CSE 599c Spring 2017
Group: Fahad Pervaiz (fahadp) Trevor Perrier (tperrier)
Project Title: Comparing TensorFlow and Spark MLib

Introduction

In our class projected we wanted to compare the machine learning features of TensorFlow and Spark. Our motivation was recent developments from multiple organizations trying to run TensorFlow code on top of a Spark cluster to speed up deep learning workflows. For example, Yahoo recently open sourced a version of TensorFlow that runs on top of Spark and Databricks has a tutorial on running TensorFlow on top of Spark as well. These examples talk mention that Spark is not well optimized for the neural network trainin algorithms that TensorFlow can run. For our class project we wanted to compare vanilla TensorFlow with TensorFlow plus Spark. We identified several data sets that are commonly used for this sort of training and chose to use the hand written images in the MNIST Digits data set since examples on both Spark and Tensorflow existed.

After several difficulties in configuration and running the systems that we needed, we narrowed the scope of our project to simply compare neural network training time on both Tensorflow with Spark MLib. We aimed to study the performance of the two systems on different types of machines with varying CPU and memory sizes. In addition, we wanted to test each system with a larger data set than MNIST Digits and so found EMNIST dataset which is four times as large but contains similar data.

Our results show that Tensorflow is significantly better than Spark's MLib. This was expected since MLib in Spark is not optimized for the same sort of neural network training as Tensoflow. We also tried running the Spark implementation on clusters of 3, 5, and 17 nodes and still Spark could not perform better than Tensorflow. With respect to type of machines, the general purpose clusters perform really well comparing to computing optimized and memory optimized clusters. We did see a trend that the performance gets better with increased number of CPUs but using computing optimized clusters over general purpose ones does not help. It is not surprising that memory does not play a huge role since machine learning algorithm are computation heavy rather than memory heavy.

Systems and Configurations

We configured Amazon AWS clusters to run our systems on multiple types of machines with various number of CPUs and size of memory. After initially attempting - and failing - to set up an EC2 instance from scratch to run Spark we discovered Elastic Mapreduce (EMR) clusters. Which are preconfigured with Hadoop and other big data systems like Spark. EMR clusters can be easily configured for any time of EC2 instance and size of cluster. We ran Spark version 2.1.0. and Tensorflow version 0.10.0 on our clusters. We used these systems out of the box with default parameters. In order to compare Spark and Tensorflow equally we limited our initial

tests to single node clusters. However, after comparing all the different types of EC2 instances we also ran several tests on larger EMR clusters. We ran tests on included the following instance types:

- General purpose machines (m3 and m4): These machines are well rounded with a balance of compute power, memory, network resources. We ran both the slightly older M3 instances as well as the newer m4 instance which have the newer Xeon processors.
- Compute optimized machines (c3): These instances are configured for more CPU intensive operation. Our reasoning to selecting these was that machine learning algorithms are more about computation than memory.
- Memory optimized machines (r3): These machines have larger memory and configured to support large in-memory operations. We used them since Spark is in-memory database so it might help in getting better performance.

Each of these machine types comes in varying sizes from *large* through *16xlarge*. Appendix B lists the instance types that we used. We did perform one experiment where we compared Spark MLib performance with 2, 4 and 16 workers. We ran this on general purpose cluster of type m4.xlarge. It would have been nice to use the latest generation of machines in the Compute Optimized or Memory Optimized instances (c4 & r4) however our Amazon account was configured incorrectly and we needed to have a Virtual Private Cloud and NAT Gateway configured to run any of the 4th generation instances. A classmate helped us by running the m4 instances in their account.

Difficulties

Being relatively unfamiliar with setting up instances on EC2 we expected to have issues in some of our setup. First, since our initial intention was to run TensorFlow on top of Spark we tried to setup up Spark from scratch on a basic EC2 machine. The scripts for doing this all used either the python library Boto or the aws command line interface which were both new to us. We ran into multiple issues and couldn't even get a Spark node running because of permission issues to access the Spark process. In order to get something working we started using the EMR framework that comes with preconfigured Spark, however, this meant that we could no longer run our initial designs since the EMR configuration does not include TensorFlow and we therefore decided to compare Tensorflow and Spark's default MLib.

