

# Assignment4

December 5, 2018

## 1 CSE 252A Computer Vision I Fall 2018 - Assignment 4

1.0.1 Instructor: David Kriegman

1.0.2 Assignment Published On: Wednesday, November 27, 2018

1.0.3 Due On: Friday, December 7, 2018 11:59 pm

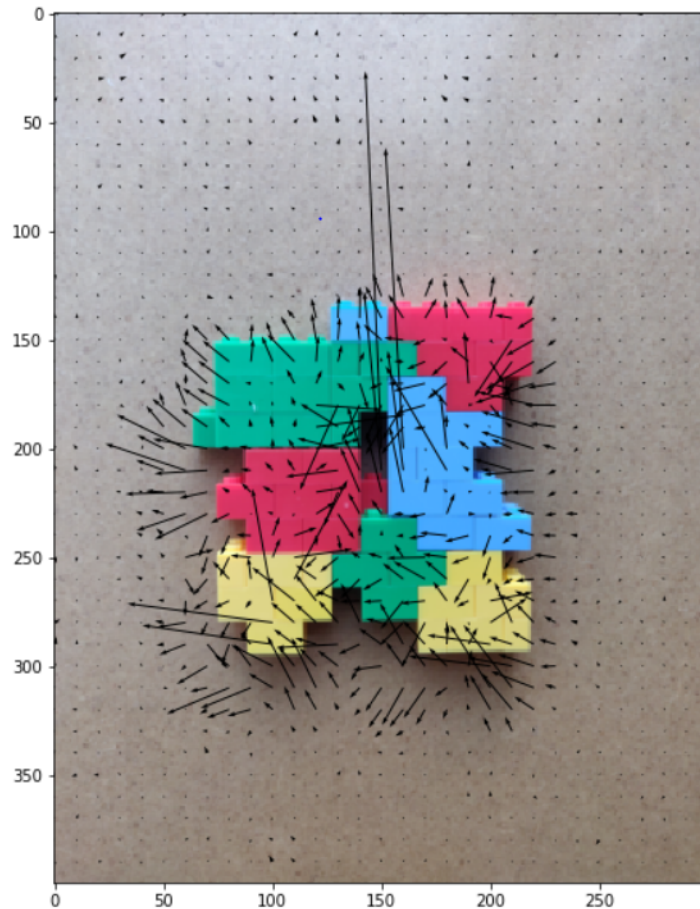
### 1.1 Instructions

- Review the academic integrity and collaboration policies on the course website.
- This assignment must be completed individually.
- Programming aspects of this assignment must be completed using Python in this notebook.
- If you want to modify the skeleton code, you can do so. This has been provided just to provide you with a framework for the solution.
- You may use python packages for basic linear algebra (you can use numpy or scipy for basic operations), but you may not use packages that directly solve the problem.
- If you are unsure about using a specific package or function, then ask the instructor and teaching assistants for clarification.
- You must submit this notebook exported as a pdf. You must also submit this notebook as .ipynb file.
- You must submit both files (.pdf and .ipynb) on Gradescope. You must mark each problem on Gradescope in the pdf.
- **Late policy** - 10% per day late penalty after due date up to 3 days.

### 1.2 Problem 1: Optical Flow [10 pts]

In this problem, the single scale Lucas-Kanade method 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.



### 1.2.1 Part 1: Lucas-Kanade implementation [5 pts]

Implement the Lucas-Kanade method for estimating optical flow. The function 'LucasKanade' needs to be completed.

```
In [76]: %matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import convolve2d as conv2

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):
    """
    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.show()

images=[]
for i in range(1,5):
    images.append(plt.imread('optical_flow_images/im'+str(i)+'.png'))

In [77]: def imconv(img, operator):
    convded_img = np.zeros(img.shape)
    r_center = int((operator.shape[0]-1)/2)
    c_center = int((operator.shape[1]-1)/2)
    num_row, num_col = img.shape

    for i in range(r_center, num_row-r_center):
        for j in range(c_center, num_col-c_center):
            convded_img[i][j] = (img[(i-r_center):(i+r_center+1),\
                                   (j-c_center):(j+c_center+1)] * operator).sum()
    return convded_img #[r_center : (num_row-r_center), c_center : (num_col-c_center)]

def LucasKanade(im1, im2, windowSize):
    '''
    Inputs: the two images and window size
    Return U,V
    '''
    U = np.zeros(im1.shape)

```

```
V = np.zeros(im1.shape)

'''
Your code here
'''

sum_window = np.ones((windowSize, windowSize))

img_dy, img_dx = np.gradient(im1) # y is row, x is column
img_di = im2 - im1

img_dxdx = img_dx * img_dx
img_dydy = img_dy * img_dy
img_dxdy = img_dx * img_dy
img_dxdi = img_dx * img_di
img_dydi = img_dy * img_di

cov_sum_dxdx = imconv(img_dxdx, sum_window)
cov_sum_dxdy = imconv(img_dxdy, sum_window)
cov_sum_dydy = imconv(img_dydy, sum_window)

cov_sum_dxdi = imconv(img_dxdi, sum_window)
cov_sum_dydi = imconv(img_dydi, sum_window)

for i in range(im1.shape[0]):
    for j in range(im1.shape[1]):
        M = np.array([[cov_sum_dxdx[i][j], cov_sum_dxdy[i][j]],
                       [cov_sum_dxdy[i][j], cov_sum_dydy[i][j]]])

        b = - np.array([[cov_sum_dxdi[i][j]], [cov_sum_dydi[i][j]]])

        result = np.linalg.pinv(M).dot(b)
        U[i][j] = result[0][0]
        V[i][j] = result[1][0]

return U,V
```

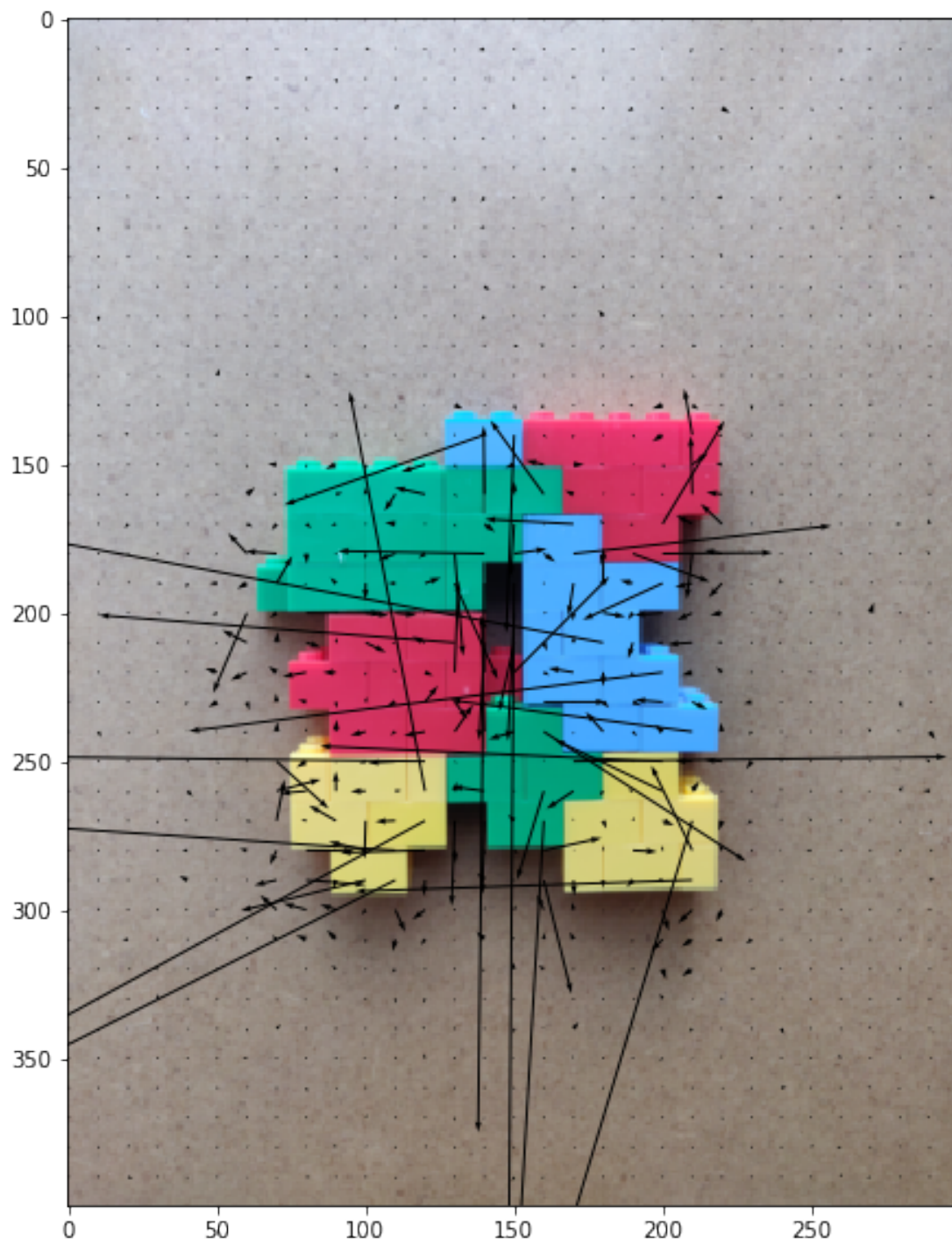
### 1.2.2 Part 2: Window size [2 pts]

