CS802 Assignment 2

Outline

This assignment is worth 25% of the mark for this class. It has two sections:

1. **Perceptron** will require you to implement the perceptron in Python to perform handwriting recognition. [20 marks]
2. **Neural Net** will require you to implement a multi-layer neural net in Python with a variety of activation functions. [20 marks]

On MyPlace you will have to submit two Jupyter notebooks.

Section 1 Perceptron

This section requires you to implement the Perceptron in Python to perform handwriting recognition. A large dataset of hand-written digits is available on MyPlace. A template for the Perceptron class is also available, with the methods not yet implemented.

This section is worth 12.5% of the mark for this class. It has 5 parts totalling **20 marks**:

1. (6 marks) Complete the implementation of the Perceptron and use it to identify the digit "7".
2. (4 marks) Update the perceptron implementation to use *batch learning*.
3. (4 marks) Use multiple nodes to classify every handwritten digit.
4. (3 marks) Update the perceptron to use the sigmoid activation function.
5. (3 marks) Add a method to print the weights of the perceptron and view the MNIST data.

Each part is described below. Note that you might like to begin part 5 first, so that you can enjoy the printed data as you work.

On MyPlace you will have to submit a single Jupyter notebook for this section.

1.1. Complete the implementation of the Perceptron.

The first part is to complete the implementation of the perceptron so that you can use it to recognise one digit. The following objectives must be met:

- The perceptron is implemented as a class (you can start from the template on MyPlace).
- A separate cell creates a Perceptron.
- The *mnist_train.csv* file is loaded and used to train the node.
- The *mnist_test.csv* file is loaded and used to test the node, printing the *accuracy*, *precision*, and *recall*.

You can follow the instructions on the lecture slides to help with this part of the coursework.

The output from your script should look something like:

```
Loading data...
Training...
Testing...
Accuracy:         0.9896
Precision:         0.973002159827
Recall:         0.919387755102
```

The exact details are up to you.

1.2. Update the perceptron implementation to use batch learning

In this part of the coursework you need to add a new method to the perceptron called **train_batch**. The batch training method will perform a prediction on every element from the training data set, and then update the weights once. The pseudocode for this is:

```
for i in range(max_iterations):
    weight_update = [0,0,...,0]
    for d in training_data:
        prediction = predict(d)
        weight_update += learning rule
    end for
    weights += weight_update / total_samples
end for
```

The new training method will update the weights by the average gradient vector over all of the elements in the training data. *Do not delete the original training method.*

1.3. Use multiple nodes to classify every handwritten digit.

The third part of this section is to create a new cell that does the following things:

- Creates 10 nodes, one for each digit.
- Trains all of the nodes so that they recognise the digits 1-10.
- Tests all of the nodes.
- Iterates through the testing set and prints the prediction.

For the final part, the pseudocode could look as follows:

```
for d in testing_data:
    prediction = predict(d)
    print(prediction)
    print(d)
end for
```

Where *print(d)* displays the correct label.

1.4. Use the sigmoid activation function.

In this part you must add a new method to the perceptron class called **predict_sigmoid** that performs a prediction using the sigmoid activation function:

$$\sigma(a) = \frac{1}{1 + e^{-a}}$$

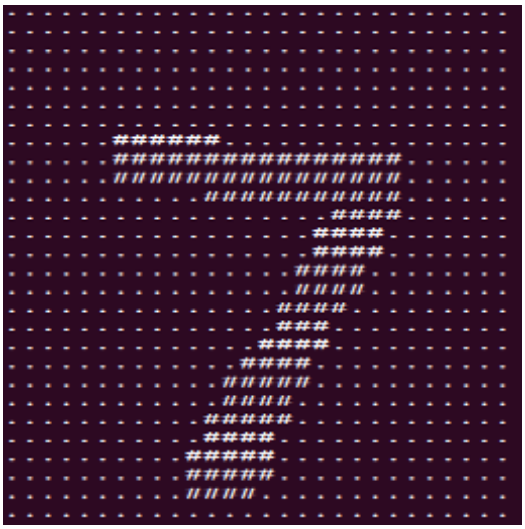
Do not delete the original prediction method.

