Analysis Between Serial and Parallel Implementation for a Machine Learning Model

Saaif Ahmed

Rensselaer Polytechnic Institute

ahmeds7@rpi.edu

Abstract:

Machine learning is a complex field with many requirements. Not only does it have a high skill or technical level to master and understand its complexities but there is a mechanical need that must be met in order for any progress to be made. In this paper we analyze the difference in accuracy and efficiency within 2 different implementations of a simple logistic regression machine learning problem. Our two implementations are serial versus parallel both written in Python. The serial implementation will use NumPy, a Python package for complex mathematical functions, and standard python functions apart of the library. The parallel implementation will use Tensorflow 2.X, a powerful Python package optimized for machine learning which is optimized for operation on NVIDIA GPUs where operations are performed in parallel. We compare speed and accuracy results from both models, and describe the motivations for each implementation. Then we describe the future of parallel programming for machine learning.

1: Introduction

The purpose of this analysis is to tackle the computational problems surrounding machine learning. From a pure computer science "O-notation" standpoint, the machine learning problem is quite simple. A machine learning model that we implement is of depth 1 and loops iterations through the whole dataset. Thus it is $O(n * 1 * k)$ where n is the size of the dataset and k is a constant number of iterations. Thus this is a linear time implementation of solving the problem.

However, this issue becomes massively apparent when dealing with extremely large datasets, as per normal in machine learning problems. Furthermore, because iterations can be in the hundreds or thousands, the time complexity of the learning process can become orders of magnitude larger. A solution needs to be implemented and approaching the learning process from a parallel point of view, we argue, is a very good choice.

2: Problem Statement

For a scenario where the classes $y \in \{1, \dots, K\}$ we can use a multi-classifier logistic regression learning model to build a reasonably accurate system that can predict y by using a series of weights $\vec{w}_1, \dots, \vec{w}_k$ in a matrix Θ such that if the data set is $N \times M$ then $W^{M \times K}$ with each $\overrightarrow{w_i}^{M \times 1}$and passing them into a soft max function for logistic regression.

In order to build towards an accurate model we must minimize the negative logarithmic likelihood of these weights against the training sample to make a prediction. In this case we have

our data set $D^{M\times N}$ that has outputs $y \in \{1,2,3,4,5\}$ and input parameters $\vec{x}[m]$. The output y follows the 1 of K encoding and as such is an output vector $\vec{y}[m]$. In other words $D = \{\vec{x}[m], \vec{y}[m]\}, m = 1,2,..,M$. We have a matrix $\Theta = (\vec{w}_k, w_{k_0})$, where k=1,2,3, 4, 5, $\mathbf{W_k}$ is a vector for kth discriminant function, and $W_{k,0}$ is its bias. We define the loss function as:

$$L(D:\Theta) = -\sum_{m=1}^{M}\sum_{(k=1)}^{K} \vec{y}[m][k] \log \sigma_M(\vec{x}^T[m]\Theta_k)$$

Such that $\sigma_M()$ is the soft-max function.

$$\sigma_M(\vec{x}^T[m]\Theta_k) = \frac{e^{\vec{x}^T[m]\Theta_k}}{\sum_{k=1}^{K} e^{\vec{x}^T[m]\Theta_k}}$$

To build the model over many iterations we build the gradient descent method for the logistic regression. We define the gradient of the loss function as:

$$\nabla_\theta L(D:\Theta) = [\nabla_{\Theta_1} L(D:\Theta_1), \dots, \nabla_{\Theta_k} L(D:\Theta_k)]$$

With the gradient of each Θ_k computed as:

$$\frac{\partial L(D:\Theta)}{\partial \Theta_k} = -\sum_{m=1}^{M} \{\vec{y}[m][k] - \sigma_M(\vec{x}^T[m]\Theta_k)\}\vec{x}[m]$$

With this gradient we can update the Θ matrix at iteration t as

$$\Theta^t = \Theta^{t-1} - \eta \nabla_\theta L(D:\Theta)$$

With η as a hyper parameter of the model known as the "learning rate".

In the report below we analyze the performance of an implemented multi-classifier logistic regression model on the MNIST dataset. The objective is to build a model that accurately predicts the numbers on an image that is 28 by 28 pixels. The image is grayscale and is unrolled into a 784x1 vector. For the training set there are 25112 images. Thus our input data $X^{25112\times 784}$ with output labels $\vec{y}^{5\times 1}$. As such $\Theta^{785\times 5}$ with each $\vec{w}_k^{785\times 1}$ since each $\overrightarrow{w_k}$ has an additional w_0 bias.

