Acceleration of Python Artificial Neural Network in a High Performance Computing Cluster Environment

Christopher Rosser, Ebrahim Alareqi, Anthony Wright, and Khalid Abed
Dept. of Electrical & Computing Engineering and Computer Science
Jackson State University
Jackson, MS. 39217
{christopher.rosser; ebrahim.a.al_areqi; anthony.wright}@students.jsums.edu, khalid.abed@jsums.edu

Abstract- Object recognition is one of the important and growing areas of machine learning and deep learning applications. Research areas of Machine Learning and Deep Learning provide numerous techniques for analyzing data. These techniques can enable software systems to make predictions that would enhance software performance by allowing software systems to learn. Techniques such as linear regression and k-Nearest Neighbor are great for making predictions on small datasets but can become computationally expensive for larger problems. Neural Networks provide more sophisticated means for making predictions but can also become computationally expensive for larger problems such as image recognition models. Hence, the need for greater and smarter computational power to accelerate the intense computation of such models. In this research paper, we attempt to motivate the adoption of Intel Xeon-based computing platforms with Intel's distribution of Python along with the offloading of deep learning intense computations of the code to the host co-processors by demonstrating the speedup of an optimized Python artificial neural network on a high-performance computing cluster. Given a labeled dataset of ship images, and the implementation of deep learning models with different parameters, results show a significant increase in the binary classification performance of the application.

Keywords— Neural Networks, Keras, Tensorflow, Xeon Phi, Image Classification, MPI

I. INTRODUCTION

Scientific, engineering, and analytics problems are larger than ever due to the explosion of data created by new devices and technologies, as well as old devices and technologies such as basic sensors that now support internet connectivity. Solving complex science and engineering problems programmatically generally requires programming and domain expertise. However, industry, government, and open-source users are hard at work developing ways for application and software users of every level to get involved in science and engineering problem-solving. Keras [1] has emerged as a go-to deep learning API in which users can dive head first into creating and testing basic and custom neural networks. Keras boasts extensibility, ease of debugging due to Python functionality, and, best of all, user-friendliness [1]. Keras runs on top of Tensorflow [2], which is popularly known as Google’s machine learning library. Of particular interest is the stackable-layer and convolution neural network features. Figure 1 shows a Convolutional Neural Network (CNN) with different parameters which can be implemented in only a few lines of Python code using the Keras API.

With the readily available and free deep learning and machine learning libraries and APIs, there is a growing number of websites that provide free datasets – of which Kaggle is popular. We have selected an already-optimized Python-based image classification program to modify and use as a demonstration of the ease and straightforwardness with which a neural network can be created, compiled, and used in classification.

The code used for this research project is a solution that addresses the human error in having to identify shapes in satellite imagery. The testing data consists of hundreds of 80x80 pixel image chips that are labeled as ships (1) or not ships (0). The test images are ~2700x~2700 pixel satellite images from which the training data was developed. The code was run in Jupyter Notebook to verify that it is in fact optimal.

Interactively, we were able to see that two cells/blocks of code performed the slowest: the block of code in which forward and backward propagation occurs, and the cells/block of code in which various-sized filters are applied to the test image. These two cells/blocks of code make this image classification script a great candidate for speedup on the Intel Xeon Processor.
The rest of this article is organized as follows: Section II is a description of the problem and the dataset. Section III describes the environment where the code was tested. Section IV describes the methodology used to approach the dataset and the computational challenge. Section V demonstrates the implementation and results using both the single CPU (2 Cores) and the Cluster (61 Cores). Section VI provides a conclusion and identifies potential future work.

II. PROBLEM AND DATASET DESCRIPTION

The dataset that is extracted from Planet [4] satellite imagery and collected over the San Francisco Bay and San Pedro Bay areas of California consists of images that are from. It includes 2800 of 80x80 RGB images labeled with either a "ship" or "no-ship" classification. Image chips were derived from PlanetScope [4] full-frame visual scene products, which are orthorectified to a 3-meter pixel size. Each individual image filename follows a specific format: {label} ___ {scene id} ___ {longitude} _ {latitude}.png as the following:

a. label: Which is either 1 or 0, representing the "ship" class and "no-ship" class, respectively.

b. scene id: The unique identifier of the PlanetScope visual scene the image chip was extracted from. The scene id can be used with the Planet API to discover and download the entire scene.

c. longitude_latitude: The longitude and latitude coordinates of the image center point, with values separated by a single underscore. The dataset is distributed as a JSON formatted text file shipsnet.json [6]. The loaded object contains data, label, scene_ids, and location lists.

