TensorFlow for Machine Learning

Gautham Kumar Ramappa Shanmugam
Computer Science
Illinois Institute of Technology
Chicago, USA
gramappashanmugam@hawk.iit.edu

https://github.com/GFA22SCM80K/TensorFlow-on-Machine-Learning

Abstract

An end-to-end open-source machine learning platform is called TensorFlow. TensorFlow is a machine learning system that works in diverse situations and at scale. Machine learning models are created and trained using the TensorFlow API. TensorFlow provides a versatile, complete ecosystem of tools, libraries, and community resources to make it easier to create and use machine learning applications. A TensorFlow is an implementation for machine learning algorithms as well as an interface for defining such algorithms. we also discuss use cases of TensorFlow. In this research paper, we explore TensorFlow and discuss it in relation to current deep learning concepts.

Keywords

TensorFlow, machine learning, deep learning

I. Introduction

Modern machine learning algorithms and artificial intelligence systems have completely changed how many sectors of science and technology approach problems. Modern computer vision, natural language processing, speech recognition, and other approaches have seen notable quality advances. Additionally, the advantages of modern discoveries have trickled down to the person, enriching daily living in a variety of ways. Current machine learning techniques have improved—if not enabled—personalized digital assistants, recommendations on e-commerce platforms, individualized web search results and social network feeds. It's interesting to note that the popularity of deep learning has only recently increased. This is primarily due to the increased availability of large data sets with more training examples, as well as the effective use of graphical processing units (GPUs) and massively parallel commodity hardware to train deep learning models on these similarly enormous data sets.

Even though deep learning algorithms and specific architectural elements like representation transformations, activation functions, or regularization methods may originally be stated in mathematical notation, they ultimately need to be converted into a computer program for use in the real world. There are numerous open source and paid machine learning software libraries and frameworks available for this purpose. TensorFlow is a unique machine learning software library. TensorFlow is intended to be "an interface for expressing machine learning algorithms" in "large-scale on heterogeneous distributed networks" .

This paper provides an in-depth analysis of TensorFlow and situate it within the current machine learning landscape. The paper is further broken down into the following sections. In Section II, we discuss best python libraries used for machine learning. After that, Section III goes into great detail about the computational paradigms that underlie TensorFlow. We describe the present programming interface in Section IV. The comparison of TensorFlow and competing deep learning libraries is presented in Section V. Section VI examines recent TensorFlow use cases before we wrap up our review in Section VII.

II. Python libraries for Machine Learning

As the name implies, machine learning is the science of programming a computer so that it can learn from various types of data. A more general definition given by Arthur Samuel is – "Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed."

A. TensorFlow

TensorFlow is a framework for creating and executing tensor-based calculations. Deep neural networks that can be utilized to create a variety of AI applications can be trained and operated by it. In the area of deep learning research and application, TensorFlow is frequently used.

B. NumPy

NumPy is a well-known Python toolkit for processing huge multidimensional arrays and matrices with the aid of a variety of sophisticated mathematical operations. Its capabilities in linear algebra, the Fourier transform, and random numbers are particularly advantageous. High-end libraries like TensorFlow internally manipulate Tensors using NumPy.

C. SciPy

SciPy has several modules for optimization, linear algebra, integration, and statistics. For manipulating images, SciPy is a great tool.

D. Scikit-learn

One of the most well-liked ML libraries for traditional ML algorithms is Scikit-learn. Scikit-learn is an excellent tool for someone just getting started with machine learning because it can also be used for data analysis and data mining.

E. Theano

A well-known Python package called Theano is employed to effectively define, assess, and optimize mathematical equations involving multi-dimensional arrays. It is widely used to identify and diagnose various kinds of issues during unit testing and self-verification. Theano is a highly powerful library that has long been used in complex, computationally demanding scientific tasks, yet it is straightforward and approachable enough for users to utilize it for their own projects.

F. Keras

Keras is a high-level neural network API that may be used with Theano, CNTK, or TensorFlow. Both the CPU and GPU can operate without any issues. Keras makes creating and designing a neural network incredibly simple for ML beginners. One of Keras' best features is that it makes prototyping simple and quick.

G. PyTorch

PyTorch, an open-source machine learning framework that is built in C with a wrapper in Lua, is the foundation for PyTorch, a well-known open-source machine learning library for Python. It supports a wide range of ML algorithms, including Computer Vision, Natural Language Processing (NLP), and many others. It enables GPU-accelerated Tensor computations for developers and aids in the construction of computational graphs.

