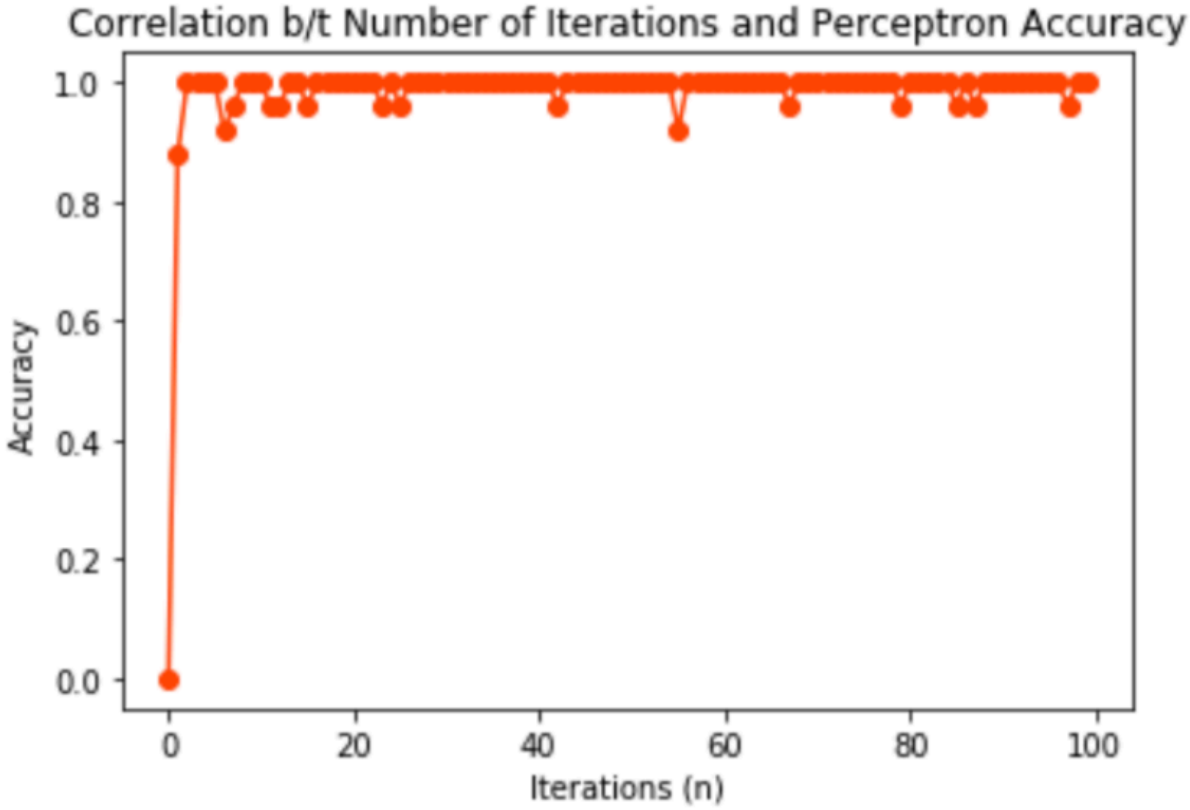
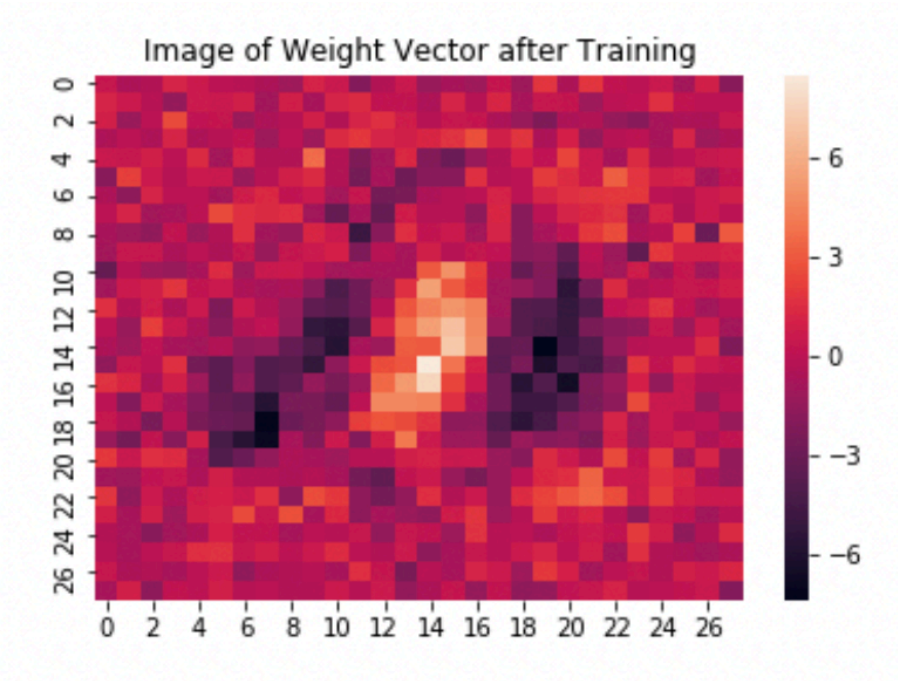