3: Serial Implementation

```
import pathlib
import matplotlib.pyplot as plt
import numpy as np
import pickle
import random as r
import tensorflow as tf
import pandas as pd
import math
```

```
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"

# read images
def load_images(path_list):
    number_samples = len(path_list)
    Images = []
    for each_path in path_list:
        img = plt.imread(each_path)
        # divided by 255.0
        img = img.reshape(784, 1) / 255.0
        # add bias
        img = np.vstack((img, [1]))
        Images.append(img)
    data = tf.convert_to_tensor(np.array(Images).reshape(number_samples, 785),
dtype=tf.float32)
    return data

#sigmoid function as determined in the report and derived in lecture.
def sigmoid(X, theta,k,index):
    denom = 0
    for i in range(len(k)):
        denom += math.exp(np.inner(X,theta[i]))

    numer = (math.exp(np.inner(X,theta[index])))/denom
    return numer

#loss function as determined in the report and derived in lecture.
def loss(X,theta,y):
    cost = 0
    for i in range(len(X)):
        for j in range(len(y[i])):
            cost += y[i][j] * math.log(sigmoid(X[i],theta,y[i],j))
    return (cost*-1)

#gradient for multi class logisitic regression. Builds the whole gradient matrix.
def gradient(X,y, theta):
    grad = np.zeros(np.shape(X[0]))
    new_theta = np.zeros(np.shape(theta))
    for j in range(len(y[0])):
        for i in range(len(X)):
            grad+= (y[i][j] - sigmoid(X[i],theta,y[i],j)) * X[i]
        new_theta[j] =grad*-1
    return new_theta
```

```
#Test the model learned parameters by making predictions with the sigmoid
function
#chooses the maximum value of the individual class outputs of the sigmoid
function
def test_model(X,y,theta):
    y_guess = [0,0,0,0,0]
    for i in range(len(X)):
        correct = np.where(y[i]==1)[0][0]
        guess = []
        for j in range(len(y[i])):
            g= (sigmoid(X[i],theta,y[i],j), j)
            guess.append(g)
        if max(guess)[1] != correct:
            y_guess[correct] +=1
    return y_guess


# load training & testing data
train_data_path = 'train_data' # Make sure folders and your python script are in
the same directory. Otherwise, specify the full path name for each folder.
test_data_path = 'test_data' # Make sure folders and your python script are in
the same directory. Otherwise, specify the full path name for each folder.
train_data_root = pathlib.Path(train_data_path)
test_data_root = pathlib.Path(test_data_path)

# list all training images paths, sort them to make the data and the label
aligned
all_training_image_paths = list(train_data_root.glob('*'))
all_training_image_paths = sorted([str(path) for path in
all_training_image_paths])

# list all testing images paths, sort them to make the data and the label aligned
all_testing_image_paths = list(test_data_root.glob('*'))
all_testing_image_paths = sorted([str(path) for path in all_testing_image_paths])

# load labels
training_labels = np.loadtxt('labels/train_label.txt', dtype = int) # Make sure
folders and your python script are in the same directory. Otherwise, specify the
full path name for each folder.
# convert 1-5 to 0-4 and build one hot vectors
training_labels = (tf.reshape(tf.one_hot(training_labels - 1 , 5,
dtype=tf.float32), (-1, 5))).numpy()
testing_labels = np.loadtxt('labels/test_label.txt', dtype = int) # Make sure
folders and your python script are in the same directory. Otherwise, specify the
full path name for each folder.
```

```
testing_labels = (tf.reshape(tf.one_hot(testing_labels - 1 , 5,
dtype=tf.float32), (-1, 5))).numpy()

#--Block-- (Pickle block so we don't need to convert data all the time and can
just load pickle)
#comment these lines out if uncommenting above block to load new data
train_set = pickle.load( open( "training_set.txt", "rb" ) ).numpy()
test_set = pickle.load( open( "testing_set.txt", "rb" ) ).numpy()
#--Block--

#--Block-- (Set of our hyper parameters of traning the model)
theta = np.zeros((len(training_labels[0]),len(train_set[0])))
learning_rate=0.0001
iterations = 40
count = 0
y_count = [0,0,0,0,0]
for i in range(len(testing_labels)):
    k_to_1=np.where(testing_labels[i]==1)[0][0]
    y_count[k_to_1] +=1
#--Block--

#--Block--(Block that trains the model using gradient descent and gathers data)
loss_per_iter=[]
avg_acc = []
label_acc = []
x_axis = []
for i in range(len(testing_labels[0])):
    foo = []
    label_acc.append(foo)

while count<iterations:
    #collects data every 10 iterations
    if count%4 == 0:
        loss_per_iter.append( loss(train_set,theta,training_labels))
        iter_acc = test_model(test_set,testing_labels,theta)
        for i in range(len(iter_acc)):
            label_acc[i].append(iter_acc[i]/y_count[i])
        avg_acc.append(1-np.mean(np.divide(iter_acc,y_count)))
        print(count)
        x_axis.append(count)

    #updates theta matrix every iteration
    theta = theta - learning_rate*gradient(train_set,training_labels,theta)
    count +=1
#--Block--
```

```
#--Block-- (Use to dump the theta matrix or W matrix of learned weights
filehandler = open("multiclass_parameters.txt","wb")
pickle.dump(theta, filehandler)
filehandler.close()
#--Block--
```