H. Pandas

A well-liked Python package for data analysis is Pandas. Pandas is helpful in this situation because it was created primarily for data preprocessing and extraction. It offers a large range of tools for data analysis as well as high-level data structures. It offers a variety of built-in techniques for gathering, integrating, and filtering data.

I. Matplotlib

A well-liked Python library for data visualization is Matplotlib. Plotting is made simple for programmers by the pyplot module, which offers capabilities to control line styles, font attributes, axes formatting, etc. It offers a variety of graphs and plots for visualizing data, including histograms, error charts, bar charts, etc.

III. TensorFlow programming model

We go into great detail about the abstract computational ideas that underlie the TensorFlow software library in this part. we go on to describe how machine learning algorithms can be described in TensorFlow's dataflow graph language. The execution model of TensorFlow is then examined, and it is revealed how TensorFlow graphs are allocated to available hardware units in both a local and a distributed environment.

Machine learning techniques are modeled as computational graphs in TensorFlow. A directed graph with vertices or nodes that represent operations and edges that indicate data flowing between these actions is known as a computational or dataflow graph. If a binary operation on two inputs a and b results in an output variable c, then we draw directed edges from a and b to an output node representing c and annotate the vertex with a label explaining the action that was carried out. The key components of a dataflow graph, including operations, tensors, variables, and sessions, are discussed in the paragraphs that follow.

1.) Operations: The main advantage of expressing an algorithm as a graph is that the description of a node inside the graph can be maintained highly general, in addition to providing an understandable way to communicate connections between computational model components. Nodes in TensorFlow represent operations, which explain how data is combined or transformed as it flows through the graph.

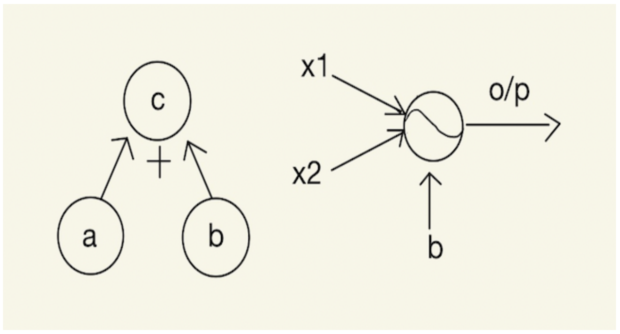


Fig.1 : A left graph shows a relatively straightforward calculation that only requires adding the two input variables, and b. As indicated by the annotation, c in this instance is the outcome of the operation +. A more intricate example of creating a logistic regression variable is shown in the right graph. As indicated by the annotation, x1 and x2 are the inputs, b is the bias and o/p are the Output.

A process can produce zero or more outputs and zero or more inputs. An operation can therefore be a file I/O operation, a control flow directive, a variable or constant, a mathematical equation, or even a network connection port. That an operation, which the reader might equate with a function in the mathematical sense, might represent a constant or variable could appear counterintuitive. A constant, on the other hand, can be conceptualized as an operation that has no inputs and consistently generates the same output corresponding to the constant it represents.

2.) Tensors: Tensors are edges in TensorFlow that represent data moving from one action to another. A multidimensional collection of uniform values having a fixed, static type is referred to as a tensor. The rank of a tensor is its total number of dimensions. The tuple representing a tensor's size, or the number of components, in each dimension, is referred to as its form. In the context of mathematics, a tensor is a generalization of a scalar, a one-dimensional vector, a two-dimensional matrix, and a tensor of rank zero. A tensor can be thought of as a symbolic handle to one of an operation's outputs in terms of the computational graph.

A tensor just offers an interface for obtaining values; it does not itself hold or store them in memory. A tensor object is returned when creating an operation in the TensorFlow programming environment, such as for the expression a + b. The source and destination operations will be connected by an edge when this tensor is used as input in other calculations. Data travels across a TensorFlow graph in this way.

3.) Variables: In a typical situation, the graph of a machine learning model is executed from beginning to end numerous times for a single experiment, as is the case when doing stochastic gradient descent (SGD). The majority of the tensors in the graph are destroyed and do not persist between two of these invocations. However, it is frequently required to preserve state between graph evaluations, for example, when adjusting a neural network's weights and parameters. TensorFlow has variables for this purpose, which are merely extra operations that may be added to the computation graph.

