CSE 252A Computer Vision I Fall 2019 - Homework 5

Instructor: Ben Ochoa

Assignment Published On: Thursday, November 21, 2019

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

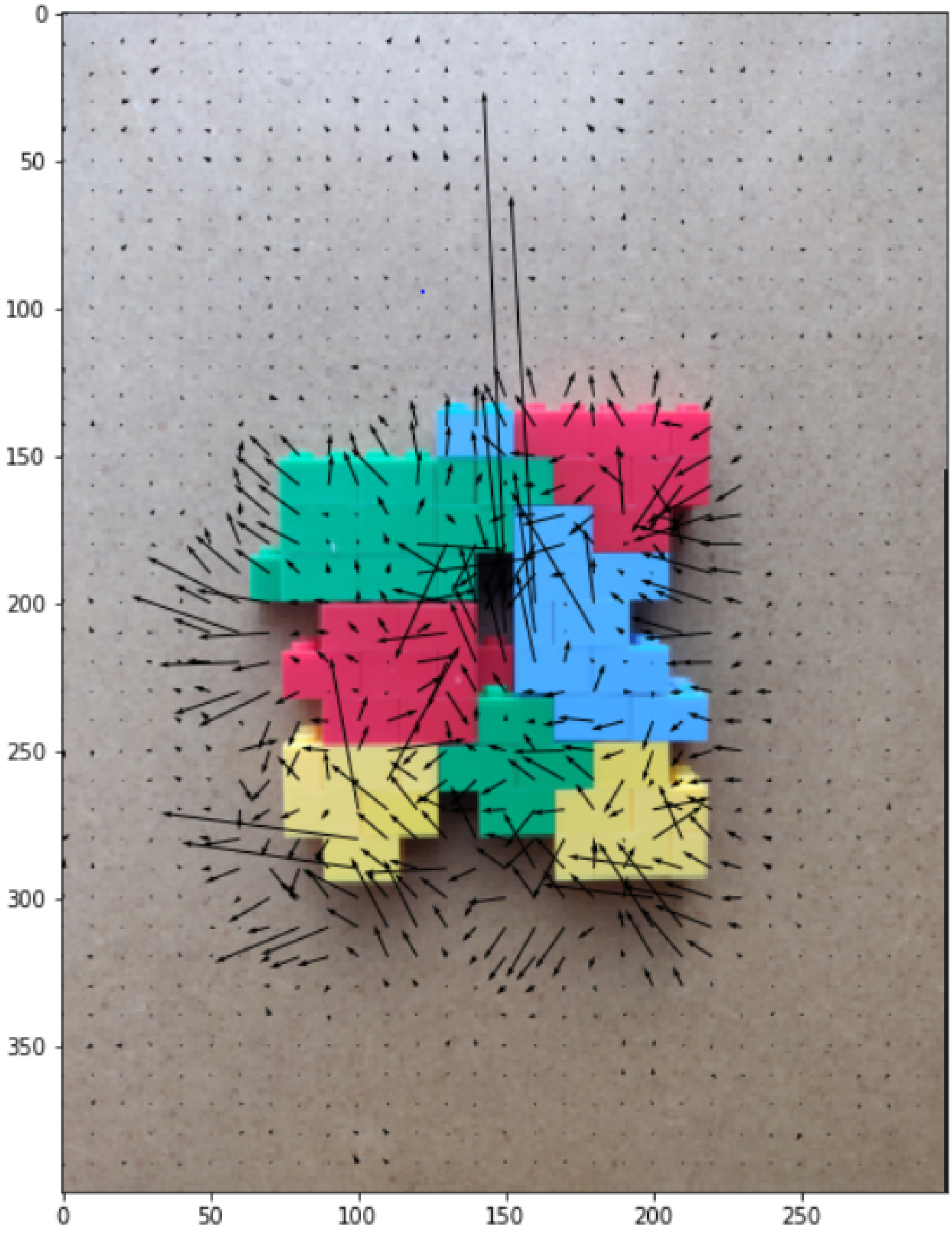
Instructions

- Review the academic integrity and collaboration policies on the course website.
  - This assignment must be completed individually.
- All solutions must be written in this notebook.
  - Programming aspects of the assignment must be completed using Python in this notebook.
- If you want to modify the skeleton code, you may do so. It has only been provided as a framework for your solution.
- You may use Python packages (such as NumPy and SciPy) for basic linear algebra, but you may not use packages that directly solve the problem.
  - If you are unsure about using a specific package or function, then ask the instructor and/or teaching assistants for clarification.
- You must submit this notebook exported as a PDF. You must also submit this notebook as `.ipynb` file.
  - Submit both files (`.pdf` and `.ipynb`) on Gradescope.
  - **You must mark the PDF pages associated with each question in Gradescope. If you fail to do so, we may dock points.**
- It is highly recommended that you begin working on this assignment early.
- **Late policy: assignments submitted late will receive a 15% grade reduction for each 12 hours late (i.e., 30% per day). Assignments will not be accepted 72 hours after the due date. If you require an extension (for personal reasons only) to a due date, you must request one as far in advance as possible. Extensions requested close to or after the due date will only be granted for clear emergencies or clearly unforeseeable circumstances.**

Problem 1: Optical Flow [14 pts]

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

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



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

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

In [23]:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import interpolate
from scipy.signal import convolve
from scipy.ndimage import gaussian_filter,correlate
import math
from tqdm import tqdm_notebook

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

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

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

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

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

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


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

In [230]:

```python
# you can use interpolate from scipy
# You can implement 'upsample_flow' and 'OpticalFlowRefine'
# as 2 building blocks in order to complete this.
def upsample_flow(u_prev, v_prev):
    ''' You may implement this method to upsample optical flow from
    previous level
    u_prev, v_prev -> optical flow from prev level
    u, v -> upsampled optical flow to the current level
    '''
    """ ==========
    YOUR CODE HERE
    ========== """
    x = np.arange(u_prev.shape[0])*2
    y = np.arange(v_prev.shape[1])*2
    u_interp = interpolate.interp2d(x,y,u_prev.T,kind='linear')
    v_interp = interpolate.interp2d(x,y,v_prev.T,kind='linear')
    x_new = np.arange(u_prev.shape[0]*2)
    y_new = np.arange(u_prev.shape[1]*2)
    u = u_interp(x_new,y_new).T
    v = v_interp(x_new,y_new).T

    return u, v

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

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

    """ ==========
    YOUR CODE HERE
    ========== """
    Iy,Ix = np.gradient(im1)
    Iy = -Iy
    Ix2 = Ix*Ix
    Iy2 = Iy*Iy
    Ixy = Ix*Iy
    radi = window//2
    if u_prev.all() == None:
        u_prev = np.zeros_like(im1)
        v_prev = np.zeros_like(im1)
    else:
        u_prev,v_prev = upsample_flow(u_prev,v_prev)

    for row in range(radi,im1.shape[0]-radi):
        for col in range(radi,im2.shape[1]-radi):
            d_x = int(np.round(u_prev[row,col]))
            d_y = int(np.round(v_prev[row,col]))
            if (radi<=col+d_x) and (col+d_x<im1.shape[1]-radi) and \
                (radi<=row+d_y) and (row+d_y<im1.shape[0]-radi):
                im1_window = im1[row-radi:row+radi+1,col-radi:col+radi+1]
                im2_window = im2[row-radi+d_y:row+radi+d_y+1,col-radi+d_x:col+radi+d_x+1]
                It_window = im2_window-im1_window

                Ix_window = Ix[row-radi:row+radi+1,col-radi:col+radi+1]
                Iy_window = Iy[row-radi:row+radi+1,col-radi:col+radi+1]

                Ix2_window = Ix2[row-radi:row+radi+1,col-radi:col+radi+1].sum()
                Ixy_window = Ixy[row-radi:row+radi+1,col-radi:col+radi+1].sum()
                Iy2_window = Iy2[row-radi:row+radi+1,col-radi:col+radi+1].sum()
                Ixt = (Ix_window*It_window).sum()
```

```python
            Iyt = (Iy_window*It_window).sum()

            M = np.array([[Ix2_window,Ixy_window],[Ixy_window,Iy2_window]])
            b = np.array([[-Ixt,-Iyt]])
            uv_vec = np.dot(np.linalg.pinv(M),b.T)
            u[row,col] = uv_vec[0]
            v[row,col] = uv_vec[1]

    u = u+u_prev
    v = v+v_prev
    return u, v
```

In [231]:

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

    """ ==========
    YOUR CODE HERE
    ========== """

    # You can call OpticalFlowRefine iteratively

    img1_layer = []
    img2_layer = []
    img1_layer.append(im1)
    img2_layer.append(im2)
    for i in range(1,numLevels):
        img1_layer.append(gaussian_filter(im1[::2**i,::2**i],sigma = 1))
        img2_layer.append(gaussian_filter(im2[::2**i,::2**i],sigma = 1))