Correlation b/t Number of Iterations and Perceptron Accuracy
Accuracy
Iterations (n)
0.0
0.2
0.4
0.6
0.8
1.0
0
20
40
60
80
100

- Although there are some points that are not perfectly 1.0, I can safely say that the graph converges on 100% as the graph as a whole does keep going back to convergence of 100% for many consecutive iterations. Since new data is introduced, it's expected that the graph will have these minor adjustments, but it does adjust rather very quickly to converge to 100%.
- As to the linear separability of 0 and 1, it does mean that 0's and 1's are very well separated from each other as to be able to draw a straight line between the clusters of each and divide them exactly.

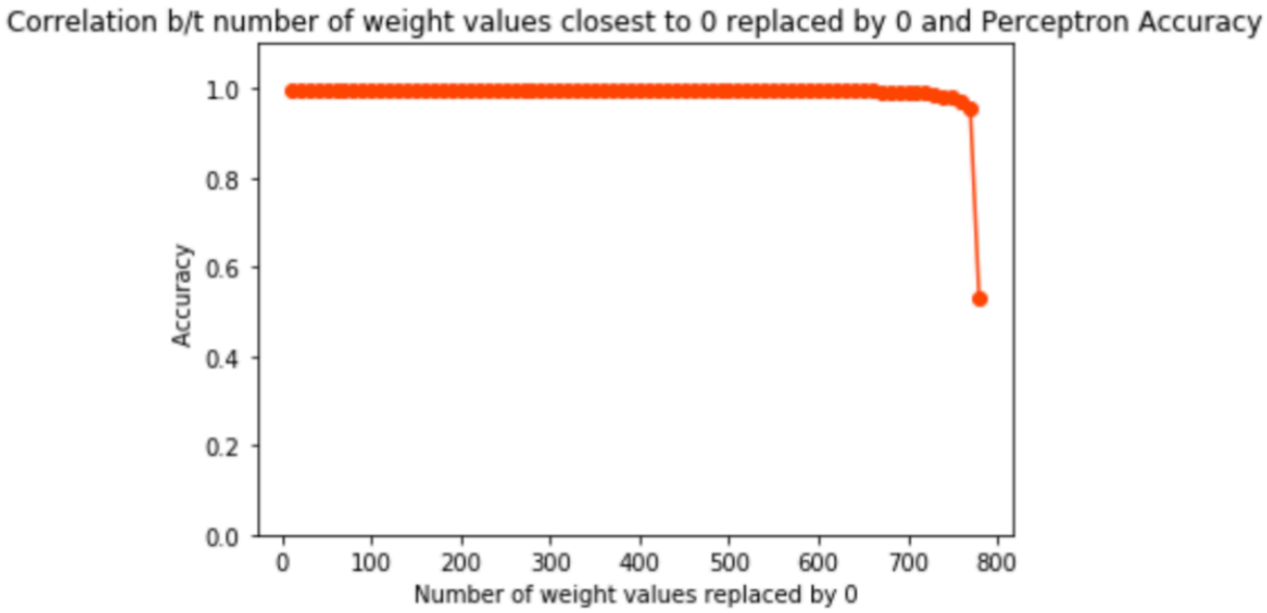
```
[(8.87013876168719, '[13][15]'), (7.9412573865888865, '[14][15]'), (7.148240915448326, '[16][14]'), (7.12901904915489
1, '[15][16]'), (6.9703029666968, '[17][15]'), (6.947589013452074, '[15][15]'), (6.547150453255847, '[16][16]'), (5.8
94566154704274, '[13][17]'), (5.520996555690247, '[13][16]'), (5.469865830629079, '[16][15]'), (5.160057681331686,
'[12][15]'), (4.998351748635079, '[14][16]'), (4.988305414033546, '[18][13]'), (4.910332096364416, '[11][15]'), (4.70
394795943642, '[18][14]'), (4.249041413255642, '[15][17]'), (4.210920742264989, '[18][15]'), (4.131889400171184, '[1
4][17]'), (4.117664885992024, '[22][21]'), (3.781350464994091, '[23][9]'), (3.738394328799423, '[17][16]'), (3.673864
3097535517, '[13][14]'), (3.6669196913308335, '[17][14]'), (3.6631342742715507, '[8][22]'), (3.5572665462957263, '[1
2][16]'), (3.333448334874686, '[20][13]'), (3.237752863604709, '[14][14]'), (3.09338165066349, '[15][14]'), (3.022896