Variables can be thought of specifically as mutable, persistent handles to tensor-storing in-memory buffers. As a result, variables have a specific structure and a fixed type. The assign family of graph operations in TensorFlow allows for the manipulation and updating of variables.
The tensor with which the variable will be initialized upon graph execution must be provided when constructing a variable node for a TensorFlow graph. The variable's form and data type are then inferred from this initializer. It's interesting to note that this initial tensor is not stored in the variable itself. Instead, creating a variable causes the graph to gain three new, separate nodes:

1) The variable node itself, which contains the modifiable state.

2) A process that generates the initial value, frequently a constant.

3) An initializer operation, which, upon evaluation of the graph, assigns the variable tensor its initial value.

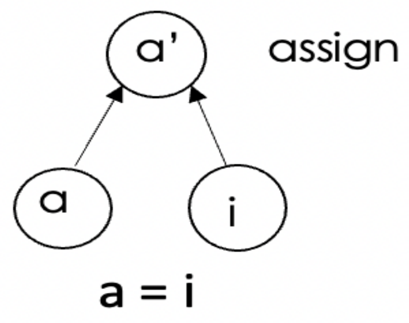


Fig. 2: The three nodes that each variable definition adds to the computational network. The first one, v, is the variable operation that stores the value tensor of the variable in a mutable in-memory buffer. The variable's initial value is produced by the second node, I which can be any tensor. Finally, when the assign node is invoked, the variable will be assigned to the initializer's value. The assign node also generates a tensor that references the variable's initialized value, v 0, allowing it to be attached to other nodes as required.

4) Sessions: Only in a specific setting known as a session is it possible to execute operations and evaluate tensors in TensorFlow. The administration and allocation of resources, such as variable buffers, is one of the duties of a session. Additionally, the TensorFlow library's Session interface offers a run procedure, which serves as the main entry point for running a computational graph in part or in its entirety. The nodes in the graph whose tensors should be computed and

returned are the input for this method. Additionally, an optional feed node mapping from any arbitrary graph nodes to associated replacement values may be provided to execute. As soon as the run command is sent, TensorFlow will begin at the requested output nodes and go backward, analyzing the graph dependencies and computing the full transitive closure of all required nodes. The actual execution units (CPUs, GPUs, etc.) on one or more machines may then be assigned to one or more of these nodes.

IV. TensorFlow programming interface

After presenting the TensorFlow computational model's abstract notions in Section III, this paper will discuss those concepts into concrete form and discuss TensorFlow's programming interface.
We start out by having a quick discussion of the various language interfaces. Then, by guiding you through a straightforward real-world example, this paper gives a more in-depth look into TensorFlow's Python API.

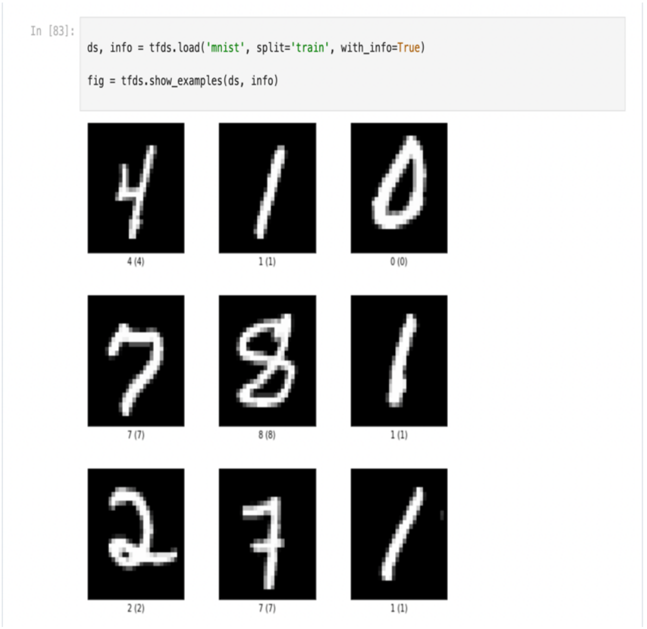
The TensorFlow backend may presently be interacted with using two C++ and Python programming interfaces. For the design and execution of computational graphs, the Python API boasts an extremely robust feature set. At the time of writing, the C++ interface which is actually just the basic backend implementation offers a far more constrained API. Although experimental, C++ does not presently have as much functionality as Python for creating computational graphs.

