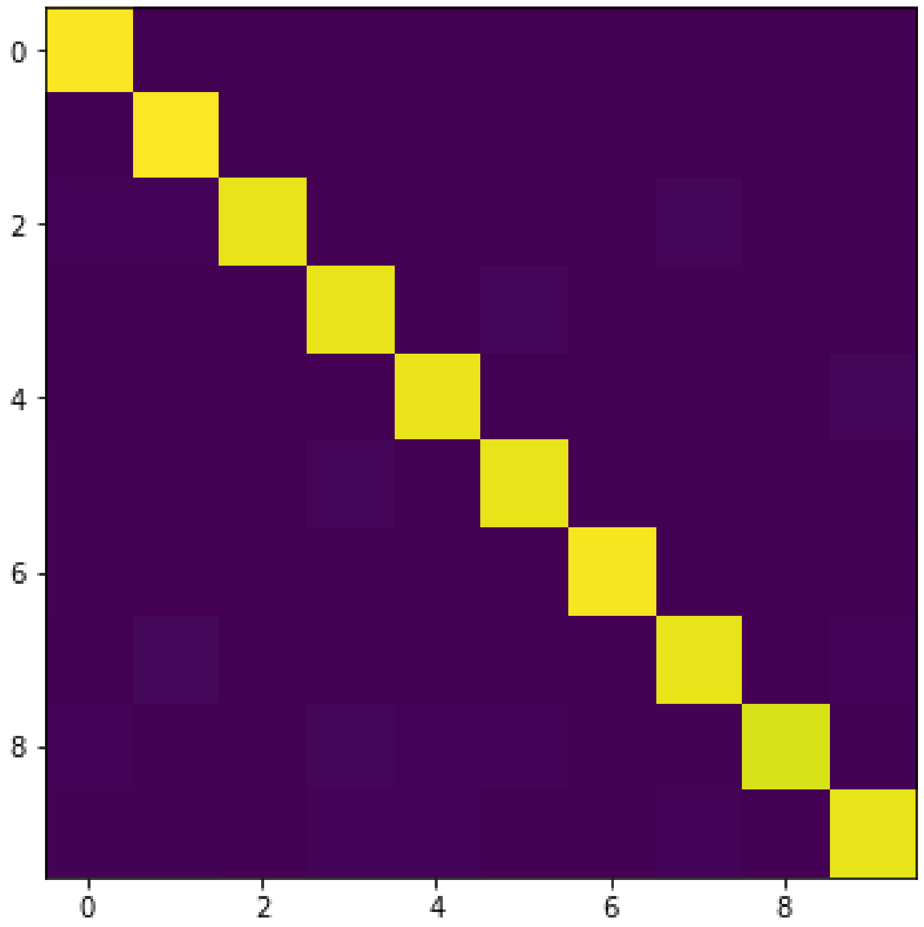
- Here you will implement a simple knn classifier. The distance metric is Euclidean in pixel space. k refers to the number of neighbors involved in voting on the class, and should be 3. You are allowed to use sklearn.neighbors.KNeighborsClassifier.
- Display confusion matrix and accuracy for your KNN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

[11]:
```
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.neigh = KNeighborsClassifier(n_neighbors=k)

    def train(self, trainData, trainLabels):
        """ ==========
        YOUR CODE HERE
        ========== """
        n,h,w = trainData.shape
        trainData = trainData.reshape(n, h*w)
        self.neigh.fit(trainData, trainLabels)

    def __call__(self, x):
        # this method should take a batch of images
        # and return a batch of predictions
        """ ==========
        YOUR CODE HERE
        ========== """
        n,h,w = x.shape
        x = x.reshape(n, h*w)
        return self.neigh.predict(x)


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

```
KNN classifier accuracy: 64.760000
```

[12]:
```
# test your classifier with all the training examples (This may take a while)
knnClassifer = KNNClassifer()
knnClassifer.train(trainData, trainLabels)
```

[13]:
```
# display confusion matrix for your KNN classifier with all the training␣
 ↪examples
# (This may take a while)
""" ==========
YOUR CODE HERE
========== """
M, acc = Confusion(testData, testLabels, knnClassifer)
VisualizeConfusion(M)
print("Accuracy =", acc)
```



```
[[0.99 0.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   1.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.01 0.01 0.97 0.   0.   0.   0.   0.01 0.   0.  ]
 [0.   0.   0.   0.97 0.   0.01 0.   0.01 0.   0.  ]
 [0.   0.01 0.   0.   0.97 0.   0.   0.   0.   0.02]
 [0.01 0.   0.   0.01 0.   0.96 0.01 0.   0.   0.  ]
 [0.01 0.   0.   0.   0.   0.   0.99 0.   0.   0.  ]
 [0.   0.02 0.   0.   0.   0.   0.   0.96 0.   0.01]
 [0.01 0.   0.   0.02 0.01 0.01 0.   0.   0.94 0.  ]
 [0.   0.   0.   0.01 0.01 0.   0.   0.01 0.   0.96]]
Accuracy = 97.05
```

Mention the number that the number '7' is most often predicted to be, other than '7' and justify your answer. The number that the number '7' is most often predicted to be, other than '7', is '1', because in the row for ground truth class 7, entry for class 1 has the highest value other than class 7. This happens bacause 7 looks alike 1 and some badly written 7's are even recognized as 1 by human. So badly written 7's are very likely to be predicted as 1 by the classifier.

1.3.4 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. Note that this method was discussed during the whiteboarding session of lecture 14.

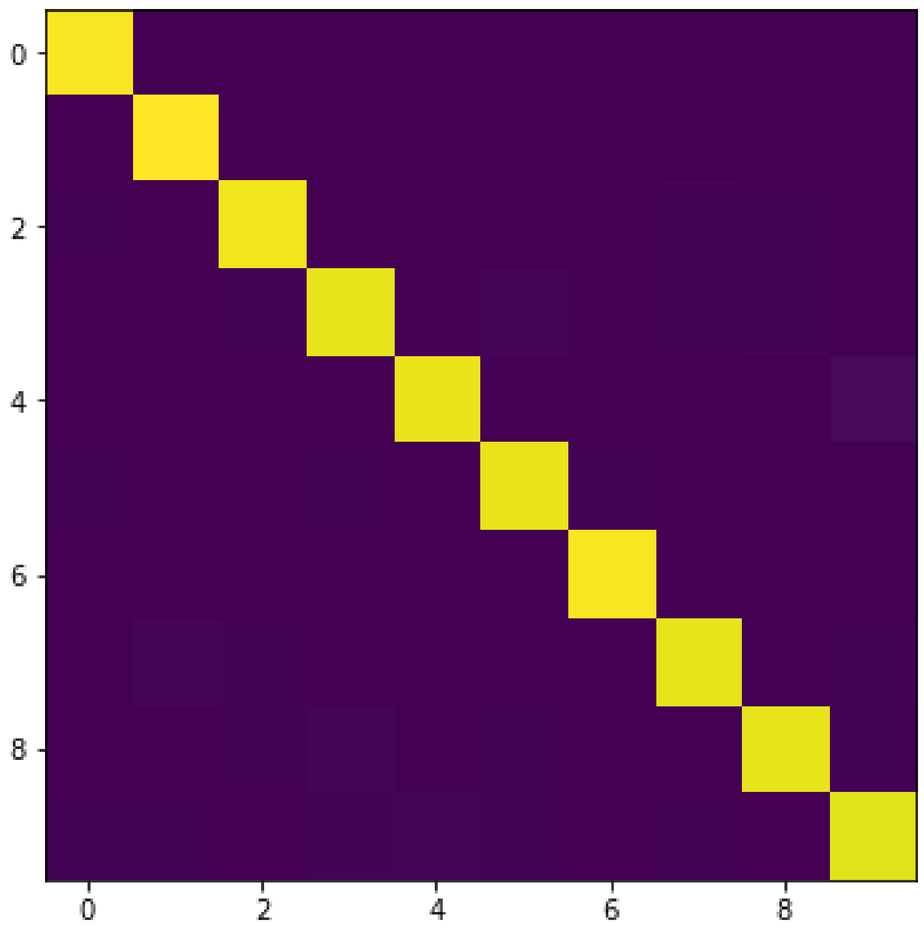
Is the testing time for PCA KNN classifier more or less than that for KNN classifier? Comment on why it differs if it does.

[14]:
```python
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.neigh = KNeighborsClassifier(n_neighbors=k)

    def train(self, trainData, trainLabels):
        """ ==========
        YOUR CODE HERE
        ========== """
        n,h,w = trainData.shape
        trainData = trainData.reshape(n, h*w)
        self.mean = np.sum(trainData, axis=0) / n
        centered_data = trainData - self.mean
        U,S,VT = np.linalg.svd(centered_data, full_matrices=False)
        V = VT.T
        self.W = V[:,:self.components]
        X = centered_data.dot(self.W)
        self.neigh.fit(X, trainLabels)

    def __call__(self, x):
        # this method should take a batch of images
        # and return a batch of predictions
        """ ==========
        YOUR CODE HERE
        ========== """
        n,h,w = x.shape
        x = x.reshape(n, h*w)
        centered_x = x - self.mean
        X = centered_x.dot(self.W)
        return self.neigh.predict(X)


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

```
KNN classifier accuracy: 66.160000
```

[15]:
```
# test your classifier with all the training examples
pcaknnClassifer = PCAKNNClassifer()
pcaknnClassifer.train(trainData, trainLabels)
```

[16]:
```
# display confusion matrix for your PCA KNN classifier with all the training␣
 ↪examples
""" ==========
YOUR CODE HERE
========== """
M, acc = Confusion(testData, testLabels, pcaknnClassifer)
VisualizeConfusion(M)
print("Accuracy =", acc)
```



```
[[0.99 0.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   1.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.98 0.   0.   0.   0.   0.01 0.   0.  ]
 [0.   0.   0.   0.96 0.   0.01 0.   0.01 0.01 0.  ]
 [0.   0.   0.   0.   0.97 0.   0.   0.   0.   0.03]
 [0.01 0.   0.   0.01 0.   0.97 0.01 0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.99 0.   0.   0.  ]
 [0.   0.02 0.01 0.   0.   0.   0.   0.96 0.   0.01]
 [0.   0.   0.   0.02 0.   0.01 0.   0.   0.96 0.  ]
 [0.   0.01 0.   0.01 0.01 0.   0.   0.   0.   0.95]]
Accuracy = 97.31
```

Comments: The testing time for PCA KNN classifier is much less than that for KNN classifier. This is because PCA has reduced the dimensionality of training and test data. Input data for PCA KNN classifier are 25 dimensional, while input data for KNN classifier are 28 * 28 dimensional. Data in lower dimension will save more runtime.

1.4 Problem 2: Deep learning [14 pts]

Below is some helper code to train your deep networks.

1.4.1 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.

