Comparative Analysis of Deep Learning Training Times: Local Machines vs. Spark Distributed Cluster

Evan Yip and Mark Qiao

Introduction:

In the ever-evolving landscape of distributed computing systems, diverse platforms like Apache Spark, Hadoop MapReduce, Apache Storm, Apache Hive, and Google Cloud Dataflow continually emerge. The proliferation of options, coupled with the escalating system overheads in large computing clusters, has led to skepticism about the indispensability of such systems. Drawing inspiration from Dr. Frank McSherry's critique [1], which demonstrated the potential superiority of local implementations over distributed systems in graph searching algorithms, we embarked on an exploration into the realm of deep learning. Our inquiry was centered around a fundamental question: do distributed databases and computing truly enhance the performance of state-of-the-art neural networks? To answer this, we adapted a neural network implementation from scratch following Omar Aflak's tutorial [2], comparing it against established local Python libraries and distributed machine learning frameworks. The neural network built from scratch allowed us to discern any optimization disparities compared to widely used libraries like Keras. Subsequently, we replicated the neural network structure on Databricks using MLlib, leveraging Spark's Resilient Distributed Datasets (RDDs). Through 10 exhaustive trials on the MNIST digits dataset, we compared the training times and test accuracy of each model. Moreover, we wanted to explore whether the distributed nature of large scale data processing platforms could significantly improve runtimes compared to local implementations. By scaling the cluster resources in terms of worker nodes, we were able to compare the performance of the same neural network on different clusters.

Evaluated systems:

Databricks Spark

The distributed computing platform chosen for this project was databricks, as it provides an easy to work with collaborative interface that works well with Apache Spark. The platform works through a notebook user interface, allowing our team to easily work together on a live databricks notebook.

Apache Spark was chosen as the distributed computing engine to build and test the performance of a neural network classification model on the MNIST dataset. Spark uses dataframes (which are built upon RDDs) which are distributed across the multiple nodes that exist within a cluster. This allows each data partition to be processed separately on each individual node. Several Spark clusters on Databricks were obtained throughout the project, detailed in the table shown below.

Name	**Memory**	**Nodes**	**Cores**
Local	16 GB (M1)	1	1
Community Cluster	15.25 GB	1	1
1 Worker Node (Paid Cluster)	14 GB	2	8
2 Worker Node (Paid Cluster)	14 GB	3	12
5 Worker Node (Paid Cluster)	14 GB	6	24
7 Worker Node (Paid Cluster)	14 GB	8	32

Table 1: Computing memory, number of nodes, and number of cores used to run neural networks.

Throughout the course of this project, a variety of clusters were created in order to test the performance of neural networks on clusters with more worker nodes and cores. These clusters are shown in **Table 1**, where we increased the number of worker nodes from 1 to 2, 5, and 7 worker nodes.

Problem Statement and Method:

The question we sought out to answer was: do distributed databases and computing truly enhance the performance of state-of-the-art neural networks? To approach this question, we implemented a neural network implementation from scratch locally and compared it to both widely used local and distributed deep learning frameworks.

For this analysis we utilized the MNIST dataset which can be accessed from Kaggle here. The files are stored in a binary format developed by Yann LeCun. They are in an idx3-ubyte format. We will be able to read these files using sample code provided on Kaggle. Additionally, the Keras library provides functionality for the MNIST dataset to be loaded in directly, and thus we will be using this for the databricks implementation.

Description of neural network architecture

The architecture for our scratch neural network was relatively simple. The raw data was first flattened from a (28, 28) format to a one dimensional array of length 784. This was then passed through a series of fully connected layers of size 100, 50 and finally 10 for classification. This architecture was consistent across all implementations that we tested. In terms of activation functions, the first two layers utilized Rectified Linear unit functions (ReLu) and the final layer had a hyperbolic tangent function (Tanh). ReLu functions are standard across neural network architectures and Tanh activations functions are typically utilized for the final layer to map outputs between 0 and 1.

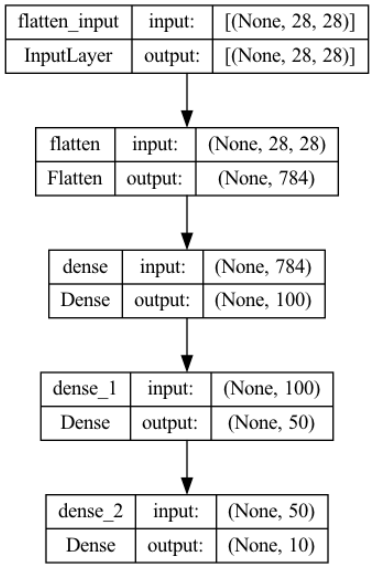

Figure 1: The fully-connected neural network architecture utilized across all implementations for classification.