0803046845, '[20][14]'), (3.00454037823687, '[24][9]'), (2.9682666141482335, '[21][5]'), (2.918880562958727, '[25]
'[16][11]'), (-3.5975828913449934, '[14][11]'), (-3.64637848345323, '[12][19]'), (-3.720807026228437, '[13][10]'), (-
3.7253782884684803, '[17][18]'), (-3.7493589409985155, '[14][7]'), (-3.786047116977925, '[20][10]'), (-3.949788217171
653, '[18][19]'), (-3.95176054972446, '[15][9]'), (-3.97114051484922, '[14][19]'), (-4.001068799158803, '[17][20]'),
(-4.013493116839517, '[18][7]'), (-4.01797809912474, '[18][6]'), (-4.026777649783714, '[9][13]'), (-4.07958951566795
2, '[18][9]'), (-4.409399717069011, '[15][12]'), (-4.50408132795838, '[13][7]'), (-4.593416909315928, '[13][20]'), (-
4.672588520053095, '[13][11]'), (-4.81057852068069, '[14][22]'), (-4.879466391871424, '[14][12]'), (-4.99525756238431
1, '[17][9]'), (-5.019516702214586, '[19][18]'), (-5.052983013959858, '[19][6]'), (-5.35069359585558, '[16][8]'), (-
5.357804703137458, '[17][19]'), (-5.838396716413561, '[14][20]'), (-5.921989839849385, '[20][17]'), (-5.9781463913144
86, '[15][20]'), (-6.077224358691966, '[13][12]'), (-8.165086088872759, '[16][19]')]
```

-First half of the weights list screenshot is part of the largest positive value group and second half is part of the largest negative values. As shown in the heat map, the large positive values correspond to the brightest of the map, which is shown to be the center of the map. As opposed to the large positive values, large negative values correspond to darkest regions of the map which is right around the center of the brighest region.