[17]:
```
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 F
```

[18]:
```
# 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.
class DNN(nn.Module):
    def __init__(self):
        super(DNN, self).__init__()
        pass

    def forward(self, x):
        raise NotImplementedError

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


        for epoch in range(epochs):
            self.train()  # set network 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))

                # Now train the model using the optimizer and the batch data
                """ ==========
                YOUR CODE HERE
                ========== """
                prediction = self.forward(data)
                loss = criterion(prediction, labels)
                self.zero_grad()
                loss.backward()
                optimizer.step()
                # xxxxxxxxxx End of your code, don't change anything else here␣
↪xxxxxxxxxx

            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)
```

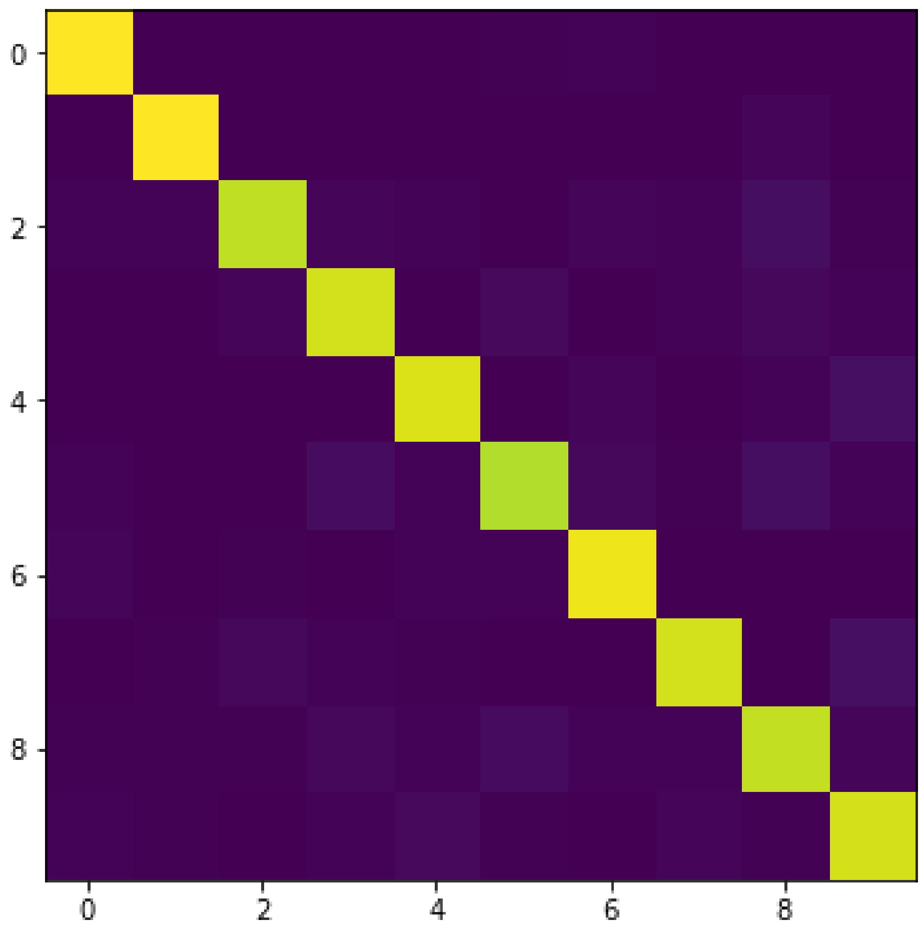
[19]:
```
# example linear classifier - input connected to output
# you can take this as an example to learn how to extend DNN class
class LinearClassifier(DNN):
    def __init__(self, in_features=28*28, classes=10):
        super(LinearClassifier, self).__init__()
        # in_features=28*28
        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
```

[20]:
```
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'))))
```

[24]:
```
# 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: 89.330000
Epoch:2 Accuracy: 90.680000
Epoch:3 Accuracy: 91.260000
Epoch:4 Accuracy: 91.560000
Epoch:5 Accuracy: 91.720000
Epoch:6 Accuracy: 92.020000
Epoch:7 Accuracy: 92.200000
Epoch:8 Accuracy: 92.230000
Epoch:9 Accuracy: 92.410000
Epoch:10 Accuracy: 92.400000
```

[25]:
```
# display confusion matrix
""" ==========
YOUR CODE HERE
========== """
M, acc = Confusion(testData, testLabels, linearClassifier)
VisualizeConfusion(M)
print("Accuracy =", acc)
```

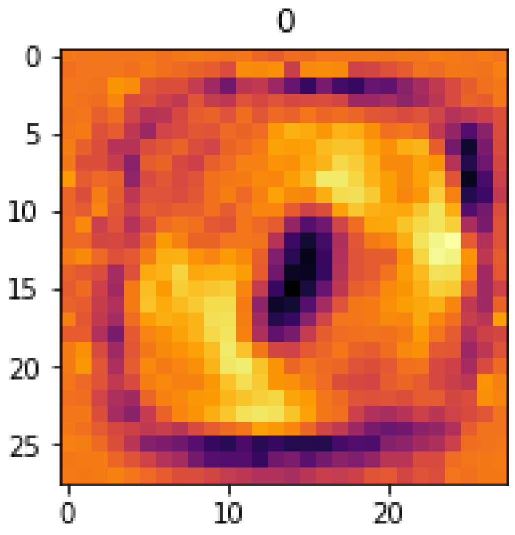


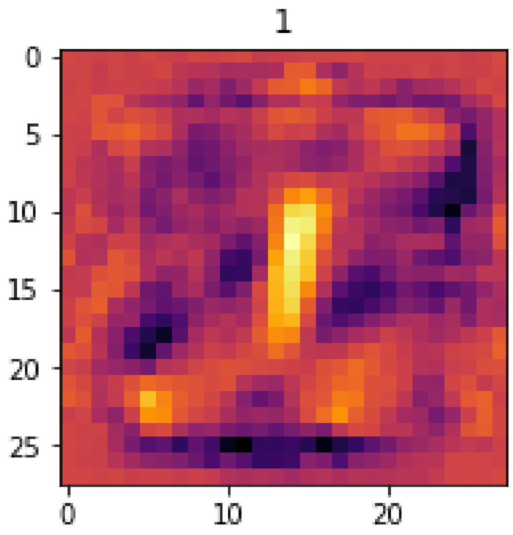
```
[[0.98 0.   0.   0.   0.   0.01 0.01 0.   0.   0.  ]
 [0.   0.98 0.   0.   0.   0.   0.   0.   0.01 0.  ]
 [0.01 0.01 0.89 0.02 0.01 0.   0.01 0.01 0.04 0.  ]
 [0.   0.   0.02 0.91 0.   0.03 0.   0.01 0.02 0.01]
 [0.   0.   0.   0.   0.93 0.   0.01 0.   0.01 0.04]
 [0.01 0.   0.   0.03 0.01 0.87 0.02 0.   0.04 0.01]
 [0.01 0.   0.01 0.   0.01 0.01 0.96 0.   0.   0.  ]
 [0.   0.01 0.02 0.01 0.01 0.   0.   0.91 0.   0.04]
 [0.01 0.01 0.01 0.02 0.01 0.03 0.01 0.01 0.89 0.01]
 [0.01 0.01 0.   0.01 0.03 0.01 0.   0.02 0.   0.92]]
Accuracy = 92.4
```

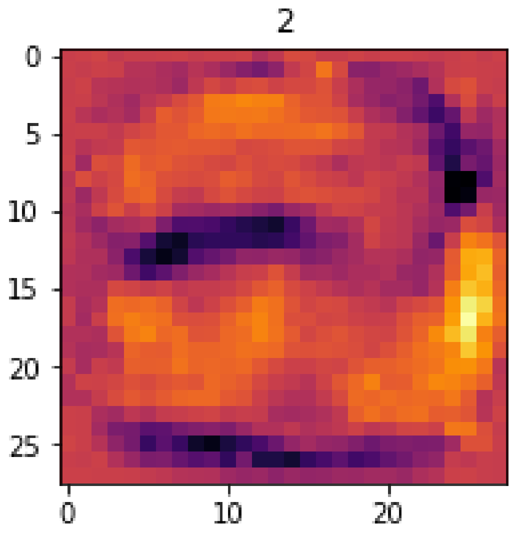
1.4.2 Part 2: Single Layer Perceptron [2 pts]

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.

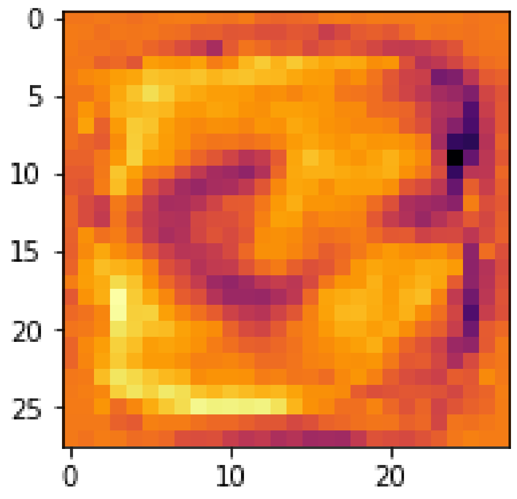
[26]:
```
# Plot filter weights corresponding to each class, you may have to reshape them␣
↪to make sense out of them
# linearClassifier.weight1.data will give you the first layer weights
""" ==========
YOUR CODE HERE
========== """
def plot_filter_weights(weights):
    weights = weights.detach().numpy()
    weights = weights.reshape((10,28,28))
    for i, w in enumerate(weights):
        w = w / np.amax(w)
        plt.figure(figsize=(3,3))
        plt.imshow(w, aspect='auto', cmap=plt.get_cmap('inferno'))
        plt.title(i)
        plt.show()

plot_filter_weights(linearClassifier.weight1)
```