Results:

Our analysis commenced with a comparison of training times for various neural network implementations, as illustrated in **Figure 1** in the appendix. The implementations included a local scratch implementation, a SparkML MLP classifier with the default optimizer, a SparkML MLP classifier with gradient descent, and a local Keras neural network. The local scratch neural network performed similarly to the default MLP model, with a runtime of around 100 seconds. However, it is important to note that because it uses a different (L-BFGS) optimizer, a one to one comparison is not possible. Moreover, when the default optimizer was changed to gradient descent to match the scratch implementation, the accuracy on the test set plummeted from near 90% to 10%. The exact reasoning behind this discrepancy is unknown, but likely has to do with network architecture in the MLP model. MLP with gradient descent as its optimizer, however, did record a runtime of around 20 seconds. Finally, a Keras implementation of the neural network recorded a runtime of around 120 seconds.

These initial findings prompted us to try to find a method of controlling confounding variables such as optimizers, so we chose to run a neural network using Keras locally and on Spark with one worker node. The results shown in **Figure 2** display a one to one comparison of the same neural network. Upon visual inspection, it is obvious that Keras neural networks running locally performed significantly better than Keras running on Spark. Running a two sample t-test of means between local Keras and Spark Keras confirms this observation – a p-value of $9.89 * 10^{-16}$ shows that the two means are significantly different.

Having established the disparity between running the same implementation locally versus on Spark, our focus shifted towards understanding the impact of resource scaling in Spark clusters. **Figure 3** in the appendix displays a boxplot comparing the Keras neural network implementation between clusters with 1 worker node, 2 worker nodes, 5 worker nodes, and 7 worker nodes. We observe that Keras on 1 worker node performs at a runtime of 580 seconds, while 2, 5, and 7 worker nodes all perform at a runtime at above 600 seconds. It is hypothesized that the extra overhead produced by adding worker nodes caused these clusters to perform worse. Statistically, running a one way ANOVA to test whether the means of the clusters are equal produced a p-value of 0.0017. Therefore, we reject the null hypothesis that the means are equal. Based solely on visual inspection, it is obvious that Keras on a single worker node performs faster than the other clusters.

Because Keras does not utilize any parallel computing, we wanted to explore just by how much distributed processing can impact the training of neural networks. To do this, we ran the same experiment as Keras, but instead using SparkML MLP classifier to utilize the distributed processing capabilities of the platform. **Figure 5** and **Figure 6** display boxplots for a MLP classifier with its default (L-BFGS) optimizer and gradient descent optimizer respectively. As we can see from the plots, the MLP classifier model with the default optimizer performed significantly better when adding worker nodes. Increasing from 1 to 2 worker nodes decreased runtime by 28.87%, while increasing from 2 to 5 worker nodes decreased runtime by 3.91%. Increasing worker nodes from 5 to 7 decreased runtime by 4.32%. The same trend can be observed with gradient descent as its default optimizer, as the runtime became significantly faster with the addition of new worker nodes. One way ANOVA tests were run for both and resulted in p-values of $1.81 * 10^{-23}$ and $2.06 * 10^{-33}$ respectively, further cementing the fact that the groups are significantly different. While increasing the number of worker nodes did result in faster runtimes, the largest decrease in runtime occurred between 1 and 2 worker nodes, and did not change drastically when going from 2 to 5 worker nodes. Therefore, we speculate that there are diminishing returns when scaling resources in clusters for most tasks.

As a brief aside, we also wanted to experiment with how the size of data impacts training a neural network in a distributed processing environment. To do this, we varied the size of the training data set from 90% to 30% of the original size in increments of 10%. Each neural network was run on a 5 worker node Spark cluster. **Figure 7** displays two line plots that show the runtime and accuracy of a MLP classifier model with the default optimizer on a 5 worker node cluster for different proportions of training data. Dropping the proportion of training data from 90% to 80% lowers the runtime drastically, but does not lower the accuracy. In fact, not until the proportion of training data is lowered to 50% does accuracy begin to suffer. Interestingly, the runtime plots display exponential behavior implying that with a specific worker node cutoff, Spark is able to maintain high performance, but at a certain point, there is a threshold at which performance significantly decreases.

