Complexity Estimation of Deep Learning Models

CS572 - Data Science Practicum - Project Report

Submission Date: 08/08/2020

Sponsor: CCC Information Services
Husam Sweidan (Senior Systems Engineer)
Manoochehr Assa (Senior DevOps Engineer)

Illinois Institute of Technology:
Niti Wattanasirichaigoon
A20406934
(nwattanasirichaigoon@hawk.iit.edu)

Hardev Ranglani
A20449870
(hranglani@hawk.iit.edu)

Index

1. Project Background

Sponsor's Line of Business

Founded in 1980, CCC Information Services Inc. (acronym for Certified Collateral Corporation) is a leading provider of innovative cloud, mobile, telematics, hyperscale technologies and apps for the automotive, insurance, and collision repair industries.

CCC Information Services connects the automotive, insurance, and collision repair industries by providing technological solutions through their network and innovative platform. CCC's vast network consists of more than 350 insurance companies, 24,000 repair facilities, OEMs, hundreds of part suppliers, and dozens of third-party data service providers. Their large network allows them to access most of the claims data in the industry, which they use to create innovative solutions for their clients and become one of the biggest firms in the auto-insurance industry.

Description of Sponsor's Problem

CCC provides services that allow their clients to estimate claim amounts from images of damaged cars. To do this, they train many deep learning models based on data of car images. These models often involve the use of Convoluted Neural Networks (CNN) as they are a standard tool for image recognition. The training of deep learning models involves passing batches of data into the neural network multiple times to allow the network to gradually adjust or 'learn' its parameters to best fit the dataset. Because of this, the computational cost of training a model can escalate very high when we have a large training dataset or a model with numerous parameters. As such, the cost benefit analysis becomes important to assess whether the benefits realized from the application of the CNN model can be worth the cost of training the same.

Training of large deep learning models are commonly done using GPUs as they are much faster than a normal CPU at processing matrix computations which are present in neural networks. GPUs, however, come at a cost. CCC might be able to train their deep learning models on cloud platforms such as AWS or MS Azure, but it will cost them a large sum of money, especially for larger models or datasets because they take a longer time to train. An alternative solution is for CCC to train their models using their own local GPUs, but in turn their resources are much more limited. Therefore, CCC would benefit from methods that allow them to accurately estimate the amount of GPU time it takes to train a particular deep learning model as they will be able to schedule and prioritize their resources more efficiently.

This particular business problem forms the basis of our practicum project, that given a particular Convolutional Neural Network model with a given model architecture, dataset size and the specifications of the available GPU hardware, we have to predict with reasonable accuracy the GPU time required to train the model for a given number of epochs.

Below is a table listing the run time of standard DNN models and their training time. The overall question then becomes that given a new DNN model with a given model architecture of 'X' layers and 'Y' parameters, how to estimate with a reasonable accuracy the time required to train the model.

Model	# Layers	# Parameters	Training Time
VGG16	16	140 M	3 weeks
AlexNet	8	62 M	6 days
GoogleNet	7	7 M	1 week
MNIST data CNN model	4	24 k	20-30 minutes
A new model	X	Y	??

There are several studies conducted on training time estimation of deep learning models. One approach, called the PALEO method(https://openreview.net/pdf?id=SyVVJ85lg), decomposes the total execution time analytically. It introduces methods of calculating the computation time from factors such as size of the computation inputs within the network architecture, the complexity of the algorithms and operations, and the performance of the hardware used. This method systematically breaks down the problem into mathematical computations which is great for understanding the relationship between performance and architecture. However, the study had only formulated the complexity for popular model architectures such as *VGG16*, (formulation of complexity of a custom model to come in later releases) and it also does not consider the difference in optimizers or activations functions used in the model. Hence, we believe that there are many factors left unconsidered from the PALEO method and decided to not adopt it for this project.

Another research paper, "Predicting the Computational Cost of Deep Learning Models"(https://arxiv.org/pdf/1811.11880.pdf), uses specifically trained deep learning models to predict the run time for each layer of the deep learning model. These predictions can then be summed up to provide a prediction for the overall training time of a model.

The overall approach here is to break up each deep learning network into single components, considering individual layers as the atomic operations that are going to be used for prediction of execution time. A random selection of these atomic operations is constructed and embedded within the simplest 1-layer network. Each of these atomic operations is then executed either using forward and backward passes and the execution times are recorded from multiple executions. The feature set for the atomic operation along with the execution timings are then used to train a fully connected feed forward network. This network can then be used for predicting the execution time for a new operation based just on the feature set. Once a prediction has been made for an individual operation these can be combined across all layers to provide a prediction for the overall

performance of the deep learning network. This approach can thus be used to compare the overall predicted v/s actual training time for a given Deep Learning Model.

We came to the consensus that this method is easier to apply to complex network architectures as it breaks down the problem to estimating the execution time for each layer, and it is also robust to the model features, layer specific features, hardware features, etc. It can also adapt to new hardware easily by retraining the models on data benchmarked on new GPUs. In this project, we will implement this method with *Tensorflow MirroredStrategy*, a module that supports distributed training on multiple GPUs on the same machine.

2. Objectives

2.1 Delivered Objectives

1. A methodology for benchmarking training time of individual neural network layers, compatible with Tensorflow 2.0 and onwards
2. A methodology for building a model to predict the total training time of a given deep learning model with Tensorflow MirroredStrategy(hereafter interchangeably referred to as "MS" in this document) implementation, taking the model structure (each layer and its features), dataset dimensions, hardware specifications, etc. as input
3. A demonstration that such a methodology can provide reasonable accurate predictions for the total training time for a given Convolutional Neural Network Model

2.2 Researched but not delivered Objectives

1. An analysis of the difference in performance of different GPUs (or different GPUs used in sync) of the exact same type in terms of the training time for a given model

3. Deliverables

A list of project deliverables is provided below. Please note that the python codes for data generation will require a GPU environment to run.