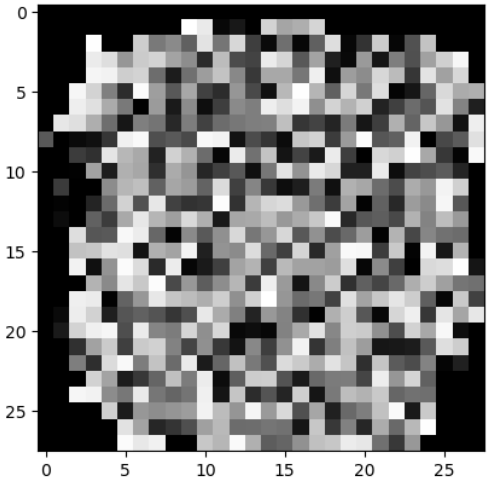
1.5. Print weights and MNIST data.

In this part you must implement some code to display the MNIST data and weights of the perceptron.

- For the MNIST data, you can follow the notes in the lecture slides to print the data to the screen as characters.
- For the weights, you could use matplotlib to plot the data using colours, or grayscale.

How you display the data is up to you. Possible visuals could look like this:





Section 2 Neural Nets

This section requires you to implement a multi-layer neural net in python. The same large dataset of hand-written digits can be used from section 1, which is available on MyPlace. A template file is also available, with the methods not yet implemented.

This coursework is worth 12.5% of the mark for this class. It has 5 parts worth a total of **20 marks**:

1. (3 marks) Complete the implementation of the main method.
2. (5 marks) Complete the initialisation of the neural net.
3. (5 marks) Complete the training implementation.
4. (3 marks) Complete the testing implementation.
5. (4 marks) Implement the rectifier activation function.

Each part is described below.

On MyPlace you will have to submit a single Jupyter notebook for this section.

2.1. Complete the implementation of the main method.

The first part is to complete the implementation of the template's main method. This should be straightforward, with the objectives:

- Loading the training and testing data;
- Creating an instance of the net; and
- Calling the training and testing methods.

You can follow the instructions on the lecture slides to help with this part of the coursework.

Note: you may choose not to use the template python file on MyPlace and start from scratch, as long as the three objectives above are completed.

2.2. Complete the initialisation of the neural net.

In this part you need to complete the implementation of the **init** method. This method constructs the neural net, and the following objectives must be met:

- The learning rate and number of iterations are both set by parameter.
- The number of layers is set by parameter.
- The number of nodes in each hidden layer is set by parameter.
- The number of nodes in the output layer is set by parameter.
- Weights (and biases) are created and initialised for each layer.

2.3. Complete the training implementation.

In this part you need to complete the training method. The training method should use online *stochastic gradient descent*. The pseudocode for this is as follows:

```
for i in range(max_iterations):
    shuffle(training_data)
    for d in training_data:
        calculate_forward_phase
        # backpropagation
        for l in layers_reversed:
            calculate_error_term
            partial_derivative = error_term * previous_layer_output
        end for
        weights = weights - partial_derivatives * learning_rate
    end for
end for
```

In order to complete this method, you will also need to complete the activation function method. This should be kept as a separate method called **activate**. In a later section you will add alternative activation functions.

As training and testing data, you should use the MNIST data set from section 1. A smaller training set for the purpose of testing the implementation is available on MyPlace. It contains the first 10 rows of the full training set.

2.4. Complete the testing implementation.

In this part you need to complete the testing method. This testing method should resemble the training method for the perceptron:

- The forward phase should be calculated for each item in the testing set.
- The accuracy, precision, and recall should be printed.

Test your implementation on the full training and testing sets. The following objective should be met:

- Better than 95% accuracy.

In a markdown cell, comment on the following questions:

- How much better are the results for digit recognition, compared to the single-layer perceptron?
- How did you modify the initial weights, learning rate, and iterations to achieve this?
- How much faster/slower is the training time, compared to the single-layer perceptron?
- How much quicker/slower does the learning converge, compared to the single-layer perceptron?

2.5. Implement the rectifier activation function.

In this part you will update the neural net to use an alternative activation function. The function is:

$$r(a) = \begin{cases} 0, & \text{if } a \leq 0 \\ a, & \text{otherwise} \end{cases}$$

It's derivative should be implemented as:

$$\frac{\partial r(a)}{\partial a} = \begin{cases} 0, & \text{if } a \leq 0 \\ 1, & \text{otherwise} \end{cases}$$

The following parts of the training and testing methods will need to be updated:

- The forward phase should be updated to use the alternate activation function.
- Backpropagation should be updated to use the alternative derivative.

The following objective should be met:

- Better than 95% accuracy.

In a markdown cell, comment on the following questions:

- How much better are the results for digit recognition, compared to the sigmoid activation function?
- How much quicker/slower does the learning converge, compared to the sigmoid activation function?
- How did you modify the initial weights, learning rate, and iterations to achieve this?