# Parallelization of Multi-Layer Perceptron and Long-Short Term Memory on Elgato

*ECE 677 Distributed Computing Course Project Report, 2017.12.04

1st Zhengzhong Liang
*Electrical and Computer Engineering Department*
*The University of Arizona*
Tucson, United States
zhengzhongliang@email.arizona.edu

***Abstract*—Artificial Neural Networks (ANNs) are considered as very powerful computation tools which are able to fit very complicated patterns. However, one major drawback of ANNs is that they usually require a large amount of computation, especially for deep neural networks such as Long-Short Term Memory (LSTM). One solution to speed up the training of ANN is to parallelize the training process. In this article I implemented two types of distributed ANNs in Tensorflow, which are Multi-Layer Perceptron (MLP) and LSTM. I compared the training accuracy, testing accuracy and execution time of our algorithm using one CPU, one GPU and two GPUs. The effect of batch size is also studied in this article.**

***Index Terms*—Multi-Layer Perceptron, Long-Short Term Memory, Distributed Gradient Descent, GPU, Tensorflow, Elgato**

## I. Introduction

ANNs are a type of machine learning algorithm which mimics the structure and information flow of human's brain, meanwhile maintaining a good level of computation efficiency. ANNs consist of "neurons" and "weights". Information is passed through neurons and weights in a particular direction. and the output is given at last. There are two stages in the using of ANNs, training and testing. At training stage, some training samples are fed into network, and the output of neural network is compared with the desired output, which is also known as label. Then depending on the difference between the network output and the desired output, some feedback signal is given, and the weights of network are adjusted accordingly. Especially in an MLP, the adjust of weights is achieved by "back-propagation" (BP). Then at the testing stage, testing samples are fed into the network, and the output is generated. At testing stage, weights are not allow to change. Many tasks can be finished in this way using ANNs, including image classification and text prediction [1] [2].

Many types of ANNs have been proposed since the birth of ANN. MLP is one of the most-classic ANNs [3]. MLP has three layers, input layer, hidden layer and output layer. Data is fed into input layer, then passed to hidden layer through fully-connected weights, and finally passed to output layer through another bunch of fully-connected weights. Training is done by back-propagation. Another type of ANNs, which is very popular recently, is called LSTM [4]. LSTM also has input layer, hidden layer and output layer, but it has more variables than MLP. For example, LSTM can be expanded through time. The state of LSTM network of the previous time slot is stored in a variable called "cell state". And the training of LSTM is done by "Back-Propagation Through Time" (BPTT), which is a variation of classical BP. Compared with MLP, LSTM usually has more trainable weights, thus being able to fit more complex patterns.

However, more trainable parameters usually means longer training time. And this is indeed a major drawback of LSTM. A possible solution to this problem is the parallelization of computation. Fortunately the parallelization of machine learning algorithms are supported from both software perspective and hardware perspective. TensorFlow is an open-source machine learning library developed by Google Inc. [5]. It supports multiple programming language including Java, C++ and Python. Algorithms are mapped to computation graphs in TensorFlow with very concise language, thus defining an algorithm is fairly easy in TensorFlow. Moreover, TensorFlow provides friendly interface of GPU programming. Algorithms can be designed to run on multiple GPUs using those interfaces. As for hardware support, NVIDIA provides multiple types of GPU. Those GPUs can be detected and utilized by Google's TensorFlow through NVIDIA's CUDA and CuDNN library [6]. Combing TensorFlow and NVIDIA's GPU, multiple machine learning algorithms including LSTM and MLP can be implemented in a graceful way.

Elgato is a platform of the University of Arizona's High Performance Computing center [7]. Elagto consists of NIVIDIA K20X GPUs and Intel Xeon Phi 5110p coprocessors. Each node has two NIVIDIA K20X GPUs where shared memory is used. To build an application which requires more than two GPUs, Message Passing Interface (MPI) should be used to communicate among different nodes. Elagto supports to run TensorFlow application on it. User can use the "singularity" image of TensorFlow in the Elgato system, which contains all dependencies needed by TensorFlow.

Thanks for the support from ECE 677 faculties and TAs

In this project I implemented both LSTM and MLP in TensorFlow. Then the models are trained on Elgato's GPU cluster. Models are tested using the MNIST hand-written digit dataset. The performances of models trained by GPUs are compared with those using local CPU. The influence of batch size is also studied. In section II I present the structure of the neural network implemented in this article. In section III the parallelization techniques employed in this paper is shown. Section V shows the experiments and the results. The implementation for multi-GPU LSTM is extremely valuable because seldom people managed to achieve multi-GPU LSTM in TensorFlow (at least no code could be found on Internet). Besides, by experiments I show that LSTM is extremely suitable for multi-GPU parallelization. And finally section VI shows some discussions.

## II. Neural Network Model

Neural networks are very popular models nowadays, known for their superior ability of fitting any given functions. In this article I implemented 2 types of neural networks, which are Multi-Layer Perceptron (MLP) and Long-Short Term Memory (LSTM). MLP is the most classical artificial neural network model. It has an input layer, an output layer and a hidden layer in between. Long-Short Term Memory (LSTM) is a variant a MLP. LSTM usually has more layers than traditional MLP, thus it is usually considered as deep neural network.

### A. *Multi-Layer Perceptron*

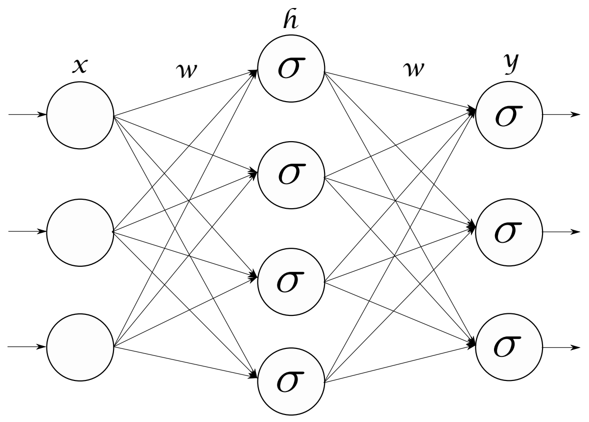

Fig. 1. Structure of MLP

Figure 1 shows the structure of a typical MLP. Layer $x$ is the input layer, $y$ is the output layer, $h$ is the hidden layer and $w$ is the weight. The input layer takes real-valued data as input. Each neuron in the input layer will have one value. Then the value of each neuron is passed to the hidden layer through weights $w$. In the hidden layer, the weighted inputs are summed up and input to an activation function, which is represented by $\sigma$ in the figure. $\sigma$ is usually a non-linear function which maps a real-valued amount to a scale either between zero to one or minus one to one. In my model I use Rectified Linear Unit (ReLU) function as the activation function. The output of activation function is further passed through a second sets of weights to the output layer, where similar computations are conducted again, until outputs are generated.

In training session, the outputs $y$ are compared with some targets $t$. Then the difference between $y$ and $t$ is computed. Depending on this difference, the weights $w$ are adjusted such that next time when the input data is fed into the network, the output $y$ is likely to be closer to $t$. The difference between $y$ and $t$ is called "loss function". In our model I define the loss function as:

$$J(w) = -\frac{1}{K}\sum_{k=1}^{k=K} t_k \ln y_k(x, w),$$

where $w$ is the weight of network, $k$ is the $k^{th}$ neuron in the output layer, $t_k$ is the target (desired output) of the $k^{th}$ output neuron and $y_k(x, w)$ is the actual output of the $k^{th}$ output neuron. Here $y_k(x, w)$ is a function of input $x$ and weights $w$. Our objective is to minimize this loss function with respect to $w$. Thus I utilize stochastic gradient here:

$$w^{(i+1)} = w^{(i)} - \eta \bigtriangledown_w J(w),$$

where $\eta$ is a small positive number, also known as learning rate. The intuition of gradient descent is to change $w$ a bit every time, thus making $w$ to be near optimal value after some iterations.

In the model, I use 784 input neurons in the input layer, with each neuron reading the value of a single pixel of a image in MNIST dataset. The hidden layer has 50 neurons and output layer has 10 neurons. A softmax function of output layer is used in order to get the classification result. Weights between layers are fully-connected.

### B. *Long-Short Term Memory*

Long-Short Term Memory is a new type of neural network in that it does not only have multiple layers, but also that it can be unfolded with respect to time. Figure 2 shows the structure of a typical LSTM network. In the figure, the LSTM network has input layer, output layer and hidden layer, which is the same as the traditional MLP. However, the LSTM has multiple time steps. As illustrated in figure, the LSTM network has 4 time states, and each time states will impose an influence on the network states of the next time step.

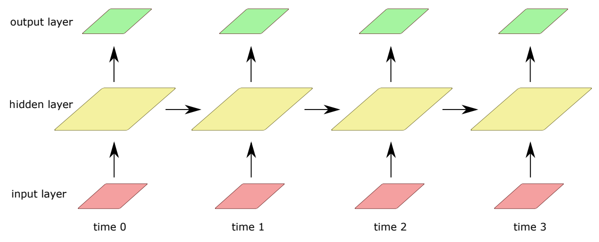

Fig. 2. Structure of LSTM

The training of an LSTM network is similar to the training of MLP. Back-Propagation Through Time (BPTT), which is a variation of traditional Back-Propagation algorithm is used. BPTT can propagates back the errors through the time steps.

Each image in MNIST data set has a dimension of 28×28, so I choose to scan a row at each time step and feed it to our LSTM network. Thus my LSTM network has 28 time steps, with each time step having an input dimension of 28. I use 128 hidden neurons in the hidden layer. The loss function is the same as proposed in section II-A.

## III. Model Parallelization

### A. Data Partitioning

In the project, I tested three training setups: only one CPU, one CPU with one GPU, and one CPU with two GPUs. The data generation schemes are different under different training setups. Figure 3 shows the data partition scheme in the project.

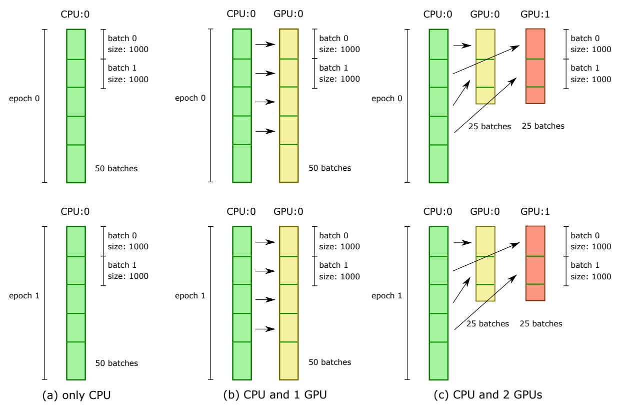

Fig. 3. Data Partition Scheme

Panel (a) of figure 3 shows the data partition scheme using only one CPU. In each epoch, 50 non-overlapping batches of data are generated. These batches are sent to the model sequentially. After all batches in the epoch are used, the model enters a new epoch. Since there are 50,000 training samples in the dataset, the new epoch will have the same data as the old epoch.

Panel (b) of figure 3 shows the data partition scheme using one CPU and one GPU. The training data is initialized on CPU:0. Again in each epoch the training data is split into 50 batches. When training session starts, each batch batch is sent to GPU:0. After the training using batch 0 finishes, the next batch will be copied from CPU:0 to GPU:0.

Panel (c) of figure 3 shows the partition scheme using two GPUs. The data is initialized on CPU and split into 50 batches. Then in the first batch, batch 0 will be sent to GPU:0 and at the same time batch 1 will be sent to GPU:1. Then the tasks on two GPUs will be run simultaneously. After the first batch finishes, the result will be collected by CPU. Then in the next batch, batch 2 will be sent to GPU:0 and batch 3 will be sent to GPU:1. Since the total number of samples in each epoch is fixed (which is 50,000), the number of batches in each epoch under two-GPU setup is only 25 for each GPU. While under one-CPU setup or one-GPU setup, there are 50 batches in each epoch for the GPU or CPU.

### B. Parallel Gradient Descent

Figure 4 shows the parallel gradient descent algorithm implemented in the project when two GPUs are utilized. The neural network with parameter $w$ is initialized on CPU:0. Then on GPU:0 and GPU:1, the loss of network is computed individually. GPU:0 and GPU:1 share the value of $w$, however the data fed into GPU:0 and GPU:1 and is different. Thus the loss and gradient obtained from each GPU is different. If only one GPU or one CPU is used, the gradient calculation and loss calculation in figure 4 will be on a single device, either GPU:0 or CPU:0.

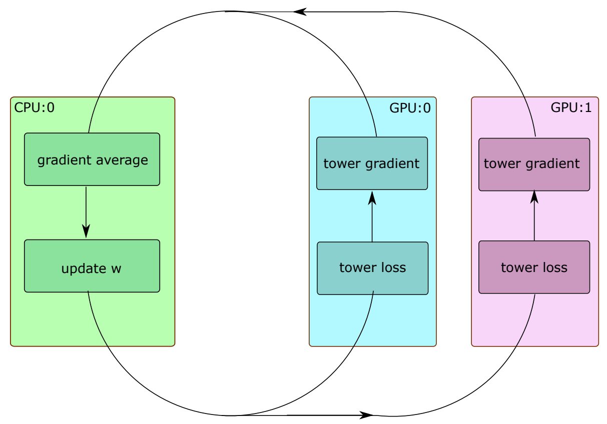

Fig. 4. Parallel Gradient Descent

After each batch, the gradients from 2 GPUs are aggregated together at the CPU. Then parameter $w$ is updated at CPU, which means the neural network model is updated. In the following text I use "tower loss" to denote the loss obtained from a single GPU, and "tower gradient" to refer to the gradient obtained from a single GPU.

## IV. Implementation

I adopt three tools to realize my designed model. Firstly, TensorFlow is used to built the main computation model. Secondly, CUDA is used to build the bridge between my designed model and GPU. Thirdly, a container tool called "singularity" is used in order for me to run TensorFlow application on Elgato. In this section I will introduce several usages in these three tools which are essential to my implementation.

### A. Input Queue

Input queue is a TensorFlow tool which can split input data and send data to the model automatically. The conventional usage of input queue is like the following. Firstly when defining computation graph, a queue object should be defined. This defined queue object serves as an index of a dataset. Then this defined index should be input to the dataset Tensor (multi-dimensional array in TensorFlow), so that different slices of Tensor can be fetched. Thirdly before the computation starts, the queue object should be initialized. Then in the computation, the queue object will automatically generates indices, thus different data slices are generated.

For multi-GPU implementation, an independent input queue should be declared for each individual GPU, because we want the GPUs to train on different samples. In my model the multi-GPU implementation has two GPUs, thus two input queues should be should be declared and fed into the model.

### B. Parameter Sharing

In the model, the weights of neural network $w$ are shared by all GPUs. However, the loss and gradient are computed individually on each GPU. Thus in TensorFlow, some tools can be used to clarify which variables are shared by all devices and which variables are possessed exclusively by the current device. In our program, the weights $w$ are shared by all devices, thus its name is ”layer1/weight1”. However, the gradient name on GPU:0 is ”tower_0/gradients”, which denotes that the gradient variable is possessed by GPU:0 exclusively. Here the prefix ”tower_0” specifies the device to which the variable belongs.

### C. Gradient Collection

The gradients from both of the two GPUs should be collected in order to complete parallel gradient descent. Here I built a python list to store the gradients of each GPU. After each batch, this list is sent to CPU, and the average gradient of the two GPUs are computed.

### D. Scalability

Although the main application of this project uses two GPUs, the program is designed to be able to use any numbers of GPUs, from zero to many. This scalability is achieved by two designs. Firstly, the ”allow_soft_placement” flag is set to true in the program. Thus if no GPU is found available to TensorFlow, the program will automatically map all computation tasks to CPU (or any other available devices). Secondly, in the program, each GPU will be given a unique name automatically, and the order goes like: ”tower_0”, ”tower_1”, ... ,”tower_k”. And the dataset is also automatically split according to the number of available GPUs. User just have to change the parameter ”N_GPU” in the beginning of the script.

### E. Device Usage Tracking

To make sure the computation is truly utilizing two GPUs, I print the device mapping for all operations during the code execution. This is done by setting ”log_device_placement” to be true. The output file (.out) contains the whole mapping. In the LSTM model, 1855 operations are finished on CPU:0, 2373 operations are finished on GPU:0 and also on GPU:1.

### F. Singularity Environment

The TensorFlow Multi-GPU Version needs a lot of dependencies, including NVIDIA drive and CuDNN library, in order to take utilization of multi GPUs while running. However, these dependencies are not installed to Elgato. But with the help of a container called ”Singularity” I am able to run multi-GPU version TensorFlow without installing any of those dependencies on Elgato [8]. The rationale of ”Singularity” is to generate a container including all necessary dependencies in it. Then the python script is executed by invoking the generated container instead of the default python interpreter. The encapsulated container is provided by the HPC team of The University of Arizona and it is directly available in Elgato.

### G. Other Implementation Issues

TensorFlow supports to automatically parallelize computation with a GPU. In other words, user could not control how tasks are split within each GPU. The assignment of threads, blocks and grids in each individual GPU is handled by TensorFlow automatically. However, user must manually define how to split computation tasks among multiple GPUs.

## V. Experiment and Result

The original tutorial code for parallelized gradient descent is for Convolutional Neural Network (CNN), and the model is tested on CIFAR10 dataset [9]. In my project I adopted the parallelized gradient descent algorithm from the tutorial, but I used MLP model and LSTM model, which is different from CNN model in the tutorial. Besides, I use MNIST data to train and test my model, instead of CIFAR10 in the tutorial. Four different batch sizes, 100, 500, 1000 and 2000, are tested in the experiment. Finally, three types of training setups are used. The first setup is local CPU (Intel i7-6700), where no GPU is used. The second setup is Elgato’s CPU and GPU, but only one GPU (Tesla K20X) is used. In this setup, data partitioning is done automatically by TensorFlow’s internal function, and no explicit data partitioning operation is done in code. The third setup is Elgato’s CPU and GPU, where two Tesla K20X GPUs are used. In this setup, the data needs need to be explicitly partitioned and assigned to two GPUs.

### A. Result of MLP

Figure 5 shows the training accuracy of an MLP model using different training setups. The four sub-figures shows the accuracy under different batch sizes. When two GPUs are used, the training accuracy is calculated individually on each GPU. In the figure I call the accuracy on the first GPU to be ”tower 0” and the accuracy of the second GPU to be ”tower 1”.

Figure 5 shows that the training accuracy of using one CPU and one GPU is exactly the same, because in all sub-figures of figure 5, their accuracies completely overlap. The accuracies of tower 0 tower 1 are similar as well, while they do not completely overlap. However, in all sub-figures of figure 5, the tower accuracy using two GPUs is always lower than CPU

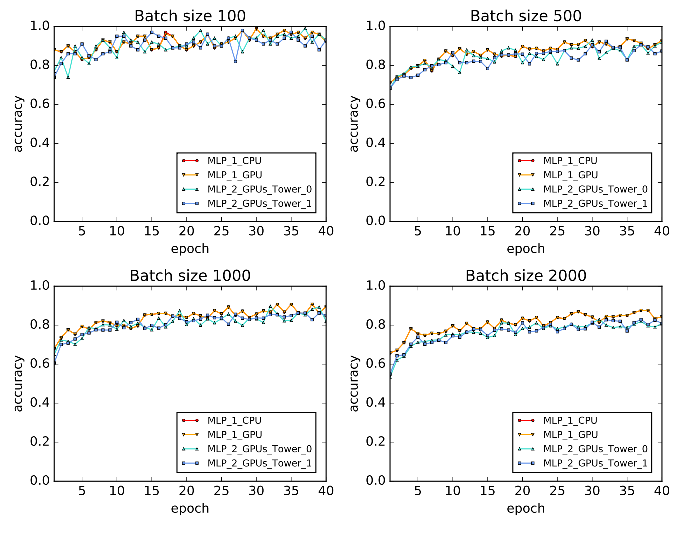

Fig. 5. Training Accuracy of MLP

accuracy and one-GPU accuracy. This trend becomes more obvious when the batch size increases.

Figure 6 shows the testing accuracy and time consumption of MLP model using different setups. Again the accuracy of MLP model using CPU and one GPU completely overlap. The left panel of figure 6 shows that as the batch size increase, the accuracy of MLP model under all setups decrease. However the decrease rates are different. For MLP using CPU or one GPU, the decrease is more steady. There is no sudden drops as the batch size increases. However, this is not the case for MLP using two GPUs. When two GPUs are used, the accuracy seems to drop dramatically the the batch size increases from 500 to 1000. However, when the batch size continues increasing from 1000 to 2000, the accuracy does not drop much.

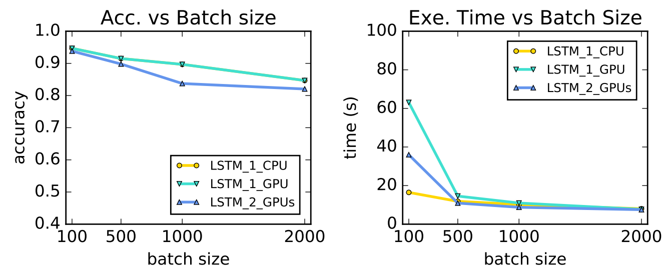

Fig. 6. MLP Test Accuracy and Time Consumption

The right panel of figure 6 shows the total time consumption of MLP model under different setups. Time consumption here is the wall time between the beginning of program and the ending of program. I recorded the time consumption trend as the batch size changes. As the batch size increases, the time consumption for two-GPU setup drops largely, as well for one-GPU setup. However, the time consumption for single CPU setup does not change a lot. One possible reason for this outcome is the communication overhead. However, TensorFlow does not provide an interface which is able to monitor the communication time of devices. To account for the observed outcome, I built a theoretical model. The total consumed time of program can be written as:

$$T = \frac{V_{tr}}{S_b} t_g + (\frac{V_{tr}}{2} + V_{te}) t_s \qquad (1)$$

where $V_{tr}$ is the volume of training set, $V_{te}$ is the volume of testing set, $S_b$ is the batch size, $t_s$ is the time for processing each sample and $t_g$ is the time for communicating each batch between CPU and GPU. Time $T$ in equation 1 mathematically formulates the total execution time when our MLP model is executed using one CPU and two GPUs. In the equation, $V_{tr}$ is 50,000 since we have 50,000 training samples, $V_{te}$ is 10,000 since we have 10,000 testing samples, and $S_b$ can be 100, 500, 1,000 or 2,000. For simplicity, we assume $t_g$ and $t_s$ to be constants. To evaluate the value of $t_g$ and $t_s$, we use the result from figure 6. If we use the result when batch size is 100 and batch size is 2,000, the following equation can be obtained:

$$\begin{cases} 62 &= \frac{50000}{100} t_g + (\frac{50000}{2} + 10000) t_s \\ 8 &= \frac{50000}{2000} t_g + (\frac{50000}{2} + 10000) t_s \end{cases} \qquad (2)$$

Then the value of $t_s$, $t_g$ can be solved:

$$\begin{cases} t_g &= 0.114 \\ t_s &= 0.00015 \end{cases} \qquad (3)$$

Equation 3 shows that the time for communicating each batch is 0.114 s, while the time for processing each image is about 0.00015 s. To testify our proposed model, we can calculate the total execution time for batch size 500 and 1,000, then compare the estimated time with the actual time. After plug in the solved $t_s$ and $t_g$ to equation 1, we get the estimated execution time:

$$\begin{cases} T_{500} &= 16.56 \\ T_{1000} &= 10.86 \end{cases} \qquad (4)$$

In equation 4, $T_{500}$ is the estimated execution time of two-GPU setup when batch size is 500, and $T_{1000}$ is the estimated execution time when batch size is 1,000. These estimated values are close to the result given in figure 6, so we can conclude that our estimation formulation shown in equation 1 is valid in general.

From the result of equation 3 we know that communicating each batch between one CPU and two GPUs takes about 0.115 s. When batch size is 100, the total communication time is about 57.5 s. Recall that the total execution time under this batch size is 62 s. Thus 92.7% of total execution time is because of communication. One solution is to increase the batch size. However, even when the batch size increases to 2,000, the communication time is still as high as 2.85 s, taking up 35.6% of total execution time. If we continue to increase the batch size, the test accuracy drops. So the conclusion is that the MLP model is too simple to use multi-GPU framework on

Elgato, because the improvement of using multiple GPU could not offset the impair caused by communication overhead.

## *B. Result of LSTM*

In the project I tested another neural network, LSTM. Figure 7 shows the training accuracy of LSTM model using different training setups (one CPU, one GPU and two GPUs). The influence of different batch sizes are also recorded. Result shows that under all batch sizes, the epoch accuracy using two GPUs is always lower than one CPU or one GPU setup in the first few epochs. This makes sense in that under two-GPU setup, each the dataset in epoch is split to two parts. For example, assume the data volume in each epoch is 10,000, then if only one GPU or one CPU is used, the model can use all of the 10,000 samples. However, if two GPUs are used, then each GPU can only use 5,000 out of 10,000. Thus in each epoch, each GPU will see less samples. Another trend is that as the batch size increase, the accuracy of two-GPU model drops a lot.

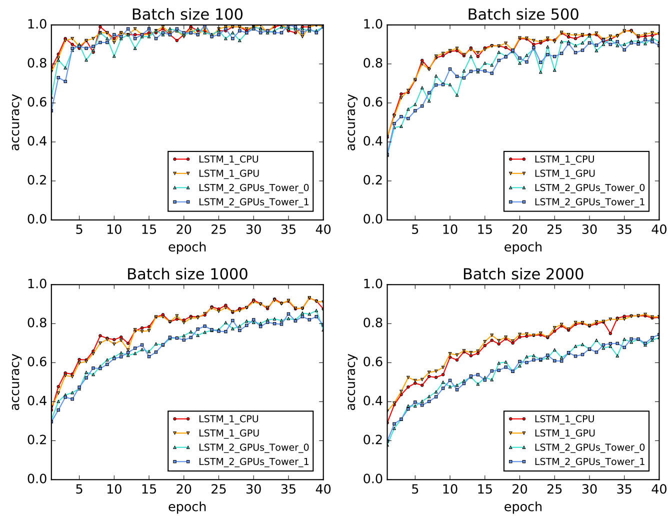

Fig. 7. Training Accuracy of LSTM

Figure 8 shows the test accuracy and execution of LSTM model using different batch sizes. The same as the trend using MLP model, the test accuracy drops as the batch size increases. However for LSTM model, the accuracy decrease using two GPUs is more significant than it is using MLP model. The right panel of figure 8 shows the execution time comparison using different training setups. One interesting finding is that two-GPU setup is now the most time-efficient setup, even when the batch size is as small as 100.

By applying equation 1 on LSTM model we can get the solution for $t_g$ and $t_s$:

$$\begin{cases} t_g & = 0.61 \\ t_s & = 0.0027 \end{cases} \tag{5}$$

Equation 5 shows that the communication time for passing each batch is about 0.61 s. This is about 6 times longer than