The second major issue we ran into was starting EMR instances that had EBS only storage. Our account was configured in a way that required us to start those clusters in VPC. Once we managed to do this we could not figure out how to SSH into the VPC and access individual instances. We tried our best to configure a NAT that connects the public traffic to private subnets. Even after allowing incoming SSH traffic on port 22 in all security groups associated with the NAT we were still not able to access even the NAT Gateway. This limited our capabilities to run experiments on the latest version of compute optimized and memory

optimized clusters. In order to run a few tests on the m4 instances we a classmate started instances in their account.

Workload

We used multilayer perceptron classifier on Tensorflow and Spark MLib, which is a feedforward artificial neural network model. It maps sets of input data to appropriate output data using layers of connected graphs where each layer is fully connected to the next one. Each node in this is a neuron and it utilizes supervised techniques to train this network. This is a good neural network algorithm for image based datasets. Even though these neural networks are part of the Spark MLib they are not as optimized on this platform as in TensorFlow where they whole framework is designed to run deep learning neural netoworks. On each platform we measured only the time to train the model and ignored the data ingest and testing times. The datasets we used were much smaller than than the available memory, however, while the actual memory requirments are much larger than the data set size since the algorithm uses lots of memory while calculating the model. This meant we expected to see some effect due to both compute power and memory availability.
For all the single instance trials we copied the both data sets directly to the machine. In the Spark cluster trials we pushed the data to HDFS from the master.

Data

We used an image dataset for training and testing the models. We choose to work with the NIST database of images of handwritten digits with the labels of what is written in the image because there were good examples on how to import this format into both Spark and Tensorflow. Hand drawn digit recognition with NIST is a common neural net example and these tutorials all use the MNIST data set which has 60,000 digits for training and 10,000 for testing. Since MNIST is rather small we also added the larger EMNIST dataset which uses the same data format as MNIST but has 240,000 training examples and 40,000 test set. These two datasets lets us explore how well both system performs when the training data size is increased.

Results

We expected that within each EC2 series (m3,r3,c3 and m4) that the larger instance type would perform better and that the c3 instances would perform better than the general instances (m3 and m4). The model train time for all single node instances are shown in Fig 1 colored by instance size (xlarge, 2xl, and 4xl). This grouping clearly shows that paying for a large instance dramatically improves performance. However, within each size category the m4 series is always the fastest (and from Table 1 we see that the m4 series is actually the cheaper than the other three). We suspect that if we had been able to test the c4 and r4 series that they woudl perform slightly better than the r4 series in the same way that c3 and r3 are marginally faster than m3.