This research paper provides a detailed overview of a useful, real-world example of the Python API for TensorFlow. To categorize handwritten digits in the MNIST dataset, we will train a straightforward multi-layer perceptron (MLP) with a single input layer and one output layer. The samples in this dataset are tiny images of 28 by 28 pixels showing handwritten digits in the range of 0 to 9. Each of these examples is given to us as a flattened vector of 784 pixel intensities in grayscale. Each example is labeled with the digit it is meant to represent. We load the TensorFlow library into memory and read the MNIST dataset before starting our walkthrough. For this, we presume that the mnist data utility module has a method called read that anticipates a path to extract and save the dataset.

```
import tensorflow as tf
import pandas as pd
import tensorflow_datasets as tfds
d=
pd.read_csv('/content/sample_data/
mnist_test.csv')
print(d.head)

d =
pd.read_csv('/content/sample_data/
mnist_train_small.csv')
print(d.head)

import tensorflow as tf
(train_images, train_labels),
(test_images, test_labels) = tf.

from __future__ import
absolute_import
from __future__ import division
from __future__ import
print_function
```

```
ds = tfds.load('mnist',
split='train', shuffle_files=True)
assert isinstance(ds,
tf.data.Dataset)
print(ds)
```

Using the tf.Graph constructor, a fresh computational graph is created. We must designate this graph as the default graph in order to add operations to it. The TensorFlow API was created in such a way that library functions that produce new operation nodes always connect these to the currently active default graph. By utilizing it as a Python context manager in a with-as statement, we register our graph as the default. We are now prepared to add operations to our computational graph. We start by adding two examples and labels for placeholder nodes. When a graph is executed, placeholders—special variables—must be replaced with actual tensors. To map tensors to replacement values, they must be provided in the feed dict parameter to Session.run(). Each of these placeholders has a shape and data type that are specified. We can currently use the Python keyword None for the first dimension of each placeholder shape in TensorFlow, which is an intriguing feature. The learning task then applies an affine transformation, XW+b, to an example matrix, X R n784 containing n images, where W is a weight matrix, R 78410, and b is a bias vector, R 10.



```
In [83]:
ds, info = tfds.load('mnist', split='train', with_info=True)

fig = tfds.show_examples(ds, info)
```

A new matrix, Y R n 10, is produced as a result, and it contains the scores of our model for each case and each potential digit. These scores don't necessarily need to be between [0, 1] or add up to one because they are more or less arbitrary values and not based on a probability distribution. The softmax function, shown in Equation 3, is used to convert the logits into a legitimate probability distribution, giving the likelihood Pr[x = I that the x-th occurrence represents the digit i. Thus, softmax(X W + b) is used to calculate our final estimates.

```
mnist_train, mnist_test =
tf.keras.datasets.mnist.load_data(
)
 train_data =
np.float16(mnist_train[0])
 train_labels =
np.asarray(mnist_train[1],
dtype=np.int32)
 eval_data =
np.float16(mnist_test[0])
```

```
 eval_labels =
np.asarray(mnist_test[1],
dtype=np.int32)
graph = tf.Graph()
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
with graph.as_default():
  examples =
tf.placeholder(shape=[None, 784],
dtype=tf.float32)
  labels =
tf.placeholder(shape=[None, 10],
dtype=tf.float32)
  weights =
tf.Variable(tf.random_uniform([784
, 10]))
  bias =
tf.Variable(tf.constant(0.1,
shape=[10]))
  logits = tf.matmul(examples,
weights) + bias
  estimates =
tf.nn.softmax(logits)
  logits = tf.matmul(examples,
weights) + bias
  estimates =
tf.nn.softmax(logits)
  cross_entropy = -
tf.reduce_sum(labels *
tf.log(estimates),
reduction_indices=[1])
  loss =
tf.reduce_mean(cross_entropy)
```

We can use (stochastic) gradient descent to update the model weights now that we have an objective function. TensorFlow offers a GradientDescentOptimizer class to help with this. It offers an operation minimize, which we give our loss tensor to, and is initialized with the algorithm's learning rate. To train our model, we will periodically conduct the following operation in a session environment:

```
optimizer =
tf.train.GradientDescentOptimizer(
0.5).minimize(loss)
correct_predictions =
tf.equal(tf.argmax(estimates,
dimension=1), tf.argmax(labels,
dimension=1))
accuracy =
tf.reduce_mean(tf.cast(correct_pre
dictions, tf.float32))
```