    u_prev=np.array([None])
    v_prev=np.array([None])
    for i in range(numLevels-1,-1,-1):
        u,v = OpticalFlowRefine(img1_layer[i],img2_layer[i],window,u_prev,v_prev)
        u_prev = u
        v_prev = v
    return u, v
```

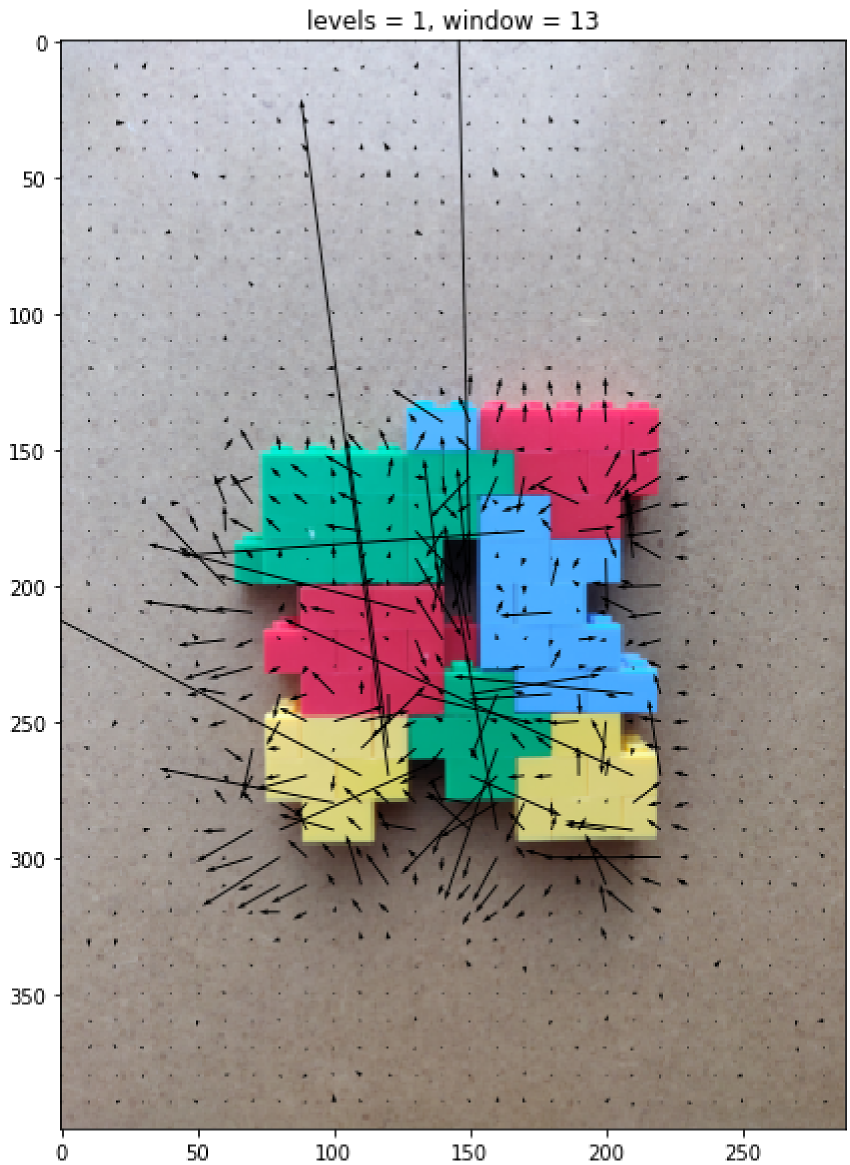
Part 2: Number of levels [2 pts]

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

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

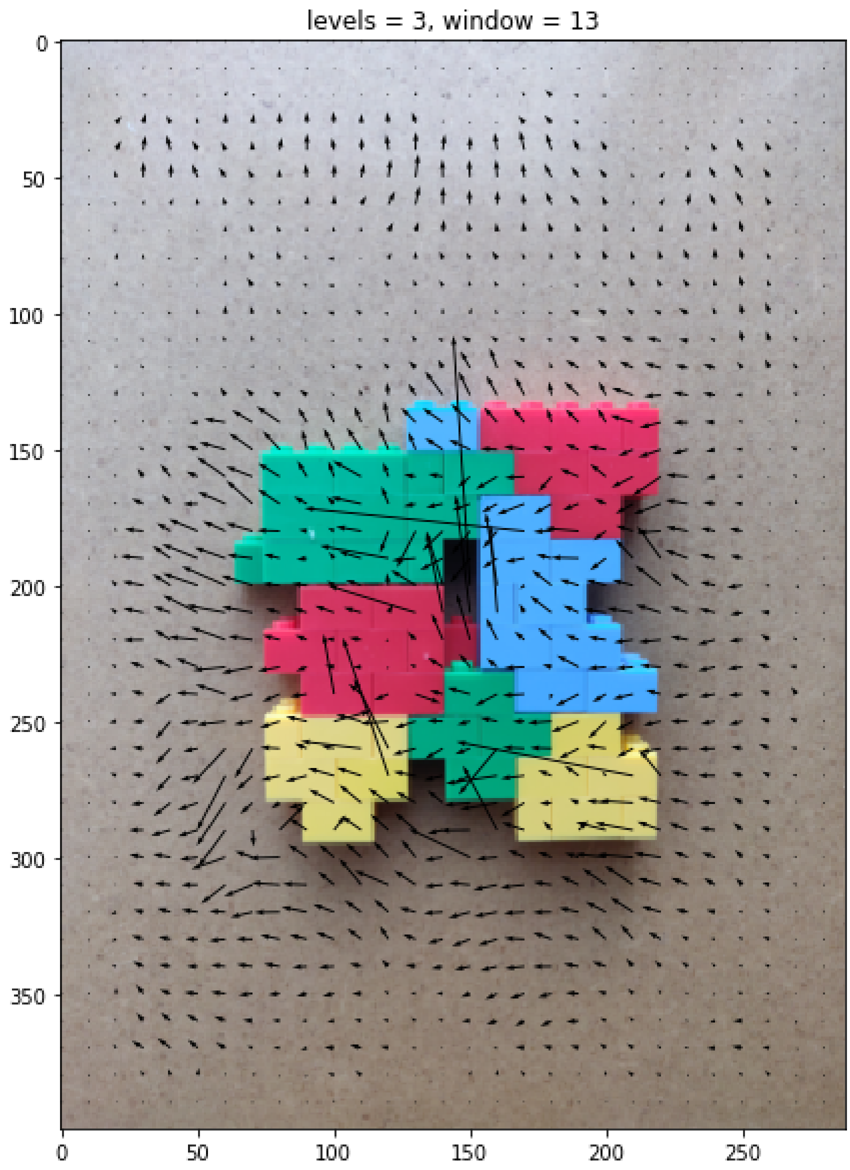
In [232]:

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

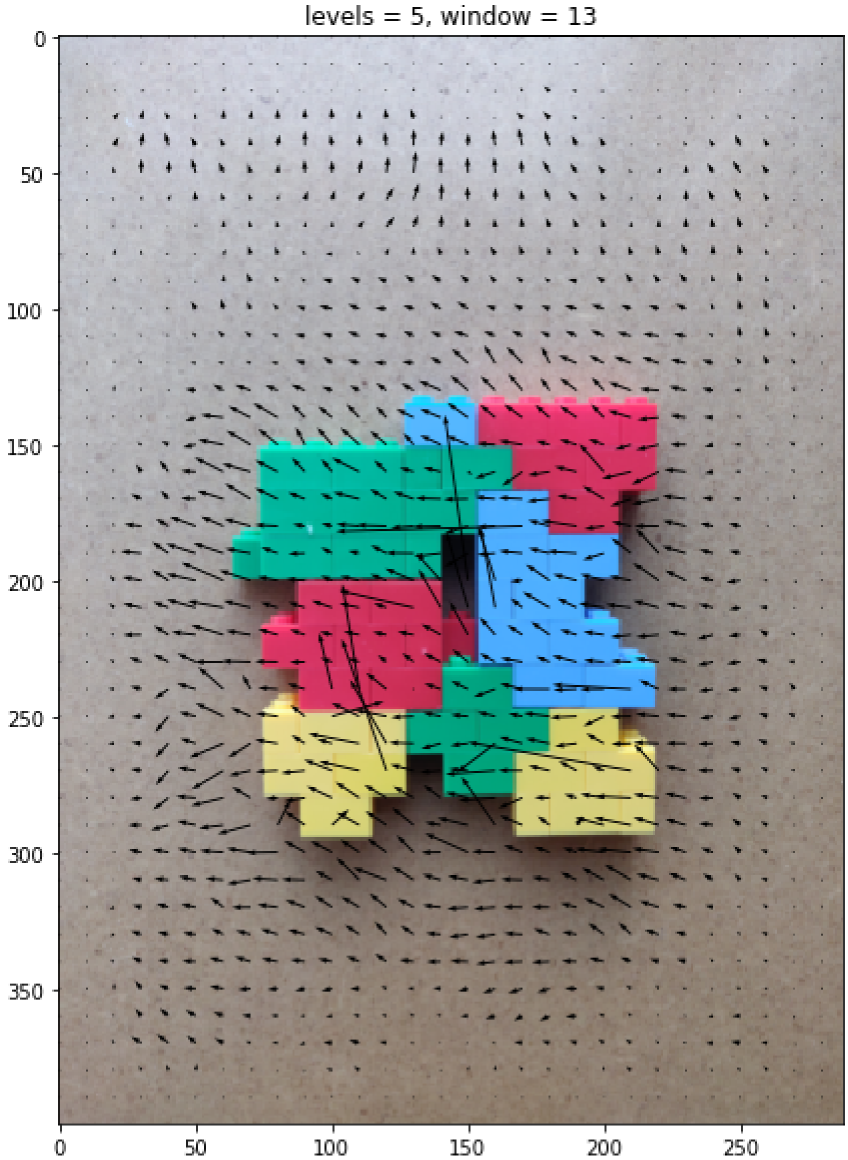


In [233]:

```
window=13
numLevels=3
# Plot
U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[1]),\
                          window,numLevels)
plot_optical_flow(images[0],U,V, \
                  'levels = ' + str(numLevels) + ', window = '+str(window))
```



In [234]:
```
window=13
numLevels=5
# Plot
U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[1]),\
                          window,numLevels)
plot_optical_flow(images[0],U,V, \
                  'levels = ' + str(numLevels) + ', window = '+str(window))
```



Your Comments on the results of Part 2:

By implementing multi-scale-LK method with the same windowSize and different number of sample levels, it can be noticed that the optical flow of the image tends to be more "uniformed".

Within a reasonable range of downsize scale of image, the LK performs better with more times that the u,v is downsampled.

The transformation from image1 and image2 is a simple horizontally movement, the positional relations between each toy brick are remained.

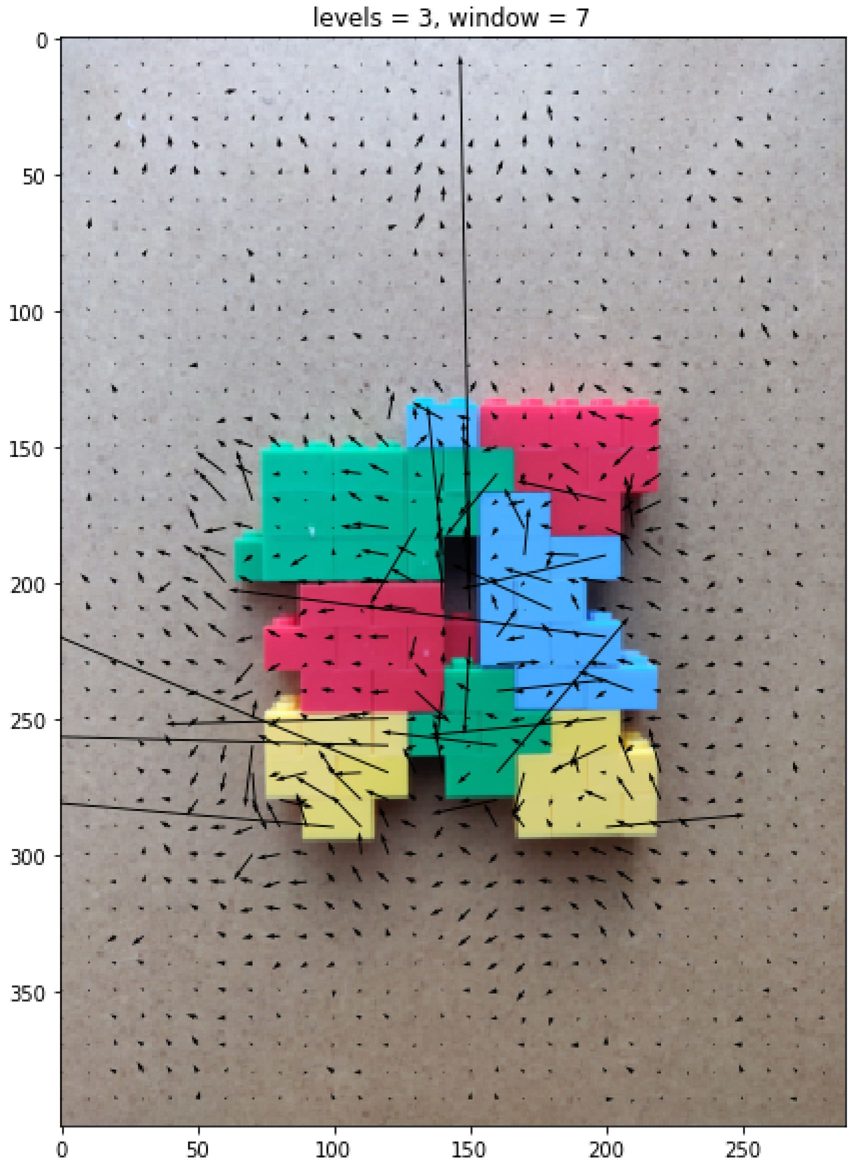
Idealy, the optical flow of image1 to image2 should be presented by a set of parallel arrows. Due to the fact that the refining procedure may be disturbed because of the changing of light condition and other noises on these images, the flow is not as perfect as it should be.

Part 3: Window size [3 pts]

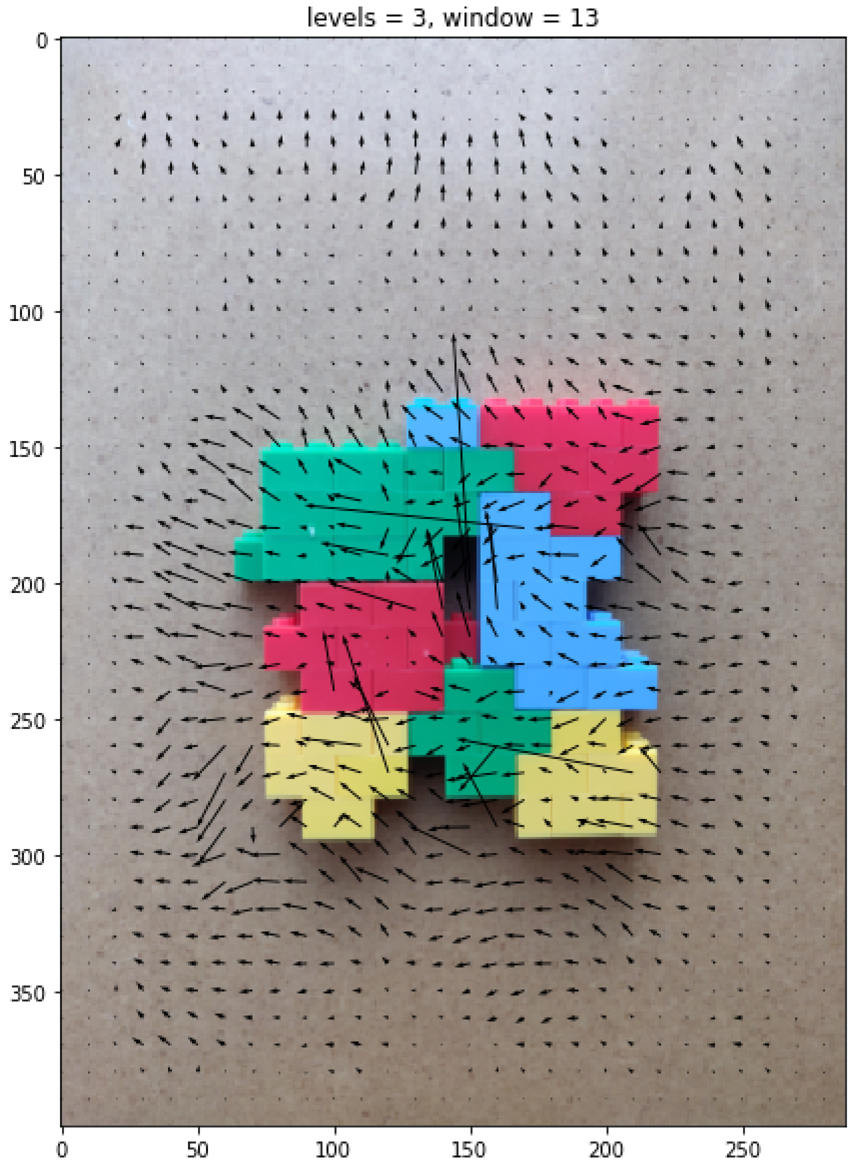
Plot optical flow for the pair of images im1 and im2 for at least 3 different window sizes which leads to observable difference in the results. Comment on the effect of window size on results and justify. For this part fix the number of levels to be 3.

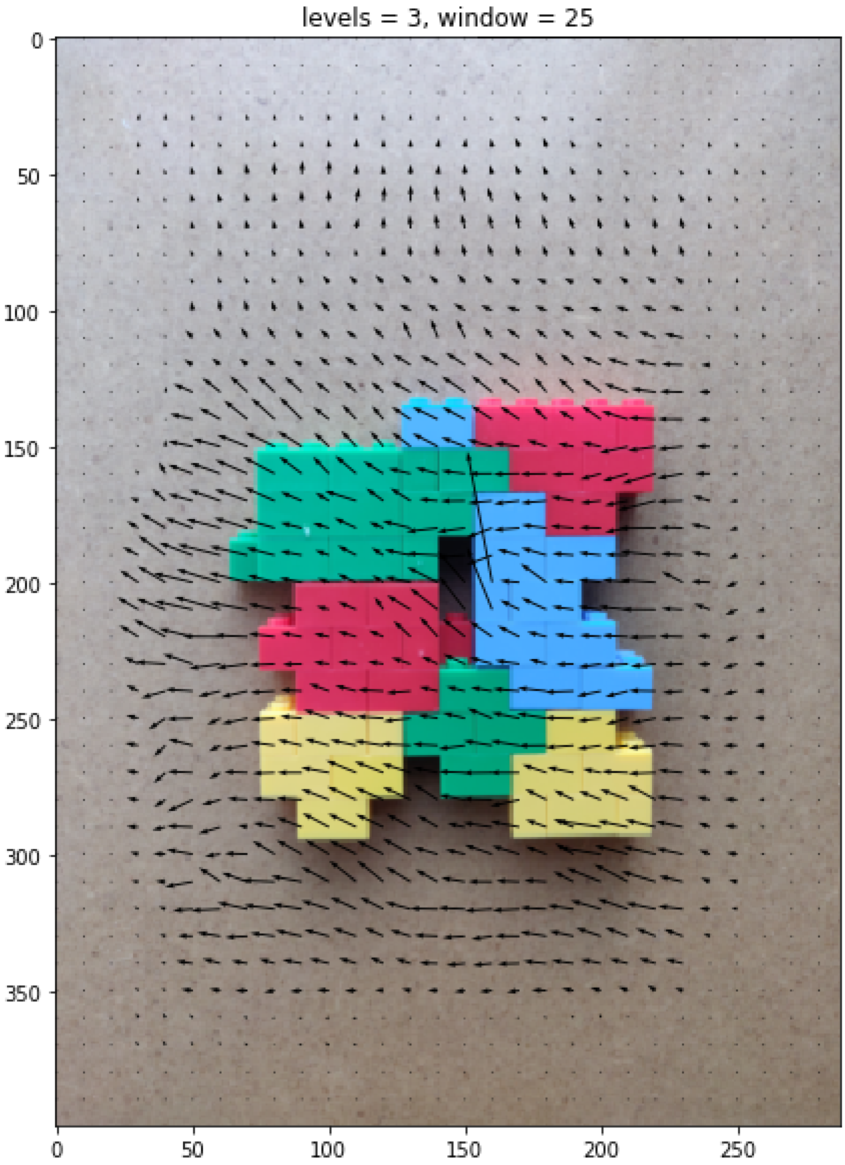
In [235]:

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

w1, w2, w3 = 7, 13, 25
for window in [w1, w2, w3]:
    U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[1]),\
                              window,numLevels)
    plot_optical_flow(images[0],U,V, \
                      'levels = ' + str(numLevels) + ', window = '+str(window))
```