No.	Deliverable	Description
1.	Python scripts for generating the benchmarking data for individual layers of a model, for training both on a single GPUs or multiple GPUs(distributed training using MS)	● Creates the simplest models for a 1- layer operation with randomized parameters in a given range and records their average training times ● Compatible with Tensorflow 2.0 ● Can be used with any number of GPUs for training using MS ● The generated data using this script can be used to build the model to predict then run time for an individual layer of a model
2.	Data generated using the benchmarking script on the GPUs we were able to access	● 15,000 records for Conv layers on single GPUs (K80, M40, P100CLIE, P100SMX2) ● 3,000 records for Dense layers on P100CLIE with MS Implementation
3.	Models built using the benchmarking data	● DNN models built from the benchmarking data we generated (also available is the researchers' original model) ● These models can predict the run time for the individual layers both for training on a single GPU or multiple GPUs (distributed training using MS) ● Inputs: layer features (batch size, matrix size, kernel size, channels, optimizer, strides, activation function etc.); Output: Predicted Training time of layer per batch
4.	Python Scripts for getting the predicted time of training the overall model	● Using the models built on the benchmarking data, these python scripts can take input features from a given model architecture, along with hardware specs, etc. and provide the predicted run time for the overall model as the output
5.	Python scripts for recording the actual training times for different models	● These scripts load the standard datasets(the MNIST, CIFAR10, and patch camelyon datasets) available from the Tensorflow

		library and record the model training time (for various batch sizes) for a model suitable to each dataset trained on a single GPU as well as multiple GPUs ● The actual training time recorded can be used to compare the predictions of training time for the overall model
6.	Analysis of predicted v/s actual run times	● Using the predicted and actual run times, the analysis of the accuracy of our predictions of run time ● If possible, list the reasons for low accuracy of predictions wherever applicable

Availability of Deliverables

As some of the final tasks of the project are yet to be discussed and completed, we have not clearly discussed with the clients how the final deliverable will be shared with them. However, we have all the available codes placed in our GitHub repository(https://github.com/profnote/ml-performance-prediction) as of now, and even after the project completes, the entire code and deliverables will be available in this repository, even if we share the deliverables with the clients separately.

4. Project Complications and Constraints

4.1 Complications

As we worked on our project, we faced several challenges we had to overcome. The main complications we encountered are summarized below.

Lack of Access to Client's Environment

In our original timeline, we had planned to work on the client's GPU environments. This would allow us to benchmark individual layer training times directly on the client's machines, yielding us results that would better predict the actual runtimes on the client's system. However, due to CCC's internal legal and IT policies, we are not allowed to access the company's system nor can we receive any data from them. As an alternative, we switched to Chameleon Cloud, a publicly available testbed for computer science experiments, to access the GPUs which can be used to generate the benchmark data. It took us several weeks to fully understand and get accustomed to using the platform.

Tensorflow MirroredStrategy

Our initial project scope did not include any implementation of TF MirroredStrategy(or any other implementation of distributed computing), even the original research paper did not talk about any kind of distributed computing, but since CCC uses distributed training in their model training process, it became our new goal of the project. Both of our team members did not have any experience with distributed systems prior to the project. We spent a great amount of time understanding the mechanism behind TF MirroredStrategy and implementing
it to our benchmark script.

Failure of the original paper's approach while using Tensorflow MirroredStrategy

As our goal shifted to prediction of training time for a DNN model using TensorFlow mirrored strategy, we tried to replicate the same approach as mentioned in the original paper. We generated benchmarking data for the individual layers using the MS and used this data to build a model for predicting the run time of each layer for a model to be trained using MS. While using this model for getting the predictions of training time for a model on mirrored strategy, we saw that the predictions of the training time were in 100's milliseconds per batch, however, the actual training times were <10 milliseconds. We double checked the entire process again and found that both the predicted run times and the actual run times are getting recorded correctly. This meant that the original approach is failing when applied using the TensorFlow MS implementation. Thankfully, after some exploration, we were able to understand why this approach was failing and we were able to come up with a new approach which gave much better predictions of run time for the model.

4.2 Constraints

There are also some limitations of the platforms and resources that we are using for the project.

Original Script in Tensorflow 1.0

The original codes of the researchers were written in Tensorflow 1.0 Although we can automatically upgrade the python scripts from Tensorflow 1.0 to Tensorflow 2.0 by using `tf_upgrade_v2`, most of the commands are still deprecated and may be removed in future versions of Tensorflow. Because of the time constraint, we did not have enough time to go over every line of code to change the TF1.0 code to its TF2.0 counterpart. The mlpredict library that is used for making full model predictions requires us to set up the package manually and also runs only in TF1. Other version constraints include scikit-learn that cannot be used on version 0.23 or higher.

Chameleon Cloud GPUs

Chameleon Cloud is not a commercial cloud platform like AWS, therefore the types of GPUs available, the number of GPUs per machine, and the number of available machines is much more limited. Also, as it is a shared platform, we had to make sure we make a reservation in advance for the specific type of GPU we need to use. Sometimes, the required reservations were not available, which led to delays in running the script and specific tasks taking longer time than expected to complete. The available GPUs on Chameleon Cloud include the Tesla K80, M40, P100CLIE, and the P100 Nvlink (SXM2) with most having only upto 2 GPUs per machine. This environment is far from our ideal as our client uses much stronger GPUs and larger numbers of GPUs per cluster. We focused on conducting our experiments on the P100CLIE GPU (of which we will only refer to as P100 from now on) and implementing MS over 2 GPUs but the methodology is applicable to other GPU types and numbers as well.

5. Project Execution

5.1 Understanding the Methodology of the Original Research Paper

We first examine the methodology described in our main research paper of reference, *Predicting the Computational Cost of Deep Learning Models*. The process can be summarized as follows:

- Generating the benchmarking data
- Preprocessing the data for model building
- Using this data for building an optimal DNN model that can be used to get the predictions for run-time for a given DNN model
- Using the python package "mlpredict" (which utilizes the above model) to get predictions of run time for a given DNN model.

These steps are described in detail as follows:

5.1.1 Generating the benchmarking data

The original paper's implementation methodology and code are available on their GitHub repository(https://github.com/CDECatapult/ml-performance-prediction). This code allows to train a machine learning model that can predict the execution time for commonly used layers within deep neural networks - and, by combining these, for the full network. A python package that utilizes this model for inference can be found at https://github.com/CDECatapult/mlpredict.

The first step is to run the python script `benchmark.py` to benchmark the training time of individual layers on a GPU. The script takes, among others, an optional parameter '`num_val`' which indicates the number of records to generate. For each value of '`num_val`', the script will create an atomic operation embedded within the simplest 1-layer neural network possible. The layer specific features (batch size, input shape, kernel size, strides, number of channels, padding, optimizer, activation, use of bias, etc.) are randomized from a range of realistic values. These 1-layer models are then executed using forward and backward passes and the median of execution times from multiple executions, along with the feature set for the 1-layer model are used to train a fully connected deep learning model. This model can then be used to get the predictions of execution times for a similar new operation based only on the same kind of feature set.

These predictions can thus be obtained for all the layers of a given DNN model and can be combined across layers to provide a prediction for the overall training time for the DNN model.

Each benchmark run will result in a dataset containing all the layer feature values and the time taken to execute the operations of the 1-layer model. The researchers benchmarked up to 50,000 combinations of layer features on each of their GPUs.

5.1.2 Data Preparation

After generating the benchmarking datasets for several GPUs, these datasets are combined and processed into a larger dataset that is ready for training. This includes creating additional features related to number of floating-point operations, memory usage and one-hot encoding of categorical features like the activation function, model optimizer, etc. Other features related to the hardware such as GPU bandwidth, number of GPU cores, GPU memory, etc. are also added to the dataset.

