CMPT310-HW4

Detecting Handwritten Prime Digits with Neural Networks

November 20, 2019

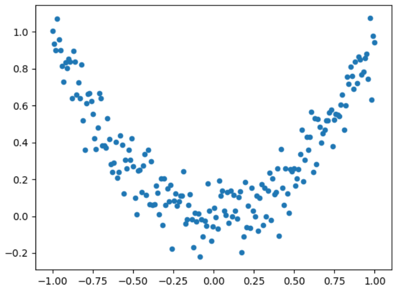
100 points

1 Introduction

Prime numbers are fantastic. In this assignment we will use Multi Layer Perceptrons to detect handwritten prime digits. But before doing such a difficult task I suggest to try and solve an easier problem with MLP. If you succeed, which I know you will, you can proceed with tackling the challenging problem of detecting handwritten prime digits.

2 Regression - Toy Data

The first task is to learn the function that generated the following data, using a simple neural network.



The function that produced this data is actually $y = x^2 + \epsilon$, where $\epsilon \sim N(0, \sigma)$ is a random noise from a normal distribution with a small variance. We are going to use an MLP with one hidden layer, which each has 5 neurons, to learn an approximation of this function using the data that we have. This assignment comes with a starting code, which is incomplete and you are supposed to complete it.

2.1 Technical Details

2.1.1 Code

The code that comes with this assignment has multiple files including:

```
assignment
      |__toy_example_regressor.py
      |__layers.py
      |__neural_network.py
      |__utils.py
```

- `toy_example_regressor.py` contains most of the codes that are related to the training procedure, including loading data and iteratively feeding mini-batches of data to the neural network, plotting the approximated function and the data, etc. Please read this file and understand it but you don't need to modify this.
- `layers.py` contains definition of the layers that we use in this assignment, including `DenseLayer`, `SigmoidLayer`, and `L2LossLayer`. *Your main responsibility is to implement* `forward` and `backward` *functions for these layers.*
- `neural_network.py` contains the definition of a neural network (`NeuralNet` class), which is an abstract class. This class basically takes care of running forward pass and propagating the gradients, backwards, from loss to the first layer.
- `utils.py` contains some useful functions.

2.1.2 Data

The training data for this problem, which consists of input data and labels, can be generated by the function `get_data()`, which you can find in the main file, `toy_example_regressor.py`.

2.1.3 Network Structure

For the regression problem (i.e. the first task) we defined a new class, `SimpleNet` , which is inherited from `NeuralNet`. `SimpleNet` contains two `DenseLayer` s, which one of them has hidden neurons with `Sigmoid` activation functions. Network definition can be found in `toy_example_regressor.py`.

2.2 Your Task (total: 80 points)

2.2.1 Implementing `compute_activations`, `compute_gradients`, and `update_weights` functions

There are three type of layers completely implemented in the `layers.py` file: `DenseLayer`, `Sigmoid`, and `L2LossLayer`. However, implementation of `DenseLayer` is incomplete. You are supposed to implement the following functions

- `DenseLayer`: This is a simple dense (or linear or fully connected) layer that has two types of parameters: weights `w` and biases `b`.
  - `compute_activations` (15 points): The value of every output neuron is $o_i = x.w_i + b_i$. The number of input and output neurons are specified in the `__init__` function.
  - `compute_gradients` (20 points): Assume that gradient of the loss with respect to the output neurons, `self._output_error_gradient` , are computed by the next layer already. You need to compute the gradients of the loss with respect to all the parameters of this layer (i.e. b and w) and store them in `self.dw` and `self.db` so that you can use them the update the parameters later. Needless to say that shape of `dw` and `w` should be equal, and same goes for `db` and `b`. In addition, you should compute the gradient of the loss with respect to the input, which is the output of the previous layer, and store it in `self._input_error_gradient`. This value will be passed on to the previous layer in the network, which will be used to compute the gradients recursively (Back Propagation).
  - `update_weights` (10 points): You should perform Stochastic Gradient Descent and update the weights using the current weights, gradients, and the given learning rate ($new_weights = current_weights - learning_rate * gradient$)

You can refer to the implementations of `Sigmoid` and `L2LossLayer`. **Note: It's up to you how to implement these functions. However, it would be computationally less expensive if you use `numpy` matrix operations to compute the value of the neurons or gradients.**

Your next task is to implement same functions for the `NeuralNet` class.

