**Assignment 4 Report**

Georgios Giovanis

301254694

2019.12.07

**Task 2.2.1**

To complete the first task of learning the function that generated the data shown in **Figure 1** (left), I implemented the compute\_activations(), compute\_gradients(), and update\_weights() functions of the DenseLayer class in layers.py and the compute\_activations(), compute\_gradients(), and update\_weights() functions of the NeuralNet class within neural\_network.py:

*NOTE: numpy matrix operations were utilized in all of the calculations mentioned below to improve computational efficiency.*

1. For compute\_activation() I apply the formula *oi­ = x.wi + bi* *to* every output neuron *i*. I used numpy.dot to calculate the dot product of *x* and *w­i* and numpy.add to add the result to *bi*.
2. For compute\_gradient(), I compute the gradients of loss with respect to the layers’ parameters *b* (biases), *w* (weights), and *x* (inputs). To calculate , which specifies the gradient of loss with respect to the weights *w*, I follow the formula provided in layers.py: The dot product of *x*T and gives the desired result. I calculate , using 1⃗, where 1⃗ is a row of ones, and by taking the dot product of and *w*T.
3. For update\_weights(), I update the layers *w* and *b* parameters by applying the following formulae , and , where is the learning rate.
4. compute\_activations(), loops through every layer in self.\_layers and calculates the activation of the layer using the previous layers output as the current layers input, with the exception of the first layer, which uses the input *x* as its input. After this, the loss is calculated by obtaining the activation of the L2LossLayer. The output of the last layer along with the loss is then returned.
5. For compute\_gradients(), the gradient of all weights with respect to the error is computed. To do this, I first compute the gradient of the loss by making a call to self.loss.compute\_gradient(). I then initialize the variable prev\_gradient with the computed loss value. Back propagation is then performed by looping through all of the previous layers in reverse. During each iteration, a call to set\_output\_error\_gradient(prev\_gradient) is made to set the current layer’s output\_error\_gradient to the previous layer’s input\_error\_gradient. Compute\_gradient() is then called to compute the input\_error\_gradient of the current layer, and the variable prev\_gradient is then updated with the current layer’s input\_error\_gradient.
6. ­For update\_weights() I update the weights of each layer by simply iterating through each layer and calling update\_weights(learning\_rate) for that layer.

When I was passing all of the tests within test\_layers.py and test\_neural\_network.py, I moved on to **Task 2.2.2**, which is discussed on the next page.

**Task 2.2.2**

To create and train the multi-layer perceptron (MLP), I simply ran toy\_example\_regressor.py. The last line of output indicating the validation loss of the MLP after 1000 epochs was 0.011963719361285076, and can be seen in **Table 1**. Based on this value, I concluded that the MLP was predicting a function g(x) very similar to the desired f(x) = x2. I corroborated this finding by analyzing the predicted function overlay in image data\_function.png. **Figure 1** contrasts the plot of original data points (left) to the predicted function g(x) (right, shown in red). As can be seen, the predicted function is similar to f(x) = x2 but is not an exact representation. The slight difference between f(x) and g(x) is most apparent at the outer limits of the function , where 12 and (-1)2 are plotted with values closer to 0.9 than

A screenshot of a cell phone

Description automatically generated1.0.

**Figure 1 - Data and MLP Predicted Function**. Raw input data shown on the left, MLP predicted function g(x) shown on the right in red.

**Task 3.3**

The main objective of this task was to run and read/understand the prime\_classifier.py code. The code within prime\_classifier.py contains a declaration of the PrimeNet class, which is a 2-layer MLP, where each layer contains a dense layer and a sigmoid activation layer. The neural network is trained for 50 epochs on the MNIST dataset, which contains 28 x 28 pixel greyscale images of handwritten numerals. Each data point is labelled with a 1 if the image contains a prime digit, or a 0 otherwise.

The output I obtained by running prime\_classifier.py can be seen in **Table 1**. A validation loss of 0.02454812, and validation accuracy of 97.08% was obtained. A 97.08% success rate is nowhere near the level required for automated systems that perform text recognition, but clearly shows that the neural network was able to learn to identify prime numbers with a high level of accuracy.

**Table 1 – Training Results**

|  |  |  |  |
| --- | --- | --- | --- |
| Test file | # Epochs | Validation Loss | Validation Accuracy |
| toy\_example\_regressor.py | 1000 | 0.011963719361285076 | - |
| Prime\_classifier.py | 50 | 0.02454812 | 97.08% |

**Reflection**

This assignment took me about 20 hours to complete. I enjoyed learning how a neural network is trained. Things I might improve are more exposure to the neural network structure: Is it possible to fully implement a basic neural network from scratch as an assignment? If so, that would be a really cool learning experience.