5.1.3 Model Training & Making Predictions

The prepared dataset is split into an 80:10:10 ratio for training, validation, and testing. A simple neural network model with 4-6 fully connected layers and a dropout layer is used to fit the data, depending whether the dataset is from a single GPU or multiple GPUs. The specific model configurations can be found in the "`model.py`" file. The resulting model will take layer features as input and predict the training time of that particular layer (using RMSE as an accuracy metric). The researchers achieved a model with an RMSE of 3.88 milliseconds on their benchmark data generated from multiple GPUs and showed that a fully connected DNN model outperforms a linear regression model.

Finally, this model can be used to make predictions for the run times of individual layers of a given DNN model. The total training time per batch for the overall model can then be derived from the sum of predicted execution time for each layer. The researchers built a separate python package for it, called 'mlpredict'. This package can be used to define a new Deep Learning model with a given architecture and save the model parameters as a .json file. This .json file can then be loaded and the predictions of run time for each layer for a given batch size on a given GPU can be easily obtained. This package uses the same model built using the researcher's benchmark data. This predicted time per batch can then be multiplied by the number of batches in the data and the number of epochs to be trained to get the final prediction of the overall training time for a model.

5.2 Replicating the Researcher's Results

While we were able to run the researcher's code on our systems and use their benchmark data to build the same model and get the predictions, we also wanted to generate benchmark data on our own and replicate the entire end-to-end process of data generation, model building and getting predictions of run time based on the GPUs we accessed. This involved migration of the code from TensorFlow 1.0 to TensorFlow 2.0 and setting up our own GPU environments to run the code.

5.2.1 Migration from Tensorflow 1 to Tensorflow 2

The original codes of the researchers were written in Tensorflow 1 and uses a docker environment to run the benchmarking script. We did not use docker in our implementation and instead migrated the original scripts from Tensorflow 1 to Tensorflow 2 using the `tf_upgrade_v2` command. This allows us to run the python scripts in Tensorflow 2 though there will be some deprecated warnings.

5.2.2 Accessing GPUs through Chameleon Cloud

Since we cannot use our client's GPU environments and we did not have a GPU on our local system, we were looking for an alternative GPU environment. Suitable for use, Chameleon is a large-scale, deeply reconfigurable experimental platform built to support Computer Sciences systems research. Community projects range from systems research developing new operating systems, virtualization methods, performance variability studies, and power management research to projects in software defined networking, artificial intelligence, and resource management. With the help from Professor Shlomo Argamon, we were able to access the bare metal instances on Chameleon Cloud and use the GPUs to run our benchmark scripts.

5.2.3 Setting up and using a working environment on Chameleon Cloud

As Chameleon Cloud is a shared platform, we must first make a node reservation for the particular type of resource we want to use. Once a reservation starts and becomes active, we need to launch a bare-metal instance on the leased node. The entire process of setting up a lease reservation is explained on their documentation website (https://chameleoncloud.readthedocs.io/en/latest/) which we were able to follow and set up a working GPU environment.

We ran the benchmark script to generate a total of 15k records of data from four types of GPUs available on Chameleon Cloud (Tesla K80, M40, P100 PCIE, and P100 SXM2). This was much lesser than the number of records the researchers generated (around 50k), as the data generation process itself was slow, and 2.5k records took 12 hours to generate.

We then transferred the generated data files to our local machines where we prepared and concatenated all the data. Following the researcher's method, we used our data to train a DNN, achieving a model with a test RMSE of 8.45 ms (our model is less accurate than the researcher's because our dataset is much smaller). Nonetheless, we were able to use the model to make single layer and full model training time predictions of a simple CNN model built for the MNIST dataset. The predictions closely resemble that of the actual runtimes, confirming that the researcher's method is valid, and we can replicate the researcher's experiment.

5.3 Tensorflow MirroredStrategy

Even though we were able to replicate the results of the original research paper and get the predictions of run time for a given DNN model, the client wanted to expand the project scope further. As model architectures and dataset sizes get larger, a single GPU may not be sufficient to train the model within a given time. Instead, multiple GPUs can be used simultaneously to train the model through distributed training. One of Tensorflow's inbuilt methodology for distributed training is called the MirroredStrategy (https://www.tensorflow.org/api_docs/python/tf/distribute/MirroredStrategy). This strategy is typically used for training on one machine with multiple GPUs. Overall, the time required for a model to train on 2 GPUs using MirroredStrategy is close to half the time it takes for training on a single GPU and so on.

5.3.1 Applying the TensorFlow MirroredStrategy:

As CCC uses MirroredStrategy to train deep learning models, the client wanted a solution that can provide run time estimates for models trained using this strategy. As the original paper did not mention any implementation of distributed computing, we had to come up with our own approach for the same, as mentioned below:

- Method 1: Using the same approach as the original research paper for MirroredStrategy
- Method 2: Combining the mirrored strategy approach with the original single-GPU approach
- Method 3: Applying a correction factor to the single GPU approach and combining it with the mirrored strategy approach.

5.3.2 Method 1: Similar approach as original research paper

As the client wanted to get the run time estimates for a model on the mirrored strategy, we followed the same approach to generate the benchmarking data. We modified the original script so that the individual atomic operations embedded within the 1-layer model are getting defined within the MirroredStrategy.

We generated 5k records of benchmarking data using this approach. The structure of this data was same as the data generated from the original script, and it contained the execution times for 1-layer models defined within the TensorFlow MirroredStrategy. We used this data to build our model in a similar manner as before and we were able to use our model to get the predictions of run times of a given DNN model using MirroredStrategy. We then recorded the actual training times for an MNIST data Convolutional model and compared them to the predicted times using our model. We

saw that the predicted training times were in 100's milliseconds per batch while the actual training times were <10 milliseconds per batch. We thus realized that this approach is not correct, and we modified our approach.

5.3.3 Method 2: Combining single-GPU approach and MirroredStrategy approach

We tried to explore further why the predicted run times were much higher than the actual run times and we realized that this was because of the way the Tensorflow MirroredStrategy works. The MirroredStrategy creates a model replica on each of the GPUs being used and processes a batch of the data separately on each GPU and then aggregates the output from each of the GPU to calculate the aggregated loss values and gradients. This aggregated gradient then gets pushed back to the model during backpropagation.

However, we realized that this extra step of aggregation of the output is only happening at the last layer and all the layers above the last layer are performing the same operations as they were performing on a single GPU. Hence, it does not make sense to benchmark the layers above the last layer using the MirroredStrategy. We thus decided to modify our approach so that we are benchmarking only the last layer using MirroredStrategy and the layers above the last layer using the original single GPU strategy, after modifying the batch sizes appropriately.