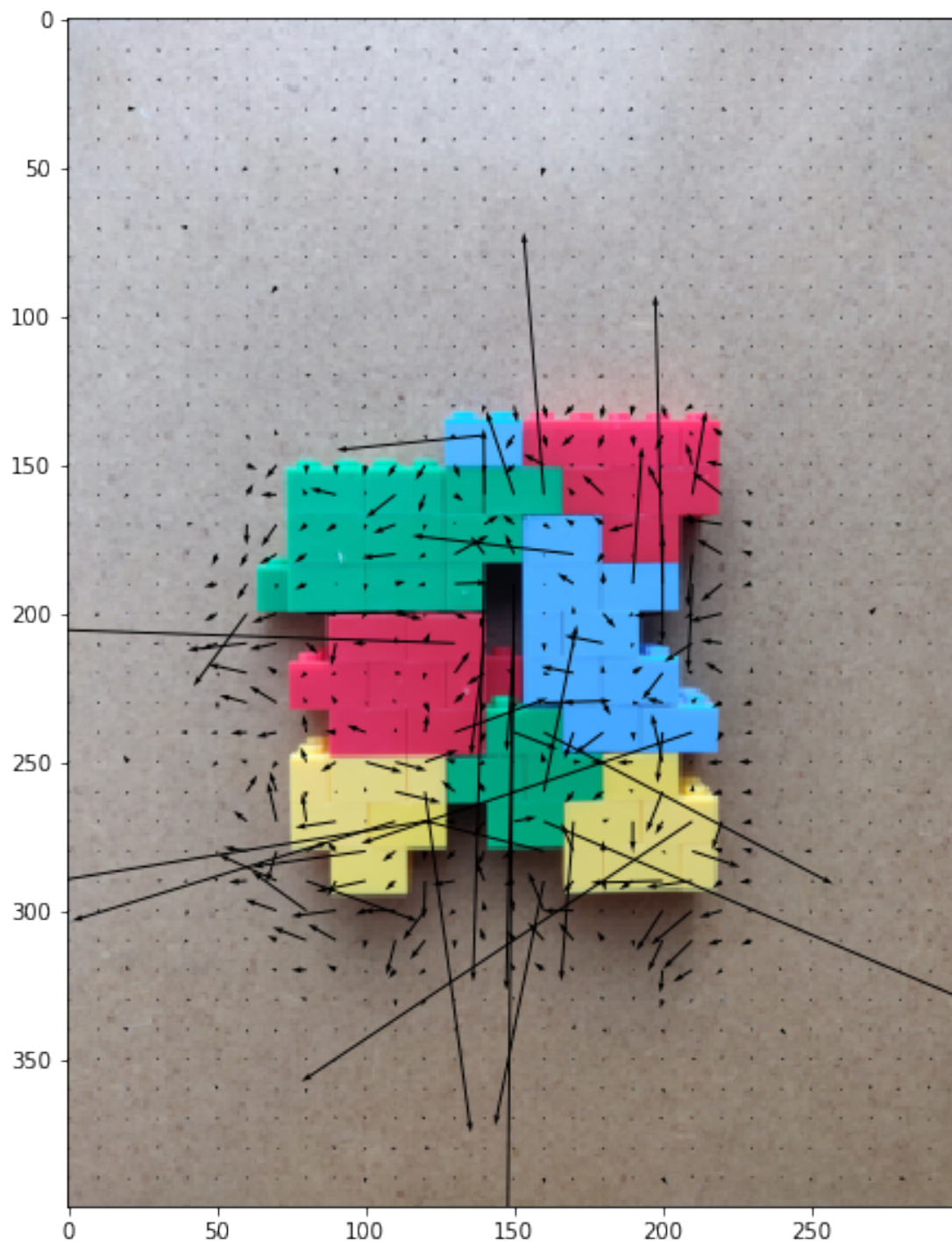
Plot optical flow for the pair of images im1 and im2 for at least 3 different window sizes which leads to observable difference in the results. Comment on the effect of window size on results and justify.

```
In [11]: # Example code, change as required
         window=5

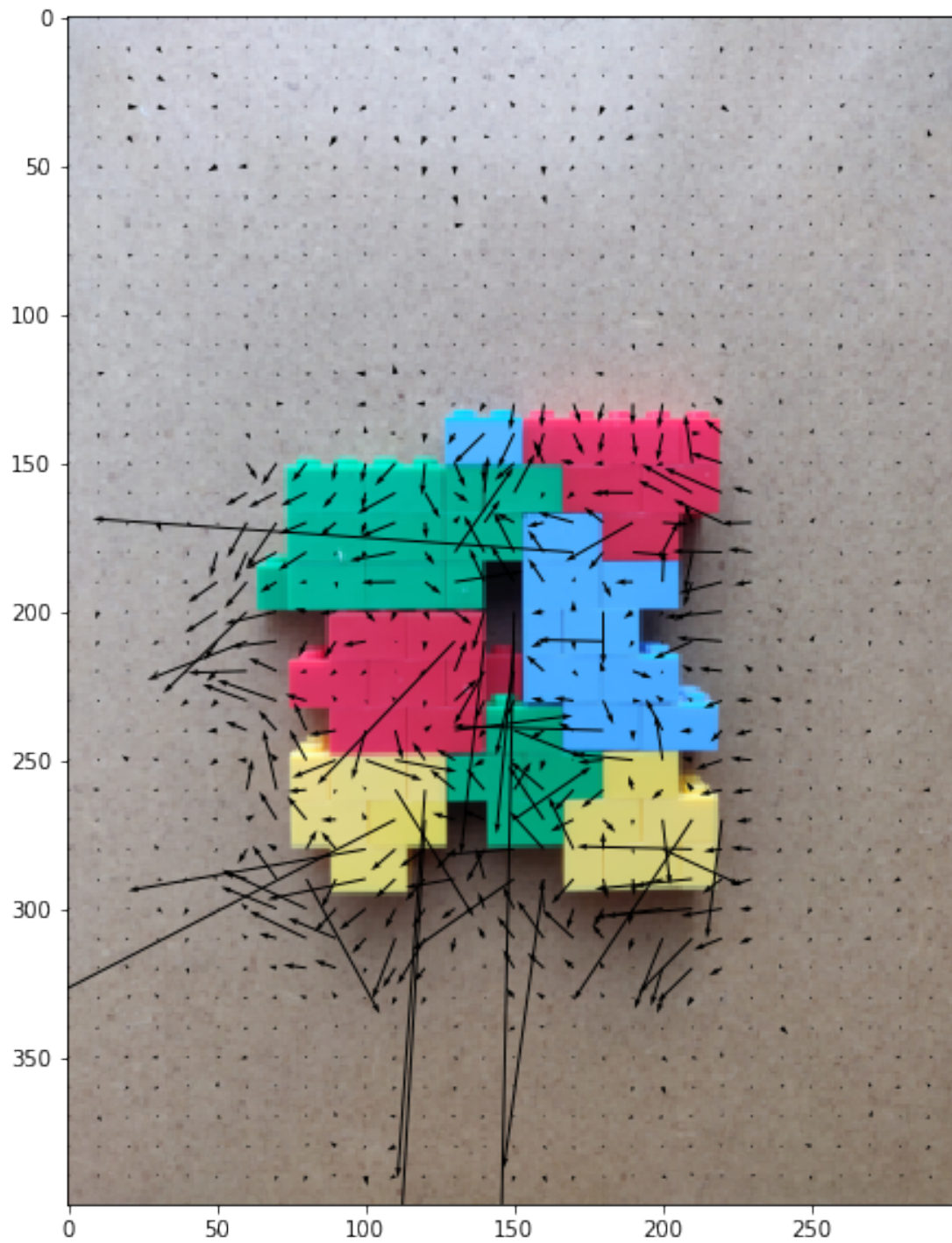
         for window in [5, 11, 15]:
```

```
U, V=LucasKanade(grayscale(images[0]),grayscale(images[1]),window)
plot_optical_flow(images[0],U,V)
```









