ROB313 Assignment 3

Alvin Pane [1004281118]

March 26, 2020

Objectives

The objective of this assignment was to study use cases of logistic regression models and neural networks. Logistic regression models were studied through the application of gradient descent with both full-batch and mini-batch variants. The performance of the two variants was compared by analyzing the test accuracy ratio and the test log-likelihood. The loss, negative log-likelihood was plotted against iteration number to study the convergence trends as iterations increase. The second part of this assignment introduced autograd in the implementation of a neural network to make classifications on the dataset mnist_small. The implemented neural network had a log-softmax output layer with 10 outputs, and was trained using stochastic gradient descent with a mini-batch size of 250. Considering 100 neurons per hidden layer, plots of negative log-likelihood vs. iterations were produced for the training and validation sets to analyze the difference in performance. Lastly, a few data entries with low confidence in their classification were plotted in order to gain insight into potential pitfalls of the algorithm.

Code Structure

The code was designed in a way to optimize the experience for the person running the code. The main strategy was to make the code modular. This was achieved by defining different functions, each responsible for handling a small task. Where applicable, general functions were written such that they could be called multiple times with different inputs. This was in the interest of space efficiency. The code uses print statements and plots frequently to present data to the user in a clear manner. The main section calls functions depending on what question is to be answered. There are 2 variables: Q1 and Q2, initialized to False. To run a question, simply set the variable equal to True.

- Q1 uses 2 functions, Q1f and plot. Q1f performs full-batch gradient descent and stochastic gradient descent, and returns the test accuracy ratio, the test log-likelihood, as well as a list of log-likelihood values. This list is then passed into the plot function, which plots the values against iteration number.
- Q2 uses 4 functions; plot_digit, forward_pass, negative_log_likelihood, and run_sgd. Plot_digit is used in order to form grayscale plots of digits from the mnist data set. Forward_pass computes the forward pass of the two-layer neural network, with a numerically stable log-softmax activation function on the output layer. Negative_log_likelihood computes the negative log likelihood of the neural network, and run_sgd computes SGD estimates of the negative log-likelihoods over 7500 iterations, and then produces a plot of this.

Q1

Gradient descent techniques were used to learn the weights of a logistics regression model. The Bernoulli likelihood with binary classification was trained using 2 methods, full-batch gradient descent [GD] and stochastic gradient descent [SGD] with a mini-batch size of 1. This model was trained be maximizing the log-likelihood derived from the Bernoulli function. Inferences were made using the sigmoid function:

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

This sigmoid function returns a number on the range 0 to 1, which represents the conditional probability of a data point being Class 1. Figure 1 and 2 show plots of the negative log-likelihood which was computed over 7500 iterations. 3 learning rates were used; 0.01, 0.001, and 0.0005. Table 1 summarizes the results for test negative log-likelihood and test accuracy ratio for full-batch GD and SGD with a mini-batch size of 1. In the plots, we see that for GD, the learning rates converge fairly quickly, at around 3000 iterations. For GD, however, it takes much longer for the rates to converge, and by 7500 iterations, they still have not done so. To determine how long it takes for SGD to converge, the number of iterations can simply be increased in the code, at the expense of run time, however. To obtain the max likelihood estimate [MLE], we need to maximize the log-likelihood function. This is equivalent to finding the minimum negative log-likelihood, which is in this case **0.01**.

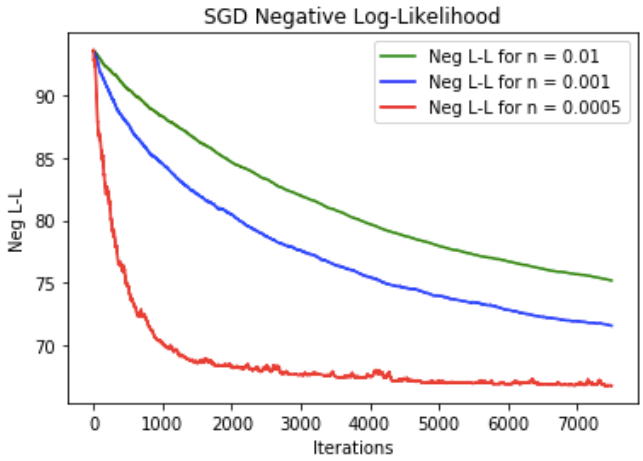

Figure 1: Negative Log-Likelihood vs Iterations (SGD)

