

DISTRIBUTED DEEP LEARNING MODELS: USING TENSORFLOW AND
PYTORCH ON NVIDIA GPU_s AND CLUSTER OF RASPBERRY PIs

By

Jagadish Kumar Ranbirsingh, B.TECH.
Biju Patnaik University of Technology

A Thesis Submitted in Partial Fulfillment of
the Requirements for the Degree of Master of Science in Computer Science
to the office of Graduate and Extended Studies of
East Stroudsburg University of Pennsylvania

May 10, 2019

SIGNATURE/APPROVAL PAGE

The signed approval page for this thesis was intentionally removed from the online copy by an authorized administrator at Kemp Library.

The final approved signature page for this thesis is on file with the Office of Graduate and Extended Studies. Please contact Theses@esu.edu with any questions.

ABSTRACT

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science to the office of Graduate and Extended Studies of East Stroudsburg University of Pennsylvania.

Student's Name: Jagadish Kumar Ranbirsingh

Title: DISTRIBUTED DEEP LEARNING MODELS: USING TENSORFLOW AND PYTORCH ON NVIDIA GPU_s AND CLUSTER OF RASPBERRY PIs

Date of Graduation: May 10, 2019

Thesis Chair: Haklin Kimm, PhD

Thesis Member: Eun-Joo Lee, PhD

Thesis Member: Minhaz Chowdhury, PhD

Abstract

This thesis work focuses on distributed deep learning approaches implementing Human Activity Recognition (HAR) using Recurrent Neural Network (RNN) Long Short-Term Memory (LSTM) model using University of California at Irvine's machine learning database. This work includes developing the LSTM residual bidirectional architecture using Python 3 programming language over distributed TensorFlow and PyTorch programming frameworks on top of two testbed systems: the first one is Raspberry Pi cluster that is built upon 16 Raspberry Pis, clustered together by using parameter server architecture. Another one is the NVIDIA GPU cluster which is equipped with 3 GPUs named Tesla K40C, Quadro P5000 and Quadro K620. Here we compare and observe the performance of our deep learning algorithms in terms of execution time and prediction accuracy with varying number of deep layers with hidden neurons in the neural networks. Our first comparison is based on using TensorFlow and PyTorch over NVIDIA Maximus distributed multicore architecture. The second comparison is the execution of the Raspberry Pi cluster and Octa core Intel Xeon CPU. In this research we present that the implementations of distributed neural network over the GPU cluster perform better than the Raspberry Pi cluster and the multicore system.

ACKNOWLEDGMENTS

I would like to express my deep gratitude to my advisor, Dr. Haklin Kimm, for his everlasting supports and guidance during research and thesis study. Without his invaluable ideas, encouragements, suggestions and corrections, I couldn't have achieved this outcome. I really appreciate his mentorship in this journey. I am also thankful to the members of my thesis examining committee, Dr. Eun-Joo Lee and Dr. Minhaz Chowdhury.

I would like to express my sincere gratitude to the NVIDIA Corporation with the donation of the Tesla K40c GPU and ESU Computer Science department for other GPUs and Raspberry PIs used on this research project.

Most importantly none of this would have been possible without the patience and sacrifice of my family. My wife to whom this dissertation is dedicated to, have been a constant source of courage to push against all odds in these two years. I would like to express my hearty gratitude to my parents.

TABLE OF CONTENTS

LIST OF TABLES.....	vii
LIST OF FIGURES.....	viii
Chapter	
I. INTRODUCTION.....	1
1.1 Machine Learning with Big Data.....	1
1.2 Deep Learning.....	2
1.3 Deep Learning Using GPU.....	3
1.4 Neural Nets.....	3
1.4.1 Recurrent Neural Nets.....	4
1.5 Motivation.....	4
1.6 Thesis Contribution.....	5
1.7 Outline of the Thesis.....	6
II. PREVIOUS STUDIES.....	7
2.1 Big Data.....	7
2.2 TensorFlow.....	8
2.2.1 Architecture.....	9
2.3 Distributed TensorFlow.....	10
2.4 PyTorch.....	15
2.5 MXNet.....	16
III. RELATED WORKS.....	18
3.1 Distributed GraphLab Framework.....	18
3.2 Parameter Server Framework.....	19
3.2.1 Distributed Synchronous Stochastic Gradient Descent.....	21
3.3 Deep Gradient Compression.....	24
IV. LSTM FOR HUMAN ACTIVITY RECOGNITION.....	27
4.1 Human Activity Recognition.....	27
4.1.1 Surveillance System.....	28
4.1.2 Healthcare.....	28
4.1.3 Human Computer Interaction.....	29
4.1.4 HAR Sensing Technologies.....	29
4.2 Data (UCI Repository).....	30
4.2.1 Dataset Information.....	30
4.2.2 Attribute Information.....	31
4.2.3 Feature Notes.....	31
4.3 LSTM.....	35
4.3.1 Why LSTM.....	35

4.3.1.1 CNN.....	35
4.3.1.2 Back Propagation	37
4.3.1.3 RNN.....	42
4.3.1.4 LSTM.....	46
4.3.1.5 Distributed LSTM.....	47
4.3.1.5.1 Synchronous All – Reduce SGD.....	48
4.3.2 Baseline LSTM.....	50
4.3.3 Bidirectional LSTM.....	54
4.3.4 Residual LSTM.....	55
4.3.5 Deep Residual Bidirectional LSTM.....	58
V. TEST BED SETUP.....	64
5.1 NVIDIA GPU Test Bed Setup.....	64
5.2 Cluster of Raspberry PIs Setup.....	72
5.3 Simulation using Raspberry PIs Cluster.....	77
5.4 Notes on TensorFlow Setup.....	82
5.5 Notes on using PyTorch Setup.....	85
VI. IMPLEMENTATION.....	87
6.1 Best Learning Rate.....	87
6.2 CPU Execution Time between Layers.....	89
6.3 GPU Execution Time between Layers.....	91
6.4 Bidirectional Vs Non-Bidirectional Execution between Layers.....	99
6.5 Stack Bidirectional Vs Stack Non-Bidirectional Execution between Layers.....	101
6.6 Best Accuracy between all Layers.....	103
6.7 Between Deep Residual Bidirectional between 3 x3 and 4 x4.....	105
6.8 Between Deep Layer vs Prediction Accuracy vs Exe Time in CPU.....	106
6.9 Between Deep Layer vs Prediction Accuracy vs Exe Time in GPU.....	108
6.10 Deep Layer CPU Execution.....	110
6.11 Lower GPU vs Higher CPU.....	111
6.12 4 x 4 CPU vs GPU Layers.....	113
6.13 Bidirectional Lower vs Stack Higher Layers.....	115
6.14 Stack vs Hidden layer on Execution Time and Prediction Accuracy.....	117
6.15 PyTorch vs TensorFlow Efficiency Comparison.....	120
6.16 Raspberry PI Cluster vs Intel Xeon CPU Efficiency Comparison.....	122
VII. CONCLUSION.....	124
7.1 Summary.....	124
7.2 Future Work.....	125
APPENDIX A – TEST BED ARCHITECTURE.....	127
APPENDIX B – SOURCE CODE.....	143
APPENDIX C – TENSORFLOW SET UP.....	209
APPENDIX D – PYTORCH SETUP.....	212
APPENDIX E – RASPBERRY PI CLUSTER.....	214
REFERENCES.....	217

LIST OF TABLES

Table

1. Dataset Feature Parameters.....	32
2. Hardware configuration of CPU.....	65
3. Hardware configuration of GPUs.....	65
4. Hardware configuration of Raspberry Pi 3 Model B+.....	73
5. Raspberry Pi Cluster Monte Carlo Simulation2. Best Learning Rate.....	81
6. Best Learning Rate.....	87
7. Execution rate between Layers in CPU.....	89
8. Execution rate between Layers in GPU.....	91
9. Bidirectional vs Non-bidirectional layers execution time.....	99
10. Stack Bidirectional vs Stack Non-bidirectional execution time.....	101
11. Best Accuracy in 3 deep layers.....	103
12. Deep Residual 3 x3 vs 4 x 4 layers.....	105
13. Execution matrix of all layers by CPU.....	106
14. Execution matrix of all layers by GPU.....	108
15. 4 x 4 Layers deep CPU execution matrix.....	110
16. 3 x 3 Layer GPU vs 4 x4 Layer CPU execution matrix.....	111
17. 4 x 4 Layers CPU vs GPU Execution.....	113
18. 2 x 2 Bidirectional Stack Layer vs 3 x 3 Stack Layer.....	115
19. 2 x 2 stacked hidden layers vs 4 x 4 stacked hidden layers.....	117
20. Efficiency between PyTorch and TensorFlow.....	120
21. Efficiency between Raspberry Pi Cluster and Intel Xeon CPU.....	122

LIST OF FIGURES

Figure

1. TensorFlow General Architecture.....	9
2. TensorFlow Master Worker Model.....	11
3. Distributed Master workflow.....	12
4. NVIDIA MultiGPU NCCL.....	13
5. Parameter Server Framework.....	21
6. Distributed SGD.....	21
7. Deep Gradient Compression.....	24
8. Dataset File Structure.....	34
9. Basic Feed-Forward and Recurrent cell.....	35
10. Two Connected Neurons with weights.....	40
11. Back Propagation Rule.....	40
12. Convolution Neural Network.....	41
13. RNN Sequential Data Learning Approach.....	43
14. Simple RNN Structure.....	44
15. LSTM Forget Gate.....	50
16. LSTM Input Gate.....	51
17. LSTM Processing Data.....	51
18. LSTM Output Gate.....	52
19. The unfolded structure of one-layer baseline LSTM.....	53
20. The structure of single layer bidirectional LSTM.....	54
21. The structure of single layer residual LSTM.....	57
22. The structure of 2 x 2 residual bidirectional LSTM.....	60
23. The residual bidirectional LSTM parameters.....	61
24. NVIDIA Driver Version.....	67
25. CUDA Toolkit Version.....	68
26. GPU Memory Array.....	71
27. Distributed TensorFlow API.....	72
28. Raspberry PIs NFS connection.....	76
29. NFS Status.....	76
30. TensorFlow Cluster API.....	79
31. TensorFlow Device API.....	79
32. TensorFlow Session API.....	80
33. TensorFlow Server API.....	80
34. Pi Cluster Execution Graph.....	82
35. TensorFlow GPU Growth API.....	82
36. GPU StreamExecutor.....	83
37. GPU Device Selection.....	83
38. Distributed TF Multi GPU.....	84
39. PyTorch Distributed API.....	85
40. PyTorch Memory Shuffle.....	85

41. Best Learning Rate.....	88
42. Big Machine CPU Details.....	89
43. Bubble Chart of CPU Execution.....	90
44. Column Graph of CPU Execution between Layers.....	91
45. Bubble Chart of GPU Execution.....	92
46. Column Graph of GPU Execution between Layers.....	93
47. 2 x 2 Layers GPU Utilization Snapshot.....	94
48. 3 x 3 Layers GPU Utilization Snapshot.....	96
49. 4 x 4 Layers GPU Utilization Snapshot.....	97
50. 3 x 3 Layers CPU Utilization Snapshot.....	98
51. Column Graph of Execution time between bidirectional and non-bidirectional.....	100
52. Deep Bidirectional vs Deep Non-bidirectional Execution time.....	102
53. Best Accuracy among all types of 3 stacked layers.....	104
54. Column graph of 3 x 3 vs 4 x 4 Deep Residual Layers.....	105
55. CPU Execution Graph for all Layers.....	107
56. GPU Execution Graph for all Layers.....	109
57. Column Graph of 4 x 4 deep Layers CPU execution.....	111
58. Column Graph of 3 x 3 GPU vs 4 x 4 CPU Execution Result.....	113
59. Graph of 4 x 4 deep layers CPU vs GPU Execution.....	114
60. Graph of 2 x 2 Bidirectional Stack Layer vs 3 x 3 Stack Layer.....	116
61. Execution time Graph of 2 x2 vs 4x 4 stacked layers.....	117
62. Execution time graph with stack layers vs hidden layers.....	119
63. Execution graph between TensorFlow and PyTorch.....	121
64. Execution graph between Single CPU vs Pi Cluster.....	123
65. NVIDIA GPU Cards.....	128
66. NVIDIA Driver Repository.....	129
67. Graphics Display	130
68. GDM Session.....	132
69 NVIDIA Driver Successful Installation Snapshot.....	133
70. Ubuntu Driver Display.....	134
71. CUDA Toolkit.....	135
72. An L2-regularized version of the cost function used in SGD of RNN.....	140
73. TF Project Screen.....	211
74. PyTorch Project Screen.....	213

CHAPTER I

INTRODUCTION

1.1 Machine Learning With Big Data

More than 2.5 quintillion bytes of data are created each day. The prevalence of data will only increase, so we need to learn how to deal with such large data. Storing this data is one thing, but what about processing it and developing machine learning algorithms to work with it? Solving complex computational problems in a short amount of time, as well as dealing with large-sized data sets and massive amounts of continuously growing data, are some challenges that are being addressed by parallel processing algorithms. Data centers deployed with high-end GPUs enable computational storage and network processing power to support such highly demanding workloads. Access to thousands of cores of each GPU with high-capacity network and high-IOPS (Input/Output Operations Per Second) storage allows for ideal infrastructure, which are built for HPC and Big Data applications. But this is not enough in future. This line of research should focus on developing new Machine Learning (ML) models on adapting (scaling up) existing models in order to handle larger scale datasets.

1.2 Deep Learning

Deep Learning [1] is a sub-field of machine learning concerned with algorithms inspired by the structure and function of the brain called artificial neural networks. It uses non-linear processing units with multiple layers for feature transformation and extraction. It also reflects concepts in multiple hierarchical fashions which corresponds to various levels of abstraction. As per Jeff Dean scientist of Google AI Brain, “When you hear the term deep learning, just think of a large deep neural net. Deep refers to the number of layers typically and so this kind of the popular term that’s been adopted in the press. I think of them as deep neural networks generally.” Modern neural network architectures trained on large datasets can obtain impressive performance across a wide variety of domains, from speech and image recognition, natural language processing and industry-focused applications such as fraud detection and recommendation systems.

Deep Learning (DL) has become a true enabler of AI services. In fact, it is the key driver behind today’s entire field of AI with its real-life practical applications. DL’s business utilization and its ability to support business objectives have enabled AI services to take a hot spot at the company strategic table. From life and health sciences, through engineering and financial modeling, to natural language processing and image recognition, the employment of DL is growing exponentially year by year. This growth in applications of AI services is primarily due to the infrastructure behind the curtain and its utilization of parallel computing with increasingly more advanced GPU technologies to enable such progress.