levels = 3, window = 7
0
50
100
150
200
250
300
350
0
50
100
150
200
250


levels = 3, window = 13
0
50
100
150
200
250
300
350
0
50
100
150
200
250



Your Comments on the results of Part 3

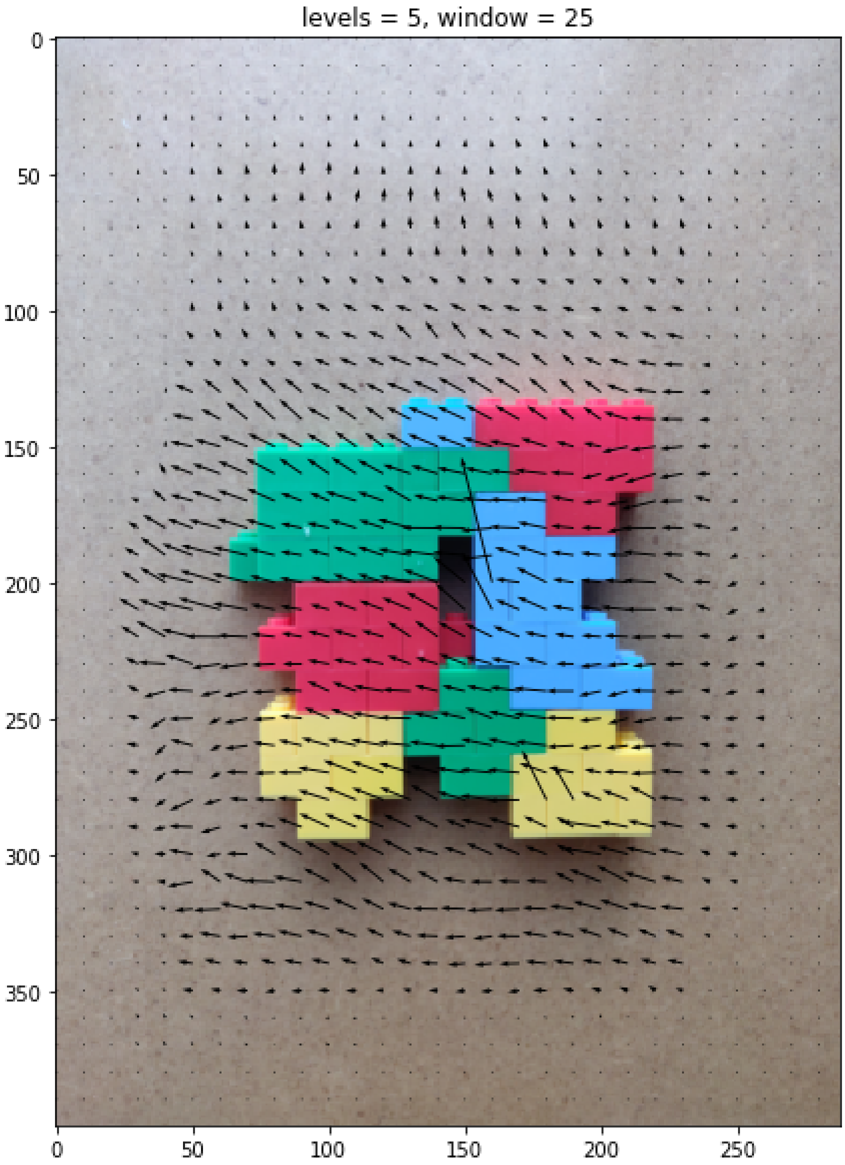
It can be concluded that, within a relatively reasonable range, the larger size of window performs better. Since I_x, I_y, I_t of each window contains more information of the movement of image, and the disturbance from the noise in each window is weaken because of the larger size, the calculation should be more accurate intuitively. Also, make the optical flow, visually, more uniform.

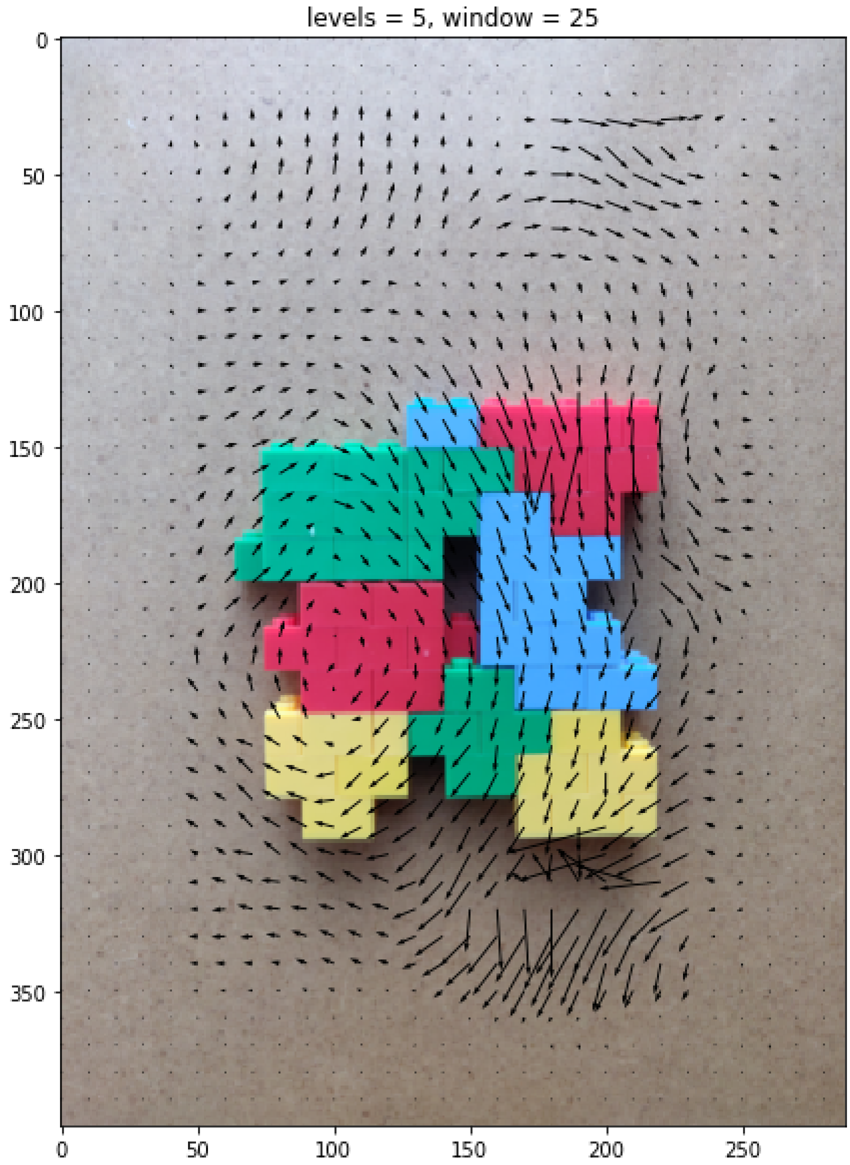
Part 4: All pairs [3 pts]

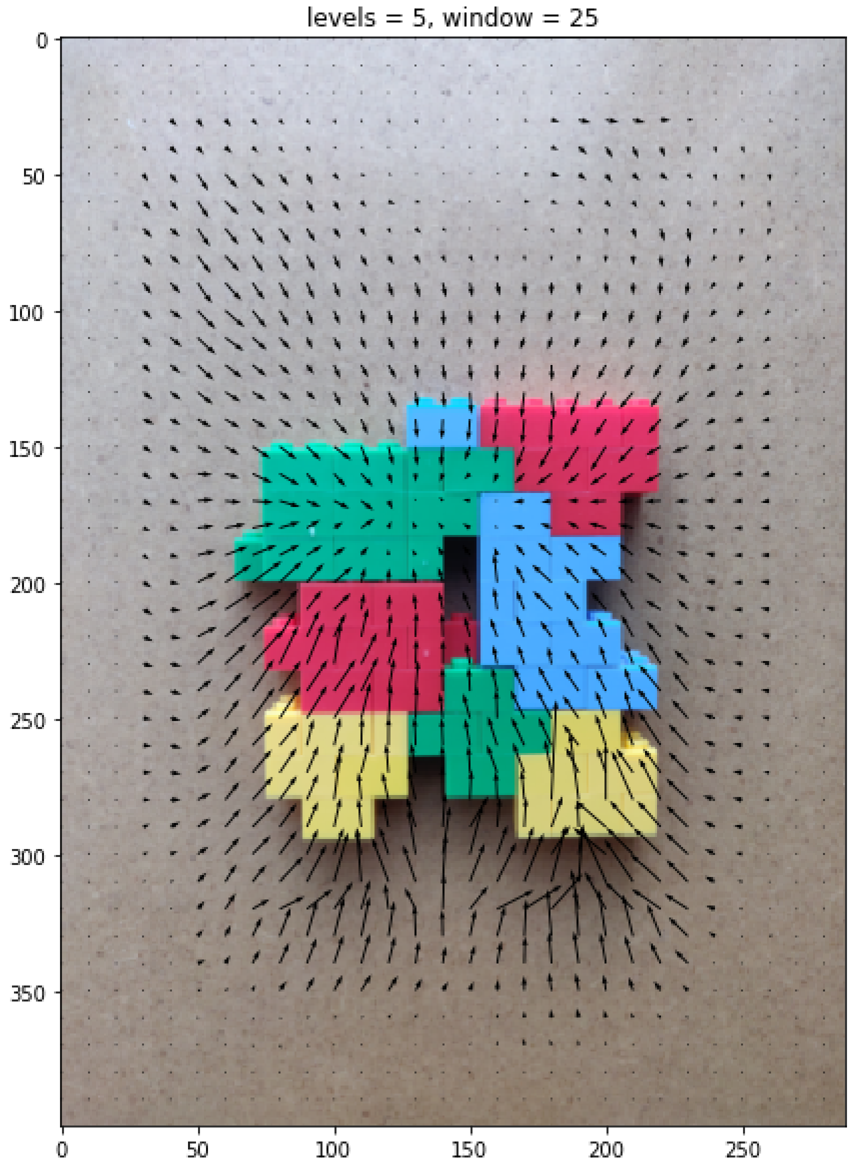
Find optical flow for the pairs (im1,im2), (im1,im3), (im1,im4) using one good window size and number of levels. Does the optical flow result seem consistent with visual inspection? Comment on the type of motion indicated by results and visual inspection and explain why they might be consistent or inconsistent.

In [236]:

```
# Your code here
# use one fixed window and numLevels for all pairs
window = 25
numLevels = 5
for i in [1,2,3]:
    U,V=LucasKanadeMultiScale(grayscale(images[0]),grayscale(images[i]),\
                              window,numLevels)
    plot_optical_flow(images[0],U,V, \
                      'levels = ' + str(numLevels) + ', window = '+str(window))
```


levels = 5, window = 25
0
50
100
150
200
250
300
350
0
50
100
150
200
250


levels = 5, window = 25
0
50
100
150
200
250
300
350
0
50
100
150
200
250



Your Comments on the results of Part 4:

By visually inspection, the transformation from image1 to image2 is a translational motion from right to left, the transformation from image1 to image3 is a clockwise rotation transformation, and the transformation from image1 to image4 is a zoom-in transformation. These visually inspections are consistent with the optical flow results in general, except for some specific optical flow in certain areas are misdirected.