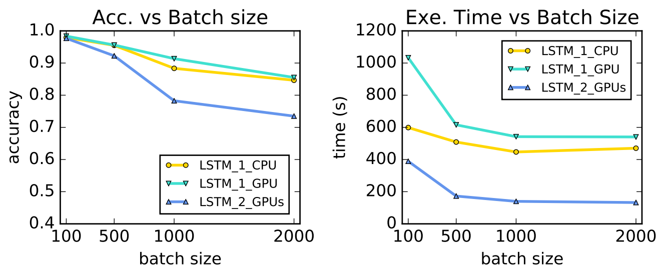

Fig. 8. LSTM Test Accuracy and Time Consumption

in the MLP model. Moreover, the time needed for processing each image using LSTM is about 0.0027 s. This is about 10 times longer than the time using MLP model. When the batch size for LSTM model is 2000, the communication time is about 15.25 s, while the total execution time is 110 s. The ratio of communication time becomes 13.9%. In comparison, the ratio for MLP model is 35.6%. So the communication overhead is less severe in LSTM model than in MLP model. It is not because the communication time for passing each batch decreases. Instead, it because the time need for processing each image using LSTM increases. However, when the batch size becomes 2,000, the accuracy of LSTM model is too slow. So in this proposed model, the best batch size for 2-GPU LSTM seems to be 500, where considerable speedup is obtained, while an acceptable accuracy is maintained.

## VI. Conclusion and Discussion

In this project I implemented two types of neural networks, MLP and LSTM, and used them to do image classification. The two models are then parallelized using either one GPU or two GPUs on Elgato. Result shows that multi-GPU design for the proposed MLP model does not gain a desirable result. The accuracy is almost the same as the CPU implementation, whereas the communication overhead of multi-GPU implementation is severe, thus making the total execution time of multi-GPU implementation larger than CPU implementation. However, the multi-GPU implementation of LSTM manifests a good property. The time consumption using two GPUs is significantly less than using one GPU or one CPU. This is because that the time consumption for processing each single image is large.

This project is valuable in that seldom people succeed in parallelizing an LSTM network in TensorFlow. And by experiment, we show that LSTM is very suitable for parallelization, because parallelization will save a lot of time, meanwhile being able to maintain an desirable accuracy. One future direction of the project could be to test the proposed network on more datasets. For example, LSTM is known for its superb performance on text processing problems. Thus the proposed LSTM can be tested on text prediction problems.

Secondly, the parameters in the model may also affect the accuracy and execution time, but these factors are not included in this project. In previous experiments, I notice that "dropout probability" may largely affect the execution time of a neural network model. Thus the next step for this project could be to investigate the influence of the parameters on the execution time of program.

Thirdly, in this project I did not use more than two GPUs. This is because there are only two GPUs on each Elgato's node. Thus TensorFlow can only detect two available GPUs at most in the execution. To use more GPUs, the user needs to use MPI to pass message among different nodes. However it is still unknown whether MPI is compatible with TensorFlow. So another potential topic is to study how to use more than two GPUs on Elagto and to study the collaboration of TensorFlow and MPI.

## REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, ImageNet classification with deep convolutional neural networks, Communications of the ACM, vol. 60, no. 6, pp. 8490, 2017.

[2] M. Sundermeyer, R. Schlter, and H. Ney. "LSTM neural networks for language modeling." In Thirteenth Annual Conference of the International Speech Communication Association. 2012.

[3] C. M. Bishop, Pattern recognition and machine learning. springer, 2006.

[4] F. A. Gers, Jrgen Schmidhuber, and Fred Cummins. "Learning to forget: Continual prediction with LSTM." (1999): 850-855.

[5] "API Documentation — TensorFlow", TensorFlow, 2017. [Online]. Available: https://www.tensorflow.org/api_docs/. [Accessed: 08- Dec- 2017].

[6] "TensorFlow Framework & GPU Acceleration from NVIDIA Data Center", NVIDIA, 2017. [Online]. Available: https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/tensorflow/. [Accessed: 08- Dec- 2017].

[7] "El-Gato — El-Gato, Super Computer", Elgato.arizona.edu, 2017. [Online]. Available: http://elgato.arizona.edu/. [Accessed: 08- Dec- 2017].

[8] "Training - UA HPC - UA HPC Documentation", Docs.hpc.arizona.edu, 2017. [Online]. Available: https://docs.hpc.arizona.edu/display/UAHPC/Training. [Accessed: 08- Dec- 2017].

[9] "Convolutional Neural Networks — TensorFlow", TensorFlow, 2017. [Online]. Available: https://www.tensorflow.org/tutorials/deep_cnn. [Accessed: 08- Dec- 2017].