Using GPUs for Convolutional Neural Network Training with Mnist Data-set

Steven Ramirez Rosa
Department of Computer Science and Engineering
University of Puerto Rico, Mayaguez Campus
Mayaguez, Puerto Rico
steven.ramirez3@upr.edu

***Abstract*—Convolutional Neural Networks play an important role in the field of Image Classification. Despite their fundamental role, some architectures can take a lot of time to train. In this work, a time analysis of an existing Convolutional Neural Networks using CPU and GPUs will be presented. The proposed approach has several advantages, since as the network architecture and the data-set grows, the GPUs become more efficient. The network is tested using 32, 64, 128, 256, 512, 1024, and 2048 batch sizes and 10 epochs on 1 Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz CPU, 1 Nvidia Tesla P100 GPU, and 2 Nvidia Tesla P100 GPUs. Changes were done to the source code to support GPUs and display the results. Our results show that using GPUs for training is faster than using a CPU and yield better accuracy and loss result. The GPU was up to 36 times faster than the CPU, contained up to 1.6 times less loss and up to 1.005 times more accuracy.**

Keywords—Convolutional Neural Networks, CNN, Machine Learning, Deep Learning, CPU, GPU, Intel, Nvidia, Image Classification, TensorFlow, Keras, Mnist

I. Introduction

Technology has evolved to an extent where machines surpass humans in certain areas. One example in the computer science field is the area of Image Classification. Contextual Image Classification is defined as an approach of classification based on contextual information in images [1]. One important feature is that machines can classify images by learning from previous information to an extent where they can do it without human supervision. They can accomplish this by using Artificial Neural Networks. Artificial Neural Networks are machine learning techniques that simulate the mechanism of learning in biological organisms [2]. Even though machines classify images more efficiently than humans, there are certain limitations.

Some of the biggest problems that systems confront are related to accuracy, storage and execution time. Since the main objective is to classify images correctly, the accuracy of the system must be as highest as possible. This requires that the machine receive more information as reference for the training, therefore, requiring more storage. By increasing the information that the machine must analyze, more computational time is required.

Intrigued by this problem, we start to research new solutions for reducing the training execution time. To that end, we modify an existing Convolutional Neural Network algorithm to support GPUs and test it in different processing units, CPU and GPU, to study their behaviors and try to determine on which environment they perform better. The goal is to achieve the best of two worlds, reduce execution time without sacrificing to much accuracy and loss.

II. Mnist Algorithm

This section introduces the existing MNIST algorithm [3]. The algorithm consists of three parts: 1) Data prepossessing; 2) the neural network architecture; and 3) the training function. The rest of the section will introduce the three functions in details.

A. Data Prepossessing

First, the data-set used in this algorithm is the MNIST data-set. MNIST database of handwritten digits [4] is a data-set that contains 70,000 28x28 gray-scale images of the 10 digits, which are from zero to nine. It has a training set of 60,000 examples and a test set of 10,000 examples. It is a subset of a larger set available from NIST [5]. First, the algorithm splits the data-set between the training and test sets. Then, it re-shapes the images depending of the image's format. Finally, it convert class vectors to binary class matrices. The data prepossessing covers from line 16 to line 45 of the Mnist final code.

B. Neural Network Architecture

The architecture consist of two convolutional, one max pooling, two dropouts, one flatten, and two dense layers. The network starts by receiving a 28x28 gray-scale image. Then, it creates two 2-dimension convolutional layers with a 3x3 kernel, 28x28 input shapes, and 32 and 64 depths respectively. The spatial size is reduced by half using a 2-dimension max pooling layer with a 2x2 pool size. Afterwards, a dropout layer with a .25 rate is created to reduce over-fitting. A flatten layer is created to collapse the spatial dimension of the input. Then, a dense layer with 128 neurons using a relu function is created. Another dropout layer with a .50 rate is created to further reduce over-fitting. Finally, a dense layer with 10 neurons using a softmax function is created, each one representing a different class. The Neural Network architecture covers from line 47 to line 59 of the Mnist final code. Figure 1 shows an AlexNet representation of this architecture.

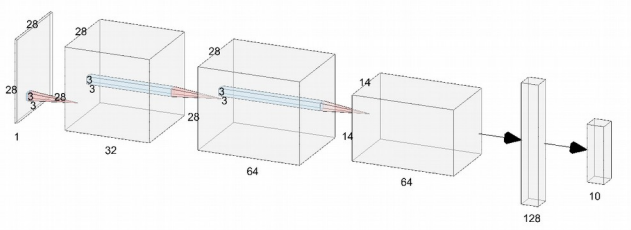

Fig. 1. AlexNet style representation of the neural network architecture

C. *Training Function*

The model is compiled using categorical cross-entropy for the loss, Adelta for the optimizer, and accuracy for the metrics and is trained using the training sets and validated using the test sets. The training function covers from line 61 to line 78 of the Mnist final code.