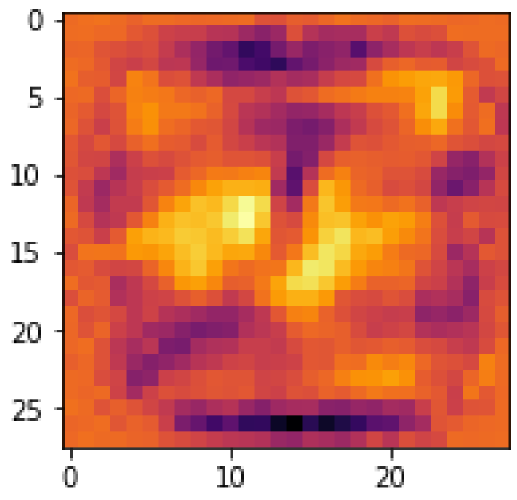

1
0
5
10
15
20
25
0
10
20

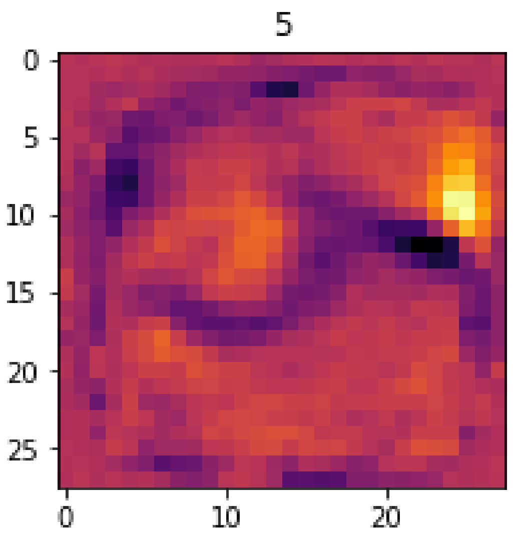

2
0
5
10
15
20
25
0
10
20

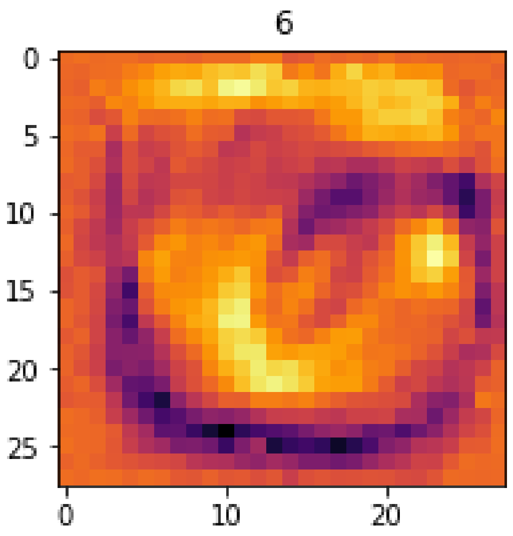
3

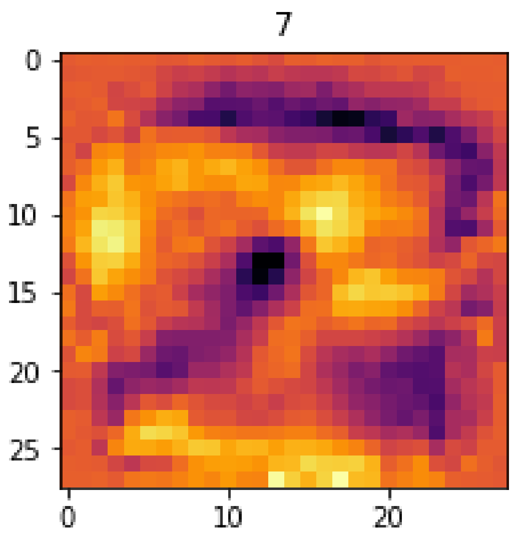


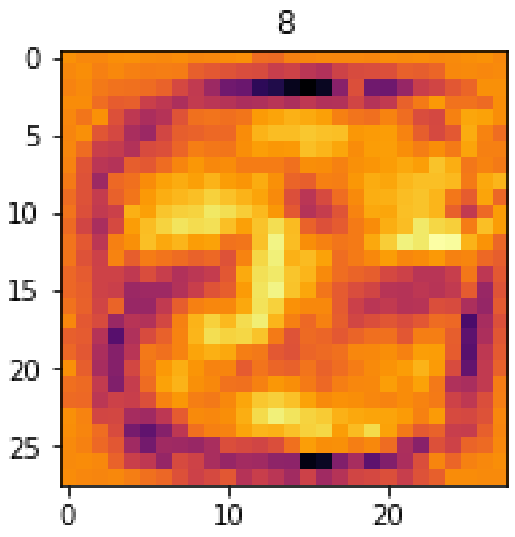
4

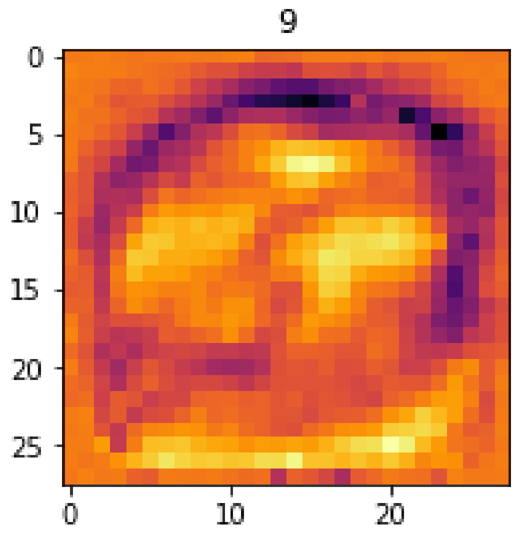



5
0
5
10
15
20
25
0
10
20


6
0
5
10
15
20
25
0
10
20


7
0
5
10
15
20
25
0
10
20


8
0
5
10
15
20
25
0
10
20



Comment on what the weights look like and why that may be so In the colormaps, brighter pixels are associated with higher weights and darker pixels are associated with lower weights. We can see the brighter areas look the same in shape as the corresponding numbers, which means the pixels where the corresponding number may appear are given higher weights. This is good for classification because data in each class will get larger results when multiplied with corresponding weights associated with the class and the larger results can easily classify the data.

1.4.3 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.

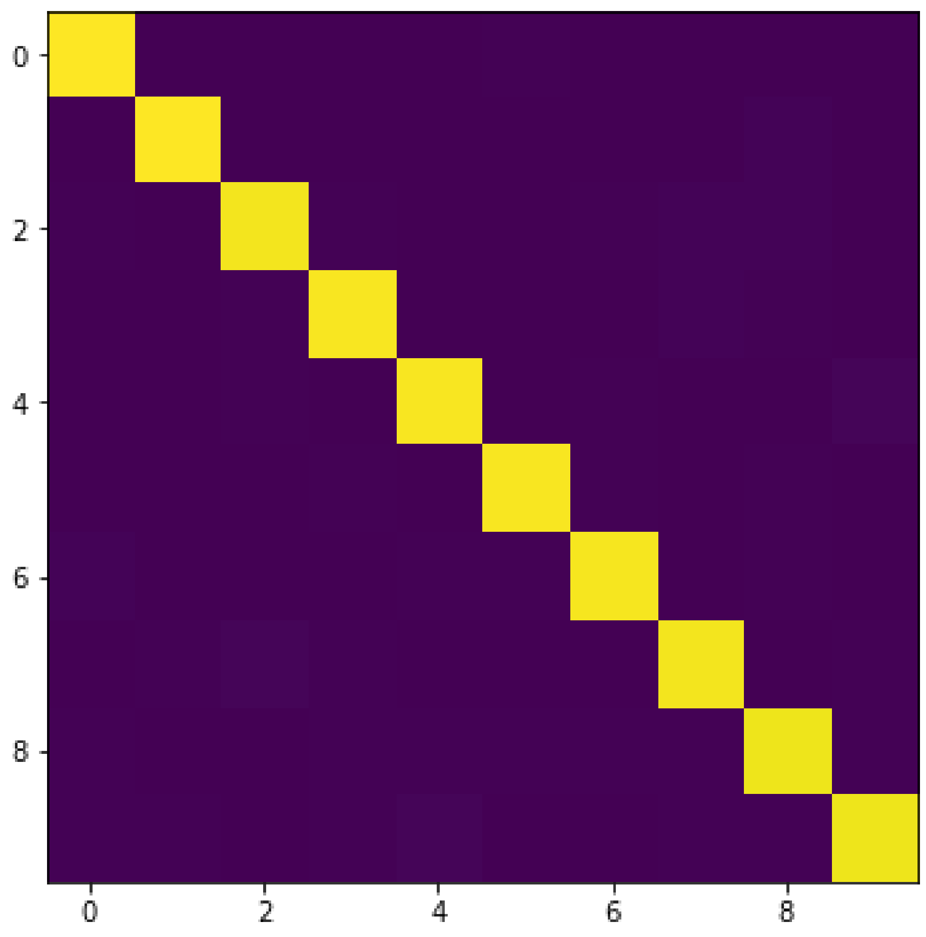
Display the confusion matrix and accuracy after training. Note: You should get ~ 97 % accuracy for 10 epochs and batch size 50.

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?

[27]:
```
class MLPClassifier(DNN):
    def __init__(self, in_features=28*28, classes=10, hidden=100):
        super(MLPClassifier, self).__init__()
        """ ==========
        YOUR CODE HERE
        ========== """
        self.weight1 = weight_variable((hidden, in_features))
        self.bias1 = bias_variable((hidden))
        self.weight2 = weight_variable((classes, hidden))
        self.bias2 = bias_variable((classes))

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

mlpClassifier = MLPClassifier()
mlpClassifier.train_net(trainData, trainLabels, epochs=10, batchSize=50)
```

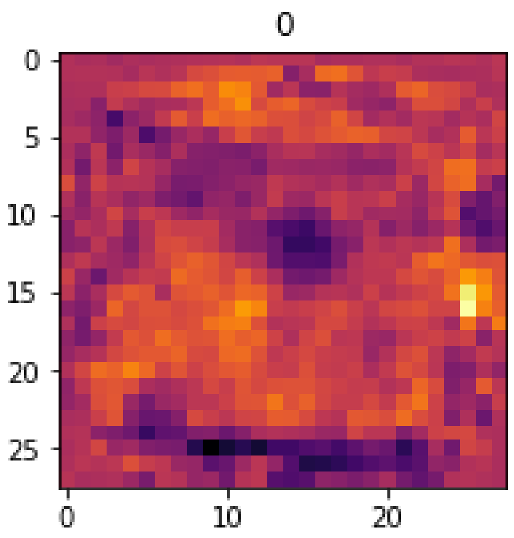
```
Epoch:1 Accuracy: 91.370000
Epoch:2 Accuracy: 92.810000
Epoch:3 Accuracy: 93.880000
Epoch:4 Accuracy: 94.710000
Epoch:5 Accuracy: 95.480000
Epoch:6 Accuracy: 95.910000
Epoch:7 Accuracy: 96.330000
Epoch:8 Accuracy: 96.720000
Epoch:9 Accuracy: 96.930000
Epoch:10 Accuracy: 97.070000
```

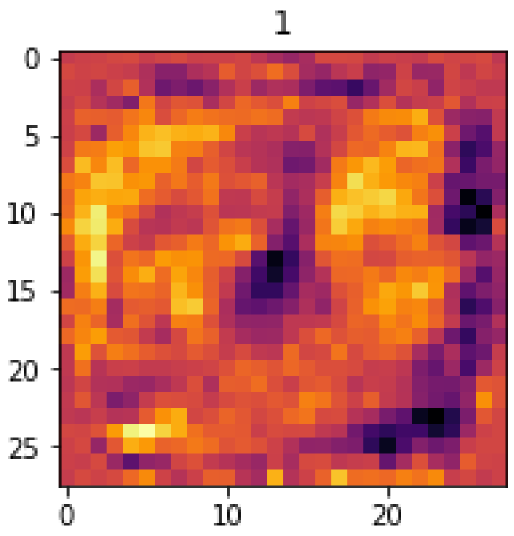
[28]:
```
# Plot confusion matrix
""" ==========
YOUR CODE HERE
========== """
M, acc = Confusion(testData, testLabels, mlpClassifier)
VisualizeConfusion(M)
print("Accuracy =", acc)
```

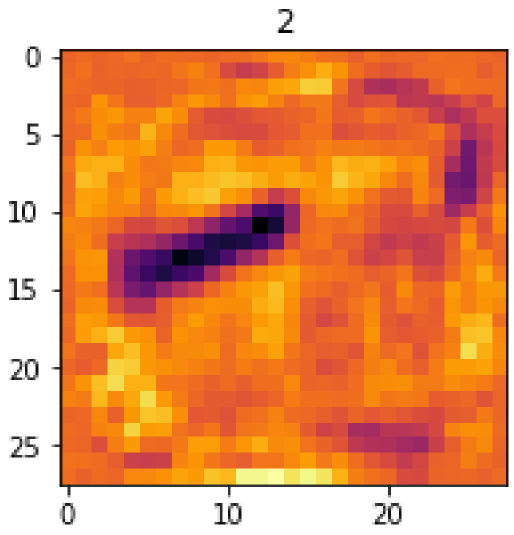


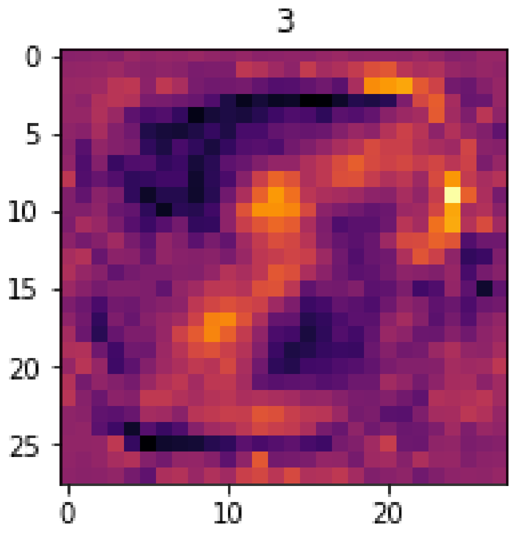
```
[[0.98 0.   0.   0.   0.   0.01 0.   0.   0.   0.  ]
 [0.   0.98 0.   0.   0.   0.   0.   0.   0.01 0.  ]
 [0.   0.   0.97 0.   0.   0.   0.   0.01 0.01 0.  ]
 [0.   0.   0.   0.97 0.   0.   0.   0.01 0.01 0.  ]
 [0.   0.   0.01 0.   0.97 0.   0.01 0.   0.   0.01]
 [0.   0.   0.   0.01 0.   0.97 0.   0.   0.01 0.  ]
 [0.01 0.   0.   0.   0.01 0.01 0.97 0.   0.   0.  ]
 [0.   0.   0.01 0.   0.   0.   0.   0.97 0.   0.01]
 [0.01 0.   0.   0.01 0.01 0.01 0.01 0.   0.96 0.01]
 [0.   0.   0.   0.01 0.01 0.   0.   0.   0.   0.96]]
Accuracy = 97.07000000000001
```

