Benchmarking Open Source Machine Learning Frameworks and Development of Auto-coding Interface for Entry-Level Engineers

Louis Yu
AE 8900 MAV – Special Problems, Spring 2017

ABSTRACT. The implementation of machine learning into multiple domains has renewed the interest of artificial intelligence enthusiasts to develop multiple frameworks to best resolve a problem. One of the most prevalent subsets, Neural Networks, has had a growing community of research enthusiasts. This paper covers a brief comparative study on four frameworks—Tensorflow, Theano, MXNet, and Caffe on several aspects: speed, utilization, and scalability onto commercial platforms. Tensorflow achieved the overall best score and was utilized for the language of choice for an interface between entry level engineers and an open source framework. Many aerospace engineers do not formally use lower level programming languages (Java, C, C++, etc.) and typically Matlab (versus Python) is the academic-program of choice for high-level interpreted language. However as machine learning permeates into the aerospace domain, immediate work may be stifled by some delay due to picking up a new syntax. The aim of this paper and likewise product, is to devise an interface that allows for problems to be devised in Matlab and then properly ported over to Tensorflow to utilize the extensive benefits this open-source tool provides for either GPU/CPU based hardware configuration.

Nomenclature

ANN	=	artificial neural networks
NN	=	neural networks
ML	=	machine learning
TF	=	Tensorflow
GiB	=	Gigabyte
VM	=	Virtual Machine
FLOPS	=	Floating Point Operations
CPU	=	Central Processing Unit
GPU	=	Graphical Processing Units
OS	=	Operating System
cuDNN	=	CUDA® Deep Neural Network Library
MNIST	=	Mixed National Institute of Standards and Technology database
ReLU	=	Rectifier Linear unit
LMDB	=	Lightning Memory-Mapped Database Manager
HDF5	=	Hierarchical Data Format

I. Introduction

ARTIFICIAL intelligence has been migrating towards new forms of knowledge representation and processing capabilities that emulate the human brain. There now exists new computational paradigms that have been established for new applications, namely artificial neural networks. An artificial neural network is best defined as a biologically inspired computational model that maintains a network architecture composed of artificial neurons. Each of these neurons and the network architecture layout are customizable with different mathematical parameters to tune in order to have the network perform a certain task: classify or regression. Neural networks today follow a production-like based scenario typically designated in a two-step process: development and deployment. Figure 1 shows that the neural network is first trained, i.e. labeled examples of inputs and desired outputs are passed through a neural network model. Second, the compiled and trained neural network is deployed to run inferences on typically unknown variations of inputs. This deployment allows a previously trained neural network to classify data at a certain confidence rate based off of its prior learning performance.

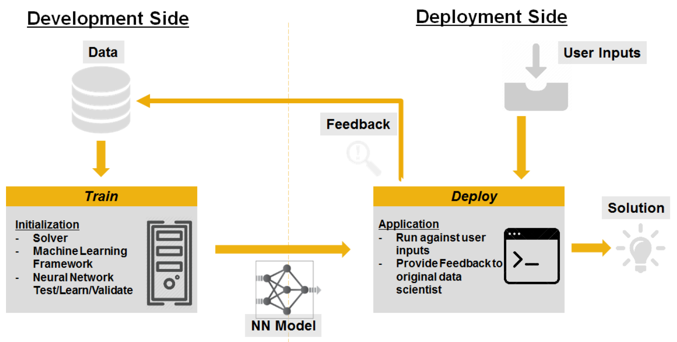

Development Side
Deployment Side
Data
User Inputs
Feedback
Train
Initialization
- Solver
- Machine Learning Framework
- Neural Network Test/Learn/Validate
NN Model
Deploy
Application
- Run against user inputs
- Provide Feedback to original data scientist
Solution

Figure 1. Production Pipeline of Neural Networks. A graphical adaptation of NVIDIA Production Pipeline showing how neural networks follow a two-stage feedback cycle.

Artificial neural networks have begun to permeate throughout various industry and domains as a useful way of solving complex issues in a more humanlike fashion. In addition, the more complex neural networks have begun a new sensational wave known as deep learning where the neurons and networks are utilizing learning methods based off of learning how data is represented. Deep learning itself has begun to be the frontrunner consideration for solutions to self-driving cars, automatic image captioning and real-time language translation.

Former state of the art algorithms and heuristic methods to tackle some of the largest problems have seen significant performance improvements based off of the commercial sectors momentum in publicly showcasing their deep learning projects. For instance, Google's AlphaGo deep learning network was the first computer program to beat a Human Go Professional and the results were showcased in late 2015 and early 2016[14]. In another instance, Baidu Deep Speech 2 became a well versed neural network that could perform English and mandarin speech recognition at a higher accuracy rate than humans (error rate of 3.7% vs 4% for humans)[15]. These two cases and many others like them provide continual public and research interest in realizing the potential that neural networks have for the future.

It would be unfair to not also attribute the major success of neural networks if it was not for innovative solutions on existing hardware, namely Graphical Processing Units. GPUs have brought the most influential shifts in this industry by providing a reduced computational and economical footprint for neural networks to train on. GPU computing contain characteristics that are perfectly aligned with neural networks: (1) they are inherently parallel, (2) filled with matrix operations and (3) contain FLOPS. The GPU footprint has already drawn down 80% of the typical run-time associated with neural networks that were usually represented as 5% of the code[4]. As many pieces have begun to come together for neural networks and subsequently deep learning methods to succeed, it's evident that the community it has fostered and the projects that have come to fruition will create a foundation where neural networks will become pervasive in throughout many industries.

II. Motivation

As attention continues to grow over neural networks and deep learning architectures, a wide variety of tools have broken ground and commenced gaining massive traction over their open source features and content. However each tool differs in their approach of programmatically building neural networks even though the underlying mathematics remains the same. The primary differences such as API language, syntax, and compilers utilized provide each machine learning framework the ability to reside isolated in a crowded space of open source frameworks. This however has garnered users to become specialist in a specific framework and require a deeper understanding of the intricacies of a toolbox versus being able to construct a neural network and tune the parameters to best suit a specific dataset.

Engineers throughout industry and academia that are not in tune with the computer science open-source community may see this as an inhibiting factor that is not easily mitigated. The number of computer languages used and content sharing procedures pervasive throughout an open-source community make it very intimidating to enter into this continually changing space. These inhibiting factors not only turn away engineers from the ability to characterize or classify datasets using open source neural network frameworks, but also prevent engineers from tapping into the most up-to-date technical capabilities available in this field. As these barriers may likely not be drawn down in the next few years due to a lack of standardization throughout open source communities, a need was found to allow engineers to utilize these tools without a heavy learning curve in the start-up phase and allow for a focus on implementing solutions for a wide variety of datasets. Thus in order to ensure the most optimal framework was used based off of several key metrics, this study aims to provide a clear and concise overview of four prevalent frameworks (Theano, Caffe, Tensorflow, and MXNet) and also develop a means of autocoding the syntax of one selected framework via a GUI interface in Matlab. In other words, the goal is to let the engineer focus on the mathematics and technicalities of neural networks versus engaging with intricacies of a specific framework.

III. Related Work

As over 29 machine learning frameworks have been pushed out to the open source community through GitHub, it is evident that many of these frameworks have been compared against one another in some form or fashion. Benchmarking over selections of frameworks have covered hardware/network configurations[1], evaluations over forward time and gradient computation time[2], and even operations based off of syntax for computational speed-up[3]. It has become apparent though that the various publications all focus on specific areas and each provide different recommendations for the overall top contender—as everything is relative to a specific hardware and software configuration. However these benchmarks do provide insight on the typical time factors associated with running a certain framework—as some frameworks are heavily dependent on CPUs to scale and be optimized[1].

Other frameworks tend to be heavily favoring GPU speed-ups with standard CPU performance relative to the other frameworks investigated[2]. Among all of the machine learning frameworks available, the most pervasive static configuration always references the GPU-based deep learning cudNN engine provided by NVIDIA. NVIDIA has maintained the forefront of standards for GPU performance improvements for training neural networks[4]. NVIDIA also offers their own graphical interface to build neural network models, known as DIGITS. DIGITS is a web-based application specifically for image classification, segmentation and object detection.

ASDL also holds an interface known as BRAINN[17] (Basic Regression Analysis for Integrated Neural Networks) which is a Matlab-based interface utilizing the neural network toolbox. The tool allows for a single layer architecture mapped to a set of inputs, user settings on epochs to iterate, convergence criteria to stop training, and many more options. Some inhibiting factors to using BRAINN and Matlab's Neural Network toolbox application are the license controls on Matlab and GPU-based processing requires the Parallel Computing Toolbox purchased to run. This study aims to draw on the capability to build neural networks via open source frameworks (that have CPU/GPU capabilities) and also utilize a familiar tool, Matlab, to act as the interface for a framework that is not native to Matlab. The benefits that entail this study is to identify the top open-source candidate and allow open sharing of neural networks through this framework while minimizing the ramp up time associated with an open source framework.

When surveying past ASDL projects, it is evident that many Neural Network projects could utilize the chosen machine learning framework whether it's for adaptive neural network modeling[19] or investigating the impact of additional hidden layers[20]. The variety of projects performed prior are key supporting studies that mold the framework for potential case studies that would be performed on the GUI developed. In addition, similar-like data sets with well-known equations such as Hill's equations morphed into a Neural Net optimization[21] helped shape datasets utilized in this study (airfoil dataset) to allow researchers to validate models between neural networks and empirical models.

IV. Benchmarking Setup

In order to obtain a holistic grading criteria of the four machine learning frameworks chosen, several benchmarks were configured to evaluate the performance against four datasets spanning different applications. The four datasets evaluated were: (1) MNIST[5], (2) Airfoil Self-Noise Data Set[6,7], (3) Naval Propulsion Plants[6,8], and (4) Household Power Consumption[6,9]. Non-performance related benchmarking was undertaken qualitatively by inspecting code syntax, adoption into commercial services and popularity amongst open-source communities (GitHub) & research institutions.

A. System Configuration

All experiments are performed on a single machine running on Windows 10 Pro (64 Bit) with Intel® Core™ i7-4790K CPU @ 4.00GHz 3.60 GHz; Nvidia GeForce GTX 750 Ti (Ver. 378.66); 16 GiB DDR3 memory; and WD Blue HDD. Table 1 below shows the software framework and versions utilized for the evaluation.

Table 1 Table of Machine Learning Frameworks Investigated

Framework	Tensorflow[10]	Theano[11]	MXNet[12]	Caffe[13]
Version	R1.00	0.8.2	0.9.3	1.0.0.rc3
Core Language	C++, Python	Python	C++	C++
Interface Language	C++, Python	Python	C++, Python, Julia, Matlab, Javascript, Go, R, Scala	C, C++, Python, Matlab
cuDNN Support	V5.1	V5	v.5.1	V5.0
Python Version	V3.5	V2.7	V2.7	V2.7
Docker Image Pulled	Latest:py3	Latest	Latest	Latest

Docker v.1.13.1 for Windows was chosen as the runtime environment for each set of experiments to be run in isolated containers via HyperV. Figure 2 shows how Docker allows for each ML framework pre-compiled image to be invoked on separate Docker containers on the VM and separately monitored. Docker was allocated (of the original machine) 8 GiB of memory and all 8 CPU Cores. No GPU runs were conducted since Nvidia Docker currently does not have a windows interface. Each Machine Learning Framework in Table 1 were run separately from one another (e.g. using all CPU resources on its own and not sharing between other frameworks).

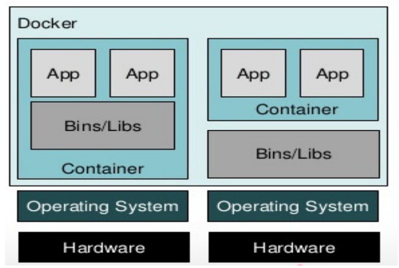


Figure 2. Docker Image over hardware. Docker containers are isolated, but share the same OS, and where appropriate, bins/libraries.

B. Datasets investigated

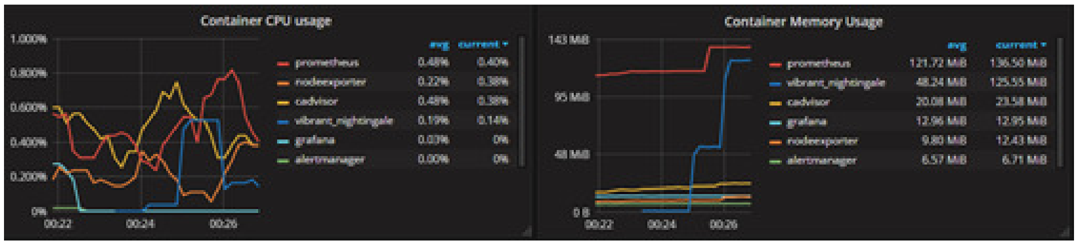
The datasets evaluated were intended to provide a swath of values that go from as low as thousands of data points up to over 2 million data points. The goal was to characterize varying sizes of data files in order to report how each framework could scale up in regards to datasets. Certain datasets were not provided with a specific training & testing dataset (like MNIST), so an 80/20 training/testing split was utilized to separate the data. Table 2 below covers a brief summary of all the different datasets covered and certain attributes they contained.[*]

Table 2 Table of Datasets Investigated and Benchmarked

	MNIST[5]	Airfoil Self-Noise[7]	Naval Propulsion Plant[8]	Household Power Consumption[9]
Attribute Characteristics	Integer	Real	Real	Real
# of Instances	60000 (Training) 10000 (Testing)	1503	11934	2075259
# of Attributes*	10 (28 x 28 image)	6	16	9
Testing/Training Data	Testing & Training Provided	Data Only	Data Only	Data Only
Data Provider	NYU/Google Labs	UCI ML Repository	UCI ML Repository	UCI ML Repository

C. Monitoring Solution

A blend of 5 different Docker plugins were utilized to capture each containers specific CPU usage and memory usage. Prometheus, AlertManager, Grafana, NodeExporter, and cAdvisor were compiled in separate Docker containers and communicated via localhost proxies. These overlaying containers were segregated from one another when collecting each of the performance metrics over each set of runs. Figure 3 shows how each container reports its own metrics. These metrics allow for timestamps to be utilized in calculating the duration of how long each network took to complete the training of a network.



	avg	current
prometheus	0.48%	0.40%
nodeexporter	0.22%	0.38%
cadvisor	0.48%	0.38%
vibrant_nightingale	0.19%	0.14%
grafana	0.03%	0%
alertmanager	0.00%	0%

	avg	current
prometheus	121.72 MiB	136.50 MiB
vibrant_nightingale	48.24 MiB	125.55 MiB
cadvisor	20.08 MiB	23.58 MiB
grafana	12.96 MiB	12.95 MiB
nodeexporter	9.80 MiB	12.43 MiB
alertmanager	6.57 MiB	6.71 MiB

Figure 3. Docker Monitoring Solution. Docker containers are isolated and each container reports back its CPU and memory usage.

V. Benchmark Results

The experiment results are presented in two subsections: quantitative results and qualitative results. For quantitative results, the focus was on running a specific dataset over a chosen neural network framework in each

[*] Attributes for MNIST refer to the classes of objects definable (e.g. digits from 0 to 9), whereas for each of the other datasets the attributes refer to a feature of the data collected for a certain configuration or timestamp

separate tool. The Machine Learning framework was monitored for hardware utilization (CPU usage and memory usage), as well as time elapsed from start to finish for the entire network to be trained. This format was conducted to take a systematic view of an entire training session for each dataset versus metrics concerning the highest contributing Pareto factors in neural network training (forward time, gradient computation time)[1,2]. For qualitative results, the focus on this interpretation was to identify key characteristics of each network as they related to syntax, utilization, and extensibility.

A. Common Coding Structure across Frameworks

An important feature in model comparisons across each of the four frameworks required the maximum use of consistent initialization methods, data formatting, and random states to ensure all frameworks started from a common ground. For this, the Python functions were built to be modular and inserted into all ML frameworks to conduct data partitioning, data read-in and datatable formats to be exactly the same. The only exception to this for benchmarking was for the Caffe framework since it relied on a separate data format (HDF5) for regression analysis and Lightning Memory-Mapped Database (LMDB) for image classification.

Several key functions were used throughout the initialization methods are described in Table 3.

Table 3 Table of Common Python Functions Used

Python Function	Description
`def encode_numeric_zscore(df, name,mean=None,sd=None)`	Function that computes the following on a set of data: $$z = \frac{x-\mu}{\sigma}$$ Where μ is the mean, σ is the standard deviation and z is the distance between raw score (x) and population mean.
`def to_xy(df,target)`	Function that converts a pandas dataframe to an x,y input that each ML Framework utilizes. For all cases covered, it converts the x and y variables into float64 format for regression analysis.
`def build_mlp(input_var=None)`	Builds each respective neural network framework in the ML specific syntax and returns a neural network object to be fitted and predicted from.
`Train_test_split(x, y, test_size=0.20, random_state=42)`	A python module (sklearn) function that takes in x/y variables and splits the dataset randomly into an 80% training and 20% validation dataset. The random state can be maintained using the same code value across each framework.

All networks utilized the Adaptive Moment Estimation (Adam) Gradient Descent Optimization. This method computes an adaptive learning rate for each parameter, generalized in equation 1. The Adam method has been shown to work well in practice and compares favorably when compared against other learning-method algorithms[18].

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} m_t \tag{1}$$

Where m_t is average of past gradients, η is learning rate, and θ is a parameter.

The loss function utilized for all networks was the Mean Squared Error which is shown in equation 2. This loss function computes the mean over the entire data set over the actual value y_i and the predicted value $\hat{y_i}$ based off how the current neural network is defined at that training or testing instance.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2 \tag{2}$$

An 80/20 split in the dataset was utilized for the training/validation datasets and a random state of 42 was used (pseudo-random number to designate random sampling). The split designated was consistent across all three regression test sets in order to keep certain parameters fixed. In addition, since two of the frameworks used iterations (versus epochs) to step through training the data, the following equation 3 is used to translate between epochs and iterations.

$$N_{Iterations} = \frac{N_{Sample\ Size}}{B_{Batch\ Size}} * N_{Epochs} \tag{3}$$

B. MNIST Image Classification

The first benchmark dataset was over the MNIST dataset that holds a database of handwritten digits containing a training set of 60,000 examples and a test set of 10,000 examples. A validation set of 5000 examples was taken out of the training sets' 60,000 examples. The MNIST digits are centered in a 28x28 image with the center at the mass of the pixels. This dataset was trained on a LeNet5 convolutional neural network developed by Yann Lecun[16]. This network shown in Figure 4 contains two convolutional layers (containing both an activation function and subsampling frame) and three fully connected hidden layers that draw down the outputs from 120 to a one-hot vector of 10 to classify the 10 attributes. The input parameters for the network was a batch size of 64, 2 filters with a filter width and height of 5x5. The learning rate was set to 0.05 initially and allowed to change with momentum factors in each tool.

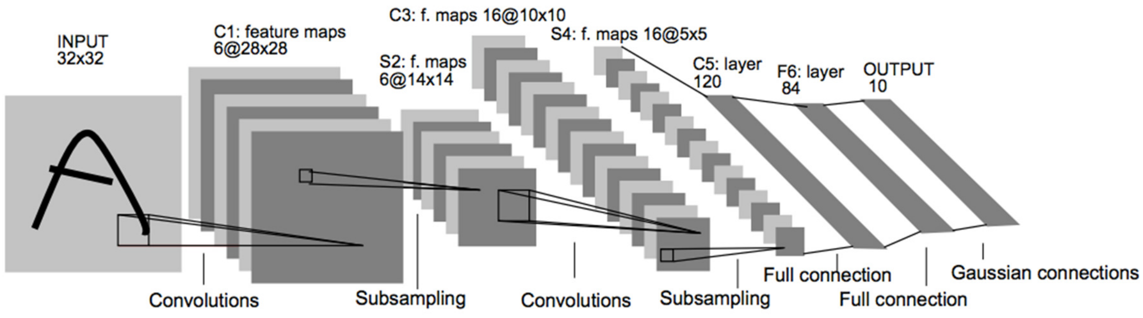
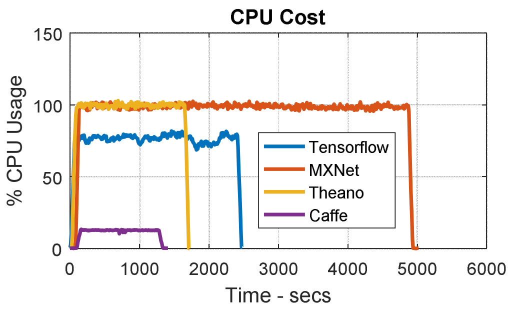
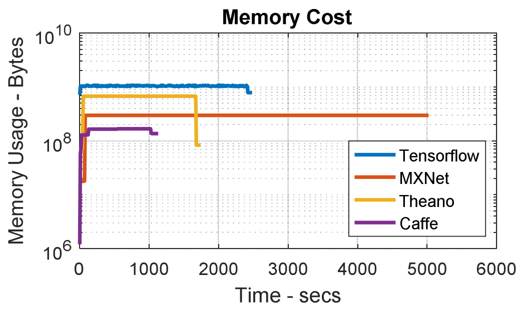


Figure 4. LeNet5 Neural Network Architecture. The architecture shows 2 convolutional layers with subsampling and 3 fully connected output layers to drive the output to 10.

Figure 5 shows the computational results gathered on a complete run of the MNIST database over each of the ML frameworks. It can be seen that of all the frameworks, MXNet and Theano both consumed the most CPU usage throughout the entire training and validation phase. Tensorflow allocated the most memory during the entire training phase. MXNet took the longest to complete and Caffe was the quickest. In addition, Caffe also only took a fraction of the time as all of the other frameworks took upwards of 30 minutes to complete.





a) **b)**

Figure 5. MNIST Computational Results. Results show a) CPU usage and b) Memory used for each of the four frameworks investigated on the MNIST LeNet5 Neural Network

Some of the contributing factors for the behavior each ML framework showcased is foremost that Caffe (Docker Image) could not utilize all 8 CPU cores due to additional libraries that could not compile properly. In addition, Caffe utilizes a different data format (Lightning Memory-Mapped Database - LMDB) versus the compressed zip files format that was used for the other three frameworks. The final LeNet5 classification results based off of accuracy against the entire dataset are shown in Table 4. The accuracies reported show that the LeNet5 structure across all four frameworks ultimately provided similar results for digit classification.

Table 4 Table of Final Accuracies for LeNet5 Network on MNIST Dataset

ML Framework	Train Accuracy	Test Accuracy
Tensorflow	99.3%	99.3%
Theano	99.02%	98.98%
MXNet	99.01%	99.06%
Caffe	99.08%	99.01%

C. Airfoil Self Noise Data Set

The airfoil self-noise dataset contains 1503 x 6 data entries and consists of the following variables: frequency (Hz), angle of attack (degrees), chord length (m), span (m), freestream velocity (ms^{-1}), suction side displacement thickness (m), retarded observer distance (m), observer position angles [theta and phi] (degrees), and sound press level (dB).

This dataset also has a semi-empirical model known as the Brooks model which was formulated from a set of wind tunnel tests. Therefore, in following the methodology of the Brooks model, the 9 variables are reduced to 5 inputs: frequency, angle of attack, chord length, freestream velocity, and suction side displacement thickness (m). The output/target variable was the scaled sound pressure level.

The network architecture chosen was three layers of fully connected layers with ReLU activation as summarized in Table 5.

Table 5 Airfoil Dataset Neural Network Architecture

Neural Network Setup	Hidden Neurons	Activation Type
Layer 1	25	ReLU
Layer 2	5	ReLU
Layer 3 (Output)	1	None

The batch size and learning rate were chosen as 128 and 0.001 respectively. The number of epochs run over the dataset was chosen as 5000 (or 46975 iterations).

The training of the dataset shown in Figure 6 logs the training cost for each of the four frameworks. It can be evidently seen that MXNet has the highest final mean squared error (per the loss function). The one that consumed the most CPU usage was MXNet by far and nearly all frameworks utilized the same amount of memory during the training phase. It can be seen in the CPU/Memory cost plots that Tensorflow took the longest to complete and Theano was the quickest.

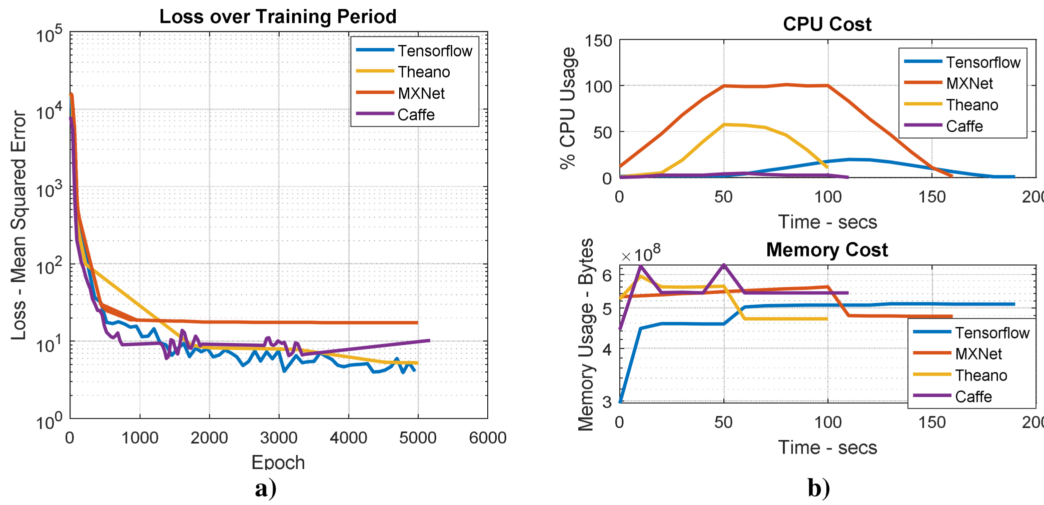


Figure 6. Training Statistics for Airfoil Dataset. Results show a) Loss Evaluation over Epochs and b) the Cost on Memory and CPU Usage during training

The final regression results are shown in Figure 7. Even with the performance of training as shown above by each of the frameworks, Tensorflow and Theano are nearly identical in a linear regression case. Both Caffe and MXNet had issues in converging to a more condensed prediction and had a heavier dose of outliers that seemed to drive the system to have R^2 values much lower than Tensorflow and Theano.

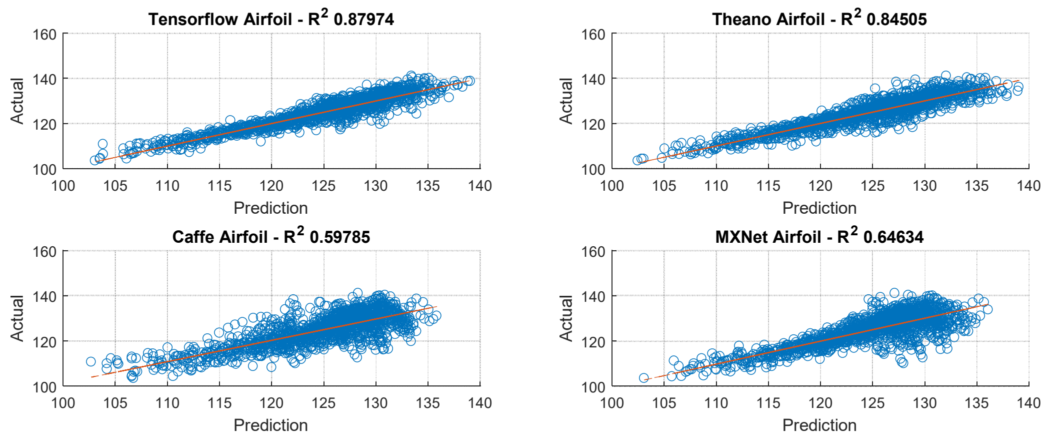


Figure 7. Regression Statistics for Airfoil Dataset. Results show Tensorflow (top-left), Theano (top-right), Caffe (bottom-left), and MXNet (bottom-right) Regression results after Training were completed.

D. Condition Based Maintenance of Naval Propulsion Plants Data Set

The naval propulsion dataset contains 11934 x 16 data entries and consists of the following variables: lever position, ship speed, (4) gas turbine parameters, starboard torque, port propeller torque, (2) high pressure parameters, (4) gas turbine parameters, turbine injection control, fuel flow, and (2) decay state coefficient.

This dataset was built by a numerical simulator of a Gas Turbine model mounted on a naval vessel (Frigate) and was formulated as a way to monitor decay state coefficients. The 16 input variables were reduced down to 15, with the gas turbine compressor inlet air temperature removed since it is a constant temperature. The output/target variables were the gas compressor/turbine decay state coefficients.

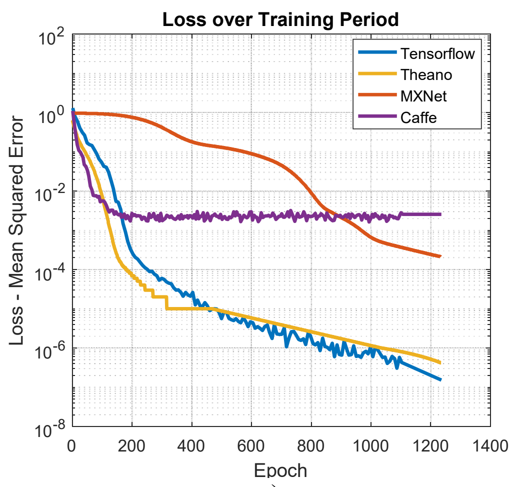
The network architecture chosen was four layers of fully connected layers with ReLU activation shown in Table 6.

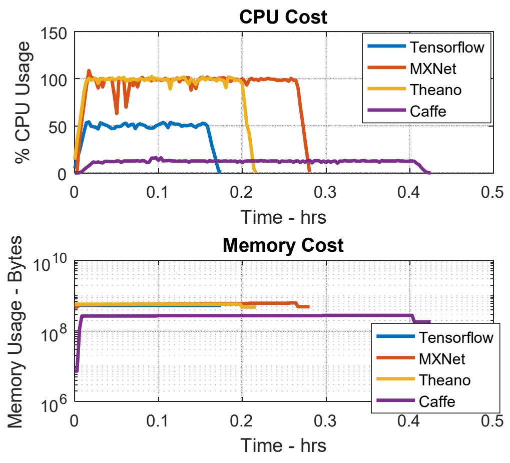
Table 6 Naval Propulsion Dataset Neural Network Architecture

Neural Network Setup	Hidden Neurons	Activation Type
Layer 1	500	ReLU
Layer 2	255	ReLU
Layer 3	5	ReLU
Layer 4 (Output)	2	None

The batch size and learning rate were chosen as 128 and 0.000001 respectively. The number of epochs run over the dataset was chosen as 1235 (or 92150 iterations). The goal of this network was to demonstrate multi-output regression for two variables or more would not degrade the R^2 value or severely impact the performance during training.

The training of the dataset shown in Figure 8 logs the training cost for each of the four frameworks. As compared to the airfoil case, it can be seen that Tensorflow and Theano had found convergence much quicker and accelerated down to a smaller loss evaluation than both Caffe and MXNet. It seemed as though Caffe had found a local minimum and could not search for any better solutions, whereas MXNet was gradually lowering its loss evaluation but still was far off of Tensorflow and Theano loss values. In comparison to the actual cost of training it on the CPU, it is evident that Caffe began to take the longest due to the issues described prior in the MNIST section. In addition, the CPU usage was at the maximum for both Theano and MXNet during the training session whereas Tensorflow was more lax and had only taken up roughly 50% of the CPU. Memory usage was relatively consistent across all four frameworks.





a) **b)**

Figure 8. Training Statistics for Naval Dataset. Results show a) Loss Evaluation over Epochs and b) the Cost on Memory and CPU Usage during training

The final regression results are shown in Figure 9. It can be seen that once again Tensorflow and Theano had been able to converge the multi-output regression case with high R^2 values where overall Tensorflow barely edged out Theano for the top scores. Evidently, Caffe and MXNet for the exact same neural network setup had failed to converge onto a probable solution and ultimately had a bunching up of the parameters without any appropriate answer.

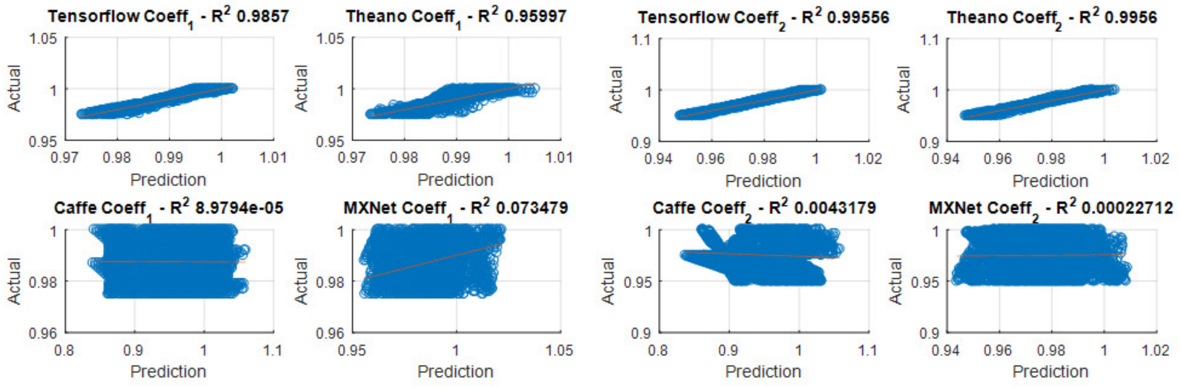


Figure 9. Regression Statistics for Airfoil Dataset. Results show two output coefficient regressions. For each coefficient: Tensorflow (top-left), Theano (top-right), Caffe (bottom-left), and MXNet (bottom-right) Regression results after training were completed.

E. Individual Household Electric Power Consumption Data Set

The household power consumption dataset contains 2075526 x 9 data entries and consists of the following variables: Date, Time, Global Active Power, Global Reactive Power, Voltage, Global Intensity, Sub Metering 1, Sub Metering 2, and Sub Metering 3. Each sub metering variable corresponds to active energy consumed in the household by electrical equipment based off of location (kitchen, laundry room, and heating/AC).

This dataset was created through measurements of electric power consumption in one household with a one-minute sampling rate over a 4 year period. The dataset used 6 of the original nine variables, with date and time being omitted, as input variables to fit the output/target variable: voltage.

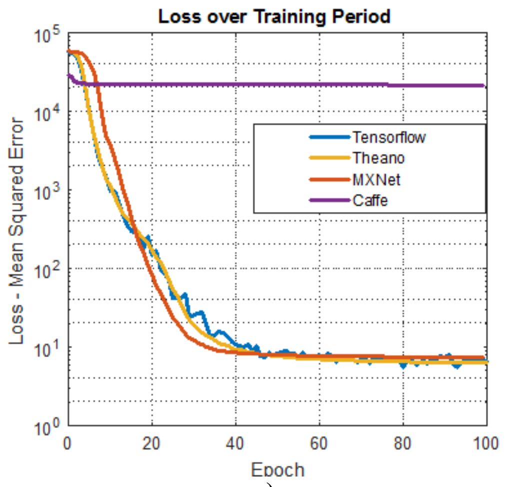
The network architecture chosen was a five layer fully connected neural network with ReLU activation shown in Table 7.

Table 7 Household Electric Power Consumption Dataset Neural Network Architecture

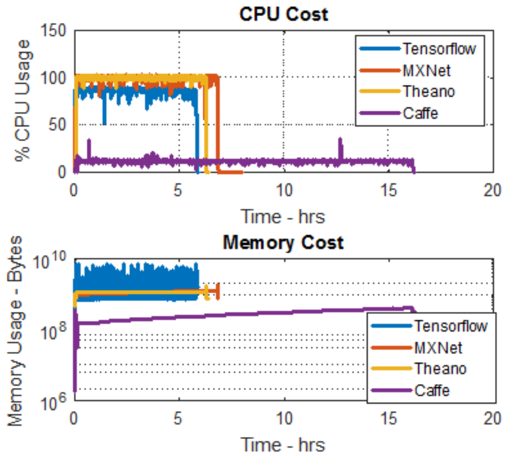
Neural Network Setup	Hidden Neurons	Activation Type
Layer 1	64	ReLU
Layer 2	512	ReLU
Layer 3	1024	ReLU
Layer 4	128	ReLU
Layer 5 (Output)	1	None

The batch size and learning rate were chosen as 512 and 0.000001 respectively. The number of epochs run over the dataset was chosen as 100 (or 320700 iterations).

The training of the dataset shown in Figure 10 logs the training cost for each of the four frameworks. This household case took the longest and showcased a revelation that MXNet could overshoot the loss criteria in comparison to Tensorflow/Theano, and yet then at a critical time-step (epoch 19) could then surpass the loss and converge relatively close to both Tensorflow and Theano's mean squared error. Caffe in this case once again found a local minimum and could not find another solution that was better fitting. The behavior of the CPU/Memory cost on the system was mimicking the same behavior as the Naval Propulsion training period, but with Tensorflow at a rough 80% of CPU usage overall. It was evident that while training the Tensorflow network, the memory regularly spiked up to 7 GBs and then continually dropped back down which may be cause for concern if datasets were larger or if not enough memory was allocated.



a)



b)

Figure 10. Training Statistics for Household Power Consumption Dataset. Results show a) Loss Evaluation over Epochs and b) the Cost on Memory and CPU Usage during training

The final regression results are shown in Figure 11. As the loss criteria had shown above, three of the networks could find some form of a solution with roughly R^2 values in the 0.35-0.42 range. The figure shows only 1/7th of the dataset for a rough comparison and it is evident that Caffe had failed to resolve a solution at all. What is most surprising is how MXNet was able to converge onto some solution since the other two trials above had shown that MXNet behaved much more like Caffe then Tensorflow and Theano. In this case, it may be suggested that for future training sessions, to extend the typical training time in order to allow MXNet to catch up in the loss criteria.

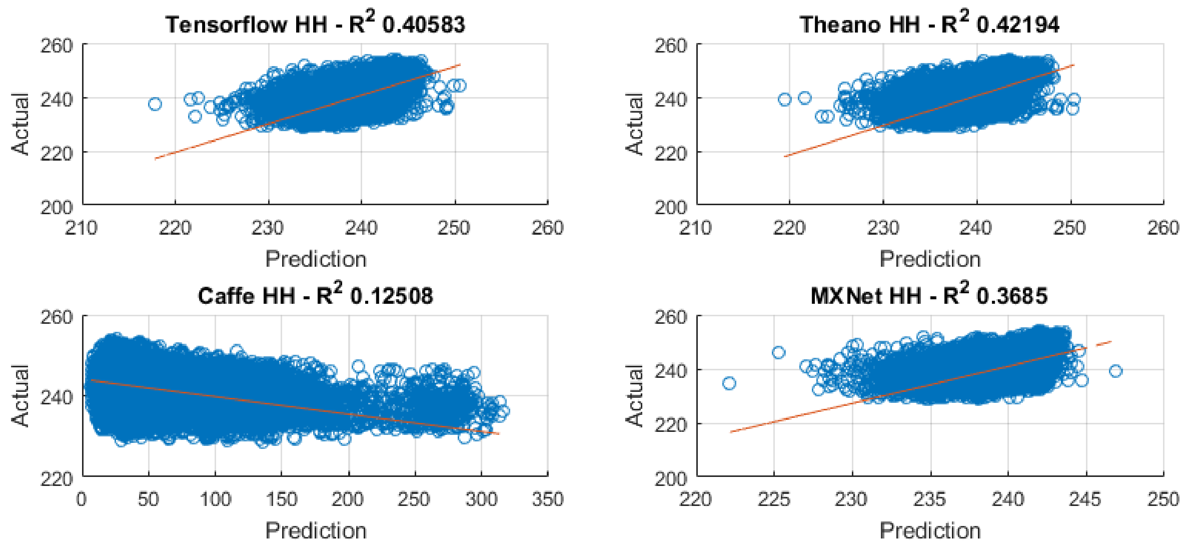


Figure 11. Regression Statistics for Household Dataset. Results show 1/7th of total data for Household power consumption for each framework labeled Tensorflow (top-left), Theano (top-right), Caffe (bottom-left), and MXNet (bottom-right) Regression results after training were completed.

The compiled regression statistics for the entire 2 million dataset is published in Table 8. The partitioned dataset found in Figure 11 is roughly similar to the compiled set of data.

Table 8 Table of Final Regression Statistics for Household Electric Power Consumption

ML Framework	Coefficient of Determination (R^2)
Tensorflow	0.384
Theano	0.400
MXNet	0.295
Caffe	0.116

F. Qualitative Results

The performance of a machine learning framework may provide great insight, but one of the biggest challenges is having the library installed and all accompanying dependencies propagated throughout the system. Throughout the beginning of this study, the windows-based installations of Theano, MXNet, and Caffe were filled with error messages that were machine-independent (e.g. certain system configurations cause different error messages). Thus, community-based support of installation issues were limited and typically unrelated to problems a user may have when attempting to install each framework. These three packages were more suited for preconfigured software package files that Linux Machines utilize to install packages on a system (aka apt-get) versus the Windows format of manipulating batch scripts to suit the needs of the individual's computer configuration. Tensorflow out of all 4 had the smoothest installation package for Windows. The installation issues were the main cause for utilizing precompiled images on Docker.

Beyond the installation phase, a deeper dive into the community was investigated and a general gauge for the community inertia was captured via GitHub for commits and forks of the official databases. In GitHub, commits are software updates & revisions versus forks being community-led projects that started from the original database and modified on their own. It was evident that Tensorflow was the most popular amongst the community for forks and came in second for number of commits—thus high reliability and community engagement was evident. In addition, cloud services were all available to be ported onto Amazon's AWS service with Tensorflow having the additional Google Cloud as an option. Higher level APIs designed to provide easier guidance in building neural networks in a specific framework were only evident in both Tensorflow and Theano. These two both had an internal high-level API (TFLearn and Lasagne respectively) while also sharing an independent open-source high-level API known as Keras.

For the implementation phase of editing Python code and working in the context of generic classification problems, it was found that MXNet had the most simplistic and compact form of neural network definitions and defining how to begin training. Caffe was found to have the hardest syntax to comprehend as it not only involved the Python API but also heavily depended on understanding a JSON format used for defining how a neural network was configured. In addition, any external data that was to be used on Caffe had to be compressed into either a HDF5 or LMBD data format. Several configuration scripts in batch or bash format also had to be edited for the network to run appropriately in Caffe. A summary of qualitative inspection of each network is found in Table 9[†].

Table 9 Qualitative Assessment of Neural Network Frameworks

	Tensorflow	Theano	MXNet	Caffe
GitHub Commits|Forks	15049 | 23183	25221 | 2014	4931 | 3226	3914 | 10105
Cloud Service	Google Cloud, Amazon AWS	Amazon AWS	Amazon AWS	Amazon AWS
Higher-Level APIs	Keras, TFLearn	Keras, Lasagne, NoLearn	---	---
Syntax	Moderate	Moderate	Simple	Complex (JSON + Python)
Graphical APIs	Tensorboard	matplotlib	matplotlib	matplotlib

G. Resulting Machine Learning Framework Chosen

Across all of the frameworks and metrics utilized in this study, it was shown that Tensorflow was the winning candidate. Although this study only reviews CPU-based training of neural networks, the everyday engineer may very likely not have immediate access to GPUs when conducting research. Therefore all additional possibilities that each framework could have unlocked in a GPU-based setting were not used as a judging criteria for this study. The key aspects for having Tensorflow as the winning candidate was due to its heavy community of support and up-to-date (in comparison to other frameworks) documentation; in addition to its internally built high-level APIs that allow casual users to no have to delve too deep into all the hyper-parameters available. The performance of Tensorflow seemed to be consistently performing favorably against most other frameworks in addition to being quicker in time spent training over a CPU-based platform for larger data sets. All of these factors are key areas that an entry level engineer may need to utilize or take advantage of when becoming introduced to open-source machine learning frameworks.

VI. GUI Interface Development

A. Functional Model

For the creation of a development package to suit the needs of a data scientist/engineer, four main phases will occur in the interface: (1) environment set up and data loading, (2) neural network creation, (3) interface to python, and (4) neural network training as summarized in Figure 12. The goal of each of these steps is to maintain modularity in the interface to allow for future modifications in a given phase as need be. Data differentiation and neural network configuration is highly important to the user and are the most configurable features. As some research projects may need to classify text or labels (e.g. [Bat, Cat Dog, Bat] would be [1, 2, 3, 1] respectively), a basic text encoding feature will be made available. Large datasets will be managed appropriately by maintaining a datastore variable that doesn't immediately read the entire file into memory. All neurons/layers that will be used will be based on a pre-defined library with editable parameters to be adjusted based upon user preferences. This library can then be added onto by other users to share different configurations.

[†] All assessments are made as of 03/9/2017

Additional configurations to be considered for implementation are GPU/CPU switching, ability to push python scripts onto Docker images/containers, and a queue capability for several different networks to be selected to run against a specific dataset.

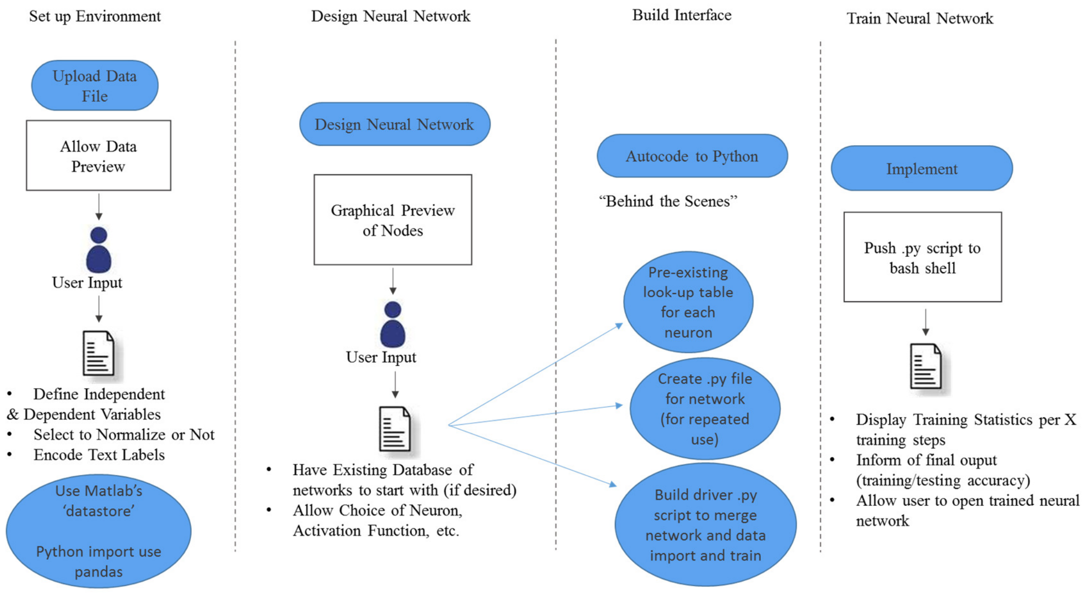
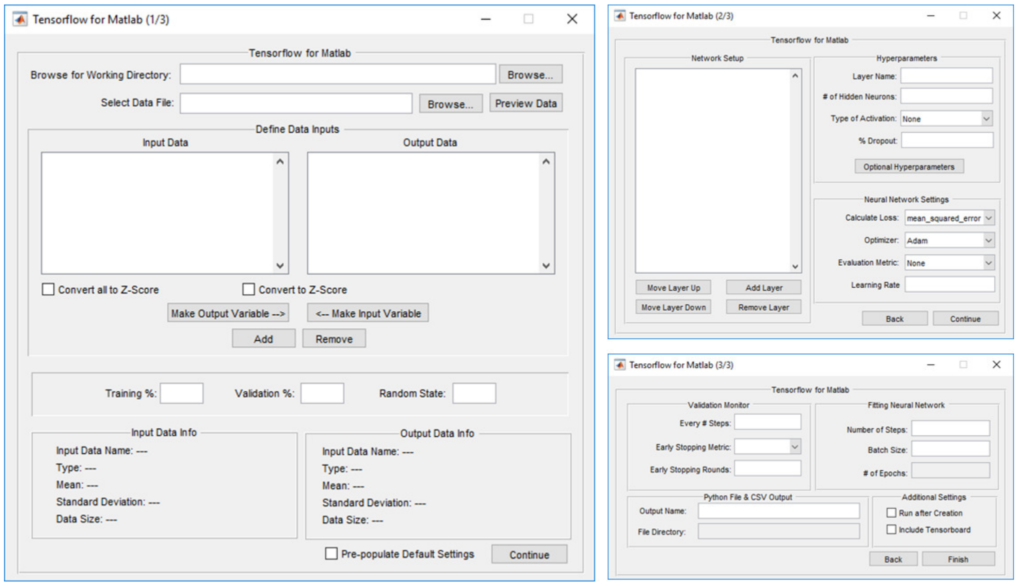

Set up Environment
Upload Data File
Allow Data Preview
User Input
• Define Independent & Dependent Variables
• Select to Normalize or Not
• Encode Text Labels
Use Matlab's 'datastore'
Python import use pandas
Design Neural Network
Design Neural Network
Graphical Preview of Nodes
User Input
• Have Existing Database of networks to start with (if desired)
• Allow Choice of Neuron, Activation Function, etc.
Build Interface
Autocode to Python
"Behind the Scenes"
Pre-existing look-up table for each neuron
Create .py file for network (for repeated use)
Build driver .py script to merge network and data import and train
Train Neural Network
Implement
Push .py script to bash shell
• Display Training Statistics per X training steps
• Inform of final ouput (training/testing accuracy)
• Allow user to open trained neural network

Figure 12. GUI Interface Functional Layout. Functional design of the GUI Interface package detailing four phases that will be modularized in Matlab code

B. Tensorflow for Matlab GUI Interface

The finalized interface built for autocoding neural networks from a GUI interface to python code was configured in a way that would allow for a wide array of hyperparameters to be modified. The expectation for the end-user was that several key Python Modules would be installed prior to use (pandas, os, shutil, numpy, tensorflow, sklearn). In addition, either Python versions 2.7 or 3.5 were supported for the end user to utilize. Currently the interface is meant for Tensorflow v1.00, but may likely handle newer versions as functions are meant to minimize reliance on Tensorflow options. The interface was broken down into three figures, as displayed in Figure 13.



Input Data Name: ---
Type: ---
Mean: ---
Standard Deviation: ---
Data Size: ---

Input Data Name: ---
Type: ---
Mean: ---
Standard Deviation: ---
Data Size: ---

Type of Activation: None

Calculate Loss: mean_squared_error
Optimizer: Adam
Evaluation Metric: None

Figure 13. Tensorflow for Matlab GUI Interface. Three panel interface designed to allow easy implementation of Neural Network designs on a dataset in Tensorflow via Matlab GUI

Based off of the three test cases above (airfoil, naval, household), it was evident what features and hyperparameters were typically altered in order to build each set of scripts for all frameworks. In addition, other ASDL projects that concentrated over Neural Network development also guided the interface based off of unique challenges each one faced and focused on[19,20,21]. Besson constructed an approach to build autonomously ANN surrogate models via interfacing with a design of experiments to BRIANN—the same could be done in Tensorflow by exchanging it in place of BRIANN during the development process described[19]. Perners' investigation on the impact of additional hidden layers on Neural network Performance could very easily be emulated similarly in Tensorflow and the approach and delivery of his study via BRIANN could also be replicated exactly as his study showed[20]. Akinlis' method of building a neural network to generalize non-spherical gravitational potentials for orbit controls of a small spacecraft can likewise be emulated in Tensorflow and validated against the Hill's equations as the Airfoil Dataset has also shown[21].

This knowledge then transferred into requirements for how the GUI was developed and standards set in place. The interface expects a data file that is in a structured format where the first row displays two values (rows, columns). The second row refers to the column names for each variable. The remaining rows would be the dataset separated by instance (row-major data stored). In order to maintain large data files (>100 MBs), the Matlab tabular-datastore format was utilized to minimize performance issues with the end user. It may be expected that novice users may come across the GUI so a "Pre-populate Default Settings" checkbox was placed on the first panel of the GUI to allow settings and options be initialized before the user continues through the GUI. The interface also offers the user the option to build a python file and run it through the system commands via the Matlab container or it can just create the python file and allow the user to make additional edits if need be before running the script. This consideration was made in the expectation that as the Tensorflow module is continually built-up, there are either options or features that may appear in later releases that may not be covered currently in V1.00.

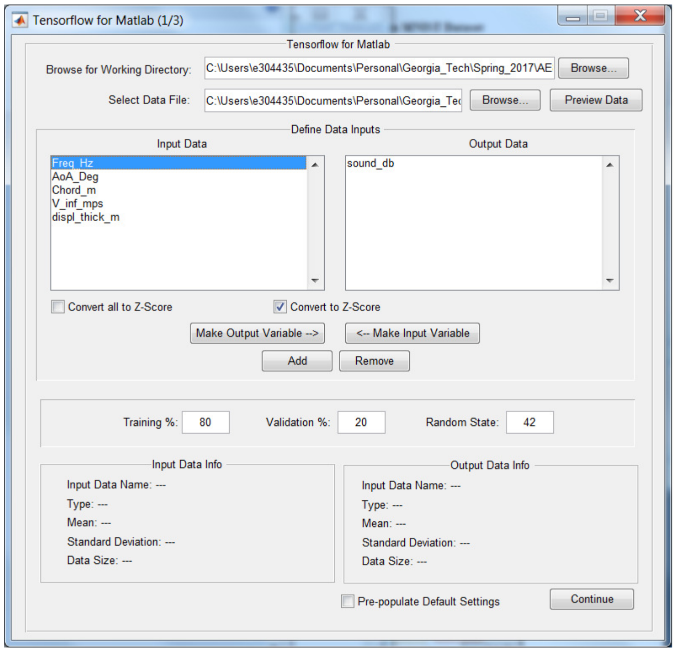
The parsing script that constructs the python file builds each section of the autocoding file based off of the user options. The major sections that are built, in following order, are an import module block, a function definition block (ranging from data file extraction up to model creation), and finally the driver script that designates file locations, data formats, and runs & predicts the final results into an excel file. Model files are also saved into a model directory chosen by the user in order for the user to then take the trained neural network model for deployment purposes.

Tensorflow also has a conditional 'with' statement that can automatically include a set of GPU devices if that hardware is available to the user. By utilizing Matlab's 'gpudevice' command to identify if a CUDA enabled device is present on the system, it can automatically recognize the GPU and set the device number into the Model Creation to allow for the GPU to be utilized during training. This feature is also toggled inside the GUI on the final panel.

C. Tensorflow for Matlab Example

A quick run through the application shows the parameters used to emulate the same set-up as the airfoil self noise database. This is meant to demonstrate the ease of utilizing the interface to conduct the same study as done in the study described in this study. First, the user will insert the working directory and load the data file into the application. Then the user will specify out of the 6 variables inside the excel datasheet, which resides as an output or input variable. In this case, the sound pressure in dB was pushed over to the output data side. Next, each of the 5 input variables was set to be converted to the Z-score value, as defined previously in Table 3. The training and validation percentages were set to 80/20 % respectively and the random state was set to 42. All of these parameters can be viewed in Figure 14.

The next step of the process is to build the neural network layers. First, the user can add each layer to the listbox by selecting 'Add Layer' button which also brings up a separate dialog for all the



Browse for Working Directory: C:\Users\e304435\Documents\Personal\Georgia_Tech\Spring_2017\AE

Select Data File: C:\Users\e304435\Documents\Personal\Georgia_Tec

Input Data: Freq_Hz, AoA_Deg, Chord_m, V_inf_mps, displ_thick_m

Output Data: sound_db

☐ Convert all to Z-Score ✓ Convert to Z-Score

Training %: 80 Validation %: 20 Random State: 42

Input Data Name: ---
Type: ---
Mean: ---
Standard Deviation: ---
Data Size: ---

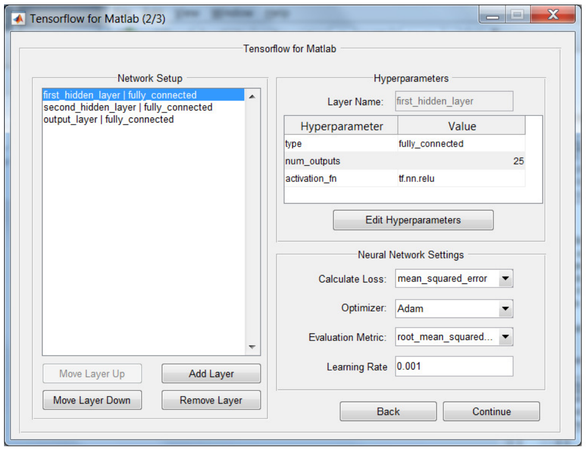
Input Data Name: ---
Type: ---
Mean: ---
Standard Deviation: ---
Data Size: ---

☐ Pre-populate Default Settings

Figure 14. Panel 1 of Airfoil Self-Noise. Snapshot of Tensorflow for Matlab GUI Interface Panel 1 inputs

hyperparamters to be available be to the user. These hyper-parameters are synced to all of the tensorflow contrib.layers structure. In this instance, only three layers were created and given the respective outputs of [25 5 1]. The first two layers had the ReLU activation function and the last having none as the output. As users select between layers under the 'Network Setup' box, all hyperparameters edited are shown in the right-hand side. In addition, neural network settings can be altered on this panel in regards to the optimizer chosen, how loss is calculated, evaluation metrics and the learning rate. These inputs are shown in Figure 15.

Finally the user can move onto the third and final panel to set-up the validation monitor, set the output file name and determine the number of steps and batch size to train to. In addition, an epoch calculator is embedded to let the user know the number of epochs the user can expect to cover based upon (1) size of sample data, (2) number of steps/iterations, and (3) batch size chosen. The user can then choose the run the file after creation in order to immediately push the script through python. All standard outputs are then routed and echoed inside the Matlab Command Window. The settings chosen are shown in Figure 17. When the Python file is created, all functions and settings described in the interface is then routed over and auto-coded. An example of the final output for this airfoil self noise case is shown in Figure 16.



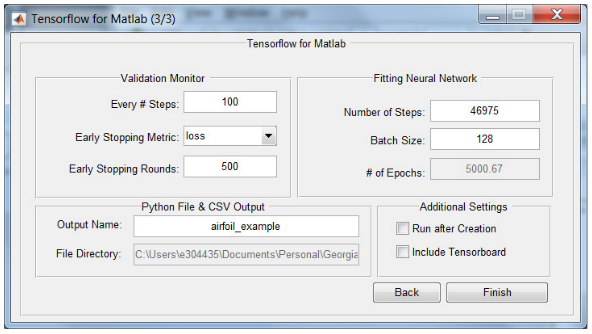
first_hidden_layer | fully_connected
second_hidden_layer | fully_connected
output_layer | fully_connected

Layer Name: first_hidden_layer

Hyperparameter	Value
type	fully_connected
num_outputs	25
activation_fn	tf.nn.relu

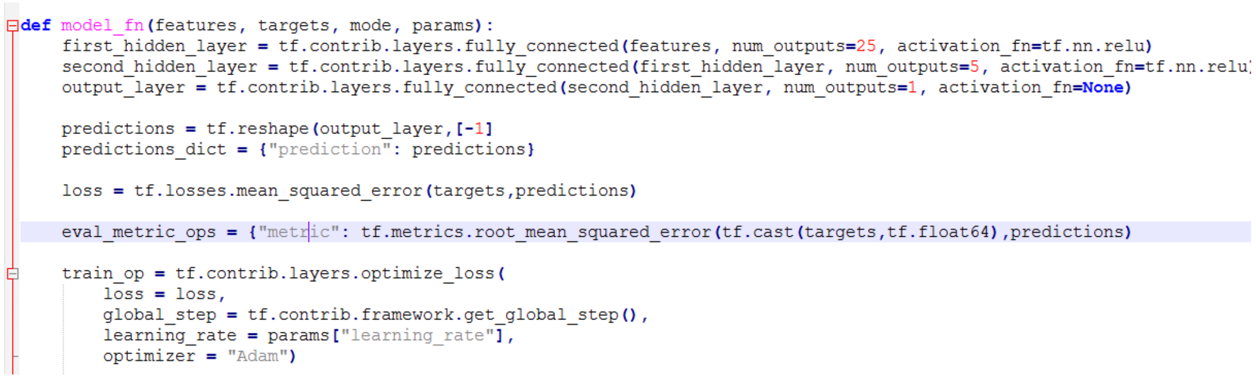
Calculate Loss: mean_squared_error
Optimizer: Adam
Evaluation Metric: root_mean_squared...
Learning Rate 0.001

Figure 15. Panel 2 of Airfoil Self-Noise. Snapshot of Tensorflow for Matlab GUI Interface Panel 2 inputs



Every # Steps: 100
Early Stopping Metric: loss
Early Stopping Rounds: 500
Number of Steps: 46975
Batch Size: 128
# of Epochs: 5000.67
Output Name: airfoil_example
File Directory: C:\Users\e304435\Documents\Personal\Georgia
☐ Run after Creation
☐ Include Tensorboard

Figure 17. Panel 3 of Airfoil Self-Noise. Snapshot of Tensorflow for Matlab GUI Interface Panel 3 inputs



```
def model_fn(features, targets, mode, params):
    first_hidden_layer = tf.contrib.layers.fully_connected(features, num_outputs=25, activation_fn=tf.nn.relu)
    second_hidden_layer = tf.contrib.layers.fully_connected(first_hidden_layer, num_outputs=5, activation_fn=tf.nn.relu
    output_layer = tf.contrib.layers.fully_connected(second_hidden_layer, num_outputs=1, activation_fn=None)

    predictions = tf.reshape(output_layer,[-1]
    predictions_dict = {"prediction": predictions}

    loss = tf.losses.mean_squared_error(targets,predictions)

    eval_metric_ops = {"metric": tf.metrics.root_mean_squared_error(tf.cast(targets,tf.float64),predictions)

    train_op = tf.contrib.layers.optimize_loss(
        loss = loss,
        global_step = tf.contrib.framework.get_global_step(),
        learning_rate = params["learning_rate"],
        optimizer = "Adam")
```

Figure 16. Output of Matlab for Tensorflow. Snapshot of Tensorflow for Matlab Auto-Coding Output

VII. Conclusion

This work aims to evaluate the performance of a set of four machine learning frameworks to see how they performed on four unique data sets over a CPU-based platform. The experimental results have shown that even with a similar neural network setup, some ML frameworks with their different solvers may actually present different results than their counterparts. This however also concludes that no one single framework is without its own intricacies and behaviors that are not easily comprehendible from the onset without much more detailed work on each of them. The selection of Tensorflow does imply that currently the best fit solution for entry level engineers is one that has the reliability, community support and efficient with the hardware resources available.

The development of an interface for any type of engineer conducting neural network research has shown that each part of the process can be generalized and auto-coded through the python API and quickly utilized for engineers to evaluate a neural networks performance against a specific data set. By leveraging the capabilities open source frameworks currently employ in their ties to the most up to date hardware configurations (GPUs), the community of data scientists and engineers will continue to thrive and grow a sustainable opportunity for more interfaces to be developed and utilized by all.

VIII. Additional Considerations and Future Work

As the production side of neural network development is heavily coveted in this study, it is also valuable to note that the deployment side is also supported by Tensorflow via an internally built module known as Tensorflow Serving (tfserve). The module itself lends itself to allow saved models be exported to a serialized Tensorflow model that can be coupled with a standard Tensorflow model server. In addition, one of the most highly coupled cloud services today, Amazon Web Services (AWS) also allows users to run their applications on a pre-built Ubuntu 14.04 image that contains Nvidia Drivers, cuDNN and Tensorflow in one complete solution.

The interface created was meant for Tensorflow specifically, but the architecture has been set-up and integrating several other ML Frameworks investigated in this study could be feasible for future work. In addition, further integrating Tensorflow into Matlab could include interfacing the outputs from the standard output of Tensorflow's logging feature into real-time plots displayed by Matlab for an interactive session the end-user can visualize how Tensorflow trains the network. Finally, other improvements can also allow the user to integrate BRAINN into the interface in order to conduct comparison tests between Tensorflow and BRAINN. All future work done on the interface will be maintained and updated in the Matlab file exchange link shown in the Appendix.

Appendix

A quick comparison of the syntax of each model set up for all frameworks is summarized in table 3. This example uses a simple 3 layer fully connected network with ReLU activation for syntax comparison.

Tensorflow	Theano

`layer1 = tf.contrib.layers.relu(features, 25)`
`layer2 = tf.contrib.layers.relu(layer1, 5)`
`output_layer =`
`tf.contrib.layers.linear(layer2, 1)`
`predictions = tf.reshape(output_layer, [-1])`
`predictions_dict = {"prediction":`
`predictions}`
`loss = tf.losses.mean_squared_error(targets,`
`predictions)`
`return model_fn_lib.ModelFnOps(loss=loss,`
`*params)`
	`input=layers.InputLayer(shape=(None, 5),`
	`input_var=input_var)`
	
	`layer1=layers.DenseLayer(input, num_units=25,`
	`nonlinearity=nonlinearities.rectify)`
	
	`layer2=layers.DenseLayer(`
	`layer1, num_units=5,`
	`nonlinearity=nonlinearities.rectify)`
	
	`output_layer = layers.DenseLayer(`
	`l_hid2, num_units=1,`
	`nonlinearity=None)`
	
	`net = NeuralNet(output_layer, *params)`

MXNet	Caffe
`outLabl = mx.sym.Variable('softmax_label')`	`net = caffe.NetSpec()`
`input = mx.symbol.Variable('data')`	`net.data,`
`flat = mx.symbol.Flatten(data=input)`	`net.label=cl.HDF5Data(batch_size=batch_size,`
`layer1 = mx.symbol.FullyConnected(data =`	`source=inputfile, ntop=2)`
`flat, name='fc1', num_hidden=25)`	`net.layer1=cl.InnerProduct(net.data,num_outp`
`act1 = mx.symbol.Activation(data = layer1,`	`ut=25,weight_filler=dict(type='xavier'))`
`name='relu1', act_type="relu")`	`net.relu1 =cl.ReLU(net.layer1,in_place=True)`
`layer2 = mx.symbol.FullyConnected(data =`	`net.layer2=cl.InnerProduct(net.relu1,num_out`
`act1, name='fc2', num_hidden=5)`	`put=5,weight_filler=dict(type='xavier'))`
`act2 = mx.symbol.Activation(data = layer2,`	`net.relu2=cl.ReLU(net.layer2,in_place=True)`
`name='relu2', act_type="relu")`	`net.output_layer =cl.InnerProduct(net.relu2,`
`output_layer = mx.symbol.FullyConnected(data`	`num_output=1,weight_filler=dict(type='xavier`
`= act2, name='fc3', num_hidden=1)`	`'))`
`net = mx.symbol.LinearRegressionOutput(data=`	`net.loss = cl.EuclideanLoss(net.outer_layer,`
`output_layer, label=outLabl, name='linreg1')`	`net.label)`

A Github repository of all code generated for this project can be found at: github.com/projectlouis/mlbenchmark

The Matlab for Tensorflow GUI can be found on the Mathworks Matlab Central File Exchange at: https://www.mathworks.com/matlabcentral/fileexchange/62808-tensorflow-for-matlab.

Acknowledgments

The author would like to thank Dimitri Mavris and Burak Bagdatli for their support of this special problems project.

References

[1]Shi, S., Xu, P., Wang, Q., and Chu, X., "Benchmarking State-of-the-Art Deep Learning Software Tools," Department of Computer Science, Hong Kong Baptist University, Jan. 2017.

[2]Bahrampour, S., Ramakrishnan, N., and Schott, M. S. Lukas, "Comparative Study of Deep Learning Frameworks," Research and Technology Center, Robert Bosch LLC, Mar. 2016.

[3]Vanhoucke, V., Senior, A., and Mao, M. Z., "Improving the speed of neural networks on CPUs," Google Inc., 2011.

[4]NVIDIA, "GPU-Based Deep Learning Inference: A Performance and Power Analysis," NVIDIA Whitepaper, 2015.

[5]Lecun, Y., Cortes, C. and Burges, C. J.C., "MNIST Database," *Special Database* [http://yann.lecun.com/exdb/mnist/], NYU/Google, New York, NY, 1998.

[6]Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science

[7]Brooks, T.F., Pope, D.S., and Marcolini, A.M. "Airfoil Self-Noise Data Set," [https://archive.ics.uci.edu/ml/datasets/Airfoil+Self-Noise], NASA, NASA RP-1218, July 1989.

[8]Coraddu, A., Oneto, L., Ghio, S., Savio, S., Anguita, D., and Figari, M. "Machine Learning Approaches for Improving Condition Based Maintenance of Naval Propulsion Plants," Journal of Engineering for the Maritime Enviornment [https://archive.ics.uci.edu/ml/datasets/Condition+Based+Maintenance+of+Naval+Propulsion+Plants], (In Press), 2014.

[9]Habrail, G., and Bacrard, A., "Individual Household Electric Power Consumption," EDF R&D [https://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption], Clamart, France, 2012.

[10]TF, Tensorflow, Software Package, Ver. 1.0, Google Brain Team, Mountain View, CA, 2017.

[11]Theano, Software Package, Ver. 0.8.2, Universite de Montreal, Quebec, Canada, 2016.

[12]MXNet, Software Package, Ver. 0.9.3, University of Washington, Seattle, Washington, 2016.

[13]Caffe, Software Package, Ver. 1.0.0.rc3, Berkeley Vision and Learning Center, Berekley, CA, 2017

[14]Google, "AlphaGo," https://deepmind.com/research/alphago/, Mountain View, CA, 2016.

[15]Baidu, "Deep Speech 2," http://usa.baidu.com/tag/deep-speech/, Beijing, China, 2016.

[16]Lecun, Y., Bottou, L., Haffner, P., "Gradient-based learning applied to document recognition," Proceedings of the IEEE, November 1998.

[17]Johnson, C. and Schutte, J., "Basic Regression Analysis for Integrated Neural Networks (BRAINN) Documentation," Georgia Institute of Technology (2011).

[18]Kingma, D. and Ba, J., "ADAM: A Method for Stochastic Optimization," Published at ICLR 2015. University of Amsterdam, OpenAI. (2015).

[19]Besson, C., "Autonomous and Adaptive Neural Network Surrogate Modeling Technique," Georgia Institute of Technology. (Fall 2014).

[20]Perner, D., "Investigation into the Impacts of Additional Hidden Layers on Neural Network Performance," Georgia Institute of Technology. (Summer 2009).

[21]Akinlir, C., "Neural Net-Enabled Optimization for Gravity-Based Spacecraft Formation Control," Georgia Institute of Technology. (Fall 2008).