As the computational power of the machines grow exponentially, the need come to move to higher computation CPUs, when CPUs couldn’t provide enough solutions then technology leaped from CPU to GPU. For DL to take full advantage of the GPU hardware architecture and acceleration, there needs to be an “easy” way to allow algorithms to leverage, scale up and consume underlying infrastructure. DL frameworks represent and combine such sets of tools, interfaces, and libraries, which allow data

scientists, engineers, and developers to build, deploy and manage their training models and networks. They are the building blocks of modern DL deployments. Today, the most popular DL Frameworks include, but are not limited to Tensorflow, Keras, Caffe 2, Pytorch, Theano, Chainer, CNTK, and MXNET. Each of these frameworks is built in a different manner and serves different purposes.

1.3 Deep Learning Using GPU

Deep Learning Neural Networks are becoming continuously more complex. The number of layers and neurons in a Neural Network is growing significantly, which lowers productivity and increases costs. DL deployments leveraging GPUs [2] drastically reduce the size of the hardware deployments, increase scalability, dramatically reduce the training and ROI times and lower the overall deployment cost. The new GPU based systems with access to the latest NVIDIA GPU architectures with PCIe interface or with NVLink interconnections can utilize the access to a massive amount of DL computing power by using GPU clusters.

1.4 Neural Nets

There are three classes of artificial neural networks in general. They are:

Multilayer Perceptrons (MLPs)

Convolutional Neural Networks (CNNs)

Recurrent Neural Networks (RNNs)

In this project we have extensively used RNNs [3] because of their internal memory.

RNNs are able to remember important things about the input they receive, which enables them to be very precise in predicting the future value.

1.4.1 Recurrent Neural Nets

RNNs are the state of the art algorithm for sequential data and used by Apples Siri and Googles Voice Search. This is because, it is the first algorithm that remembers its input, due to an internal memory, which makes it perfectly suited for Machine Learning problems that involve sequential data. It is one of the algorithms behind the scenes of the amazing achievements of Deep Learning [2] in the past few years. In a RNN, the information cycles through a loop. When it makes a decision, it takes into consideration the current input and also learned values received from previous inputs. Therefore a RNN has two inputs, the present and the recent past. A usual RNN has a short-term memory. In combination with a LSTM [4] they also have a long-term memory which is very powerful and used in computation of complex datasets.

1.5 Motivation

The single CPU machine learning is old. It's there from 1959 which is coined by Samuel, Arthur L [5]. It was published in IBM Journal of Research and Development. Then comes GPU. GeForce 256 was marketed as "worlds first 'GPU', or "Graphics Processing Unit", a term coined by NVIDIA at that time as "a single-chip processor with integrated lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second. The GeForce 256 is the original release in NVIDIA's "GeForce" product-line announced on August 31, 1999 and released on October 11, 1999. [6] The machine learning using GPU based data warehouse is new and still going on. Now a days the rate of data generation is very high because of social networking sites, like Facebook, Twitter, WhatsApp, WeChat, Instagram and the list goes on. With advancement in technologies, sensor networks, IoT things, automated systems generate much more data every seconds. So in near future, the data warehouses would be established in multiple geographical areas across the globe. Unfortunately, current deep learning methodologies which based on single location or single dataset won't work. Distributed optimization and inference is becoming a

prerequisite for solving large scale deep learning problems. At scale no single machine can solve these problems efficiently, due to the growth of data and the resulting model complexity, often manifesting itself in an increased number of parameters. [7]

1.6 Thesis Contribution

Inspired by Scaling distributed machine learning with the parameter server [8], we proposed a cluster based platform which is designed by parameter server architecture. The thesis focus on distributed deep learning models to simulate Human Activity Recognition. The Deep Learning LSTM model do the iterations on UCI dataset by using distributed TensorFlow, PyTorch programming frameworks. This includes writing the LSTM residual bidirectional architecture using Python 3 programming language and TensorFlow and PyTorch APIs, where both the APIs support the distributed architecture. Following which, the program is verified in the distributed platform. To meet the distributed hardware demand two platforms are created, first hardware is Raspberry Pi cluster having 16 nodes, which is built upon 16 Raspberry Pis 3 B+ models clustered together by using parameter server architecture, each having 1 GB of RAM and 32 GB of flash storage. Second hardware is, the NVIDIA GPUs cluster which is having 3 GPUs named Tesla K40c, Quadro P5000 & Quadro K620. It is built by NVIDIA Maximus formation on top of Octa-core Intel Xeon CPU having 32 GB RAM and 2 TB SSD primary storage with 10 TB HDD secondary storage. Thus, comparing and observing the performance in terms of executing speed and efficiency of deep learning iterations by varying number of deep layers with hidden neurons in GPUs and CPUs. While the first approach is based on using TensorFlow and PyTorch over NVIDIA GPUs parallel and distributed multicore architecture. The second approach is by comparing the execution speed and efficiency of CPUs of Pi cluster along with Inter Xeon CPU. The research focuses on energy-efficient deep learning computing, which is at the intersection between deep learning and distributed computer.

1.7 Outline of the Thesis

The remaining of the thesis is organized in the following.

Chapter 2 gives some background information about work on the distributed deep learning models. It introduces the distributed deep learning APIs of TensorFlow, PyTorch.

Chapter 3 demonstrates related research work on this topic. It introduces the similar problems and previous research happened on it.

Chapter 4 discusses most about the Why LSTM? , Different LSTM architectures and our proposed LSTM residual bidirectional layer.

Chapter 5 shows the preparation of test beds for this research. It shows the related works during the hardware cluster development.

Chapter 6 implements the deep learning models in distributed platform and compares the GPU computational power between two API in GPU cluster and the computational power between the CPU cluster and standalone CPU machine while doing the iterations. It gives all the implementations with execution time and predicted accuracy with varying dense layers along with different hidden nodes.

Chapter 7 is the conclusion of our thesis work.

CHAPTER II

PREVIOUS STUDIES

2.1 Big Data

We now live in the era of the big data. In this era, the volume of data has exploded. The magnitude of data generated and shared by businesses, public administrations, numerous industrial sectors, not-for-profit sectors and scientific research has increased immeasurably [9]. These data include textual content (i.e. structured, semi-structured as well as unstructured) to multimedia content (e.g. videos, images, audio) on a multiplicity of platforms (e.g. machine-to-machine communications, social media sites, sensor networks, cyber-physical systems and Internet of Things [IoT]). Dobre and Xhafa [10] report that every day the world produces around 2.5 quintillion gigabytes of data (2.3 trillion gigabytes), with 90% of these data generated in the world being unstructured. It is asserted that by 2020, over 40 Zettabytes (or 40 trillion gigabytes) of data will be generated, imitated, and consumed. With this overwhelming amount of complex and heterogeneous data pouring from any-where, any-time and any-device there is undeniably an era of Big Data – a phenomenon also referred to as the Data Deluge. In essence, Big Data is the artifact of each human individual as well as collective intelligence generated and shared mainly through the technological environments where virtually anything and everything

can be documented, measured and captured digitally, and while doing that transformation into data – a process that Mayer-Schönberger and Cukier [11] also referred as datafication. Regardless of where Big Data is generated from and shared to, with the reality of Big Data come the challenges of analyzing it in a way that brings Big Value. Nevertheless, the growth of data in volumes in the digital world seems to out-speed the advancement of many extant computing infrastructures. The well-established data processing technologies, for example databases and data warehouses are becoming inadequate in front of the amount of data the world is going to generate. The massive amount of data needs to be analyzed in an iterative, as well as in a time sensitive manner. The ability to work with this massive scale of datasets is very critical. Traditional computing approaches with a single computer having a multicore processor to deal with some amount of data are not suitable for this massive scale datasets.

In the post-ImageNet [12] era, computer vision and machine learning researchers are solving more complicated AI problems using larger datasets which drives the demand for more computation. However, Moore’s Law is slowing down, Dennard scaling has stopped, and the amount of computation per unit cost and power is no longer increasing at its historic rate. This mismatch between supply and demand of computation highlights the need for co-designing efficient machine learning algorithms and domain-specific hardware architectures for massive scale datasets. The vast design space across algorithm and hardware is difficult to be explored by available engineered applications or tools. Therefore, we need different architectures with distributed workloads to bridge the gap.

2.2 TensorFlow

Created by the Google Brain team, TensorFlow [13] is an open source library for numerical computation and large-scale machine learning. TensorFlow bundles together a slew of machine learning and deep learning (aka neural networking) models and algorithms and makes them useful by way of a common metaphor. It uses Python to provide a convenient front-end API for building applications with the framework, while

executing those applications in high-performance C++. TensorFlow can train and run deep neural networks for handwritten digit classification, image recognition, word embeddings, sequence-to-sequence models for machine translation, natural language processing, and PDE (partial differential equation) based simulations. TensorFlow supports production prediction at scale, with the same models used for training.

2.2.1 Architecture

The TensorFlow is a cross-platform library. Figure 1 illustrates its general architecture. C API separates user level code in different languages from the core runtime.

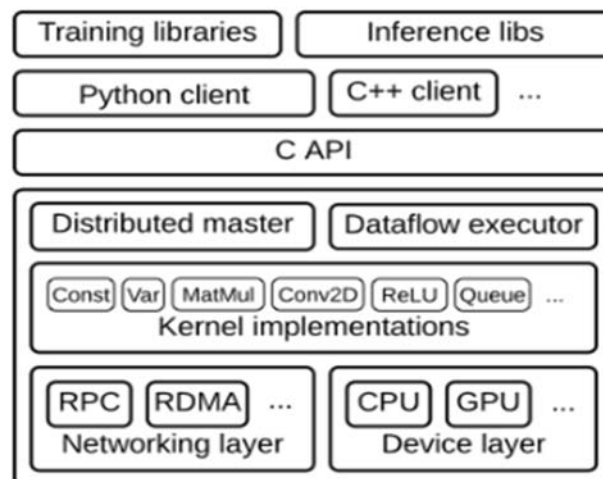


Figure .1 TensorFlow General Architecture

Client

- Defines the computation as a dataflow graph.
- Initiates graph execution using a session.

Distributed Master

- Prunes a specific subgraph from the graph, as defined by the arguments to `session.run()`.

- Partitions the subgraph into multiple pieces that run in different processes and devices.
- Distributes the graph pieces to worker services.
- Initiates graph pieces execution by worker services.

Worker Services (one for each task)

- Schedule the execution of graph operations using kernel implementations, appropriate to the available hardware (CPUs, GPUs, etc).
- Send and receive operation results to and from other worker services.

Kernel Implementations

- Perform the computation for individual graph operations.

2.3 Distributed TensorFlow

TensorFlow is designed for large-scale distributed training and inference, but it is also flexible enough to support small scale new machine learning models and system-level optimizations.

tf.distribute.Strategy is a TensorFlow API to distribute training across multiple GPUs, multiple machines or TPUs. Using this API, users can distribute their existing models and training code with minimal code changes.

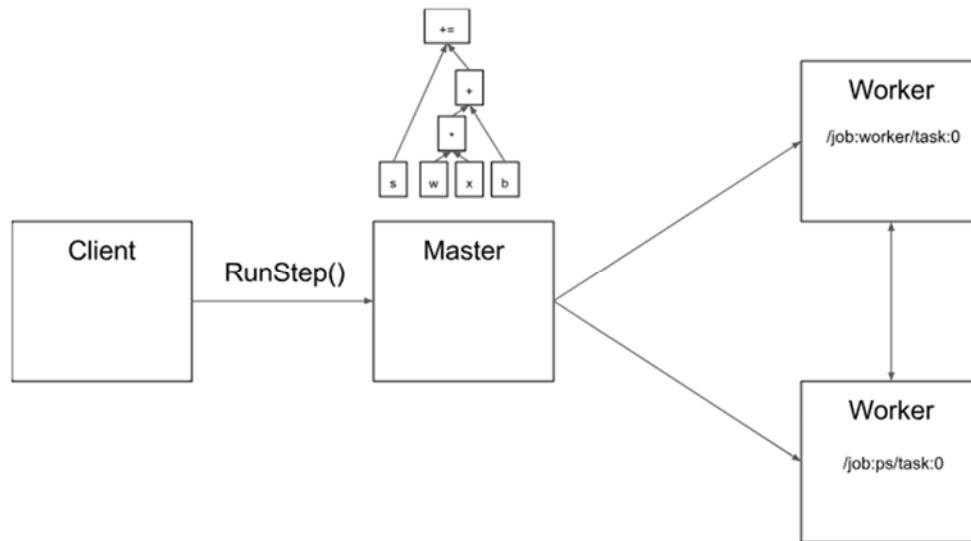


Figure. 2 TensorFlow Master Worker Model

Client

Users write the client TensorFlow program that builds the computation graph. This program can either directly compose individual operations or use a convenience library like the Estimators API to compose neural network layers and other higher-level abstractions. TensorFlow supports multiple client languages but prioritized Python and C++ for use. The client creates a session, which sends the graph definition to the distributed master as a `tf.GraphDef` protocol buffer. When the client evaluates a node or nodes in the graph, the evaluation triggers a call to the distributed master to initiate computation. In Figure 3, the client has built a graph that applies weights (w) to a feature vector (x), adds a bias term (b) and saves the result in a variable (s).

The Distributed Master

The master prunes the graph to obtain the subgraph required to evaluate the nodes requested by the client, then partitions the graph to obtain graph pieces for each participating device, and caches these pieces so that they may be re-used in subsequent steps.

Since the master sees the overall computation for a step, it applies standard optimizations

such as common subexpression elimination and constant folding. It then coordinates execution of the optimized subgraphs across a set of tasks.