Conclusion:

In this project, we hoped to determine if Frank McSherry's general sentiment was echoed through our results. To determine this, we attempted to compare various implementations of the same neural network architecture between single node and multi-node clusters. We showed that running non-distributed deep learning frameworks such as Keras locally resulted in a significantly faster runtime compared to running

Keras on Spark. We then attempted to implement the same model architecture and training parameters on SparkML however were not able to replicate the same setup due to optimization algorithm differences. Despite this limitation, we showed that using SparkML's default optimization algorithm showed faster runtimes to Keras, but similar runtimes to our own LocalScratch implementation.

The later portions of the project displayed how scaling resources might not always be the best choice, as increasing the number of worker nodes from 2 to 5 yielded much smaller returns compared to 1 to 2 worker nodes. Had we had access to clusters that could run 10, 20, or even 100 worker nodes, we might have even seen an increase in runtime. From a business perspective, users that want to run their machine learning models at-scale on a distributed computing platform are advised to first start small with a single node cluster. After measuring a baseline performance, users can increase the number of cluster nodes until runtime performance no longer outweighs the economic burden.

In addition to varying cluster size, we analyzed the effect of training set size on neural network training time on a 5 worker node Spark cluster. Results showed that this cluster was able to maintain low runtimes until being trained on 0.8 of the training set (~48000 samples). At this point performance significantly decreased. Interestingly we identified that though we saw significant improvements in runtime, test accuracy remained high until the training set size decreased from 0.4 to 0.3 of the original training set. In this case, using less data was associated with shorter training times and competitive accuracy performance. Though these results are dependent on this particular dataset, model architecture, and platform they challenge the popular notion that more data equates to better model performance.

To label distributed computing platforms as the optimal solution to every problem would be an oversimplification – oftentimes situations are unique and depend on key factors in the project. Is the data large enough to warrant usage of a distributed computing platform? Does the package or library used for the project even utilize or provide the opportunity to utilize distributed computing at all? These are all considerations that should be made when determining the platform of choice.

Appendix:

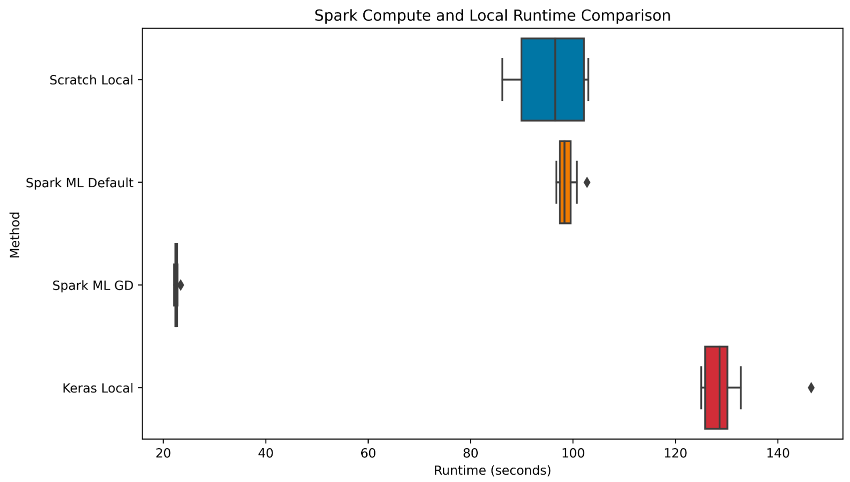

Figure 1: Comparison of training runtime for neural network implementations and platforms. Boxplots represent 10 trials.

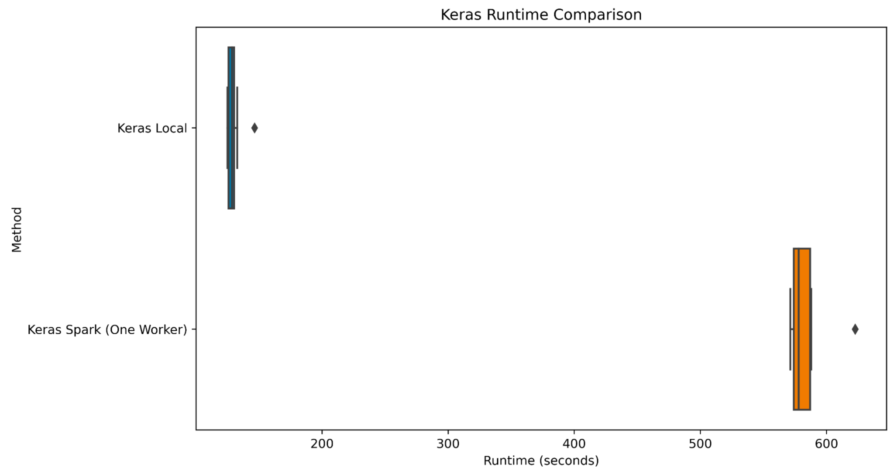

Figure 2: Comparison between running Keras on a local machine vs on a single node cluster on Spark. 10 training trials were run on the MNIST dataset.

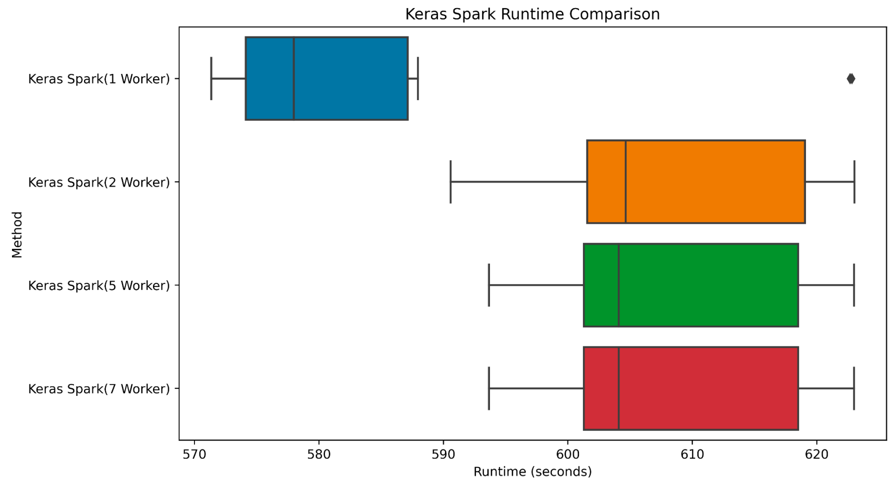

Figure 3: Neural Networks implemented through Keras run on a variety of Spark clusters with 1, 2, 5, and 7 worker nodes. 10 training trials were run on the MNIST dataset.

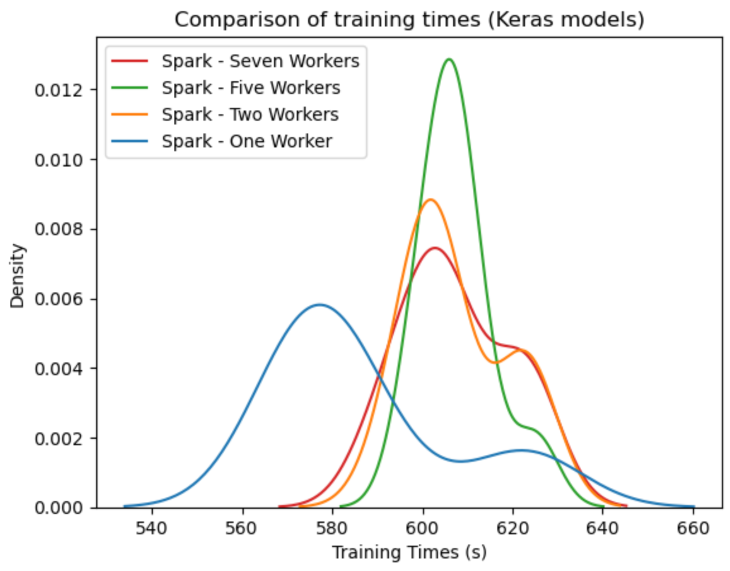

Figure 4: Plot of runtimes of 10 trials for neural networks trained on a variety of Spark clusters.