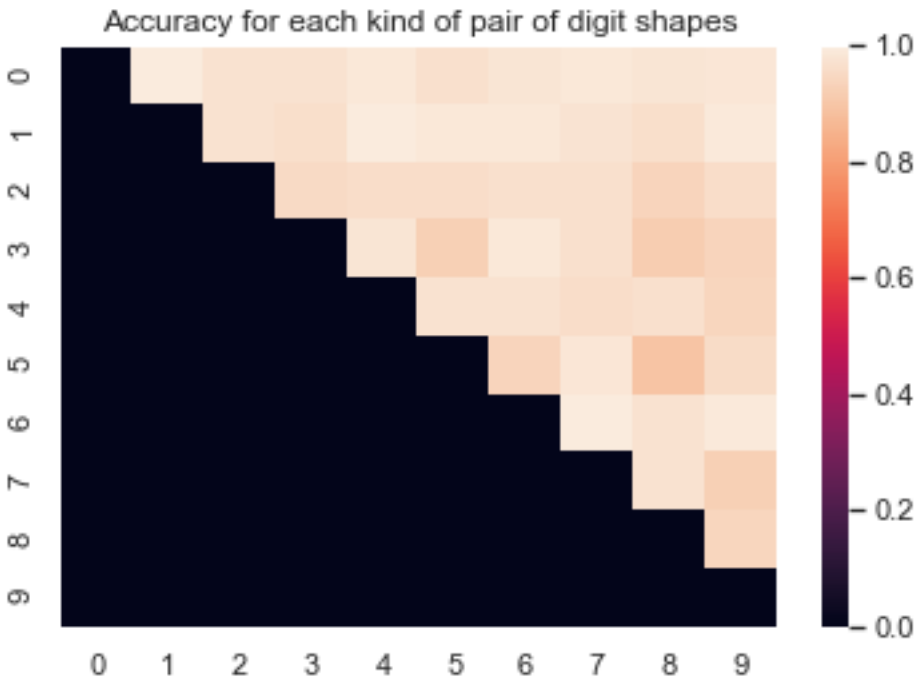
- Intuitively, it means that large positive values correspond to the strong learning correlation, therefore, perceptron aligns with the expected result whereas large negative values correspond to the weak learning correlation, hence, perceptron does not align with the expected result. The values close to 0 are dominated in the regions that's not within the center of the graph or around the center of the graph. This means that these values are neutral as for the perceptron learning process.

- The matrix looks like this because the the numbers 0 and 1 in the image files are written in a way that the correct number(when the decision matched the label) was written in the very center and the incorrect number(when the decision didn't match the label) was written right around the perimeter of the correct number as to distinguish the shape of the number and again the correct number around the border. So for example, if I look at the image files for '0', the center is encoded as 0 and right around that center it's encoded as 1 to make the shape of 0. So center region will be the region that is correct for perceptron, which will result in the increased weights, which in turn leads to large positive values. On the other hand, the region right around the center that didn't get the correct output will result in decreased weight, which in turn leads to large negative weight values.



- I expected the graph to maintain the 1.0 or near 1.0 accuracy at least until 700, which turned out to be close to what I actually got in my plot.

- As for the proportion of the image, this exactly aligns with my explanation about how only small portion of weight values were actually what was being used for perceptron learning(large positive values) and un- learning(large negative values). Since there were much higher number of neutral weights(values close to 0) than large positive or negative values, replacing the values closest to 0 does not adversely affect the accuracy of our perceptron learning. As the graph shows, it only starts to take an effect around 650th weight values replaced by 0 and hits its minimum at when all of its weight values(n=780) are replaced by 0. This observation is obvious because all the large positive values and negative values are now completely eliminated as well, which were important weight values for our learning algorithm. All in all, whether it's "0" or "1", replacing values close to 0 did not affect our accuracy much until when it started to substitute large positive or negative values around when # of weight values replaced was 650 and aftertwards.



*Ignore the bottom black part as those are intentionally blacked out to ignore the duplicate pairs and diagonals and to consider the upper half of the plot only.