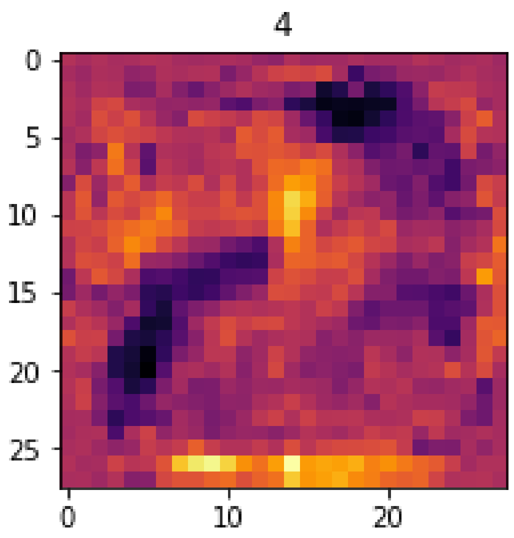
[29]:
```
# Plot filter weights
""" ==========
YOUR CODE HERE
========== """
plot_filter_weights(mlpClassifier.weight1[:10])
```

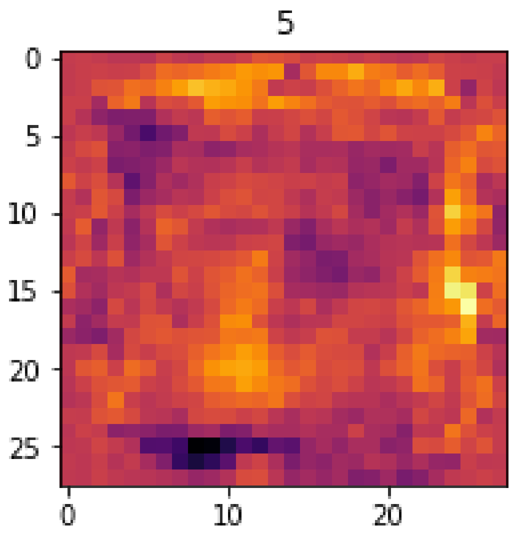

0
0
5
10
15
20
25
0
10
20

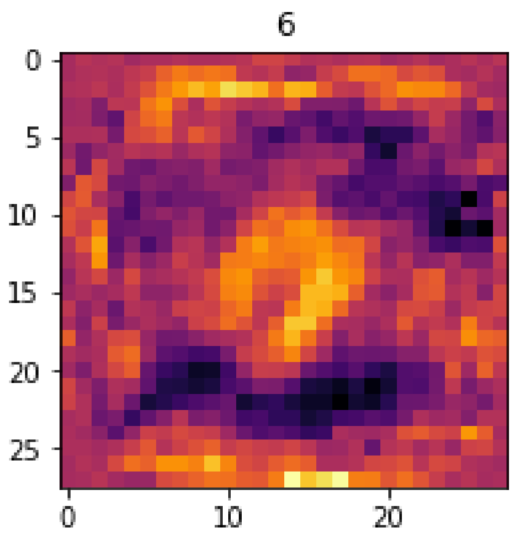

1
0
5
10
15
20
25
0
10
20

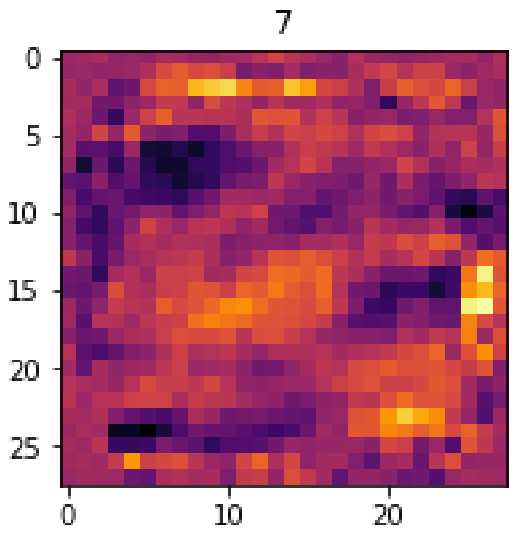

2
0
5
10
15
20
25
0
10
20

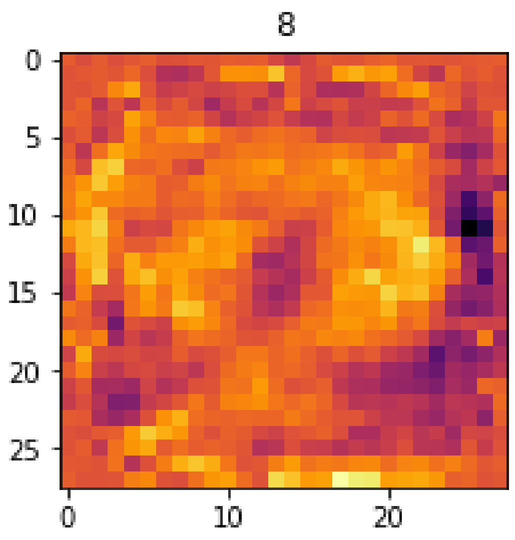

3
0
5
10
15
20
25
0
10
20

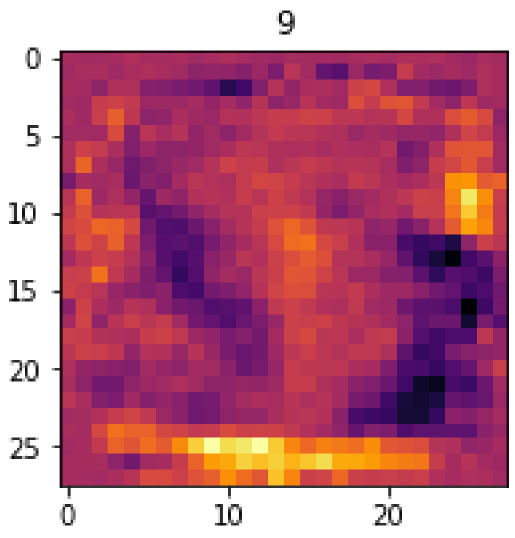

4
0
5
10
15
20
25
0
10
20


5
0
5
10
15
20
25
0
10
20


6
0
5
10
15
20
25
0
10
20


7
0
5
10
15
20
25
0
10
20





Do the weights look similar to the weights plotted in the previous problem? Why or why not? The weights does NOT look similar to the weights plotted in the previous problem, because the weights is intended for the intermediate layer and the intermediate layer does not tell much information about the classes.

1.4.4 Part 4: 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)

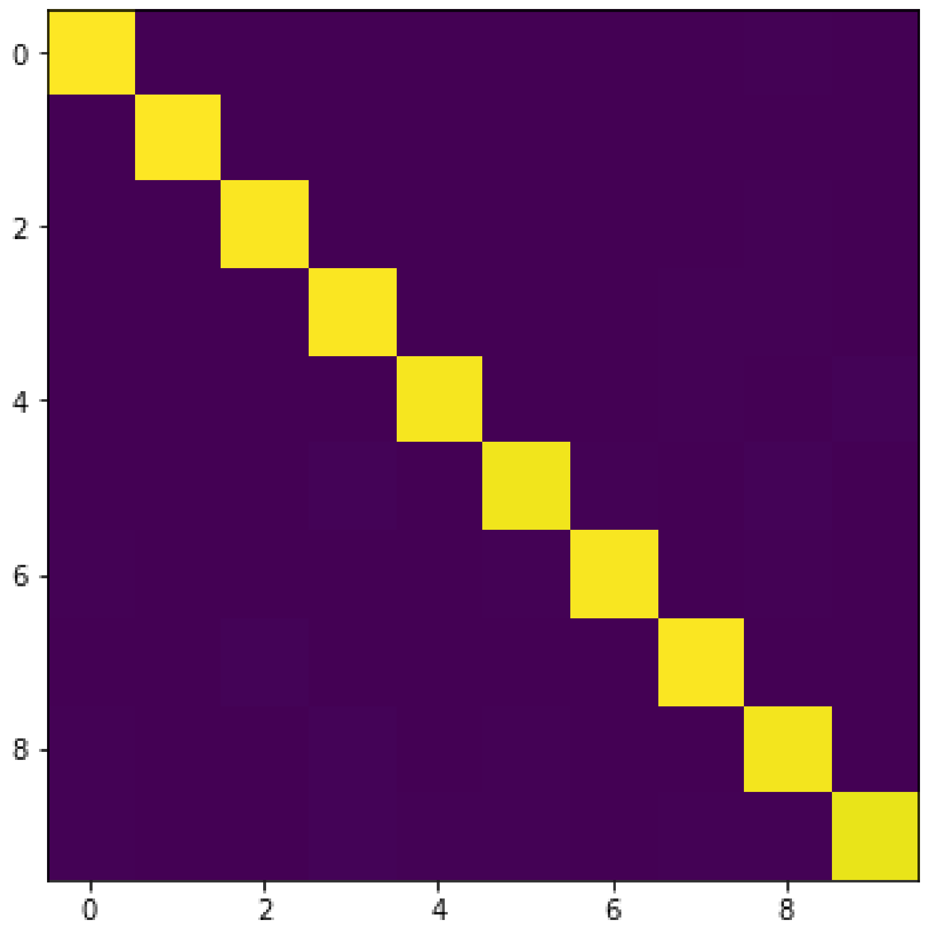
So, 2 convolutional layers, followed by 1 fully connected hidden layer and then the output layer

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.