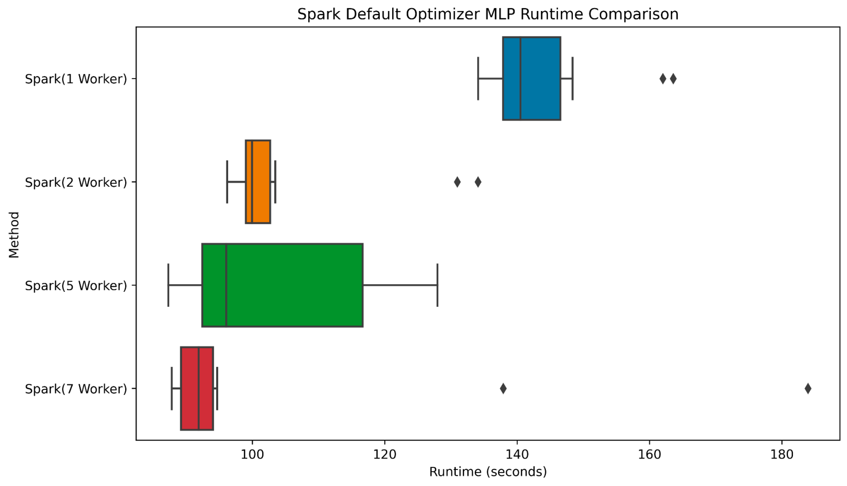

Figure 5: Boxplot of runtimes for 10 trials of Spark Multilayer Perceptron Classifier with default L-BFGS optimizer trained on MNIST dataset for different resource intensive clusters.

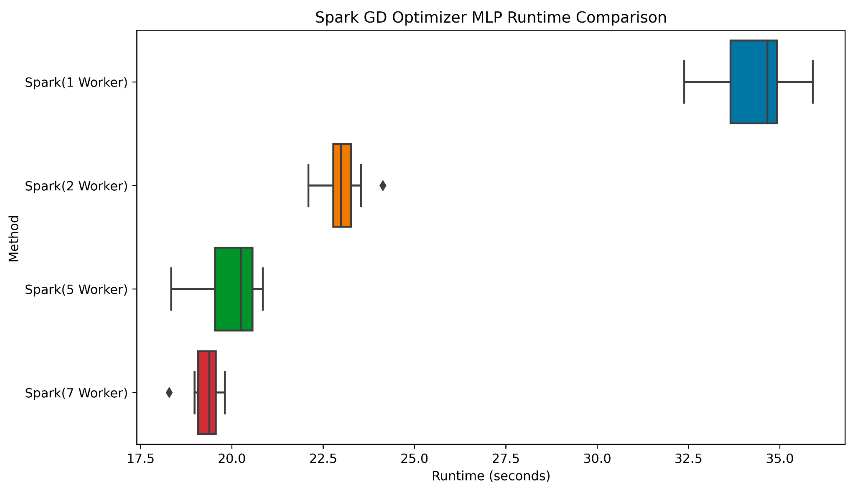

Figure 6: Boxplot of runtimes for 10 trials of Spark Multilayer Perceptron Classifier with gradient descent optimizer trained on MNIST dataset for different resource intensive clusters.

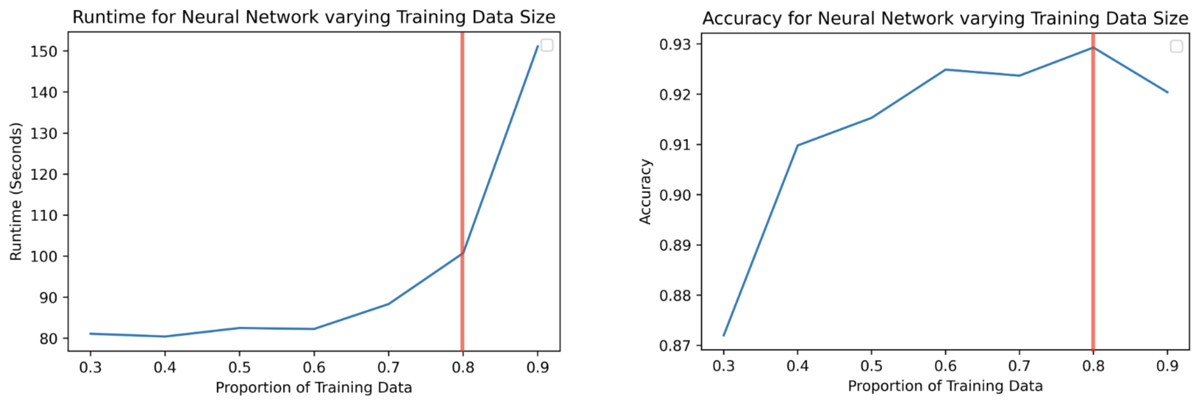

Figure 7: Runtime and accuracy line plots varying the proportion of training data for a MLP classifier trained on MNIST data on a 5 worker node cluster.

References

1. McSherry, Frank, Michael Isard, and Derek G. Murray. "Scalability! but at what {COST}?." 15th Workshop on Hot Topics in Operating Systems (HotOS XV). 2015.
2. Aflak, Omar. “Neural Network from Scratch in Python.” Medium, Towards Data Science, 24 May 2021, towardsdatascience.com/math-neural-network-from-scratch-in-python-d6da9f29ce65. Accessed 04 Dec. 2023.