Assignment #2

Name: **Mohsin Hayat**
Roll Number: **L16-4333**

Artificial Intelligence

CS-D

Submitted To: Sir Mubashar Mirza

MAKING A PROGRAM TO READ HANDWRITTEN NUMBERS: COMPETITION BETWEEN FAMOUS LEARNING BASED AI ALGORITHMS

Abstract

This study aims to create an efficient (fast and accurate) learning based algorithm to determine weights for handwritten digits taken from MNIST dataset. These weights will be used to classify unknown handwritten images. This report compares the result of some learning based algorithms.

1. K-NN Classifiers

In this classifier, K nearest neighbors determines the prediction of an input image. I was supposed to find the best value of K which gives the best test accuracy. For this classification, I'm using cosine similarity and Euclidean distance for computing nearest neighbors of an input image using raw images (pixel values). The "validate_euclidean()" and "validate_cosine()" functions are for validating the classifiers for N number of randomly chosen test images.

1.1. Cosine Similarity

The cosine similarity of two vectors is calculated by calculating dot product of the vectors divided by product of magnitudes of both the vectors. I've done this without the numpy builtin function and it takes 17-20 seconds to compute cosine similarity between a testing image and all training images. Majority voting is used in the function to predict digits.

$$cos\theta = \frac{O_i . O}{|O| . |O_i|}$$

1.2. Euclidean Distance

For two images, Euclidean distance is simply an under-root of sum of squared differences of corresponding intensities of the pixel values. I've done this in steps and this classifier takes 14-15 seconds to compute all distances of a testing image from 60,000 training images. To conclude a prediction, I've used majority voting in Euclidean distances classifier.

$$d = \sqrt{\sum (O_i - O_j)^2}$$

1.3. Results

I've randomly picked 10 images from the testing data and tried to validate the value of K based on the accuracy of the results.

- When K=1

    - Euclidean Distance returns 8 out of 10 correct results. It varies from 7-10 but mostly the result is 8. It takes around 15 seconds to evaluate a single image.
    - Cosine Similarity also returns 8 correct results and takes almost 20 seconds for a single image.
- When K=2
    - Euclidean Distance returns 9 out of 10 correct results.
    - Cosine also returns mostly 9 correct answers.
- When K=3
    - Euclidean Distance returns mostly 10 but the results still contain wrong answer sometimes.
    - Cosine mostly returns 9/10 results.
- When K=4
    - Euclidean Distance returns 10/10 results. I've tested the results in 12-15 iterations and the values are mostly accurate every time. Although the result contains very slight mistakes.
    - The cosine similarity also returns correct values in most iteration.
- When K>=5
    - Euclidean distance classifier is completely accurate
    - Cosine similarity returns 100% results

Valid Value of K

The valid value of K is 5 in the cases that I've tested.

Accuracy Check

For K>=5, I've tested the data set with 100 randomly chosen images and the results are as under:
100 out of 100 were correct in Euclidean distance classifier.
100 out of 100 were correct in Cosine Similarity classifier.

From the observation, this can be concluded that the result is 100% when K>=5.

2. Neural Network Learning

We are going to train a system of ten neurons (i.e. sigmoid unit) each trained to discriminate a single digit from the rest of the digits. We will use the gradient descent algorithm for finding optimal weights of each of the neurons in the system. Both tanh and logistic sigmoid activation functions are used in feed-forward nets.

2.1. Perceptron Learning Rule

Perceptron learning rule uses a Threshold unit to determine labels of unknown images. The value of this perceptron is either (0 or 1) is used to classify an object as

either a positive or a negative instance, in the case of a binary classification problem. This threshold is learned over samples of 60,000 images given to the function. Threshold is an array that contains threshold for each of the digit.
To classify images using threshold, I've written this simple function that returns 1 when an image is correctly classified and 0 otherwise:

```
def classify(obj, i, threshold=0):
    if (np.dot(obj, weights[i]) + B) >= threshold:
        return 1
    else:
        return 0
```