In general, with appropriate large window size, the optical flow result looks more accurate than results from smaller window size.

### 1.2.3 Part 3: All pairs [3 pts]

Find optical flow for the pairs (im1,im2), (im1,im3), (im1,im4) using a 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.

```
In [96]: # Example code, change as required
        window = 21

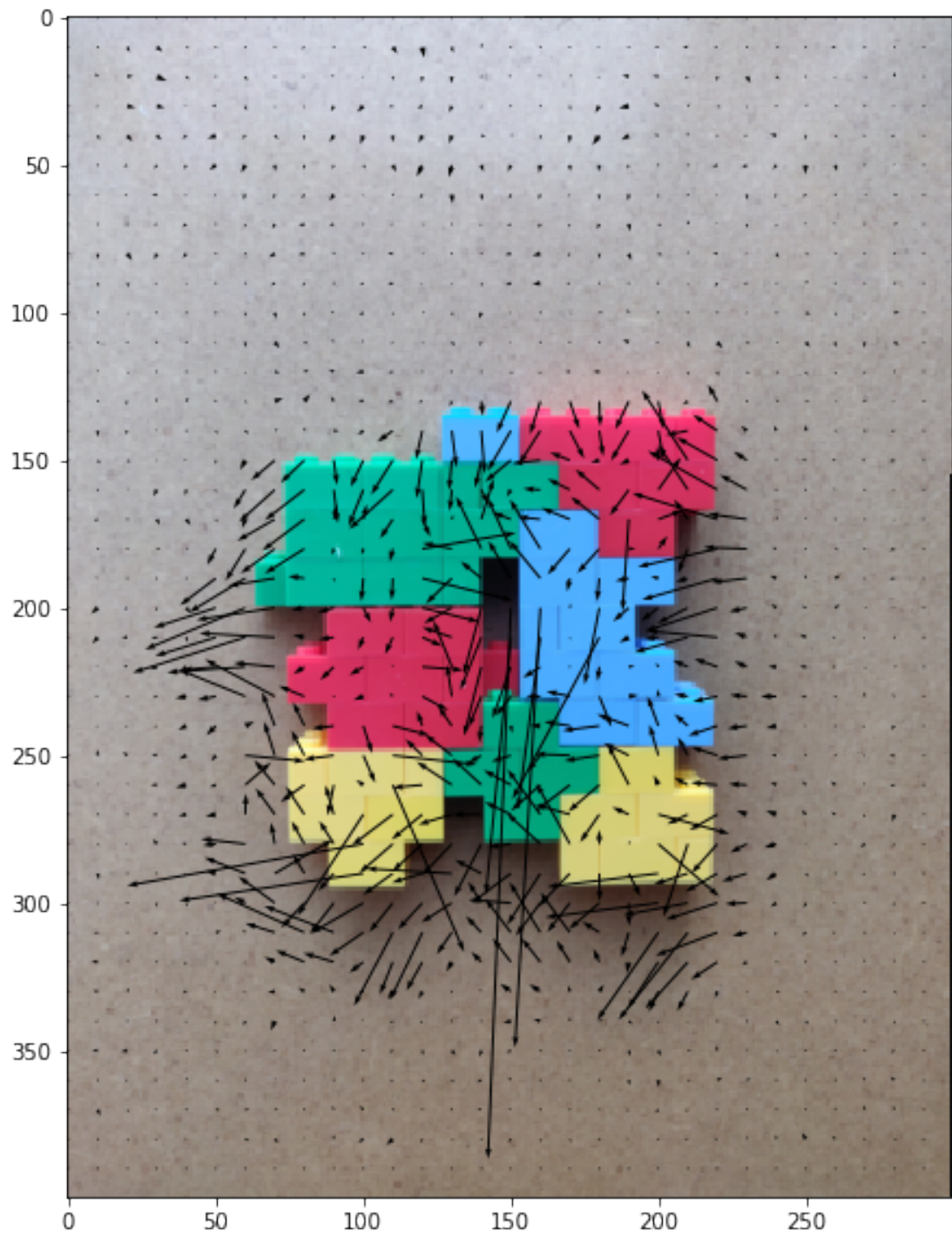
        print('im1 and im2')
        U, V=LucasKanade( grayscale(images[0]), grayscale(images[1]), window)
        plot_optical_flow(images[0],U,V)

        print('im1 and im3')
        U, V=LucasKanade( grayscale(images[0]), grayscale(images[2]), window)
        plot_optical_flow(images[0],U,V)

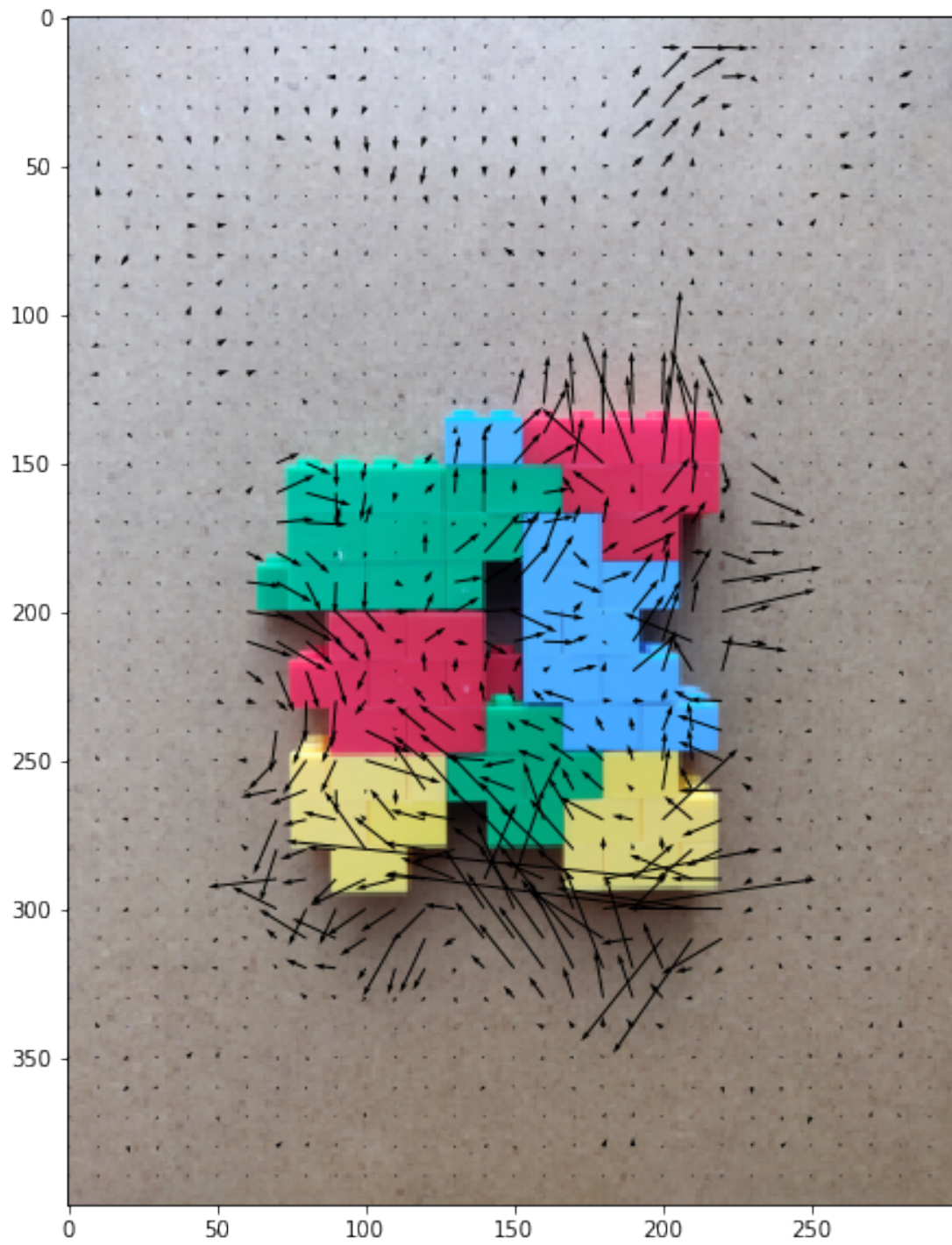
        print('im1 and im4')
        U, V=LucasKanade( grayscale(images[0]), grayscale(images[3]), window)
        plot_optical_flow(images[0],U,V)

im1 and im2
```