III. Modifications to the Source Code

For the purpose of this work, several modifications were made to the source code. To add GPUs support, we added lines 63 to 67. The idea is to use Tensorflow mirrored strategy [6] to convert the model to a model that supports an established numbers of GPUs. Finally, we created multiple prints to display the results of the accuracy, loss and execution time which are lines 80 to 84.

Mnist Final Code

```
1.  from __future__ import print_function
2.  import keras
3.  from keras.datasets import mnist
4.  from keras.models import Sequential
5.  from keras.layers import Dense, Dropout, Flatten
6.  from keras.layers import Conv2D, MaxPooling2D
7.  from keras import backend as K
8.  from keras.utils import multi_gpu_model
9.  import tensorflow as tf
10. import time
11.
12. batch_size = 2048
13. epochs = 10
14. num_GPUs = 2;
15.
16. # part 1: Data prepossessing
17.
18. # input image dimensions
19. img_rows, img_cols = 28, 28
20. num_classes = 10
21.
22. # the data, split between train and test sets
23. (x_train, y_train), (x_test, y_test) =
            mnist.load_data()
24.
25. # Re-shape images
26. if K.image_data_format() == 'channels_first':
27.     x_train = x_train.reshape(x_train.shape[0], 1,
                                img_rows, img_cols)
28.     x_test = x_test.reshape(x_test.shape[0], 1,
            I                   mg_rows, img_cols)
29.     input_shape = (1, img_rows, img_cols)
30. else:
31.   x_train = x_train.reshape(x_train.shape[0],
                        img_rows, img_cols, 1)
32.     x_test = x_test.reshape(x_test.shape[0],
                        img_rows, img_cols, 1)
33.     input_shape = (img_rows, img_cols, 1)
34.
35. x_train = x_train.astype('float32')
36. x_test = x_test.astype('float32')
37. x_train /= 255
38. x_test /= 255
39. print('x_train shape:', x_train.shape)
40. print(x_train.shape[0], 'train samples')
41. print(x_test.shape[0], 'test samples')
42.
43. # convert class vectors to binary class matrices
44. y_train = keras.utils.to_categorical(y_train,
                                num_classes)
45. y_test = keras.utils.to_categorical(y_test,
                                num_classes)
46.
47. # part 2: Neural network Architecture
48.
49. model = Sequential()
50. model.add(Conv2D(32, kernel_size=(3, 3),
51.                 activation='relu',
52.                 input_shape=input_shape))
53. model.add(Conv2D(64, (3, 3), activation='relu'))
54. model.add(MaxPooling2D(pool_size=(2, 2)))
55. model.add(Dropout(0.25))
56. model.add(Flatten())
57. model.add(Dense(128, activation='relu'))
58. model.add(Dropout(0.5))
59. model.add(Dense(num_classes,
                activation='softmax'))
60.
61. # part 3: Training function
62.
63. # Run on GPU
64. strategy = tf.distribute.MirroredStrategy()
65. with strategy.scope():
66.             model = multi_gpu_model(model,
                                gpus=num_GPUs)
67.             model.compile(loss =
            keras.losses.categorical_crossentropy,
            optimizer = keras.optimizers.Adadelta(),
            metrics=['accuracy'])
68.
69. # Run on CPU
70. #model.compile(loss=keras.losses.categorical_cross
                    entropy, optimizer =
                    keras.optimizers.Adadelta(),
                    metrics=['accuracy'])
71.
72. start = time.time()
73. model.fit(x_train, y_train,
74.           batch_size=batch_size,
75.           epochs=epochs,
76.           verbose=1,
77.           validation_data=(x_test, y_test))
78. end = time.time()
79.
80. total = end - start
81. print('Training time:', total)
82. score = model.evaluate(x_test, y_test, verbose=0)
83. print('Test loss:', score[0])
84. print('Test accuracy:', score[1])
```

IV. Experimental Results

In this section, we will show three sets of experimental results to evaluate the performance of the algorithm: 1) In the first set of experiments, we evaluate the effectiveness on a CPU; 2) In the second set of experiments, we evaluate the effectiveness on a GPU; and 3) In the last set of experiments, we evaluate the effectiveness on 2 GPUs. Every experiment is run using 10 epochs.