- My original intuition for ‘easy’ was any pair of numbers that has good contrast such as (7, 3), (0, 1), (3, 4), etc. and for ‘hard’ was anything that has similarities such as pair(0, 8) or pair(3, 8); ‘8’ has double ‘0’ up and down and ‘3’ has half the shape of ‘8’. When I plotted the color map, the graph confirmed my intuition about ‘easy’ pairs for most part as pairs with high contrast did have high accuracy. For ‘hard’ pairs, I was right about (3, 8) being ‘hard’ with the lower accuracy range, but (0, 8) turned out to be on the higher accuracy range. Surprisingly, pair (3, 5) and (5, 8) turned out to have the lowest accuracy range as well. I believe this might be due to the fact that lower curve of the number of ‘3’, ‘5’ and ‘8’ has a similar curve, hence resulting in the more difficulty of distinguishing between each other than other pairs of numbers.

1

In [ ]:
```python
import os
import numpy as np
import random
import matplotlib.pyplot as plt

# Functions that might be useful (please read the documentation)
# x.flatten() (take a N-dimensional numpy array and make it one-dimensional)
# numpy.random.choice -- choose from the list of images
# numpy.dot -- compute the dot product
# numpy.random.normal -- set up random initial weights

DIM = (28, 28)  # these are the dimensions of the image


def load_image_files(n, path="images/"):
    # helper file to help load the images
    # returns a list of numpy vectors
    images = []
    for f in os.listdir(os.path.join(path, str(n))):  # read files in the path
        p = os.path.join(path, str(n), f)
        if os.path.isfile(p):
            i = np.loadtxt(p)
            assert i.shape == DIM  # just check the dimensions here
            # i is loaded as a matrix, but we are going to flatten it into a single vector
            images.append(i.flatten())
    return images


# Load up these image files
A = load_image_files(0)
B = load_image_files(1)

N = len(A[0])  # the total size
assert N == DIM[0] * DIM[1]  # just check our sizes to be sure

# set up some random initial weights
weights = np.random.normal(0, 1, size=N)

#Your code here:
NUM_BLOCK = 25

step_func = lambda x: 0 if x < 0 else 1

#This tags image of A & B with their actual value
shuffled_training_data = random.sample(list(zip(A, [0]*len(A))) + list(zip(B, [1]*len(B))), len(A) + len(B))

shuffled_copy = shuffled_training_data

accuracy_list = []
weights_list = []


def training_func():
    global weights, shuffled_copy
    accuracy_count = 0

    for i in range(NUM_BLOCK):
        output = 0
        actual_value = shuffled_copy[i][1]
        for n in range(N):
            output += (weights[n] * shuffled_copy[i][0][n])

        weights_list.append(output)
        weights_list.append(np.dot(shuffled_copy[i][0], weights))

        y = step_func(output)

        if actual_value == 0 and y == 1:
            weights -= shuffled_copy[i][0]
        elif actual_value == 1 and y == 0:
            weights += shuffled_copy[i][0]

        if actual_value == y:
            accuracy_count += 1

    #copies same shuffled copy but without the iterated 25 images.
    shuffled_copy = shuffled_copy[25:]

    accuracy_list.append(accuracy_count/NUM_BLOCK)



#runs the training_func() 100 iterations
i = 0
while True:
    training_func()
    i += 1

    if i == 99:
        print(accuracy_list)
        break


x = [a for a in range(100)]
y = [0] + accuracy_list

print(accuracy_list)


plt.plot(x, y, color='orangered', marker = "o")
plt.xlabel("Iterations (n)")
plt.ylabel("Accuracy")
plt.title('Correlation b/t Number of Iterations and Perceptron Accuracy')

plt.savefig("foo.png")
plt.show()
```

2

- Although there are some points that are not perfectly 1.0, I can safely say that the graph converges on 100% as the graph as a whole does keep going back to convergence of 100% for many consecutive iterations. Since new data is introduced, it's expected that the graph will have these minor adjustments, but it does adjust rather very quickly to converge to 100%.
- As to the linear separability of 0 and 1, it does mean that 0's and 1's are very well separated from each other as to be able to draw a straight line between the clusters of each and divide them exactly.