[34]:
```
def conv2d(x, W, stride, bias):
    # x: input
    # W: weights (out, in, kH, kW)
    return F.conv2d(x, W, stride=stride, padding=2, bias=bias)

# Defining a Convolutional Neural Network
class CNNClassifier(DNN):
    def __init__(self, in_features=28*28, classes=10, n=5):
        super(CNNClassifier, self).__init__()
        """ ==========
        YOUR CODE HERE
        ========== """
        self.weight1 = weight_variable((n, 1, 5, 5))
        self.bias1 = bias_variable((n))
        self.weight2 = weight_variable((n*2, n, 5, 5))
        self.bias2 = bias_variable((n*2))
        self.weight3 = weight_variable((64, int(n*2*in_features/16)))
        self.bias3 = bias_variable((64))
        self.weight4 = weight_variable((classes, 64))
        self.bias4 = bias_variable((classes))

    def forward(self, x):
        """ ==========
        YOUR CODE HERE
        ========== """
        x = conv2d(x, self.weight1, 2, self.bias1)
        x = F.relu(x)
        x = conv2d(x, self.weight2, 2, self.bias2)
        x = F.relu(x)
        x = torch.addmm(self.bias3, x.view(list(x.size())[0], -1), self.weight3.
↪t())
        x = F.relu(x)
        y_pred = torch.addmm(self.bias4, x.view(list(x.size())[0], -1), self.
↪weight4.t())
        return y_pred

cnnClassifier = CNNClassifier()
cnnClassifier.train_net(trainData, trainLabels, epochs=10)
```

```
Epoch:1 Accuracy: 92.290000
Epoch:2 Accuracy: 94.320000
Epoch:3 Accuracy: 95.460000
Epoch:4 Accuracy: 96.430000
Epoch:5 Accuracy: 96.780000
Epoch:6 Accuracy: 97.350000
Epoch:7 Accuracy: 97.320000
Epoch:8 Accuracy: 97.700000
Epoch:9 Accuracy: 97.690000
Epoch:10 Accuracy: 98.030000
```

[35]:
```
# Plot Confusion matrix
""" ==========
YOUR CODE HERE
========== """
M, acc = Confusion(testData, testLabels, cnnClassifier)
VisualizeConfusion(M)
print("Accuracy =", acc)
```



```
[[0.99 0.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.99 0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.98 0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.99 0.   0.   0.   0.   0.01 0.  ]
 [0.   0.   0.   0.   0.98 0.   0.   0.01 0.   0.01]
 [0.   0.   0.   0.01 0.   0.97 0.01 0.   0.01 0.  ]
 [0.01 0.   0.   0.   0.   0.   0.98 0.   0.   0.  ]
 [0.   0.   0.01 0.   0.   0.   0.   0.99 0.   0.  ]
 [0.   0.   0.   0.01 0.   0.01 0.   0.   0.97 0.  ]
 [0.   0.   0.   0.01 0.   0.   0.   0.01 0.01 0.96]]
Accuracy = 98.03
```

- Note that the ConvNet approach leads 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.

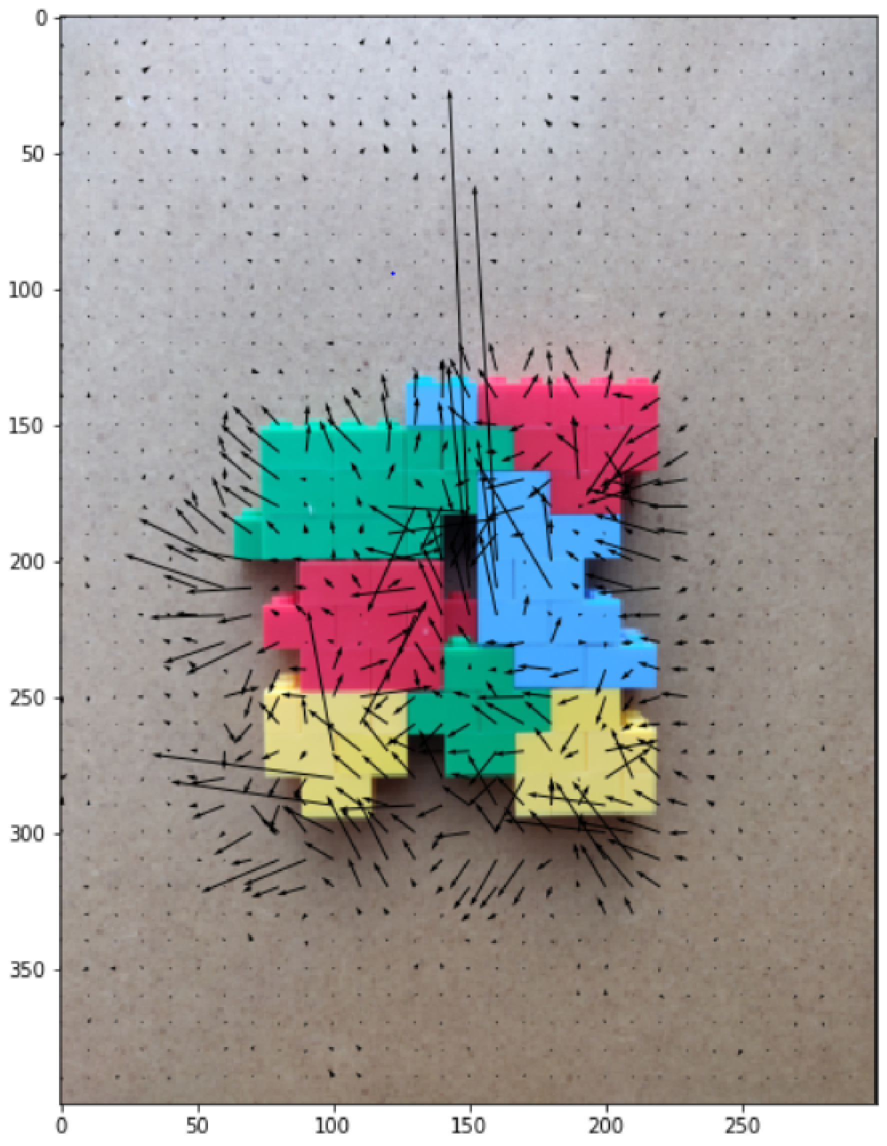
- You can look at the performance of various machine learning methods on this problem at http://yann.lecun.com/exdb/mnist/
- You can learn more about neural nets/ pytorch at https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
- You can play with a demo of neural network created by Daniel Smilkov and Shan Carter at https://playground.tensorflow.org/

1.5 Problem 3: Optical Flow [Optional] [0 pts]

NOTE: This problem is optional. Your submission for this problem would be graded but you would not receive a score for solving Problem 3. However, you are welcome and encouraged to try it out and bring any questions that you have to the instructional team.

In this problem, the 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.



1.5.1 Part 1: Lucas-Kanade implementation

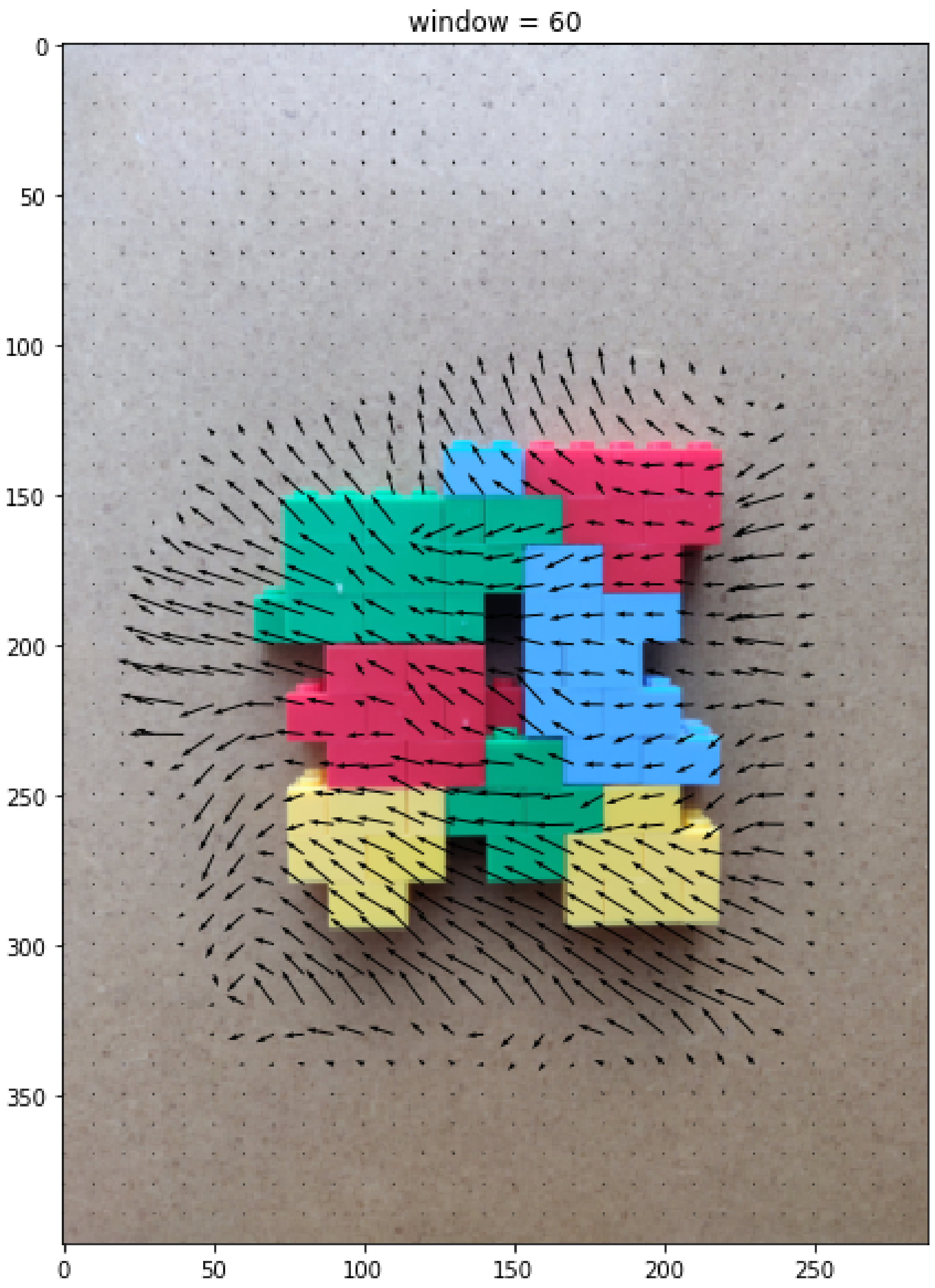
Implement the Lucas-Kanade method for estimating optical flow. The function 'LucasKanade' needs to be completed. Test your code using the sample window size given.

[36]:
```python
import numpy as np
import matplotlib.pyplot as plt
#from scipy.signal import convolve
from scipy.ndimage import convolve
from skimage.transform import resize
# 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)
    # since plt.quiver() considers positive y-axis opposite to
    # image processing convention, use -V1
    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
```

[37]:
```
def LucasKanade(img1,img2,window):
    '''
    Inputs: the two images, window size
    Returns: U, V - the optical flow
    '''
    """ ==========
    YOUR CODE HERE
    ========== """
    if window % 2 == 0:
        window += 1

    U = np.zeros_like(img1)
    V = np.zeros_like(img2)
    pad_size = int((window - 1) / 2)
    padded_img1 = np.pad(img1, (pad_size,pad_size), mode='reflect')
    padded_img2 = np.pad(img2, (pad_size,pad_size), mode='reflect')
    dx_filter = np.array([[1/2,0,-1/2]])
    dy_filter = np.array([[1/2,0,-1/2]]).T
    img1_Ix = convolve(padded_img1, dx_filter, mode='mirror')
    img1_Iy = convolve(padded_img1, dy_filter, mode='mirror')
    img1_It = padded_img2 - padded_img1
    h,w = img1.shape
    for i in range(h):
        for j in range(w):
            Ix = img1_Ix[i:i+window,j:j+window]
            Iy = img1_Iy[i:i+window,j:j+window]
            It = img1_It[i:i+window,j:j+window]
            M = np.zeros((2,2))
            M[0,0] = np.sum(np.square(Ix))
            M[1,1] = np.sum(np.square(Iy))
            M[0,1] = np.sum(Ix * Iy)
            M[1,0] = np.sum(Ix * Iy)
            b = np.zeros((2,1))
            b[0,0] = -np.sum(Ix * It)
            b[1,0] = -np.sum(Iy * It)
            r = np.linalg.inv(M).dot(b)
            U[i,j] = r[0,0]
            V[i,j] = r[1,0]

    return U, V
```

[38]:
```
## Test your implementation on sample parameter values
window = 60
U, V = LucasKanade(grayscale(images[0]),grayscale(images[1]), window)
```