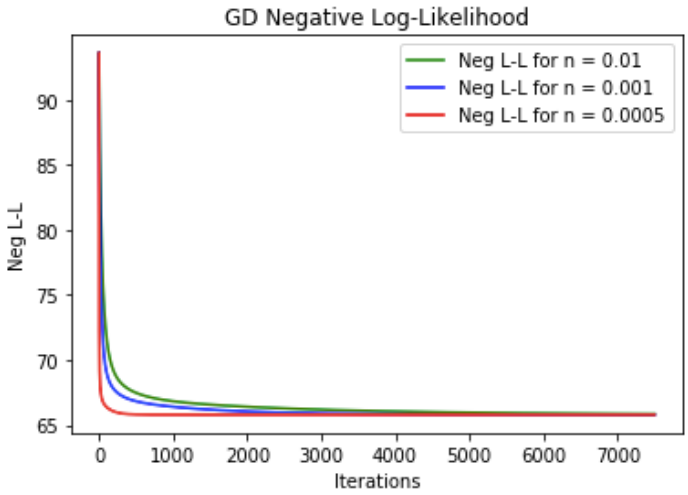

Figure 2: Negative Log-Likelihood vs Iterations (GD)

Table 1: Results for Q1

	Test Negative Log-Likelihood	**Test Accuracy Ratio**
Gradient Descent (Full-Batch)	6.974	0.734
SGD with Mini-Batch Size=1	8.282	0.734

a) If the model predicts Class 1 with full certainty but the correct label is Class 0, one of the terms in the second part of the expression for log-likelihood would contain a log(0) which is not defined:

$$\log(1-\sigma(z)) = \log(0) = \text{undefined}$$

Hence the log-likelihood would be undefined in this case. This behavior is not reasonable as it is a sign that the model we are using is overfitting, and thus making incorrect classifications with a high certainty. $\sigma(z) = 1$, given our sigmoid function in this case, implies that z tends towards infinity. This then implies that our selected weights also tend to infinity, thus showing overfitting.

b) The test log-likelihood is a preferable performance metric as it looks at how correct or incorrect a prediction may be through leveraging conditional probability from the sigmoid function. On the other hand, test accuracy ratio is only a ratio of correct classifications to model predictions. This is less preferable as the sigmoid functions makes classifications in a range between 0 and 1. Test ratio does not allow us to leverage the information of where exactly on the range a classification lies. With log-likelihood we can use this information as a better indicator to the true accuracy of our model. In Table 1 we see that both models had the same test ratio, but different log-likelihoods.

Q2

a) The log-softmax function implemented here is:

$$logsoftmax = (x - b) - \log\left(\sum_i (\exp(x_i - b))\right)$$

With $b=\max(x_i)$. This activation function has had the LogSumExp trick applied to it in order to make it numerically stable:

$$logsoftmax = \log\left(\frac{\exp(x)}{\sum_i(\exp(x_i))}\right)$$

$$logsoftmax = \log\left(\frac{\exp(x-b)\exp(b)}{\sum_i(\exp(x_i - b))\exp(b)}\right)$$

$$logsoftmax = (x - b) - \log\left(\sum_i (\exp(x_i - b))\right)$$

Since we have set $b=\max(x_i)$, we ensure that the equation has numerical stability for both overflow and underflow situations. This is because the term, (x – b), is now bounded by the max value of x_i, and cannot grow too small or too large as opposed to the previously unbounded x. Additionally, the equation ensures that we will never run into a log(0)=undefined situation.

b) The modifications made to the function negative_log_likelihood(…) were such that the negative log-likelihood would be returned. The modifications are as follows:

```
def negative_log_likelihood(X,W1,W2,W3,b1,b2,b3,Y):
    prediction=forward_pass(X,W1, W2, W3, b1, b2, b3)
    return -np.sum(np.multiply(prediction,Y))
```

First, a prediction is formed by running a forward pass through the neural network. The outputs of the forward pass function are class-conditional log probabilities. Then, in order to return the negative log-likelihood, the array is multiplied with the input array for Y, and then summed (the dot product of predictions with Y). Lastly the expression is multiplied by -1 to return the negative.

c) Considering 100 neurons per hidden layer and a mini-batch size of 250, the negative log-likelihoods at various iteration numbers were computed for the training set and validation set. A learning rate of 0.0005 was used. These values are plotted in Figure 3. The Accuracy Ratio was also computed, yielding a result of **0.952**, which suggests that the neural network was quite accurate in its performance.

