CSE 252A Computer Vision I Fall 2018 - Assignment 4

Instructor: David Kriegman

Assignment Published On: Tuesday, November 27, 2018

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

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.

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.

title

Part 1: Lucas-Kanade implementation [5 pts]

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

In [1]:

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import convolve2d as conv2
from numpy.linalg import inv
from numpy import linalg as LA
```

In [2]:

```python
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 [7]:
```

```
def LucasKanade(im1,im2,w):
    '''
    Inputs: the two graryscale images and window size
    Return U,V
    '''
    U = np.zeros(im1.shape)
    V = np.zeros(im1.shape)
    I_dx = conv2(im1, np.array([[-1,0,1]]),mode = 'same')
    I_dy = conv2(im1, np.array([[-1,0,1]]).T,mode = 'same')
    #I_dy, I_dx = np.gradient(im1)
    I_t = im2 - im1
    hght = im1.shape[0]
    wth = im1.shape[1]
    R = int(w/2)
    M = np.zeros((2,2))
    b = np.zeros((2,1))
    for i in range (R, hght-R):
        for j in range (R, wth-R):
            M[0,0] = np.sum(I_dx[i-R:i-R+w, j-R:j-R+w]**2)
            M[0,1] = (I_dx[i-R:i-R+w, j-R:j-R+w]*I_dy[i-R:i-R+w, j-R:j-R+w]).sum
()
            M[1,0] = M[0,1]
            M[1,1] = (I_dy[i-R:i-R+w, j-R:j-R+w]**2).sum()
            b[0] = -(I_dx[i-R:i-R+w, j-R:j-R+w]*I_t[i-R:i-R+w, j-R:j-R+w]).sum()
            b[1] = -(I_dy[i-R:i-R+w, j-R:j-R+w]*I_t[i-R:i-R+w, j-R:j-R+w]).sum()

            if np.linalg.det(M) != 0 :
                flow = np.dot(inv(M),b)
                U[i,j] = flow[0]
                V[i,j] = flow[1]
            else :
                b =np.matrix(b)
                flow = np.linalg.pinv(M) *b

                U[i,j] = flow[0]
                V[i,j] = flow[1]

    '''
    Your code here
    '''

    return U,V
```

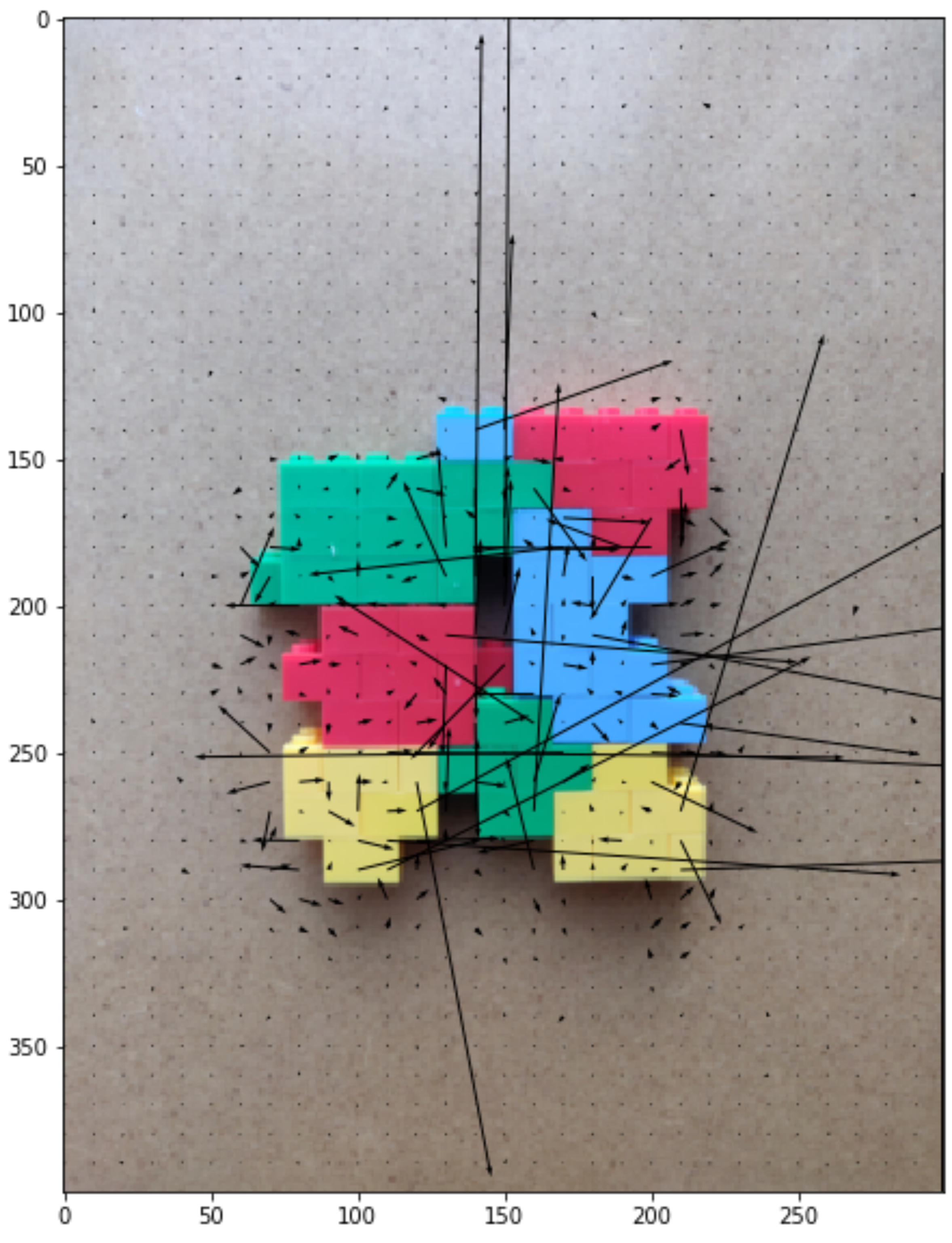
Part 2: Window size [2 pts]

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.

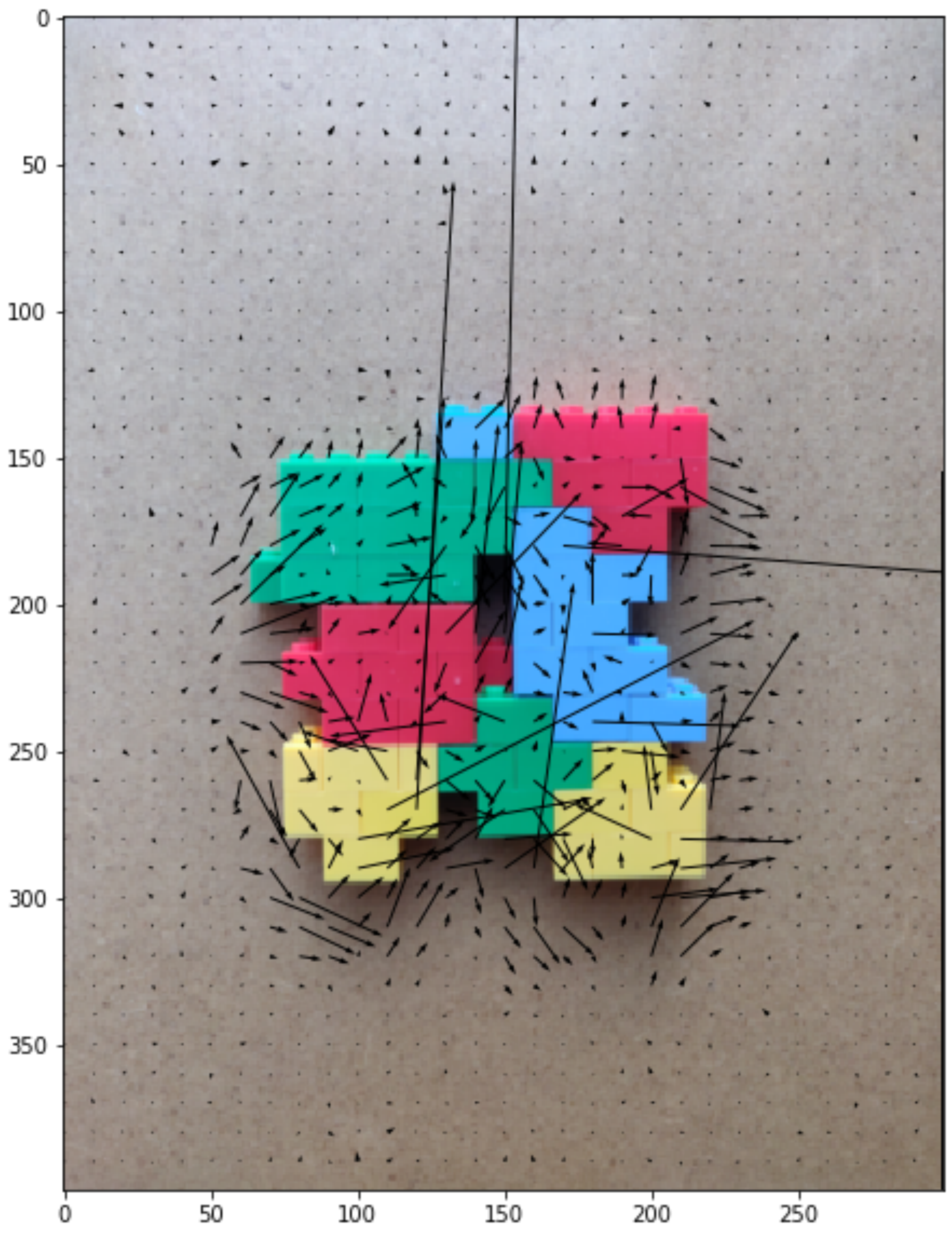
In [6]:

```python
window = np.array([5,15,25])
for i  in np.arange(3):
    plt.figure()
    print('Window Size == %s'%window[i])
    U,V=LucasKanade(grayscale(images[0]),grayscale(images[1]),window[i])
    plot_optical_flow(images[0],U,V)
```

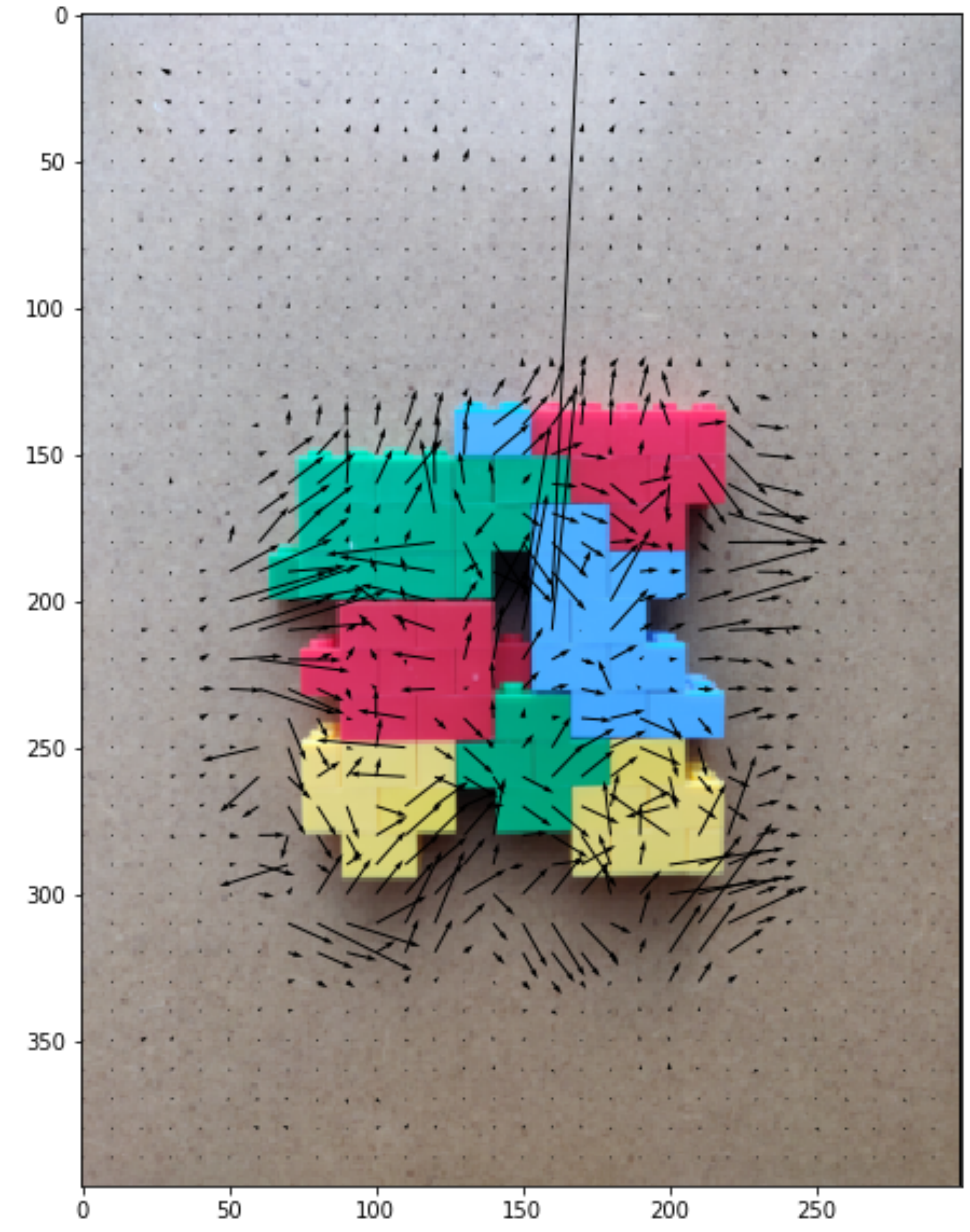
```
Window Size == 5