[39]:
```
# Plot
plot_optical_flow(images[0],U,V, 'window = '+str(window))
```

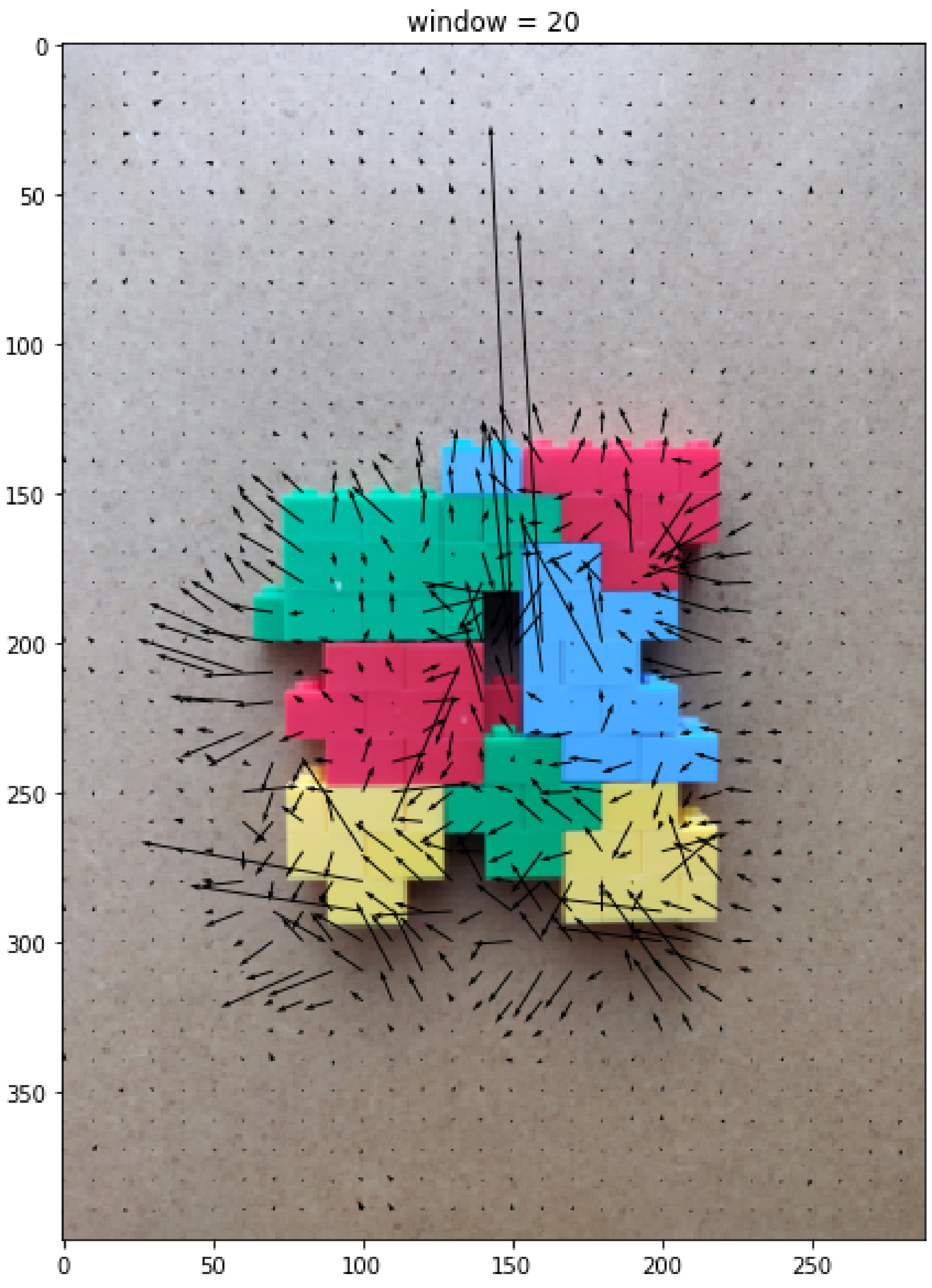

window = 60
0
50
100
150
200
250
300
350
0
50
100
150
200
250

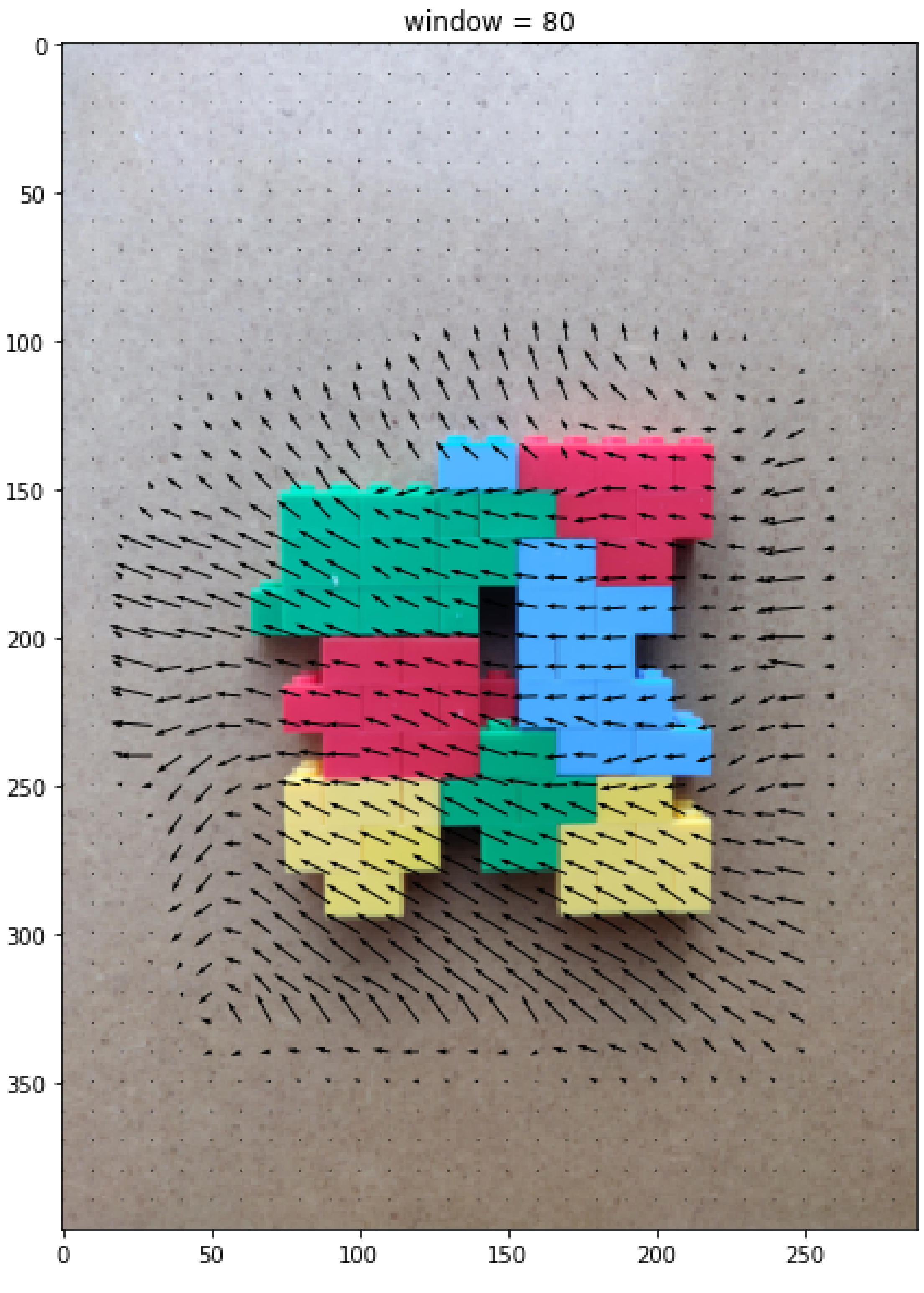
1.5.2 Part 2: Window size

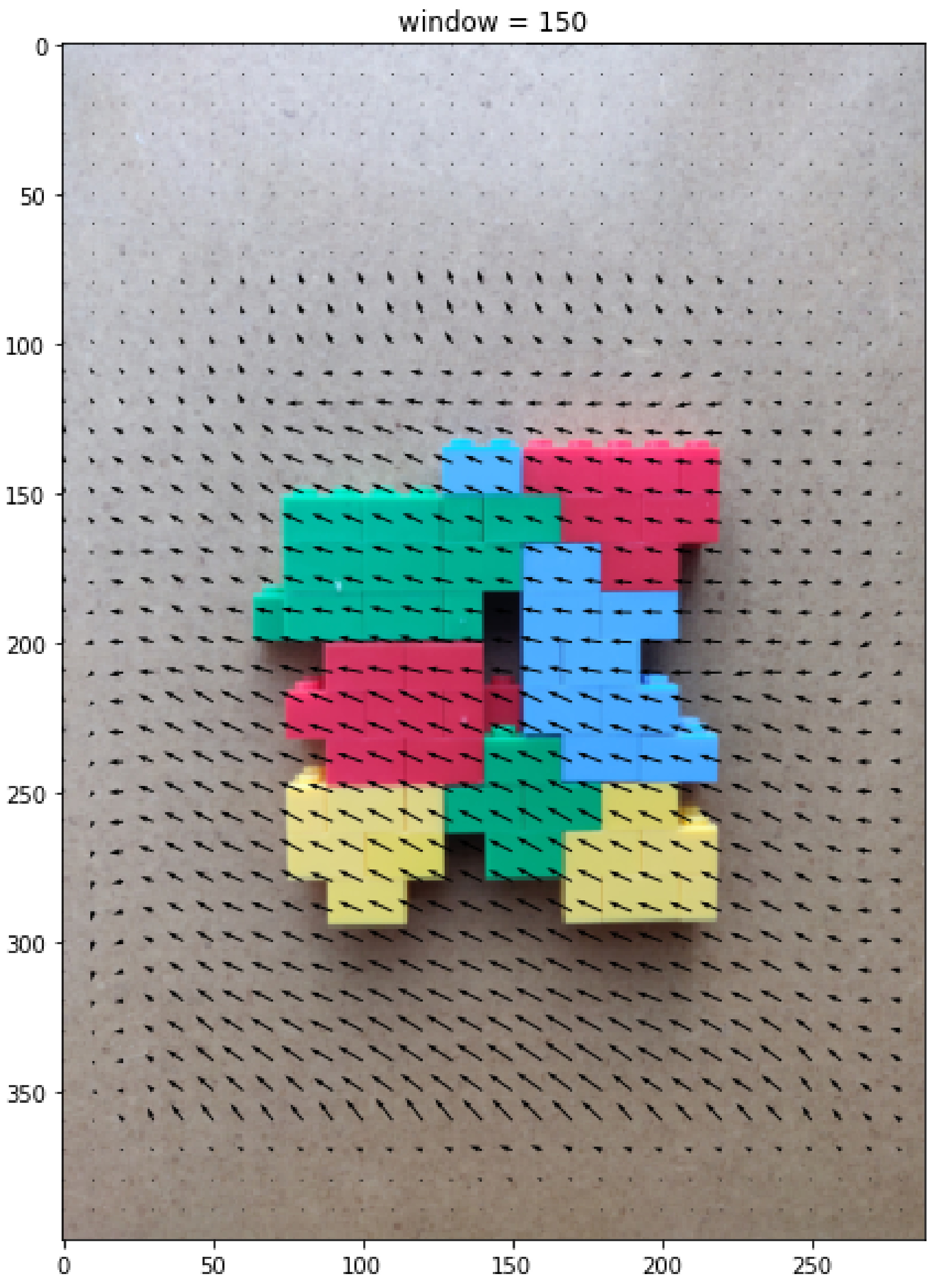
Plot optical flow for the pair of images im1 and im2 for at least 3 different window sizes which leads to observable difference in the results. Comment on the effect of window size on results and justify.