The pixel value data for each 80x80 RGB image is stored as a list of 19200 integers within the data list. The first 6400 entries contain the red channel values, the next 6400 the green, and the final 6400 the blue. The image is stored in row-major order, so that the first 80 entries of the array are the red channel values of the first row of the image. The list values at index i in labels, scene_ids, and locations each correspond to the i-th image in the data list.

Seven hundred images are labeled under "ship" class. Images in this class are near-centered on the body of a single ship. Ships of different sizes, orientations, and atmospheric collection conditions are included. Sample images from this class are shown in figure 2. In the second class labeled "no-ship" has two thousand one hundred images. A third of these are a random sampling of different landcover features - water, vegetation, bare earth, buildings, etc. that do not include any portion of a ship. Example images from this class are shown in Figure 3.

The code used for this research project is a solution to the described problem that addresses the human error in having to identify shapes in the satellite imagery mentioned earlier. The testing data has hundreds of 80x80 pixel image chips that are labeled as ships (1) or not ships (0). The test images are ~2700x~2700 pixel satellite images from which the training data was developed. The code was run in Jupyter Notebook to verify how optimal the code is. Interactively, we were able to see that two cells/blocks of code performed the slowest: the block of code in which forward and backward propagation occurs, and the cells/block of code in which various-sized filters are applied to the test image. These two cells/blocks of code make this image classification script a great candidate for speedup on the Intel Xeon Processor.

III. ENVIRONMENT

The deep learning neural network code was tested in two environments: Intel Celeron CPU, 4.00 GB RAM, 2.16 GHz (2 Cores) and a cluster that consists of six compute nodes. The system used is a Cray CS400-AC optimized for HPHC. It consists of eight components: a login node, a head node and six compute nodes. The login and head nodes are for management and operational networking purposes while application execution is performed on the compute nodes. The head node, named tiger, has an Intel Xeon E5-2650 v3 CPU, a Connect-IB (InfiniBand) Single Port QSFP, FDR (Fourteen Data Rate) adapter card and a ConnectX-3 EN10GbE Dual-Port SFP+ PCIe3.0 x8 8GT/s Network Interface Card. It also has an Intel RS2BL08D 8-portSAS RAID controller. Each compute node, named node 001-006, has an Intel Xeon X5-2650 v3 CPU, a Connect-IB (InfiniBand) Single Port QSFP, FDR (Fourteen Data Rate) adapter card and a ConnectX-3 EN10GbE Dual-Port SFP+ PCIe3.0 x8 8GT/s Network Interface Card. Each node consists of an Intel Xeon E5-2698 2.3GHz and dual Intel Xeon Phi 7120 passively cooled 1.25 GHz coprocessors. The Xeon Phi 7120 has 61 in-order 64-bit cores able to execute two instructions per cycle with 16GB of GDDR memory.

The cores are fully functional computational units which allow the coprocessor to run its own Linux based operating system called uOS. Each node consists of an Intel Xeon E5-2698 2.3GHz and dual
Intel Xeon Phi 7120 passively cooled 1.25 GHz coprocessors. The Xeon Phi 7120 has 61 in-order 64-bit cores able to execute two instructions per cycle with 16GB of GDDR memory. The cores are fully functional computational units which allow the coprocessor to run its own Linux based operating system called uOS [13].

Each core has a 32KB L1 instruction cache, a 32KB L1 data cache, a 512KB L2 cache and a vector processing unit (VPU) that contains 32 512-bit registers capable of processing 16 single-precision or 8 double-precision floating-point arithmetic operations or 32-bit integers in parallel [6]. Each core is capable of multithreading and contains four hardware threads. Thus our computational environment is capable of running 488 threads. The cores communicate with each other and maintain L2 cache coherency through a bi-directional ring interconnect with distributed tag directories. The ring interconnect also allows the cores to access data and instructions from main memory via the memory controller [7]. Figure 4 illustrates the main components of the Xeon Phi’s architecture.

