TensorFlow Julia API

Patrick Lowe

Introduction

Artificial intelligence (AI) has been around for a while and in recent years machine learning has become increasingly popular. Hardware has seen dramatic performance increase and cost reduction according to Moore's law. This has enabled practical implementations of theoretical AI systems. Many commercial companies have begun to invest heavily in machine learning projects and research. Google's Brain Team within its Machine Intelligence research organization has been developing TensorFlow, a machine learning library for such projects. The team has released a Python and C++ API. In this project I have developed a Julia API and compare the performance in training machine learning systems on the MNIST (Mixed National Institute of Standards and Technology) database for handwriting recognition. I examine the performance of Python, Python/C++, and Julia/C++ in both serial and parallel implementations.

TensorFlow

TensorFlow was released as open source software in the hopes that it would become widely adopted and a standardized framework. Google set several goals for TensorFlow that make it unique. They wanted the framework to be general and scalable. TensorFlow aims to be just as useful in research as it is for a released product. You should be able to used models generated and trained in a lab and plug it into an actual application using TensorFlow without needing to rewrite the model. Google also hopes that the framework will be used for general AI research not just neural nets as DistBelief (TensorFlow's predecessor) was.

In terms of hardware scalability TensorFlow is designed to work on everything from a handheld device to a GPU accelerated cluster. This ease of parallelization seems to align with the goals of Julia. Both systems claim to be designed for parallelization and ease of use while also maintaining high performance speed.

TensorFlow uses a Data Flow Graph (as seen in Figure 1) to represent mathematical computation. This directed graph uses nodes to represent operations or computations such as matrix multiply or convolution. Edges carry data as inputs or outputs between nodes. These data sets are represented as tensors or multi dimensional arrays. Nodes are assigned to computational devices and are allowed to execute asynchronously and in parallel once all the input tensors become available on the inflow edges. This representation is structured in a way that makes distribution and parallelization easy to see and implement.

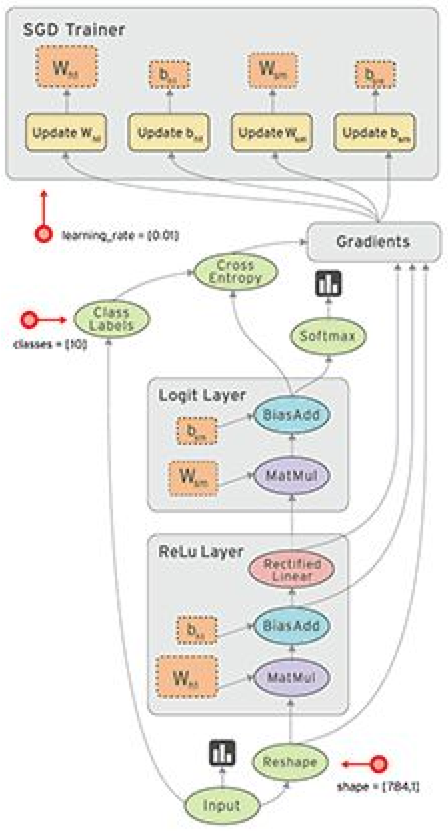


Figure 1: Data Flow Graph

While there are many different options for how to implement parallelism we will examine a method which splits up the training data set and allows multiple copies of the same model to be trained in parallel. There are two models two computational models to consider, asynchronous and synchronous.

The asynchronous system has every model running as fast as possible. It sends and receives updates as soon as it is done or ready. While this may maximise the speed and utilization of the hardware it can lead to imperfect training. It is easy to see a situation in which a model will be fast and update before others have a chance to finish their computation. Then the models will lose their update synchrony and may be working with outdated parameters.

Alternatively in synchronous parallelization all processes happen in lockstep which means that updating parameters across the parallel models is straightforward. After a round of training is finished all the parallel instances send their results to a centralized process which combines them. THis process then redistributes new model parameters and the training tests are rerun. One problem is that you can only operate as fast as your slowest model. If all of the hardware is identical and work is evenly split then this may not be an issue. However for systems that aren’t homogeneous or load balanced this will lead to inefficiencies.