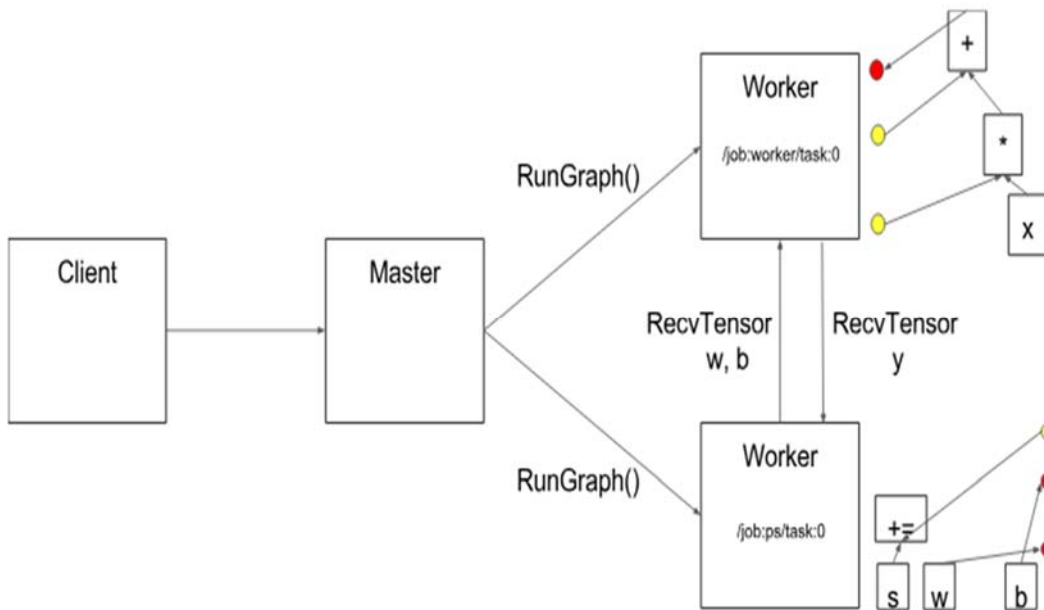


Figure. 3 Distributed Master workflow

Worker Service

The worker service in each task handles requests from the master, schedules the execution of the kernels for the operations that comprise a local subgraph, and mediates direct communication between tasks. TensorFlow optimize the worker service for running large graphs with low overhead. This current implementation can execute tens of thousands of subgraphs per second, which enables a large number of replicas to make rapid, fine-grained training steps. The worker service dispatches kernels to local devices and runs kernels in parallel when possible, for example by using multiple CPU cores or GPU streams.

TensorFlow specialize Send and Recv operations for each pair of source and destination device types. Transfers between local CPU and GPU devices use the `cudaMemcpyAsync()` API to overlap computation and data transfer. Transfers between two local GPUs use peer-to-peer DMA, to avoid an expensive copy via the host CPU. For this reason, working on GPUs using TensorFlow is much faster as compared to CPU. For transfers between tasks, TensorFlow uses multiple protocols, including: gRPC over TCP, RDMA over Converged Ethernet.

TensorFlow have preliminary support for NVIDIA's NCCL library for multi-GPU communication. The supported API is `tf.contrib.nccl`.

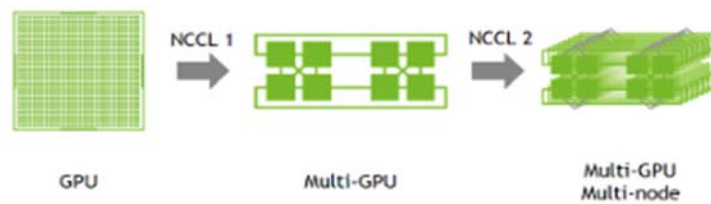


Figure. 4 Nvidia MultiGPU NCCL

The NVIDIA Collective Communications Library (NCCL) implements multi-GPU and multi-node collective communication primitives that are performance optimized for NVIDIA GPUs. NCCL provides routines such as all-gather, all-reduce, broadcast, reduce, reduce-scatter, that are optimized to achieve high bandwidth over PCIe and NVLink high-speed interconnect. In Figure 4, it reflects NCCL communication.

Kernel Implementations

The runtime contains over 200 standard operations including mathematical array manipulation, control flow and state management operations. Each of these operations

can have kernel implementations optimized for a variety of devices. In many of the operations, kernels are implemented using Eigen::Tensor, which uses C++ templates to generate efficient parallel code for multicore CPUs and GPUs; TensorFlow uses libraries like cuDNN where a more efficient kernel implementation is possible. TensorFlow implements quantization, which enables faster inference in environments such as mobile devices and high-throughput datacenter applications, and use the gemmlowp low-precision matrix library to accelerate quantized computation. (gemmlowp is a library for multiplying matrices whose entries are quantized as 8-bit integers. It is used in mobile neural network applications. It has received contributions from Intel and ARM, ensuring that it is efficient on various mobile CPUs).

If it is difficult or inefficient to represent a subcomputation as a composition of operations, users can register additional kernels that provide an efficient implementation written in C++. For better computation, TensorFlow recommends registering own fused kernels for some performance critical operations, such as the ReLU and Sigmoid activation functions and their corresponding gradients. The XLA Compiler has an experimental implementation of automatic kernel fusion.

TensorFlow provides eager execution mode for developers who need to debug and gain introspection into TensorFlow apps, which lets you evaluate and modify each graph operation separately and transparently, instead of constructing the entire graph as a single opaque object and evaluating it all at once. The TensorBoard visualization suite lets the developer inspect and customize the graphs by way of an interactive, web-based dashboard.

2.4 PyTorch

PyTorch [14] is a Python open source deep learning framework that was primarily developed by Facebook's artificial intelligence research group and was publicly introduced in January 2017.

Building Block #1: Tensors

PyTorch provides a basic data structure called a Tensor, which is very similar to NumPy's ndarray. But unlike the latter, tensors can tap into the resources of a GPU to significantly speed up matrix operations.

Building Block #2: Computation Graph

When a neural network is trained, researchers need to compute gradients of the loss function, with respect to every weight and bias, and then update these weights using gradient descent. With neural networks hitting billions of weights, doing the above step efficiently can make or break the feasibility of training.

In PyTorch, the computation graph is simply a data structure that allows to efficiently apply the chain rule to compute gradients for all of your parameters.

Building Block #3: Variables and Autograd

The Variable is just like a Tensor, is a class that is used to hold data. Variables are specifically tailored to hold values which change during training of a neural network, i.e. the learnable parameters of the network. Tensors on the other hand are used to store values that are not to be learned. For example, a Tensor maybe used to store the values of the loss generated by each example.

The graph is differentiated using the chain rule. If any of tensors are non-scalar (i.e. their data has more than one element) and require gradient, the function additionally requires specifying grad_tensor. It should be a sequence of matching length, which contains gradient of the differentiated function with respect to corresponding tensors.

Building Block #4: Function

PyTorch abstracts the need to write two separate functions (for forward, and for backward pass), into two member of functions of a single class called `torch.autograd.Function`.

PyTorch combines Variables and Functions to create a computation graph.

Dynamic Computation Graphs

A Dynamic Computational Graph framework is a system of libraries, interfaces, and components that provide a flexible, programmatic, run time interface that facilitates the construction and modification of systems by connecting operations. PyTorch creates the runtime dynamic computation graphs.

To qualify as a Dynamic Computational Graph framework, the framework must merely support the deferring of the determination of algorithm to run time, therefore opening the door to a plethora of operations on the computational dependencies and data flow at run time. The basics of the operations deferred must include the specification, manipulation, execution, and storage of the directed graphs that represent systems of operations.

The advantage of Dynamic Computational Graphs appears to include the ability to adapt to a varying quantities in input data. It seems like there may be automatic selection of the number of layers, the number of neurons in each layer, the activation function, and other neural network parameters, depending on each input set instance during the training.

2.5 MXNet

MXNet [15] is a deep Learning framework created by Apache, which supports a plethora of languages, like Python, Julia, C++, R, or JavaScript. It's been adopted by Microsoft, Intel, and Amazon Web Services.

The MXNet framework is known for its great scalability, which is used by large companies mainly for speech and handwriting recognition, NLP, and forecasting.

CHAPTER III

RELATED WORKS

3.1 Distributed GraphLab Framework

There are several distributed machine learning framework works available today. The high-level data parallel frameworks, like MapReduce, simplify the design and implementation of large-scale data processing systems, but they do not efficiently support many important data mining and machine learning algorithms and can lead to inefficient learning systems. To fill this critical void, the GraphLab abstraction is introduced which naturally expresses asynchronous, dynamic, graph-parallel computations while ensuring data consistency and achieving a high degree of parallel performance in the shared-memory setting. [16]

Turi is a graph-based, high performance, distributed computation framework written in C++. The GraphLab project was started by Prof. Carlos Guestrin of Carnegie Mellon University in 2009. It is an open source project using an Apache License. While GraphLab was originally developed for Machine Learning tasks, it has found great success at a broad range of other data-mining tasks; out-performing other abstractions by orders of magnitude. [17]

As the amounts of collected data and computing power grows (multicores, GPUs, clusters, clouds), modern datasets are no longer fit into one computing node. Efficient

distributed/parallel algorithms for handling large scale datasets are required. The GraphLab framework is a parallel programming abstraction targeted for sparse iterative graph algorithms. GraphLab provides a high level programming interface, allowing a rapid deployment of distributed machine learning algorithms. [18] The main design considerations behind the design of GraphLab are, sparse data with local dependencies, iterative algorithms, potentially asynchronous execution.

Main features of GraphLab are

- a unified multicore and distributed API, write once run efficiently in both shared and distributed memory systems
- It is tuned for performance by optimized C++ execution engine leverages extensive multi-threading and asynchronous IO
- Scalable, GraphLab intelligently places data and computation using sophisticated new algorithms
- HDFS Integration
- Powerful Machine Learning Toolkits

GraphLab framework is extended to the substantially more challenging distributed setting while preserving strong data consistency guarantee. The developed graph based extensions, used to pipelined locking and data versioning to reduce network congestion and mitigate the effect of network latency. The introduced fault tolerance in the GraphLab abstraction using the classic Chandy-Lamport snapshot algorithm demonstrate how easily it can be implemented by exploiting the GraphLab abstraction itself.

3.2 Parameter Server Framework

The parameter server is designed to simplify developing distributed machine learning applications as shown in Figure. 5 [8]. An instance of the parameter server can run more than one algorithm simultaneously. Parameter server nodes are grouped into a server group and several worker groups as shown in Figure 5. A server node in the server group

maintains a partition of the globally shared parameters. Server nodes communicate with each other to replicate and/or to migrate parameters for reliability and scaling. A server manager node maintains a consistent view of the metadata of the servers, such as node liveness and the assignment of parameter partitions. Each worker group runs an application. A worker typically stores locally a portion of the training data to compute local statistics such as gradients. Workers communicate only with the server nodes (not among themselves), updating and retrieving the shared parameters. There is a scheduler node for each worker group. It assigns tasks to workers and monitors their progress. If workers are added or removed, it reschedules unfinished tasks.

The parameter server supports independent parameter namespaces. This allows a worker group to isolate its set of shared parameters from others. Several worker groups may also share the same namespace: we may use more than one worker group to solve the same deep learning application [19] to increase parallelization. Another example is that of a model being actively queried by some nodes, such as online services consuming this model. Simultaneously the model is updated by a different group of worker nodes as new training data arrives.

The shared parameters are presented as (key, value) vectors to facilitate linear algebra operations. They are distributed across a group of server nodes. Any node can both push out its local parameters and pull parameters from remote nodes. By default, workloads, or tasks, are executed by worker nodes; however, they can also be assigned to server nodes via user defined functions. Tasks are asynchronous and run in parallel. The parameter server provides the algorithm designer with flexibility in choosing a consistency model via the task dependency graph and predicates to communicate a subset of parameters.

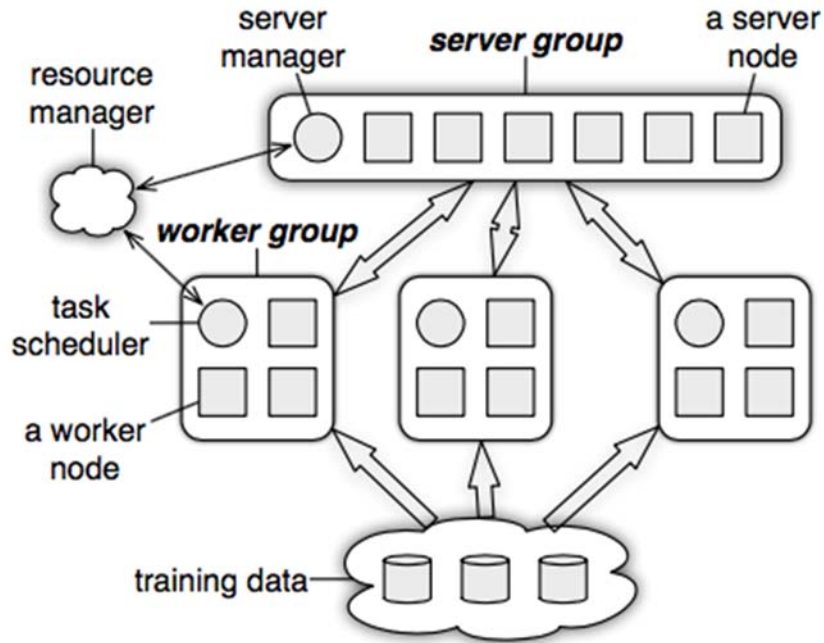


Figure. 5 Parameter Server Framework

3.2.1 Distributed Synchronous Stochastic Gradient Descent

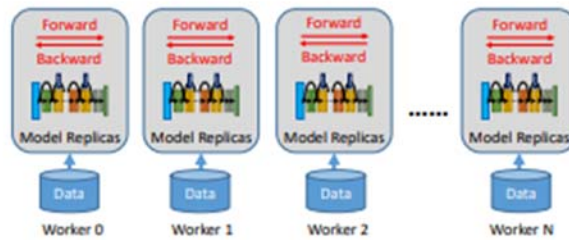


Figure. 6 Distributed SGD

In Figure. 6 each node independently calculates gradients by worker nodes.

In real time scenario, each training node performs the forward-backward pass on different batches sampled from the training dataset with the same network model. The gradients from all nodes are summed up to optimize their models. By this synchronization step, models on different nodes are always the same during the training. The aggregation step can be achieved in two ways. One method is using the parameter servers as the intermediary which store the parameters among several servers [20]. The nodes push the gradients to the servers while the servers are waiting for the gradients from all nodes. Once all gradients are sent, the servers update the parameters, and then all nodes pull the latest parameters from the servers.