The Xeon Phi communicates with its host processor via a Peripheral Component Interconnect Express (PCIe) bus. Because the card does not have any input and output options, all data travels through the PCIe interface and thus the PCIe bus is a source of data transfer overhead. Also, the Xeon Phi coprocessors run at approximately one third the rate of the Xeon host processors [10].

The bi-directional ring interconnect, the large VPUs, the PCIe bus and the 512-bit SIMD capability make it necessary to determine how and when to utilize the Xeon Phi MIC architecture [9].

### IV. METHODOLOGY

The neural network constructed in this code was designed to search the test image using varying filter sizes beginning with 80x80 pixels, which is consistent with the 80x80x3 image chips in the training data [11]. This is achieved using the Keras convolutional layer Conv2D. The filter is then halved using the Keras Pooling layer MaxPool2D, and reapplied to the same test image. The filter is halved and re-applied until it is at its smallest size for the scope of this research, 5x5 [8]. A Keras Dropout filter is added every time the filter is halved to help reduce over-fitting [12]. The Keras core layers are used to ensure the neural network is densely-connected (Dense) and to reshape the input data (Flatten) in order to control and preserve the desired output shape. The Neural Network model in Keras was configured as shown in table 1:

![Figure 4: Xeon Phi Architecture](image)

**TABLE I**

<table>
<thead>
<tr>
<th>Network Model Summary</th>
<th>Output Shape</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv2d_1 (Conv2D)</td>
<td>(None, 40, 40, 32)</td>
<td>956</td>
</tr>
<tr>
<td>MaxPooling2_1 (MaxPooling2)</td>
<td>(None, 20, 20, 32)</td>
<td>0</td>
</tr>
<tr>
<td>Dropout_1 (Dropout)</td>
<td>(None, 40, 40, 32)</td>
<td>0</td>
</tr>
<tr>
<td>Conv2d_2 (Conv2D)</td>
<td>(None, 20, 20, 32)</td>
<td>9208</td>
</tr>
<tr>
<td>MaxPooling2_2 (MaxPooling2)</td>
<td>(None, 10, 10, 32)</td>
<td>0</td>
</tr>
<tr>
<td>Dropout_2 (Dropout)</td>
<td>(None, 10, 10, 32)</td>
<td>0</td>
</tr>
<tr>
<td>Conv2d_3 (Conv2D)</td>
<td>(None, 10, 10, 32)</td>
<td>102432</td>
</tr>
<tr>
<td>MaxPooling2_3 (MaxPooling2)</td>
<td>(None, 5, 5, 32)</td>
<td>0</td>
</tr>
<tr>
<td>Dropout_3 (Dropout)</td>
<td>(None, 5, 5, 32)</td>
<td>0</td>
</tr>
<tr>
<td>Flatten_1 (Flatten)</td>
<td>(None, 5)</td>
<td>0</td>
</tr>
<tr>
<td>Dense_1 (Dense)</td>
<td>(None, 512)</td>
<td>410112</td>
</tr>
<tr>
<td>Dropout_4 (Dropout)</td>
<td>(None, 512)</td>
<td>0</td>
</tr>
<tr>
<td>Dense_2 (Dense)</td>
<td>(None, 2)</td>
<td>1024</td>
</tr>
</tbody>
</table>

Total parameters: 552,962
Trainable parameters: 552,962
Non-trainable parameters: 0

The optimized code was run on two different environments. Table 2 shows the two computational environments.

**TABLE II**

<table>
<thead>
<tr>
<th>Compilers/Software</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 8.1 (64-bit) Keras-Tensorflow Anaconda Environment</td>
<td>Intel Celeron CPU, 4.00 GB RAM, 2.16 GHz (2 Cores)</td>
</tr>
<tr>
<td>Red Hat Linux (64-bit) Keras-Tensorflow Anaconda Environment</td>
<td>Intel Xeon Processor E5-2698, 40 M Cache, 2.3 GHz (16 Cores) + Intel Xeon Phi Co-Processor, 16 GB RAM, 1.238 GHz (61 Cores)</td>
</tr>
</tbody>
</table>
The Keras environment with a Tensorflow backend was created using Anaconda:
```
conda create --n keras python=3.6
```
in which `--n` declares the new environment named `keras` using Python version 3.6. The Python package `mpi4py` was added to this environment to support message passing parallelization of the code. The environment was activated using `source activate keras` before we ran the code. This environment is a placeholder for all packages and their dependencies that are imported in the code.