A. Computing Platform

We performed the experiment on the "22805a4a-e695-4b48-86b9-bcf5cd2074de" node located in Chitacc servers from Chameleoncloud. This node is located in a PowerEdge R730 HPC system and consist of 2 Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz CPU with 48 threads, 128 GiB of RAM, and 2 Nvidia p100 GPUs. Figure 2 [7] shows detailed information about the system.

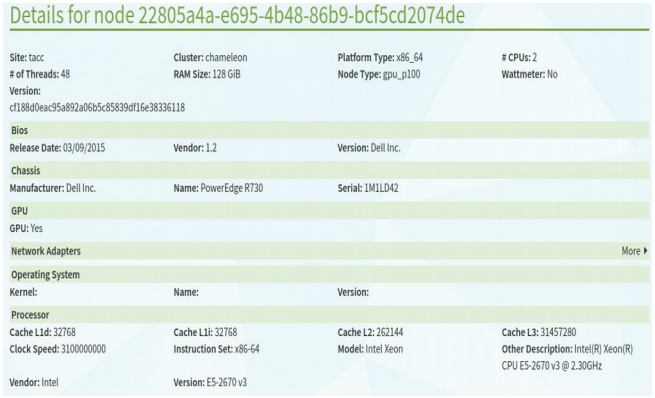

Fig. 2. More details about the node.

B. Execution Time

In terms of execution time, every experiment presented the same behavior. The CPU went from 790 to 506 seconds, the single GPU went from 119 to 17, and the 2 GPUs went from 180 to 14. We can see that the GPU performs way better than the CPU, achieving a speed-up of up to 36 times. Also, we can determine that for the CPU, 512 batches is the last point where there is a significant decrease in execution time and for the GPUs is 256. Finally, 1 GPU performs better than 2 GPUs until the batch size is 2048. This information is presented in table 1 and figure 3.

C. Accuracy

In terms of accuracy, every experiment presented similar behaviors. The CPU went from 99.09 to 98.01 percentage, the single GPU went from 99.00 to 98.41, and the 2 GPUs went from 98.90 to 98.37. We can see that the GPU performs better than the CPU from a batch size of 128. Also, that 2 GPUs and the CPU present the similar erratic behavior, while 1 GPU has a more predictable behavior. Another interesting fact is that at a batch size of 1024, the accuracy of the 3 experiments almost converge. Finally, most of the time, 1 GPU has more accuracy than 2 GPUs. This information is presented in table 2 and figure 4.

D. Loss

In terms of loss, the behavior is similar to the behavior in accuracy. The CPU went from 0.02910 to 0.05927 loss, the single GPU went from 0.02925 to 0.04666, and the 2 GPUs went from 0.03974 to 0.05034. We can see that the GPU performs better than the CPU from a batch size of 128. Also, that 2 GPUs and the CPU present the similar erratic behavior, while 1 GPU has a more predictable behavior. At a batch size of 1024, the loss of the 3 experiments converge. Finally, most of the time, 1 GPU has less loss than 2 GPUs. This information is presented in table 3 and figure 5.

TABLE I. EXECUTION TIME

Batch Size	Execution Time		
	CPU	*1xGPU*	*2xGPUs*
32	790.19	119.89	180.50
64	640.23	68.56	94.22
128	585.06	41.34	52.05
256	556.50	28.07	30.85
512	517.46	22.63	21.87
1024	512.07	19.14	17.79
2048	506.80	17.64	14.67

TABLE II. ACCURACY

Batch Size	Accuracy		
	CPU	*1xGPU*	*2xGPUs*
32	0.9909	0.9900	0.9890
64	0.9921	0.9917	0.9906
128	0.9905	0.9918	0.9923
256	0.9904	0.9916	0.9906
512	0.9858	0.9909	0.9889
1024	0.9887	0.9891	0.9894
2048	0.9801	0.9841	0.9837

TABLE III. LOSS

Batch Size	Loss		
	CPU	*1xGPU*	*2xGPUs*
32	0.02910	0.02925	0.03974
64	0.02507	0.02829	0.02793
128	0.02992	0.02671	0.02469
256	0.02786	0.02716	0.02869
512	0.04480	0.02679	0.03397
1024	0.03195	0.03194	0.03188
2048	0.05927	0.04666	0.05034