Initially, the threshold is 0 and it makes itself align with the data when many training examples are given. For that, I've implemented this learning function:

```
def learn(digit):
    for i in range(0, num_of_cycles):
        for img, lab in zip(images, labels):
            org = 0
            if lab == digit:
                org = 1
            pred = classify(img, digit, T[digit])
            dif = org - pred
            weights[digit] = weights[digit] + ((S * dif) * np.asarray(img))
            T[digit] -= (S * dif)
```

The learn function takes a digit, that is being trained right now, and calculates the threshold. Num_of_cycles are 100 in my case, these are also called epochs. If the original label matches with the digit being trained, org becomes 1, otherwise 0. Weight is calculated as:

$$W = W + \eta(y - y^*)O_i$$

And threshold is calculated as:

$$T = T - \eta(y - y^*)$$

This is repeated for all 60,000 training images.

Results:
I randomly take 1000 images from the test image data set and count the number of correct classifications from this classifier. After taking 5 tests, I observed that the results vary between **75% and 85%.**

Report:

Confusion Matrix :

[[0 0 0 0 0 0 0 0 0 0]

```
[ 0 10  0  0  0  0  0  0  0  0  0]
[ 0  0 18  0  0  0  0  0  0  0  0]
[ 1  0  0  6  0  0  0  0  0  0  0]
[ 1  0  0  0  5  0  0  0  0  0  0]
[ 2  0  0  0  0  6  0  0  0  0  1]
[ 2  0  0  0  1  0  2  0  0  0  0]
[ 0  0  0  0  0  0  0  8  0  0  0]
[ 3  0  0  0  0  0  0  0  6  0  0]
[ 2  0  0  0  2  0  0  0  0  9  0]
[ 1  0  2  0  0  1  0  0  0  1 10]]
```

Accuracy Score : **0.80**

	precision	recall	f1-score	support
0	1.00	1.00	1.00	10
1	0.90	1.00	0.95	18
2	1.00	0.86	0.92	7
3	0.62	0.83	0.71	6
4	0.86	0.67	0.75	9
5	1.00	0.40	0.57	5
6	1.00	1.00	1.00	8
7	1.00	0.67	0.80	9
8	0.90	0.69	0.78	13
9	0.91	0.67	0.77	15

micro avg	0.80	0.80	0.80	100
macro avg	0.84	0.71	0.75	100
weighted avg	0.92	0.80	0.84	100

2.2. Sigmoid Learning Rule

Second way of learning a perceptron is by using sigmoid activation function and gradient descent. Gradient descent searches for local minima where Error is as less as

possible. This depends on alpha or η. More η means less epochs, so the convergence is done quickly and accurately. Sigmoid function is written as:

```
        def sigmoid (x):
                return 1/(1 + np.exp(-x))
```

And its derivative is written as:

```
        def sigmoid_prime(z):
                return sigmoid(z)*(1-sigmoid(z))
```

To learn weights, I'm using gradient descent for each of the digits. The weight is calculated as:

```
y = sigmoid(np.dot(start_weights.T, x));
dy = sigmoid_prime(np.dot(start_weights.T, x))
start_weights = start_weights + (alpha * (((original_label[j] - y) * dy) * x))
```

Where start_weights (of size 28x28) are initially 0 and then the program repeats to learn gradient using the formula:

$$W = W + \eta(y - y^*) * (f'(y))O_i$$

Where the f'(x) is the derivative of the sigmoid function. It is given above in the sigmoid_prime() function.

Results:

I've tested this to classify 1000 random images and the perceptron accurately classifies around **85% to 95%** of the test images. To predict, I'm using the digit weights and the test image features (28x28) by taking dot product and using sigmoid function to get a value. The maximum value is considered the correct answer.