One major disadvantage is network bandwidth. Large-scale distributed training improves the productivity of training deeper and larger models (Chilimbi et al., 2014; Xing et al., 2015; Moritz et al., 2015; Zinkevich et al., 2010). Synchronous stochastic gradient descent (SGD) is widely used for distributed training. By increasing the number of training nodes and taking advantage of data parallelism, the total computation time of the forward-backward passes on the same size training data can be dramatically reduced. However, gradient exchange is costly and dwarfs the savings of computation time (Li et al., 2014; Wen et al., 2017), especially for recurrent neural networks (RNN) where the computation-to-communication ratio is low. Therefore, the network bandwidth becomes a significant bottleneck for scaling up distributed training. [21]

Algorithm 1. Distributed Subgradient Descent

Task Scheduler:

- 1: issue LoadData() to all workers
- 2: **for** iteration $t = 0, \dots, T$ **do**
- 3: issue WORKERITERATE(t) to all workers.
- 4: **end for**

Worker $r = 1, \dots, m$:

- 1: **function** LOADDATA()
- 2: load a part of training data $\{y_{i_k}, x_{i_k}\}_{k=1}^{n_r}$
- 3: pull the working set $w_r^{(0)}$ from servers
- 4: **end function**
- 5: **function** WORKERITERATE(t)
- 6: gradient $g_r^{(t)} \leftarrow \sum_{k=1}^{n_r} \partial \ell(x_{i_k}, y_{i_k}, w_r^{(t)})$
- 7: push $g_r^{(t)}$ to servers
- 8: pull $w_r^{(t+1)}$ from servers
- 9: **end function**

Servers:

- 1: **function** SERVERITERATE(t)
- 2: aggregate $g^{(t)} \leftarrow \sum_{r=1}^m g_r^{(t)}$
- 3: $w^{(t+1)} \leftarrow w^{(t)} - \eta (g^{(t)} + \partial \Omega(w^{(t)}))$
- 4: **end function**

As shown in Algorithm 1, the training data is partitioned among all the workers, which jointly learn the parameter vector w . Because each worker works independently, the system uses a mechanism by expressing the updates as a subgradient—a direction in which the parameter vector w should be shifted and aggregates all subgradients before applying them to w . Data is sent between nodes using push and pull operations. A task is issued by a remote procedure call. It can be a push or a pull that a worker issues to servers. It can also be a user-defined function that the scheduler issues to any node. Tasks may include any number of subtasks. Tasks are executed asynchronously. In Algorithm

1, a worker pushes its temporary local gradient g to the parameter server for aggregation. The most expensive step in Algorithm 1 is computing the subgradient to update w . This task is divided among all of the workers, each of which execute `WORKERITERATE`. The task `WORKERITERATE` in Algorithm 1 contains one push and one pull. In Algorithm 1 each worker pushes its entire local gradient into the servers, and then pulls the updated weight back. The aggregation logic in `SERVERITERATE` updates the weight w only after all worker gradients have been aggregated.

3.3 Deep Gradient Compression

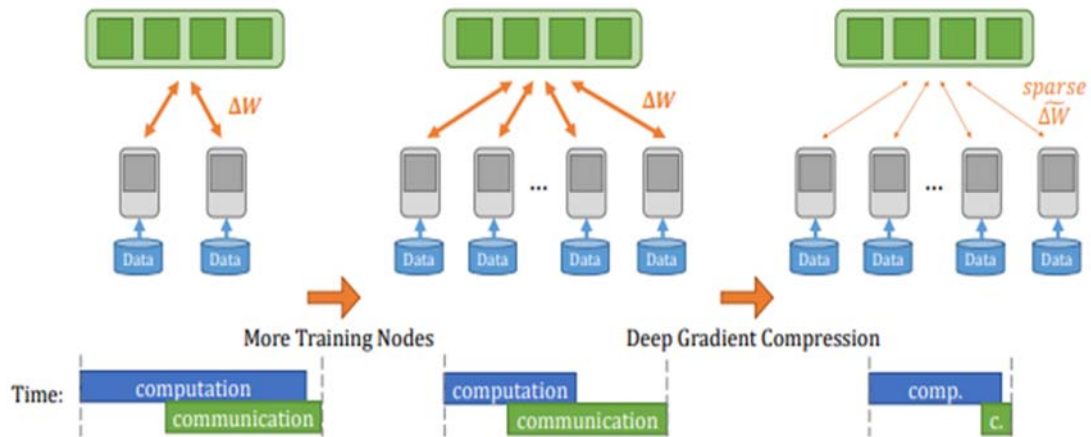


Figure. 7 Deep Gradient Compression

Deep Gradient Compression (DGC) solves the communication bandwidth problem by compressing the gradients, as shown in Figure 8. To ensure no loss of accuracy, DGC employs momentum correction and local gradient clipping on top of the gradient sparsification to maintain model performance. DGC also uses momentum factor masking and warmup training to overcome the staleness problem caused by reduced communication. [21]

Techniques in Deep Gradient Compression

Gradient Sparsification

Reduce the communication bandwidth by sending only the important gradients.

User the gradient magnitude as a simple heuristics for importance.

Only gradients larger than a threshold are transmitted which is top 0.01%.

Local Gradient Accumulation Gradient Accumulation algorithms represent an important component of distributed training systems. These algorithms are responsible for accumulating the local gradients from each worker node and distributing the updated global gradients back to the worker nodes. The All Reduce algorithm makes for a very good fit for this functionality and also removes the need for a master server by espousing a peer to peer paradigm for data exchange.

Local Gradient Clipping Gradient clipping is widely adopted to avoid the exploding gradient problem [22]. The method proposed by Pascanu et al. (2013) rescales the gradients whenever the sum of their L2-norms exceeds a threshold. This step is conventionally executed after gradient aggregation from all nodes. The accumulation of gradients over iterations on each node can be performed independently, where the gradient clipping is performed locally before adding the current gradient G_t to previous accumulation (G_{t-1} in Algorithm 2) [23].

Momentum Factor Masking Mitliagkas et al. (2016) discussed the staleness caused by asynchrony and described it as implicit momentum. Inspired by that, it introduce momentum factor masking, to alleviate staleness. Instead of searching for a new momentum coefficient as suggested in Mitliagkas et al. (2016) [24], it simply apply the same mask to the accumulated gradients. This mask stops the momentum for delayed gradients, preventing the stale momentum from carrying the weights in the wrong direction.

Algorithm 2. All-reduce Algorithm with local gradient clipping

Input: Dataset χ
Input: minibatch size b per node
Input: the number of nodes N
Input: Optimization Function SGD
Input: Init parameters $w = \{w[0], \dots, w[M]\}$

- 1: **for** $t = 0, 1, \dots$ **do**
- 2: $G_t^k \leftarrow 0$
- 3: **for** $i = 1, \dots, B$ **do**
- 4: Sample data x from χ
- 5: $G_t^k \leftarrow G_t^k + \frac{1}{Nb} \nabla f(x; w_t)$
- 6: **end for**
- 7: All-reduce $G_t^k : G_t \leftarrow \sum_{k=1}^N G_t^k$
- 8: $w_{t+1} \leftarrow SGD(w_t, G_t)$
- 9: **end for**

When training the recurrent neural network with gradient clipping, gradient clipping is performed locally before adding the current gradient G_t^k to previous accumulation G_{t-1}^k in Algorithm 2.

CHAPTER IV

HUMAN ACTIVITY RECOGNITION

USING LSTM

4.1 HUMAN ACTIVITY RECOGNITION

Human Activity Recognition (HAR) is a broad field of study concerned with an ability to interpret human body gesture or motion via sensors and determine human activity or action [25]. Most of the human daily tasks can be simplified or automated if they can be recognized via HAR system. Typically, HAR system can be either supervised or unsupervised [26]. A supervised HAR system requires some prior training with dedicated datasets while unsupervised HAR system is being configured with a set of rules during development. HAR is considered as an important component in various scientific research contexts i.e. surveillance, healthcare and human computer interaction (HCI) However, it remains a very complex task, due to unsolvable challenges such as sensor motion, sensor placement, cluttered background, and inherent variability in the way activities are conducted by different human. HAR covers three area of sensing technologies namely RGB cameras, depth sensors and wearable devices. The popularity of depth sensors and wearable devices in HAR research is well established.

4.1.1. Surveillance System

In surveillance context, HAR was adopted in surveillance systems installed at public places i.e. shopping malls or airports, which introduced a new paradigm of human activity prediction to prevent crimes and dangerous activities from occurring at public places. Lasecki et al. proposed a system that provides robust, deploy-able activity recognition by supplementing existing recognition systems with on-demand, real-time activity identification using inputs from the crowds at public places [27].

4.1.2. Healthcare

In the field of Healthcare, HAR is employed in healthcare systems which are installed in residential environment, hospitals and rehabilitation centers. HAR is used widely for monitoring the activities of elderly people staying in rehabilitation centers for chronic disease management and disease prevention [28]. HAR is also integrated into smart homes for tracking the elderly people's daily activities [29]. Besides, HAR is used to encourage physical exercises in rehabilitation centers for children with motor disabilities [30], post-stroke motor patients, patients with dysfunction and psycho motor slowing, and exergaming [31]. Other than that, the HAR is adopted in monitoring patients at home such as estimation of energy expenditure to aid in obesity prevention and treatment and life logging. HAR is also applied in monitoring other behaviors such as stereotypical motion conditions in children with Autism Spectrum Disorders (ASD) at home, abnormal conditions for cardiac patients and detection of early signs of illness. Other healthcare related HAR solutions such as fall detection and intervention for elderly people are available [32].

4.1.3. Human Computer Interaction

In the field of human computer interaction, HAR has been applied quite commonly in gaming and exergaming such as Kinect, Nitendo Wii and full-body motion based games for older adults and adults with neurological injury [33]. Through HAR, human body gestures are recognized to instruct the machine to complete dedicated tasks. Elderly people and adults with neurological injury can perform a simple gesture to interact with games and exergames easily. HAR also enables surgeons to have intangible control of the intraoperative image monitor by using standardized free-hand movements [34].

4.1.4 HAR Sensing Technologies

Recognizing human activity using RGB camera is simple but having low efficiency. A RGB camera is usually attached to the environment and the HAR system will process image sequences captured with the camera. Most of the conventional HAR systems using this sensing technology are built with two major components which is the feature extraction and classification [35]. Besides, most of the RGB-HAR systems are considered as supervised system where trainings are usually needed prior to actual use. Image sequences and names of human activities are fed into the system during training stage. Real time captured image sequence are passed to the system for analysis and classification by dedicated computational/classification algorithms such as Support Vector Machine (SVM).

The depth sensor also known as infrared sensor or infrared camera is adopted into HAR systems for recognizing human activities. The depth sensor projects infrared beams into the scene and recapture them using its infrared sensor to calculate and measure the depth or distance for each beam from the sensor. The reviews found that Microsoft Kinect sensor is commonly adopted as depth sensor in HAR [33]. Since the Kinect sensor

has the capability to detect 20 human body joints with its real-world coordinate, many researchers utilized the coordinates for human activity classification.

HAR using wearable-based requires single or multiple sensors to be attached to the human body. Most commonly used sensor includes 3D-axial accelerometer, magnetometer, gyroscope and RFID tag. With the advancement of current smart phone technologies, many research works use mobile phone as sensing devices because most smart phones are equipped with accelerometer, magnetometer and gyroscope [36]. A physical human activity can be identify easily through analyzing the data generated from various wearable sensing after being process and determine by classification algorithm.

4.2 Dataset (UCI Repository)

4.2.1 Data Set Information

The dataset named “Human Activity Recognition Using Smartphones Data Set” [37] is used from UCI repository in this thesis. The experiments have been carried out with a group of 30 volunteers within an age bracket of 19-48 years. Each person performed six activities (WALKING, WALKING_UPSTAIRS, WALKING_DOWNSTAIRS, SITTING, STANDING, LAYING) wearing a smartphone (Samsung Galaxy S II) on the waist. Using its embedded accelerometer and gyroscope, they captured 3-axial linear acceleration and 3-axial angular velocity at a constant rate of 50Hz. The experiments have been video-recorded to label the data manually. The obtained dataset has been randomly partitioned into two sets, where 70% of the volunteers was selected for generating the training data and 30% the test data.

The sensor signals (accelerometer and gyroscope) were pre-processed by applying noise filters and then sampled in fixed-width sliding windows of 2.56 sec and 50% overlap (128 readings/window). The sensor acceleration signal, which has gravitational and body motion components, was separated using a Butterworth low-pass filter into body

acceleration and gravity. The gravitational force is assumed to have only low frequency components, therefore a filter with 0.3 Hz cutoff frequency was used. From each window, a vector of features was obtained by calculating variables from the time and frequency domain.

4.2.2 Attribute Information

For each record in the dataset it is provided:

- Triaxial acceleration from the accelerometer (total acceleration) and the estimated body acceleration.
- Triaxial Angular velocity from the gyroscope.
- A 561-feature vector with time and frequency domain variables.
- Its activity label.
- An identifier of the subject who carried out the experiment.

4.2.3 Feature Notes

- Features are normalized and bounded within [-1, 1].
- Each feature vector is a row on the text file.
- The units used for the accelerations (total and body) are 'g's (gravity of earth → 9.80665 m/sec²).
- The gyroscope units are rad/sec.

The file structure inside dataset are described in Table 1.

File name	Information
activity_labels.txt	Links the class labels with their activity name.
features_info.txt	Shows information about the variables used on the feature vector.
features.txt	List of all features.
README.txt	Information about dataset details
test/X_test.txt	Test set
test/y_test.txt	Test labels
train/X_train.txt	Training set
train/y_train.txt	Training labels
Inertial Signals/body_acc_x_train.txt Inertial Signals/body_acc_y_train.txt Inertial Signals/body_acc_z_train.txt	The body acceleration signal obtained by subtracting the gravity from the total acceleration. Every row shows a 128 element vector. The same description applies for the 'body_acc_y_train .txt' and 'body_acc_z_train .txt' files for the Y and Z axis.
Inertial Signals/body_gyro_x_train.txt Inertial Signals/body_gyro_y_train.txt Inertial Signals/body_gyro_z_train.txt	The angular velocity vector measured by the gyroscope for each window sample. The units are radians/second. Every row shows a 128 element vector. The same

	description applies for the 'body_gyro_y_train.txt' and 'body_gyro_z_train.txt' files for the Y and Z axis.
Inertial Signals/total_acc_x_train.txt Inertial Signals/total_acc_y_train.txt Inertial Signals/total_acc_z_train.txt	The acceleration signal from the smartphone accelerometer X axis in standard gravity units 'g'. Every row shows a 128 element vector. The same description applies for the 'total_acc_y_train.txt' and 'total_acc_z_train.txt' files for the Y and Z axis.

Table. 1 Dataset Feature Parameters

The Figure. 8 shows the hierarchy of the file structure inside dataset.