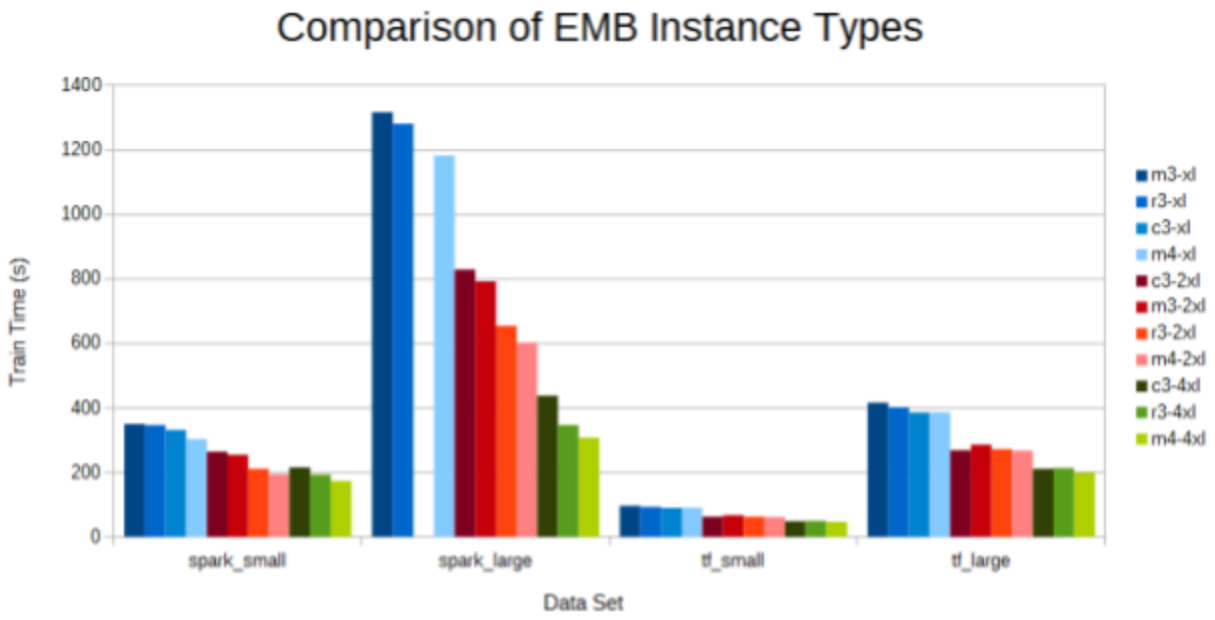

Fig 1: The neural network train time for all 11 types of EC2 instances used for both Spark and TensorFlow on both the large and small data sets. Blue shades are xlarge servers, red shades are 2xlarge servers, and green shades are 4xlarge servers.

From Fig 1 it is also clear that regardless of instance type Tenserflow performs much better than Spark. This is obviously a result of running algorithms that Spark is not optimized for. We also found that Spark ran out of memory on the large dataset for the c3-xl instance (compute optimized) which has just 7.5 GB of RAM. This made us wonder how important memory vs compute power was for our results. Since the relationship between memory and number of CPUs is not fully independent it is hard to determine which has the largest effect.
However, to try to understand this relationship we found the correlation between both number of CPUs and train time (Fig 2) as well as amount of memory and train time (Fig 3) on the Spark trails with the large dataset. We found an R^2 value of 0.853 for CPUs and train time while for memory and train time the R^2 value was 0.431. This would suggest that while it is important to have enough memory extra memory does not necessary improve the performance of either platform for this sort of neural network training.

