Distributed Deep Learning Framework in Python

Clay McLeod

University of Mississippi

December 3, 2015

Sources

- Presentation available at `http://bit.ly/cc-final-slides`.
- Final paper can be found at `http://claymcleod.github.io/papers/distributed-dnn/paper.pdf`

Problem Statement

Create a distributed architecture for commodity machines (similar to GFS) that evaluates different artificial neural network topologies for different datasets.

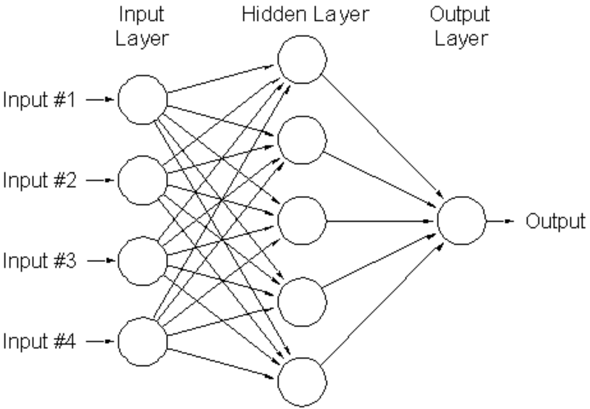
Background

- Big data is getting bigger, so how do we build statistical models to analyze this data?

Background

- Big data is getting bigger, so how do we build statistical models to analyze this data?
- One approach is a machine learning approach called an "Artificial Neural Network".

Background



Input Layer
Hidden Layer
Output Layer
Input #1
Input #2
Input #3
Input #4
Output

Background

- Big data is getting bigger, so how do we build statistical models to analyze this data?
- One approach is a machine learning approach called an "Artificial Neural Network".
  - Based off of how a human brain works.
  - Heavily researched, several good algorithms exists to build robust statistical models using ANNs.
  - Very good at capturing complex relationships embedded in the data.
  - Universal Approximator: A feedforward ANN with a single hidden layer and infinite number of hidden layer nodes can approximate any discrete or continuous mathematical function.
  - Specifically, I will be discussing "Deep Neural Networks" ($>$ 10 layers) because they are excellent at modelling large datasets.

Motivation

Why build a distributed deep neural network evaluator for commodity hardware?

Motivation

1. Layer design for Artificial Neural Networks is more of an art than a science.
   - Some papers providing rules of thumb have been published, but almost no theories exist on how to design layers for *any* dataset.
   - **Goal:** What can we learn about the design of ANN layers?

Motivation

1. Layer design for Artificial Neural Networks is more of an art than a science.
   - Some papers providing rules of thumb have been published, but almost no theories exist on how to design layers for *any* dataset.
   - **Goal:** What can we learn about the design of ANN layers?
2. Training neural networks is computationally expensive.
   - This is somewhat alleviated by using a GPU when available.
   - **Goal:** Parallelize different ANN configurations to speed up overall computing time and utilize all possible hardware accelerations.

Motivation

1. Layer design for Artificial Neural Networks is more of an art than a science.
   - Some papers providing rules of thumb have been published, but almost no theories exist on how to design layers for *any* dataset.
   - **Goal:** What can we learn about the design of ANN layers?
2. Training neural networks is computationally expensive.
   - This is somewhat alleviated by using a GPU when available.
   - **Goal:** Parallelize different ANN configurations to speed up overall computing time and utilize all possible hardware accelerations.
3. The benefit of getting this right is staggeringly large.
   - Recent searching for the so called 'Master Algorithm' has been focused around DNNs.
   - **Goal:** Can we figure out how to design DNNs that perform well for *any* dataset?

Why Python?

1. **Simplicity:** The simplicity of the Python language allows for rapid prototyping with less time spent fixing errors and more time doing actual science.

Why Python?

1. **Simplicity:** The simplicity of the Python language allows for rapid prototyping with less time spent fixing errors and more time doing actual science.
2. **Strong, Open Source Community:** Developers are able to iterate and innovate at an extremely rapid pace — this results in many Python libraries implementing bleeding edge techniques while using robust coding practices resulting from community code reviews.

Why Python?

1. **Simplicity:** The simplicity of the Python language allows for rapid prototyping with less time spent fixing errors and more time doing actual science.
2. **Strong, Open Source Community:** Developers are able to iterate and innovate at an extremely rapid pace — this results in many Python libraries implementing bleeding edge techniques while using robust coding practices resulting from community code reviews.
3. **Mature scientific libraries:** Because Python has a low entry barrier, experts from other academic backgrounds who have little programming experience can take advantage of advanced computational libraries. This results in comprehensive, domain-specific libraries that the community can take advantage of.

Why *not* Python?

Global Interpretter Lock (GIL)

Why *not* Python?

Global Interpretter Lock (GIL)

- Python's memory management is not thread safe.

Why *not* Python?

Global Interpretter Lock (GIL)

- Python's memory management is not thread safe.
- To cure this ailment, the creators of Python eventually created a lock for accessing Python objects.

Why *not* Python?

Global Interpretter Lock (GIL)

- Python's memory management is not thread safe.
- To cure this ailment, the creators of Python eventually created a lock for accessing Python objects.
- In this scheme, only one Python object can be accessed at a time.

Why *not* Python?

Global Interpretter Lock (GIL)

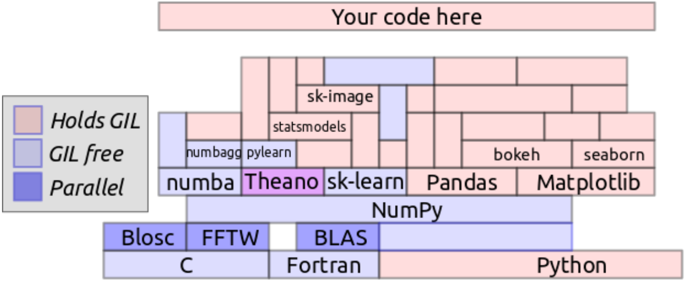
- Python's memory management is not thread safe.
- To cure this ailment, the creators of Python eventually created a lock for accessing Python objects.
- In this scheme, only one Python object can be accessed at a time.
- Each Python process has it's own GIL, but threads owned by the same process share a GIL.

Why *not* Python?

Global Interpretter Lock (GIL)

- Python's memory management is not thread safe.
- To cure this ailment, the creators of Python eventually created a lock for accessing Python objects.
- In this scheme, only one Python object can be accessed at a time.
- Each Python process has it's own GIL, but threads owned by the same process share a GIL.
- Many libraries have come to depend on the GIL's existence (although many are releasing the GIL).

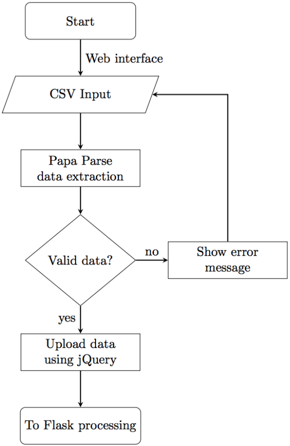
Why *not* Python?



Ecosystem

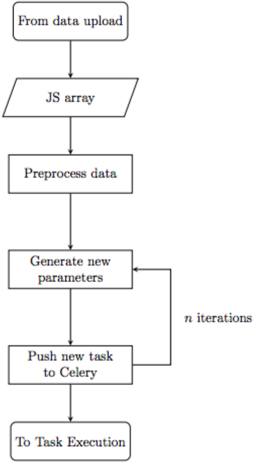
1. *System*
    - CoreOS
    - Docker
    - RabbitMQ
    - MongoDB
    - NGINX
2. *Server*
    - Python
    - Flask
    - Celery
    - NumPy
    - SciPy
    - Pandas
    - Theano
    - PyBrain
    - Keras
3. *Client*
    - HTML5
    - CSS
    - JS
    - Bootstrap
    - JQuery
    - plot.ly
    - Papa Parse

User Interface



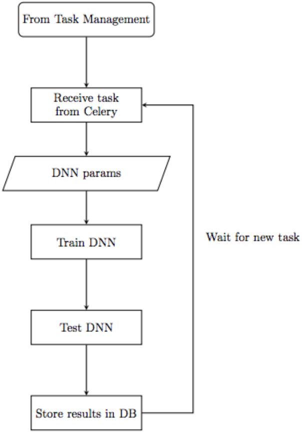
1. Users browse to a public website where they can upload a CSV file.
2. Select configuration settings for what DNNs to test.
3. User selects what data they would like to predict.
4. All of this information is pushed to the server.

Task Management



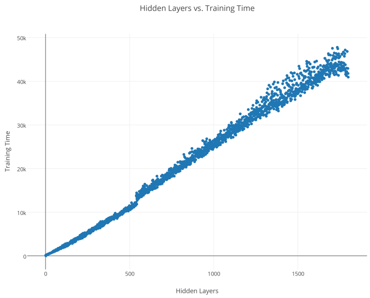
1. Server receives JS array containing the data from the client.
2. Preprocess the data using industry “best practices” such as scaling, one-hot encoding, and label-encoding.
3. Generate possible DNN layers and datasets for processing.
4. Push all of these mutations into Celery for task distribution.

Task Execution



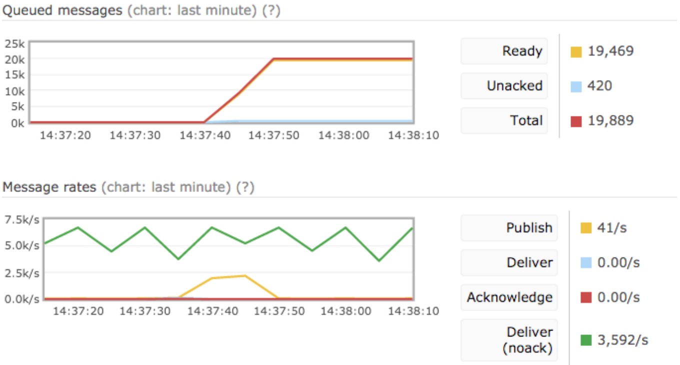
1. Pull task from central Celery task queue containing information on the DNN to be trained and the data.
2. Train DNN based on the settings stored within Celery.
3. Perform the desired accuracy tests.
4. Store results in the central MongoDB database.

Results Visualization



1. Navigate to website based on the session key provided earlier.
2. Website pulls results from database and visualizes it using plot.ly.
3. This data is accessible at any time using the key referenced above.

RabbitMQ Dashboard



Celery Dashboard

Active: 410	Processed: 67411	Failed: 22385	Succeeded: 43518	Retried: 0

Shut Down

	Worker Name	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
	celery@7336c79f2326	Offline	N/A	8	0	13	0	0.95, 0.64, 0.32
	celery@56b78d31d37c	Online	2	26	0	16	0	20.56, 11.79, 5.12
	celery@135d007cb43f	Offline	1	54	0	52	0	19.65, 17.3, 10.58
	celery@791d3c1aab6f	Offline	0	27	0	27	0	19.41, 13.97, 11.12

Results

- Every DNN tested, on small datasets and large datasets, seem to reach a critical mass of information storage around 500-700 hidden layers, wherein the DNNs performance will flatline for any number of layers greater than the critical mass point (probably due to overfitting).
- Training of DNN increases roughly linearly with the amount of layers added to the network for a significant number of layer combinations tried.
- Granular control over parameters greatly increases performance in DNNs. For instance, testing several different combinations of activation functions will produce some interesting results.

Questions?