Proceedings of Student-Faculty Research Day, CSIS, Pace University, May 5, 2017

Comparing TensorFlow Deep Learning Performance Using CPUs, GPUs, Local PCs and Cloud

John Lawrence, Jonas Malmsten, Andrey Rybka, Daniel A. Sabol, and Ken Triplin
Seidenberg School of CSIS, Pace University, Pleasantville, New York

Abstract— **Deep learning is a very computational intensive task. Traditionally GPUs have been used to speed-up computations by several orders of magnitude. TensorFlow is a deep learning framework designed to improve performance further by running on multiple nodes in a distributed system. While TensorFlow has only been available for a little over a year, it has quickly become the most popular open source machine learning project on GitHub. The open source version of TensorFlow was originally only capable of running on a single node while Google's proprietary version only was capable of leveraging distributed systems. This has now changed. In this paper, we will compare performance of TensorFlow running on different single and cloud-node configurations. As an example, we will train a convolutional neural network to detect number of cells in early mouse embryos. From this research, we have found that using a local node with a local high performance GPU is still the best option for most people who do not have the resources to design bigger system implementations.**

Index Terms— Artificial Intelligence, Convolutional Neural Network, Deep Learning, Machine Learning, Neural Networks, TensorFlow, Perceptrons

I. Introduction

First, we will give a brief introduction to Machine Learning and Deep Learning, which is a type of machine learning. Following that, we will provide an overview of TensorFlow, a deep learning framework, and the motivation behind it. Last, we will describe our test setup and results. The setup will include an example using deep learning to detect number of cells in mouse embryos.

A. *Machine Learning*

In 1959, Arthur Samuel coined the term "machine learning" which can be defined as subfield of computer science that imbues computers with the ability to learn new code and instructions without being explicitly programmed to do so. Machine learning is also closely related to the field of computational statistics (and often overlaps with it); computational statistics also focuses in prediction-making through the use of computers [1].

Machine Learning enables computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction. These methods have dramatically improved the state-of-the-art in speech recognition, visual object recognition, object detection and many other domains such as drug discovery and genomics.

B. *Deep Learning*

Deep learning discovers intricate structures in large data sets by using the backpropagation algorithm to indicate how a machine should change its internal parameters that are used to compute the representation in each layer from the representation in the previous layer. Deep convolutional nets have brought about breakthroughs in processing images, video, speech and other audio, whereas recurrent nets have shone light on sequential data such as text and speech.

Applications where Deep Learning is successful:

- State of-the-art in speech recognition
- Visual object recognition
- Object detection
- Many other domains such as drug discovery and genomics

The origin of deep learning can be traced back to Rosenblatt's perceptron, first published in 1957 [10]. His perceptron is a probabilistic model to simulate the learning mechanisms of the brain, also called a "brain model." The key idea is for a system to be able to recognize, or perceive, a large number of objects without storing explicit information about these objects. The first version of the perceptron was not "deep," it was a learning system based on 3 layers: sensory, association, and response layer. The association layer transcribes input sensory data to produce some sensible response. Another key property of the association layer is that it can be trained to recognize new objects, further described in next section. The idea of "deep" learning is to have multiple association layers to describe more complex objects. This idea was proposed back then, too, but was deemed too complex and computationally expensive to realize.

In recent years, with access to much faster computers and more training data, the ideas of deep learning have been brought back with great success. Perhaps the most well-known example is LeNet 5 by LeCun et al. [8] in the late 90's, often referred to as MNIST because of the dataset they were using. LeNet 5 is a deep convolutional neural network (CNN) with 7 layers (2 convolutional, 2 pooling, and 3 fully connected) and 60,000 trainable parameters. It resulted in a commercial implementation for reading handwritten bank checks.

The next big advancement in deep CNNs came in 2012 with

AlexNet by Krizhevsky et al. [6]. AlexNet consists of 11 layers (5 convolutional, 3 pooling, and 3 fully connected) and 60 million trainable parameters. In order to train that many parameters, two GPUs were used for 5-6 days. The network was trained on ImageNet [4] and won the yearly ILSVRC (Imagenet Large Scale Visual Recognition Challenge) in 2012. This resulted in a shift from traditional image classification methods to various deep CNN architectures, which have won this competition every year since.