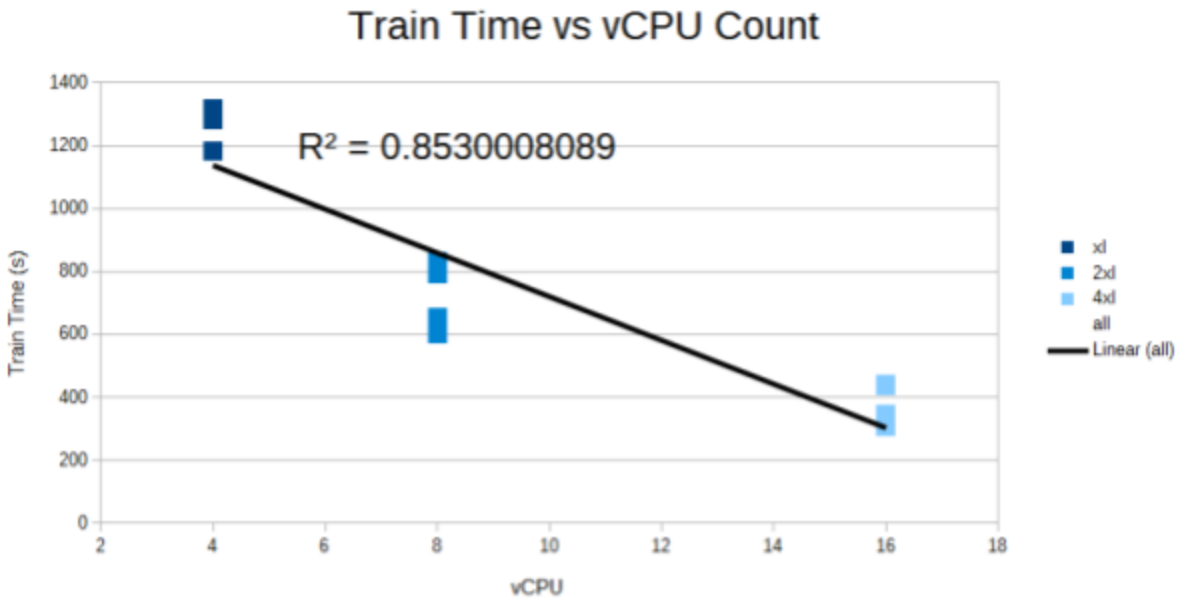

Fig 2: The relationship between number of CPU's each instance type and vs train time.

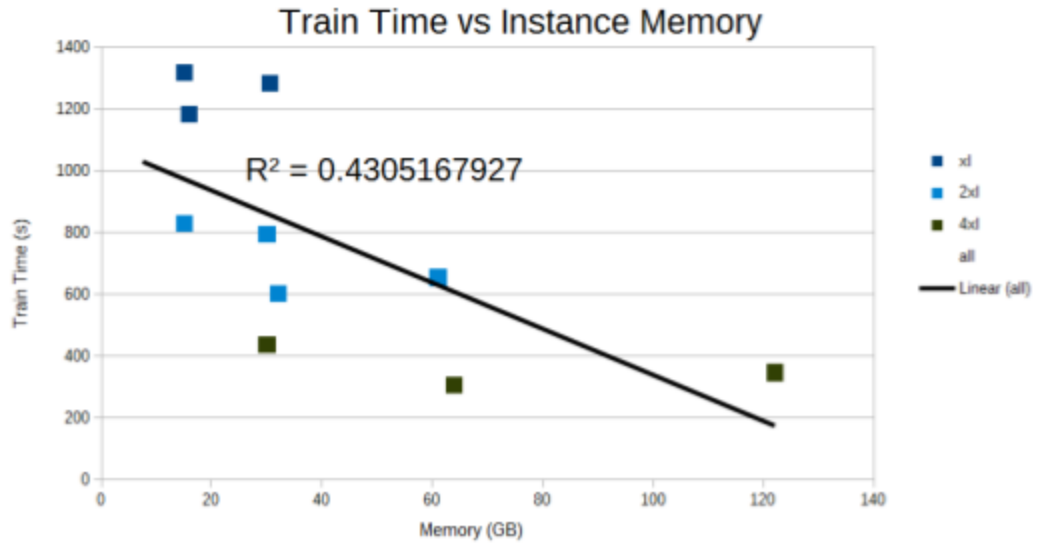

Fig 3: The relationship between memory of each instance type and vs train time.

We also ran tests with Spark on larger cluster sizes. Fig 4 shows the results for both the small and large datasets with Spark cluster sizes of 1, 3, 5, and 17. For all of these test we used m4-xl instances. Interestingly we observe no speedup on the Spark small dataset, however there is a speed up on the large dataset from one to three and from three to five nodes. There is no further speedup by going to 17 nodes. This indicates that the Spark MLib neural network code is not parallelizable. It would be very interesting to see these results with Yahoo's Tenserflow + Spark framework working.