[40]:
```
# Example code, change the window size values
w1, w2, w3 = 20, 80, 150
w_list = [w1, w2, w3]
UV_list = []
for window in [w1, w2, w3]:
    U,V=LucasKanade(grayscale(images[0]),grayscale(images[1]), window)
    UV_list.append((U,V))
```

[41]:
```
for window, (U, V) in zip(w_list, UV_list):
    plot_optical_flow(images[0],U,V, 'window = '+str(window))
```


window = 20
0
50
100
150
200
250
300
350
0
50
100
150
200
250


window = 80
0
50
100
150
200
250
300
350
0
50
100
150
200
250

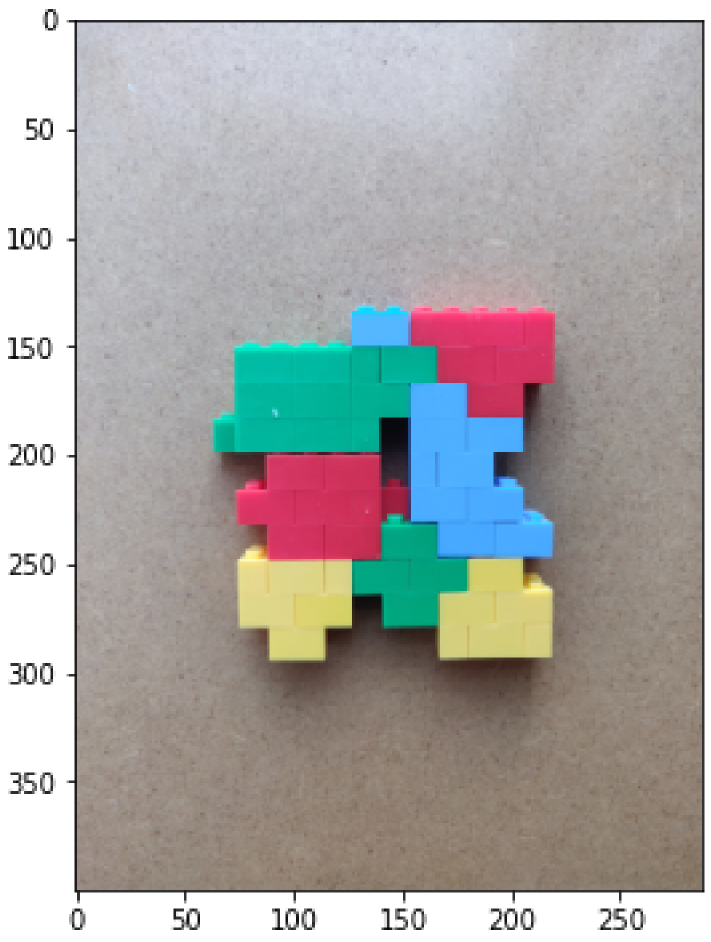

window = 150
0
50
100
150
200
250
300
350
0
50
100
150
200
250

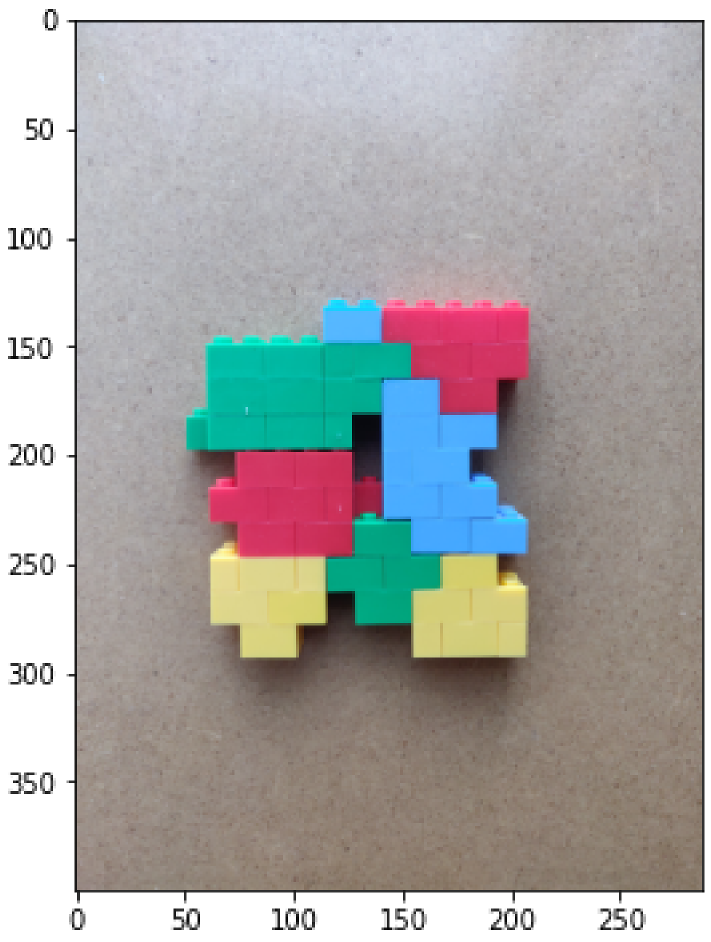
Your Comments on the results of Part 2: A too small window size will cause aperture problem and thus the motion derection might be very off. A too large window size will cause the magnitude of velocity to be very small. So a intermediate window size (60-100) will work out best.

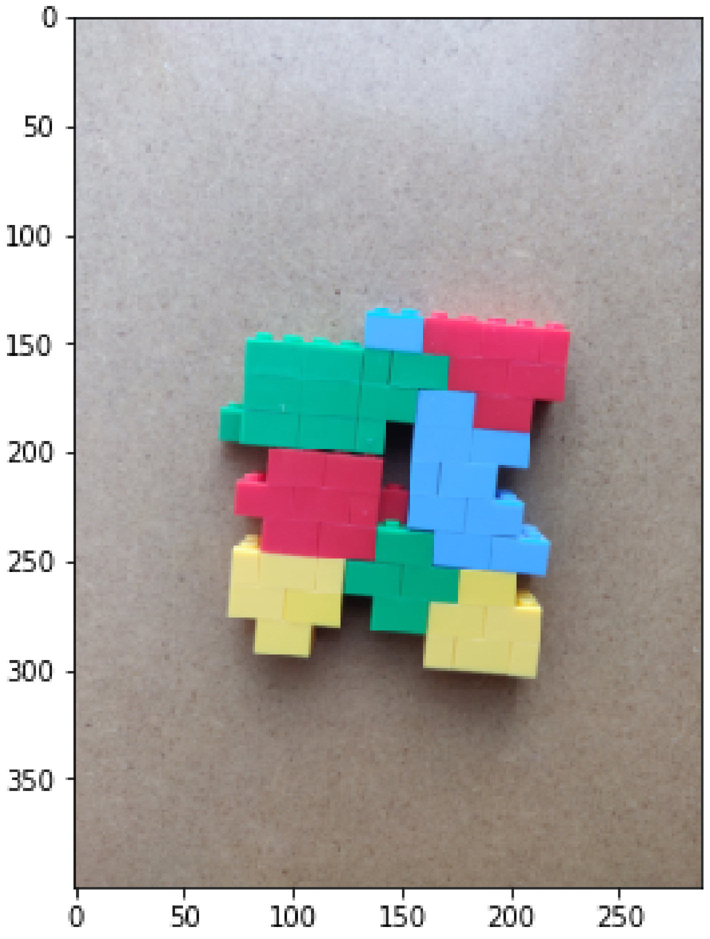
1.5.3 Part 3: All pairs

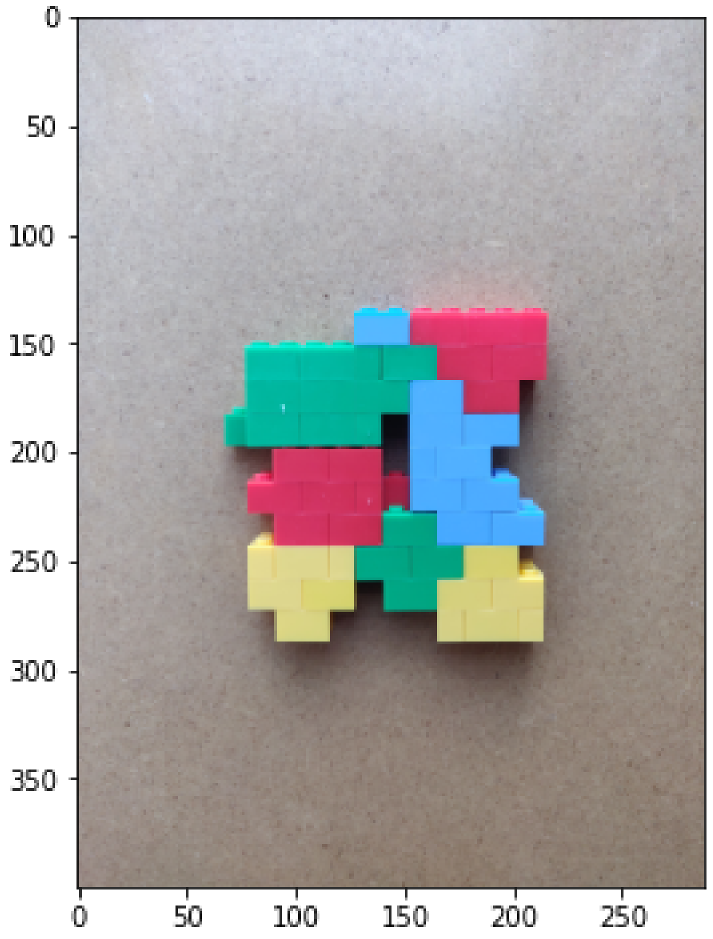
Find optical flow for the pairs (im1,im2), (im1,im3), (im1,im4) using one good window size. Does the optical flow result seem consistent with visual inspection? Comment on the type of motion indicated by results and visual inspection and explain why they might be consistent or inconsistent.