Most modern CNNs use variations of the same 3 types of layers. A convolutional layer takes an image and performs several spatial transforms (convolutions), resulting in several transformed images of same size as the original, or slightly smaller depending on which convolution method is being used. A pooling layer reduces the image size, resulting in fewer computations in subsequent layers and introduces some location invariance. The fully connected layers are traditional neural network layers where all nodes are connected with weights to all nodes in the next layer. It is common to alternate convolutional and pooling layers, and have 2-3 fully connected layers at the end of the chain. In addition to these 3 layers, many competitors in ILSVRC add new types of layers to achieve various advantages. For example, AlexNet implements a dropout layer which randomly excludes parameters from being used during training, making the network more robust to overfitting.

C. Training the association layer

The association layer, from here-on referred to as a neural network, can be described as a function of many variables f(X, W), where $X=x_1..x_n$ is the input from the perception layer and $W=w_1..w_m$ is a set of fixed parameters, or weights. The result from this function is a label Y describing X. When used to recognize objects, the set of weights W is constant. W is only updated when training the model to recognize new objects. When training, the function is applied to k examples of X with known labels Y. W is then adjusted to make the function match as many labels as possible. As W can have many dimensions (m can be in the millions), this can be a very hard problem to solve. The most common strategy is to use the gradient descent algorithm to iteratively find a better W by minimizing a loss function L(W) = $\sum_{i=1}^{k} \sqrt{(f(X_i, W) - Y_i)^2}$, describing the difference (RMSD) between f(X_i, W) and the known labels Y_i. This is done by calculating all partial derivatives of L with respect to each w_i, and gradually update W using small steps in the direction of the gradient. It is not guaranteed to find the global minimum of L, but usually a local minima will suffice. Calculating all partial derivatives is a very computationally expensive operation. The backpropagation algorithm provides a way to make this a lot less expensive.

D. Tensors

Formally, tensors are multilinear maps from vector spaces to the real numbers (V vector space, and V* dual space). A scalar is a physical quantity that can be completely described by a real number. The expression of its component is independent of the choice of the coordinate system (i.e. Temperature; Mass; Density; Potential). A vector is a physical quantity that has both direction and length. The expression of its components is dependent of the choice of the coordinate system. The shape of a tensor is described by rank and dimension. For example, a vector describing a point in $\mathbf{R}^3$ can be represented by a rank 1 tensor with shape [3], or 5 images with 64x64 pixels can be represented as a rank 3 tensor with shape [5, 64, 64].

E. TensorFlow

TensorFlow is a powerful library for doing large-scale numerical computation. One of the tasks at which it excels is implementing and training deep neural networks. TensorFlow provides primitives for defining functions on tensors and automatically computing their derivatives.

Computations are not executed immediately in TensorFlow, instead one creates graphs describing computations. Nodes in a graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. One can think of TensorFlow as a flexible data flow-based programming model for machine learning. Graphs can then be distributed for execution on multiple devices.

TensorFlow was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well.

While TensorFlow has only been available for a little over a year, it has quickly become the most popular open source machine learning project on GitHub [7].

F. Why TensorFlow?

- Any individual, company, or organization could build their own Artificial Intelligence (AI) applications using the same software that Google does to fuel everything from photo recognition to automated email replies
- However, while Google stretches its platform across thousands of computer servers, the version it released to the public could run only on a single machine thus making TensorFlow considerably less capable for others at one point
- At this time, the most notable new feature is the ability for TensorFlow to run it on multiple machines at the same time. Many researchers and startups could benefit from being able to run TensorFlow on multiple machines
- TensorFlow is based on a branch of AI called “Deep Learning”, which draws inspiration from the way that human brain cells communicate with each other
- “Deep Learning” has become central to the machine learning efforts of other tech giants such as Facebook, Microsoft, and Yahoo, all of which are busy releasing their own AI projects into the wild

G. Top 5 Advantages of TensorFlow

1. Flexibility: You need to express your computation as a data flow graph to use TensorFlow. It is a highly flexible system which provides multiple models or multiple versions of the same model, and it can be served simultaneously. The

architecture of TensorFlow is highly modular, which means you can use some parts individually or can use all the parts together. Such flexibility facilitates non-automatic migration to new models/versions, and A/B types of testing with experimental models. Lastly, allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.

