COMP3221 - Assignment 2 Report

System Overview:

In this assignment, we created a federated learning distributed system between a server and 5 clients. We simulated each process using 6 individual terminals to create and run the network.

The Federated system is implemented using pytorch which provides many of the features for the program such as the MCLR model and other tools. Initially after the program starts, the server will wait for 30 seconds after the first handshake to begin communication rounds.

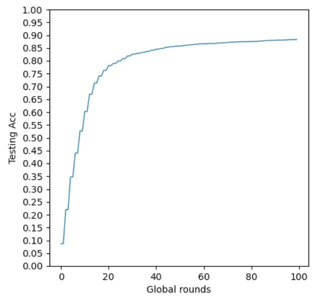
As the MCLR model is an object class, it needs to be converted to a binary string in order to be broadcast, this is done using the pickle library. Additionally, the server models were significantly larger than 1024 bytes which exceeded the limit for socket buffers hence communication between server and client were achieved through multiple packets.

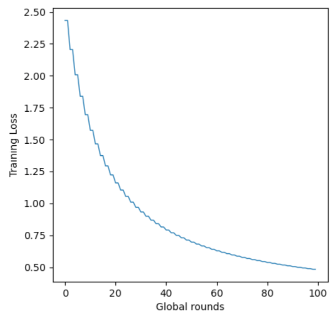
The machine learning model used is a multinominal logarithmic regression model. It contains one input layer, one output layer, and no hidden layers. The input layer is a simple linear layer, its purpose is to transform the input features into values for the output layer, squishing all the features into 10 values (since there are 10 classes). The output layer is a log_softmax layer. As defined in the torch documentation, it is the equivalent of applying a softmax function to the inputs of the function, followed by a logarithmic function. The log softmax function's purpose is to define a percentage value for each class, this network picks the class with the highest value

When training, the loss function used is the negative log likelihood loss, a commonly used function when training classification problems with multiple classes. The optimiser used is the stochastic gradient descent optimiser. Adjusting the batch size in this optimiser allows us to implement regular GD and mini batch GD.

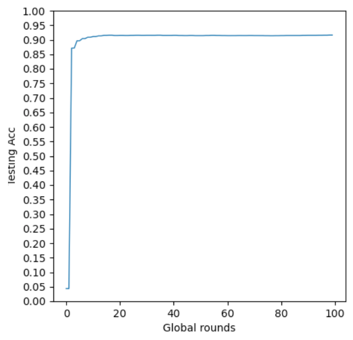
Results

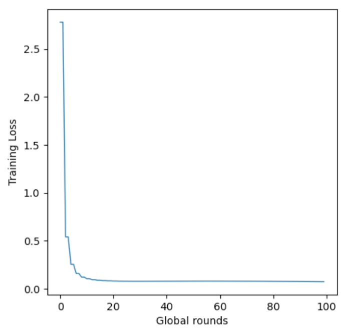
Gradient Descent (no subsampling):



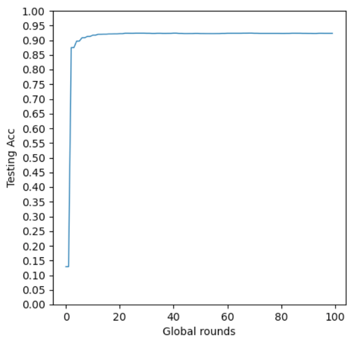


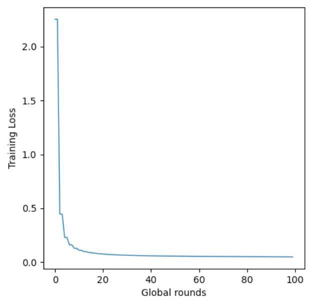
Mini Batch Gradient Descent (Batch size = 5, no subsampling):



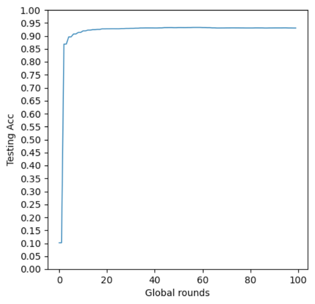


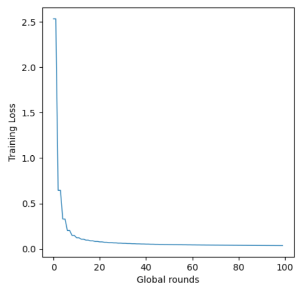
Mini Batch Gradient Descent (Batch size = 10, no subsampling):