Figure 1: Serial Implementation of the Model

Above we show the Python code for the serial implementation of the machine learning model to solve the problem described in the "Problem Statement" section. The code has comments that describe the process in how it builds the model and outputs the weights matrix. Outside of libraries such as Pickle, MatplotLib, and Pandas that help with parsing or plotting this implementation heavily uses NumPy for computation. NumPy aids this implementation by handling more complex mathematical operations necessary for the learning process.

In particular NumPy computes inner products of vectors and the transpose of matrices or vectors that we pass in. Performing these operations in standard Python would be the same time complexity as Python, however NumPy has many other operational optimizations that makes their real time computation incredibly faster at the cost of some memory. Over the course of thousands of data points and hundreds of iterations the speedup afforded by NumPy is incredibly noticeable.

Discussing the implementation further we take in our training data as input, perform batch gradient descent using the whole dataset and update the learned weights Θ every iteration. For performance metrics we test the prediction accuracy for each class of the output data, in addition we compute the average prediction accuracy for all classes. Furthermore we take the standard performance metric of training loss and training accuracy in order to determine if the model is performing well.

The implementation of the gradients and loss function are directly taken from the equations shown in the problem statement section. The exact computation is performed as if it was being computed by hand. The same holds true for the prediction performance metric.

4: Points of Parallelization

Without even trying to use other powerful packages there are design points in the standard machine learning procedure that can be parallelized. The largest point of parallelization can be seen within both the gradients and loss function. The loss function is the sum of a computation that uses individual entries in an array. Rather than doing them one by one, we can instead compute the loss for all indices in parallel and add the results at the end. A similar method can additionally be applied to the method in which we compute the gradient. Again the gradient is simply the sum of computations that happen on individual entries of the input data array. We can simply compute them all at once and add them up in order to achieve our gradient for a class k.

The speedup afforded by implementing these in parallel is not to be understated. As the learning method takes the whole gradient, which is thousands of entries in length, the ability to turn this into near constant time, assuming enough memory or CUDA cores, would drastically speed up the learning process. This is because the gradients are computed at every iteration, unlike the loss function. For performance testing purposes we compute the loss of the model every few iterations, but this is not needed for developing the model. While it can help increase the speed, it is not a significant upgrade as compared to parallelizing the gradient computation.

5: Parallel Implementation

```
import matplotlib.pyplot as plt
import numpy as np
import pickle
import random as r
import tensorflow as tf
from tensorflow import keras
from keras import layers
from google.colab import drive
import math
drive.mount('/content/gdrive')

training_labels = np.loadtxt('/content/gdrive/My
Drive/Colab_Notebooks/pc_final/labels/train_label.txt', dtype = int)
training_labels = np.array( (tf.reshape(tf.one_hot(training_labels - 1 , 5,
dtype=tf.float32), (-1, 5))))
testing_labels = np.loadtxt('/content/gdrive/My
Drive/Colab_Notebooks/pc_final/labels/test_label.txt', dtype = int)
testing_labels = np.array((tf.reshape(tf.one_hot(testing_labels - 1 , 5,
dtype=tf.float32), (-1, 5))))

train_set = np.array(pickle.load( open( "/content/gdrive/My
Drive/Colab_Notebooks/pc_final/training_set.txt", "rb" ) ))
test_set = np.array(pickle.load( open( "/content/gdrive/My
Drive/Colab_Notebooks/pc_final/testing_set.txt", "rb" ) ))

model = keras.Sequential(
    [
     layers.Dense(5, activation = 'softmax')
    ])

epochs = 20
batch_size = 80

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3, beta_1
= .75, beta_2 = .95),
```

```
                loss= keras.losses.MeanSquaredError(), metrics=['accuracy'])
completed_model = model.fit(train_set, training_labels,
                            validation_data=(test_set,
testing_labels),batch_size=batch_size, epochs=epochs)
```