2. Portability: TensorFlow has made it possible to play around an idea on your laptop without having any other hardware support. It runs on GPUs, CPUs, desktops, servers, and mobile computing platforms. You can also deploy a trained model on your mobile, as a part of your product, and that's how it serves as a true portability feature.

3. Research and Production: It can be used to train and serve models in live mode to real customers. Stated plainly, rewriting codes is not required and the industrial researchers can apply their ideas to products a lot faster. Also, academic researchers can share codes directly with greater reproducibility. In this manner, it helps to carry out research and production processes faster.

4. Auto Differentiation: It has automatic differentiation capabilities which benefits gradient based machine learning algorithms. You can define the computational architecture of your predictive model, combine it with your objective function and add data to it, and TensorFlow manages derivatives computing processes automatically. You can compute the derivatives of some values with respect to some other values results in graph extension and you can see exactly what's happening.

5. Performance: TensorFlow allows you to make the most of your available hardware with its advanced support for threads, asynchronous computation, and queues.

H. Other Frameworks

Other frameworks that compete, or are compatible to TensorFlow are 'some' of the following::

- fchollet/keras:
- Kerras
- Lasagne
- Blocks
- Pylearn2
- Theano

II. PROBLEM

There are many deep learning frameworks that work well on a single node (machine); however, some tools like TensorFlow excel by scaling out distributing computational jobs on many nodes. Originally, the open source version of TensorFlow was only available on single node machines while Google kept the distributed system version proprietary. This limitation caused some to question the performance and scalability of TensorFlow [9], but it has changed since the multi-GPU version is now available to the public. In this research project we hypothesized that running TensorFlow on such a setup would give us the best outcome for complex problems. We have tested using CPU and GPU on local machines as a baseline to compare with GPU and multi-GPU configurations in the cloud, to see what works best on a medium sized problem.

The model we used to test performance is a modified version of Google's tutorial "Deep MNIST for Experts" [11]. We modified it to train on a new mouse embryo dataset, and to output some additional performance metrics. The mouse embryo dataset consists of larger, more complex, images than the MNIST dataset. We will describe the dataset next, followed by a more detailed description of the network architecture.

A. Dataset

We used the dataset published by Cicconet et al. [3], which is Time Lapse Microscopy (TLM) images of 100 mouse embryos, numbered 0-99. Each embryo was photographed every 7 minutes between 1-cell and 4-cell stage resulting in a total of 34,133 labeled images. The learning objective of the CNN is to predict the number of cells (1, 2, 3, or 4) in each image, we set it up as a classification problem. Images were stored using loss-less PNG format with 8-bit gray scale and a resolution of 480x480 pixels.

3-cell stage is significantly shorter duration than other cell stages, resulting in only 766 images. When training a neural network for classification it is desirable to have roughly the same number of training examples for each category. We selected all 3-cell stage images, and a random subset of 766 images from each of the other cell stages, to get a total of 3064 images. 20% of these images were set aside for testing, and 80% kept for training. Embryos selected for testing were those where embryo number modulo 5 is zero. We separated test data based on embryo number rather than individual images because subsequent frames from same embryo are very similar. If test data is too similar to training data we will not be able to see if the network becomes overfitted.

B. Image Preprocessing

The embryos are always centered, and parts of neighboring embryos are visible along edges. The first transform we did was to clip images to 320x320 pixels to reduce the amount redundant information (fig 1a).

3064 images are a very small number to be training a convolutional neural network. A common strategy to increase training data is to duplicate and transform images while keeping same labels. Given that we are dealing with circular shapes, rotating images seemed like the natural transform. Unfortunately, images are taken with a light shining from the top, creating shadows that would not fall in the same direction if rotated. Before rotating we needed to eliminate these shadows. While being aware that this would reduce information, for example Giusti et al. [5] made use of directed shadows while analyzing images, some testing was done and we concluded that we needed more training data to avoid over fitting. To get rotational invariant versions of the images, we inverted all shadows by performing a histogram equalization, subtracted each pixel by 128, and used the absolute values (Fig 1). This works because majority of pixels, peak in histogram, is represented by background, which is neutral between bright and shaded edges. By rotating each image in steps of 18 degrees we increased our training data by 20 times, to a total of 61,280 image. To reduce memory requirement for training each image was downsized to 64x64 pixels.

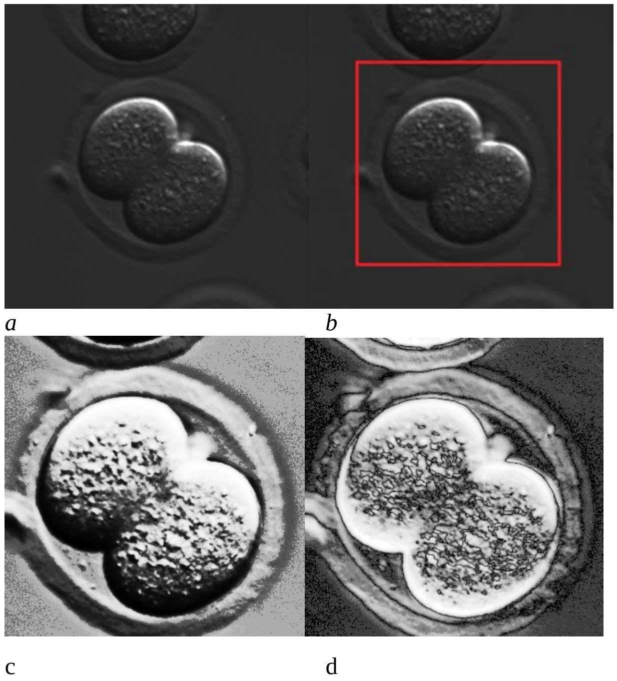


Fig 1. a) E00Frame52 original 480x480 image. b) 320x320 center part of image. c) Histogram equalized. d) Shadows inverted.

C. CNN Architecture

As mentioned in the introduction, most CNNs are constructed using at least 3 types of layers: Convolution, pooling, and fully connected. Our network consists of 6 layers, as illustrated in figure 2; 2 convolution, 2 pooling, and 2 fully connected layers. The first convolution layer takes one 64x64 pixel image as input, and outputs 32 64x64 pixel images. The first pooling layer reduces the output to 32 32x32 pixel images. The second convolution outputs 1024 32x32 pixel images (32 images from each of the 32 input images). The second pooling layers reduces the output to 1024 16x16 pixel images. The first fully connected layer further reduces the output to 1024 fixed nodes, and the second fully connected layer reduces the output to desired number of categories, 4.

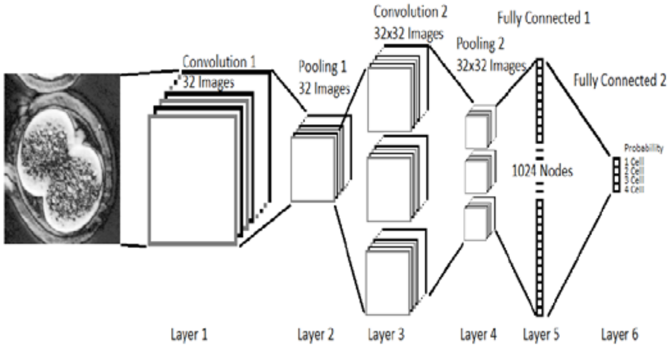


Figure 2. Convolutional Neural Network Architecture.

D. Trainable parameters

First convolution layer has 32 5x5 trainable convolution kernels. Each of the 32 outputs also has a trainable bias, resulting in a total of 832 (25x32 + 32) trainable parameters. Pooling payers have no trainable parameters. The second convolution layer has 64 5x5 trainable convolution kernels for each of the 32 outputs from the first layer, and a shared bias for each output channel, resulting in a total of 51,264 (25x64x32 + 64) trainable parameters. The first fully connected layer connects each pixel from the output images using shared weights to 1024 fixed nodes, each with a trainable bias, resulting in 16,778,240 (16x16x64x1024 + 1024) trainable parameters. The second fully connected reduces the output to number of categories (4), so only 4096 (1024x4) trainable parameters. Altogether we have 16,832,432 trainable parameters, of which most comes from the first fully connected layer.

E. Memory Usage

GPU memory is a limiting factor in most systems. One input image is described by 4096 (64x64) numbers. While these are typically one byte each, intensities ranging between 0 and 255, when used in computations numbers are usually represented as single floating points; 4 bytes each. After the first convolution we have 32 such images in memory, resulting in 131,072 (64x64x32) numbers. First pooling layer reduces this to 32,768 (32x32x32) numbers. Second convolution layer produces 64 images from each of the 32 input images, that is 2,097,152 (32x32x32x64) numbers. Second pooling layer reduces this to 524,288 (16x16x32x64) numbers. The first fully connected layer only adds 1024 numbers, and the second, which is the output, only adds 4. Adding in the 16.8 million trainable parameters, data structures for categorizing one image will require 19,622,836 (4,096 + 131,072 + 32,768 + 2,097,152 + 524,288 + 1,024 + 4 + 16,832,432) numbers, or approximately 83 MB.

F. Training

While not all numbers described in previous section need to reside in memory for every image, training on all images at once would not be reasonable. Training is done in smaller batches of images. By experimentation we found that the 980 ti GPU with 6GB of memory run out of memory when training batches of around 700 images. While TensorFlow was able to manage this at the cost of performance by swapping memory in and out, we settled for a batch size of 480, which is approximately 1% of all training images. After 100 training steps, or 1 epoch, when all training images had been used once, the order of images were randomized and the process started over.

RESEARCH APPROACH

G. Local On-premises Setup

TensorFlow was set up on the local machines of all the authors.

This group had to set up TensorFlow on numerous machines to determine and obtain the best possible training outcomes of our Mouse Embryos. The process has become quite easy as TensorFlow is now available natively for Windows. All you have to do is download python and install TensorFlow using PIP. Previous versions would need Linux or Docker to run

TensorFlow in a container. This process was very difficult to task as you had to run all things together and work in Docker with your code. Another nice feature of TensorFlow is the TensorBoard which shows the outcomes and data sets which you had run through the system.

Run python file with the dataset above on local machine with CPUs and Separately with GPUs

H. Cloud Setup - Google Cloud (GCP)

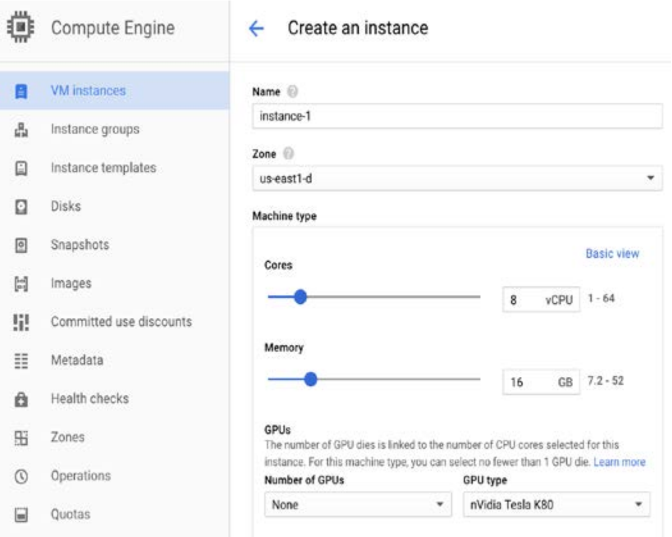
For this particular study, we will be using recently available NVIDIA® [2] Tesla® K80 on GCP with following specifications:

- 4992 NVIDIA CUDA cores with a dual-GPU design
- Up to 2.91 Teraflops double-precision performance with NVIDIA GPU Boost
- Up to 8.73 Teraflops single-precision performance with NVIDIA GPU Boost
- 24 GB of GDDR5 memory
- 480 GB/s aggregate memory bandwidth
- ECC protection for increased reliability

This particular setup is currently in Beta and only available in certain availability zones: https://cloud.google.com/compute/docs/gpus/

We will be using Virtual Machines (Figure 3) with the following parameters to closely match non-cloud configurations:

- GPU NVIDIA® Tesla® K80
- 8 VCPU
- 16 Gb of RAM



Name: instance-1

Zone: us-east1-d

Machine type

Cores: 8 vCPU 1 - 64

Memory: 16 GB 7.2 - 52

GPUs

The number of GPU dies is linked to the number of CPU cores selected for this instance. For this machine type, you can select no fewer than 1 GPU die.

Number of GPUs: None

GPU type: nVidia Tesla K80

Figure 3. Cloud based screenshot

Source code for the project can be found here: https://github.com/compscied/tensorflow

I. Testing Systems

The files which we ran were consisted of 61,280 images, of which 49,024 were used for training of the system and 12,256 were used to test results.

We tested to train our model on 6 different systems:

1) HP Envy Laptop, Dual Core CPU Intel i7 @ 2.5 GHz.
2) HP Envy Desktop, Quad Core CPU Intel i7 @ 3.4 GHz.
3) Asus NVidia 980ti STRIX GPU running on custom built system.
4) Google Cloud 1vCPU with NVidia K80 GPU.
5) Google Cloud 8vCPU with NVidia K80 GPU
6) Google Cloud 8vCPU with 2xNVidia K80 GPU

Learning rate per epoch was similar between all systems, so we will only present one as an example (Figure 4). After around 20 epochs, the learning rate was saturated at around 85% accuracy on test dataset.

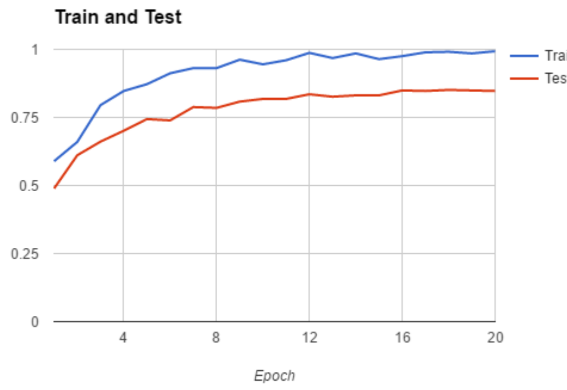


Figure 4. Learning rate for 20 epochs.

As expected, there was a big difference between CPU and GPU systems. Due to this, we separated their charts. We would have guessed time to train on a CPU system would be somewhat linearly proportional to number of cores and clock frequency of the CPU. This seems to be somewhat true, but the relation is not quite linear, figure 4. If it was linear, and the dual core@2.5GHz took 309 minutes, then quad core@3.4GHz should have taken 114 minutes, but it took 134 minutes.

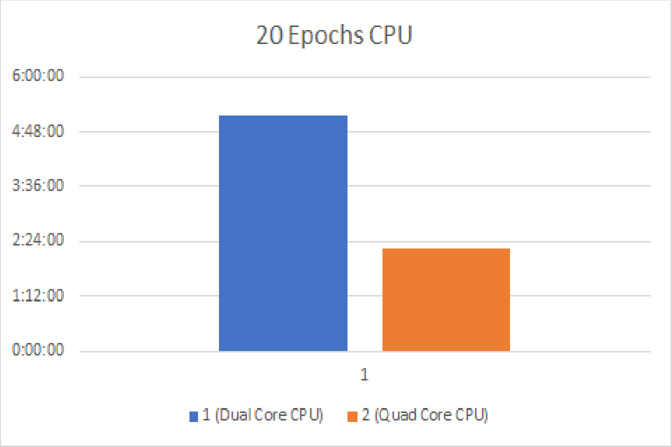


Figure 5. Number of hours to train our model 20 epochs.

Time taken for GPU systems to train our model 20 epochs (Figure 5) seems independent on what CPU is attached to the system. Cloud with 1 vCPU performed similar to cloud with 8 vCPU. The best performance came from the custom-built computer with a standard NVidia 980ti GPU, despite the 980ti GPU being one generation older than the cloud K80 GPU. The 980ti has more CUDA cores and higher memory bandwidth. We had also expected 2xK80 cloud GPU would outperform 1xK80, but it seems TensorFlow does not automatically distribute computations between multiple GPUs, and therefore their performance is similar.

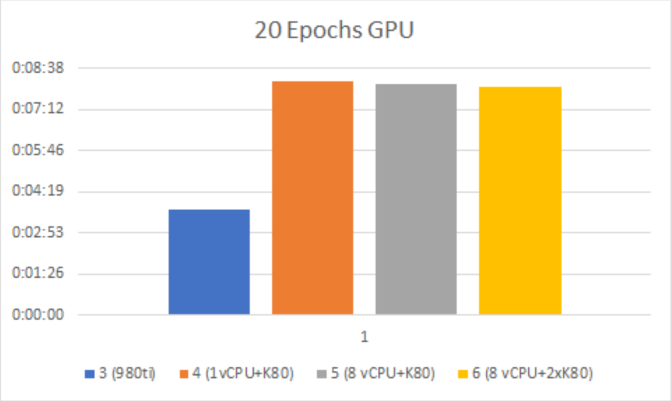


Figure 6. Number of minutes to train our model 20 epochs.

III. Conclusion

We found that unsurprisingly TensorFlow on GPUs significantly outperforms TensorFlow on CPUs. Standard machine RAM was not as important as GPU Memory. In terms of CPU in the cloud 1 vCPU vs. 8 vCPUs did not make much difference. Cloud GPU instances in Google Cloud were not as performant as local PC GPUs even though the GPU in the local PC was older model. Additionally, we found that adding multiple GPUs to the same cloud instance does not make a difference unless we specifically program for each new additional GPU.

IV. Future work

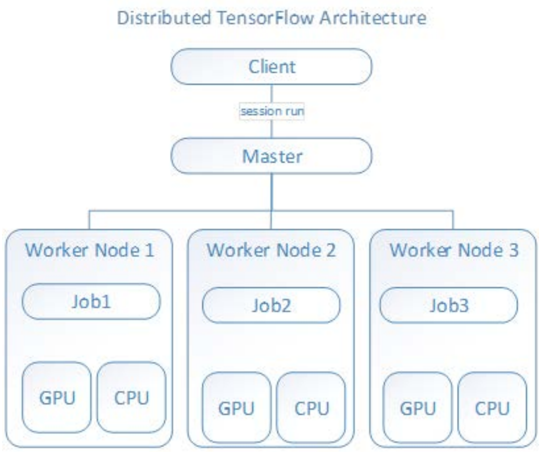


Figure 7. Distributed TensorFlow Architecture

Future work could include comparison of other GPU and CPU configurations as well as RAM (random access memory). The code can be modified for a proper comparison of multi-node systems and multi-GPU systems in the cloud. It could also be interesting to compare TensorFlow performance against other frameworks such as Caffe, Theano, or PyTorch.

References

[1] *Machine learning*. 2017 4/1/2017 [cited 2017 April 1]; Available from: https://en.wikipedia.org/wiki/Machine_learning.

[2] *NVIDIA Tesla K80*. 2017 [cited 2017 3/25/2017]; Available from: http://ww.nvidia.com.

[3] Cicconet, M., et al., *Label free cell-tracking and division detection based on 2D time-lapse images for lineage analysis of early embryo development.* Computers in biology and medicine, 2014. **51**: p. 24-34.

[4] Deng, J., et al. *Imagenet: A large-scale hierarchical image database*. in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. 2009. IEEE.

[5] Giusti, A., et al. *Blastomere segmentation and 3d morphology measurements of early embryos from hoffman modulation contrast image stacks*. in *Biomedical Imaging: From Nano to Macro, 2010 IEEE International Symposium on*. 2010. IEEE.

[6] Krizhevsky, A., I. Sutskever, and G.E. Hinton. *Imagenet classification with deep convolutional neural networks*. in *Advances in neural information processing systems*. 2012.

[7] Lardinois, F. *IBM adds support for Google's Tensorflow to its PowerAI machine learning framework*. 2017 1/26/2017 [cited 2017; Available from: https://techcrunch.com/2017/01/26/ibm-adds-support-for-googles-tensorflow-to-its-powerai-deep-learning-framework/?ncid=rss.

[8] LeCun, Y., et al., *Gradient-based learning applied to document recognition.* Proceedings of the IEEE, 1998. **86**(11): p. 2278-2324.

[9] Mayo, M. *TensorFlow Disappoints – Google Deep Learning falls shallow*. 2016; Available from: http://www.kdnuggets.com/2015/11/google-tensorflow-deep-learning-disappoints.html/2.

[10] Rosenblatt, F., *The perceptron, a perceiving and recognizing automaton Project Para*. 1957: Cornell Aeronautical Laboratory.

[11] Google.com, 'Deep MNIST for Experts', 2017. [online]. Available: https://www.tensorflow.org/versions/r0.11/tutorials/mnist/pros/. [Accessed: 14- Apr- 2017]

[12] Source code for the project can be found here: https://github.com/compscied/tensorflow