Using this approach, we generated benchmarking data for the last layer using Mirrored Strategy and used this data to build a model that can provide predictions of run time for the last layer of a model run on MirroredStrategy. As we were benchmarking only the last layer of a DNN, we reduced the number of output dimensions of the 1-layer model by reducing its randomization range to be only between 1 to 10. We generated total 3k records for benchmarking the last layer using MS.

The total run time for a model per batch using MS is broken down as the sum of the run time of the last layer predicted using MS benchmark data model and the run time of all the layers above the last layer using the original single GPU approach, after adjusting for the appropriate batch sizes. Thus, for a model being trained on a batch size of 64(Global Batch Size) using MirroredStrategy on 2 GPUs, the predicted run time per batch will be the sum of the predicted run time of the last layer on MS for batch size 64 and the run times for the remaining layers using the single GPU approach on a batch size of 32(Batch Size per replica = 64/2).

We used this approach to get the predicted run time for a much larger dataset this time, the Patch Camelyon dataset. The PatchCamelyon benchmark is a new and challenging image classification dataset. It consists of 327,680 color images (96 x 96 px) extracted from histopathologic scans of

lymph node sections. Each image is annoted with a binary label indicating presence of metastatic tissue. The dataset size is 7.48 GB, divided into a train, test and validation split of 80:10:10.

We compared the actual v/s the predicted run times for the model on MS for various batch sizes. However, even though the predicted run time was in the same range as the actual run times, the predicted run time was consistently about 35% more than the actual run time for larger batch sizes.

To investigate this further, we got the actual run times for the individual layers of the model on a single GPU. As per the original research paper, the sum of these run times should be close to the actual run time of the entire model on a single GPU. However, we found that the sum of the run times for all the layers is consistently about ~1.5 times the actual run time for the model. Thus, this proves that when benchmarking a single layer, there may be an overhead which disappears when the layers are combined to form a full model.

5.3.2 Method 3: Applying a correction factor to the single GPU approach and combining it with the mirrored strategy approach

We applied this observation while getting the predicted run times for the initial layers. We divided the predicted run times of all but the last layer(derived from the single GPU approach) by the correction factor of 1.5 and the added them to the prediction of the run time of the last layer predicted using MS.

After applying this correction factor, we found that the predictions are much more accurate. Except for the batch size of 8, the actual run times for other batch sizes were very close to the predicted run times.

5.3.3 Correction Factor of 1.5

Even though we can get better predictions after applying the correction factor, there are the below questions we are not able to answer:

- Is the factor 1.5 only applicable to this particular machine or type of GPU?
- Does this factor change for a different model architecture/dataset?
- What is the exact root cause for this overhead? Can this overhead factor be predicted for a given model/dataset/GPU?

Answering these questions will require conducting many more experiments by comparing the sum of predicted runtimes v/s actual run time for various datasets, models trained on different GPU environments and we will discuss with the clients on the next steps for this.

5.4 Results

(Note: The codes for getting the graphs or the numbers for the graphs displayed below are provided at our GitHub repository: https://github.com/profnote/ml-performance-prediction)

We present here results for training models using the features we have defined previously. We first demonstrate the replication of the results from the original paper. We then compare our full-model predictions v/s the actual run times for a model on a single GPU using a simple CNN model for the MNIST data.

5.4.1 Exploratory graph of the run-time of 1-layer models using the data generated from the original benchmarking script from the researchers' GitHub repo.

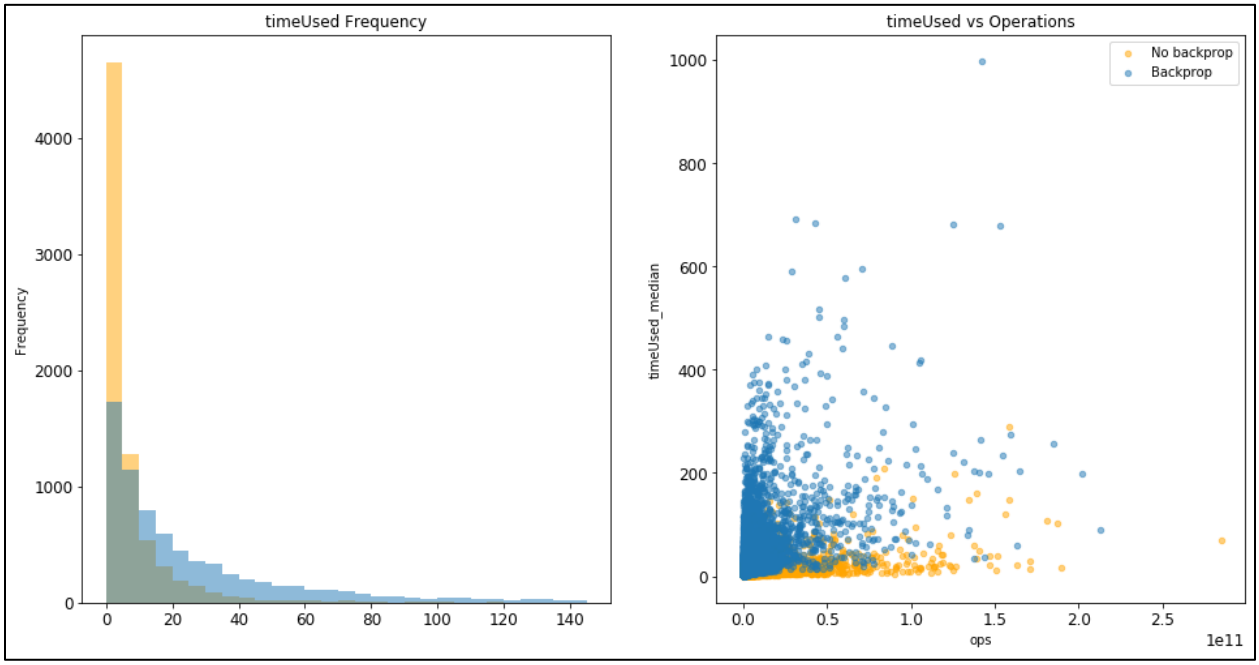
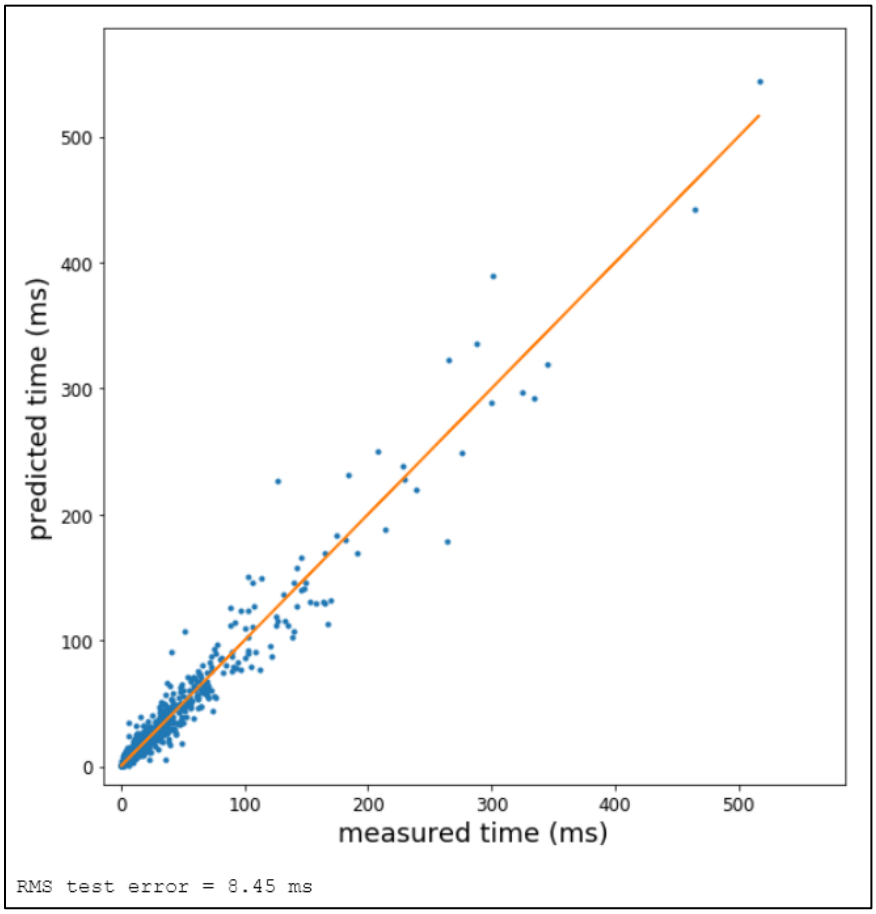