Finally, we can put our algorithm to use by training it. For this, we use a tf.Session as a context manager and enter a session environment. In order to inform its constructor of the graph to maintain, we send our graph object to it. We then have multiple ways to execute nodes. Calling Session.run() and passing a list of the tensors we want to compute is the most flexible method. As an alternative, we can explicitly use run() on operations and eval() on tensors.
We must first make sure that the variables in our network are initialized before evaluating any other nodes. Theoretically, we could initialize each variable using the Variable.initializer operation. However, the most popular method is to simply utilize the TensorFlow utility operation tf.initialize all variables(), which
with tf.Session(graph=graph) as session:

```
tf.initialize_all_variables().run(
)
  for step in range(1001):
    example_batch, label_batch =
mnist.train.next_batch(100)
  feed_dict = {examples:
example_batch, labels:label_batch}
  if step % 100 == 0:
    _, loss_value, accuracy_value
= session.run( [optimizer, loss,
accuracy], feed_dict=feed_dict)
    print("Loss at time {0}:
{1}".format(step, loss_value))
    print("Accuracy at time
{0}:{1}".format(step,
accuracy_value))
    else:
      optimizer.run(feed_dict)
```

V. DEEP LEARNING FRAMEWORK COMPARISON WITH OTHERS

There are several additional open-source deep learning software libraries outside TensorFlow, with Keras, PyTorch and Pandas being the most well-known. Here, we examine the qualitative contrasts between these libraries.

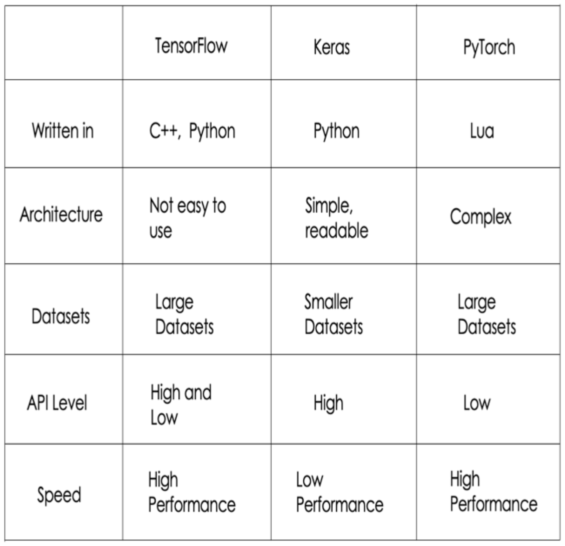
A. Qualitative Comparison

The following two paragraphs compare Keras, PyTorch to TensorFlow, respectively.

1) Keras: Keras is a high-level neural network library that runs on top of TensorFlow while TensorFlow is an open-sourced end-to-end platform, a library for multiple machine learning tasks, while Both provide high-level APIs used for easily building and training models, but Keras is more user-friendly because it’s built-in Python. People turn to TensorFlow when working with large datasets and object detection and need excellent functionality and high performance. TensorFlow runs on Linux, MacOS, Windows, and Android. The framework was developed by Google Brain and currently used for Google’s research and production needs. Keras functions acts as a wrapper to TensorFlow’s framework. It can be defined a model with Keras’ interface, which is easier to use, then drop down into TensorFlow when you need to use a feature that Keras doesn’t have, or you’re looking for specific TensorFlow functionality. Thus, you can place your TensorFlow code directly into the Keras training pipeline or model. The more user-friendly Keras interface can be used to define a model before switching to TensorFlow when you need to employ a feature that Keras lacks or you're looking for a specific TensorFlow utility.

2) PyTorch: A new deep learning framework built on Torch is called PyTorch. It was created by Facebook's AI research team and is freely available on GitHub for applications using natural language processing. The reputation of PyTorch is one of simplicity, use, adaptability, effective memory usage, and dynamic computational graphs. Additionally, it feels native, which speeds up processing and simplifies code. The method the code is executed is the primary distinction between PyTorch and TensorFlow. Tensor is the primary data type that both frameworks operate on. Since both libraries are widely used frameworks, it is impossible to say which one is better. Both are machine learning libraries that can be used for a variety of jobs. With its debugging features, visualization capabilities, and ability to store graphs as protocol buffers, Tensorflow is a helpful tool. The user-friendly nature of Pytorch, on the other hand, keeps it from losing ground and

luring Python developers. In short, developers that are more focused on research choose to utilize Pytorch, whereas Tensorflow is used to automate processes more quickly and create products linked to artificial intelligence. By minimizing boilerplate code, tools like TensorFlow and PyTorch make it easier to construct models. They vary because TensorFlow provides a multitude of options, whereas PyTorch takes a more "pythonic" and object-oriented approach.