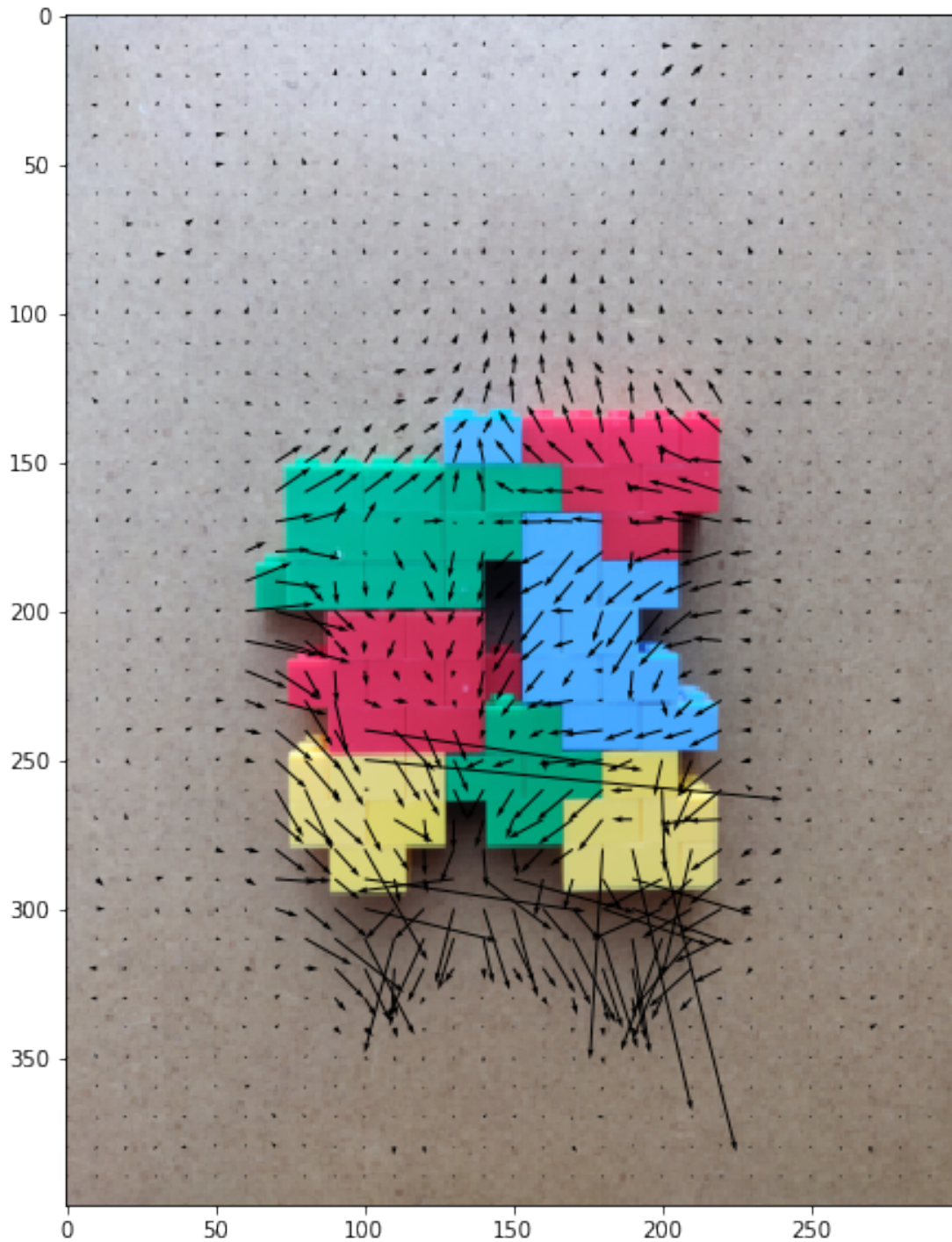




im1 and im3



im1 and im4



For im1 to im2, the ground truth is moving left, and in this optical flow result most arrows are consistent moving left. For im1 to im3, the ground truth is rotating clockwise, and in this optical flow result most arrows are consistent clockwise rotating. For im1 to im4, the ground truth operation is zoom, and in optical flow result most arrow are pointing to outside.

In general, most optical flow arrows are consistent to its inspection, but some are not. It may

be the algorithm is based on brightness constancy and motion of brightness patterns, and some pixel may not perfect obey this rule. Also, the calculation about this flow too much depend on its window size, and is easily contaminated by its surrounding pixels.

### 1.3 Problem 2: 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://www.tensorflow.org/install/> to install Tensorflow on your computer. If you are using the Anaconda distribution for python, you can check out <https://www.anaconda.com/blog/developer-blog/tensorflow-in-anaconda/>.

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

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

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

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

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

```
In [78]: import tensorflow as tf
         hello = tf.constant('Hello, TensorFlow!')
         sess = tf.Session()
         print(sess.run(hello))
```

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

```
In [79]: import os
         import struct

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

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

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



```

fname_lbl = os.path.join(path, 't10k-labels-idx1-ubyte')

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

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

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

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

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

```

Some helper functions are given below.