Report:

For 100 examples, I've computed the report as under:

Confusion Matrix :

```
[[14  0  0  0  0  0  0  0  0  0]
 [ 0 20  0  0  0  0  0  0  0  0]
 [ 0  0  7  0  0  0  0  0  0  0]
 [ 0  0  0  9  0  0  0  0  0  0]
 [ 0  0  0  0  7  0  0  0  0  1]
 [ 0  0  0  0  1  8  0  0  1  0]
 [ 0  0  0  0  0  0  5  0  0  0]
 [ 0  0  1  0  0  0  0 10  0  0]
 [ 1  0  0  0  0  0  0  0  8  0]
```

[0 0 0 0 1 0 0 0 0 6]]

Accuracy Score : **0.94**

	precision	recall	f1-score	support
0	0.93	1.00	0.97	14
1	1.00	1.00	1.00	20
2	0.88	1.00	0.93	7
3	1.00	1.00	1.00	9
4	0.78	0.88	0.82	8
5	1.00	0.80	0.89	10
6	1.00	1.00	1.00	5
7	1.00	0.91	0.95	11
8	0.89	0.89	0.89	9
9	0.86	0.86	0.86	7

micro avg	0.94	0.94	0.94	100
macro avg	0.93	0.93	0.93	100
weighted avg	0.94	0.94	0.94	100

2.3. Tanh Learning

Tanh is also like logistic sigmoid but better. The range of the tanh function is from (-1 to 1). Tanh is binary classifier and is differentiable. I'm using Numpy built in tanh to compute the values.

```
y = np.tanh(np.dot(start_weights.T, x));

start_weights = start_weights + (alpha * (((original_label[j] - y) * (1-y**2)) * x))
```

Where start_weights (of size 28x28) are initially 0 and then the program repeats to learn gradient using the formula:

$$W = W + \eta(y - y^*) * (1 - y^2)O_i$$

Where (1-y^2) is the derivative of the tanh function. Y* is the predicted output and y is the original output.

Results:
I've tested this to classify 1000 random images and the perceptron accurately classifies around **80% to 92%** of the test images. To predict, I'm using the digit weights and the test image features (28x28) by taking dot product and using Numpy's bultin tanh function to get a value. The maximum value is considered the correct answer.

Report:

For 100 examples, I've computed the report as under:

Confusion Matrix:
```
[[ 5  0  0  0  1  0  0  0  0  0]
 [ 0 13  0  0  0  0  0  0  0  0]
 [ 0  0  4  0  0  0  1  0  0  0]
 [ 0  0  0 11  0  0  0  0  0  0]
 [ 0  0  0  0 10  0  0  1  1  2]
 [ 0  0  0  1  0  4  0  0  2  1]
 [ 0  0  1  0  0  0 11  0  1  0]
 [ 0  0  0  0  0  0  0  9  0  1]
 [ 0  1  1  0  0  0  0  0  8  1]
 [ 0  0  0  0  0  0  0  1  1  7]]
```

Accuracy Score : **0.82**

	precision	recall	f1-score	support
0	1.00	0.83	0.91	6
1	0.93	1.00	0.96	13
2	0.67	0.80	0.73	5
3	0.92	1.00	0.96	11
4	0.91	0.71	0.80	14
5	1.00	0.50	0.67	8
6	0.92	0.85	0.88	13

7	0.82	0.90	0.86	10
8	0.62	0.73	0.67	11
9	0.58	0.78	0.67	9

micro avg	0.82	0.82	0.82	100
macro avg	0.84	0.81	0.81	100
weighted avg	0.84	0.82	0.82	100

Comparison with HOG Features:

```
def calculate_hog_features(images):
    list_hog_fd = [hog(t.reshape((28, 28)), orientations=9, pixels_per_cell=(14, 14),
cells_per_block=(1, 1),
                       visualise=False) for t in images]

    return np.array(list_hog_fd, dtype=np.float64)
```

The accuracy is **84%** of the results using Histogram of Oriented Gradients (HOG features) as descriptors/features to represent your digit images and using SVM to classify the digits.

- The classifier made using Perceptron learning rule is worse than the result of HOG. PLR returns 75% to 85% accuracy while HOG always has 83%-90% accuracy.
- The classifier made using sigmoid rule is near to the HOG results. It gives mostly better results than the HOG.
- The tanh classifier is also nearly as accurate as the HOG classifier. It has almost same results.