Figure 2: Parallel Implementation of the Model Using Tensorflow

Above we show the code for the parallel implementation of the model. Here we make heavy use of Tensorflow functions in order to build the model. Tensorflow is a machine learning package from Google designed for Python. Tensorflow was chosen because it synchronizes well with NVIDIA GPUs for parallelization. NVIDIA has gone out of their way to have built in optimizations for their GPUs for usage in Tensorflow, and the NVIDIA Tesla unit that this code was run on is no exception. To run this system on a local machine you would need to install Docker and install Tensorflow on top of that. To construct the same structure of model we use the Sequential method for model construction in Tensorflow, and then add 5 nodes to represent our 5 outputs. This means that there is one matrix of weights for the model just like in the serial implementation. We use Tensorflow's model.compile() and model.fit() function to properly train the model by applying our loss function, learning weight, and other hyper parameters.

For both the serial and parallel implementations of the model the performance metrics were recorded in a similar fashion, and furthermore their overall computation expense is minimal compared to model training and testing.

6: Results

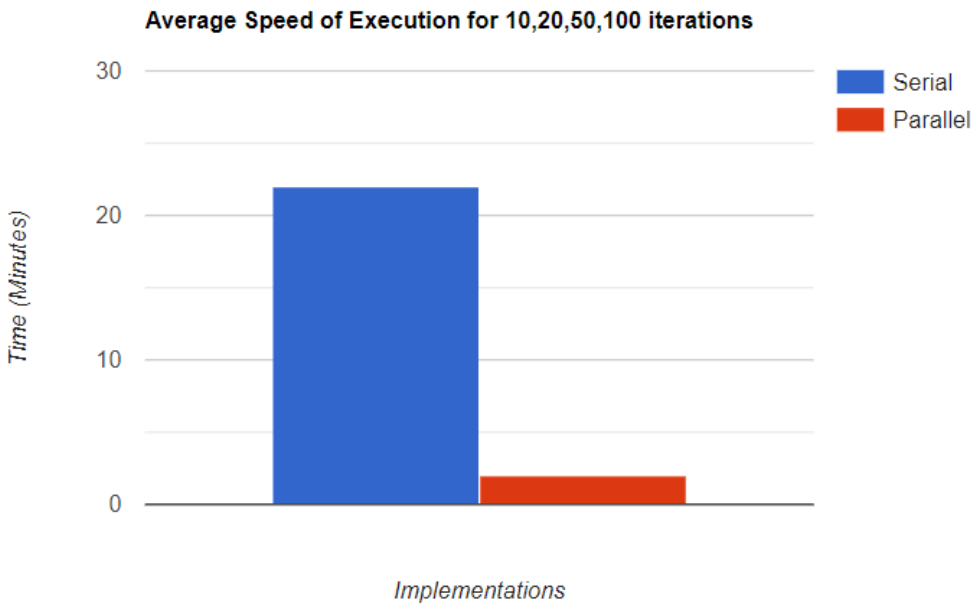

Figure 3: Average Speed Comparison for Various Iteration Values

Above we show the speed of execution between the serial and parallel implementations. As this is our main metric we are analyzing in this paper it is good to see that the parallel

implementation was significantly faster. For the average time between all these different sets of iterations to be so small as compared to the serial version is encouraging. The time scale here is in minutes rather than seconds and it shows just how large of a problem we were dealing with, even in this simple machine learning problem. The average serial time shot up drastically as we headed above 20 iterations as the scale of the computation jumped up drastically in tandem. Whereas in the parallel implementation in Tensorflow, a majority of the time was spent loading the data onto the GPU for computation. The computation time even for 100 iterations was only about 30 seconds. This is in line with the barrier of entry to parallel problems.