```

In [80]: # a generator for batches of data
# yields data (batchsize, 3, 32, 32) and labels (batchsize)
# if shuffle, it will load batches in a random order
def DataBatch(data, label, batchsize, shuffle=True):
    n = data.shape[0]
    if shuffle:
        index = np.random.permutation(n)
    else:
        index = np.arange(n)
    for i in range(int(np.ceil(n/batchsize))):
        inds = index[i*batchsize : min(n,(i+1)*batchsize)]
        yield data[inds], label[inds]

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

```

```

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

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

```

Random classifier accuracy: 9.860000

### 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  $n \times n$  where  $n$  is the number of classes. Entry  $M[i,j]$  should contain the fraction of images of class  $i$  that was classified as class  $j$ .

```

In [81]: # 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

         # row i is target, column j is prediction

classes = np.arange(10)
def Confusion(testData, testLabels, classifier):
    num_classes = 10
    M = np.zeros([num_classes, num_classes])
    batchsize=50

    for batch_data, batch_labels in DataBatch(testData, testLabels, batchsize):
        batch_predictions = classifier(batch_data)
        for k in range(batchsize):
            M[batch_labels[k], batch_predictions[k]] += 1

    for j in range(num_classes):
        M[:, j] = M[:, j]/np.sum(M[:, j])
    return M

def VisualizeConfussion(M):
    plt.figure(figsize=(14, 6))

```



```

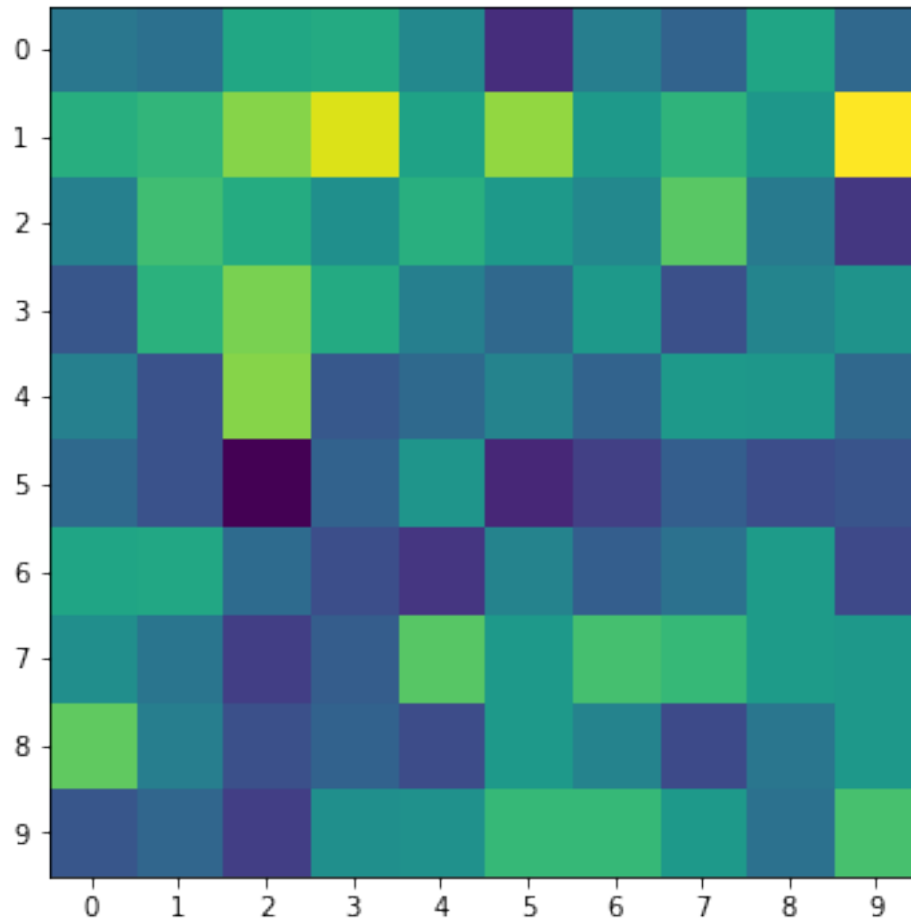
plt.imshow(M)
plt.xticks(np.arange(len(classes)), classes)
plt.yticks(np.arange(len(classes)), classes)
plt.show()
print(np.round(M,2))

```

```

M =Confusion(testData, testLabels, randomClassifier)
VisualizeConfussion(M)

```



```

[[0.1  0.1  0.11 0.11 0.1  0.08 0.1  0.09 0.11 0.09]
 [0.11 0.11 0.12 0.13 0.11 0.12 0.1  0.11 0.1  0.13]
 [0.1  0.11 0.11 0.1  0.11 0.1  0.1  0.11 0.1  0.08]
 [0.09 0.11 0.12 0.11 0.1  0.09 0.1  0.09 0.1  0.1 ]
 [0.1  0.09 0.12 0.09 0.09 0.1  0.09 0.1  0.1  0.09]
 [0.09 0.09 0.08 0.09 0.1  0.08 0.09 0.09 0.09 0.09]
 [0.11 0.11 0.09 0.09 0.08 0.1  0.09 0.1  0.1  0.09]
 [0.1  0.1  0.09 0.09 0.11 0.1  0.11 0.11 0.1  0.1 ]

```

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

### 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 '4' is most often predicted to be, other than '4'.

```
In [29]: from sklearn.neighbors import KNeighborsClassifier
class KNNClassifier():
    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
        '''

        shaped_trainData = trainData.reshape((trainData.shape[0], -1))
        self.clf = KNeighborsClassifier(self.k, weights='uniform')
        self.clf.fit(shaped_trainData, trainLabels)

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

        X = x.reshape((x.shape[0], -1))
        Y = self.clf.predict(X)
        return Y

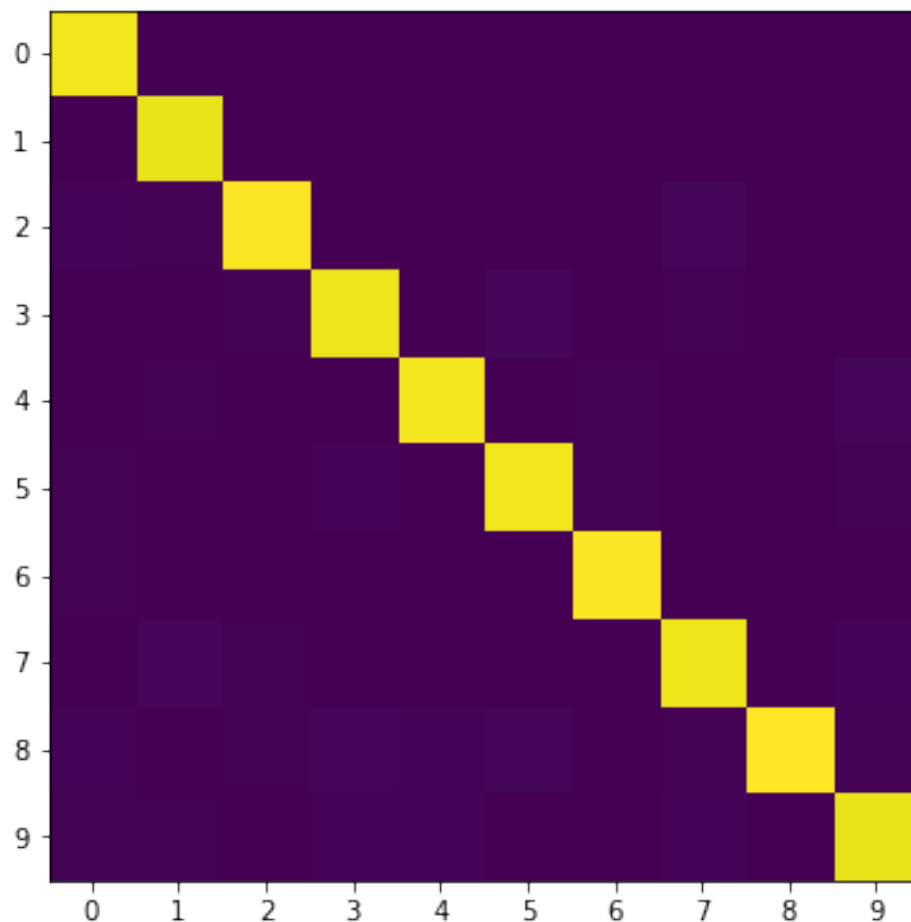
# test your classifier with only the first 100 training examples (use this
# while debugging)
# note you should get ~ 65 % accuracy
knnClassifierX = KNNClassifier()
```

```
knnClassifierX.train(trainData[:100], trainLabels[:100])
print ('KNN classifier accuracy: %f' %test(testData, testLabels, knnClassifierX))
```

KNN classifier accuracy: 64.760000

```
In [30]: # test your classifier with all the training examples (This may take a while)
knnClassifier = KNNClassifier()
knnClassifier.train(trainData, trainLabels)

# display confusion matrix for your KNN classifier with all the training examples
M = Confusion(testData, testLabels, knnClassifier)
VisualizeConfussion(M)
```



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

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

Based on the observation of confusion matrix 5th row, other than number 4, the most often predicted to be is 9.

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

Here you will implement a simple KNN classifier 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 [31]: class PCAKNNClassifier():
    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.k = k
        self.num_components = components

    def train(self, trainData, trainLabels):
        '''
        your code here
        '''

        shaped_trainData = trainData.reshape(trainData.shape[0], -1)
        self.row_mean = np.mean(shaped_trainData, axis = 0)
        U, S, Vh = np.linalg.svd(shaped_trainData - self.row_mean)

        self.pca_basis = Vh[:self.num_components+1].T # (K * 784) T
        pca_trainData = np.dot(shaped_trainData - self.row_mean, self.pca_basis)

        self.clf = KNeighborsClassifier(self.k, weights='uniform')
        self.clf.fit(pca_trainData, trainLabels)

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

```

        your code here
        '''
        X = x.reshape((x.shape[0], -1))
        X = X - self.row_mean
        pca_X = np.dot(X, self.pca_basis)

        Y = self.clf.predict(pca_X)
        return Y

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

```

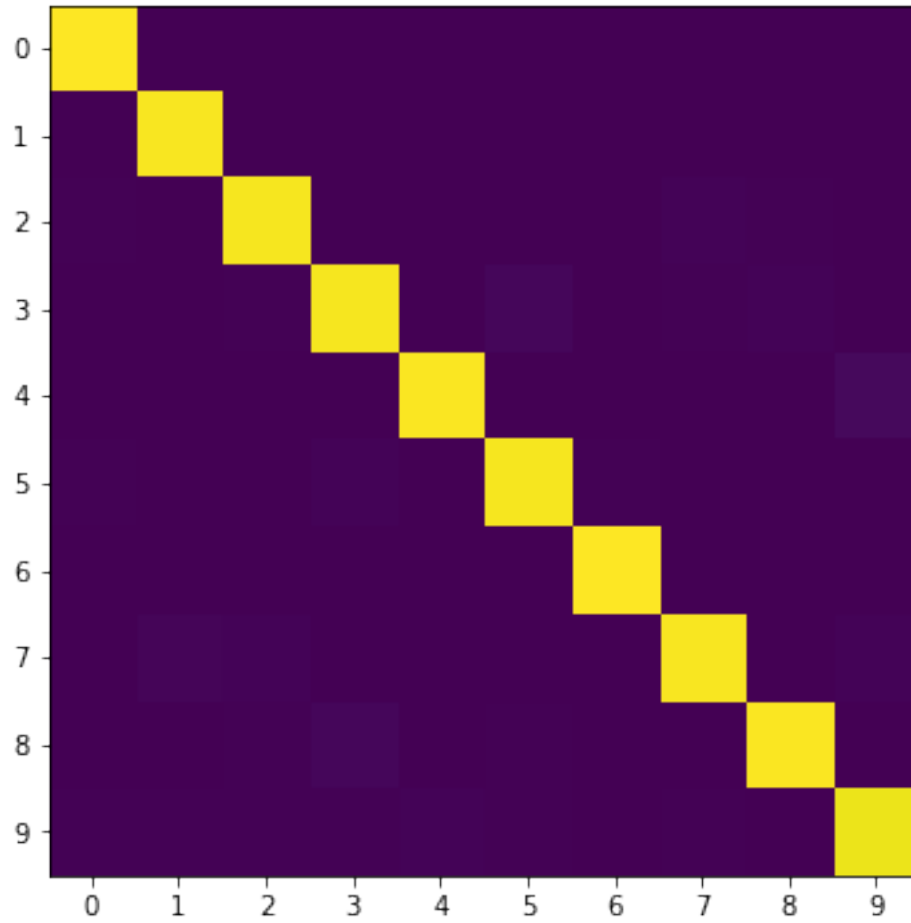
KNN classifier accuracy: 65.940000

```

In [32]: # test your classifier with all the training examples (This may take a while)
pcaknnClassifier = PCAKNNClassifier()
pcaknnClassifier.train(trainData, trainLabels)

# display confusion matrix for your PCA KNN classifier with all the training examples
M = Confusion(testData, testLabels, pcaknnClassifier)
VisualizeConfussion(M)

```



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

The testing time for PCA KNN classifier is less than that for KNN classifier, since in PCA method the data dimension has been reduced, and it should less time to compute L2 norm to find top3 best match items.



## 1.4 Problem 3: Deep learning [10 pts]

Below is some helper code to train your deep networks. You can look at [https://www.tensorflow.org/get\\_started/mnist/beginners](https://www.tensorflow.org/get_started/mnist/beginners) for reference.

```
In [85]: # base class for your Tensorflow networks. It implements the training loop
# (train) and prediction(__call__) for you.
# You will need to implement the __init__ function to define the networks
# structures in the following problems.

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

    def train(self, trainData, trainLabels, epochs=1, batchsize=50):
        self.prediction = tf.argmax(self.y, 1)
        self.cross_entropy = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits=self.y, labels=trainLabels))

        self.train_step = tf.train.AdamOptimizer(1e-4).minimize(self.cross_entropy)

        self.correct_prediction = tf.equal(self.prediction, self.y)
        self.accuracy = tf.reduce_mean(tf.cast(self.correct_prediction, tf.float32))
        self.sess.run(tf.global_variables_initializer())

        for epoch in range(epochs):
            for i, (data, label) in enumerate(DataBatch(trainData, trainLabels, batchsize=batchsize)):

                data=np.expand_dims(data,-1)

                _, acc = self.sess.run([self.train_step, self.accuracy], feed_dict={self.x: data, self.y: label})

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

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

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

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

# helper function to get bias variable
def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

```

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

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

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

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

```

### 1.4.1 Part 1: 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 [87]: # test the example linear classifier (note you should get around 90% accuracy
# for 10 epochs and batchsize 50)
linearClassifier = LinearClassifier()
linearClassifier.train(trainData, trainLabels, epochs=10)

weights1 = linearClassifier.sess.run(linearClassifier.weights)[0]
for i in range(10):
    plt.subplot(2, 5, i+1)
    weight = weights1[:,i]
    #weight = linearClassifier.sess.run(linearClassifier.weights)[0][:,i]
    weight = (weight - weight.min()) / (weight.max() - weight.min())
    plt.title(i)
    plt.imshow(weight.reshape([28,28]), cmap=plt.get_cmap('plasma'))
    frame1 = plt.gca()
    frame1.axes.get_xaxis().set_visible(False)
    frame1.axes.get_yaxis().set_visible(False)
plt.show()

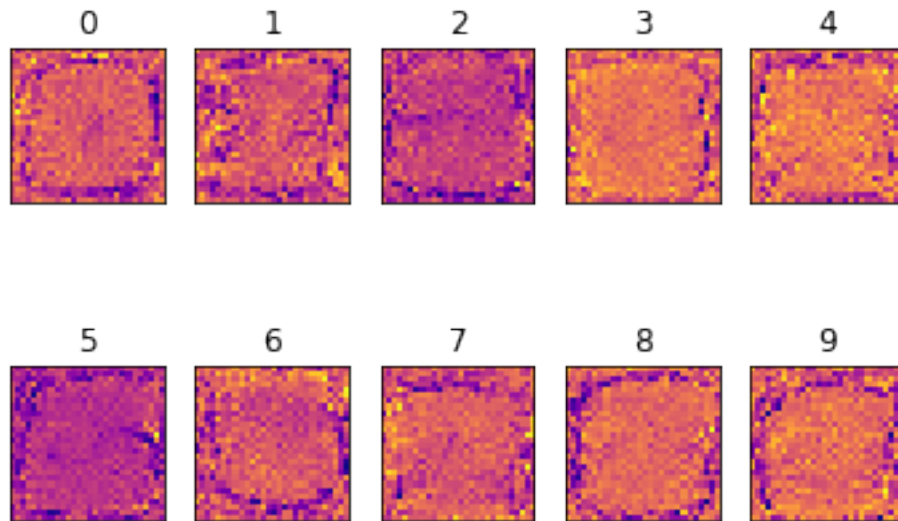
```

```

Epoch:1 Accuracy: 88.280000
Epoch:2 Accuracy: 90.170000
Epoch:3 Accuracy: 89.980000
Epoch:4 Accuracy: 89.560000
Epoch:5 Accuracy: 89.340000
Epoch:6 Accuracy: 89.090000

```

Epoch:7 Accuracy: 89.900000  
Epoch:8 Accuracy: 86.480000  
Epoch:9 Accuracy: 90.480000  
Epoch:10 Accuracy: 90.730000



The weight figures all look like a circle, and somehow look like corresponding numbers with more dark pixel on its number's edge (combine the lower level feature and higher level feature). As this model only uses 10 epoch to train, the training result may be not good enough.

#### 1.4.2 Part 2: 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 [88]: class MLPClassifier(TFClassifier):
    def __init__(self, classes=10, hidden=100):
        '''
        your code here
        '''

        self.W_layer_h1 = weight_variable([28*28*1, hidden])
        self.b_layer_h1 = bias_variable([hidden])

        self.W_layer_o = weight_variable([hidden, classes])
        self.b_layer_o = bias_variable([classes])

        self.x = tf.placeholder(tf.float32, shape=[None, 28, 28, 1]) # input batch of images
        a_layer_h1 = tf.matmul(tf.reshape(self.x, (-1, 28 * 28 * 1)), self.W_layer_h1)
        g_layer_h1 = tf.nn.relu(a_layer_h1)
        self.y = tf.matmul(g_layer_h1, self.W_layer_o) + self.b_layer_o

        self.y_ = tf.placeholder(tf.int64, shape=[None]) # input labels
        self.sess = tf.Session()

In [89]: # test the example linear classifier (note you should get around 90% accuracy
# for 10 epochs and batchsize 50)
mlpClassifier = MLPClassifier()
mlpClassifier.train(trainData, trainLabels, epochs=10)

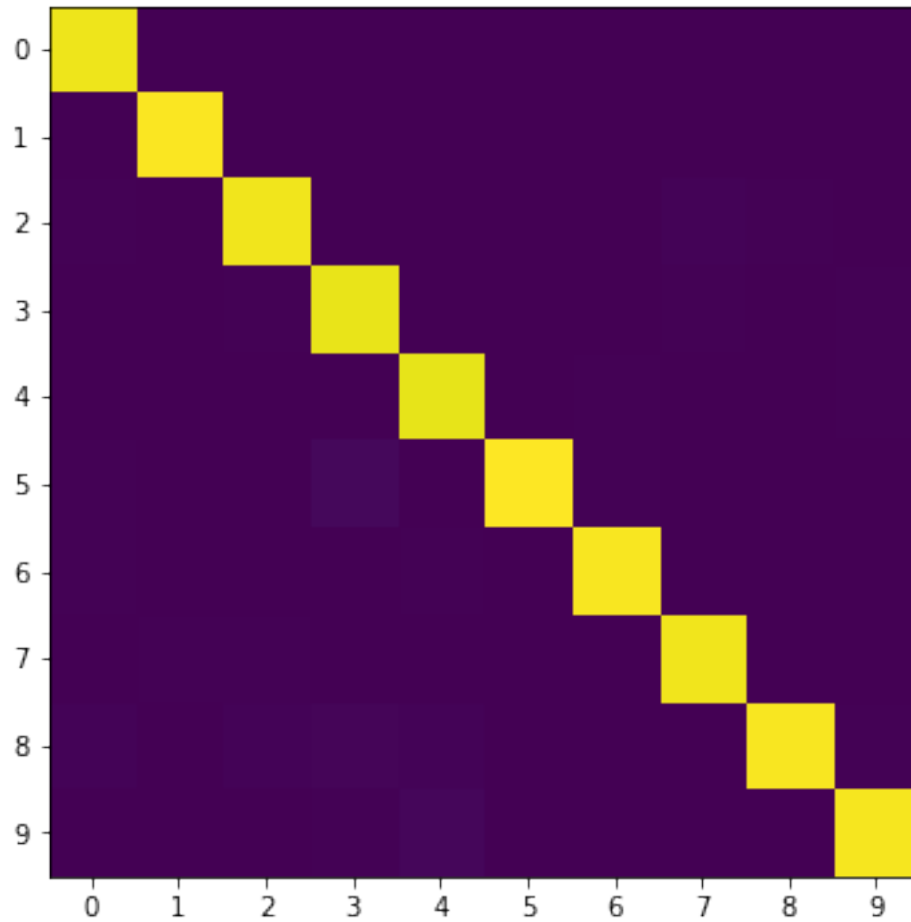
# display confusion matrix
M = Confusion(testData, testLabels, mlpClassifier)
VisualizeConfusion(M)

weights2 = mlpClassifier.sess.run(mlpClassifier.W_layer_h1)[: , :10]
for i in range(10):
    plt.subplot(2, 5, i+1)
    weight = weights2[:, i]
    #weight = linearClassifier.sess.run(linearClassifier.weights)[0][:, i]
    weight = (weight - weight.min()) / (weight.max() - weight.min())
    plt.title(i)
    plt.imshow(weight.reshape([28, 28]), cmap=plt.get_cmap('plasma'))
    frame1 = plt.gca()
    frame1.axes.get_xaxis().set_visible(False)
    frame1.axes.get_yaxis().set_visible(False)
plt.show()

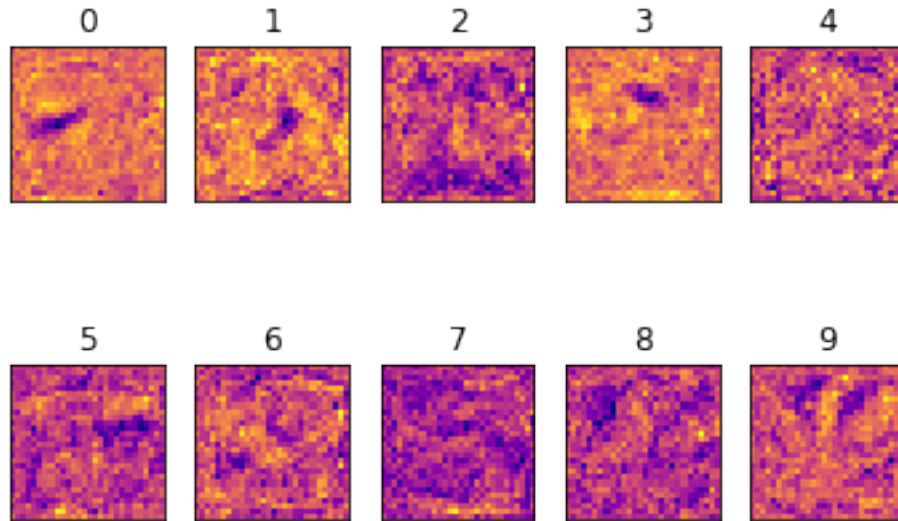
Epoch:1 Accuracy: 95.650000
Epoch:2 Accuracy: 96.360000
Epoch:3 Accuracy: 97.230000
Epoch:4 Accuracy: 97.130000
Epoch:5 Accuracy: 97.490000