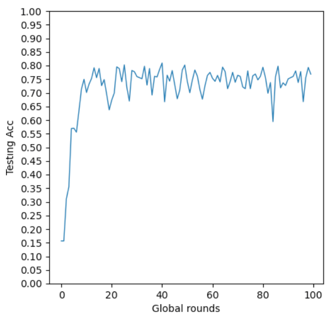


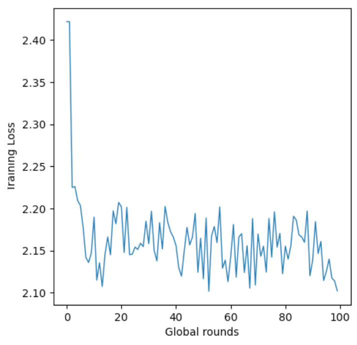
Mini Batch Gradient Descent (Batch size = 20, no subsampling):



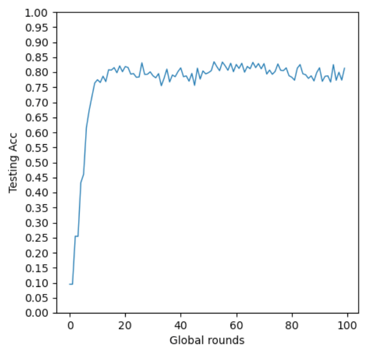


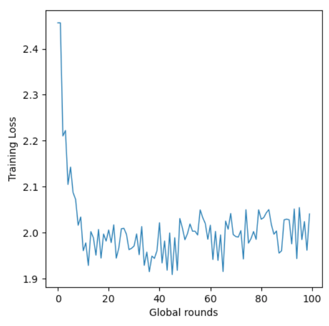
Gradient Descent (subsampling rate = 2):





Gradient Descent (subsampling rate = 3):





Discussion

When evaluating the performance of each type of model, it is important to note that every model achieved performance of above 70%. The variance of the final accuracy of each model remained in the 75% - 95% test accuracy range, meaning that the differences in models discussed will be minimal. I.e. Every model performed decently on the test set and could be applicable to a real world scenario.

That being said, there are obvious differences between each model.

Firstly, the difference between regular gradient descent and mini batch gradient descent can be seen in the curvature of both the test accuracy and train loss graphs. For example, the test accuracy of the mini batch gradient descent with batch size = 5 reached its elbow peak of 90% on its first few iterations (within the first 5-10 iterations), whereas the test accuracy of the regular gradient descent model reached its elbow of about 80% around the 20th iteration. Not only does this mean that mini batch gradient descent achieved a higher overall accuracy than regular gradient descent, it did so in less global iterations. The exact same property can be observed with the train loss graphs. This may imply that using mini batch gradient descent is much more efficient in terms of computing power, time and accuracy.

When comparing the accuracy and loss graphs within mini batch gradient descent models of different sizes, we notice that there is not much difference in the curvature of each graph. Similarly, each graph achieved a final accuracy of approximately 95%, as opposed to regular gradient descent which reached an accuracy just below 90%. There are two conclusions we can draw from this experiment. Firstly, the batch sizes tested (5, 10 and 20) did not affect final test accuracy and train loss values, and did not affect how quickly they reached their elbows (turning points). Note that this may be due to not testing a wide enough range of batch size values. Secondly, mini batch gradient descent performs much better than regular gradient descent. We suspect this may be because using the entire training corpus during each global iteration round causes overfitting, and keeping the parameters more general by using smaller batch sizes allows the model to adjust to other handwriting styles.

The next set of comparisons we make is between different subsampling rates. There is a large difference in the noise of each graph when comparing gradient descent with no subsampling, and those with subsampling. It is also important to note that models that attempted subsampling ended up with a lower final test accuracy, and it seemed random whether they achieved a high or low training loss. The reason we mention the randomness of the loss, is because each subsampling graph (rate of 2 and 3) contains a considerable amount of noise, especially in the loss values. In some interactions they would jump above values of 2, and then down to values of 1.9. Compare this to the model that did not use subsampling and achieved a smooth curve with a final training loss of approximately 0.5. The same argument can be applied to the test accuracy graphs, but it is also important to note that higher subsampling rates saw less noise in the test accuracy graphs specifically. In conclusion, we determined that using no subsampling was more beneficial for the model. It is difficult to find a reason behind this, other than using model parameters from more models which were trained on other sets of data would generally increase the accuracy of a model more, depending on the type of data it was trained on