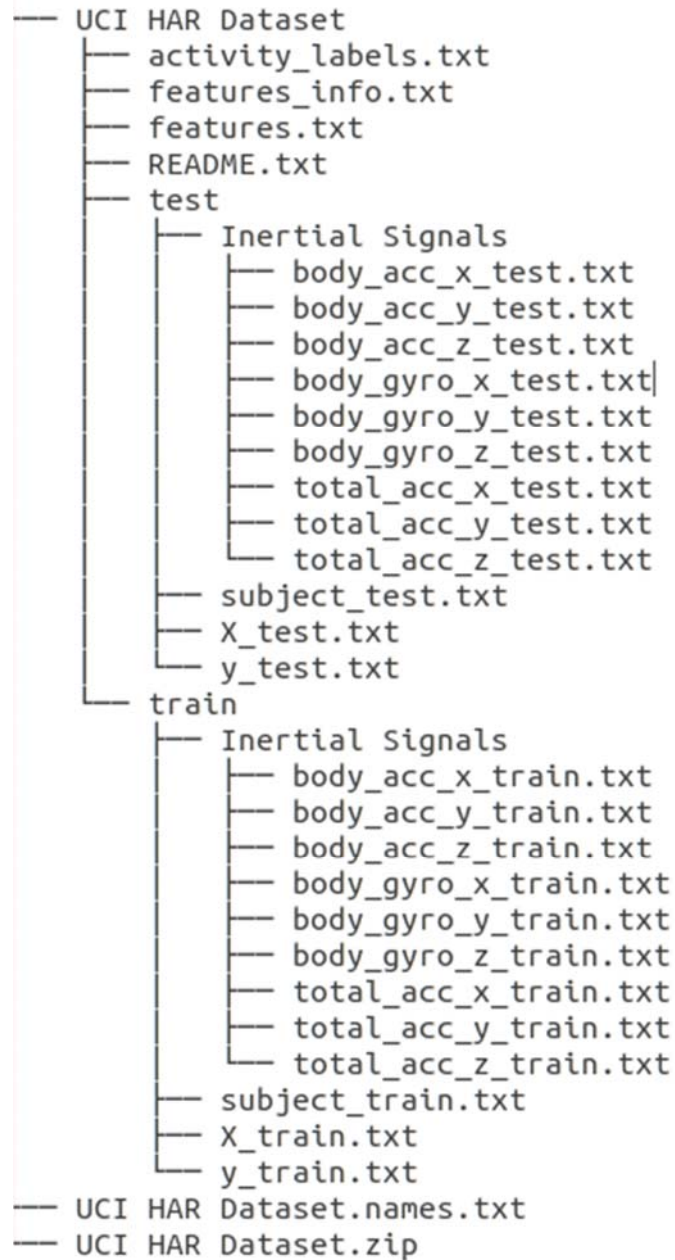


Figure. 8 Dataset File Structure

4.3 LSTM

LSTM network was proposed by Jürgen Schmidhuber in 1997 [4], is a variant of recurrent neural networks (RNNs). It has special inner gates that allow for consistently better performance than RNN for time series. Compared with other networks, such as CNN, restricted Boltzmann machine (RBM) and auto-encoder (AE), the structure of the LSTM renders it especially good at solving problems involving time series, such as those related to natural language processing, speech recognition, and weather prediction, because its design enables gradients to flow through time readily.

4.3.1 Why LSTM?

4.3.1.1 CNN

The basic difference between a feed forward neuron and a recurrent neuron is shown in Figure 9.

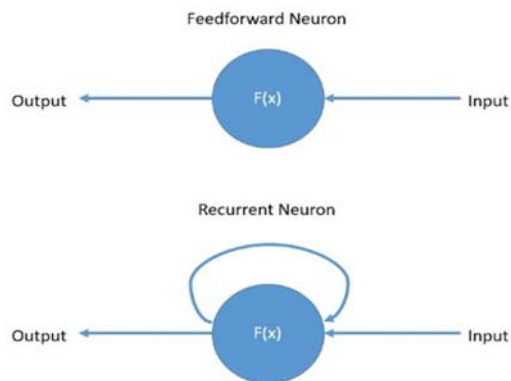


Figure. 9 Basic Feed-Forward and Recurrent cell

The feed forward neuron has two weights which connects from his input to his output. The recurrent neuron has also a connection from his output again to his input and therefore it has three weights. When many feed-forward layers are connected together, they form a Convolutional Neural Network (CNN). This third extra connection is called feed-back connection and with that the activation can flow round in a loop.

When many feed forward and recurrent neurons are connected, they form a Recurrent Neural Network (RNN). The major difference between CNN and RNN is that CNN is a feed-forward neural network, while RNN is a recurrent neural network. In CNN, the information only flows in the forward direction, while in RNN, the information flows back and forth.

In mathematics, a convolution is a grouping function. In CNNs, convolution happens between two matrices (rectangular arrays of numbers arranged in columns and rows) to form a third matrix as an output. A CNN uses these convolutions in the convolutional layers to filter input data and find information.

The University of Toronto researchers Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton trained a deep convolutional neural network to classify the 1.2 million images from the ImageNet Large Scale Visual Recognition Challenge contest, winning with a record-breaking reduction in error rate [12]. This sparked today's modern AI boom.

The convolutional layer does most of the computational works in a CNN. It acts as the mathematical filters that help computers find edges of images, dark and light areas, colors, and other details, such as height, width and depth.

There are usually many convolutional layer filters applied to an image.

Pooling layer: Pooling layers are often sandwiched between the convolutional layers. They're used to reduce the size of the representations created by CNN and reduce the memory requirements, which allows for more convolutional layers.

Normalization layer: Normalization is a technique used to improve the performance and stability of neural networks. There are different types of normalization available in CNN. Those are Weight Normalization [38], Layer Normalization [39], and Batch Normalization [40].

Fully connected layers: Fully connected layers connect every neuron in one layer to every neuron in another layer. It is using the same principle as the traditional multi layer perceptron neural network (MLP). The flattened matrix goes through a fully connected layer to classify the images.

Then the back propagation is used to calculate the gradients of error with respect to all the weights in the network. Back propagation is the method by which a neural network is trained. It doesn't have much to do with the structure of the network, but rather implies how input weights are updated. When training a feed forward network, the information is passed into the network, and the resulting classification is compared to the known training sample. If the network's classification is incorrect, the weights are adjusted backward through the network in the direction that would give it the correct classification. This is called the backward propagation of the training. So CNN is a feed-forward network, but is trained through back-propagation.

CNNs are ideally suited for computer vision, but feeding those enough data can make them useful in videos, speech, music and text as well.

4.3.1.2 Back Propagation

Algorithm 3. Back Propagation algorithm.

Consider a network with a single real input x and network function P . The derivative $P'(x)$ is computed in two phases: (1) Feed-forward: the input x is fed into the network. The primitive functions at the nodes and their derivatives are evaluated at each node. The derivatives are stored. (2) Back propagation: the constant 1 is fed into the output unit and the network is run backwards. Incoming information to a node is added and the result is

multiplied by the value stored in the left part of the unit. The result is transmitted to the left of the unit. The result collected at the input unit is the derivative of the network function with respect to x .

Back propagation is based around four fundamental equations. Together, those equations give us a way of computing both the error δ^l and the gradient of the cost function. The four equations are shown below [41].

An equation for the error in the output layer, δ^L : The components of δ^L are given by

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L). \quad (\text{BP1})$$

This is a very natural expression. The first term on the right, $\partial C / \partial a_j^L$, just measures how fast the cost is changing as a function of the j^{th} output activation. If, for example, C doesn't depend much on a particular output neuron, j , then δ_j^L will be small, which is as expected. The second term on the right, $\sigma'(z_j^L)$, measures how fast the activation function σ is changing at z_j^L .

An equation for the error δ^l in terms of the error in the next layer, δ^{l+1} :

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \quad (\text{BP2})$$

where $(w^{l+1})^T$ is the transpose of the weight matrix w^{l+1} for the $(l+1)^{\text{th}}$ layer. When we apply the transpose weight matrix $(w^{l+1})^T$, we can think of it as moving the error backward through the network, which gives some sort of measure of the error at the output of the l^{th} layer. This moves the error backward through the activation function in layer l , which gives us the error δ^l in the weighted input to layer l .

By combining (BP2) with (BP1), the error δ^l can be computed for any layer in the network.

An equation for the rate of change of the cost with respect to any bias in the network:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \quad (\text{BP3})$$

The error δ_j^l is exactly equal to the rate of change $\partial C / \partial b_j^l$, which is the same as, calculating error by (BP1) and (BP2) to compute δ_j^l . We can rewrite (BP3) as $\partial C / \partial b = \delta$, where δ is being evaluated at the same neuron as the bias b .

An equation for the rate of change of the cost with respect to any weight in the network:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l. \quad (\text{BP4})$$

This gives us to compute the partial derivatives $\partial C / \partial w_{jk}^l$ in terms of the quantities δ^l and a^{l-1} . The equation can be rewritten as

$$\frac{\partial C}{\partial w} = a_{\text{in}} \delta_{\text{out}},$$

where it's shown that, a_{in} is the activation of the neuron input to the weight w , and δ_{out} is the error of the neuron output from the weight w .

If we look at the weight w , and the two neurons connected by that weight, we can depict this as:

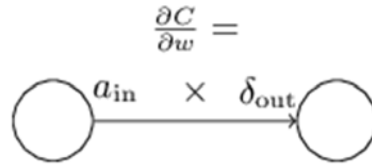


Figure. 10 Two Connected Neurons with weights

The above back propagation rules are summarized in Figure 11.

Input x : Set the corresponding activation a^1 for the input layer.

Feedforward: For each $l = 2, 3, \dots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.

Output error δ^L : Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.

Backpropagate the error: For each $l = L - 1, L - 2, \dots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.

Output: The gradient of the cost function is given by

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \text{ and } \frac{\partial C}{\partial b_j^l} = \delta_j^l.$$

Figure.11 Back Propagation Rule

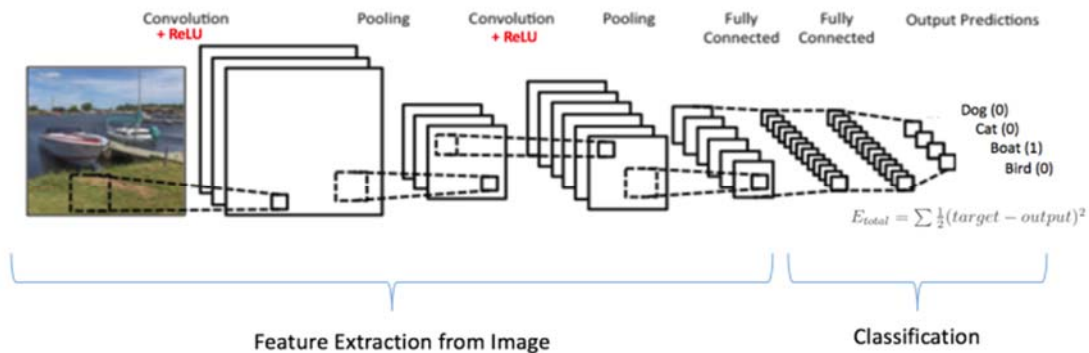


Figure. 12 Convolution Neural Network

The overall training process of the Convolution Network may be summarized as below:

Step1: We initialize all filters and parameters / weights with random values

Step2: The network takes a training image as input, goes through the forward propagation step (convolution, ReLU and pooling operations along with forward propagation in the Fully Connected layer) and finds the output probabilities for each class.

- Let's say the output probabilities for the boat image above are [0.2, 0.4, 0.1, 0.3]
- Since weights are randomly assigned for the first training example, output probabilities are also random.

Step3: Calculate the total error at the output layer (summation over all 4 classes)

$$\text{Total Error} = \sum \frac{1}{2}(\text{target probability} - \text{output probability})^2$$

Step4: Use Backpropagation to calculate the gradients of the error with respect to all weights in the network and use gradient descent to update all filter values / weights and parameter values to minimize the output error.

- The weights are adjusted in proportion to their contribution to the total error.
- When the same image is input again, output probabilities might now be $[0.1, 0.1, 0.7, 0.1]$, which is closer to the target vector $[0, 0, 1, 0]$.
- This means that the network has learnt to classify this particular image correctly by adjusting its weights / filters such that the output error is reduced.
- Parameters like number of filters, filter sizes, architecture of the network etc. have all been fixed before Step 1 and do not change during training process – only the values of the filter matrix and connection weights get updated.

Step5: Repeat steps 2-4 with all images in the training set.

The CNN have now been optimized to correctly classify images from the training set.

4.3.1.3 RNN

The major limitation of CNN is that they accept a fixed-sized vector as input and produce a fixed-sized vector as output which is the probabilities of different classes. Then these models perform the mapping using a fixed amount of computational steps or the number of layers in the model. They are enlisted as giant sequence of filters or neurons in these hidden layers that all optimize toward efficiency in identifying an image. Therefore, CNNs are called “feed-forward” neural networks because information is fed from one layer to the next. However, RNN is trained to recognize patterns across time, while a CNN learns to recognize patterns across space and hence a CNN learns to recognize components in an image like lines, edges, curves, etc.

RNN offers two major advantages:

Store Information

The recurrent network can use the feedback connection to store information over time in form of activations. This ability is significant for many applications. In the recurrent networks, they have some form of memory.

Learn Sequential Data

The main reason for using RNN, they allow us to operate over sequences of vectors. In Figure. 13, with RNN approach one to many, many to one and many to many inputs to outputs are possible.

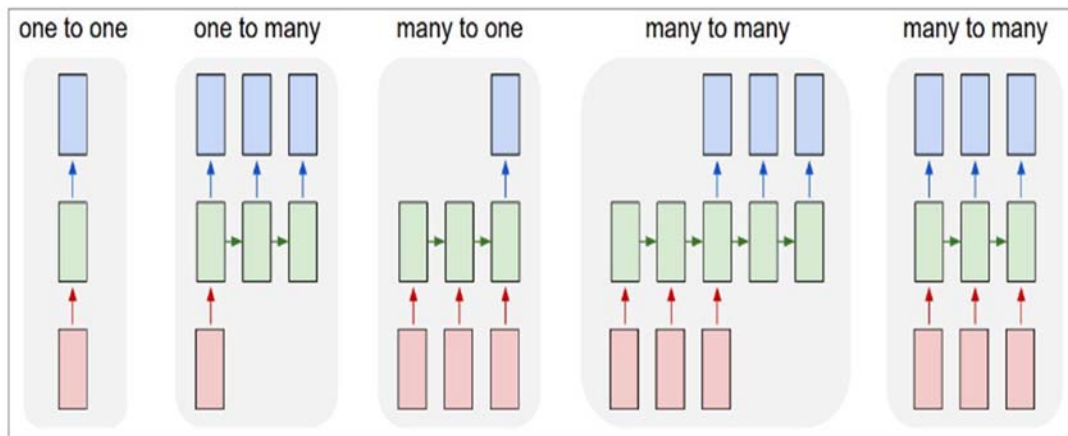


Figure. 13 RNN Sequential Data Learning Approach

In Figure. 13, each rectangle is a vector and arrows represent functions (e.g. matrix multiply). Input vectors are in red, output vectors are in blue and green vectors hold the RNN's state. The RNN can handle sequential data of arbitrary length. From left to right as shown in Figure 11: (1) On the left the default feed forward CNN is shown, which can just compute from fixed-sized input to fixed-sized output (e.g. image classification). (2)

Sequence output (e.g. image captioning takes an image and outputs a sentence of words). (3) Sequence input (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment). (4) Sequence input and sequence output (e.g. Machine Translation: an RNN reads a sentence in English and then outputs a sentence in French). (5) Synced sequence input and output (e.g. video classification where we wish to label each frame of the video).