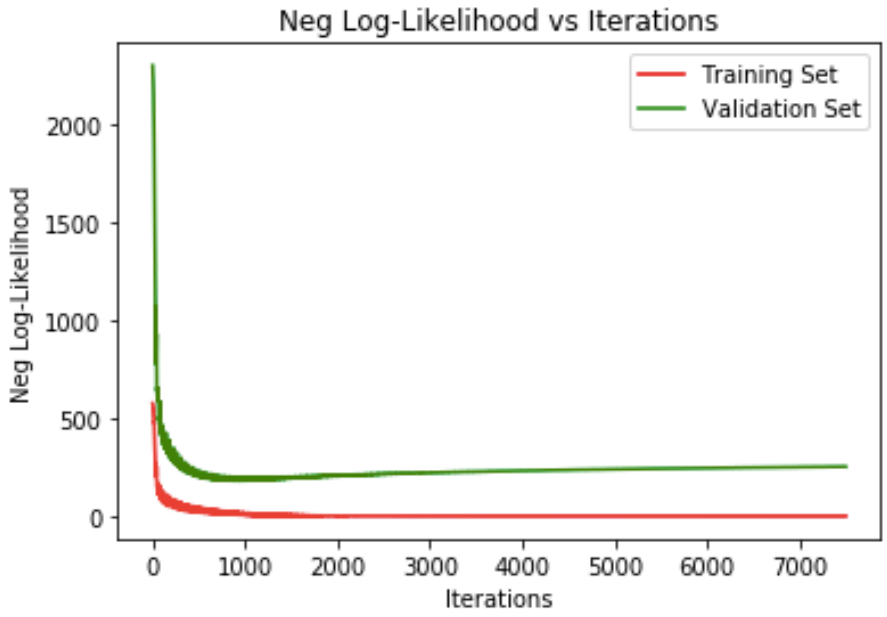

Figure 3: Negative Log-Likelihood vs Iterations for Training and Validation Set

From Figure 3, the values for the training and validation sets both follow the same initial trend, decreasing as iteration size increases. However, the training log-likelihood [L-L] begins at a much lower value. Additionally, the validation log-likelihood [L-L] plateaus at an earlier iteration number, while the training L-L continue to decrease over the entire duration of the plot. As the iteration number gets very large, the validation line tends to increase again, which is a potential sign that the neural network is overfitting the training set.

d) Predictions were made on the test set and sorted based off their class conditional probability, Figures 4-9 show a few of the test set digits with the lowest classification outputs.

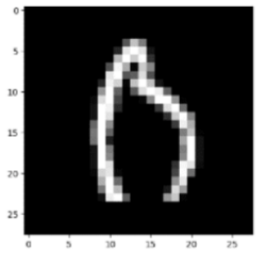

Figure 4: Digit Unknown

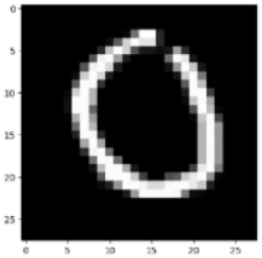

Figure 5: Digit Zero

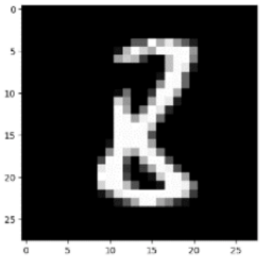

Figure 6: Digit Eight

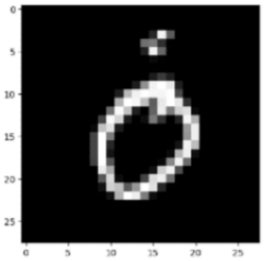

Figure 7: Digit Zero

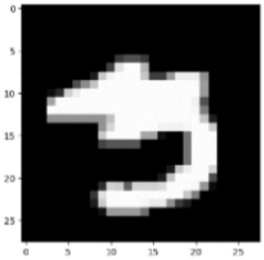

Figure 8: Digit Three

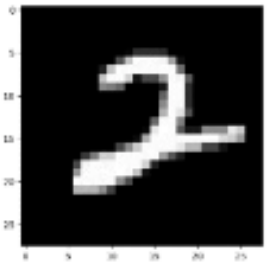

Figure 9: Digit Two

From the above figures, we can see that these lowly classified digits have certain irregularities. Figures 4, 7 and 8 have heavy deformation, and it makes sense that the neural network was not able to classify these with a high degree of certainty. However, with Figures 5 and 6, we can see a trend. Both of these figures clearly show the digits 0 and 8, respectively, albeit with a discontinuous loop. This discontinuity, while subtle to the human eye, seems to really throw off the network. This is an indicator that the model may have issues with classifying when a certain, subtle defining feature is missing. For example, the loop continuity in the digits 0 and 8 may have been a defining feature that when lacked, leads to the low confidence classification we see here. Figure 9 may also be an indicator of a similar occurrence. It clearly shows the number 2 and yet the network struggled to identify this. It is likely that this style of number 2 shares certain defining features with other digits, making it difficult for the network to classify with high probability. This perhaps can be improved by modifying our algorithm so that it can also be trained on some data with irregular digits, such that it may learn new defining features to classify digits with deformities.