<matplotlib.figure.Figure at 0x1c14d45b70>
```



```
Window Size == 15

<matplotlib.figure.Figure at 0x1c14d691d0>
```



```
Window Size == 25

<matplotlib.figure.Figure at 0x107c87dd8>
```



Comments

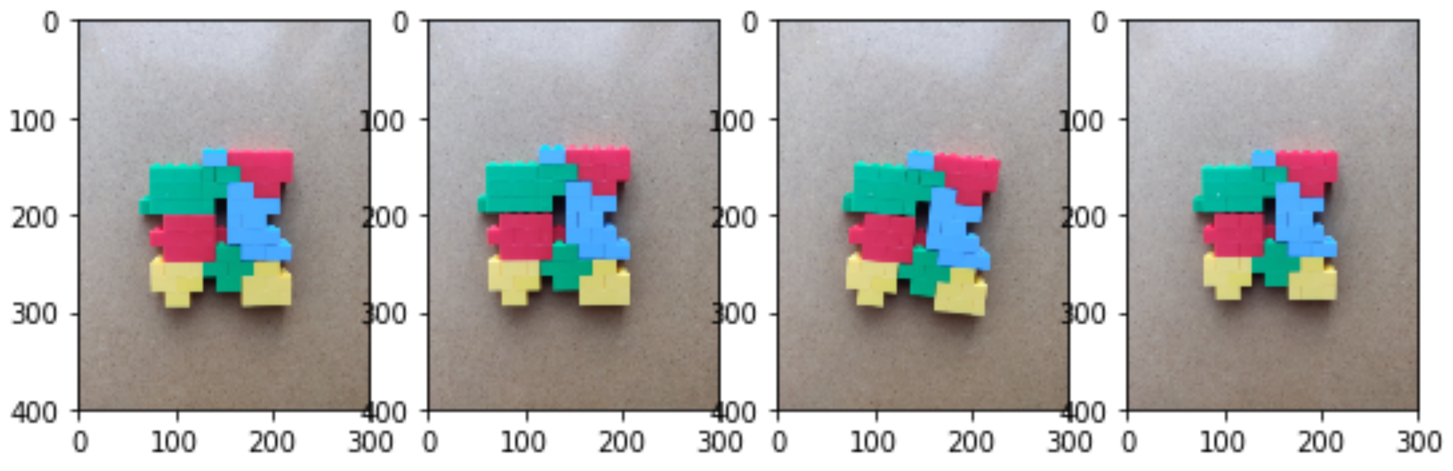
As the results shown, the optical flow has a increasingly clear orientation while increasing size of window from 5 to 25. In this range, the increasing size of window will cancel out the noise.

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 [5]:

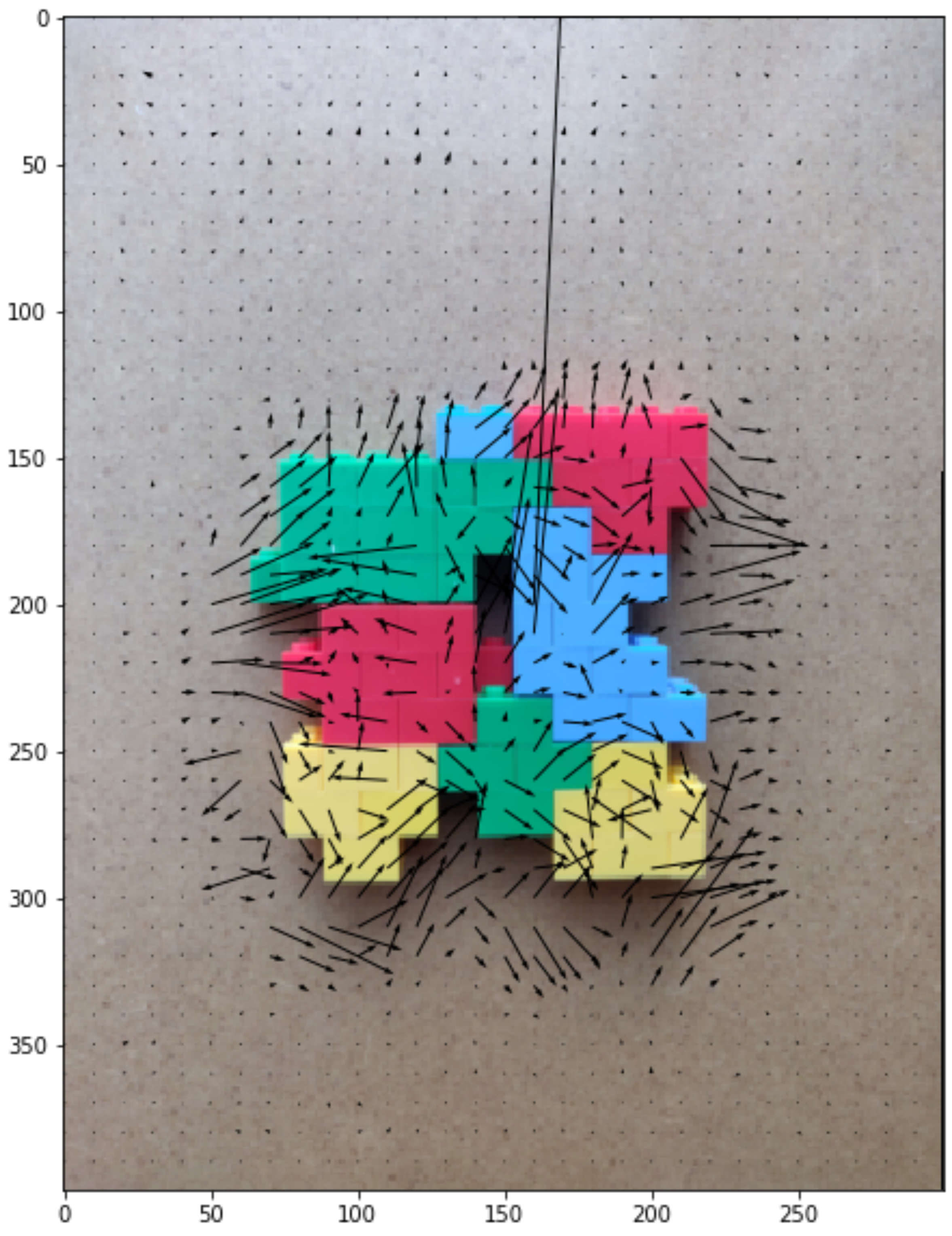
```
test_a = np.arange(4)
fig, axes = plt.subplots(nrows=1, ncols=4,figsize = (9,12))
for i ,ax in zip(test_a,axes):
    ax.imshow(images[i])
```



im1,im2

In [8]:

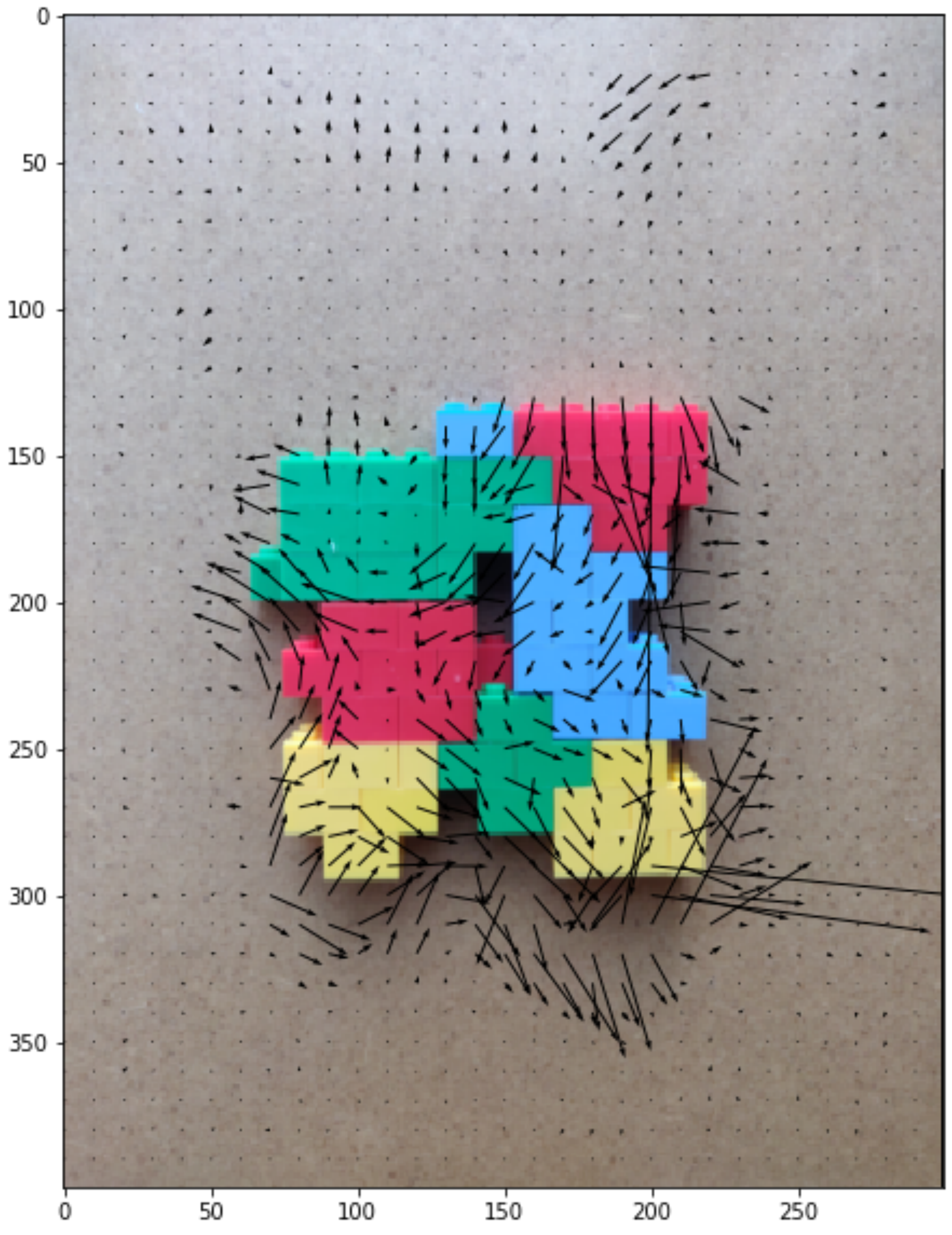
```
window=25
U,V=LucasKanade(grayscale(images[0]),grayscale(images[1]),window)
plot_optical_flow(images[0],U,V)
```



im1,im3

In [8]:

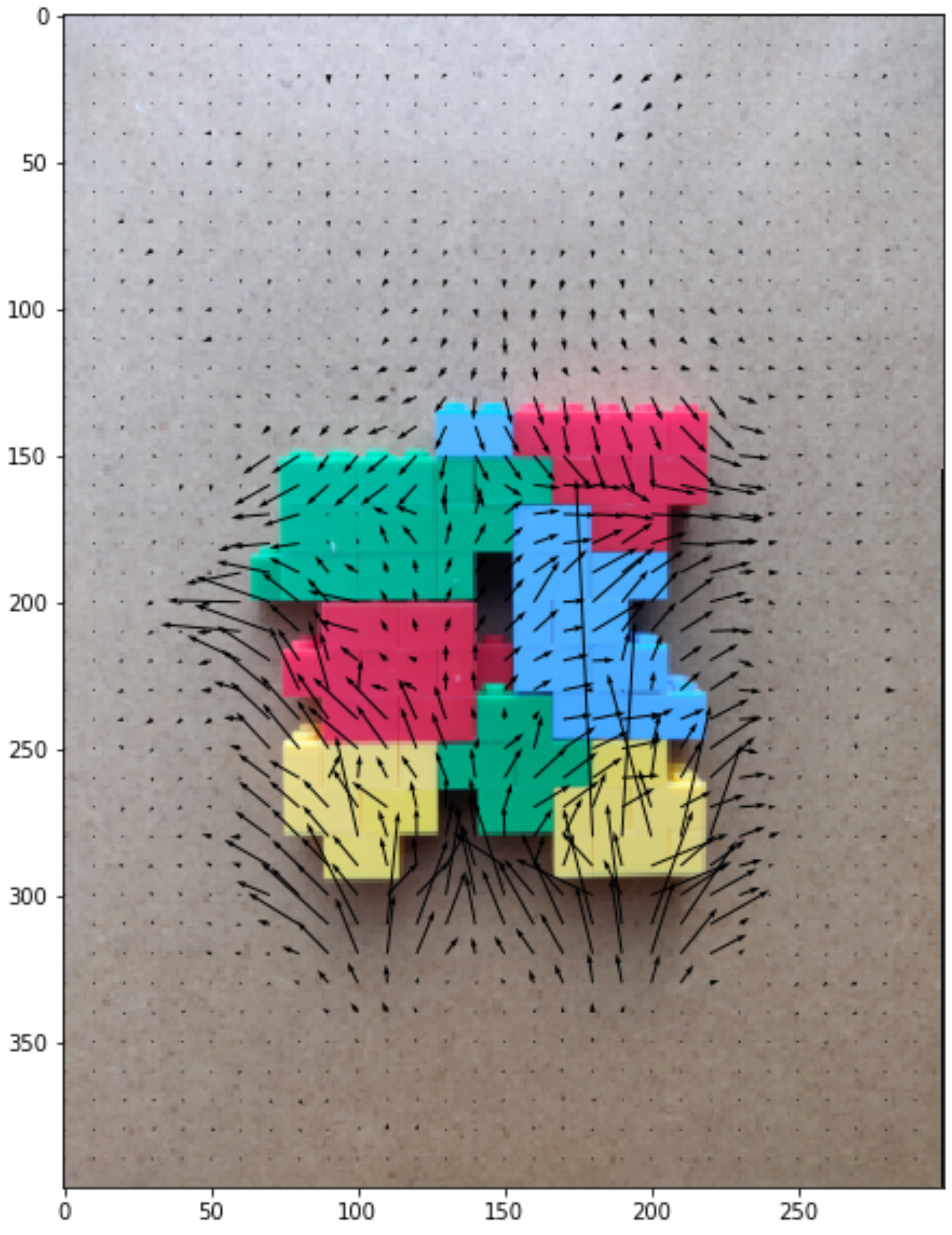
```
window=25
U,V=LucasKanade(grayscale(images[0]),grayscale(images[2]),window)
plot_optical_flow(images[0],U,V)
```



im1,im4

In [9]:

```
window=25
U,V=LucasKanade(grayscale(images[0]),grayscale(images[3]),window)
plot_optical_flow(images[0],U,V)
```



Comments

im1 --> im2 According to the results, most of the arrows are pointing towards right, which indicates the object move toward right. Based on eye observation, the object is moved towards left a little bit.

im1 --> im3 According to the results, Upper arrows around the object tend to go toward left and lower arrows around the object tend to go toward right. It indicates the object rotated counterclockwise. But based on eye observation, the object tend to rotate clockwise

im1 --> im4 According to the results, arrows to object left tends to point toward left while arrows to the right tend to point toward right. So it indicates the object is compressed along y axis and stretched along x axis or be zoomed in somehow. But for the eyes' observation, there isnt such changes in object. So the results are not inconsistent

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://www.tensorflow.org/install/ (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/ (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/ (http://yann.lecun.com/exdb/mnist/).

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

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

In [2]:

```
import time

def TicTocGenerator():
    # Generator that returns time differences
    ti = 0           # initial time
    tf = time.time() # final time
    while True:
        ti = tf
        tf = time.time()
        yield tf-ti # returns the time difference

TicToc = TicTocGenerator() # create an instance of the TicTocGen generator

# This will be the main function through which we define both tic() and toc()
def toc(tempBool=True):
    # Prints the time difference yielded by generator instance TicToc
    tempTimeInterval = next(TicToc)
    if tempBool:
        print( "Elapsed time: %f seconds.\n" %tempTimeInterval )

def tic():
    # Records a time in TicToc, marks the beginning of a time interval
    toc(False)
```

In [3]:

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

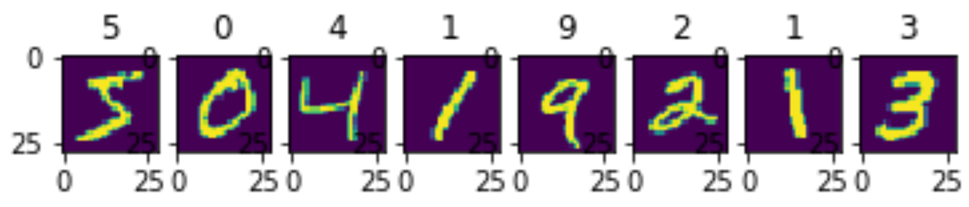
```
/Users/zhangbowen/anaconda3/lib/python3.6/site-packages/h5py/__init_
_.py:36: FutureWarning: Conversion of the second argument of issubdt
ype from `float` to `np.floating` is deprecated. In future, it will
be treated as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters

b'Hello, TensorFlow!'
```

In [4]:

```python
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')))
```

In [105]:

```
test_a = np.arange(8)
fig, axe = plt.subplots(nrows=1, ncols=8)
for i,ax in zip(test_a,axe):
    ax.imshow(trainData[i])
    ax.set_title(trainLabels[i])
```



Some helper functions are given below.

In [5]:

```
# 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))
```

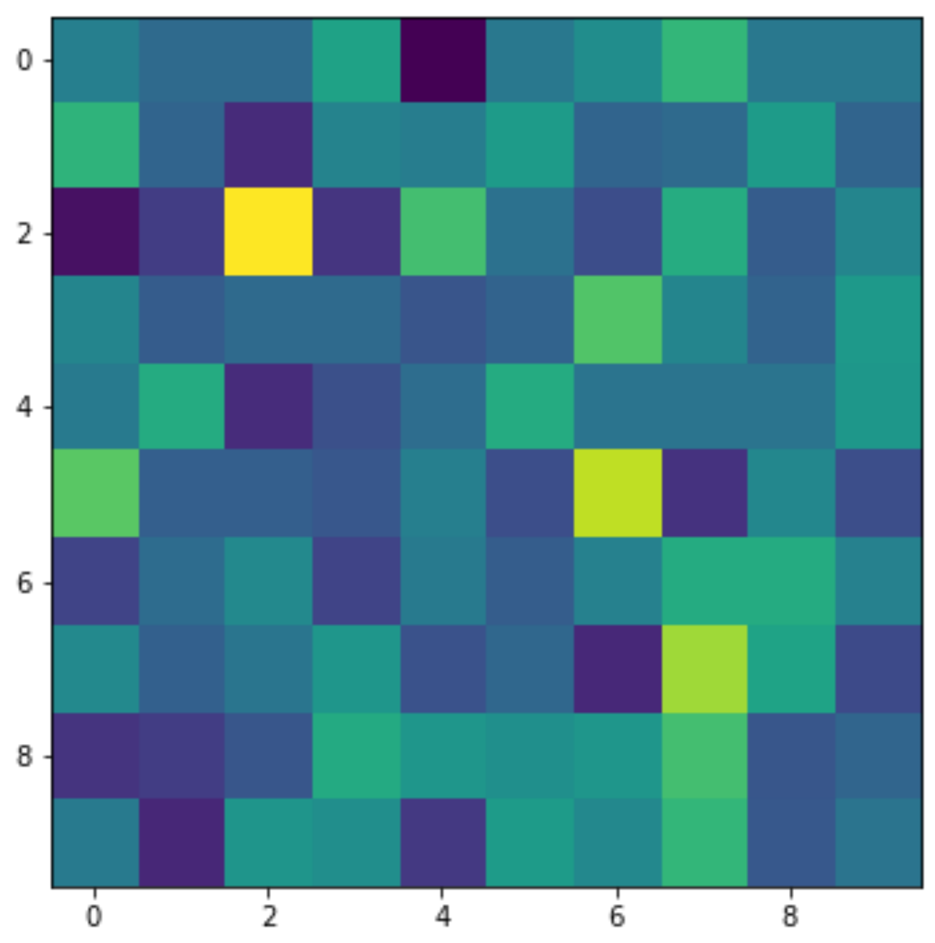
```
Random classifier accuracy: 9.920000
```

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.

In [6]:

```python
# 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):
    '''
    Your code here
    '''

    n = np.max(testLabels)+1
    M = np.zeros((n,n))
    batchsize=50
    correct=0.
    for data,label in DataBatch(testData,testLabels,batchsize):
        prediction = classifier(data)
        M[label,prediction] += 1
    for i in range(n):
        sum_M = sum(M[i])
        for j in range(n):
            M[i,j] = M[i,j]/float(sum_M)
    return M




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

M = Confusion(testData, testLabels, randomClassifier)
VisualizeConfusion(M)
```



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

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 [157]:

```
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
        '''
        trainX = np.reshape(trainData, (trainData.shape[0],-1))
        self.tr = KNeighborsClassifier(self.k, weights='uniform')
        self.tr.fit(trainX, trainLabels)

    def __call__(self, x):
        # this method should take a batch of images
        # and return a batch of predictions
        '''
        your code here
        '''
        testX = np.reshape(x, (x.shape[0],-1))
        self.y = self.tr.predict(testX)
        return self.y

# 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 [16]:

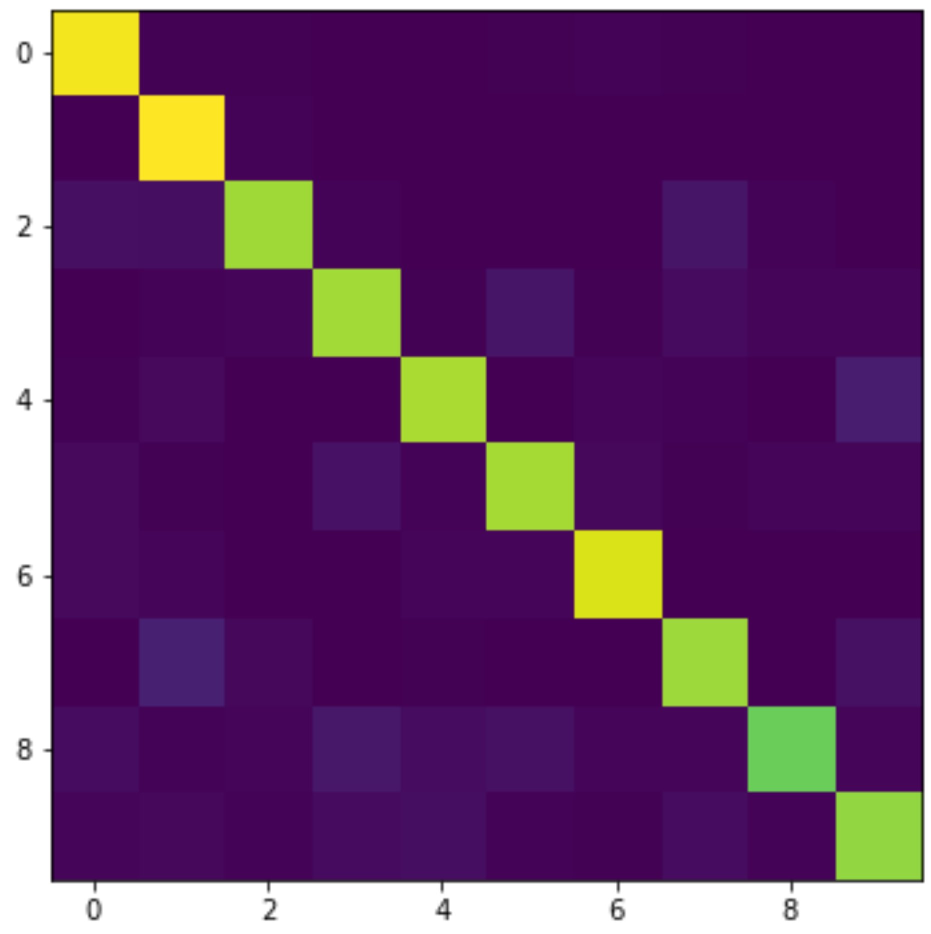
```
# test your classifier with all the training examples (This may take a while)
knnClassifer = KNNClassifer()
knnClassifer.train(trainData, trainLabels)
```