Notice that in every case, there are no pre-specified constraints on the length sequences because the recurrent transformation (green) is fixed and can be applied as many times as required.

Recurrent neural networks (RNNs) are connectionist models that capture the dynamics of sequences via cycles in the network of nodes. Unlike standard CNNs, RNNs retain a state that can represent information from an arbitrarily long context window. RNNs combine the input vector with their state vector with a fixed (but learned) function to produce a new state vector. All recurrent neural networks have the form of a chain of repeating modules of neural network as shown in Figure 14. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.

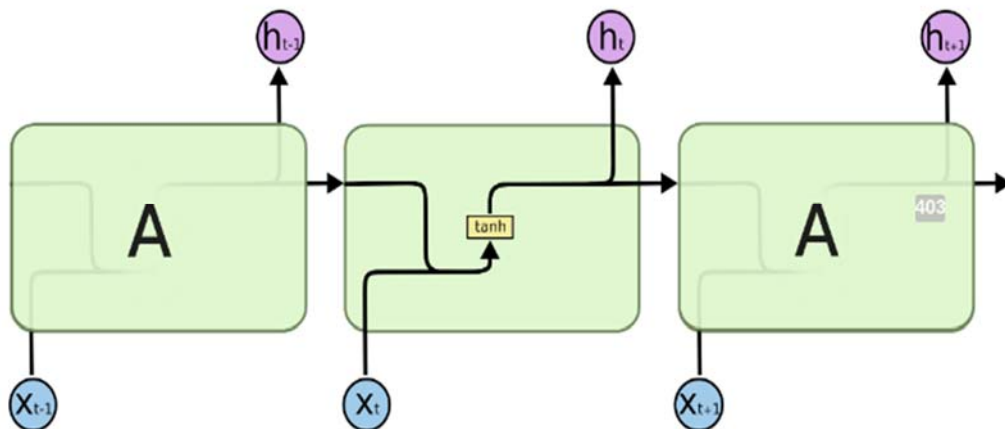


Figure. 14 Simple RNN Structure

Computational Power of Recurrent Networks

From the point of view of automata theory, all that is relevant is the identification of a set of internal states which characterize the status of the device at a given moment in time, together with the specification of rules of operation which predict the next state on the basis of the current state and the inputs from the environment [42].

Theorem 1: Rational-weighted RNNs having boolean activation functions (simple thresholds) are equivalent to finite state automata [43].

Proof: Proof shown in [43]

Theorem 2: Rational-weighted RNNs having linear sigmoid activation functions are equivalent to Turing Machines [44].

Proof: Proof shown in [44]

Theorem 3: Real-weighted RNNs having linear sigmoid activation functions are more powerful than Turing Machines. Siegelmann and Sontag noted that these networks are not likely to solve polynomially NP-hard problems, as the equality “ $P=NP$ ” in their model implies the almost complete collapse of the standard polynomial hierarchy [45].

Proof: Proof shown in [45]

Theorem 4:

All Turing machines may be simulated by fully connected recurrent networks built of neurons with sigmoidal activation functions [46].

In his model, all neurons synchronously update their states according to a quadratic combination of past activation values. Proof: Proof shown in [46]

Long-Term Dependencies Problems

What happened to Recurrent Networks? One major drawback of RNNs is that the range of contextual information is limited and the Backpropagation through time (BPTT) [47] does not store information over long time period. This is noticeable in either vanishing or exploding outputs of the network, which is known as vanishing gradient problem or exploding gradient problem [48].

These problems arise during training of a deep network when the gradients are being propagated back in time all the way to the initial layer. The gradients coming from the deeper layers have to go through continuous matrix multiplications because of the chain rule, and as they approach the earlier layers, if they have small values (<1), they shrink exponentially until they vanish and make it impossible for the model to learn, this is the vanishing gradient problem. While on the other hand if they have large values (>1) they get larger and eventually blow up and crash the model, this is the exploding gradient problem.

Dealing with Exploding Gradients

When gradients explode, it become NaN because of the numerical overflow, which results irregular oscillations in training cost when the learning curve is plotted. A solution to fix this is to apply gradient clipping; which places a predefined threshold on the gradients to prevent it from getting too large, and by doing so, it doesn't change the direction of the gradients but it only changes its length.

4.3.1.4 LSTM

What makes LSTM so desirable? For dealing with Vanishing Gradients, Long Short-Term Memory architecture (LSTM) is most popular and a widely used approach. This is a different variant of RNN which was designed to make it easy to capture long-term dependencies in sequence data. The standard RNN operates in such a way that the hidden state activations are influenced by the other local activations closest to them, which

corresponds to a “short-term memory”, while the network weights are influenced by the computations that take place over entire long sequences, which corresponds to a “long-term memory”. Hence the RNN was redesigned so that it has an activation state that can also act like weights and preserve information over long distances, hence the name “Long Short-Term Memory” [4].

4.3.1.5 Distributed LSTM

What is the need of distributed machine for LSTM? Recurrent neural networks (RNNs) have been widely used for processing sequential data. However, RNNs are commonly difficult to train due to the well-known gradient vanishing and exploding problems and hard to learn long-term patterns. Long short-term memory (LSTM) and gated recurrent unit (GRU) were developed to address these problems [49].

The LSTM architectures are usually trained in a batch setting in the architecture, where all data instances are present and processed together. However, for applications involving big data, storage issues may arise due to keeping all the data in one place. Additionally, in certain frameworks, all data instances are not available beforehand since instances are received in a sequential manner, which precludes batch training. As every second the data size is growing exponentially, in coming years most big corporations will suffer from computational power and storage issues due to large amount of data. As an example, in tweet emotion recognition applications, the systems are usually trained using an enormous amount of data to achieve sufficient performance, especially for agglutinative languages [50].

In the common distributed architectures, the whole data is distributed to different nodes but the trained parameters are merged later at a central node. However, this centralized approach requires high storage capacity and computational power at the central node. Additionally, centralized strategies have a potential risk of failure at the central node. To circumvent these issues, we distribute both the processing as well as the data to all the

nodes and allow communication only between neighboring nodes, hence, we remove the need for a central node. In particular, each node sequentially receives a variable length of data sequence with its label and exchanges information only with its neighboring nodes to train the common LSTM parameters. There are two approaches to achieve this architecture. By the use of parameter server framework, this scalable distributed deep learning approach can be achieved where both data and workloads are distributed over worker nodes, while the server nodes maintain global shared parameters, represented as dense or sparse vectors and matrices. Here the worker nodes process data and compute local gradients on a mini-batch. They then send push (key, gradient) messages to the servers. Those process the updates asynchronously. When needed, the workers pull them back with a pull (key) request. A lot of the infrastructure is borrowed from distributed (key, value) storage such as memcached. Memcached is a high-performance, distributed memory object caching system, generic in nature, but originally intended for use in speeding up dynamic web applications by alleviating database load[51]. The framework manages asynchronous data communications between nodes and supports flexible consistency models, elastic scalability and continuous fault tolerance[8].

The other approach is synchronous distributed stochastic gradient descent (SGD), which is known as distributed synchronous SGD. In practice, each training node performs the forward-backward pass on different batches sampled from the training dataset with the same network model. The gradients from all nodes are summed up to optimize their models. By this synchronous step, models of different nodes are always the same during the training. The aggregation step can be achieved by performing the All-reduce operation on the gradients among all nodes and to update the parameters on each node independently [52].

4.3.1.5.1 Synchronous all-reduce SGD

In traditional synchronous all-reduce SGD, there are two alternating phases proceeding in lock-step:(1) each node computes its local parameter gradients, and (2) all nodes

collectively communicate all-to-all to compute an aggregate gradient, as if they all formed a large distributed minibatch.

The second phase of exchanging gradients forms a barrier and is the communication-intensive phase, usually implemented by an eponymous all-reduce operation. The time complexity of an all-reduction can be decomposed into latency-bound and bandwidth-bound terms. Although the latency term scales with $O(\log(p))$, there are fast ring algorithms which have bandwidth term independent of p [52]. With modern networks capable of handling bandwidth on the order of 1–10 GB/s combined with neural network parameter sizes on the order of 10–100 MB, the communication of gradients or parameters between nodes across a network can be very fast.

Instead, the communication overhead of all-reduce results from its use of a synchronization barrier, where all nodes must wait for all other nodes until the all-reduce is complete before proceeding to the next stochastic gradient iteration. This directly leads to a straggler effect where the slowest nodes will prevent the rest of the nodes from making the progress. [53]

Algorithm 4. Synchronous all-reduce SGD

Initialize $\theta_{0,i} \leftarrow \theta_0$

for $t \in \{0 \dots T\}$ **do**

$$\Delta\theta_{t,i} \leftarrow -\alpha \nabla f_i(\theta_{t,i}; X_{t,i}) + \mu \Delta\theta_{t-1}$$

$$\Delta\theta_t \leftarrow \text{all-reduce-average}(\Delta\theta_{t,i})$$

$$\theta_{t+1,i} \leftarrow \theta_{t,i} + \Delta\theta_t$$

end for

4.3.2 Baseline LSTM

LSTM is an extension of recurrent neural networks. Due to its special architecture, which combats the vanishing and exploding gradient problems, it is good at handling time series problems up to a certain depth. The input gate, the forget gate, and the output gate of LSTM are designed to control what information should be forgotten, remembered, and updated.

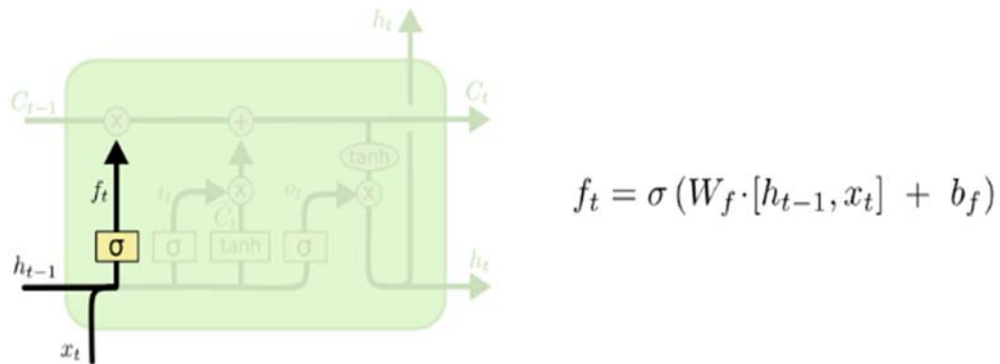
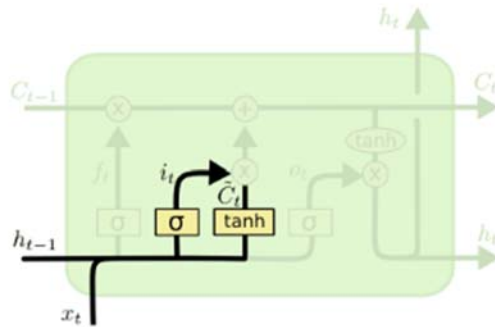


Figure. 15 LSTM Forget Gate

As shown in Figure. 15, First there is a need to forget old information, which involves the forget gate. In the first step of LSTM forget gate looks at h_{t-1} and x_t to compute the output f_t which is a number between 0 and 1 for each cell state number. This is multiplied by the cell state C_{t-1} and yield the cell to either forget everything or keep the information which is based on zero or one. For example a value of 0.5 means that the cell forgets 50% of its information. It is considered a good practice to initialize these gates to a value of 1, or close to 1, so as to not impair training at the beginning.

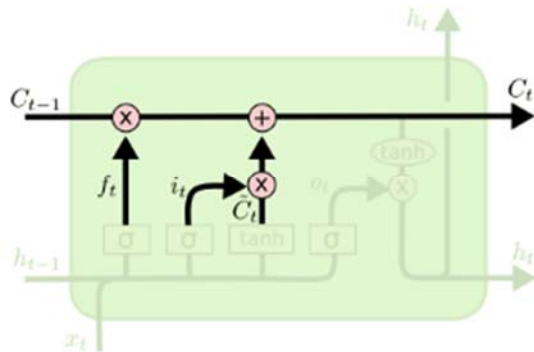


$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Figure. 16 LSTM Input Gate

As shown in Figure. 16, the next step is to determine what new information needs to keep in memory with an input gate. This has two parts. First, a sigmoid function called the “input gate” decides which values need to update. Next, a tanh function creates a vector of new candidate values, \tilde{C}_t , which could be added to the state. From that, it is possible to update the old cell state, to the new cell state, Gating is a method to selectively pass the needed information.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure. 17 LSTM Processing Data

As shown in Figure. 17, now LSTM will update the old cell state C_{t-1} , into the new cell state C_t . It multiply the old state by f_t , forgetting the things it decided to forget earlier. Then it adds $i_t * \tilde{C}_t$. This is the new candidate value, scaled by LSTM's decision to update each state value.

