CSCI 390, Artificial Intelligence

Assignment 9 – Handwritten Digit Classification

In this assignment, you will implement a perceptron classifier. You will test this classifier on a set of scanned handwritten digit images. Even with simple features -which is flatten from 28x28 pixel grayscale image resulting in 785-dimentional vector including bias- your classifiers will be able to perform quite well when given enough training data.

Handwritten digit classification is the task of extracting text from image sources. The data set on which you will run your classifier is a collection of handwritten numerical digits (0-9). The code for this project includes the following files and data, available in Moodle. Remember to keep these files and data in the same directory.

Data file	
`Data`	Contain train and test sets of MNIST digit data in csv
`sample-digit`	Contain examples of 28x28 pixel grayscale images.
File you will edit	
`Perceptron_Lib.py`	Mini library contains methods for perceptron classifier
File you will NOT edit	
`Test_Perceptron.py`	This file is used to test Perceptron implementation

Files to Edit and Submit: You will fill in portions of `Perceptron_Lib.py` only during the assignment, and submit it. You should submit this file with your code and comments. Please *do not* change the other files in this distribution or submit any of our original files other than this file.

DATASET

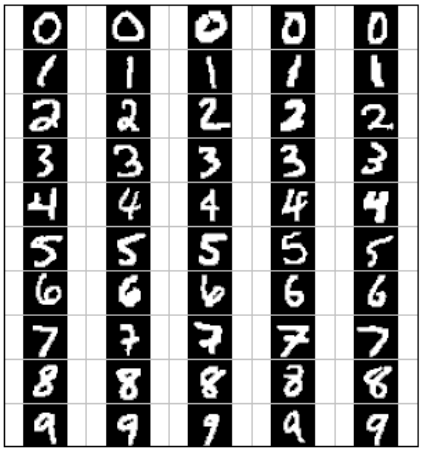

Figure. Examples of grayscale images in MNIST dataset

In this assignment, we use the MNIST dataset of handwritten digits which consist of a training set of 60,000 examples, and a test set of 10,000 examples. You can refer to sample (`png`) images in `sample-digit` folder for better understanding about data. However, you only work on `csv` data stored in `data` folder. The format of csv file is as below:

```
Label, pix-11, pix-12, pix-13,…
```

Where `pix-ij` is the pixel in the *ith* row and *jth* column.

INSTALLATION

For this assignment, you will need to install the following three libraries:

- `numpy`, which provides support for large multi-dimensional arrays
- `sklearn`, which is built upon `numpy`, and features various machine learning functions. In this assignment, we utilize two functions (i.e. `accuracy_score` and `confusion_matrix`) from `sklearn` library to evaluate the accuracy of each classifier.

TASK (5 points): PERCEPTRON IMPLEMENTATION

A skeleton implementation of a perceptron classifier is provided in `Perceptron_Lib.py`. In this part, you will fill in the `get_predictions` and `weight_update` functions.

Perceptron classifier keeps a weight vector $\boldsymbol{w}_y$ of each class y. Given a feature vector $\boldsymbol{x}$, the perceptron compute the class y whose weight vector is most similar to the input vector $\boldsymbol{x}$. Formally, given a feature vector $\boldsymbol{x}$, we score each class with:

$$score(\boldsymbol{x}, \boldsymbol{w}_y) = \boldsymbol{w}_y^T \boldsymbol{x}$$

Then we choose the class with highest score as the predicted label for that data instance.

LEARNING WEIGHTS FOR PERCEPTRON

In the basic multi-class perceptron, we scan over the data, one instance at a time. When we come to an instance $(\boldsymbol{x}, y^*)$, we find the label with highest score:

$$y = \underset{y'}{\operatorname{argmax}}\, score(\boldsymbol{x}, \boldsymbol{w}_{y'})$$

We compare y to the true label y^*. If $y = y^*$, we've gotten the instance correct, and we do nothing. Otherwise, we guessed y but we should have guessed y^*. That means w_{y^*} should have higher score, and w_y should have lower score, in order to prevent this error in the future. We update these two weight vectors accordingly:

$$\begin{aligned} \boldsymbol{w}_{y^*} &\leftarrow \boldsymbol{w}_{y^*} + \boldsymbol{x} \\ \boldsymbol{w}_{y} &\leftarrow \boldsymbol{w}_{y} - \boldsymbol{x} \end{aligned} \tag{1}$$

In actual implementation, we do not keep track of the weights as separate structures, we usually stack them on top of each other to create a weight matrix **W**. This way, instead of doing as many dot products as there are classes, we can instead do a single matrix-vector multiplication. This tends to be much more efficient in practice (because matrix-vector multiplication usually has a highly optimized implementation). Then, for each output, we provide is a 10-dimensional vector which has zeros in all positions, except for a one in the position corresponding to the class of the digit. After that, we map the predicted label y and true label y^* to two 10-dimensional vectors $\boldsymbol{y(x)}$ and $\boldsymbol{t(x)}$, respectively. Hence, to learn weight, we can use the following update rule:

$$\mathbf{W} \leftarrow \mathbf{W} + \eta \boldsymbol{x}(\boldsymbol{t}(\boldsymbol{x}) - \boldsymbol{y}(\boldsymbol{x}))^{\mathrm{T}} \tag{2}$$

Here, η is the learning rate (between 0.001 and 1) to control the magnitude of movement. If $\eta = 1$, update equations (1) and (2) are exactly the same.

We calculate the final grade of this assignment based on accuracy you achieve as following:

- >85%: 5 pts
- 70% - 85%: 4 pts
- 60% - 70%: 3 pts
- 50% - 60%: 2 pts
- <50%: 1 pts

For reference, our implementation consistently achieves an accuracy of 87% on the test data after training for around 5 epochs (i.e. each epoch corresponds to one cycle through the full training dataset).

To test your implementation, run your code with: `python Test_Perceptron.py`