Most problems that can be solved in a serial way are sufficiently quick enough to be left in a serial implementation. A parallel implementation would theoretically be faster, but in reality because the GPU memory is orders of magnitude slower than CPU memory the true execution time is much larger. It's only in problems such as machine learning ones where the extra time spent loading onto the GPU using its slow memory actually brings significant gains when the bulk of execution time is spent in an easily parallelizable section of code. It's clear from this case study that implementing more complex machine learning problems in parallel is necessary to achieve reasonable results in a reasonable amount of time.

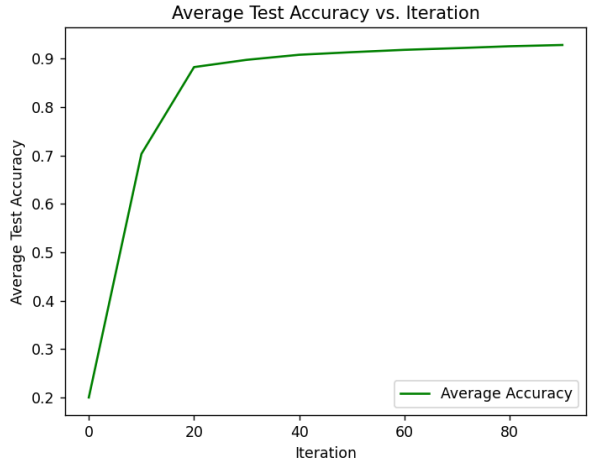

Figure 4: Serial Implementation Accuracy

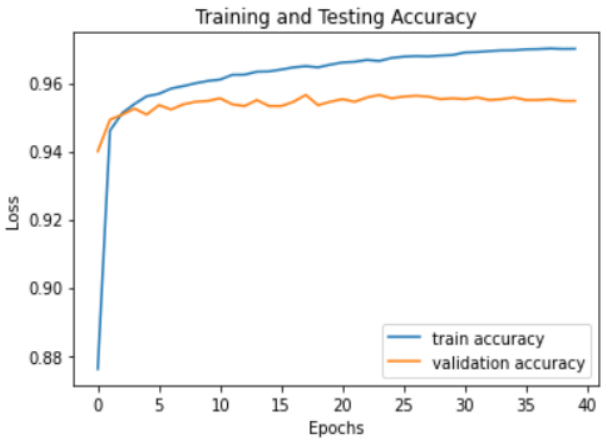

Figure 5: Parallel Implementation Accuracy

The above figures showcase our other performance metric for this paper. The accuracy comparison between serial and parallel. Machine learning problems are not truly solvable, they instead intend to sufficiently minimize a loss function and achieve a high prediction accuracy. As such a parallel machine learning model should perform similarly to a serial model if it is to be further developed, otherwise the speed up from the parallel model is worthless. As shown by the figures above the parallel model is able to achieve similar results to a serial model at around the >95% margin. With further tuning of hyper parameters the parallel model can easily exceed the margin set by the serial model. This can be performed very rapidly as testing the model takes very little time since the model is built in parallel with Tensorflow.

In addition Tensorflow has options for training other that model.fit() methods. We can define a proper gradient and tune the model to the problem at large rather than the subset of data that we

provided. Tensorflow also allows us to build a neural network quite easily that can further increase the complexity of our model to more accurately represent the problem.

7: Conclusion

From the data gathered it is clear to see that parallel implementation of machine learning models are clearly superior. As machine learning itself does not aim to explicitly solve problems, rather build an iterative solution over time, minimizing the time between iterations is paramount to generating a working product in a reasonable fashion. Tensorflow, being a free software package that works extremely well with NVIDIA GPUs, fits the criterion as a great tool for parallel and GPU engineers to build machine learning models with. Users will be limited to Python as their language of choice, but because Tensorflow is so powerful, the sluggishness of Python as compared to C++ is drastically outweighed by the computation power afforded to developers by Tensorflow. The code to run and train the models are given above in the figures and it is encouraged to see for one's self just how fast machine learning can be. With the question of speed effectively solved for machine learning as an industry, the question of preparing enough RAM space to hold models and data for computation becomes the one that looms over machine learning engineers. Fortunately Tensorflow can also help with that.