Fig.(left):Histogram plot of run-time for execution of 1-layer models('timeUsed' in milliseconds)
Fig.(right): Scatter plot of run-time v/s the floating-point operations for the 1-layer models

5.4.2 Comparison of actual run time v/s predicted run-time using the model built on the benchmark data we generated



5.4.3 Using the model built from the benchmark data we generated, we compared full-model run time predictions for a simple MNIST data CNN model by combining the predicted run-time of each layer and compared it to the actual run time for the model for various batch sizes. We trained the model on 10 epochs for various batch sizes from 2 to 128 and calculated the training time per epoch per batch (shown in the graph below). We also compared the actual run times to the predicted run times from the researcher's original model from the github repo. Below are the results:

```
model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 32)        9248
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 32)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 3, 3, 32)          9248
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 1, 1, 32)          0
_________________________________________________________________
flatten (Flatten)            (None, 32)                0
_________________________________________________________________
dense (Dense)                (None, 128)               4224
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 24,330
Trainable params: 24,330
Non-trainable params: 0
_________________________________________________________________
```

Fig. Model architecture used for building the MNIST data CNN model.

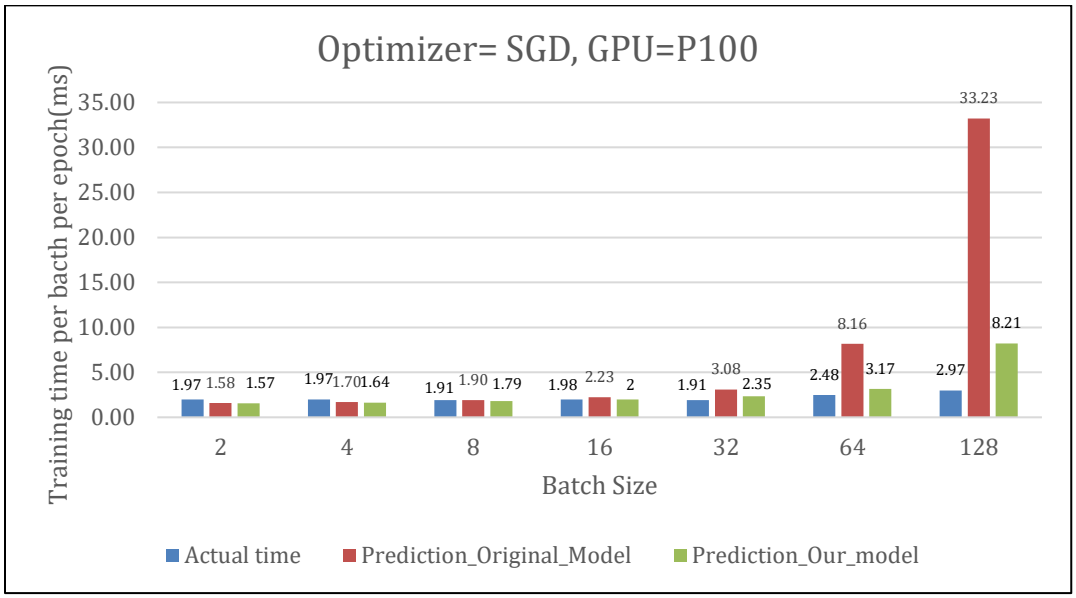


Fig. Comparison of predicted v/s actual training time for the MNIST data CNN model using the SGD optimizer on the P100 GPU

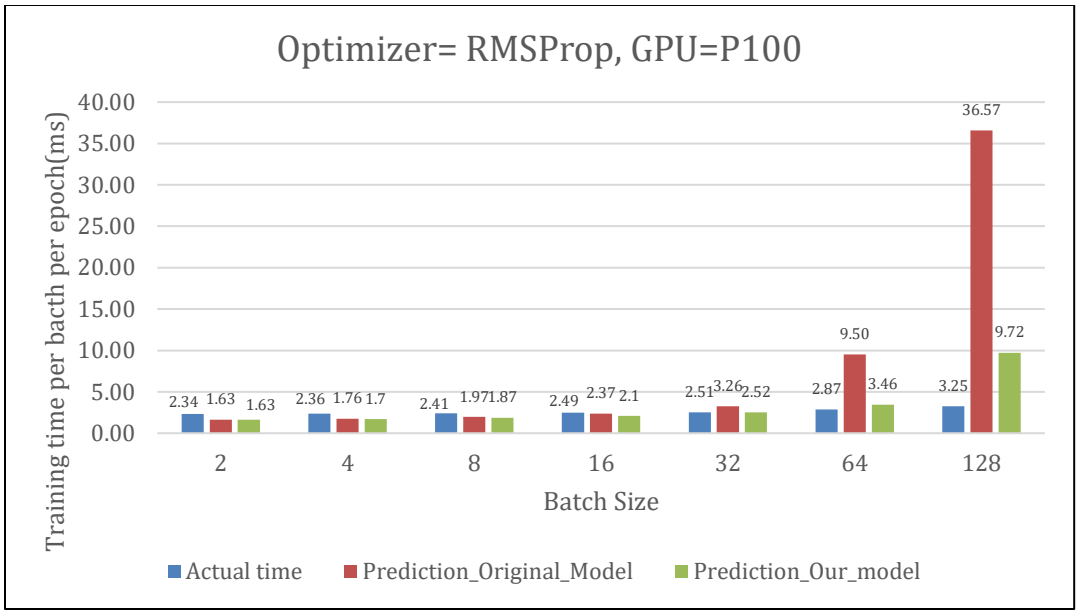


Fig. Comparison of predicted v/s actual training time for the MNIST data CNN model using the RMSprop optimizer on the P100 GPU