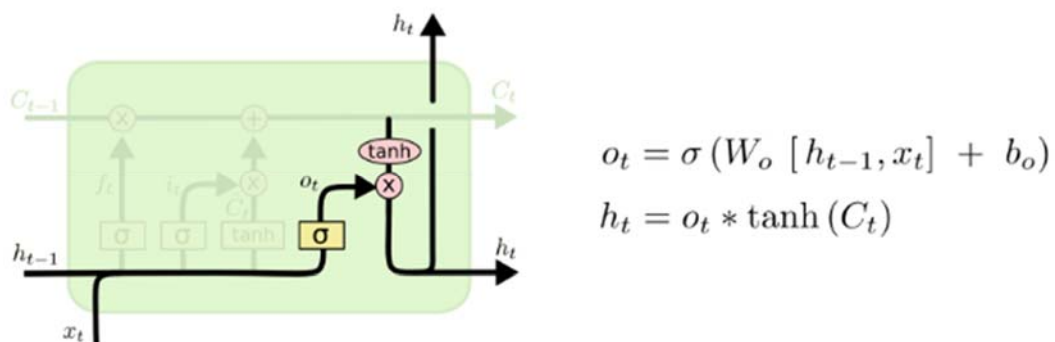


Figure. 18 LSTM Output Gate

Finally the output value has to be computed, which is done by multiplying o_t with the tanh of the result of the previous step, which yields to $h_t = o_t * \tanh(C_t)$ and $o_t = \sigma * (W_o [h_{t-1}, x_t] + b_o)$. Finally, it decides which information should be output to the layer above with an output gate.

In the LSTM cell, each parameter at moment t can be defined as follows:

$$f_t = \sigma(W_f [h_{t-1}, x_t] + b_f)$$

$$i_t = \sigma(W_i [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_c [h_{t-1}, x_t] + b_c)$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

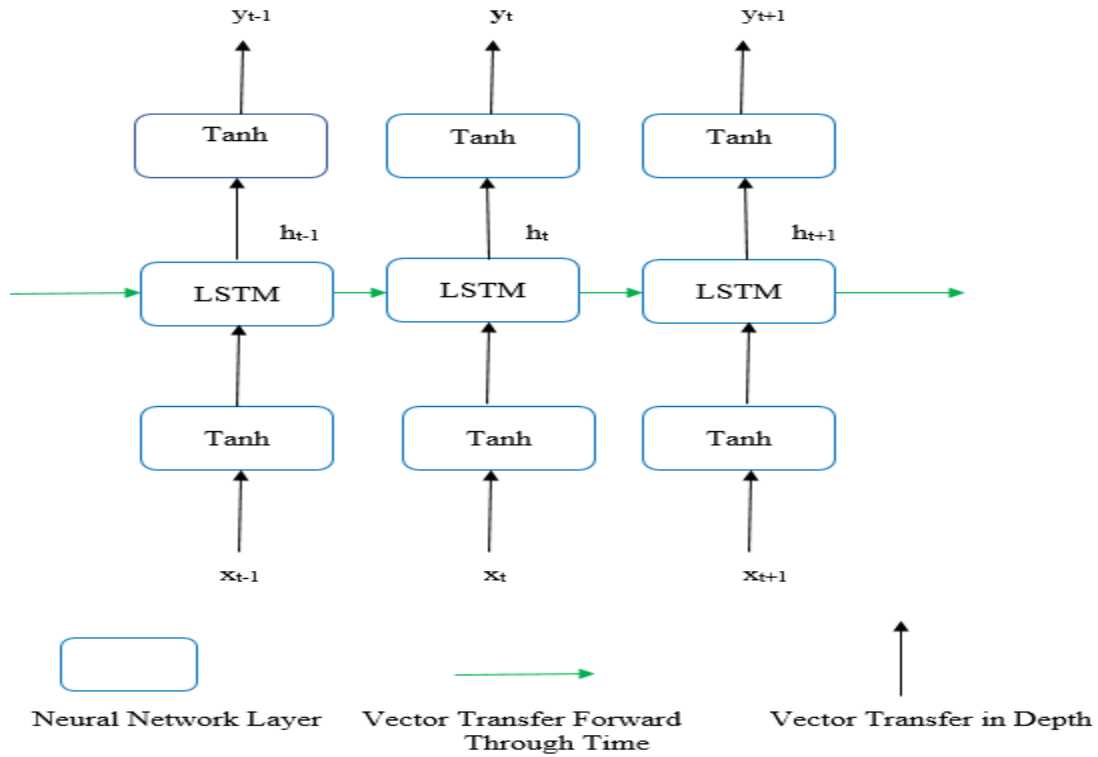


Figure. 19 The unfolded structure of one-layer baseline LSTM

In Figure 19, We define the input set as $\{x_0, x_1, \dots, x_t, x_{t+1}, \dots\}$ and the output set as $\{y_0, y_1, \dots, y_t, y_{t+1}, \dots\}$ and hidden layers as $\{h_0, h_1, \dots, h_t, h_{t+1}, \dots\}$. Then, U, W, V denote weight metrics from the input layer to the hidden layer, from the hidden layer to the hidden layer, and from the hidden layer to the output layer respectively. Baseline LSTM structure operating through the time axis, from left to right. The transfer process of the network can be described as follows: the input tensor is transformed along with the tensor of the hidden layer (at the last stage), to the hidden layer by a matrix transformation. Then, the output of the hidden layer passes through an activation function to the final value of the output layer. Formally, outputs of the hidden layer and output layer can be defined as follows:

$$\begin{aligned}
 h_i &= g(Ux_i + b_i^h) && \text{where } i = 0 \\
 h_i &= g(Ux_i + Wh_{i-1} + b_i^h) && \text{where } i = 1, 2, \dots \\
 y_i &= g(Vh_i + b_i^y) && \text{where } i = 0, 1, \dots
 \end{aligned}$$

4.3.3 Bidirectional LSTM

Baseline LSTM cells predict the current status based only on former information. It is clear that some important information may not be captured properly by the cell if it runs in only one direction. Bidirectional LSTM have been successfully applied for emotion recognition from low level frame-wise audio features which requires modeling of long range context along both input directions [52].

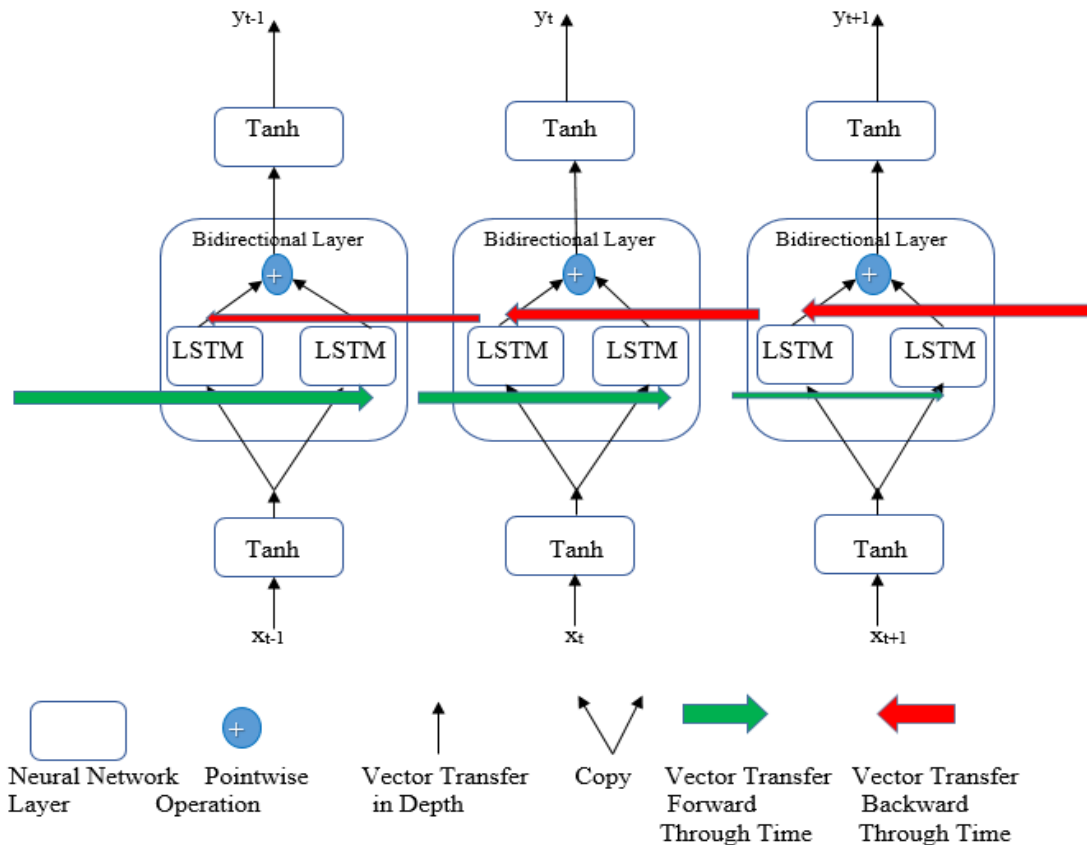


Figure. 20 The structure of single layer bidirectional LSTM

As shown in Figure. 20, the bidirectional layer gets information from vertical direction (lower layer) and horizontal direction (past and future) from two separate hidden layers, and finally outputs the processed information for the upper layer. There are forward sequences \vec{h} from left to right with green arrows and backward sequences $\leftarrow h$ from right to left with red arrows in the hidden layer. For the moment, t (0, 1, 2...) the hidden layer and the output layer can be defined as followed.

$$\begin{aligned}(\rightarrow) h_t &= g(U_h x_t + W_h h_{t-1} + b_h) \\(\leftarrow) h_t &= g(U_h x_t + W_h h_{t-1} + b_h) \\y_t &= g(V_h h_t^{\rightarrow} + V_h h_t^{\leftarrow} + b_y)\end{aligned}$$

4.3.4 Residual LSTM

The Microsoft Research Asia (MSRA) team built a 152-layer network, On the ImageNet dataset the team evaluate residual nets with a depth of up to 152 layers—8× deeper than VGG nets [54] but still having lower complexity. This result won the 1st place on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2015 classification task. The depth of representations is of central importance for many visual recognition tasks.

A residual network [54] provides an identity mapping by shortcut paths. Since the identity mapping is always on, function output only needs to learn residual mapping. Formulation of this relation can be expressed as:

$$y = F(x; W) + x$$

where y is an output layer, x is an input layer and $F(x; W)$ is a function with an internal parameter W . Without a shortcut path, $F(x; W)$ should represent y from input x , but with an identity mapping x , $F(x; W)$ only needs to learn residual mapping, $y - x$. As layers are stacked up, if no new residual mapping is needed, a network can bypass identity mappings without training, which could greatly simplify training of a deep network.

As the network deepens, the research emphasis shifts on how to overcome the obstruction of information and gradient transmission. The MSRA uses residual networks with the main idea that it is easier to optimize the residual mapping than to optimize the original, unreferenced mapping. An important advantage of residual networks is that they are much easier to train because the gradients can be passed through the layers more directly with the addition operator that enables them to bypass some layers that would have otherwise been restrictive. This enables both better training and a deeper network, because residual connections do not impede gradients and still contribute to refining the output of a highway layer composed of such residual connections [55].

Skip connections made the training of very deep networks possible and have become an indispensable component in a variety of neural architectures. The difficulty of training deep networks is partly due to the singularities caused by the non-identifiability of the model. Several such singularities have been identified in previous works: (1) overlap singularities caused by the permutation symmetry of nodes in a given layer, (2) elimination singularities corresponding to the elimination, i.e. consistent deactivation, of nodes, (3) singularities generated by the linear dependence of the nodes. These singularities cause degenerate manifolds in the loss landscape that slow down learning. We argue that skip connections eliminate these singularities by breaking the permutation symmetry of nodes, by reducing the possibility of node elimination and by making the nodes less linearly dependent. Moreover, for typical initializations, skip connections move the network away from the “ghosts” of these singularities and sculpt the landscape around them to alleviate the learning slow-down. These hypotheses are supported by evidence from simplified models, as well as from experiments with deep networks trained on real-world datasets. [56]

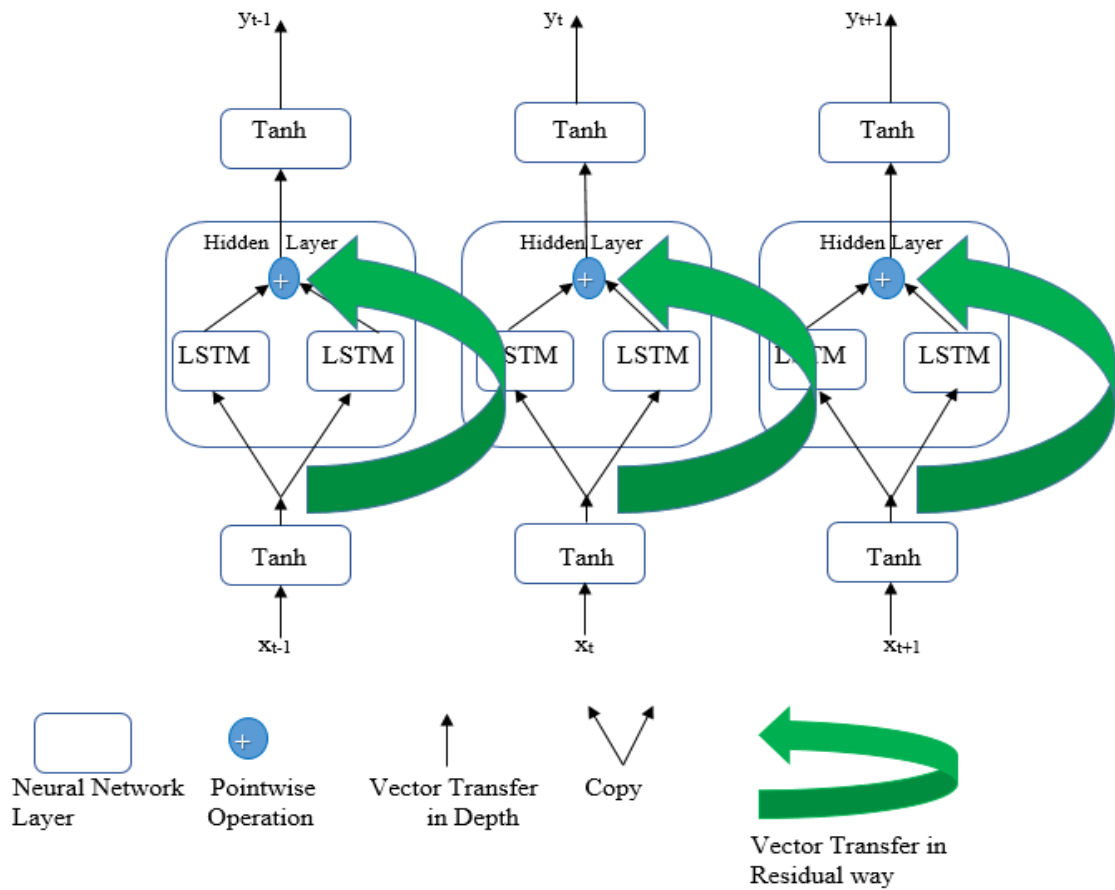


Figure. 21 The structure of single layer residual LSTM

The lower information can transmit to upper layer directly through a highway, which increases the freedom of the information flowing. The highway structure containing skip connections can connect many supplementary n ($n=0, 1, 2, \dots$) layers in height before the bottleneck. When n equals 0, there is no residual connection: it becomes like the baseline deep-stacked LSTMs layers. The output of the hidden layer i ($i=1, 2, \dots, L$) can be defined as follows:

$$\begin{aligned}
 h_1 &= \sigma(W_1 x + b_1) && \text{where } i = 1 \\
 h_i &= \sigma(W_i h_{i-1} + b_i) + h_{i-1} && \text{where } i = 2, 3, \dots, L-1 \\
 y &= \sigma(W_y h_{i-1} + b_y) + h_{i-1} && \text{where } i = L
 \end{aligned}$$

During the code implementation, indexing in the configuration file starts at one rather than zero because we included the count of the first layer that acts as a basis before the residual cells. The same counting rule applies for indicating how many blocks of residual highway layers are stacked one on top of the other.