```

Epoch:6 Accuracy: 97.590000  
Epoch:7 Accuracy: 97.730000  
Epoch:8 Accuracy: 97.660000  
Epoch:9 Accuracy: 97.570000  
Epoch:10 Accuracy: 97.500000



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



Since this model use two hidden layers, the extracted weight figures look more like lower level features and it is much different from the previous plot. It is hard to say what it really looks like in this plot.

### 1.4.3 Part 3: Convolutional Neural Network (CNN) [5 pts]

Here you will implement a CNN with the following architecture:

- $n=5$
- $\text{ReLU}(\text{Conv}(\text{kernel\_size}=4 \times 4, \text{stride}=2, \text{output\_features}=n))$
- $\text{ReLU}(\text{Conv}(\text{kernel\_size}=4 \times 4, \text{stride}=2, \text{output\_features}=n \times 2))$
- $\text{ReLU}(\text{Conv}(\text{kernel\_size}=4 \times 4, \text{stride}=2, \text{output\_features}=n \times 4))$
- $\text{Linear}(\text{output\_features}=\text{classes})$

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

```
In [90]: def conv2d(x, W, stride=2):
          return tf.nn.conv2d(x, W, strides=[1, stride, stride, 1], padding='SAME')

class CNNClassifier(TFClassifier):
    def __init__(self, classes=10, n=5):
        """
        your code here
        """
        # model variables
        W_conv1 = weight_variable([4, 4, 1, n])
        b_conv1 = bias_variable([n])

        W_conv2 = weight_variable([4, 4, n, n * 2])
```



```

b_conv2 = bias_variable([n * 2])

W_conv3 = weight_variable([4, 4, n * 2, n * 4])
b_conv3 = bias_variable([n * 4])

W_fc = weight_variable([16 * n * 4, classes])
b_fc = bias_variable([classes])

# model flows
self.x = tf.placeholder(tf.float32, [None, 28, 28, 1])

a_conv1 = conv2d(tf.reshape(self.x, (-1, 28, 28, 1)), W_conv1) + b_conv1
g_conv1 = tf.nn.relu(a_conv1)

a_conv2 = conv2d(g_conv1, W_conv2) + b_conv2
g_conv2 = tf.nn.relu(a_conv2)

a_conv3 = conv2d(g_conv2, W_conv3) + b_conv3
g_conv3 = tf.nn.relu(a_conv3)

fc_x = tf.reshape(g_conv3, [-1, 16 * n * 4])

self.y = tf.matmul(fc_x, W_fc) + b_fc

self.y_ = tf.placeholder(tf.int64, shape=[None]) # input labels
self.sess = tf.Session()

```

```

In [91]: cnnClassifier = CNNClassifier()
         cnnClassifier.train(trainData, trainLabels, epochs=10)

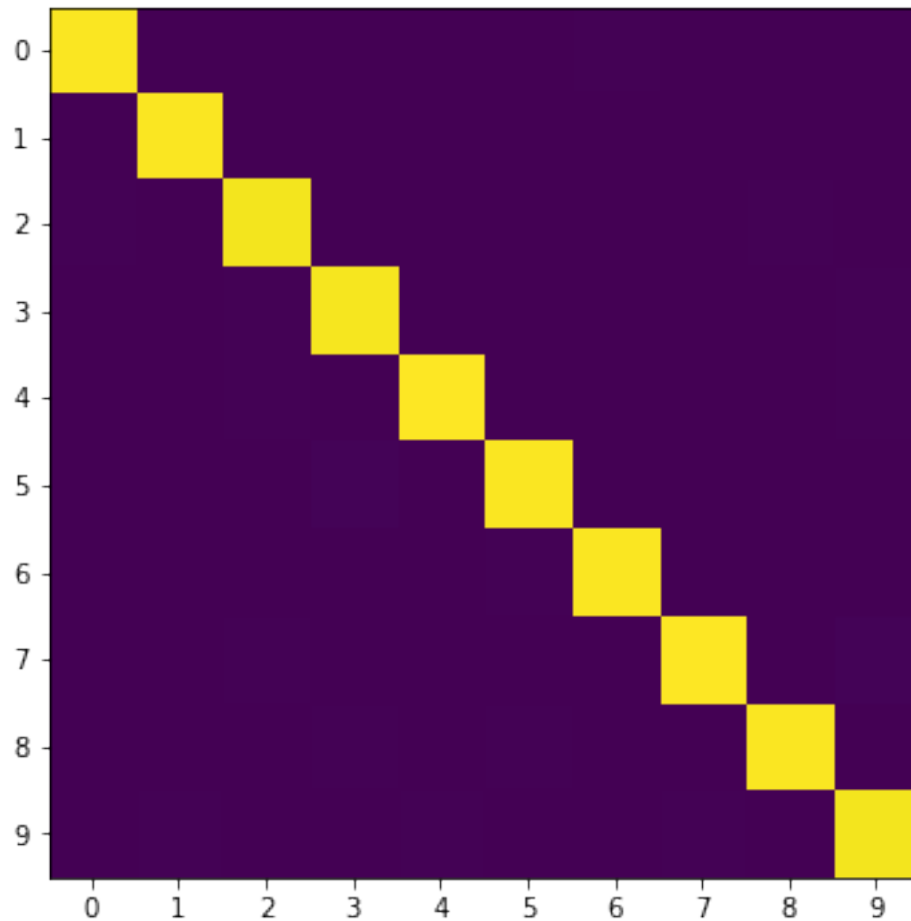
         M = Confusion(testData, testLabels, cnnClassifier)
         VisualizeConfussion(M)

```

```

Epoch:1 Accuracy: 92.960000
Epoch:2 Accuracy: 95.110000
Epoch:3 Accuracy: 96.690000
Epoch:4 Accuracy: 97.050000
Epoch:5 Accuracy: 97.380000
Epoch:6 Accuracy: 97.530000
Epoch:7 Accuracy: 97.500000
Epoch:8 Accuracy: 97.820000
Epoch:9 Accuracy: 98.230000
Epoch:10 Accuracy: 98.280000

```



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

- Note that the MLP/ConvNet approaches lead to an accuracy a little higher than the K-NN approach.
- In general, Neural net approaches lead to significant increase in accuracy, but in this case since the problem is not too hard, the increase in accuracy is not very high.
- However, this is still quite significant considering the fact that the ConvNets we've used are relatively simple while the accuracy achieved using K-NN is with a search over 60,000

training images for every test image.

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