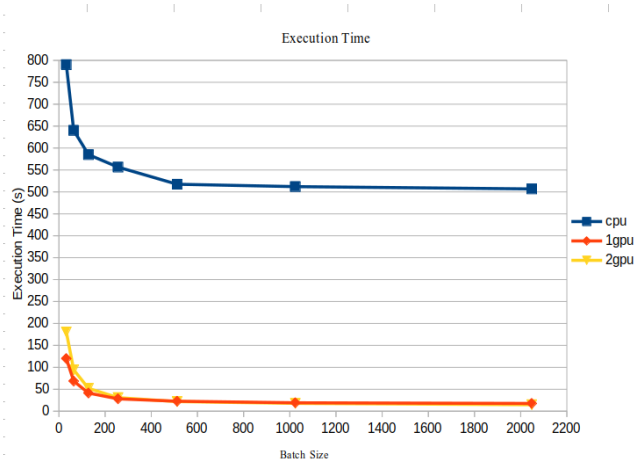

Fig. 3. Graphical representation of the execution time.

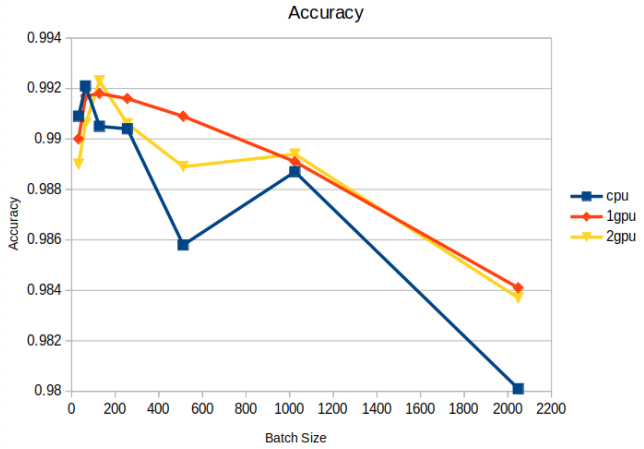

Fig. 4. Graphical representation of the accuracy.

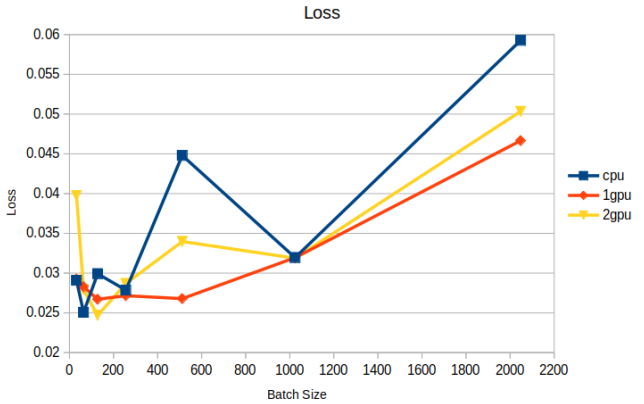

Fig. 5. Graphical representation of the loss.

V. Conclusion

In this work, we seek to reduce the execution time of an existing Convolutional Neural Network algorithm by using GPUs. We first modify the base code to add support to GPUs and print the results. Then, we train the neural network on 1 CPU, 1 GPU and 2 GPUs using different batch sizes. Compared to the CPU, the GPU was up to 36 times faster, contained up to 1.6 times less loss and up to 1.005 times more accuracy. When comparing the results of 1 GPU and 2 GPUs, 1GPU, on average, performed better. Our future work along this line is to determine, on general, how the architecture of a Convolutional Neural Network impact the results and why, in this case, 1 GPU performed better than 2 GPUs.

References

[1] Microsoft Academic. *Contextual Image classification, 2018*. Available: https://academic.microsoft.com/topic/75294576

[2] C. Aggarwal, Neural Networks and Deep Learning, 1rd ed. Switzerland: Springer, pp.1.

[3] Keras. *Mnist cnn,* 2019. Available: https://keras.io/examples/mnist_cnn/

[4] Keras. *Datasets,* 2019. Available: https://keras.io/datasets/

[5] NIST. *NIST Special Database 19,* 2019. Available: https://www.nist.gov/srd/nist-special-database-19

[6] TensorFlow. *Distributed Training with TensorFlow,* 2019. Available: https://www.tensorflow.org/guide/distributed_training

[7] ChameleonCloud. *Details for Node 22805a4a-e695-4b48-86b9-bcf5cd2074de*, 2019. Available: https://www.chameleoncloud.org/hardware/node/sites/tacc/clusters/chameleon/nodes/22805a4a-e695-4b48-86b9-bcf5cd2074de/