In [17]:

```
# display confusion matrix for your KNN classifier with all the training example
s
M_knn = Confusion(testData, testLabels, knnClassifer)
#print ('KNN classifier accuracy: %f'%test(testData, testLabels, knnClassifer))
```

In [18]:

```
VisualizeConfusion(M_knn)
```



```
[[0.97 0.   0.   0.   0.   0.   0.01 0.   0.   0.  ]
 [0.   0.99 0.01 0.   0.   0.   0.   0.   0.   0.  ]
 [0.04 0.04 0.85 0.01 0.   0.   0.   0.06 0.01 0.  ]
 [0.   0.01 0.02 0.85 0.   0.06 0.   0.03 0.01 0.01]
 [0.   0.03 0.   0.   0.87 0.   0.02 0.01 0.   0.08]
 [0.03 0.   0.   0.05 0.01 0.86 0.02 0.   0.01 0.02]
 [0.02 0.01 0.   0.   0.01 0.01 0.93 0.   0.   0.  ]
 [0.   0.09 0.02 0.   0.   0.   0.   0.85 0.   0.04]
 [0.03 0.01 0.02 0.06 0.03 0.04 0.01 0.02 0.77 0.02]
 [0.02 0.02 0.01 0.03 0.04 0.01 0.   0.03 0.01 0.83]]
```

Comment

Based on the confusion matrix, the number that the number '4' is most often predicted to be, other than '4'is number '6' (0.06)

```
np.round(M_knn,2)[3] == [0.   0.01 0.02 0.85 0.   0.06 0.   0.03 0.01 0.01
]
```

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 [131]:

```
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.cop = components
        self.kth = k




    def train(self, trainData, trainLabels):
        '''
        your code here
        '''
        trainX = np.reshape(trainData, (trainData.shape[0],-1))
        cov_1 = np.cov(trainX.T)
        U, S, V = np.linalg.svd(cov_1)
        self.p_pca = V[:26].T

        trainX = np.matrix(trainX)
        self.p_pca = np.matrix(self.p_pca)
        train_pca = trainX * self.p_pca
        self.tr = KNeighborsClassifier(self.kth, weights='uniform')
        self.tr.fit(train_pca, trainLabels)






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

        testX = np.reshape(x, (x.shape[0],-1))
        testX = np.matrix(testX)
        self.p_pca = np.matrix(self.p_pca)

        test_pca = np.dot(testX, self.p_pca)
        self.y_prd = self.tr.predict(test_pca)
        return self.y_prd







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

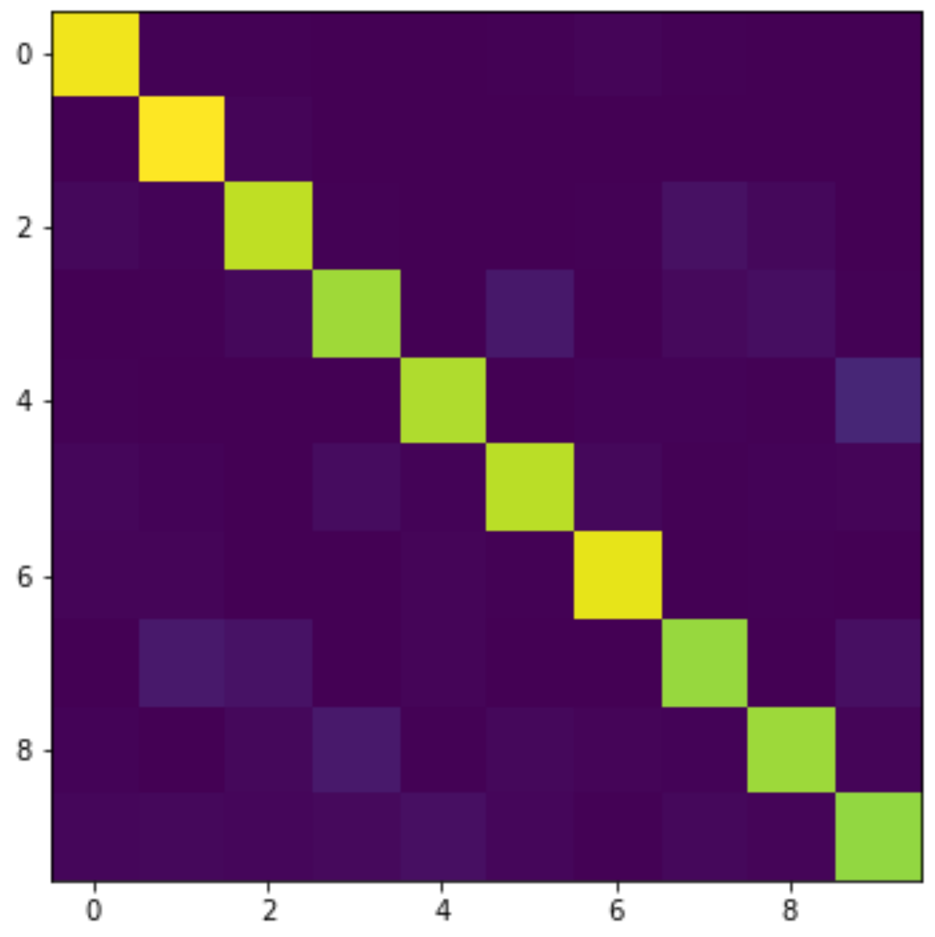
```
PCAKNN classifier accuracy: 65.940000
```

In [28]:

```
# test your classifier with all the training examples (This may take a while)
pcaknnClassifer = PCAKNNClassifer()
pcaknnClassifer.train(trainData, trainLabels)

#print ('PCAKNN classifier accuracy: %f'%test(testData, testLabels, pcaknnClassi
fer))
```

In [29]:

```
# display confusion matrix for your PCA KNN classifier with all the training exa
mples
M_pcaknn = Confusion(testData, testLabels, pcaknnClassifer)
VisualizeConfusion(M_pcaknn)
```



```
[[0.97 0.   0.   0.   0.   0.   0.01 0.   0.   0.  ]
 [0.   0.99 0.01 0.   0.   0.   0.   0.   0.   0.  ]
 [0.02 0.01 0.89 0.   0.   0.   0.   0.04 0.02 0.  ]
 [0.   0.   0.02 0.84 0.   0.06 0.   0.03 0.04 0.  ]
 [0.   0.   0.   0.   0.87 0.   0.01 0.01 0.   0.1 ]
 [0.02 0.01 0.   0.03 0.01 0.88 0.02 0.   0.01 0.01]
 [0.01 0.01 0.   0.   0.01 0.   0.95 0.   0.   0.  ]
 [0.   0.07 0.05 0.   0.01 0.   0.   0.83 0.   0.04]
 [0.01 0.   0.02 0.07 0.   0.02 0.01 0.01 0.84 0.01]
 [0.02 0.02 0.02 0.02 0.04 0.02 0.   0.02 0.01 0.83]]
```

Comment

The running time for PCAKNN classifier is less than KNN classifier. Because by applying PCA, the feature vectors' dimension has been reduced from d-dimension to k- dimension where k < d. So with this dimensionality reduction, the running time has been shortened

Problem 3: Deep learning [12 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 [7]:

```
# base class for your Tensorflow networks. It implements the training loop
# (train) and prediction(__call__)  for you.
```

```python
# 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_w
ith_logits(labels=self.y_, logits=self.y))
        self.train_step = tf.train.AdamOptimizer(1e-4).minimize(self.cross_entro
py)
        self.correct_prediction = tf.equal(self.prediction, self.y_)
        self.accuracy = tf.reduce_mean(tf.cast(self.correct_prediction, tf.float
32))
        self.sess.run(tf.global_variables_initializer())

        for epoch in range(epochs):
            for i, (data,label) in enumerate(DataBatch(trainData, trainLabels, b
atchsize, shuffle=True)):
                data=np.expand_dims(data,-1)
                _, acc = self.sess.run([self.train_step, self.accuracy], feed_di
ct={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]) + se
lf.biases[0]
```

In [9]:

```
linearClassifier = LinearClassifier()
```

In [11]:

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

    weight_slp = linearClassifier.weights[0]
    weight_slp = tf.reshape(weight, (28, 28, 10)).eval()
```

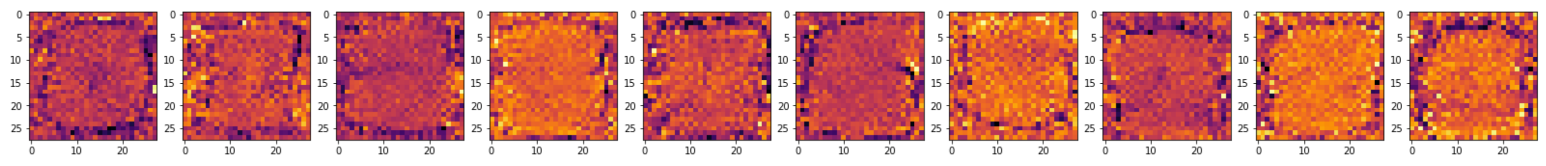
```
Epoch:1 Accuracy: 88.690000
Epoch:2 Accuracy: 89.230000
Epoch:3 Accuracy: 89.460000
Epoch:4 Accuracy: 88.700000
Epoch:5 Accuracy: 89.470000
Epoch:6 Accuracy: 89.990000
Epoch:7 Accuracy: 90.310000
Epoch:8 Accuracy: 88.730000
Epoch:9 Accuracy: 90.820000
Epoch:10 Accuracy: 89.500000
```

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 [19]:

```
fig,axes = plt.subplots(1,10,figsize=(28,28))
for i, ax in enumerate (axes):
    ax.imshow(weight_slp[:,:,i],cmap="inferno")
```



Those weights look like numbers. The reason probably is that the numbers in the data has a high contrast with the background. So it is easy to extract the boundary as features.

Part 2: Multi Layer Perceptron (MLP) [5 pts]

Here you will implement an MLP. The MLP shoud consist of 2 layers (matrix multiplication and bias offset) that map to the following feature dimensions:

- 28x28 -> hidden (100)
- hidden -> classes
- The hidden layer should be followed with a ReLU nonlinearity. The final layer should not have a nonlinearity applied as we desire the raw logits output.
- The final output of the computation graph should be stored in self.y as that will be used in the training.

Display the confusion matrix and accuracy after training. Note: You should get ~ 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 [8]:

```
class MLPClassifer(TFClassifier):
    def __init__(self, classes=10, hidden=100):
        '''
        your code here
        '''
        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.W1 = weight_variable([28*28*1, hidden])
        self.W2 = weight_variable([hidden, hidden])
        self.W3 = weight_variable([hidden, classes])
        b1 = bias_variable([hidden])
        b2 = bias_variable([hidden])
        b3 = bias_variable([classes])


        x_re = tf.reshape(self.x,(-1,28*28*1))

        l1 = tf.add(tf.matmul(x_re,self.W1), b1)
        l1 = tf.nn.relu(l1)

        l2 = tf.add(tf.matmul(l1,self.W2), b2)
        l2 = tf.nn.relu(l2)


        self.y = tf.matmul(l2,self.W3) + b3
```

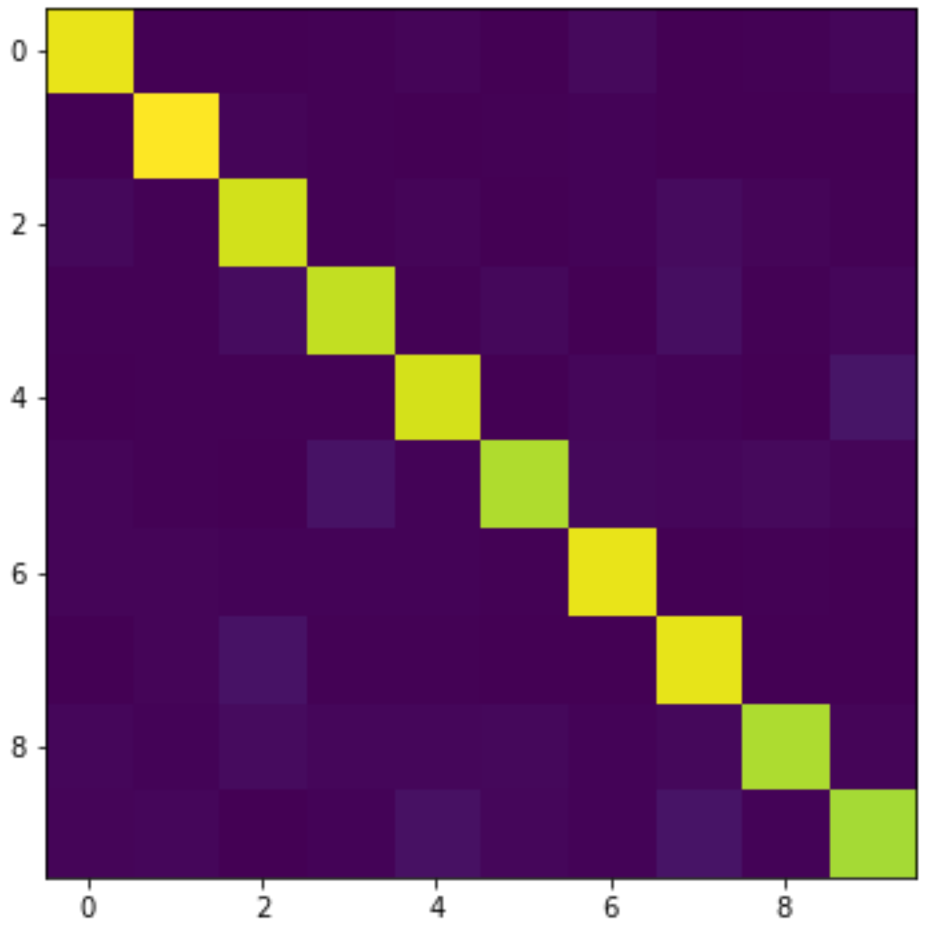
In [13]:

```
mlpClassifer = MLPClassifer()
mlpClassifer.train(trainData, trainLabels, epochs=10)
with mlpClassifer.sess as sess:
    weight_mlp = mlpClassifer.W1
    weight_mlp = tf.reshape(weight, (28, 28, 100)).eval()
```

```
Epoch:1 Accuracy: 94.660000
Epoch:2 Accuracy: 96.010000
Epoch:3 Accuracy: 96.700000
Epoch:4 Accuracy: 97.070000
Epoch:5 Accuracy: 97.200000
Epoch:6 Accuracy: 97.430000
Epoch:7 Accuracy: 97.230000
Epoch:8 Accuracy: 97.510000
Epoch:9 Accuracy: 97.420000
Epoch:10 Accuracy: 97.640000
```

In [124]:

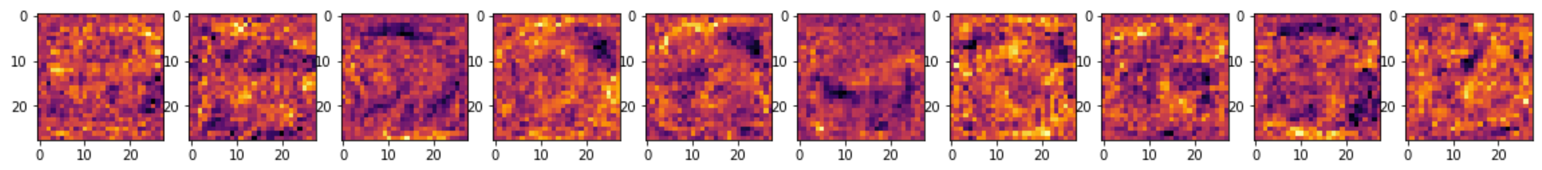
```
M = Confusion(testData, testLabels,  mlpClassifer)
VisualizeConfusion(M)
```



```
[[0.93 0.   0.   0.   0.01 0.   0.02 0.   0.   0.02]
 [0.   0.97 0.01 0.   0.   0.   0.01 0.   0.   0.  ]
 [0.02 0.   0.9  0.   0.01 0.   0.01 0.03 0.01 0.  ]
 [0.   0.   0.03 0.88 0.   0.02 0.   0.04 0.   0.02]
 [0.   0.   0.   0.   0.9  0.   0.02 0.01 0.   0.05]
 [0.01 0.   0.   0.05 0.01 0.85 0.02 0.02 0.03 0.01]
 [0.01 0.01 0.01 0.01 0.01 0.   0.93 0.   0.   0.  ]
 [0.   0.01 0.05 0.   0.   0.   0.   0.93 0.   0.  ]
 [0.02 0.01 0.03 0.02 0.02 0.02 0.01 0.02 0.85 0.01]
 [0.01 0.02 0.   0.01 0.04 0.02 0.01 0.05 0.01 0.84]]
```

In [15]:

```
fig,axes = plt.subplots(1,10,figsize=(20,20))
for i, ax in enumerate (axes):
    ax.imshow(weight_mlp[:,:,i],cmap="inferno")
```



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

Here you will implement a CNN with the following architecture:

- n=5
- ReLU(Conv(kernel_size=4x4, stride=2, output_features=n))
- ReLU(Conv(kernel_size=4x4, stride=2, output_features=n*2))
- ReLU(Conv(kernel_size=4x4, stride=2, output_features=n*4))
- Linear(output_features=classes)

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

In [24]:

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

class CNNClassifer(TFClassifier):
    def __init__(self, classes=10, n=5):

        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
        #'''
        W1 = weight_variable([4,4,1,n])
        W2 = weight_variable([4,4,n,n*2])
        W3 = weight_variable([4,4,n*2,n*4])
        Wfc = weight_variable([4*4*n*4,n*4])
        #Wout = weight_variable([n*4, classes])

        b1 = bias_variable([n])
        b2 = bias_variable([n*2])
        b3 = bias_variable([n*4])
        bfc = bias_variable([n*4])
        #bout = bias_variable([classes])

        conv1 = tf.nn.relu(conv2d(self.x, W1) + b1)
        conv2 = tf.nn.relu(conv2d(conv1, W2) + b2)
        conv3 = tf.nn.relu(conv2d(conv2, W3) + b3)


        fc = tf.reshape(conv3,[-1,4*4*n*4])

        self.y = tf.nn.relu(tf.matmul(fc,Wfc)+bfc)
```

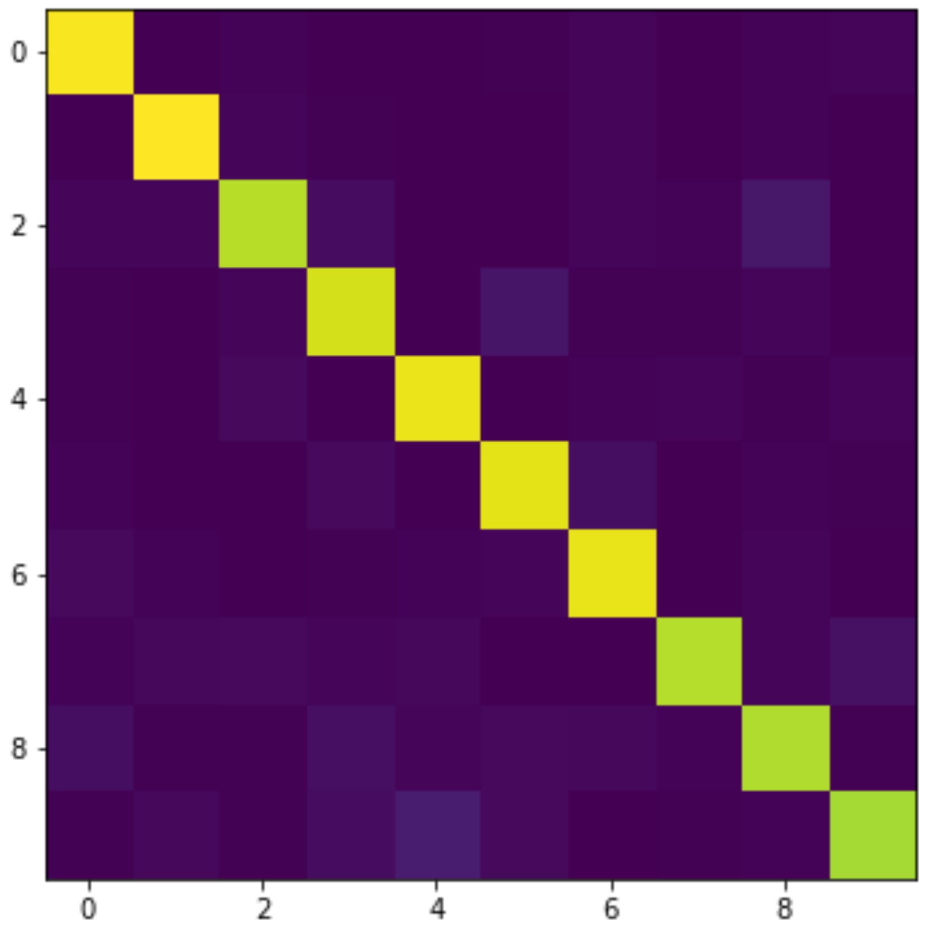
In [26]:

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

```
Epoch:1 Accuracy: 90.240000
Epoch:2 Accuracy: 92.020000
Epoch:3 Accuracy: 94.370000
Epoch:4 Accuracy: 95.060000
Epoch:5 Accuracy: 95.490000
Epoch:6 Accuracy: 96.500000
Epoch:7 Accuracy: 96.880000
Epoch:8 Accuracy: 97.280000
Epoch:9 Accuracy: 97.340000
Epoch:10 Accuracy: 97.580000
```

In [27]:

```
M_cnn = Confusion(testData, testLabels,  cnnClassifer)
VisualizeConfusion(M_cnn)
```



```
[[0.95 0.   0.01 0.   0.   0.   0.01 0.   0.01 0.01]
 [0.   0.96 0.01 0.   0.   0.   0.01 0.   0.01 0.  ]
 [0.02 0.02 0.85 0.03 0.   0.   0.01 0.01 0.06 0.  ]
 [0.   0.   0.02 0.89 0.   0.05 0.   0.   0.02 0.  ]
 [0.   0.   0.02 0.   0.93 0.   0.01 0.01 0.   0.01]
 [0.01 0.   0.   0.02 0.   0.92 0.04 0.   0.01 0.  ]
 [0.02 0.01 0.   0.   0.01 0.01 0.93 0.   0.01 0.  ]
 [0.01 0.02 0.03 0.02 0.02 0.   0.   0.85 0.01 0.04]
 [0.03 0.   0.   0.04 0.02 0.03 0.02 0.01 0.84 0.  ]
 [0.   0.02 0.   0.03 0.07 0.03 0.   0.   0.01 0.83]]
```

- 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/ (http://yann.lecun.com/exdb/mnist/)
- You can learn more about neural nets/ tensorflow at https://www.tensorflow.org/tutorials/ (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/ (https://playground.tensorflow.org/)