3

In [ ]:
```python
import os
import numpy as np
import random
import matplotlib.pyplot as plt
import seaborn as sns

# Functions that might be useful (please read the documentation)
# x.flatten() (take a N-dimensional numpy array and make it one-dimensional)
# numpy.random.choice -- choose from the list of images
# numpy.dot -- compute the dot product
# numpy.random.normal -- set up random initial weights

DIM = (28, 28)  # these are the dimensions of the image


def load_image_files(n, path="images/"):
    # helper file to help load the images
    # returns a list of numpy vectors
    images = []
    for f in os.listdir(os.path.join(path, str(n))):  # read files in the path
        p = os.path.join(path, str(n), f)
        if os.path.isfile(p):
            i = np.loadtxt(p)
            assert i.shape == DIM  # just check the dimensions here
            # i is loaded as a matrix, but we are going to flatten it into a single vector
            images.append(i.flatten())
    return images


# Load up these image files
A = load_image_files(0)
B = load_image_files(1)

N = len(A[0])  # the total size
assert N == DIM[0] * DIM[1]  # just check our sizes to be sure

# set up some random initial weights
weights = np.random.normal(0, 1, size=N)

# Your code here:
NUM_BLOCK = 25

step_func = lambda x: 0 if x < 0 else 1

# This tags image of A & B with their actual value
shuffled_training_data = random.sample(list(zip(A, [0] * len(A))) + list(zip(B, [1] * len(B))), len(A) + len(B))

shuffled_copy = shuffled_training_data

accuracy_list = []
output_list = []


def training_func():
    global weights, shuffled_copy
    accuracy_count = 0

    for i in range(NUM_BLOCK):
        output = 0
        actual_value = shuffled_copy[i][1]
        for n in range(N):
            output += (weights[n] * shuffled_copy[i][0][n])

        output_list.append(output)
        output_list.append(np.dot(shuffled_copy[i][0], weights))

        y = step_func(output)

        if actual_value == 0 and y == 1:
            weights -= shuffled_copy[i][0]
        elif actual_value == 1 and y == 0:
            weights += shuffled_copy[i][0]

        if actual_value == y:
            accuracy_count += 1

    # copies same shuffled copy but without the iterated 25 images.
    shuffled_copy = shuffled_copy[25:]

    accuracy_list.append(accuracy_count / NUM_BLOCK)


# runs the training_func() 100 iterations
i = 0
while True:
    training_func()
    i += 1

    if i == 99:
        break

# reshapes the weight to 28 x 28 plot
print(weights)
reshaped_weights = np.reshape(weights, (28, 28))
labeled_weights = []


# This lables each value by corresponding row & col
for row in range(0, 28):
    for col in range(0, 28):
        labeled_weights.append((reshaped_weights[row][col], "[{}][{}]".format(row + 1, col + 1)))

#sorts the weights in decreasing order from largest to smallest
labeled_weights.sort(reverse=True, key=lambda x: x[0])
print(labeled_weights)

sns.heatmap(reshaped_weights)
plt.title("Image of Weight Vector after Training")
plt.savefig("foo.png")
plt.show()
```