This graph shows that the predicted run times are very close to the actual run times for lower batch sizes and the predictions are diverging for larger batch sizes. This is because the data generation script we used generated data for batch sizes only up to 64 and hence our model is not able to predict correctly for batch sizes of 128, which is outside the range of batch sizes used in the training data. Similarly, as the researcher's model was trained on data which only contained batch sizes up to 32, their model is unable to predict correctly for batch sizes higher that 32, which is outside the range of batch sizes used in the training data.

5.4.4 Exploratory graph of the run-time of 1-layer models executed using MirroredStrategy.

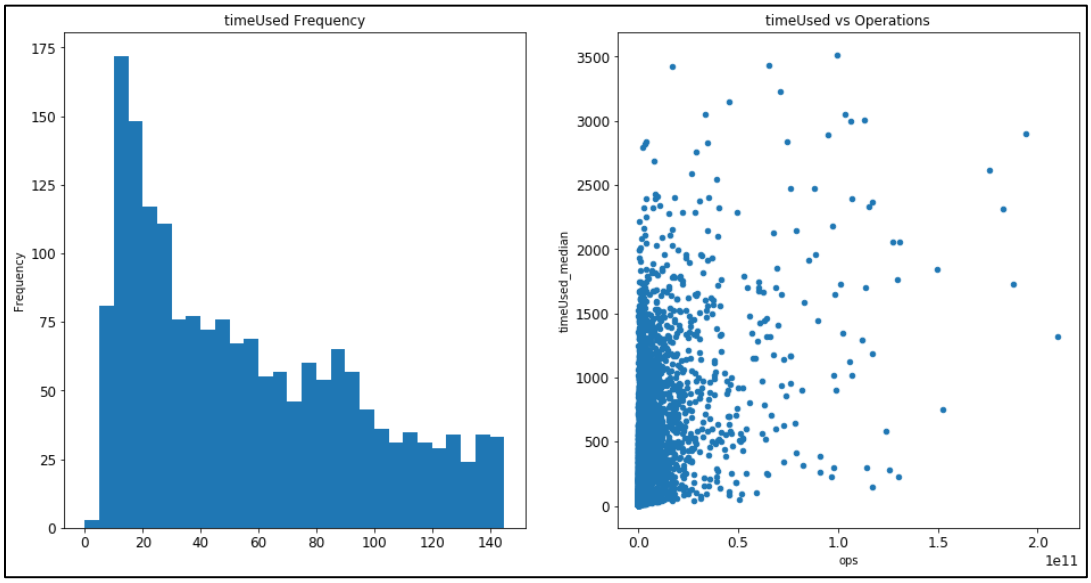
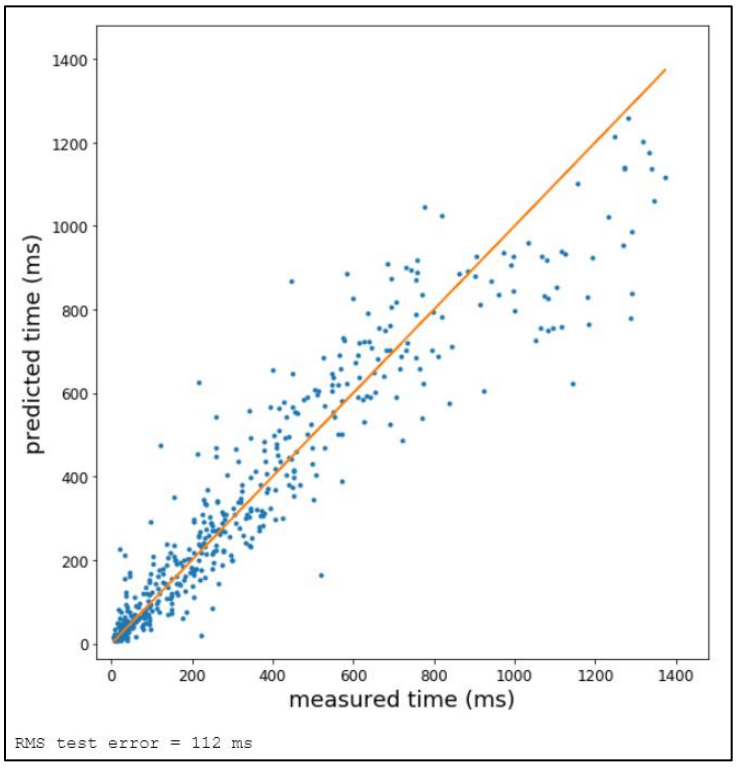


Fig.(left):Histogram plot of run-time for execution of 1-layer models(‘timeUsed’ in milliseconds) executed using MirroredStrategy

Fig.(right): Scatter plot of run-time of 1-layer models v/s the number of floating point operations

5.4.5 Comparison of actual run time v/s predicted run-time using the model built on the benchmark data for benchmarking run-time of 1-layer models executed using MirroredStrategy



5.4.6 Comparison of actual run time v/s predicted run-time(time per batch in milliseconds) for the MNIST data CNN model on Mirrored Strategy using the model built on the benchmark data we generated

Batch_Size	Actual_Run_Time	Predicted_Run_Time
2	3.85	162.89
4	3.84	196.51
8	3.88	248.9
16	3.8	310.03
32	3.84	356.58
64	4.09	500.8

This shows that the predicted run time is much higher than the actual run time. Hence, we had to modify this approach.