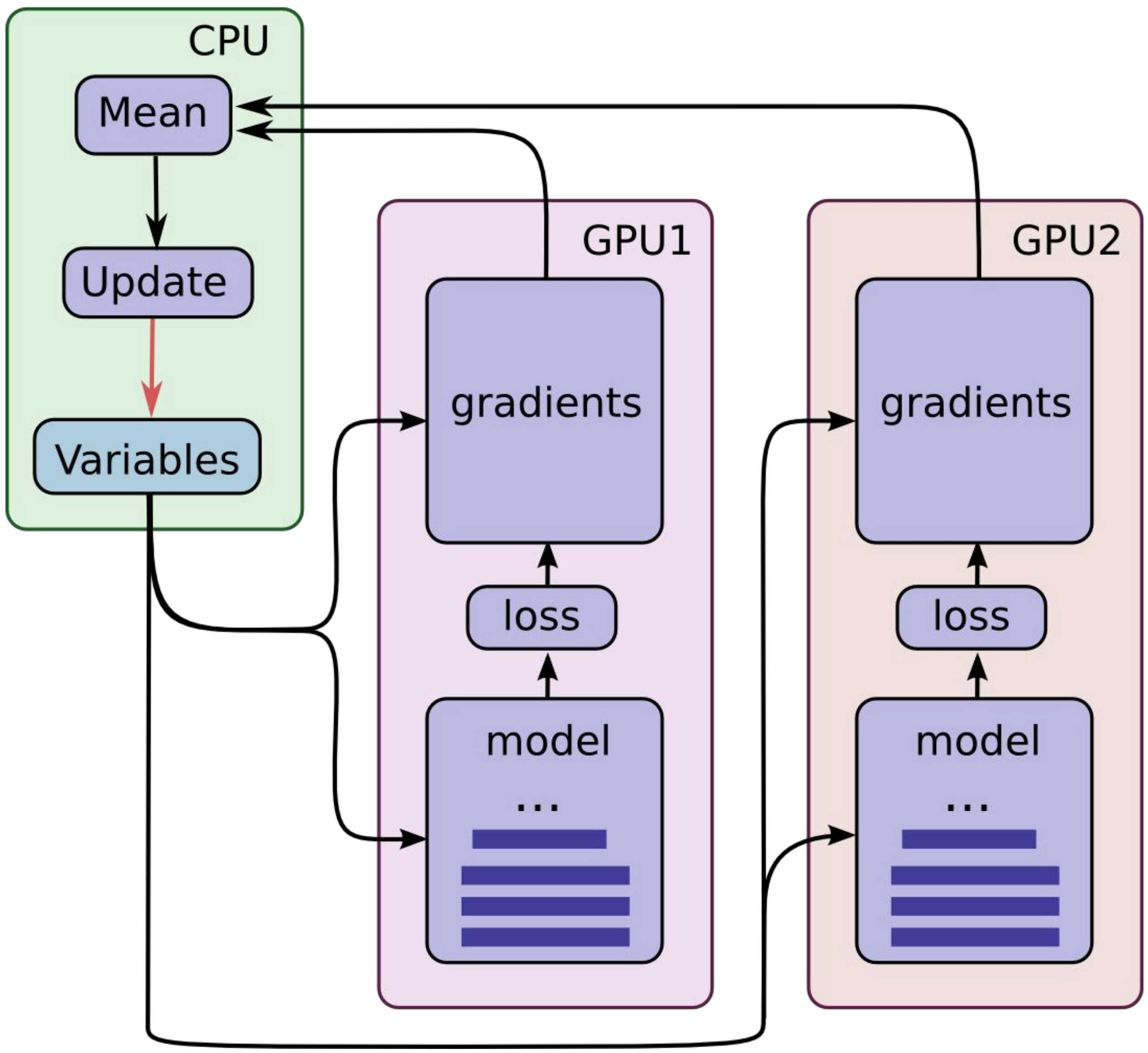


Figure 2: Parallel Training With GPUs

An interesting observation is that these models for training parallelization follow the * operator framework. In this framework a task is divided into several parallel problems that are computed in a distributed fashion. The results from each partial problem are then sent to a central process which performs some computation on the entire results data set. This central process then sends its results back to the parallel processes for a final computation. In our case the distributed processes are models with different training data. They send their training results to a central process which updates the model parameters before sending them back for each parallel process to train again.

Another framework that is quite suited to this method of parallelization is for ensemble training. In this AI model different methods of classification are developed independently and the final classifier is a weighted combination of the outputs from each classifier. In this case the different models are independent and don't depend on the results of each other's training. As such each model can be trained in parallel and asynchronously without the risk of imperfect training from stale parameters.

MNIST

The MNIST problem is a standard computer vision and AI problem. In this case the data set is hand written digits. The program must recognize and determine what digit is written in each image.