To explain, the general consistency is because of 4 times of downsampling and a large window size, the effect of choosing different window size and number of levels are explained above in part2 and part3.

The inconsistency in some certain areas may be due to the change of brightness of images, or the changeless, undistinguishable background color or the magnified error while upsampling u and v.

Problem 2: Machine Learning [12 pts]

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

Part 1: Initial setup [1 pts]

Follow the directions on https://pytorch.org/get-started/locally/ (https://pytorch.org/get-started/locally/) to install Pytorch on your computer.

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

Run the torch import statements below to verify your instalation.

Download the MNIST data from http://yann.lecun.com/exdb/mnist/ (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 [154]:

```
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)
```

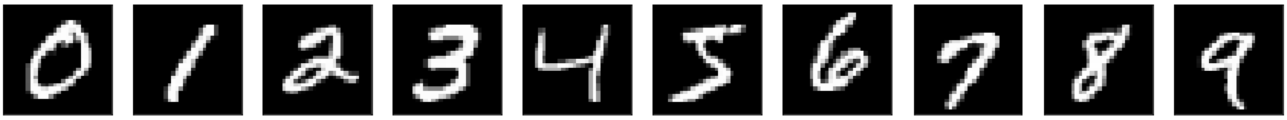
```
tensor([[0.0409, 0.2539, 0.9584],
        [0.2146, 0.7770, 0.7568],
        [0.7618, 0.6624, 0.2636],
        [0.0994, 0.9538, 0.4070],
        [0.9029, 0.0680, 0.7816]])
```

In [155]:
```python
import os
import struct

# Change path as required
path_str = r"C:\Users\yueya\Desktop\CSE252A\HW5\mnist"
# path = path_str.format(path)
path = path_str

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 [156]:
```python
# Understand the shapes of the each variable carying data
print(trainData.shape, trainLabels.shape)
print(testData.shape, testLabels.shape)
```

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

In [157]:
```
# display one image from each label
# """ ==========
# YOUR CODE HERE
# ========== """
label_list = np.zeros((1,10))
sample_digits = []
plt.figure(figsize=(15,15))
for i in range(trainLabels.shape[0]):
    if np.sum(label_list) == 10:
        break
    if label_list[0,trainLabels[i]] == 0:
        label_list[0,trainLabels[i]] = 1
        digit_sample_img = trainData[i]
        sample_digits.append([digit_sample_img])
        plt.subplot(1,10,trainLabels[i]+1)
        plt.imshow(digit_sample_img,cmap='gray')
        plt.xticks([])
        plt.yticks([])
    else:
        continue

plt.show()
```



Some helper functions are given below.

In [158]:

```
# a generator for batches of data
# yields data (batchsize, 28, 28) and labels (batchsize)
# if shuffle, it will load batches in a random order
def DataBatch(data, label, batchsize, shuffle=True):
    n = data.shape[0]
    if shuffle:
        index = np.random.permutation(n)
    else:
        index = np.arange(n)
    for i in range(int(np.ceil(n/batchsize))):
        inds = index[i*batchsize : min(n,(i+1)*batchsize)]
        yield data[inds], label[inds]

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

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

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

```
Random classifier accuracy: 9.920000
```

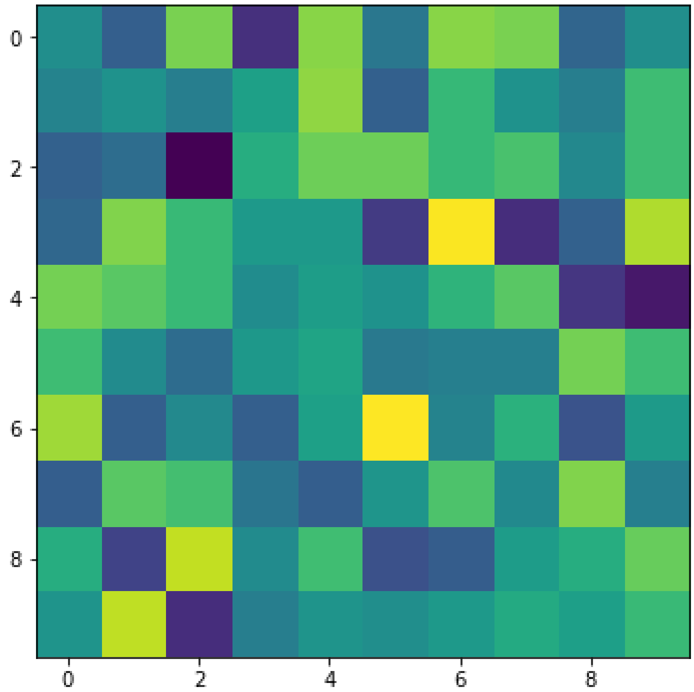
Part 2: Confusion Matrix [2 pts]

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. Can you justify the accuracy given by the random classifier?

In [161]:

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

# It would be a good idea to return the accuracy, along with the confusion
# matrix, since both can be calculated in one iteration over test data, to
# save time
def Confusion(testData, testLabels, classifier):
    M=np.zeros((10,10))
    batchsize = 50
    acc=0.0
    """ ==========
    YOUR CODE HERE
    ========== """
    acc = test(testData,testLabels,classifier)
    for data,label in tqdm(DataBatch(testData,testLabels,batchsize,shuffle=False)):
        pred = classifier(data)
        for i in range(batchsize):
            M[label[i],pred[i]] += 1

    for l_i in range(M.shape[0]):
        M[l_i] = M[l_i] / np.sum(M[l_i])

    return M, acc

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

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

```
0it [00:00, ?it/s]
200it [00:00, 20053.57it/s]

10.07
```




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

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

The accuacy of this random classifier, given 10 classes of hand-written digits from 0 to 9, is around 0.1, which is correct. For a k-class random classifier, the accuracy of each label should be 1/k.

The confusion matrix visualizes the correctness of result of classification. Each row of confusion matrix represents the true value of the label of testData, and each column is the predicted label of each testData. The diagonal of confusion matrix represent the accuacy of a classifier. Which means that, for a high accuacy classifier, the diagonal squares on the graph should be evidently brighter than other areas.

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

- Here you will implement a simple knn classifier. The distance metric is Euclidean in pixel space. k refers to the number of neighbors involved in voting on the class, and should be 3. You are allowed to use sklearn.neighbors.KNeighborsClassifier.
- Display confusion matrix and accuracy for your KNN classifier trained on the entire train set. (should be ~97 %)
- After evaluating the classifier on the testset, based on the confusion matrix, mention the number that the number '7' is most often predicted to be, other than '7'.

In [165]:
```
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.classifier = KNeighborsClassifier(n_neighbors=k,weights='distance')

    def train(self, trainData, trainLabels):
        """ ==========
        YOUR CODE HERE
        ========== """
        trainData = trainData.reshape(trainData.shape[0],\
                                      trainData.shape[1]*trainData.shape[2])
        self.classifier.fit(trainData,trainLabels)

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

# 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: 66.940000
```

In [166]:
```
# test your classifier with all the training examples (This may take a while)
knnClassifier = KNNClassifier()
knnClassifier.train(trainData, trainLabels)
```

In [167]:

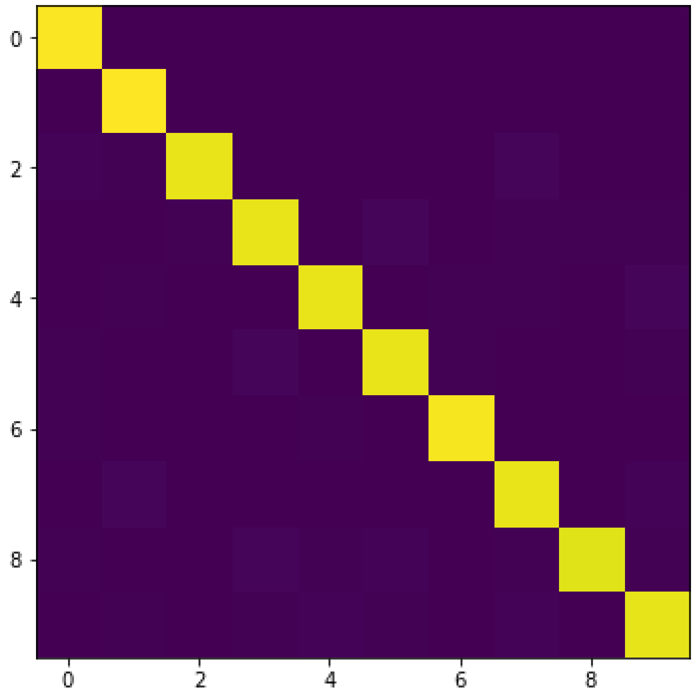
```
# display confusion matrix for your KNN classifier with all the training examples
# (This may take a while)
""" ==========
YOUR CODE HERE
========== """
M,acc = Confusion(testData,testLabels,knnClassifier)
print("KNN-Classifier accuracy:",acc)
VisualizeConfusion(M)
```

```
0it [00:00, ?it/s]
1it [00:02,  2.99s/it]
2it [00:05,  2.99s/it]
3it [00:08,  2.98s/it]
4it [00:11,  2.96s/it]
5it [00:14,  2.96s/it]
6it [00:18,  3.03s/it]
7it [00:21,  3.06s/it]
8it [00:24,  3.06s/it]
9it [00:27,  3.06s/it]
10it [00:30,  3.05s/it]
11it [00:33,  3.05s/it]
12it [00:36,  3.05s/it]
13it [00:39,  3.06s/it]
14it [00:42,  3.06s/it]
15it [00:45,  3.06s/it]
16it [00:48,  3.08s/it]
17it [00:51,  3.08s/it]
18it [00:54,  3.07s/it]
19it [00:57,  3.06s/it]
20it [01:00,  3.07s/it]
21it [01:04,  3.08s/it]
22it [01:07,  3.15s/it]
23it [01:10,  3.14s/it]
24it [01:13,  3.12s/it]
25it [01:16,  3.12s/it]
26it [01:19,  3.14s/it]
27it [01:22,  3.11s/it]
28it [01:26,  3.11s/it]
29it [01:29,  3.09s/it]
30it [01:32,  3.09s/it]
31it [01:35,  3.09s/it]
32it [01:38,  3.12s/it]
33it [01:41,  3.11s/it]
34it [01:44,  3.12s/it]
35it [01:47,  3.12s/it]
36it [01:50,  3.13s/it]
37it [01:54,  3.12s/it]
38it [01:57,  3.10s/it]
39it [02:00,  3.11s/it]
40it [02:03,  3.10s/it]
41it [02:06,  3.10s/it]
42it [02:09,  3.10s/it]
43it [02:12,  3.09s/it]
44it [02:15,  3.07s/it]
45it [02:18,  3.06s/it]
46it [02:21,  3.09s/it]
47it [02:24,  3.07s/it]
48it [02:27,  3.08s/it]
49it [02:31,  3.08s/it]
50it [02:34,  3.09s/it]
51it [02:37,  3.10s/it]
52it [02:40,  3.09s/it]
53it [02:43,  3.08s/it]
54it [02:46,  3.08s/it]
55it [02:49,  3.09s/it]
56it [02:52,  3.08s/it]
57it [02:55,  3.08s/it]
58it [02:58,  3.09s/it]
59it [03:01,  3.09s/it]
60it [03:04,  3.08s/it]
61it [03:08,  3.09s/it]
62it [03:11,  3.10s/it]
63it [03:14,  3.08s/it]
64it [03:17,  3.09s/it]
```