4

In [ ]:
```python
import os
import numpy as np
import random
import matplotlib.pyplot as plt
import seaborn as sns

# Functions that might be useful (please read the documentation)
# x.flatten() (take a N-dimensional numpy array and make it one-dimensional)
# numpy.random.choice -- choose from the list of images
# numpy.dot -- compute the dot product
# numpy.random.normal -- set up random initial weights

DIM = (28, 28)  # these are the dimensions of the image


def load_image_files(n, path="images/"):
    # helper file to help load the images
    # returns a list of numpy vectors
    images = []
    for f in os.listdir(os.path.join(path, str(n))):  # read files in the path
        p = os.path.join(path, str(n), f)
        if os.path.isfile(p):
            i = np.loadtxt(p)
            assert i.shape == DIM  # just check the dimensions here
            # i is loaded as a matrix, but we are going to flatten it into a single vector
            images.append(i.flatten())
    return images


# Load up these image files
A = load_image_files(0)
B = load_image_files(1)

N = len(A[0])  # the total size
assert N == DIM[0] * DIM[1]  # just check our sizes to be sure

# set up some random initial weights
weights = np.random.normal(0, 1, size=N)

# Your code here:
NUM_BLOCK = 25

step_func = lambda x: 0 if x < 0 else 1

# This tags image of A & B with their actual value
shuffled_training_data = random.sample(list(zip(A, [0] * len(A))) + list(zip(B, [1] * len(B))), len(A) + len(B))

shuffled_copy = shuffled_training_data

accuracy_list = []
output_list = []


def training_func():
    global weights, shuffled_copy
    accuracy_count = 0

    for i in range(NUM_BLOCK):
        output = 0
        actual_value = shuffled_copy[i][1]
        for n in range(N):
            output += (weights[n] * shuffled_copy[i][0][n])

        output_list.append(output)
        output_list.append(np.dot(shuffled_copy[i][0], weights))

        y = step_func(output)

        if actual_value == 0 and y == 1:
            weights -= shuffled_copy[i][0]
        elif actual_value == 1 and y == 0:
            weights += shuffled_copy[i][0]

        if actual_value == y:
            accuracy_count += 1

    # copies same shuffled copy but without the iterated 25 images.
    shuffled_copy = shuffled_copy[25:]

    accuracy_list.append(accuracy_count / NUM_BLOCK)


# runs the training_func() 100 iterations
i = 0
while True:
    training_func()
    i += 1

    if i == 99:
        break

# Converts the numpy array to regular array
weights = list(weights)

# Sorts the values in the order from being closest to zero to furthest to zero
sorted_weights = sorted(weights, key=abs)

zeroed_weights_list = []
zeroed_accuracy_list = []
NUM_IMGS = 1000


# Creates a list of zeroed weights list from 10 to 780
def zero_replacer(weights):
    for n in range(78):
        flagger = sorted_weights[0+10*n:10+10*n]
        for i in range(len(weights)):
            if weights[i] in flagger:
                weights[i] = 0
        zeroed_weights_list.append(weights[:])

# Parses through each list of zeroed lists and finds the accuracy for each list.
def accuracy_parser(zeroed_lists):
    global shuffled_copy

    for each in zeroed_lists:
        accuracy_count = 0
        for i in range(NUM_IMGS):
            output = 0
            actual_value = shuffled_copy[i][1]
            for n in range(N):
                output += (each[n] * shuffled_copy[i][0][n])

            y = step_func(output)

            if actual_value == y:
                accuracy_count += 1

        zeroed_accuracy_list.append(accuracy_count / NUM_IMGS)


zero_replacer(weights)
accuracy_parser(zeroed_weights_list)

x = [i * 10 for i in range(1, 79)]
y = zeroed_accuracy_list

axes = plt.gca()
axes.set_ylim([0, 1.1])

plt.plot(x, y, color='orangered', marker="o")
plt.xlabel("Number of weight values replaced by 0")
plt.ylabel("Accuracy ")
plt.title('Correlation b/t number of weight values closest to 0 replaced by 0 and Perceptron Accuracy')

plt.savefig("foo.png")
plt.show()
```