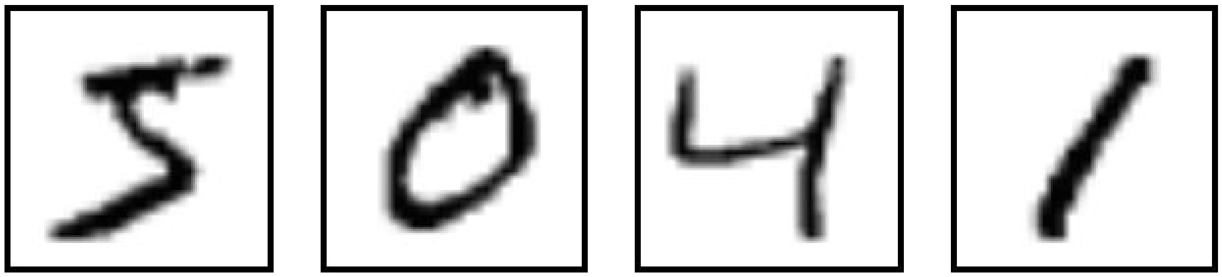
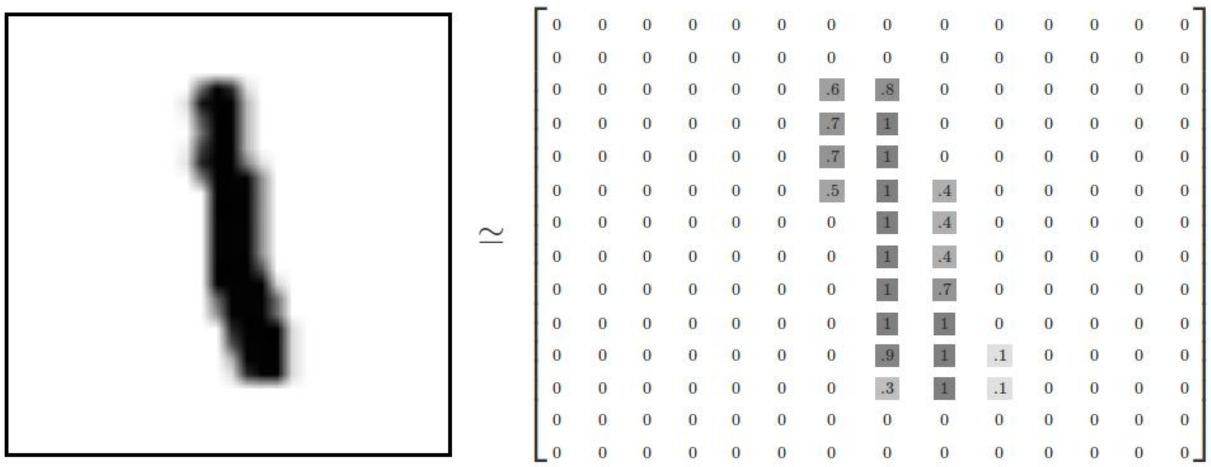


Figure 3: Example MNIST data

The dataset is composed of images and labels which contain the correct classification of each image (5, 0, 4, 1 for the data in Figure 3). In our tests we will use 60,000 data points for training and 10,000 data points for testing.



$$\simeq \begin{bmatrix} 0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&.6&.8&0&0&0&0&0&0 \\ 0&0&0&0&0&0&.7&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&.7&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&.5&1&.4&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&.4&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&.4&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&.7&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&1&0&0&0&0&0 \\ 0&0&0&0&0&0&0&.9&1&.1&0&0&0&0 \\ 0&0&0&0&0&0&0&.3&1&.1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0 \end{bmatrix}$$

Figure 4: Images are 28 x 28 pixels represented as a 2D numerical array

During the training the program is given both the images and their labels. It then does its best to create a model or classifier that can assign the correct label to each image. However if this is the only step we do then we run the risk of overfitting the data and becoming too specialized.

By keeping a separate data set for testing that the model has never seen during training we can evaluate how general our solution. Classifiers that work well on the training data but are inaccurate on the testing data are not valuable. Ultimately we want to use these classifiers on data that we have not encountered yet which the testing data mimics.

APIs

Google released TensorFlow with two APIs. The first one is a Python API which while full featured is also quite slow. Currently the methods represented by the Python API include:

- Building Graphs
- Constants, Sequences, and Random Values
- Variables
- Tensor Transformations
- Math
- Control Flow
- Images
- Sparse Tensors
- Inputs and Readers
- Data IO
- Neural Network
- Running Graphs
- Training

The other API is in C++. While this one isn't as feature rich as the Python API it is considerably faster. Currently the C++ API doesn't support the construction and setup of Data Flow Graphs which must be done using the Python API. However you can run the Data Flow Graph from C++ which is considerably faster. Currently the methods represented by the C++ API include:

- Env
- Session
- Status
- Tensor
- Thread

For this project I chose to implement the methods needed to run MNIST training in a Julia API. Because of the inherent speed already in the C++ API, I decided to use the C++ code and implement methods for those that were needed but only available in Python. Currently the methods represented by the Julia API include:

- Building Graphs
- Constants, Sequences, and Random Values
- Variables
- Tensor Transformations
- Math
- Control Flow

- Neural Network

Testing and Evaluation

For this research I evaluated 3 programs. The first program was implemented entirely using the Python API. The second program was implemented using a combination of the Python and C++ API. C++ was used whenever possible and Python was used for building the Data Flow Graphs and setting up sessions which C++ doesn't yet support. The third and final program used a combination of the Julia API I wrote and the C++ API. The C++ code in the second and third program are identical. The Julia code replaces the Python code. I ran MNIST training until I was able to get $\geq 95\%$ accuracy on the testing data (large convolutional neural nets can obtain $\geq 99\%$ accuracy). I compared the amount of time it took to train the model to that accuracy. Each program was run on 1, 2, 3, and 4 cores.

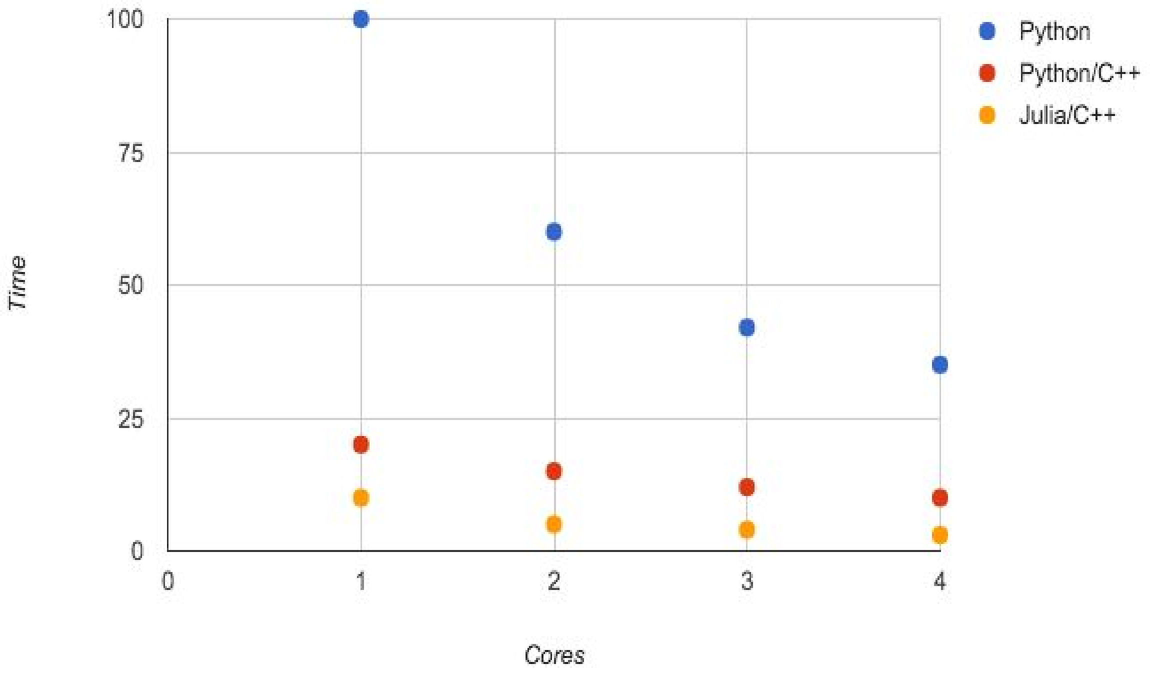


Figure 5: Results

The results of these tests support Julia's assertion that it is a significantly faster alternative to Python. Program 1, which is purely python, is much slower than either of the other programs which use C++ for computation. While it benefits from parallelization, program 1 plateaus as it approaches an asymptote limited by the overhead of spawning and combining parallel instances. Program 2 and program 3 which use C++ for computation see a significant speedup relative to program 1 even when run serially. Both of these programs also benefit from parallelization. However Program 2 starts to plateau because of the Python overhead involved with increased parallelization similar to program 1. Program 3 uses Julia to spawn and combining parallel instances. It doesn't hit such an asymptote in our study. This is likely due to Julia's faster speed and greater parallelization relative to Python.

Conclusion

In this project we were able to see the flexibility and scalability of TensorFlow synergize with that of Julia. While there were no groundbreaking or shocking discoveries from our testing and evaluation, the data continues to support Julia as a fast alternative to Python. In the future, it would be interesting to test at higher levels of parallelization to find the limit of speed up for the Julia API. It would also be useful to benchmark TensorFlow against Mocha, a native Julia deep learning framework. From my preliminary investigation in to Mocha it seems to support a similar feature set to TensorFlow. Both can utilize a C++ backend and NVidia GPUs for parallel computation. A performance comparison would help benchmark TensorFlow and my Julia API vs another Julia/C++ machine learning framework.