65it [03:20, 3.09s/it]
66it [03:23, 3.10s/it]
67it [03:26, 3.10s/it]
68it [03:29, 3.09s/it]
69it [03:32, 3.11s/it]
70it [03:35, 3.11s/it]
71it [03:39, 3.09s/it]
72it [03:42, 3.09s/it]
73it [03:45, 3.10s/it]
74it [03:48, 3.09s/it]
75it [03:51, 3.11s/it]
76it [03:54, 3.10s/it]
77it [03:57, 3.10s/it]
78it [04:00, 3.10s/it]
79it [04:03, 3.10s/it]
80it [04:06, 3.11s/it]
81it [04:10, 3.11s/it]
82it [04:13, 3.10s/it]
83it [04:16, 3.09s/it]
84it [04:19, 3.10s/it]
85it [04:22, 3.09s/it]
86it [04:25, 3.09s/it]
87it [04:28, 3.08s/it]
88it [04:31, 3.08s/it]
89it [04:34, 3.09s/it]
90it [04:37, 3.07s/it]
91it [04:40, 3.06s/it]
92it [04:43, 3.06s/it]
93it [04:46, 3.06s/it]
94it [04:50, 3.07s/it]
95it [04:53, 3.06s/it]
96it [04:56, 3.05s/it]
97it [04:59, 3.05s/it]
98it [05:02, 3.06s/it]
99it [05:05, 3.07s/it]
100it [05:08, 3.08s/it]
101it [05:11, 3.05s/it]
102it [05:14, 3.04s/it]
103it [05:17, 3.05s/it]
104it [05:20, 3.08s/it]
105it [05:23, 3.08s/it]
106it [05:26, 3.08s/it]
107it [05:29, 3.08s/it]
108it [05:33, 3.10s/it]
109it [05:36, 3.12s/it]
110it [05:39, 3.12s/it]
111it [05:42, 3.07s/it]
112it [05:45, 3.05s/it]
113it [05:48, 3.08s/it]
114it [05:51, 3.09s/it]
115it [05:54, 3.08s/it]
116it [05:57, 3.07s/it]
117it [06:00, 3.05s/it]
118it [06:03, 3.05s/it]
119it [06:06, 3.06s/it]
120it [06:09, 3.06s/it]
121it [06:12, 3.06s/it]
122it [06:15, 3.05s/it]
123it [06:19, 3.06s/it]
124it [06:22, 3.05s/it]
125it [06:24, 3.02s/it]
126it [06:28, 3.03s/it]
127it [06:31, 3.04s/it]
128it [06:34, 3.04s/it]
129it [06:37, 3.04s/it]
130it [06:40, 3.05s/it]

131it [06:43, 3.07s/it]
132it [06:46, 3.08s/it]
133it [06:49, 3.08s/it]
134it [06:52, 3.09s/it]
135it [06:55, 3.07s/it]
136it [06:58, 3.06s/it]
137it [07:01, 3.06s/it]
138it [07:04, 3.05s/it]
139it [07:07, 3.04s/it]
140it [07:10, 3.06s/it]
141it [07:14, 3.08s/it]
142it [07:17, 3.08s/it]
143it [07:20, 3.08s/it]
144it [07:23, 3.06s/it]
145it [07:26, 3.05s/it]
146it [07:29, 3.06s/it]
147it [07:32, 3.03s/it]
148it [07:35, 3.00s/it]
149it [07:38, 3.00s/it]
150it [07:41, 3.01s/it]
151it [07:44, 3.03s/it]
152it [07:47, 3.02s/it]
153it [07:50, 3.06s/it]
154it [07:53, 3.06s/it]
155it [07:56, 3.05s/it]
156it [07:59, 3.06s/it]
157it [08:02, 3.07s/it]
158it [08:05, 3.06s/it]
159it [08:08, 3.05s/it]
160it [08:11, 3.05s/it]
161it [08:14, 3.06s/it]
162it [08:18, 3.08s/it]
163it [08:21, 3.11s/it]
164it [08:24, 3.10s/it]
165it [08:27, 3.08s/it]
166it [08:30, 3.06s/it]
167it [08:33, 3.06s/it]
168it [08:36, 3.07s/it]
169it [08:39, 3.07s/it]
170it [08:42, 3.07s/it]
171it [08:45, 3.07s/it]
172it [08:48, 3.10s/it]
173it [08:51, 3.09s/it]
174it [08:55, 3.09s/it]
175it [08:58, 3.08s/it]
176it [09:01, 3.07s/it]
177it [09:04, 3.10s/it]
178it [09:07, 3.09s/it]
179it [09:10, 3.11s/it]
180it [09:13, 3.11s/it]
181it [09:16, 3.10s/it]
182it [09:19, 3.12s/it]
183it [09:22, 3.10s/it]
184it [09:26, 3.09s/it]
185it [09:29, 3.09s/it]
186it [09:32, 3.10s/it]
187it [09:35, 3.09s/it]
188it [09:38, 3.09s/it]
189it [09:41, 3.07s/it]
190it [09:44, 3.08s/it]
191it [09:47, 3.09s/it]
192it [09:50, 3.10s/it]
193it [09:53, 3.10s/it]
194it [09:56, 3.10s/it]
195it [10:00, 3.10s/it]
196it [10:03, 3.17s/it]

```
197it [10:06,  3.17s/it]
198it [10:09,  3.13s/it]
199it [10:12,  3.12s/it]
200it [10:15,  3.12s/it]

KNN-Classifier accuracy: 97.17
```




```
[[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.   0.01 0.   0.   0.02]
 [0.   0.   0.   0.01 0.   0.96 0.01 0.   0.   0.  ]
 [0.   0.   0.   0.   0.   0.   0.99 0.   0.   0.  ]
 [0.   0.02 0.   0.   0.   0.   0.   0.97 0.   0.01]
 [0.01 0.   0.   0.01 0.01 0.01 0.   0.   0.95 0.01]
 [0.   0.   0.   0.01 0.01 0.   0.   0.01 0.   0.96]]
```

Answer

"1" is the number that number '7' is most often predicted to be, other than '7'.

The probability of mispredicting "7" to be "1" is 0.02.

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 [168]:
```
class PCAKNNClassifer():
    def __init__(self, components=25, k=3):
        # components = number of principal components
        # k is the number of neighbors involved in voting
        """ ==========
        YOUR CODE HERE
        ========== """
        self.components = components
        self.k = k
        self.classifier = KNeighborsClassifier(n_neighbors=self.k)

    def train(self, trainData, trainLabels):
        """ ==========
        YOUR CODE HERE
        ========== """
        train_flat = trainData.reshape(trainData.shape[0],-1)
        train_cov = np.cov(train_flat.T)
        u,s,v = np.linalg.svd(train_cov)
        self.eigvalue = u[:,0:self.components]
        train_fin = np.dot(train_flat,self.eigvalue)
        self.classifier.fit(train_fin,trainLabels)

    def __call__(self, x):
        # this method should take a batch of images
        # and return a batch of predictions
        """ ==========
        YOUR CODE HERE
        ========== """
        x_flat = x.reshape(x.shape[0],-1)
        train_fin = np.dot(x_flat,self.eigvalue)
        return self.classifier.predict(train_fin)

# 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
```

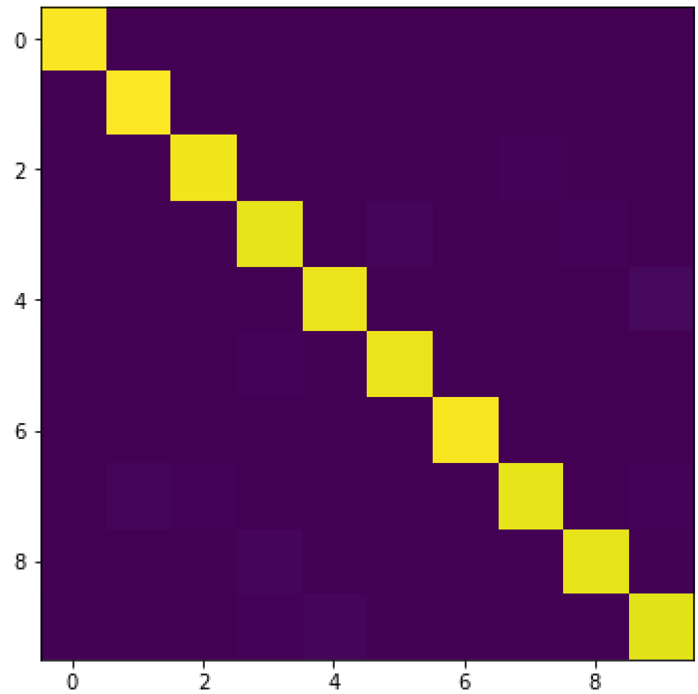
In [169]:
```
# test your classifier with all the training examples
pcaknnClassifer = PCAKNNClassifer()
pcaknnClassifer.train(trainData, trainLabels)
```

In [170]:

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

```
0it [00:00, ?it/s]
2it [00:00, 13.55it/s]
4it [00:00, 14.18it/s]
6it [00:00, 14.63it/s]
8it [00:00, 14.89it/s]
10it [00:00, 13.92it/s]
12it [00:00, 13.64it/s]
14it [00:00, 13.89it/s]
16it [00:01, 14.60it/s]
18it [00:01, 14.18it/s]
20it [00:01, 14.31it/s]
22it [00:01, 14.60it/s]
24it [00:01, 14.09it/s]
26it [00:01, 13.28it/s]
28it [00:01, 13.66it/s]
30it [00:02, 13.77it/s]
32it [00:02, 13.29it/s]
34it [00:02, 12.54it/s]
36it [00:02, 12.47it/s]
38it [00:02, 13.02it/s]
40it [00:02, 12.65it/s]
42it [00:03, 13.14it/s]
44it [00:03, 13.50it/s]
46it [00:03, 13.57it/s]
48it [00:03, 13.79it/s]
50it [00:03, 14.33it/s]
52it [00:03, 14.74it/s]
54it [00:03, 14.74it/s]
56it [00:04, 14.68it/s]
58it [00:04, 14.57it/s]
60it [00:04, 14.49it/s]
62it [00:04, 14.92it/s]
64it [00:04, 14.58it/s]
66it [00:04, 14.86it/s]
68it [00:04, 15.23it/s]
70it [00:04, 15.87it/s]
72it [00:05, 14.99it/s]
74it [00:05, 15.58it/s]
76it [00:05, 14.61it/s]
78it [00:05, 13.18it/s]
80it [00:05, 13.00it/s]
82it [00:05, 13.43it/s]
84it [00:05, 13.55it/s]
86it [00:06, 13.67it/s]
88it [00:06, 13.52it/s]
90it [00:06, 12.40it/s]
92it [00:06, 12.63it/s]
94it [00:06, 12.82it/s]
96it [00:06, 12.93it/s]
98it [00:07, 12.59it/s]
100it [00:07, 12.96it/s]
102it [00:07, 13.72it/s]
104it [00:07, 13.32it/s]
106it [00:07, 13.63it/s]
109it [00:07, 14.58it/s]
111it [00:07, 15.73it/s]
113it [00:08, 14.48it/s]
115it [00:08, 13.37it/s]
117it [00:08, 13.05it/s]
119it [00:08, 12.87it/s]
121it [00:08, 13.31it/s]
123it [00:08, 14.36it/s]
125it [00:08, 15.55it/s]
127it [00:09, 16.47it/s]
129it [00:09, 15.99it/s]
```

```
131it [00:09, 16.28it/s]
133it [00:09, 13.80it/s]
135it [00:09, 14.31it/s]
137it [00:09, 14.82it/s]
139it [00:09, 15.07it/s]
141it [00:10, 15.45it/s]
143it [00:10, 15.85it/s]
145it [00:10, 14.78it/s]
147it [00:10, 14.87it/s]
149it [00:10, 14.80it/s]
151it [00:10, 14.85it/s]
153it [00:10, 14.85it/s]
155it [00:10, 15.33it/s]
157it [00:11, 15.83it/s]
159it [00:11, 16.69it/s]
161it [00:11, 16.91it/s]
163it [00:11, 16.76it/s]
165it [00:11, 16.30it/s]
167it [00:11, 15.02it/s]
169it [00:11, 14.97it/s]
171it [00:11, 15.85it/s]
173it [00:12, 16.06it/s]
176it [00:12, 17.88it/s]
179it [00:12, 19.11it/s]
181it [00:12, 18.12it/s]
183it [00:12, 16.62it/s]
185it [00:12, 16.28it/s]
187it [00:12, 15.53it/s]
189it [00:13, 14.81it/s]
191it [00:13, 13.99it/s]
193it [00:13, 13.63it/s]
195it [00:13, 13.00it/s]
197it [00:13, 13.05it/s]
199it [00:13, 12.32it/s]
200it [00:13, 14.38it/s]

PCA-KNN classifier accuracy:  97.31
```



```
[[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]]
```

Comments:

PCA-KNN classifier is obviously faster than KNN classifier. This is because the dimension of training data and test data are reduced from 28×28 to 25, which saves a lot of calculation and memory. It can also be implied that, because the accuray of PCA-KNN classifier remains the same as KNN-classifier, a 25-dimension vector is able to contain enough info of a 28×28 hand-written digit image.

Problem 3: Deep learning [14 pts]

Below is some helper code to train your deep networks.

Part 1: Training with PyTorch [2 pts]

Below is some helper code to train your deep networks. Complete the train function for DNN below. You should write down the training operations in this function. That means, for a batch of data you have to initialize the gradients, forward propagate the data, compute error, do back propagation and finally update the parameters. This function will be used in the following questions with different networks. You can look at https://pytorch.org/tutorials/beginner/pytorch_with_examples.html (https://pytorch.org/tutorials/beginner/pytorch_with_examples.html) for reference.

In [193]:

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

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

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 = torch.optim.Adam(self.parameters(), lr = 3e-4)

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

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

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

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

# helper function to get weight variable
def weight_variable(shape):
    initial = torch.Tensor(truncnorm.rvs(-1/0.01, 1/0.01, scale=0.01, size=shape))
    return Parameter(initial, requires_grad=True)

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

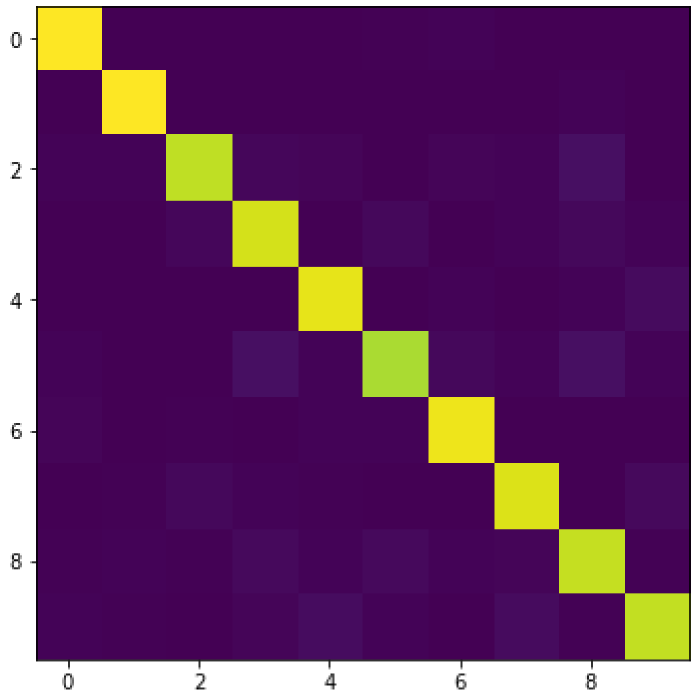
In [194]:
```python
# example linear classifier - input connected to output
# you can take this as an example to learn how to extend DNN class
class LinearClassifier(DNN):
    def __init__(self, in_features=28*28, classes=10):
        super(LinearClassifier, self).__init__()
        # in_features=28*28
        self.weight1 = weight_variable((classes, in_features))
        self.bias1 = bias_variable((classes))

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

trainData=np.array(list(read('training','images')))
trainData=np.float32(np.expand_dims(trainData,-1))/255
trainData=trainData.transpose((0,3,1,2))
trainLabels=np.int32(np.array(list(read('training','labels'))))

testData=np.array(list(read('testing','images')))
testData=np.float32(np.expand_dims(testData,-1))/255
testData=testData.transpose((0,3,1,2))
testLabels=np.int32(np.array(list(read('testing','labels'))))
```

In [195]:
```python
# 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: 89.150000
Epoch:2 Accuracy: 90.750000
Epoch:3 Accuracy: 91.330000
Epoch:4 Accuracy: 91.510000
Epoch:5 Accuracy: 91.780000
Epoch:6 Accuracy: 92.050000
Epoch:7 Accuracy: 92.120000
Epoch:8 Accuracy: 92.270000
Epoch:9 Accuracy: 92.370000
Epoch:10 Accuracy: 92.430000
```

In [196]:
```
# display confusion matrix
""" ==========
YOUR CODE HERE
========== """
M,acc = Confusion(testData,testLabels,linearClassifier)
print("Linear Classifier accuracy: ", acc)
VisualizeConfusion(M)
```

```
0it [00:00, ?it/s]
200it [00:00, 4664.51it/s]

Linear Classifier accuracy:  92.43
```

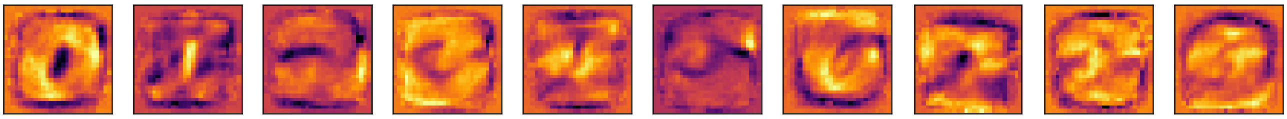


```
[[0.98 0.   0.   0.   0.   0.01 0.01 0.   0.   0.  ]
 [0.   0.98 0.   0.   0.   0.   0.   0.   0.01 0.  ]
 [0.01 0.01 0.89 0.02 0.01 0.   0.01 0.01 0.04 0.  ]
 [0.   0.   0.02 0.92 0.   0.02 0.   0.01 0.02 0.01]
 [0.   0.   0.   0.   0.94 0.   0.01 0.   0.01 0.03]
 [0.01 0.   0.   0.04 0.01 0.86 0.02 0.01 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.   0.   0.   0.93 0.   0.03]
 [0.01 0.01 0.01 0.02 0.01 0.02 0.01 0.01 0.89 0.  ]
 [0.01 0.01 0.   0.01 0.03 0.01 0.   0.03 0.01 0.89]]
```

Part 2: Single Layer Perceptron [2 pts]

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

In [197]:
```
# Plot filter weights corresponding to each class, you may have to reshape them to make sense o
ut of them
# linearClassifier.weight1.data will give you the first layer weights
weight = linearClassifier.weight1.data
plt.figure(figsize=(15,15))
for i,i_weight in enumerate(weight):
    temp = i_weight.reshape(28,28)
    temp = (temp-temp.min())/(temp.max()-temp.min())
    plt.subplot(1,10,i+1)
    plt.imshow(temp,cmap='inferno')
    plt.xticks([])
    plt.yticks([])
plt.show()
```



Comments on weights

The weights of simple linear classification are shown above. We may notics that each weight looks like a 0~9 number. It is more obvious for the shape of weight "0","2","3", "6","7","8",'9'.