Once the Keras sequential model is created, it must be compiled using the Keras built-in `.compile` function, in which optimization parameters can be passed. The optimizer used, stochastic gradient descent (SGD), has default values of `lr=0.01`, `momentum=0`, `nesterov=False` (Nesterov momentum disabled) and was previously tested using its default values. The model performs better with `momentum=0.9` and `nesterov=True`.

Next, the Keras sequential model is fit to the input data, which has been reshaped in the previous sections of the code. `Batch_size`, `epochs`, and `shuffle` are the greatest factors in affecting the fit of the model. After previous case studies, in which `batch_size`, `epochs`, and `shuffle` values were modified, it was determined that if the code remain untouched, the one value that could directly affect the results of this model is the epochs number. The base epochs presented in this code by its author is 18. An `epochs` of less than 18 was tested but classified fewer ships, but epochs greater than 18 resulted in better classification, but also more misclassifications which is not necessarily a bad thing. The objects that the code classifies and misclassifies as ships have white boxes drawn around them on the test image. Through visual inspection the misclassified objects can be identified and discarded from the final count because the `keras.callbacks` feature allows the user to view the internal information of most functions, we set the `model.fit` equal to the variable `hist` so that we could recall the loss and accuracy of each object classified and misclassified as a ship, resulting in the following text output for objects classified as a ship:

```
Figure 5: Test Image with Identified Ships.
```

V. RESULTS

The general Python compiler, `python`, was used to compile the code once the Keras-Tensorflow environment was activated. For parallelization, the general Python compiler and the `mpirun` compiler was used, in which a number of processors are passed using `–n 2`, where 2 is the number of processors to use, as well as the built-in Python profiler `cProfile`.

We observed the behaviors of the code when it was executed on two processors at once using MPI. The two instances of the code being run in parallel using two processors resulted in a ~5x speedup in comparison to running one instance of the code in the Intel Celeron environment.

We ordered the profiler results by total time of each function, which confirmed our guess that the most time-consuming portions of the code were in fact calls to the built-in Tensorflow functions and the traversal over the test image.

```
<table>
<thead>
<tr>
<th>Ncalls</th>
<th>Tottime (s)</th>
<th>Percall</th>
<th>Cumtime</th>
<th>Percall</th>
</tr>
</thead>
<tbody>
<tr>
<td>20976</td>
<td>566.419</td>
<td>0.021</td>
<td>567.189</td>
<td>0.027</td>
</tr>
<tr>
<td>22562</td>
<td>313.970</td>
<td>0.014</td>
<td>313.970</td>
<td>0.014</td>
</tr>
</tbody>
</table>
```

`TABLE IV` TIME CONSUMING FUNCTIONS (cProfile results)

The objects that the code classifies and misclassifies as ships have white boxes drawn around them on the test image. Through visual inspection the misclassified objects can be identified and discarded from the final count because the `keras.callbacks` feature allows the user to view the internal information of most functions, we set the `model.fit` equal to the variable `hist` so that we could recall the loss and accuracy of each object classified and misclassified as a ship, resulting in the following text output for objects classified as a ship:

```
Figure 5: Test Image with Identified Ships.
```
VI. Conclusion and Future Work

On average, the ship classifying code experienced a speedup of 5x. Therefore, a heterogeneous computing environment should definitely be considered for neural network computations for its computation power, if for no other reason. We expect this speedup to increase as Tensorflow and Keras work towards optimization for use on the Intel Xeon processor family.

Although message passing (MPI) and offloading schemes are ideal for mathematical computations and specifically better-suited for C-programming code, Intel’s release of an Intel-optimized Tensorflow [12] is a clear indication that its line of processors will eventually be updated to support parallelism and computation of big data problems in the coming months and years.

As of the date of this paper, there is no known Intel Xeon Phi or MPI enhanced Keras support available for the Xeon Phi, which pushes the research presented in a positive direction for future work.

References