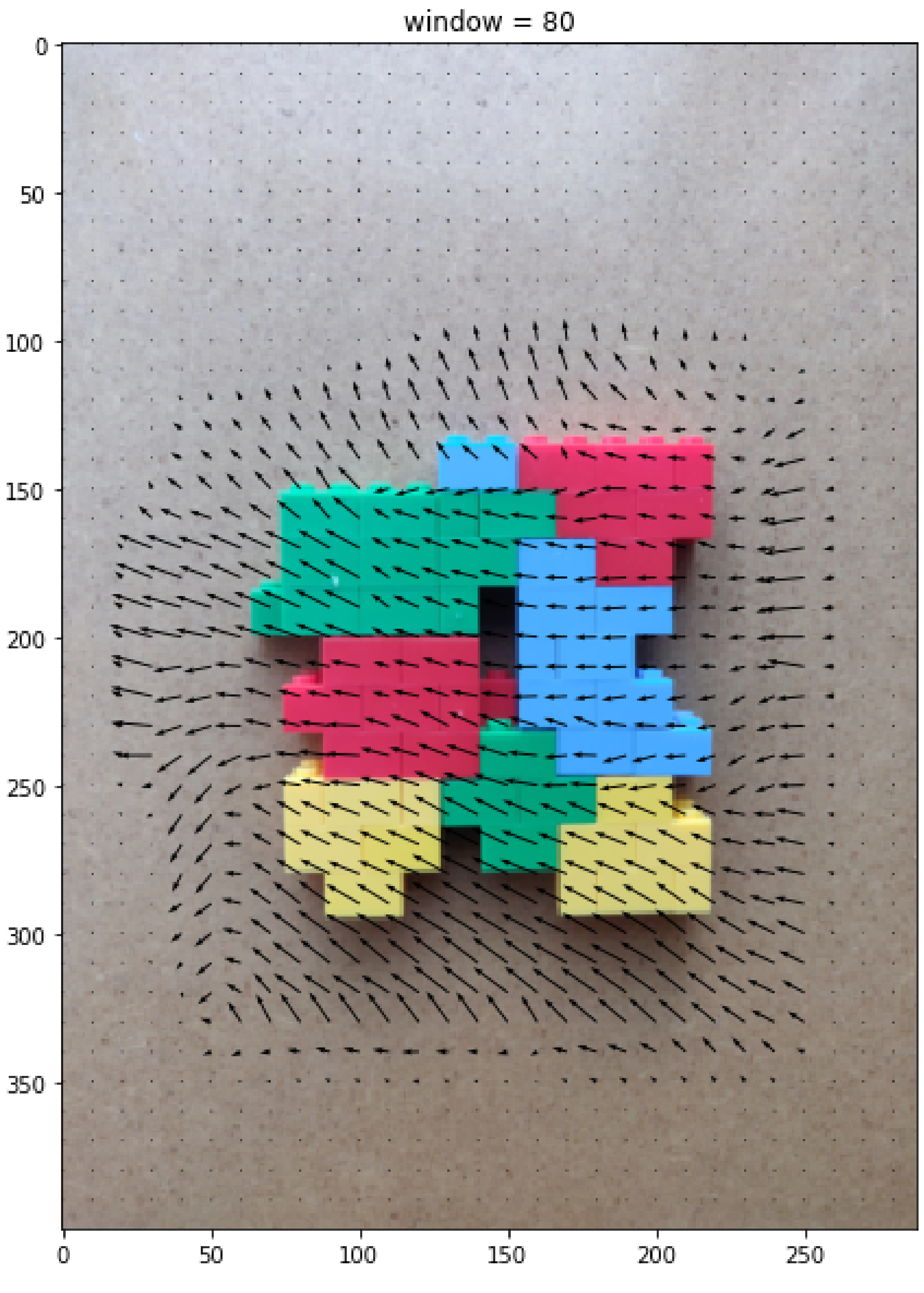
[42]:
```
# Your code here
# Fix a window size of your liking and then use it for all pairs
for img in images:
    plt.figure(figsize=(14, 6))
    plt.imshow(img)
    plt.show()

window = 80
UV_list = []
for (img1,img2) in [(images[0],images[1]), (images[0],images[2]),␣
 ↪(images[0],images[3])]:
    U,V=LucasKanade(grayscale(img1),grayscale(img2), window)
    UV_list.append((U,V))

for (U, V) in UV_list:
    plot_optical_flow(images[0],U,V, 'window = '+str(window))
```

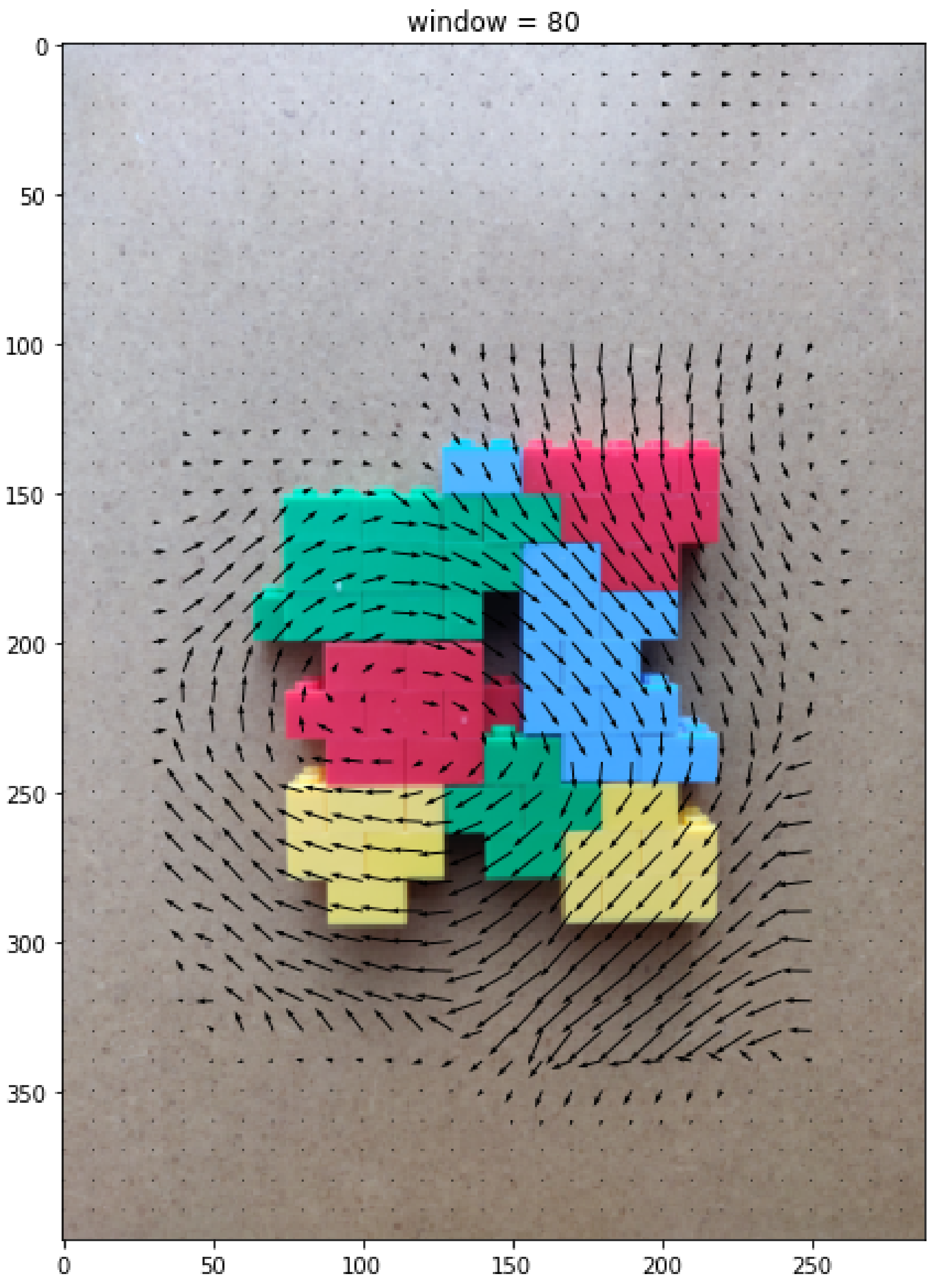

0
50
100
150
200
250
300
350
0
50
100
150
200
250

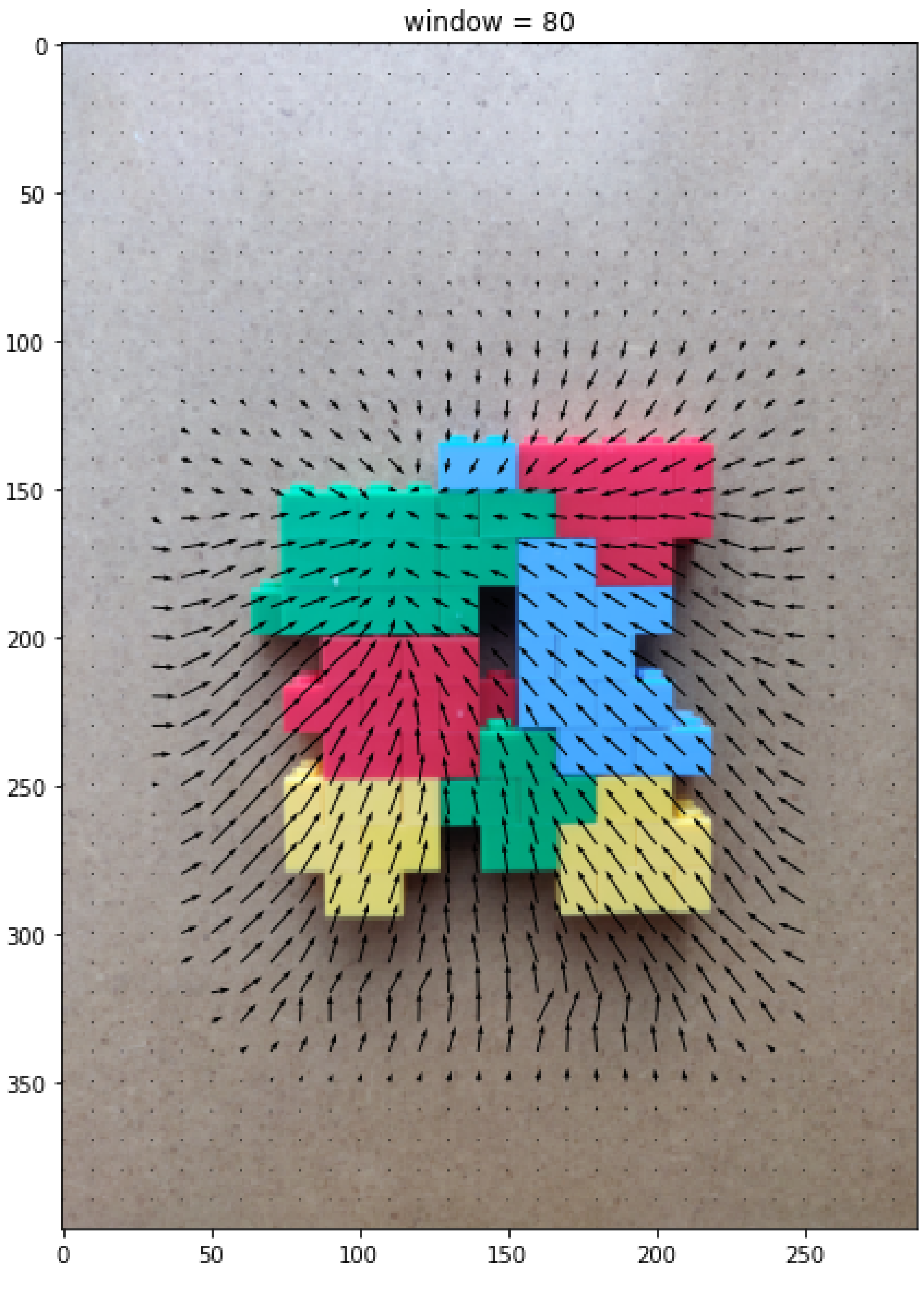

0
50
100
150
200
250
300
350
0
50
100
150
200
250


0
50
100
150
200
250
300
350
0
50
100
150
200
250


0
50
100
150
200
250
300
350
0
50
100
150
200
250


window = 80
0
50
100
150
200
250
300
350
0
50
100
150
200
250


window = 80
0
50
100
150
200
250
300
350
0
50
100
150
200
250


window = 80
0
50
100
150
200
250
300
350
0
50
100
150
200
250

Your Comments on the results of Part 3: The optical flow result is mostly consistent with visual inspection, while there are still small amount of the vectors whose directions are a little off. This might be because the displacement is more than one pixel and thus the “small motion” assumption does not perfectly hold in this case.

1.5.4 Part 4: Multi-Resolution LK

As an additional challenge, you can try implementing the multi-resolution LK optical flow tracker. You can expect to get better results with the multi-resolution LK algorithm.

1.5.5 Submission Instructions

Remember to submit a PDF version of this notebook to Gradescope. You can create a PDF via **File > Download as > PDF via LaTeX** (preferred, if possible), or by downloading as an HTML page and then “printing” the HTML page to a PDF (by opening the print dialog and then choosing the “Save as PDF” option).