During training process, because the number of classes equals to the number of labels, and the label is the hand-written digit, so that each class from the classifier only need to learn the shape of a single number. However, for that the hand writing of the same number by different person varis, the weight of each number after training does not look exactly the same as what we see in daily life. And the imput image will be weighted by these ten weights, the pair with the most similar shape will be the predicted label.

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

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

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

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

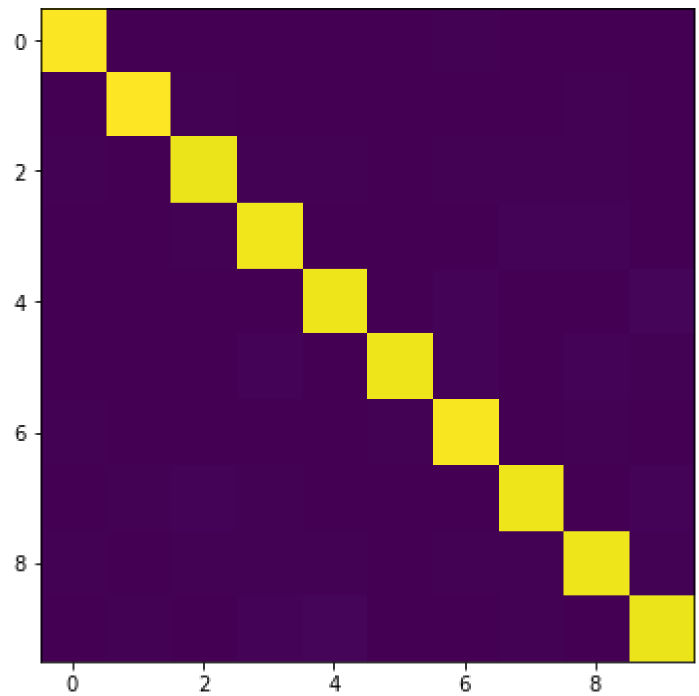
Plot the filter weights corresponding to the mapping from the inputs to the first 10 hidden layer outputs (out of 100). Do the weights look similar to the weights plotted in the previous problem? Why or why not?

In [198]:
```
class MLPClassifier(DNN):
    def __init__(self, in_features=28*28, classes=10, hidden=100):
        """ ==========
        YOUR CODE HERE
        ========== """
#         raise NotImplementedError
        super(MLPClassifier,self).__init__()
        self.feat_weight1 = weight_variable((in_features,hidden))
        self.feat_bias1 = bias_variable((hidden))

        self.feat_weight2 = weight_variable((hidden,classes))
        self.feat_bias2 = bias_variable((classes))

    def forward(self, x):
        """ ==========
        YOUR CODE HERE
        ========== """
#         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 ou
tput.

#         The final output of the computation graph should be stored in self.y as that will be
 used in the training.
        temp1 = torch.addmm(self.feat_bias1,x.view(-1,28*28),self.feat_weight1)
        temp2 = F.relu(temp1)
        Output = torch.addmm(self.feat_bias2,temp2,self.feat_weight2)

        return Output

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

```
Epoch:1 Accuracy: 91.560000
Epoch:2 Accuracy: 92.880000
Epoch:3 Accuracy: 93.880000
Epoch:4 Accuracy: 94.630000
Epoch:5 Accuracy: 95.430000
Epoch:6 Accuracy: 96.030000
Epoch:7 Accuracy: 96.170000
Epoch:8 Accuracy: 96.400000
Epoch:9 Accuracy: 96.620000
Epoch:10 Accuracy: 96.890000
```

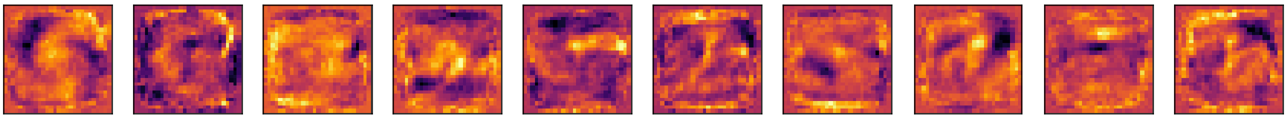
In [199]:
```
# Plot confusion matrix
M,acc = Confusion(testData, testLabels, mlpClassifier)
print("MLP Classifier accuracy: ",acc)
VisualizeConfusion(M)
```

```
0it [00:00, ?it/s]
200it [00:00, 2673.80it/s]

MLP Classifier accuracy:  96.89
```



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

In [251]:
```
# Plot filter weights
weight1 = mlpClassifier.feat_weight1.data
plt.figure(figsize=(15,15))
for i in range(10):
    temp = weight1[:,i]
    weight_temp = (temp-temp.min())/(temp.max()-temp.min())
    plt.subplot(1,10,i+1)
    plt.imshow(weight_temp.reshape((28,28)),cmap='inferno')
    plt.xticks([])
    plt.yticks([])
plt.show()
```



Comments on weights:

From the plots of weights shown above, it can be concluded that the weights in this MLP classifier do not look like any number. This is because the first layer is defined to have 100 neuron nodes. Each node does not tend to learn the feature of a whole training image, rather, feature of a partial training image. So that each weight cannnot be distinguished as a number

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

Here you will implement a CNN with the following architecture:

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

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

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

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

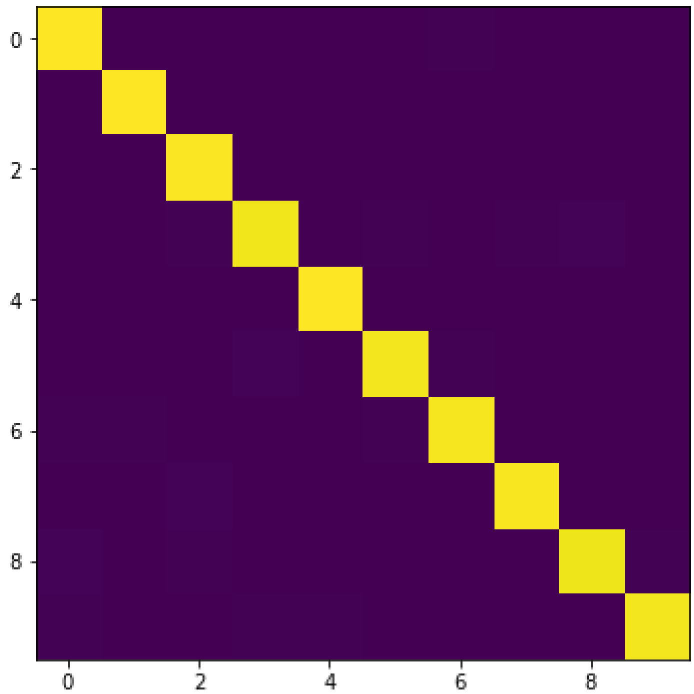
In [201]:
```
def conv2d(x, W, stride):
    # x: input
    # W: weights (out, in, kH, kW)
#     print("x.shape:",x.shape)
#     print("W.shape:",W.shape)
#     print("stride:",stride)
    return F.conv2d(x, W, stride=stride, padding=2)

# Defining a Convolutional Neural Network
class CNNClassifier(DNN):
    def __init__(self, classes=10, n=5):
        super(CNNClassifier, self).__init__()
        """ ==========
        YOUR CODE HERE
        ========== """
        self.weight_c1 = weight_variable([n,1,5,5])
        self.bias_c1 = bias_variable([14,14])

        self.weight_c2 = weight_variable([n*2,n,5,5])
        self.bias_c2 = bias_variable([7,7])

        self.weight_lin = weight_variable([64,7*7*2*n])
        self.bias_lin = bias_variable([64])

        self.weight_output = weight_variable([classes,64])
        self.bias_output = bias_variable([classes])

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

        C1 = F.relu(conv2d(x.view(-1,1,28,28),self.weight_c1,stride = 2) + self.bias_c1)
#         print(C1.shape)
#         print(self.weight_c1.shape)
#         print(self.weight_c2.shape)
        C2 = F.relu(conv2d(C1 , self.weight_c2 , stride = 2)+ self.bias_c2)
        L1 = F.relu(torch.addmm(self.bias_lin,C2.view(list(C2.size())[0],-1),\
                                        self.weight_lin.t()))
        Output = torch.addmm(self.bias_output,L1.view(list(L1.size())[0],-1),\
                                  self.weight_output.t())
        return Output


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

```
Epoch:1 Accuracy: 91.520000
Epoch:2 Accuracy: 94.070000
Epoch:3 Accuracy: 95.760000
Epoch:4 Accuracy: 96.410000
Epoch:5 Accuracy: 97.110000
Epoch:6 Accuracy: 97.580000
Epoch:7 Accuracy: 97.700000
Epoch:8 Accuracy: 97.780000
Epoch:9 Accuracy: 97.950000
Epoch:10 Accuracy: 98.240000
```

In [202]:

```
# Plot Confusion matrix
M,acc = Confusion(testData,testLabels,cnnClassifier)
print("CNN Classifier accuracy: ",acc)
VisualizeConfusion(M)
```

```
0it [00:00, ?it/s]
53it [00:00, 521.01it/s]
94it [00:00, 480.63it/s]
141it [00:00, 476.29it/s]
187it [00:00, 470.22it/s]
200it [00:00, 459.94it/s]

CNN Classifier accuracy:  98.24000000000001
```




```
[[0.99 0.   0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.99 0.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.99 0.   0.   0.   0.   0.   0.   0.  ]
 [0.   0.   0.01 0.97 0.   0.   0.   0.01 0.01 0.  ]
 [0.   0.   0.   0.   0.99 0.   0.   0.   0.   0.  ]
 [0.   0.   0.   0.01 0.   0.98 0.   0.   0.   0.  ]
 [0.01 0.   0.   0.   0.   0.01 0.98 0.   0.   0.  ]
 [0.   0.   0.01 0.   0.   0.   0.   0.98 0.   0.  ]
 [0.01 0.   0.01 0.   0.   0.   0.   0.   0.97 0.  ]
 [0.   0.   0.   0.   0.01 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/ (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 (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/ (https://playground.tensorflow.org/)