5

In [ ]:
```python
import os
import numpy as np
import random
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib import cm as CM



# Functions that might be useful (please read the documentation)
# x.flatten() (take a N-dimensional numpy array and make it one-dimensional)
# numpy.random.choice -- choose from the list of images
# numpy.dot -- compute the dot product
# numpy.random.normal -- set up random initial weights

DIM = (28, 28)  # these are the dimensions of the image


def load_image_files(n, path="images/"):
    # helper file to help load the images
    # returns a list of numpy vectors
    images = []
    for f in os.listdir(os.path.join(path, str(n))):  # read files in the path
        p = os.path.join(path, str(n), f)
        if os.path.isfile(p):
            i = np.loadtxt(p)
            assert i.shape == DIM  # just check the dimensions here
            # i is loaded as a matrix, but we are going to flatten it into a single vector
            images.append(i.flatten())
    return images


# Load up these image files
N_0 = load_image_files(0)
N_1 = load_image_files(1)
N_2 = load_image_files(2)
N_3 = load_image_files(3)
N_4 = load_image_files(4)
N_5 = load_image_files(5)
N_6 = load_image_files(6)
N_7 = load_image_files(7)
N_8 = load_image_files(8)
N_9 = load_image_files(9)

# Make a list with each corresponding number
whole = [N_0, N_1, N_2, N_3, N_4, N_5, N_6, N_7, N_8, N_9]


N = len(N_0[0])  # the total size
assert N == DIM[0] * DIM[1]  # just check our sizes to be sure

# set up some random initial weights
weights = np.random.normal(0, 1, size=N)

# Your code here:
NUM_BLOCK = 25

step_func = lambda x: 0 if x < 0 else 1

accuracy_list = []
weights_list = []
pair_accuracy_list = []
shuffled_pair_list = []

# Tags image of pairs with their actual value & shuffles them
def img_pair_shuffler(first_num, second_num):
    if first_num is second_num:
        shuffled_training_data = random.sample(list(zip(first_num, [0] * len(first_num))) + list(zip(second_num, [0] *
len(second_num))), len(first_num) + len(second_num))
    else:
        shuffled_training_data = random.sample(list(zip(first_num, [0] * len(first_num))) + list(zip(second_num, [1] *
len(second_num))), len(first_num) + len(second_num))

    return shuffled_training_data

# Creates shuffled pair list in order
def pair_shuffler():
    for m in range(10):
        for n in range(10):
            shuffled = img_pair_shuffler(whole[m], whole[n])
            shuffled_pair_list.append(shuffled)


def training_func(p):
    global weights, shuffled_pair_list, accuracy_list
    accuracy_count = 0

    for i in range(NUM_BLOCK):
        output = 0
        actual_value = shuffled_pair_list[p][i][1]
        for n in range(N):
            output += (weights[n] * shuffled_pair_list[p][i][0][n])

        y = step_func(output)

        if actual_value == 0 and y == 1:
            weights -= shuffled_pair_list[p][i][0]
        elif actual_value == 1 and y == 0:
            weights += shuffled_pair_list[p][i][0]

        if actual_value == y:
            accuracy_count += 1

    # copies same shuffled copy but without the iterated 25 images.
    shuffled_pair_list[p] = shuffled_pair_list[p][25:]

    accuracy_list.append(accuracy_count / NUM_BLOCK)

# Parses through every type of possible pair of digits
def pair_parser():
    global weights, pair_accuracy_list, accuracy_list
    pair_counter = 0

    for m in range(10):
        for n in range(10):
            for c in range(100):
                training_func(pair_counter)
            pair_accuracy_list.append(sum(accuracy_list[:-26:-1])/25)
            pair_counter += 1
            weights = np.random.normal(0, 1, size=N)


pair_shuffler()
pair_parser()



reshaped_pair = np.reshape(pair_accuracy_list, (10, 10))
mask =  np.tri(reshaped_pair.shape[0], k=-1)
reshaped_pair = np.ma.array(reshaped_pair, mask=mask)
reshaped_pair = np.triu(reshaped_pair, k=1)
sns.heatmap(reshaped_pair)
plt.title("Accuracy for each kind of pair of digit shapes")
plt.savefig("foo.png")
plt.show()


# # Say your matrix object (e.g. np.array) is corr_mat

# # Get the upper triangle without the diagonal
# corr_mat = np.triu(corr_mat, k=1)

# # Plot the heatmap
# ax = sns.heatmap(corr_mat)
```

In [ ]: