

---

# A DETAILED COMPARATIVE STUDY OF OPEN SOURCE DEEP LEARNING FRAMEWORKS

---

A PREPRINT

**Ghadeer Al-Bdour**

Faculty of Computer and Information Technology  
Jordan University of Science and Technology  
Irbid, Jordan  
gwalbdour13@cit.just.edu.jo

**Raffi Al-Qurran**

Faculty of Computer and Information Technology  
Jordan University of Science and Technology  
Irbid, Jordan  
rlalqurran11@cit.just.edu.jo

**Mahmoud Al-Ayyoub**

Faculty of Computer and Information Technology  
Jordan University of Science and Technology  
Irbid, Jordan  
maalshbool@just.edu.jo

**Ali Shatnawi**

Faculty of Computer and Information Technology  
Jordan University of Science and Technology  
Irbid, Jordan  
ali@just.edu.jo

March 4, 2019

## ABSTRACT

Deep Learning (DL) is one of the hottest trends in machine learning as DL approaches produced results superior to the state-of-the-art in problematic areas such as image processing and natural language processing (NLP). To foster the growth of DL, several open source frameworks appeared providing implementations of the most common DL algorithms. These frameworks vary in the algorithms they support and in the quality of their implementations. The purpose of this work is to provide a qualitative and quantitative comparison among three of the most popular and most comprehensive DL frameworks (namely Google's TensorFlow, University of Montreal's Theano and Microsoft's CNTK). The ultimate goal of this work is to help end users make an informed decision about the best DL framework that suits their needs and resources. To ensure that our study is as comprehensive as possible, we conduct several experiments using multiple benchmark datasets from different fields (image processing, NLP, etc.) and measure the performance of the frameworks' implementations of different DL algorithms. For most of our experiments, we find out that CNTK's implementations are superior to the other ones under consideration.

**Keywords** TensorFlow · Theano · CNTK · Performance Comparison

## 1 Introduction

Deep Learning (DL) is the hottest trend in Machine Learning (ML). The idea of DL is to train a multi-layer Artificial Neural Network (ANN) on a dataset in order to allow it to handle real world tasks. Although the theoretical concepts behind DL are not new, DL has enjoyed a surge of interest over the past decade due to many factors including its successful application to several problems (many of which have great commercial potentials) and the improved affordability of the required computing infrastructure.

DL approaches have significantly outperformed state-of-the-art approaches in many classical problems of many fields such as image processing, computer vision, speech processing, natural language processing (NLP), etc. Moreover, the scientific community (from both the academia and the industry) has quickly and massively adopted DL. Open source implementations of successful DL algorithms quickly appeared on code sharing websites, and were subsequently used by many researchers in different fields.

Several DL frameworks exist such as TensorFlow, Theano, CNTK, Caffe, Torch, Neon, pylearn, etc. Each one of these frameworks has different features and performance characteristics. Further, each framework utilizes different techniques to optimize its implementation of DL algorithms. Although the same algorithm is implemented in different frameworks, the performance of the different implementations can vary greatly. A researcher/implementer willing to employ such an algorithm in his research or application faces a difficult choice, since the number of different implementations is high and the effort invested by the research community in scientifically comparing these implementations is limited.

In this work, we aim at providing qualitative and quantitative comparison between popular open source DL frameworks. To be more specific, we focus on three very popular DL frameworks, namely Theano (from LISA Lab, University of Montreal), TensorFlow (from Google), and CNTK (from Microsoft). These frameworks support multi-core CPUs as well as multiple GPUs. All of them import cuDNN, which is a DL library from NVIDIA that supports highly tuned implementations for standard routines such as forward and backward convolution, normalization, pooling, and activation layers. We compare these frameworks by training different neural network (NN) architectures on five different standard benchmark datasets for various tasks in image processing, computer vision and NLP. In an earlier version of this work [1], we only considered two datasets. Despite their importance, comparative studies like ours that focus on performance issues are rare. A comparative study of the frameworks is important in order to enable people who are interested in applying DL in their research and/or applications to make informed decisions about which of the existing frameworks suits their needs.

The rest of this paper is organized as follows. The following section discusses the existing efforts on comparing DL frameworks highlighting what distinguishes our work. Section 2 presents our survey of the literature, while Section 3 explains in details what NN are. Section 4 discusses the frameworks, the way they were used to train the datasets and a brief comparison between them. The methodology we follow is discussed in Section 5. Experimental results and the discussion are detailed in Sections 5 and 7. The work is concluded with final thoughts presented in Section 8.

## 2 Literature Survey

Only few researchers did a comparative study between state-of-the-art DL frameworks running on different hardware platforms (CPU and GPU) to highlight the advantages and limitations for each framework when applied on different deep NN architectures, and to enable developers to optimize the running performance of DL frameworks.

One of the first and most prominent examples of comparative studies between DL frameworks was carried out by Bahrampour et al. [2]. The authors compared five DL frameworks: Caffe, Neon, TensorFlow, Theano, and Torch, in terms of speed (gradient computation time and forward time), hardware utilization and extensibility (ability to support different types of DL architectures) after applying various convolutional algorithms on the aforementioned frameworks. They conducted their experiments on a single machine for both CPU (multithreaded) and GPU (NVIDIA Titan X) environments. The comparison between frameworks was carried out by training convolutional and stacked autoencoder (AE) networks on the MNIST and ImageNet datasets. They also trained long short-term memory (LSTM) networks [3] on the IMDB dataset [4].<sup>1</sup> The authors reported several observations/findings. In terms of extensibility, they found that Theano and Torch were the best as they can support various DL architectures and libraries. Also, they found that TensorFlow was a very flexible framework especially when used in different parts of the computational graph. Finally, emphasizing ease of use, they noticed that Caffe was the easiest. In terms of performance, they noticed that Torch was the best for training and testing their DL architectures on a CPU platform. Theano came in second while Neon gave the worst performance. On a GPU platform, for convolutional and fully connected networks, they found that Torch was the best followed by Theano. Also, they noticed that Theano was the fastest on small networks and Torch was the fastest on large networks followed by Neon. For recurrent networks (LSTM), they found that Theano's results were the best in terms of performance. TensorFlow on single GPU was the worst compared to other studied frameworks.

Shi et al. [5] did a comparative study between several DL frameworks including Caffe, MXNet, CNTK, TensorFlow, and Torch. They considered three types of NN including; fully connected NN (FCN), convolutional NN (CNN) and recurrent NN (RNN). Moreover, they used different hardware environments including two CPU platforms and three GPU platforms. They considered the running time and the convergence rate as the metrics to evaluate the selected frameworks. They used synthetic datasets to measure running time performance and real-world datasets to measure the convergence rate in their experiments. The results were as follows. In synthetic datasets, they evaluated the performance of FCN using a large NN (FCN-s). They used AlexNet and ResNet-50 on ImageNet dataset. For real-world datasets, they applied MNIST dataset using a small FCN (FCN-R). Also, they applied CIFAR10 dataset using AlexNet-R and ResNet-56. For RNN, they chose two LSTM layers for testing. After experimentation, they found that all tested frameworks achieved significant speedup using GPU over CPU. For CPU platform, they found that TensorFlow was the best compared to other tools. On a single GPU, Caffe, CNTK and Torch performed better than MXNet and

---

<sup>1</sup>More details will be given later about these neural network architectures and these datasets.

Table 1: Goldsborough’s qualitative comparison of DL frameworks

Library	Frontends	Style	Gradients	Distributed Execution
TensorFlow	Python,C++	Declarative	Symbolic	✓
Theano	Python	Declarative	Symbolic	X
Torch	LuaJIT	Imperative	Explicit	X
Caffe	Protobuf	Imperative	Explicit	X

TensorFlow on FCN implementations. For small CNN, Caffe and CNTK achieved good performances. For RNN (LSTM), CNTK was the fastest as it was five to ten times better than the other tools. Finally, on multi-GPU platforms, all implementations had higher throughput and convergence rate.

Goldsborough [6] showed the timeline of ML software libraries for DL. He focused on TensorFlow’s results and its basic properties including computational paradigms, its distributed model and programming interface. He compared TensorFlow to other DL frameworks including Theano, Torch and Caffe, qualitatively and quantitatively. In qualitative terms, he compared aforementioned frameworks using several categories including frontends, programming model style, how gradients are computed, and distributing the execution of computational graph. Table 1 shows a summary of this comparison. In quantitative terms, he reviewed works (such as [2, 7, 8]), which contain comparisons between TensorFlow and other DL frameworks. From LeNet benchmark in [9], he noted that TensorFlow was ranked second after Torch in forward and backward measures, but, in terms of performance, TensorFlow came at the last rank compared to tested frameworks. From the results on [10]’s benchmark for convolutional network models, he noted that TensorFlow came at the second place behind Torch in forward and backward propagation time. Finally, in [11], the benchmarks were CNN models including AlexNet architecture, and an LSTM network operating on the Penn TreeBank dataset [12]. He noted that TensorFlow was the best framework for small model followed by Theano then Torch. For large models, TensorFlow came at the second rank after Theano, and Torch came at the last place.

Chintala [7] applied different ImageNet benchmarks for a variety convolutional network types including AlexNet, GoogleNet, Overfeat, and OxfordNet using different open source DL frameworks such as Caffe, Theano, Torch, TensorFlow, Chainer, etc. He conducted his experiments on NVIDIA Titan X GPU and two new packages for Fast Fourier Transform (FFT) computation [13]. The first one is based on the NVIDIA library (cuFFT) and another on a Facebook FFT (fbfft). Moreover, he used native version of each DL frameworks. After experimentation, he found that using fbfft resulted in a speedup over cuFFT for all applied CNNs. Also, he found that the fastest framework for CNN was Torch followed by Tensorflow.

A Theano development team of Al-Rfou et al. [8] discussed the Theano framework, its features, how to use it, and showed recent improvements on it. They did a performance comparison between Theano and other frameworks including Torch and TensorFlow on three types of ML models including CNN, RNN and sequence-to-sequence mapping RNN. Finally, they showed the computation speed using multiple GPUs. The results were as follows. On a single GPU platform, for CNN, they found the processing time for four different convolutional models (AlexNet [14], OverFeat [15], VGG [16], and GoogLeNet [17]) on ImageNet dataset. They reported results for each framework per minibatch for forward and backward pass. They found that Theano was slower than Torch and TensorFlow. However, in overall performance-wise, they were close to each other. For RNN, they used LSTM on Penn TreeBank dataset [12] and reported the results on small, medium and large LSTM models. They found that Theano was in the second place after TensorFlow for the small model, but Theano was the fastest for the medium and large models. They also showed that Torch was slower than Theano on all tested models. Finally, for sequence-to-sequence model [18], the input was video frames and the output was the English sentence describing the input. The input video frame was preprocessed by a GoogLeNet (pre-trained for classification on ImageNet). They compared Theano to TensorFlow and excluded Torch because there was no available implementation in Torch. They reported the processing time per minibatch using three different batch sizes (32, 64 and 128). They found that Theano was the fastest on smaller batches. However, on large ones, TensorFlow was the superior one. They repeated the previous RNN model (LSTM) on multi-GPU platforms (2-GPUs and 4-GPUs) using platoon. Their measured processing speed when synchronizing after each batch was found to provide speedups between 1.6X and 1.7X for 2-GPUs and 3.2X for 4-GPUs. When synchronizing after every 100 batches, they found a 2X speedup for 2-GPUs and 3.9X-4X speedup for 4-GPUs.

Kovalev et al. [19] presented a comparative study between five DL frameworks: Theano, Torch, Caffe, TensorFlow, and DeepLearning4J, in terms of training and prediction speed and classification accuracy. They used MNIST dataset of handwritten digits for testing five FCN frameworks. Their computation experiments were applied only on CPU; they reported out the results for two kinds of scaling factors applied on FCN networks including changing the network’s *depth* (number of internal layers) and changing the network’s *width* (number of neurons). Also, they tested the NN with two different activation functions, Tanh and ReLU. In Tanh nonlinearity function, they found that the training time

is approximately 30 seconds when they changed the number of layers from one to four in all frameworks except for DeepLearning4J. For DeepLearning4J, the training time started from 140 seconds for one layer and grew up to 210 seconds when they used four layers. For the prediction time, they found that Theano, Torch, Caffe, and TensorFlow consumed less than 0.4 second. However, for DeepLearning4J, it started from 0.75 second when using one layer and increased up to 1.1 second for four layers. In terms of classification accuracy, they found that Theano, DeepLearning4J, and Caffe achieved high accuracy starting from 94% for one layer and going up to 98% for four layers. For Torch and TensorFlow, the accuracy dropped with increasing the number of network layers. With ReLU nonlinearity function, the training time was much lower compared with the case of Tanh function. As for the prediction time, they observed that the use of ReLU gave results that were similar to Tanh results. In terms of accuracy, they found that Torch's accuracy grew up while increasing the number of layers, whereas other frameworks had the same behavior as with Tanh. Finally, they changed the number of neurons in the hidden layers of networks for ReLU function only, and reported the speed and accuracy values. They found that DeepLearning4J framework was the slowest in training and prediction times, and time consumed increased with the increase in the number of neurons. The final classification accuracy when changing the internal layer sizes from 64 to 1024 neurons remained around 97% for Theano, Caffe, TensorFlow, and DeepLearning4J. However, in the case of Torch, the classification accuracy grew with the growing layer size (started from 70% and reached 98%).

Bastien et al.[20] suggested a new feature to be added to the main features of Theano in order to improve its performance in different benchmarks. They conducted a comparative study (in terms of features and performance) between Theano and Torch7 on NN benchmarks, and between Theano and RNNLM on RNN benchmarks. In their comparison, they used three learning architectures: *logistic regression*, *NN with one hidden layer* (500 units) and *DNN with three hidden layers* (1000 hidden units each). They found that when applying NN on CPU using one hidden layer models without using mini-batches, Theano's results overcome Torch7. However, on the logistic regression benchmark, Torch7 thrived because, at each call, it decreased the amount of performed computation. On the GPU, with batch size equal to one, Torch7 overcame Theano. When using mini-batches, Theano was faster than (or has an equivalent speed to) Torch7 on all three learning architectures under consideration. When they applied RNN on Theano and RNNLM with batch size of one, they found that RNNLM was faster than Theano on smaller models. However, for bigger sizes, Theano was faster.

Ding et al. [21] introduced an implementation of Theano-based AlexNet on ImageNet dataset and applied it on multiple GPUs in order to accelerate the training process. They compared their results of Theano to Caffe library which runs on single GPU in terms of training time. On a single GPU with batch size equal to 256, Caffe showed to be faster than Theano. However, on 2-GPUs with batch size equal to 128, they found that Theano was faster than Caffe on 1-GPU.

In previous studies, the comparison goal focused only on processing time. None of those comparative studies dealt with CPU and GPU utilization or memory consumption. This work covered these metrics to find which of the considered frameworks achieve the best performance. Finally and most importantly, the comparisons involved more datasets from more fields compared with previous studies.

### 3 Neural Networks

We start this section with a glimpse of history related to neural networks.

#### 3.1 Single Layer Neural Network

A single layer NN is a network that consists of a single hidden layer between the input layer and the output layer. The hidden layer has many units called neurons. See Figure 1.

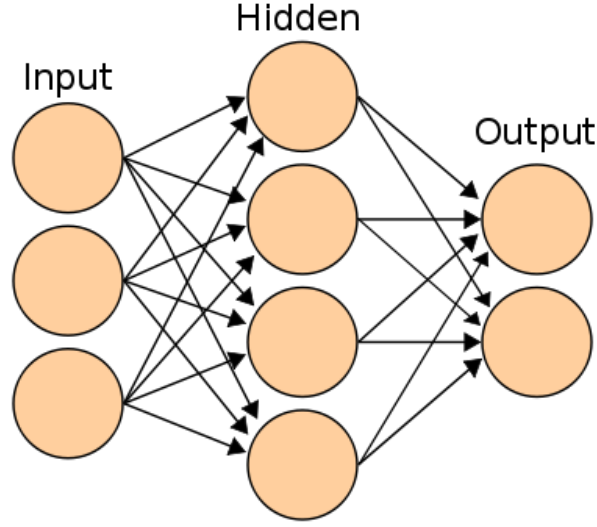
##### 3.1.1 Artificial Neurons

A neuron is one of the most important parts of human brain, where a bunch of neurons build what is called a neural network. This network with its tiny components processes and transmits signals in a form of electrical and chemical pulses. These signals are passed through the neurons via synapses. Same theory applies to artificial neurons where these units when combined with each other perform as a single impenetrable network which takes input signals and performs complex calculations to produce output. This process is depicted in Equation 1, where  $w_i$  represents connection weights and  $x_i$  represents the input values. The left side of the equation  $Y$  represents the outputs.

$$Y = f\left(\sum_{i=0}^n w_i x_i\right) \quad (1)$$

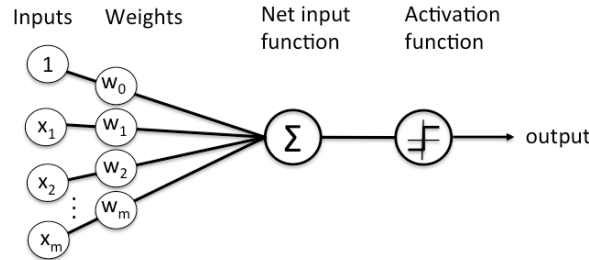
---

<sup>2</sup>[https://en.wikibooks.org/wiki/Artificial\\_Neural\\_Networks/Neural\\_Network\\_Basics](https://en.wikibooks.org/wiki/Artificial_Neural_Networks/Neural_Network_Basics)

Figure 1: Single layer Neural Network <sup>2</sup>

### 3.1.2 The Perceptron

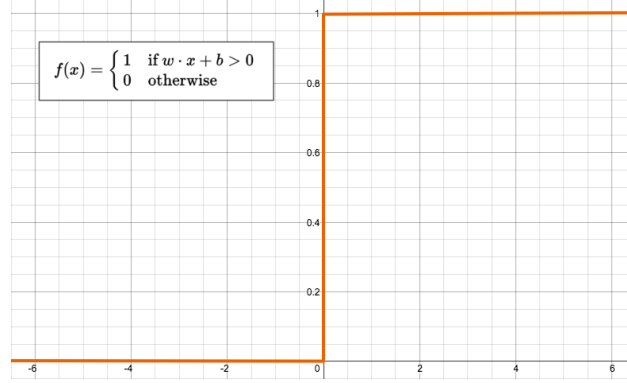
The perceptron algorithm was devised in 1957 by Rosenblatt [22] for image recognition tasks. A perceptron is the basic form of a NN as it does not contain any hidden layer and the input signals are fed directly to output neurons using a series of weights as shown in Figure 2. The perceptron contains only an input layer and one output layer that consists of one or more artificial neurons.

Figure 2: Rosenblatt's perceptron<sup>3</sup>

A perceptron is considered a binary classifier as it has only two possible results 1 or 0, which is determined by computing a single output from multiple input values. This is done by evaluating a linear combination of input values and their weights and then putting the output through some nonlinear activation function like the Heaviside step function (as Threshold Function) [23], as shown in Figure 3. The output neuron in the output layer connects to all inputs to produce one output value.

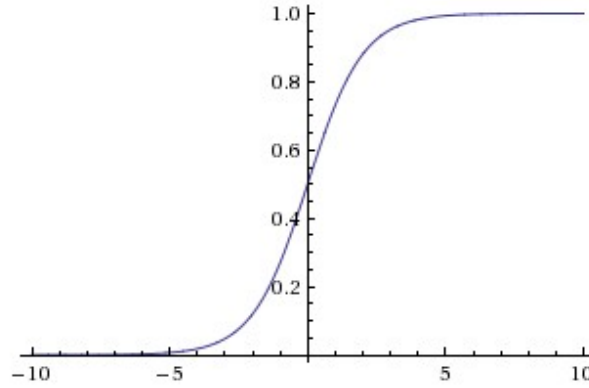
In spite of the successes of the complex architectures of DL, the perceptron is still used because of the simplicity of its architecture and its “light” computation requirement allowing its efficient use with very large datasets. The main drawback of the perceptron is that it only learns linearly-separable functions. In order to solve this problem, a multilayer perceptron (also known as Deep Neural Network (DNN)) was suggested by Ivakhnenko et al. [24] to get more powerful learning mechanisms.

<sup>3</sup><https://goo.gl/h9cAJm>

Figure 3: Perceptron's activation function<sup>4</sup>

### 3.1.3 Activation Function

An activation function is a nonlinear function that is used in different types of NN. It takes weighted input data and transforms them into a nonlinear output by performing some mathematical operations on them such as matrix multiplication between inputs and weights. Regarding DL implementations, nonlinear activation functions create complex features with every layer. Implementations with a linear activation function would behave like a single-layer network (no matter how many hidden layers) because summing these layers would give just another linear function. This is the reason why nonlinear activation functions are used more widely at DL networks. However, it is possible that some NN may contain neurons with linear activation function in the output layer. These neurons require a nonlinear activation function in previous parts of the network. There are several types of activation functions including the sigmoid function shown in Figure 4.

Figure 4: Sigmoid function<sup>5</sup>

The sigmoid function has the following mathematical form.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (2)$$

It takes a real-value number and converts it into values in the range [0, 1] (large negative numbers convert to 0 and large positive numbers convert to 1). Its main drawbacks are the vanishing gradient problem and the fact that its output is not zero-centered.

Another activation function called *Tanh* is shown in Figure 5.

$$\text{Tanh}(x) = 2\sigma(2x) - 1 \quad (3)$$

<sup>4</sup> <https://appliedgo.net/perceptron/>

<sup>5</sup> <http://cs231n.github.io/neural-networks-1/>

It is a nonlinear activation function that takes a real-value number and converts it into the range  $[-1, 1]$ . It causes the vanishing gradient like sigmoid, but its output is zero-centered which enables the Tanh nonlinearity to be used more widely than the sigmoid function.

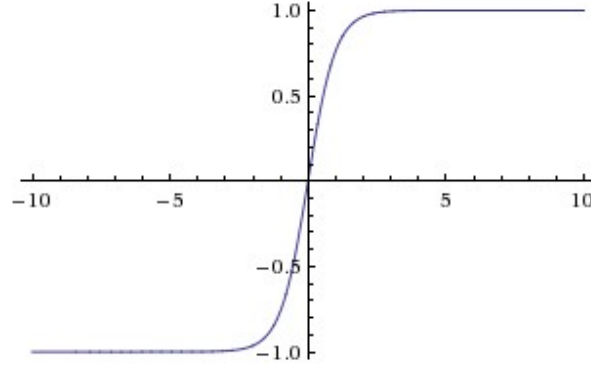


Figure 5: Tanh function <sup>6</sup>

The *Rectified Linear Unit (ReLU)* activation function (Figure 6) has formula as in the following equation.

$$f(x) = \max(0, x) \quad (4)$$

It was found to be more accelerated than the Tanh and sigmoid functions due to its linear and non-saturating form. Moreover, it can be implemented in a less expensive way compared with Tanh and sigmoid. Unfortunately, ReLU unit may “die” during training where it outputs zero value for any given input. This happens when the input to its units are negative or after a large negative bias value is inputted for its weights (gradients will be zero). ReLUs cannot recover from this problem because they will not modify the weights (block backpropagation).

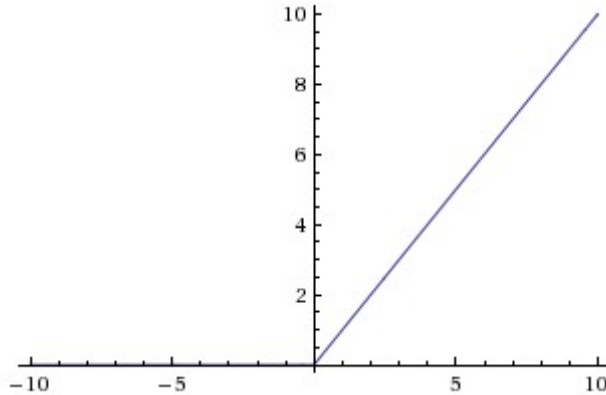


Figure 6: ReLU function <sup>6</sup>

The *Maxout* [25] is a nonlinear activation function that applies dot product between the weights and data, and its output is the maximum of a set of inputs. The Maxout neuron uses the following formula.

$$\max(w_1^T x + b_1, w_2^T x + b_2) \quad (5)$$

It is to be noted that the ReLU function is a special case of the Maxout function. The Maxout neuron has all the benefits of a ReLU unit and does not have its drawbacks (dying unit), which makes Maxout one of the most common activation functions used in DL networks.

Another form of activation functions is called the Logistic Regression. It is a regression model developed by Cox in 1958 [26] that estimates the relationship between statistical input variables to make prediction of an output variable.

<sup>6</sup><http://cs231n.github.io/neural-networks-1/>

It uses the logistic sigmoid function to generate a prediction as to which of multiple classes the input data belongs. Logistic regression is used for different areas like medical and social sciences where it is used for analytical purposes and interpretation of results from experiments. It is used for very large datasets because of its simplicity and speed. The final layer at a DL algorithm can be constructed using a logistic regression, where the network has multiple feature learning layers that pass features into a logistic regression layer to classify inputs.

Logistic regression can be binomial, ordinal or multinomial. In the binomial type, the observed output for a dependent variable has two possibilities, either zero or one. In multinomial logistic regression, the output can have more than two types (e.g., “Disease A” vs “Disease B” vs “Disease C”) which are not ordered. In ordinal type, the dependent variables must be ordered.

In linear regression algorithms (using least square), Gradient Descent (GD) is an optimization algorithm that is used to find the values of parameters in a way that minimizes the cost function and the least square error. If a huge dataset is trained using GD, calculating the parameters will be expensive and take long time. If there are millions of sample points, every iteration, GD must go through these points to calculate the parameters. To solve these problems, a variation of GD called Stochastic GD (SGD) is used. SGD is an optimization method used to train models including support vector machines (SVM), logistic regression, graphical models, etc. To calculate the parameters in SGD, a sample of training set or one training value is used instead of using the entire sample in every iteration. This method is much faster and less costly than GD.

Backpropagation of errors is a learning method that is used to train NN. It is used along with an optimization algorithm such as GD. The modern version of backpropagation was proposed by Linnainmaa [27], where he published the general method for automatic differentiation (AD) of discrete connected networks of nested differentiable functions. Backpropagation repeatedly performs two steps for training the network; propagation and weights update. At the first step, the input vector is forward propagated layer-by-layer until it reaches the output layer and produces the output of this vector. For each of the neurons in the output layer, an error value is calculated using a loss function, which compares the output of the network to the desired output. It then calculates the gradient of the loss function with respect to the weights. Backpropagation sends these error values backwards starting from the output layer until it reaches the first layer. At the second step, this gradient value is fed to an optimization method (e.g., GD) to update the weights in order to minimize the loss function. Backpropagation is considered as a supervised learning method, because it requires the knowledge of the desired output to calculate the loss function gradient, but some unsupervised networks such as autoencoders (AE) can use it.

The rapid improvement in DL methods makes the training of any DL network a complex and time consuming process. To address these issues, many software tools (frameworks) have appeared to develop these methods in an easy and efficient manner.

### 3.2 Multilayer Neural Networks

A multilayer NN is a network consisting of more than one hidden layer between the input layer and the output layer, as shown in Figure 7.

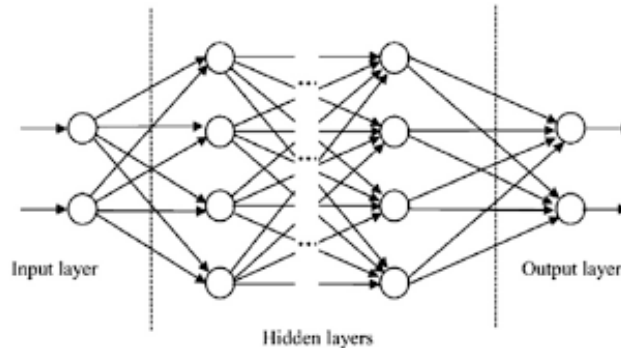


Figure 7: Multilayer Neural Network<sup>7</sup>

One popular example is the Multilayer Perceptron (MLP) network, which consists of an input layer, one or more hidden layers of computation neurons, and an output layer. MLP can learn linear and nonlinear functions in contrast to the single layer perceptron that only supports linear functions. MLP has a large number of features. Also, it uses

<sup>7</sup>[https://www.researchgate.net/figure/257194238\\_fig2](https://www.researchgate.net/figure/257194238_fig2)



backpropagation technique for training the network. Each node in its hidden layers is a neuron that applies a nonlinear activation function. The input values are passed from the input nodes to the first hidden layer, which applies some calculations to them using the activation function. The resulting signals are then passed as input signals to the next hidden layer. This procedure is repeated until the signals reach the output layer.

### 3.3 Deep Learning (DL)

The wide adoption of Deep Neural Networks (DNN) gave rise to new field of *Deep Learning* (DL). The DNN is simply a NN with more than one hidden layer of nonlinear processing units which extract the features by passing input data from a layer to another until a desirable output is produced. One of the most used algorithms in DL is the Backpropagation algorithm, which is used to train the network by updating the weights of the connections between sequential hidden layers. This process is repeated many times until the output matches the desired output.

There are several types of DL architectures such as DNN, Convolutional DNN (CDNN), Deep Belief Networks (DBN) and Recurrent NN (RNN). Approaches based on these architectures have been achieving significant performance improvements when applied to several tasks in speech recognition, computer vision, natural language processing (NLP), etc.

More than half a century ago, Ivakhnenko et al. [24] introduced deep MLP (Figure 8), where thin but deep models (three hidden layers) with polynomial activation functions were used. The authors used statistical methods to select the best features in each layer, and forwarded these features to the next layer until the output layer is reached. Finally, they used layer-by-layer backpropagation algorithm to train the network. A deeper network with eight layers was introduced in [28] which was trained using the Group Method of Data Handling (GMDH) algorithm. In 1980, a network with multiple convolutional and pooling layers was introduced in [29], where it was trained using a reinforcement learning. The challenge for this model was the training of the multiple layers. At that time, backpropagation of errors was an inefficient and incomplete form to train such deep models. LeCun in 1989 gave the first efficient and practical application of backpropagation at Bell Labs [30]. He applied backpropagation to a deep convolutional network in order to classify the handwritten digits of the MNIST dataset. This approach achieved good results. Unfortunately, it consumed a lot of time which made it useless for many years later. In 1993, recurrent neural networks (RNNs) were introduced to solve the time consumption problem. RNNs learned by unsupervised learning, which was implemented and used with very deep learning tasks (more than 1,000 subsequent layers) [3]. After that, a DL method called the long short-term memory (LSTM) for RNN was proposed by Hochreiter and Schmidhuber in 1997 [3], where it was used in the deep learning tasks that require memories of events (like speech). Also, it avoided the vanishing gradient problem at which no learning signals reached to early layers in the network during training the deep network.

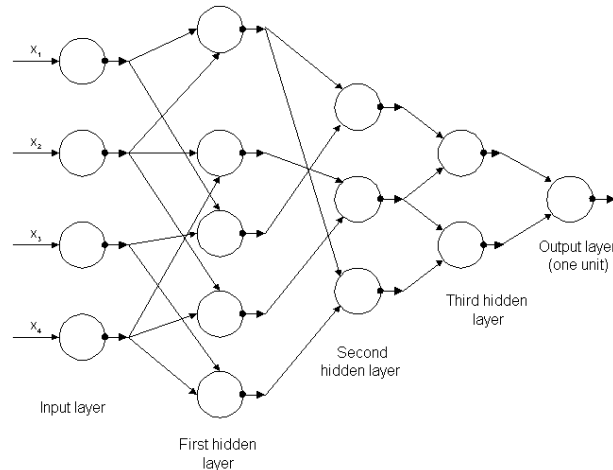


Figure 8: The architecture of the first known deep network by Ivakhnenko<sup>8</sup>

A big shift in the field of DL occurred when more people started to use the graphics processing units (GPUs) in the training process. This increased the computational speed allowing NN to produce better results by using more training data. However, training using huge amounts of data brought back to light the vanishing gradient problem. To solve this problem, the model was learned in a layer-by-layer fashion using unsupervised learning. This requires the features

<sup>8</sup><https://goo.gl/vcqNjP>

of early layers to be initialized with suitable features beforehand (pre-trained). However, in supervised learning, the features at early layers need to be adjusted during learning process. One solution, which is the pre-training solution, was initially developed for RNN in 1992 [31] and for feedforward networks in 2006 [10]. A second solution for the vanishing gradient problem in RNN was the LSTM [3].

In the year 2011, the rapid increase in speed of GPUs reached its glory which led many researchers such as Ciresan et al. to train deep networks without using pre-training techniques and started to introduce deep learning networks that were constructed from convolutional layers, max-pooling layers, and several fully connected layers followed by a final classification layer [32, 11]. Moreover, Krizhevsky et al. used a similar architecture with rectified linear activation functions and dropout function [33]. Since then, the research of DL field using GPUs has accelerated rapidly.

### 3.4 Convolutional Neural Networks (CNN)

Convolution, which is widely used in DL networks, is a mathematical process used to mix input data function,  $g$ , with convolutional kernel (filter),  $f$ , in order to produce a transformed feature map (a modified version of the original input) as shown in the following equation.

$$(f \times g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau = \int_{-\infty}^{\infty} f(t - \tau)g(\tau)d\tau \quad (6)$$

Convolution has different applications in many fields such as probability, statistics, computer vision, imaging, etc. In probability theory, convolution is similar to cross-correlation, while in statistics, it updates the weights over normalization of input vector. Figure 9 shows a Convolutional Neural Network (CNN or ConvNet), depicted in Figure 9, is a network with multiple layers of convolutions that apply a nonlinear activation function like ReLU or Tanh (mostly used for image processing). CNN layers filter input data to produce useful feature map information. These layers have parameters that are updated repeatedly to produce the desired output. In feedforward NN with fully connected layers, each input neuron is connected to all neurons of the next layer. However, with CNN, after calculation of the output by applying convolutions on the input data, each node only connects itself with the closest neighboring neurons (local connectivity). Convolutional layers sizes shrink as they become deeper.

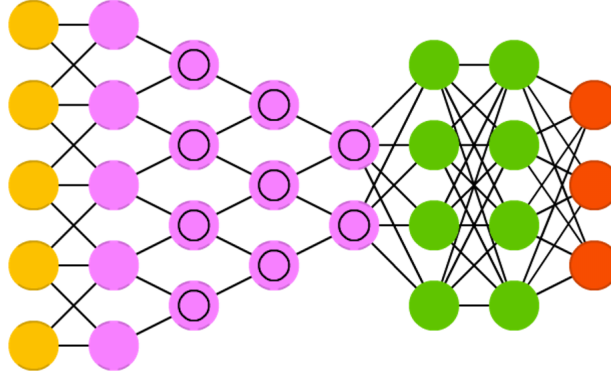


Figure 9: Convolutional Neural Network <sup>9</sup>

The architecture of CNN primarily has three types of layers including; *convolutional layers*, *pooling layers*, and *fully connected layers*. The convolutional layers are the main block of a CNN that do most of the computational operations. The pooling layers are applied after the convolutional layers, where these layers partition the input data into non-overlapping sets (windows), and reduce each set to a single value (subsampling) by applying a max operation (outputs the maximum value in each set) to the result of each filter. The pooling layers have benefits of reducing the spatial size of data, reducing the number of parameters, reducing computations, and controlling overfitting. Finally, after all generated features are combined, they are used to find the final classification via fully connected layers, whose neurons are fully connected to all activations in the previous layer.

Many CNN architectures exist such as:

- LeNet: Developed by LeCun et al. in 1990's [34], LeNet was used to read zip codes, recognize characters, etc.

<sup>9</sup><http://www.asimovinstitute.org/neural-network-zoo/>

- AlexNet: Developed by Krizhevsky et al. [33], AlexNet was used in computer vision tasks. It has a similar architecture to LeNet, but with deeper, bigger, and more stacked convolutional layers on top of each other.
- GoogleNet: Introduced by Szegedy et al. from Google [17], GoogleNet had a reduced number of parameters in the network (it had 4M parameters, compared to AlexNet’s 60M).
- VGGNet: Introduced by Simonyan and Zisserman [16], the goal of VGGNet’s architecture was to prove that a good performance depends on the depth of the network.
- ResNet (Residual Network): Introduced by He et al. [35], ResNet’s architecture does not have fully connected layers at the end of the network. Also, it uses batch normalization.

### 3.5 Recurrent Neural Networks (RNN)

RNN is a type of ANN, first introduced by Elman in 1990 [36]. After that, Elman and others began to develop the concept of RNN. In 1993, a modified RNN model was developed to solve very deep learning tasks which required more than 100 subsequent layers. RNN connections between neurons form a directed cycle (fed data from previous layer and from themselves) as shown in Figure 10. This makes it available to be used for sequential information. Because RNN is built in a way that fits sequential information, it is used in many tasks in NLP, speech recognition, image capturing, language translation, etc.

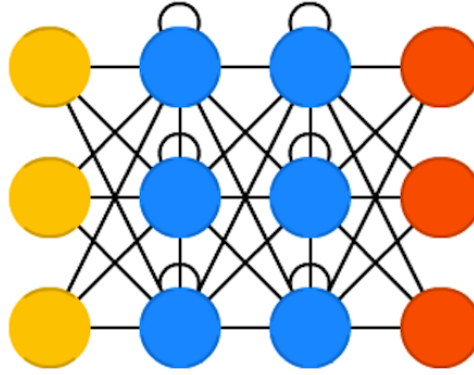


Figure 10: Recurrent Neural Network <sup>10</sup>

Simple ANN and CNN models assumed that all inputs are independent of each other. However, in RNN, the output is dependent on the previous computations. Thus, a RNN has an internal memory to save previous computations. For instance, RNN is popular in NLP tasks where it predicts next word depending on previous words in any given sequence using the internal memory in neurons of the hidden layers.

Long short-term memory (LSTM) is a special type of RNN (shown in Figure 11) proposed by Hochreiter and Schmidhuber in 1997 [3] in order to solve the vanishing gradient problem. It uses LSTM neurons (memory cells with three gates: input, output and forget) instead of simple neurons in the hidden layers. In addition, LSTMs are designed in a way to avoid the long-term dependency problem. While simple RNN can “remember” previous information for short time periods, LSTM can remember them for much longer time periods. However, if the information is not used, it will be lost.

An LSTM cell is smarter than a simple neurons as it has a memory to store previous sequences. Each cell contains gates that manage its state and output. Each gate within a unit uses the sigmoid activation function to decide whether it is triggered or not, which makes it conditional to change state or add information.

These three gates within a memory cell are **forget gate** to decide what information to discard from each LSTM cell, **input gate** to decide the update of memory state depending on input values and **output gate** to decide what to output based on the input and the memory of the LSTM cell.

<sup>10</sup><http://www.asimovinstitute.org/neural-network-zoo/>

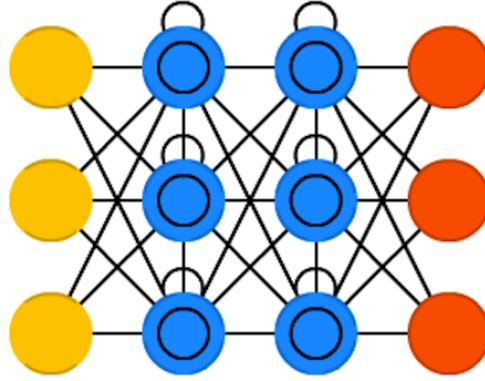
Figure 11: LSTM Neural network <sup>10</sup>

Table 2: Frameworks used for this comparative study

Framework	Major Version	Github Commit ID
CNTK	2.0	7436a00
TensorFlow	1.2.0	49961e5
Theano	0.10.0.dev1	8a1af5b
Keras	2.0.5	78f26df

## 4 Deep Learning Frameworks

The frameworks considered in this comparative study are: CNTK, TensorFlow and Theano. We also use Keras on top of these frameworks as will be discussed later. All of these frameworks provide flexible APIs and configuration options for performance optimization. Software versions of the frameworks<sup>11</sup> are shown in Table 2 and their properties are shown in Table 3.

### 4.1 CNTK

Microsoft Cognitive Toolkit (CNTK) is an Open source DL framework developed by Microsoft Research [37] for training and testing many types of NN across multiple GPUs or servers. CNTK supports different DL architectures like Feedforward, Convolutional, Recurrent, LSTM, and Sequence-to-Sequence NN.

A Computational Network learns any function by converting it to a directed graph where each leaf node consists of an input value or a learning parameter, whereas, each other node represents a matrix operation applied on its children. In this case, CNTK has an advantage as it can automatically find the derive gradients for all the computations which are required to learn the parameters. In CNTK, users specify their networks using a configuration file that contains information about the network type, where to find input data, and the way to optimize parameters [38].

<sup>11</sup>This work was conducted in the summer of 2017. The versions we consider were the latest ones. Since then, these frameworks have been updated.

Table 3: Properties of the considered frameworks

Property	CNTK	TensorFlow	Theano	Keras
Core	C++	C++	Python	Python
CPU	✓	✓	✓	✓
Multi-Threaded CPU	✓	Eigen	Blas, conv2D, Limited OpenMP	✓
GPU	✓	✓	✓	✓
Multi-GPU	✓	✓	X (experimental version)	✓
NVIDIA cuDNN	✓	✓	✓	✓

CNTK interface supports different APIs of several languages such as Python, C++ and C# across both GPU (CUDA) or CPU platforms. According to its developers [39], CNTK was written in C++ in an efficient way, where it removes duplicated computations in forward and backward passes, uses minimal memory and reduces memory reallocation by reusing them. The framework’s installation is discussed in [40].

## 4.2 Theano

Theano<sup>12</sup> is an open source Python library developed at MILA lab at University of Montreal as a compiler for mathematical expressions that let users and developers optimize and evaluate their expressions using a NumPy’s syntax (a Python library that supports a large and multi-dimensional arrays) [41, 8]. Theano starts performing computations automatically by optimizing the selection of computations, translates them into other machine learning languages such as C++ or CUDA (for GPU) and then compiles them into Python modules in an efficient way on CPUs or GPUs.

Theano is being actively developed since 2008, which makes it more popular on a research and ecosystem platform than many DL libraries. Several software packages have been developed to build on top of Theano, with a higher-level user interface which aims to make Theano easier to express and train different architectures of deep learning models, such as Pylearn2, Lasagne, and Keras. The framework’s installation is discussed in [40].

## 4.3 TensorFlow

TensorFlow is an open source framework developed by Google Brain Team [42], that uses a single data flow graph, expressing all numerical computations, to achieve excellent performance. TensorFlow constructs large computation graphs where each node represents a mathematical operation, while the edges represent the communication between nodes. This data flow graph executes the communication between sub-computations explicitly, which makes it possible to execute independent computations in parallel or to use multiple devices to execute partition computations [43]. The framework’s installation is discussed in [40].

Programmers of TensorFlow define large computation graphs from basic operators, then distribute the execution of these graphs across a heterogeneous distributed system (can deploy computation to one or more CPUs or GPUs on a different hardware platforms such as desktops, servers, or even mobile devices). The flexible architecture of TensorFlow allows developers and users to experiment and train a wide variety of deep neural network models, and it is used for deploying machine learning systems into production for different fields including speech recognition, NLP, computer vision, robotics, and computational drug discovery. TensorFlow uses different APIs of several languages such as Python, C++, and Java for constructing and executing a graph (Python API is the most complete and the easiest to use) [44]. The framework’s installation is discussed in [40].

## 4.4 Keras

Keras is an open source DL library developed in python. It runs on top of CNTK, Theano or TensorFlow frameworks. Keras was founded by Google engineer Chollet [45] in 2015 as a part of the research project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System). Keras is designed in a way that allows fast expression with deep neural networks and easy and fast prototyping (modularity and extensibility) [46]. The framework’s installation is discussed in [40].

# 5 Methods

The goal of this experimental study is to compare the aforementioned frameworks (Theano, TensorFlow and CNTK) by using them to train CNN and RNN models on standard benchmark datasets of classical problems in image processing (MNIST, CIFAR-10, and Self-driving Car) and NLP (Penn TreeBank and IMDB). We then evaluate each framework’s performance through the following metrics:

- Running time
- Memory consumption
- CPU and GPU utilization
- Number of epochs

---

<sup>12</sup>Theano is no longer supported but it was so when this paper has been written.  
<https://groups.google.com/forum/#!topic/theano-users/7Poq8BZutbY>

We aim at comparing the aforementioned frameworks using a GPU-equipped laptop that runs Windows 10 operating system, and has the following specifications:

- Intel Core i7-6700HQ CPU @ 2.60GHz (4 cores)
- 16 GB RAM
- 64-bit operating system, x64-based processor
- NVIDIA GEFORCE GTX 960m graphics card (Laptop), with PCI Express 3.0 bus support, equipped with 4 GB GDDR5 memory and 640 CUDA cores.

It is worth mentioning that our goal is to compare the resources consumed by each framework to reach a certain accuracy level for each problem. So, we experimented with different epoch counts in order to make sure the accuracy for all frameworks are close to each other.

CNN is applied for different classical problems including: MNIST, CIFAR-10, IMDB and Self-Driving Car datasets, where each dataset has a different network architecture. When applying CNN datasets on both Theano and Tensorflow, Keras is used for coding all of these datasets, while, in CNTK, Keras is used only when applying Self-Driving car and IMDB datasets. Thus, MNIST and CIFAR-10 are implemented without Keras.

For the MNIST and CIFAR-10 datasets, two convolutional layers with ReLU activation function are used after the input layer. The activation function is used to reduce the training time and to prevent vanishing gradients. After each CNN layer, a max-pooling layer is added in order to down-sample the input and to reduce overfitting. In the max-pooling layer, the stride value must be specified with which the filter is slid. When the stride is one, the filter (window) is moved one pixel at a time. When the stride is two, the filter moved two pixels at a time. This will produce smaller output volumes spatially. After each max-pooling layer, the dropout method is used in order to reduce overfitting by forcing the model to learn many independent representations of the same data through randomly disabling neurons in the learning phase. The sequential architecture (layers part only) used on the MNIST and CIFAR-10 datasets are shown in [40].

Another example of applying CNN is on the Self-Driving Car dataset, where the network has the same components as the ones used with the MNIST and CIFAR-10 datasets, but with deeper model that consists of five convolutional layers with Exponential Linear Unit (ELU) activation function. The sequential architecture of this CNN is shown in [40]. The convolutional layers are used for feature engineering. The fully connected layer is used for predicting the steering angle (final output). The dropout avoids overfitting and, finally, the ELU activation function is used to solve the problem of the vanishing gradient.

The final example of applying CNN is on the IMDB dataset. The movie reviews in this dataset are composed of sequences of words of different lengths. These words are encoded by mapping movie reviews to sequences of word embeddings where words are mapped to vectors of real numbers; the network architecture consists of an embedding layer followed by a 1D convolution layer which is used for temporal data followed by a global max-pooling operation. These sequences are padded to have the same size as the largest sequence because they have different lengths. The sequential architecture used on IMDB dataset is shown in [40].

The other neural network type we consider is RNN with LSTM. One of the most popular uses of LSTM is for text analysis tasks such as the ones associated with the Penn TreeBank (PTB) dataset. Word-level prediction experiments on PTB was adopted, which consists of 929k training words, 73k validation words, and 82k test words. It has 10k words in its vocabulary. We trained models of two sizes (small LSTM and medium LSTM) and we uses the same architecture presented in [47].

To evaluate language models of the PTB implementation, a special metric called a perplexity is used, where better prediction accuracy is achieved when perplexity value is as low as possible. Perplexity is the inverse of probability definition. This means that minimizing perplexity value is the same as maximizing probability. The goal of applying PTB dataset is to match a probabilistic form which assigns probabilities to sentences. This process is done by predicting next words in a text given a history of previously located words. LSTM cell presents the core of the model which processes one word at a time and computes probabilities of the possible values for the next word in the sentence. A vector of zeros is used for the memory state of the network to get initialized and updated after reading each word.

In the small LSTM model, two hidden layers (with 200 LSTM units per layer) are used with Tanh activation function. Its weights are initialized to 0.1. We train it for four epochs with a learning rate of one (number of epochs trained with initial learning rate), and then the learning rate is decreased by a factor of two after each epoch (the decay of the learning rate for each epoch after four epochs), for a total of 13 training epochs. The size of each batch is 20, then the network is unrolled for 20 steps. The sequential architecture used for PTB dataset is shown in [40].

## 5.1 Benchmark Datasets

In this subsection, we discuss the datasets used in our experiments.

### 5.1.1 MNIST

The MNIST (Mixed National Institute of Standards and Technology) dataset (shown in Figure 17) is a computer vision database for handwritten digits. It is widely used for training and testing in the field of machine learning [34, 48]. MNIST has a training set of 60,000 images and a testing set of 10,000 images. It is a subset of a larger set available from NIST. Each image is  $28 \times 28$  pixels which can be represented as a big array of numbers. We can flatten this array into a vector of  $28 \times 28 = 784$  numbers. Each image in MNIST has a corresponding label, a number between 0 and 9 representing the digit appearing in the image. See Figure 17.

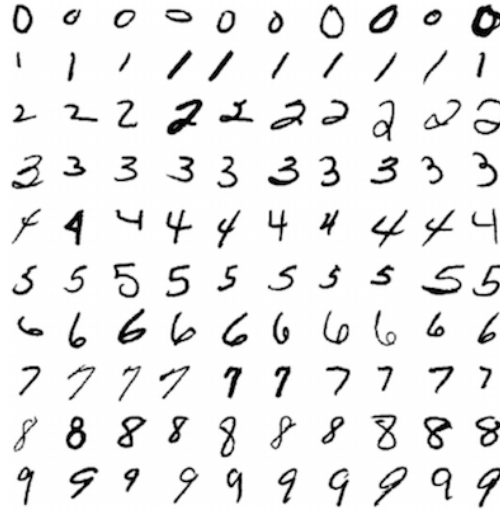


Figure 12: MNIST digits <sup>13</sup>

Our goal is to construct a CNN to classify MNIST images. The training will be carried out on both CPU and GPU environments using different frameworks including the aforementioned ones. We then evaluate the performance of each framework.

### 5.1.2 CIFAR-10

The CIFAR-10 dataset is one of the 80 million images datasets, collected by Krizhevsky et al. [33, 49]. It consists of 60,000  $32 \times 32$  color images evenly distributed over ten classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. There are 50,000 training images and 10,000 test images.

Figure 13 shows the classes in the dataset, as well as 10 random images from each class. The classes are completely mutually exclusive. I.e., there is no overlap between them. For instance, the “Automobile” class includes sedans, SUVs, etc. On the other hand, the “Truck” class includes only big trucks. To avoid overlap, neither one of these two classes includes pickup trucks.

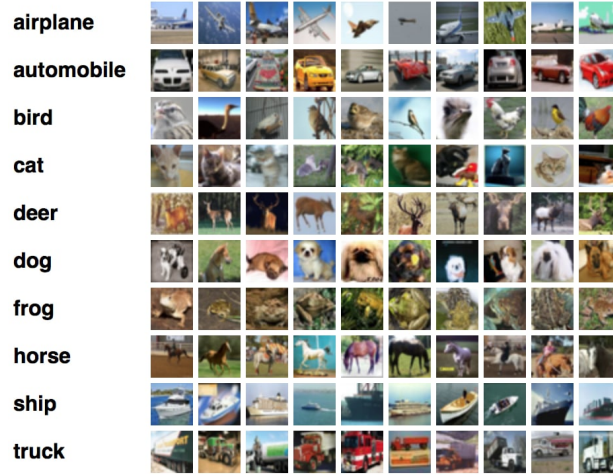
### 5.1.3 Penn TreeBank

In 1993, Marcus et al. [12] wrote a paper on constructing a large annotated corpus of English called the Penn TreeBank (PTB). They reviewed their experience with constructing one large annotated corpus that consists of over 4.5 million words of American English. The project was divided into phases. For the first three-year phase, the corpus was annotated for part-of-speech (POS) tag information (See Figure 14). Moreover, half of the corpus was annotated for skeletal syntactic structure.

<sup>13</sup><https://goo.gl/Gm8xR7>

<sup>14</sup><https://www.cs.toronto.edu/~kriz/cifar.html>

<sup>15</sup><http://wenhoujx.blogspot.com/2013/06/penn-treebank-pos-tags.html>

Figure 13: CIFAR-10 dataset classes <sup>14</sup>

The Penn Treebank POS tagset.

1. CC	Coordinating conjunction	25. TO	<i>to</i>
2. CD	Cardinal number	26. UH	Interjection
3. DT	Determiner	27. VB	Verb, base form
4. EX	Existential <i>there</i>	28. VBD	Verb, past tense
5. FW	Foreign word	29. VBG	Verb, gerund/present participle
6. IN	Preposition/subordinating conjunction	30. VBN	Verb, past participle
7. JJ	Adjective	31. VBP	Verb, non-3rd ps. sing. present
8. JJR	Adjective, comparative	32. VBZ	Verb, 3rd ps. sing. present
9. JJS	Adjective, superlative	33. WDT	<i>wh</i> -determiner
10. LS	List item marker	34. WP	<i>wh</i> -pronoun
11. MD	Modal	35. WP\$	Possessive <i>wh</i> -pronoun
12. NN	Noun, singular or mass	36. WRB	<i>wh</i> -adverb
13. NNS	Noun, plural	37. #	Pound sign
14. NNP	Proper noun, singular	38. \$	Dollar sign
15. NNPS	Proper noun, plural	39. .	Sentence-final punctuation
16. PDT	Predeterminer	40. ,	Comma
17. POS	Possessive ending	41. :	Colon, semi-colon
18. PRP	Personal pronoun	42. (	Left bracket character
19. PP\$	Possessive pronoun	43. )	Right bracket character
20. RB	Adverb	44. "	Straight double quote
21. RBR	Adverb, comparative	45. '	Left open single quote
22. RBS	Adverb, superlative	46. "	Left open double quote
23. RP	Particle	47. '	Right close single quote
24. SYM	Symbol (mathematical or scientific)	48. "	Right close double quote

Figure 14: PTB POS tagset <sup>15</sup>

The dataset is large and diverse. It includes the Brown Corpus (retagged) and the Wall Street Journal Corpus, as well as Department of Energy abstracts, Dow Jones Newswire stories, Department of Agriculture bulletins, Library of America texts, MUC-3 messages, IBM Manual sentences, WBUR radio transcripts, and ATIS sentences.

### 5.1.4 IMDB

IMDB dataset [4, 50] is another example of applying CNN, which is an online dataset of information regarding films, TV programs and video games. It consists of 25,000 reviews labeled by the sentiment (positive/negative) of each review. The reviews have been preprocessed and encoded as integers in a form of a sequence of word indexes (See Figure 15). Words are indexed by overall frequency in the dataset, so that the index  $i$  encodes the  $i$ th most frequent word in the data in order to allow operations of quick filtering.

### 5.1.5 Self-Driving Car

This dataset uses a Udacity's Self-Driving Car simulator as a testbed for training an autonomous car. This work started in the 1980s with Carnegie Mellon University's Navlab and ALV projects [51]. The training phase starts with activating

<sup>16</sup><https://goo.gl/S8a6P4>



Positive words	it!, turns!, blown, good!, wow!, down!, amazingly, book!, book!!, next!, read!, movie!, brilliantly, masterful, awesome, superb, fabulous, it!, wait, wonderfully, highly, turner!, incredible, toes, fantastic, bed, masterfully, thank, prime, loved, favor, I, blew, excellent, master, time!, chilling, amazing, crafted, end!, roller, story!, seat, loves, edge, gift, twice, beautiful, insightful, layers, constantly, wow, keeps, night, coaster, pieces, terrific, sleep, genius, predict, unpredictable, morning, thrilling, reading!, intricate, complex, fascinating, funny, immediately, enjoys, woven, late, unfolds, minute, love, beautifully, brilliant, surface, perfect, witty, till, fast-paced, intense
Negative words	waste, poorly, wasted, worst, ridiculous, dumb, badly, awful, skipped, horrible, worse, depressing, pathetic, terrible, stupid, silly, boring, annoying, unrealistic, bother, poor, contrived, unbelievable, stuck, miserable, profanity, implausible, selfish, sorry, mistake, unlikeable, unlikable, struggled, bothered, mess, quit, hated, death, book?, shallow, negative, disliked, cliché, dull, care, really?, annoyed, suspend, pass, sadly, cared, skip, holes, stopped, plain

Figure 15: Positive/Negative movie reviews sentiment for IMDB <sup>16</sup>

the simulator which is an executable application. A user initiates the service of collecting the data for training followed by collecting the data as images and saving them locally on the computer. So, the framework can take these images and train them. The training is done via distinguishing the image's edges which are taken by the three cameras laying on the front of the car in the simulator. After the training phase is done, the testing phase begins by taking the file generated whenever the performance in the epoch is better than the previous best. Finally, the last generated file is executed in order to make the car drive autonomously to observe the testing phase results. See figure 16.

Figure 16: Training Self-Driving Car dataset via Udacity's simulator<sup>17</sup>

## 6 Results

After implementing the experiments of each dataset on each framework the target was to find which of these frameworks had the best performance. Relating to table 4, regarding image recognition datasets (MNIST and CIFAR-10), one can observe the superiority of CNTK over Tensorflow and Theano in terms of GPU and CPU multithreading, but in CIFAR-10 using 8,16 and 32 threads in CPU Tensorflow was faster than CNTK. On the other hand, Theano revealed to be more time consuming than other frameworks. Transitioning to sentiment analysis dataset (IMDB),

<sup>17</sup><https://github.com/upul/Behavioral-Cloning>

Table 4: Processing time for each dataset (measured in seconds)

Dataset	Environment	Threads	CNTK	TensorFlow	Theano
MNIST	CPU	1	847	5130	3560
	CPU	2	630	3180	2500
	CPU	4	574	2070	2260
	CPU	8	560	1740	2060
	CPU	16	567	1920	2050
	CPU	32	588	2010	2050
	GPU	-	66.67	328.93	377.86
CIFAR-10	CPU	1	20196	25905	26700
	CPU	2	14520	16610	18700
	CPU	4	13662	11550	17250
	CPU	8	11484	9955	15800
	CPU	16	11550	10340	15850
	CPU	32	11649	10835	15750
	GPU	-	926	2166.4	2386.1
IMDB	CPU	1	-	1244	538
	CPU	2	-	642	412
	CPU	4	-	390	380
	CPU	8	486	290	368
	CPU	16	-	249	368
	CPU	32	-	302	384
	GPU	-	73.1	62.4	220.41
Self-Driving Car	CPU	1	-	33.3 hours	50 hours
	CPU	2	-	19.8 hours	44.2 hours
	CPU	4	-	15 hours	42.6 hours
	CPU	8	47.6 hours	14.1 hours	43.5 hours
	CPU	16	-	16.4 hours	43.5 hours
	CPU	32	-	16.4 hours	43.5 hours
	GPU	-	8.7 hours	6 hours	6.8 hours
Penn TreeBank	CPU	1	-	40560	27066
	CPU	2	-	26819	23244
	CPU	4	-	18733	21541
	CPU	8	4290	16407	21450
	CPU	16	-	16848	21476
	CPU	32	-	18369	21541
	GPU	-	2106	1342.28	1630

CPU multithreading was not performed because CNTK is written in python in which multithreading is not supported. Without CPU multithreading (CPU uses the default number of existing physical cores which are equal one thread per core), the superiority of TensorFlow is revealed in both CPU and GPU environments. In text analysis dataset (Penn TreeBank), as mentioned in IMDB dataset (CNTK does not support python multithreading), the default number of threads (8) is taken in consideration while comparing the results. This shows the superiority of Tensorflow over CNTK and Theano, as well as the case in GPU. Moving forward to video analysis dataset (Self-Driving Car), using 8 CPU threads only for the same reason mentioned in IMDB and Penn TreeBank datasets, the superiority of TensorFlow is revealed in both CPU and GPU environments, while CNTK showed to be more time consuming than the other two frameworks.

## 7 Discussion

At this section, a visual representation of tables that appeared in the results section is presented. Figures (17 – 22) include CPU multithreading and GPU processing time, as well as figures (23, 24, 25) represent memory utilization for both CPU and GPU for each framework. The processing times clearly show the advantage of GPU over CPU for training deep convolutional and recurrent neural networks. The advantage of fast GPU would be more significant when training complex models with larger data as in the Self-Driving Car dataset. From the CPU results, the best performance

occurred when the number of threads is equal to the number of physical CPU cores, where each thread possesses a single core. In our work we used a laptop with 8 cores, so in each dataset the best performance in terms of processing time was achieved while using 8 threads as shown in figures (17 – 22). The metrics measurement of each framework was conducted to explain the failure of one of the selected frameworks.

We noticed poor performance of Theano at most datasets comparing to CNTK and TensorFlow. This could be explained because of low CPU utilization comparing to aforementioned frameworks, meanwhile CNTK and TensorFlow use all available resources (high CPU utilization). CNTK outperformed both TensorFlow and Theano while training MNIST and CIFAR-10 datasets. This achievement is highly likely due to the use of BrainScript [52] format which is a custom network description language that makes CNTK more flexible for neural networks customization. On the other hand, TensorFlow uses Eigen [53], which is a C++ template library (BLAS library) for linear algebra including matrices, vectors, numerical solvers, and related algorithms. It is used to make TensorFlow perform better than CNTK and Theano in RNN.

Comparing our work to previous work such as S.Bahrampour’s et al. work presented at [2] and S.Shi et al. work presented at [5], we reveal the following findings.

S.bahrampour et.al based their comparative study on three main aspects including speed, hardware utilization, and extensibility. Besides they used three neural network types such as CNNs, AE, and Recurrent LSTM to train MNIST, ImageNet, and IMDB datasets on Caffe, Neon, Tensorflow, Theano and Torch frameworks. They used the following hardware specs; Intel Xeon CPU E5-1650 v2 @3.5GHz (with multi-threading), NVIDIA GeForce GTX Titan X/PCI/SSE2, 32GB DDR3 Memory, and a SSD drive to come up with the following results; while training on CPU, Torch performed the best followed by Theano, but Neon had the worst performance, also Theano and Torch are the best in extensibility term, as well as TensorFlow and Theano were very flexible and Caffe was the easiest to find the performance. Regarding training datasets on GPU, and for larger convolutional and fully connected networks, Torch was the best followed by Neon. For smaller networks Theano was the best. For LSTM, Theano results was the best in performance, while TensorFlow performance is not competitive compared to other studied frameworks.

On the other hand, S.Shi et al. based their comparative study on two comparative terms including processing time and convergence rate. The neural networks used are fully connected NN, CNNs and RNNs to train ImageNet, MNIST, and CIFAR10 datasets on Caffe, CNTK, MXNet, TensorFlow, and Torch frameworks. They used the following hardware specs; Two types of multi-threaded CPU: desktop CPU (intel i7-3820) (8 threads) and server-grade CPU (intel xeon E5-2630) (32 threads), as well as three types of NVIDIA GPU (GTX980, GTX1080, Tesla K80), and Two Tesla K80 cards used to evaluate the multi-GPU performance. The results of TensorFlow were the best while using CPU. While using single GPU; on FCNs, Caffe, CNTK and Torch performed better than MXNet and TensorFlow. As for small CNN; Caffe and CNTK achieved a good performance, and for RNN (LSTM), CNTK was the fastest (5-10x faster than other frameworks). Using Multi-GPU implementation, all frameworks had higher throughput and accelerated the convergence speed.

Our comparative study was based on measuring performance for CNTK, TensorFlow, and Theano using two types of networks CNNs and RNNs to train MNIST, CIFAR-10, Penn TreeBank, IMDB, and Self-Driving Car datasets on CNTK, TensorFlow, and Theano. The results were as follows: regarding image recognition datasets (MNIST and CIFAR-10) one can observe the superiority of CNTK over Tensorflow and Theano in terms of GPU and CPU multithreading, but in CIFAR-10 using 8,16 and 32 threads in CPU Tensorflow was faster than CNTK. On the other hand, Theano revealed to be more time consuming than be aforementioned frameworks. Transitioning to sentiment analysis dataset (IMDB), CPU multithreading was not performed because of CNTK is written in python in which the multithreading is not supported yet, but while not using CPU multithreading (CPU uses the default number of existing physical cores which are equal one thread per core) the superiority of TensorFlow reveals in both CPU and GPU environments. In text analysis dataset (Penn TreeBank), as mentioned in IMDB dataset CNTK does not support python multithreading, so the default number of threads (8) are taken in consideration while comparing the results which shows the superiority of Tensorflow over CNTK and Theano, as well as the case in GPU. Moving forward to video analysis dataset (Self-Driving Car), using 8 CPU threads only for the same reason mentioned in IMDB and Penn TreeBank datasets, the superiority of TensorFlow reveals in both CPU and GPU environments while CNTK showed to be more time consuming than the other two frameworks.

## 8 Conclusions and Future Work

### 8.1 Conclusion

In this paper, we have provided a qualitative and quantitative comparison between three of the most popular and most comprehensive DL frameworks (namely Microsoft’s CNTK, Google’s TensorFlow and University of Montreal’s Theano). The main goal of this work was to help end users make an informed decision about the best DL framework

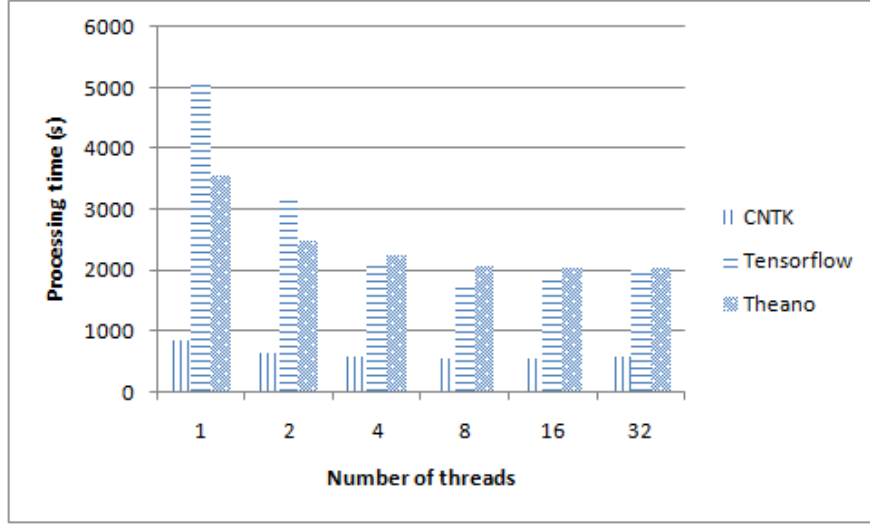


Figure 17: CPU processing time for MNIST dataset

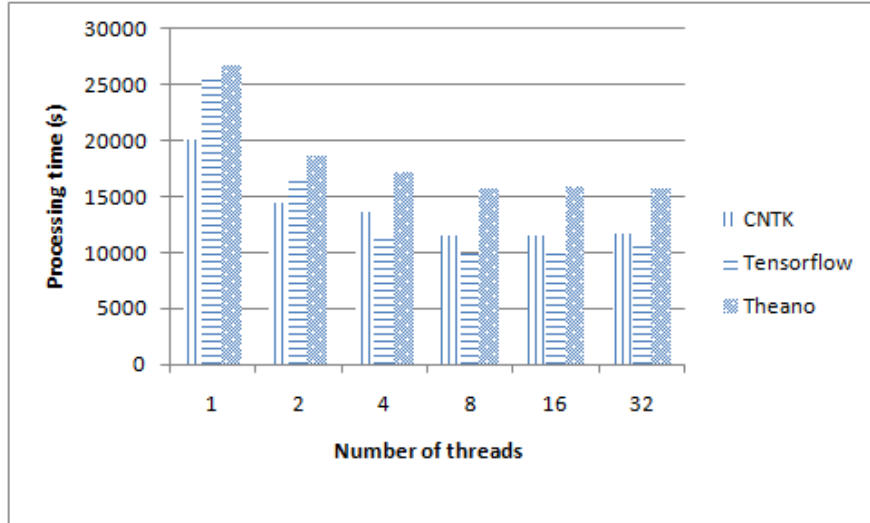


Figure 18: CPU processing time for CIFAR-10 dataset

that suits their needs and resources. To ensure that our study is as comprehensive as possible, we have used multiple benchmark datasets namely MNIST, CIFAR-10, Self-Driving Car, and IMDB which were trained via multilayer CNN network architecture and Penn TreeBank dataset which was trained via RNN architecture. We have run our experiments on a laptop with windows 10 operating system. We have measured performance and utilization of CPU multithreading , GPU and memory.

After the evaluation of the datasets, we have observed the following:

- In all implementations, the processing time clearly shows the advantage of GPU over CPU for training networks.

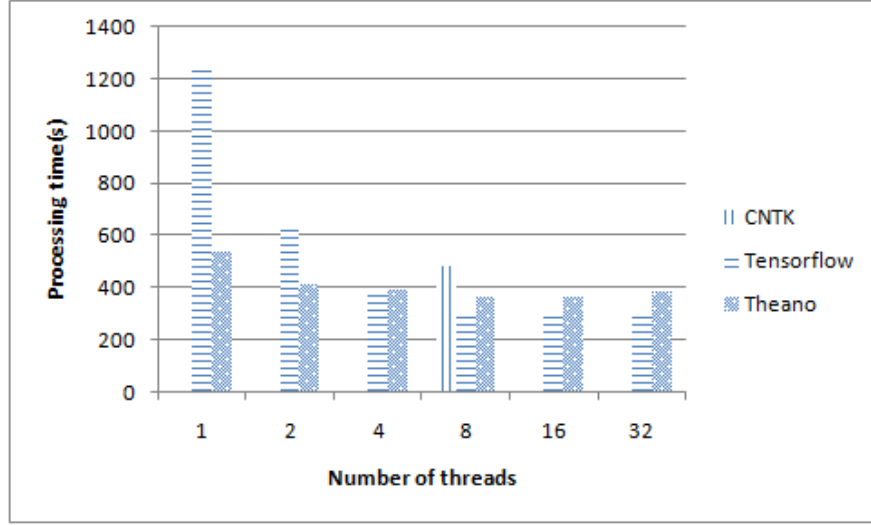


Figure 19: CPU processing time for IMDB dataset

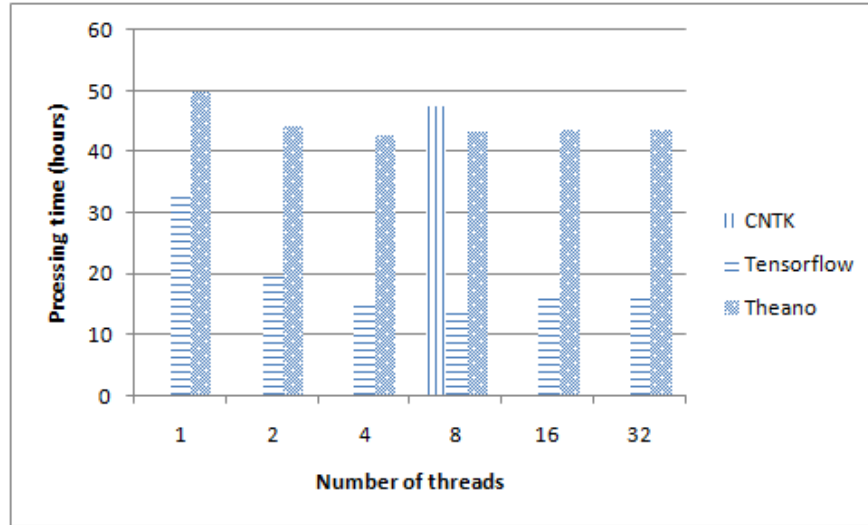


Figure 20: CPU processing time for Self-Driving Car dataset

- For GPU utilization metric in MNIST and CIFAR-10 datasets, TensorFlow had the lowest utilization, followed by Theano and CNTK. For CPU utilization metric, Theano had the lowest utilization followed by TensorFlow and CNTK. For Memory utilization while using CPU and GPU, the results were close to each other.
- For GPU utilization metric in IMDB dataset, the results were close to each other. For CPU utilization metric, Theano had the lowest utilization followed by TensorFlow and CNTK. For Memory utilization while using CPU and GPU, TensorFlow exhibited the worst memory consumption.
- For GPU utilization metric in Self-Driving Car dataset, the results were close to each other. For CPU utilization metric, Theano had the lowest utilization followed by TensorFlow and CNTK. For Memory utilization while using CPU and GPU, Theano revealed the best memory consumption.

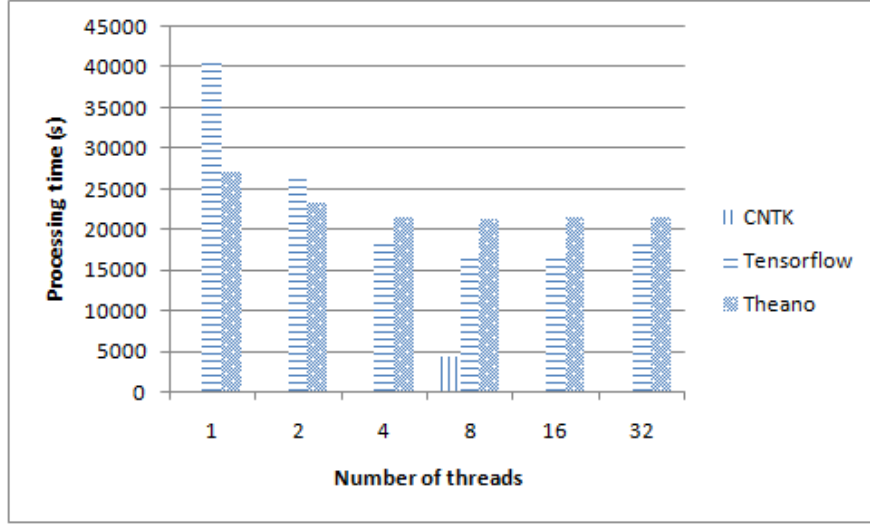


Figure 21: CPU processing time for Penn TreeBnk dataset

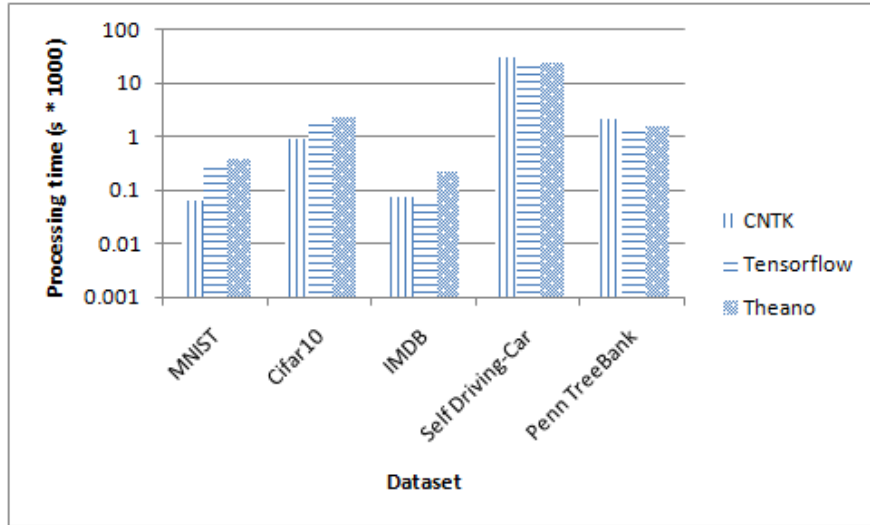


Figure 22: GPU processing time for each dataset

- For GPU utilization metric in Penn TreeBank dataset, the results were close to each other. For CPU utilization metric, Theano had the lowest utilization followed by TensorFlow and CNTK. For Memory utilization while using CPU and GPU, CNTK revealed the best memory consumption.
- In MNIST and CIFAR-10 datasets one can observe the superiority of CNTK over Tensorflow and Theano in terms of GPU and CPU multithreading, but in CIFAR-10 using 8, 16 and 32 threads in CPU Tensorflow was faster than CNTK. On the other hand Theano revealed to be more time consuming than be aforementioned frameworks.
- In sentiment analysis dataset (IMDB), CPU multithreading was not performed because of CNTK is written in python in which the multithreading is not supported yet, but while not using CPU multithreading (CPU

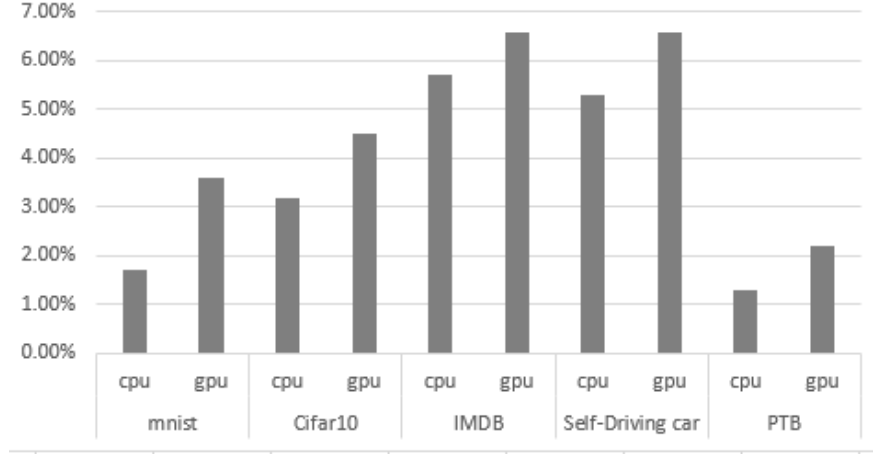


Figure 23: Memory utilization for CNTK framework

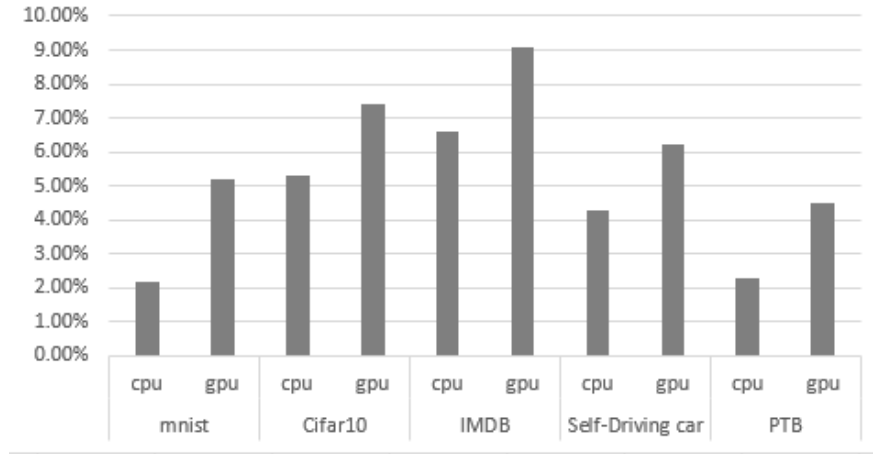


Figure 24: Memory utilization for Tensorflow framework

uses the default number of existing physical cores which are equal one thread per core) the superiority of TensorFlow reveals in both CPU and GPU environments.

- In text analysis dataset (Penn TreeBank), as mentioned in IMDB dataset CNTK does not support python multithreading, so the default number of threads (8) are taken in consideration while comparing the results which shows the superiority of Tensorflow over CNTK and Theano, as well as the case in GPU.
- Moving forward to video analysis dataset Self-Driving Car, using 8 CPU threads only for the same reason mentioned in IMDB and Penn TreeBank datasets, the superiority of TensorFlow reveals in both CPU and GPU environments while CNTK showed to be more time consuming than the other two frameworks.

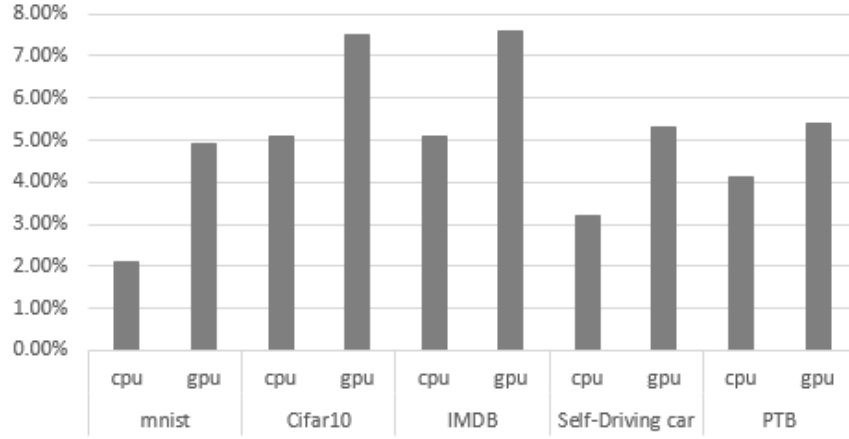


Figure 25: Memory utilization for Theano framework

## References

- [1] Ali Shatnawi, Ghadeer Al-Bdour, Raffi Al-Qurran, and Mahmoud Al-Ayyoub. A comparative study of open source deep learning frameworks. In *2018 9th International Conference on Information and Communication Systems (ICICS)*, pages 72–77. IEEE, 2018.
- [2] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. Comparative study of deep learning software frameworks. *arXiv preprint arXiv:1511.06435*, 2015.
- [3] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [4] Andrew L Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 142–150. Association for Computational Linguistics, 2011.
- [5] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art deep learning software tools. *arXiv preprint arXiv:1608.07249*, 2016.
- [6] Peter Goldsborough. A tour of tensorflow. *arXiv preprint arXiv:1610.01178*, 2016.
- [7] Soumith Chintala. Convnet benchmarks. <https://github.com/soumith/convnet-benchmarks> [Accessed: July 2017].
- [8] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016.
- [9] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.
- [10] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.
- [11] Hector P Martinez, Yoshua Bengio, and Georgios N Yannakakis. Learning deep physiological models of affect. *IEEE Computational Intelligence Magazine*, 8(2):20–33, 2013.
- [12] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.
- [13] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, and Yann LeCun. Fast convolutional nets with fbfft: A gpu performance evaluation. *arXiv preprint arXiv:1412.7580*, 2014.
- [14] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.



- [15] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.
- [16] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [17] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.
- [18] Li Yao, Atousa Torabi, Kyunghyun Cho, Nicolas Ballas, Christopher Pal, Hugo Larochelle, and Aaron Courville. Describing videos by exploiting temporal structure. In *Proceedings of the IEEE international conference on computer vision*, pages 4507–4515, 2015.
- [19] Vassili Kovalev, Alexander Kalinovsky, and Sergey Kovalev. Deep learning with theano, torch, caffe, tensorflow, and deeplearning4j: Which one is the best in speed and accuracy? In *Pattern Recognition and Information Processing (PRIP 2016)*, 2016.
- [20] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590*, 2012.
- [21] Weiguang Ding, Ruoyan Wang, Fei Mao, and Graham Taylor. Theano-based large-scale visual recognition with multiple gpus. *arXiv preprint arXiv:1412.2302*, 2014.
- [22] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.
- [23] Lokenath Debnath and Dambaru Bhatta. *Integral transforms and their applications*. CRC press, 2014.
- [24] Alekseï Grigor’evich Ivakhnenko and Valentin Grigorévich Lapa. Cybernetic predicting devices. Technical report, DTIC Document, 1966.
- [25] Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.
- [26] David R Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 215–242, 1958.
- [27] Seppo Linnainmaa. The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. Master’s thesis, Univ. Helsinki, 1970.
- [28] Alexey Grigorevich Ivakhnenko. Polynomial theory of complex systems. *IEEE transactions on Systems, Man, and Cybernetics*, 1(4):364–378, 1971.
- [29] Kuniyiko Fukushima and Sei Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.
- [30] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.
- [31] Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992.
- [32] Dan Claudiu Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.
- [33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [34] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.
- [36] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.
- [37] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Zhiheng Huang, Brian Guenter, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Huaming Wang, et al. An introduction to computational networks and the computational network toolkit. *Microsoft Technical Report MSR-TR-2014-112*, 2014.

- [38] Dong Yu, Kaisheng Yao, and Yu Zhang. The computational network toolkit [best of the web]. *IEEE Signal Processing Magazine*, 32(6):123–126, 2015.
- [39] Microsoft. The microsoft cognitive toolkit. <https://docs.microsoft.com/en-us/cognitive-toolkit/cntk-evaluation-overview> [Accessed: June 2017].
- [40] Ghadeer Al-Bdour. Comparative study between deep learning frameworks using multiple benchmark datasets. Master’s thesis, Jordan University of Science and Technology, 2017.
- [41] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, pages 1–7, 2010.
- [42] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.
- [43] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, Georgia, USA, 2016.
- [44] Google team. Google. <https://www.tensorflow.org/> [Accessed: June 2017].
- [45] François Chollet et al. Keras, 2015.
- [46] François Chollet. Keras (2015). URL <http://keras.io>, 2017.
- [47] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.
- [48] Yann Lecun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 2009. URL <http://yann.lecun.com/exdb/mnist>, 2009.
- [49] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.
- [50] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.
- [51] Richard S Wallace, Anthony Stentz, Charles E Thorpe, Hans P Moravec, William Whittaker, and Takeo Kanade. First results in robot road-following. In *IJCAI*, pages 1089–1095, 1985.
- [52] Microsoft. Brainscript network builder. <https://docs.microsoft.com/en-us/cognitive-toolkit/BrainScript-Network-Builder> [Accessed: August 2017].
- [53] Blas. Eigen library. [http://eigen.tuxfamily.org/index.php?title=Main\\_Page](http://eigen.tuxfamily.org/index.php?title=Main_Page) [Accessed: August 2017].