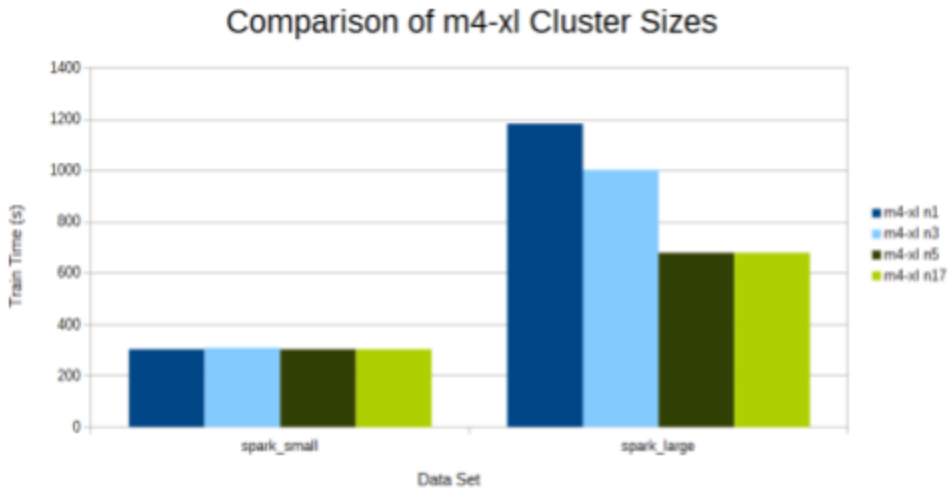

Fig 4: Train times for both the small and large datasets on Spark clusters of size 1, 3, 5, and 17 nodes. For the small dataset we see no speedup while there is a speedup for the larger dataset.

Lessons Learned

The biggest takeaway from this work is that Tensorflow performs better at neural network training than Spark's MLib. This is not surprising since Tensorflow is optimized for exactly this and Spark's API is a general purpose in memory data management system. There are other ML algorithms that are better suited for running on Spark. The second many lesson from our experiments is that unless you have a very specific reason to use a memory optimized or compute optimized EC2 instance the general purpose instances will work just fine. Not surprisingly the newer m4 generation is better than the m3 generation, however, it is surprising that the m4 instances are cheaper than the m3 instances. We also observed that for the same type of instance, adding number of CPUs increases the performance but memory does not have the same effect - at least for the types of neural network training we did. Lastly, we saw that based on the size of the input data there is an optimal number of nodes for the Spark cluster. After this point adding more nodes is superfluous.

There are obviously many limitations to our work. First, much of our time was spent configuring and setting highly nuanced aspects of AWS. While this was a great learning experience and an important skill to learn from this class, the steep learning curve meant that we did not get as far as we wanted on the project. Getting an implementation of TensorFlow + Spark working is the obvious next step. This would allow us to compare the TensorFlow performance on single nodes to TensorFlow on multiple nodes. Also it would be great to try running TensorFlow on a GPU enabled EC2 instance. We found several guides on how to set this up, but did not have time to create a working environment for this project.

Appendix A: Train in seconds for all single node trials

	spark_small	spark_large	tf_small	tf_large
c3-xl	332.476		90.667	385.979
m3-xl	350.705	1317.392	96.262	416.761
m4-xl	304.386	1183.191	91.849	387.05
r3-xl	346.8	1281.98	93.937	402.197
c3-2xl	264.9625	829.947	62.771	268.731
m3-2xl	255.105	793.532	67.17	287.095
m4-2xl	194.78	603.622	62.13	267.938
r3-2xl	211.474	654.85	63.057	272.94
c3-4xl	216.41	438.573	49.169	211.728
m4-4xl	174.162	308.103	46.532	199.665
r3-4xl	193.564	347.021	50.028	213.722

Appendix B: Train time in seconds for m4-xl clusters

	spark_small	spark_large
m4-xl n1	304.386	1183.191
m4-xl n3	309.405	1003.274
m4-xl n5	304.02	679.665
m4-xl n17	304.111	679.941

Appendix C: The memory, cpu count, and hourly cost of all EC2 instances used

Type	Instance	vCPU	Mem (GiB)	Cost ($/hr)
M3/M4 General Purpose	m3.xlarge	4	15	$0.27
	m3.2xlarge	8	30	$0.52
	m4.xlarge	4	16	$0.2
	m4.2xlarge	8	32	$0.4
	m4.4xlarge	16	64	$0.8
C3 Compute Optimized	c3.xlarge	4	7.5	$0.21
	c3.2xlarge	8	15	$0.42
	c3.4xlarge	16	30	$0.84
R3 Memory Optimized	r3.xlarge	4	30.5	$0.33
	r3.2xlarge	8	61	$0.67
	r3.4xlarge	16	122	$1.33