5.4.7 Exploratory graph of the run-time of 1-layer models using the data generated from the original benchmarking script from the researchers' GitHub repo

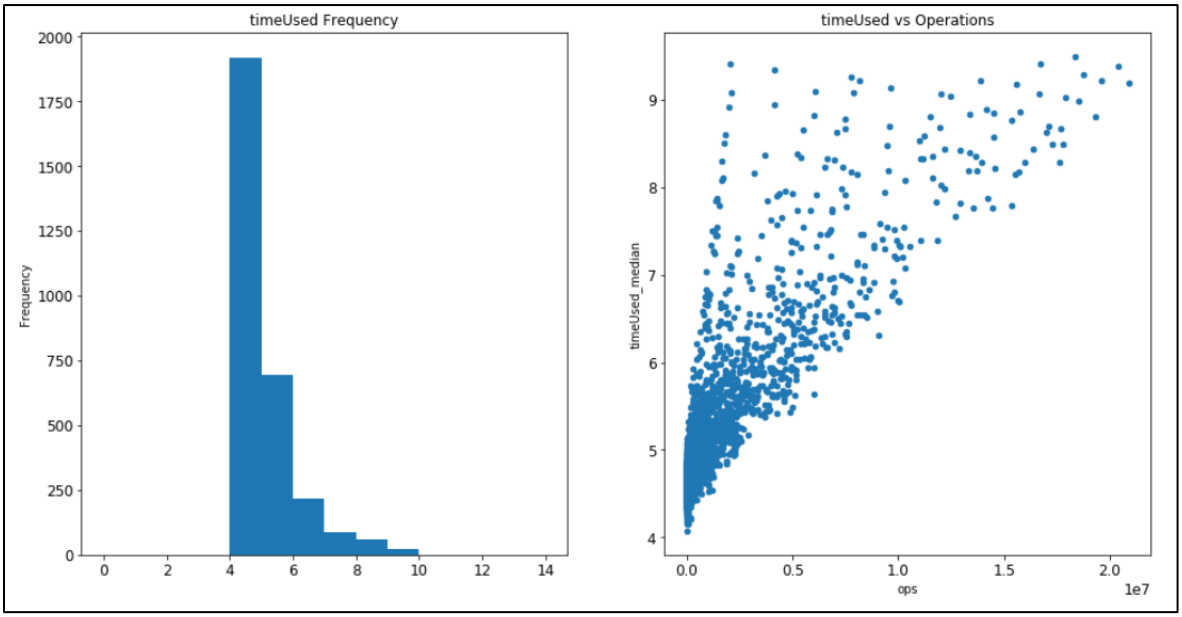
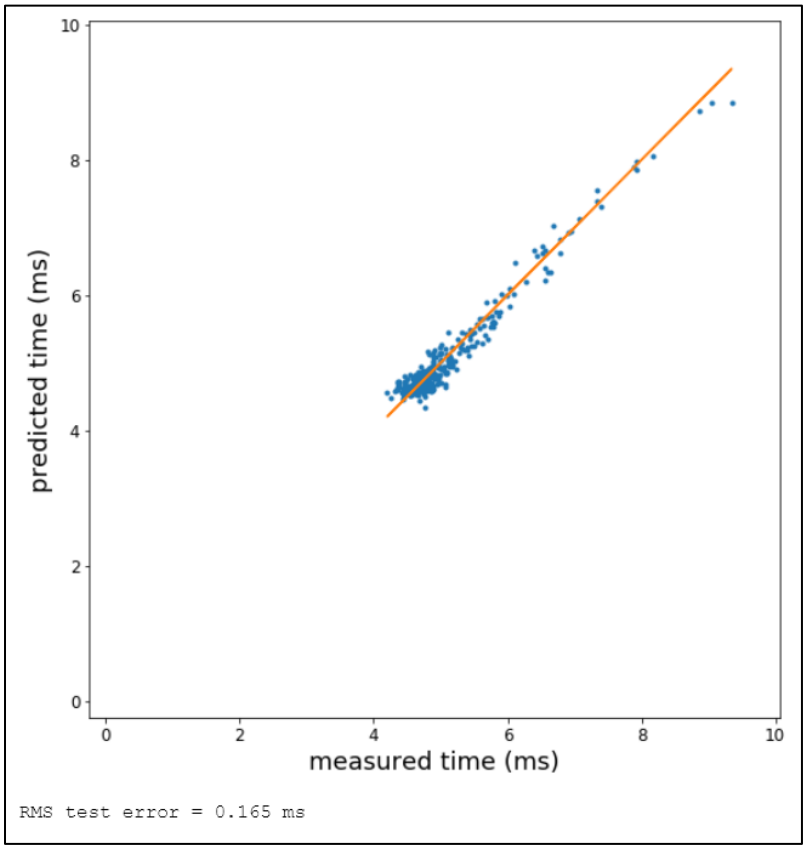


Fig.(left):Histogram plot of run-time for execution of 1-layer models('timeUsed' in milliseconds) executed using MirroredStrategy
Fig.(right): Scatter plot of run-time of 1-layer models v/s the number of floating point operations

5.4.8 Comparison of actual run time v/s predicted run-time using the model built on the benchmark data for benchmarking run-time of 1-layer models executed using MirroredStrategy

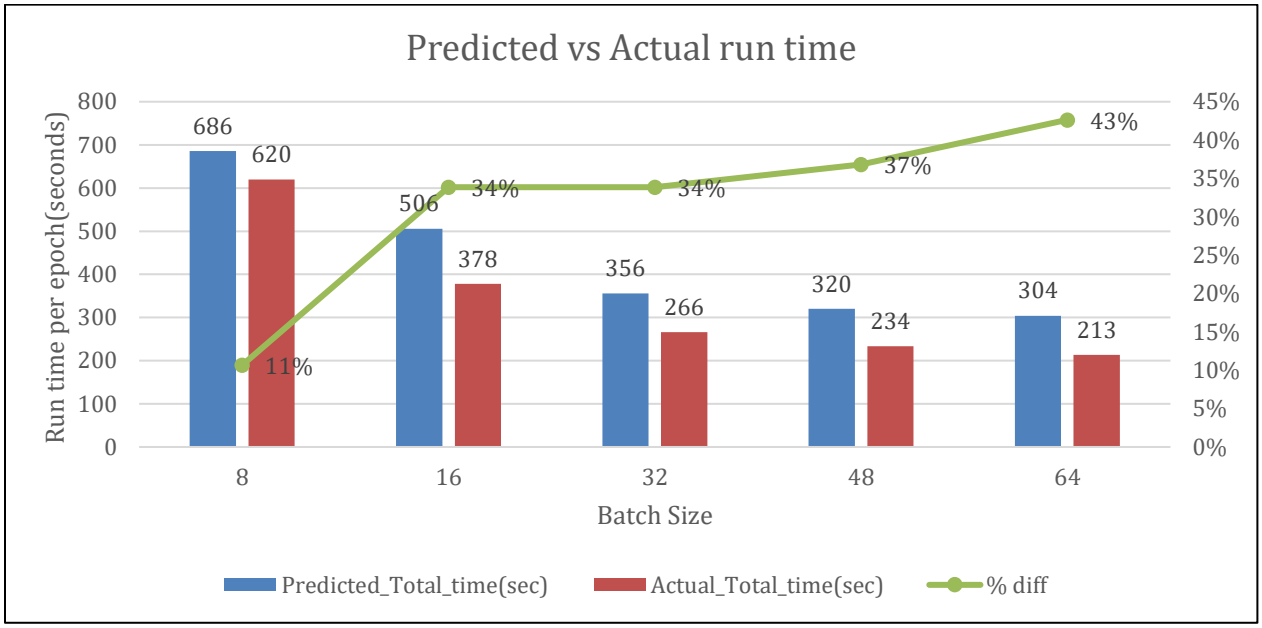


5.4.9 Using the model built from the benchmark data we generated, we compared full-model run time predictions for a larger CNN model for the Patch Camelyon dataset by combining the predicted run-time of the last layer using MirroredStrategy and the run times for the remaining layer using the original single GPU approach. Below are the results:

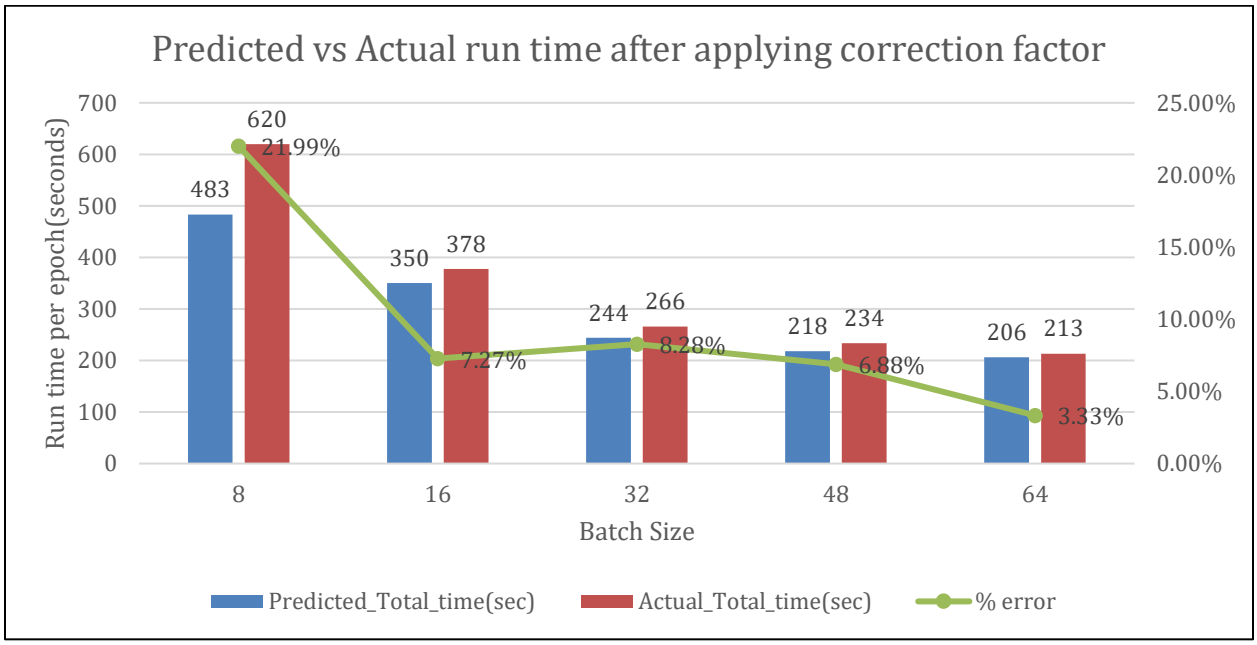
```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 96, 96, 256)       7168
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 48, 48, 256)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 48, 48, 256)       590080
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 24, 24, 256)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 24, 24, 512)       1180160
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 12, 12, 512)       0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 12, 12, 512)       2359808
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 6, 6, 512)         0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 6, 6, 1024)        4719616
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 3, 3, 1024)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 3, 3, 1024)        9438208
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 1, 1, 1024)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_2 (Dense)              (None, 1028)              1053700
_________________________________________________________________
dense_3 (Dense)              (None, 512)               526848
_________________________________________________________________
dense_4 (Dense)              (None, 128)               65664
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 129
=================================================================
Total params: 19,941,381
Trainable params: 19,941,381
Non-trainable params: 0
_________________________________________________________________
None
```

Model Architecture used for the CNN model using the Patch Camelyon dataset.

Comparison of actual run times v/s predicted run times for the Patch Camelyon data for run time per epoch



Comparison of actual run times v/s predicted run times for the Patch Camelyon data for run time per epoch after applying the correction factor



5.5 Conclusion

The overall conclusions from the project results are as follows:

- We are able to provide reasonably accurate predictions of run time for a DNN model using the TensorFlow MirroredStrategy, as demonstrated in the case of the model trained on the Patch Camelyon dataset.
- However, we are still not able to explain the root cause for the correction factor of 1.5, which is required to correct the predictions of training time of the initial layers of the model.
- It will be required to conduct multiple experiments to understand the reason behind the requirement of the correction factor of 1.5, and we will discuss further with the clients on the next steps on this
- The remaining steps for this project are to finalize our model and package it within a tool which can be used to make predictions of run time for a given DNN model using MirroredStrategy

6. Project Retrospective

6.1 Suggestions for Improvements

- Perform further hyper-parameter tuning on both the single GPU and MirroredStrategy models to see if the accuracy can be improved.
- Investigate the root cause of the overhead of benchmarking individual layer times. Is the correction factor of about 1.5 the same for different model architectures, datasets and GPU types? Can this correction factor be calculated or predicted?
- Embed the current models in a python package/tool that can take the model parameters, dataset shape, and GPU specs as input and get the predicted training times (including the adjustment for the correction factor).

6.2 Team Thoughts and Lessons Learned

We are very grateful to Professor Shlomo Argamon, Manoochehr Assa, and Husam Sweidan for creating this great project and giving us this invaluable learning experience. Over the course of of project we have acquired many new technical skills, including how to use a cloud platform (Chameleon Cloud) and setting up the compatible Tensorflow and Cuda environments, working on the linux terminal, working with distributed training via Tensorflow MirroredStrategy, and how to conduct experiments following a scientific process.

One of the most important lessons we learned is how we could have planned our project scope and timeline to better account for the risks that were going to happen. For example, we considered the risk of not being able to use the client's system for the project, but did not consider how it would affect our timeline. In the end, most of the project work was not following the original timeline, and we did not complete all the tasks we had originally planned. We realized that we should have designed a more flexible project timeline, one that considers the risks and obstacles that we may encounter so that we are able to manage our time more efficiently.

7. Project Support

We are both eager to continue to work on this project if given the opportunity, as there is still room for improvement. There are still experiments we wish to run in order to identify the cause of the overhead in single layer benchmarking, or possibly even finding a better alternative method that can benchmark the run times without the overhead. We believe that solution is not far beyond our reach, and would like to close the project by being able to deliver a working prototype that CCC could use.