- `compute_activations` (10 points): Iterate over all layers starting from the first one and compute the activations for each layer. At the end return the output of the last layer along with the value of loss.
- `compute_gradients` (10 points): You are supposed to perform back propagation. In other words, first compute the gradient of the lass layer and pass it to the last layer. Then starting from the last layer iterate over all layers backwards, first compute its gradient and then pass it to the previous layer.

- `update_weights` (5 points): You should update the parameters of all the layers (i.e. those who have parameters). You can use the `update_weight` function of each layer

2.2.2 Training the model (10 points)

Once you are done with implementing and testing the correctness of the implemented functions, you are ready to build a multi layer Perceptron and train it.

There is already an existing starter code for you at `toy_example_regressor.py`. It's a script that contains definition of a simple two layer MLP with scripts that loads the data and trains the MLP. At the end of the training the code plots data and the approximated function. Also the network weights will be saved to a file with this name `simple_net_weights-{timestamp}.pkl`. **You should change its name to `simple_net_weights.pkl` and include it in your submission.**. The timestamp is added to make sure you don't overwrite some previously well trained model.

In addition, you should check the loss and the saved image. Check if the predicted function is similar to $f(x) = x^2$ and matches the validation data. **You should include the saved image, `data_function.png`, both in your report and in your submission.**

Note that this is a regression task, so the last layer of the MLP only has one neuron without any activation functions.

3 Detecting Prime Digits

Now we can use the same layers to distinguish one digit prime numbers (i.e. 2,3,5, and 7) from one digit composite numbers (i.e. 1, 4, 6, 8, and 9). This is a binary classification problem. So the MLP that we are going to use will have only one neuron in the last layer with a sigmoid activation function.

3.1 Data

The dataset that you will be using for this task is the MNIST dataset, which contains gray scale images of hand written digits. Sizes of images are 28 by 28 pixels. We have already preprocessed it for you. We have set the labels for prime digits to 1, and 0 otherwise. Also we have normalized the values of pixels (i.e. pixel values are in the interval of $[0, 1]$).

3.2 Network Structure

The input of the network is a vector with 784 neurons ($28 * 28$) The network has one hidden layer with 20 neurons with sigmoid activation functions. The output of the network is one neuron with sigmoid activation function.

3.3 Your Task (20 points)

For this task, all you need to do is to read and understand the `prime_classifier.py` code and then run it. Over the course of training, loss values and accuracies on the validation set is printed. At the end of the training, network parameters will be saved in a file with the following format `prime_net_weights-{timestamp}.pkl`. **You should change its name to `prime_net_weights.pkl` and include it in your submission.**. The timestamp is added to make sure you don't overwrite some previously well trained model.

4 Testing Your Code

To help you with testing your code, a number of tester files have been included. You can use them to test your implementations. For example if you run:

```
$ python test_layers.py
```

You can see if the three functions (`DenseLayer.compute_activations()`, `DenseLayer.compute_gradients()`, and `DenseLayer.update_weights()`) that you implemented in `layers.py` file work properly or not. We recommend to use all the four test files that are included:

```
testers
    ├── test_layers.py
    ├── test_neural_network.py
    ├── test_toy_example_regressor.py
    └── test_prime_classifier.py
```

Also, you can **estimate** the total points that you might get for this assignment by running the following code:

```
$ python evaluate_assignment.py
```

Note: We will use stronger test cases to test your code and grade your assignment. So passing these tests does not guarantee anything. These tests are only meant to help you with this assignment.

5 What to submit

You should include all the following files in a `tar.gz` or `zip` file with your student id (either `YOUR_STUDENT_ID.tar.gz` or `YOUR_STUDENT_ID.zip`).

1. Your code. Do not change the signature of the functions that you were supposed to implement. Do **NOT** include the dataset (`assignment4-mnist.pkl`)
2. You should include the following files that are automatically saved in your submission:
    - `prime_net_weights.pkl`
    - `simple_net_weights.pkl`

- `data_function.png`

3. A `report.pdf` file concisely explaining what you did in this assignment. Also in your report include your model's loss (for both problems) and accuracy (only for prime digit detection).

Note: your code will be evaluated with an automated script. So if you don't follow the above steps, you will lose all or a portion of your points for this assignment.