	TensorFlow	Keras	PyTorch
Written in	C++, Python	Python	Lua
Architecture	Not easy to use	Simple, readable	Complex
Datasets	Large Datasets	Smaller Datasets	Large Datasets
API Level	High and Low	High	Low
Speed	High Performance	Low Performance	High Performance

Table I: A Table comparing TensorFlow to Keras and PyTorch.

VI. TensorFlow Use cases

A. Voice/Sound Recognition

The most well-known applications for deep learning are those that involve voice and sound identification. In fact, neural networks are able to comprehend audio signals if they are given the right input data flow. Additionally, there are voice-search and voice-activated mobile assistants like Apple's Siri and Google Now (Android).

B. Text based applications

Deep learning is frequently used in text-based applications. programs that analyze text, including fraud detection, threat detection, and sentiment analysis for social media and customer relationship management (CRM). Google Translate, as an illustration, supports more than 100 languages.

C. Image Recognition

Face recognition, image search, motion detection, machine vision, and photo clustering are all applications of image recognition. The automobile, aviation, and healthcare industries can also use picture recognition. Image recognition, for instance, can be used to detect and recognize individuals and objects in pictures. TensorFlow has the advantage of aiding in the classification and identification of arbitrary items inside larger images, which is a benefit when utilizing object recognition algorithms.

D. Time series

Time Series algorithms are used for analyzing time series data in order to extract useful statistics. Time series can be used, for instance, to forecast the stock market. Therefore, deep learning is utilized to produce several time series versions as well as anticipate non-specific time periods.

E. Video Detection

Motion detection and real-time thread detection in gaming, security, airports, and user experience/user interface (UX/UI) areas are the two main applications of video detection.

VII. Conclusion

TensorFlow, a cutting-edge open-source deep learning toolkit built on computational graphs, has been covered in this paper. The programming interface is also covered in the paper. TensorFlow and competing deep learning packages are compared and presented. The paper also looks at current TensorFlow use cases. Fine-grained control over neural network development is provided by its low-level programming interface, while TensorFlow's abstraction modules, such TFLearn, enable quick prototypes. TensorFlow enhances and adds new features in comparison to previous deep learning toolkits like Keras and PyTorch. TensorFlow is a fantastic framework that can be used to compute data graphically and numerically when building deep learning networks. It is the library that is most frequently used for a variety of applications, including Google Search, Google Translate, Google Photos, and many others.

VIII. References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Largescale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org

[2] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, Mohak Shah. Comparative Study of Deep Learning Software Frameworks. Research and Technology Center, Robert Bosch LLC.

[3] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. Advances in Neural Information Processing Systems 32 (NeurIPS 2019).

[4] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich. Going Deeper with Convolutions. arXiv:1409.4842.

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen,

John Tran, Bryan Catanzaro, Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. arXiv:1410.0759.

[6] Peter Goldsborough. A Tour of TensorFlow. arXiv:1610.01178

[7] Laith Alzubaidi1,5* , Jinglan Zhang1, Amjad J. Humaidi2, Ayad Al-Dujaili3, Ye Duan4, Omran Al-Shamma5, J. Santamaría6, Mohammed A. Fadhel7, Muthana Al-Amidie4 and Laith Farhan. Review of deep learning: concepts, CNN architectures, challenges, applications, future directions. journal of Big Data volume 8, Article number: 53 (2021).

[8] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, Andrew Ng. Large Scale Distributed Deep Networks. Advances in Neural Information Processing Systems 25 (NIPS 2012).

[9] Y. Bengio, I. J. Goodfellow, and A. Courville. Deep learning. Book in preparation for MIT Press, 2015.

[10] Sanjeev Arora, Aditya Bhaskara, Rong Ge, Tengyu Ma. Provable Bounds for Learning Some Deep Representations. arXiv:1310.6343.

[11] Ilya Sutskever, James Martens, George Dahl, Geoffrey Hinton. On the importance of initialization and momentum in deep learning. Proceedings of the 30th International Conference on Machine Learning, PMLR 28(3):1139-1147, 2013.