4.3.5 Deep Residual Bidirectional LSTM

The deep bidirectional LSTM (BDLSTM) architectures are networks with several bidirectional stacked LSTM hidden layers, in which the output of a LSTM hidden layer will be fed as the input into the subsequent LSTM hidden layer. This stacked layers mechanism enhances the power of neural networks [57]. Previous research [58] has shown that, the BDLSTM takes the spatial time series data as the input and predict future speed values for one time-step. The BDLSTM is also capable of predicting values for multiple future time steps based on historical data. When feeding the spatial-temporal information of the traffic network to the BDLSTMs, both the spatial correlation of the speeds in different locations of the traffic network and the temporal dependencies of the speed values can be captured during the feature learning process. In this regard, the BDLSTMs are very suitable for being the first layer of a model to learn more useful information from spatial time series data. When predicting future speed values, the top layer of the architecture only needs to utilize learned features, namely the outputs from lower layers, calculates iteratively along the forward direction and generates the predicted values. But as complexity and volume of data grows the model may not work due to the obstruction of information and gradient transmission as discussed in residual LSTM section. In general, gradient vanishing is a widespread problem for deep networks. Then there is a need for a hybrid LSTM model which would work on those cases. The residual, bidirectional, and stacked layers (hence, the name “Deep Residual Bidirectional LSTM” (RBDLSTM)) [59] help counter this problem, because some bottom layers would otherwise be too hard to optimize when using backpropagation.

The RBDLSTM layer contains a BDLSTM layer as the first feature-learning layer and a LSTM layer as the last layer. For sake of making full use of the input data and learning complex and comprehensive features, the RBDLSTM includes one or more middle BDLSTM layers along with residual LSTM layers. These architectures can take formation of 2 x 2 layers, 3 x 3 layers or 4 x 4 layers depending on the complexity nature of the issues along with learning rate, where there would be n residual layers which contains each n bidirectional hidden layers. Combined with batch normalization on the top of each residual layer, residual connections act as shortcut for gradients. It prevents restrictions in the hidden layer feature space from being too complex and avoids outlier values at test time, against overfitting.

In Figure 22, the information flows bidirectional fashion in the horizontal direction (temporal dimension) and unidirectional fashion in the vertical direction (depth dimension). With the exception of the input and output layers, there are 2 hidden layers which have residual connection inside (hence, called “residual layer”). Moreover, each residual layer contains 2 bidirectional layers. The network in Figure. 22 demonstrated 2 x 2 architecture, which can also be thought of as 8 LSTM cells in sum working as a network. In our network, the activity function is unified with ReLU, because it always outperforms with deep networks to counter gradient vanishing. Although the output is a tensor for a given time window, the time axis has been crunched by the neural network. That is, we need only the last element of the output and can discard the others. Thus, only the gradient from the prediction at the last time step is applied. This also causes a LSTM cell to be unnecessary: the uppermost backward LSTM in the bidirectional pass. Hopefully, this is not of great concern because TensorFlow should evaluate what to compute and what not to compute. Additionally, the training dataset should be shuffled during the training process. The state of the neural network is reset at each new window for each new prediction.

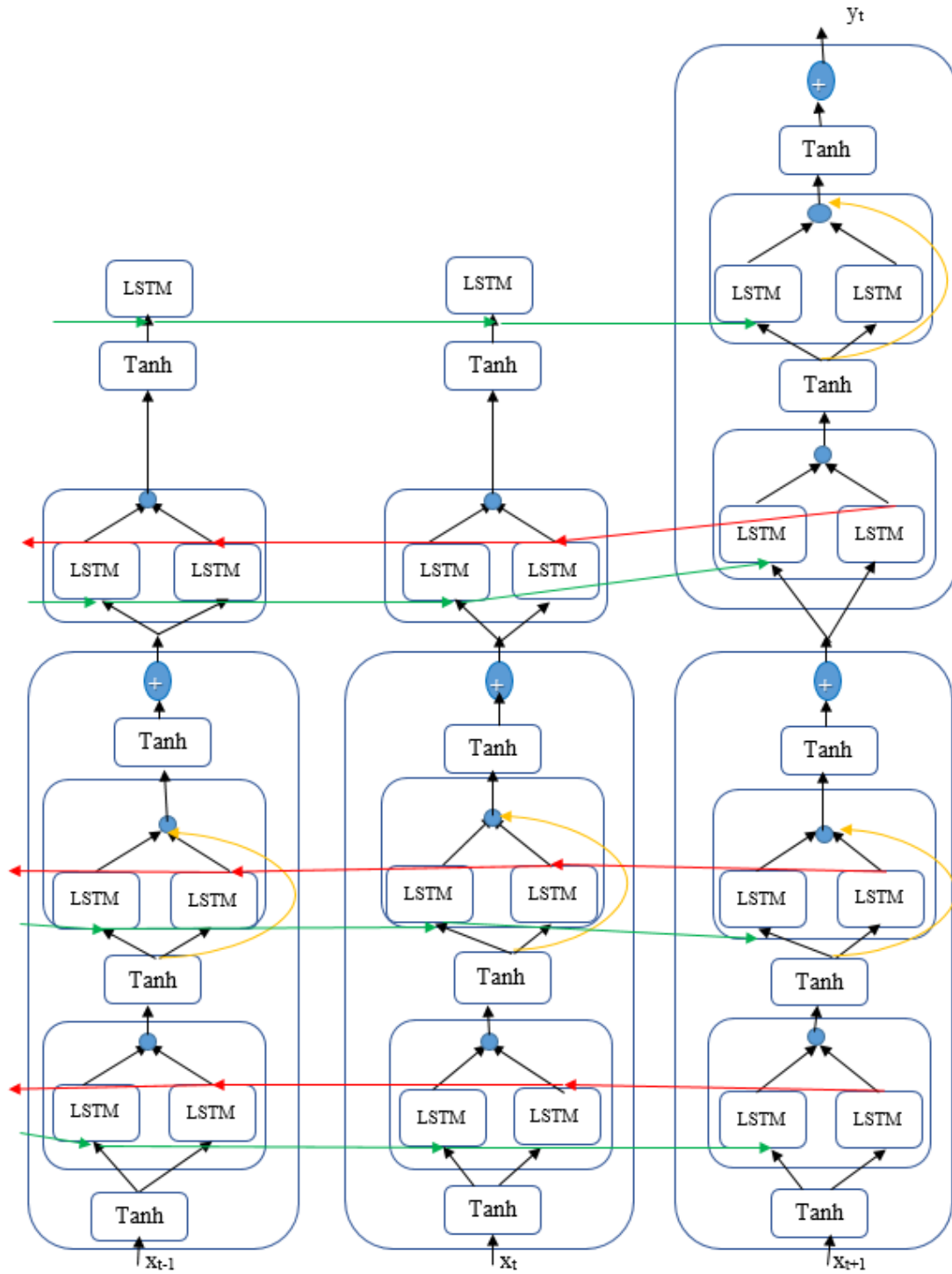


Figure. 22 The structure of 2 x 2 residual bidirectional LSTM

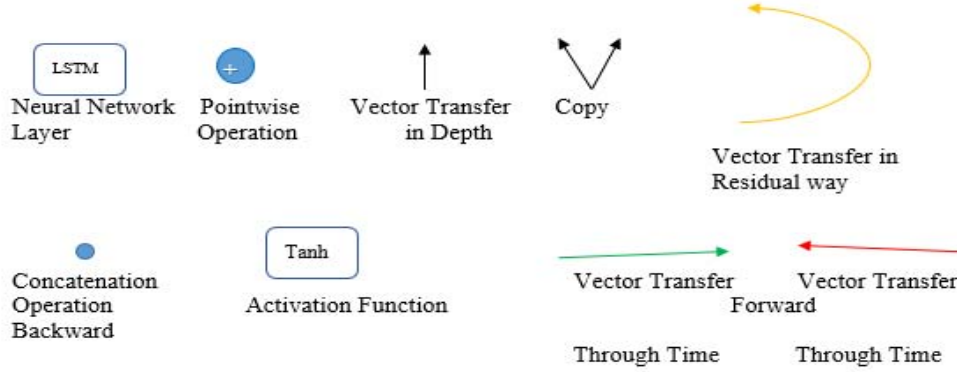


Figure. 23 The residual bidirectional LSTM components

The residual bidirectional LSTM is the hybrid of all the above layers, shown below the output of hidden layer and output layer in a series as follows:

Stacked LSTM without residual connections:

Let $LSTM_i$ and $LSTM_{i+1}$ be the i^{th} and $(i+1)^{th}$ LSTM layers in a stack, whose parameters are W^i and W^{i+1} respectively. At the t^{th} time step, for the stacked LSTM without residual connections, we have:

$$\begin{aligned}
 c_t^i, m_t^i &= LSTM_i(c_{t-1}^i, m_{t-1}^i, x_t^{i-1}; W^i) \\
 x_t^i &= m_t^i, \\
 c_t^{i+1}, m_t^{i+1} &= LSTM_{i+1}(c_{t-1}^{i+1}, m_{t-1}^{i+1}, x_t^i; W^{i+1})
 \end{aligned}$$

where x_t^i is the input to $LSTM_i$ at time step t , and m_t^i and c_t^i are the hidden states and memory states of $LSTM_i$ at time step t , respectively.

Stacked LSTM with residual connections:

With residual connections between LSTM_i and LSTM_{i+1}, the above equations become:

$$\begin{aligned}c_t^i, m_t^i &= \text{LSTM}_i(c_{t-1}^i, m_{t-1}^i, x_t^{i-1}; W^i) \\x_t^i &= m_t^i + x_t^{i-1}, \\c_t^{i+1}, m_t^{i+1} &= \text{LSTM}_{i+1}(c_{t-1}^{i+1}, m_{t-1}^{i+1}, x_t^i; W^{i+1})\end{aligned}$$

Residual connections greatly improve the gradient flow in the backward pass, which allows us to train very deep networks.

Stacked LSTM with residual bidirectional connections:

In an LSTM stack with residual connections there are two accumulators: c_t^i along the time axis and x_t^i along the depth axis. In theory, both of the accumulators are unbounded, but in practice, we noticed their values remain quite small. For inference, we explicitly constrain the values of these accumulators to be within $[-\delta, \delta]$ to guarantee a certain range that can be used for calculation purpose later. The forward computation of an LSTM stack with residual connections is modified to the following:

$$\begin{aligned}c_t^i, m_t^i &= \text{LSTM}_i(c_{t-1}^i, m_{t-1}^i, x_t^{i-1}; W^i) \\c_t^i &= \max(-\delta, \min(\delta, c_t^i)) \\x_t^i &= m_t^i + x_t^{i-1}, \\x_t^i &= \max(-\delta, \min(\delta, x_t^i)) \\c_t^{i+1}, m_t^{i+1} &= \text{LSTM}_{i+1}(c_{t-1}^{i+1}, m_{t-1}^{i+1}, x_t^i; W^{i+1}) \\c_t^{i+1} &= \max(-\delta, \min(\delta, c_t^{i+1}))\end{aligned}$$

It can be quantized further with effective quantization methods by reducing bit-widths of weights, activations and gradients of a neural network which can shrink its storage size and

memory usage, and also allow for faster training and inference by exploiting bitwise operations.[60]. This area is not researched in this thesis.

CHAPTER V

TESTBED SETUP

In this section, we are going to set up the hardware for this research and run the simulation programs to verify that the platform is ready for the research. This introduces distributed machine learning where we need cluster of machines, which are either connected physically with each other or connected by the web networks. Here we built a Raspberry Pi cluster which consists of 16 Raspberry Pi 3 B+ models connected together by a switch hub where the switch is connected to LAN of the research lab. The next platform we built a NVIDIA GPU cluster which consists of 3 GPUs Tesla K40c, Quadro P5000 and Quadro K620 on top of a multicore CPU with 32 GB RAM and 2 TB SSD with 10 TB HDD space. We presented every details of set up with simulation results in below section.

5.1 NVIDIA GPU Test Bed Setup

We use Ubuntu 18.04 LTS 64-bit version for our development environments to perform the experiments. We use one multicore CPU machine with enough memory and disk space to support 3 GPUs named Tesla K40c, Quadro P5000 and Quadro K620. In this experiment, we have used NVIDIA Maximus formation by using the computational power of NVIDIA Tesla GPU and visualization power of NVIDIA Quadro GPU. This is the most efficient formation recommended by NVIDIA for deep learning performance. The cluster of this GPUs is connected by 100 Mbps LAN. Hardware configuration of the machine along with NVIDIA GPUs are listed in Table 2, 3 and the details workstation set

up is described in APPENDIX A.

Processor	Dual Intel Xeon E5-2609 v4, 8-Core, 1.7 Ghz, 20MB L3 Cache, 85 Watts
Memory	32GB DDR4- 2400MHz (4 x 8GB)
Motherboard	Asus Z10PE-D16 WS Intel Xeon
Power Supply	750 Watt EGVA SuperNOVA, 80Plus Bronze Certified
Hard Drive 1	2TB Samsung 960 Pro PCIe 3.0 SSD
Hard Drive 2	2 x 4 TB 7200rpm SATA 600 with 64MB Cache
GPUs	Tesla K40c, Quadro P5000, Quadro K620

Table. 2 Hardware configuration of CPU

Device 0:	Quadro P5000
CUDA Driver Version / Runtime Version	10.0 / 10.0
CUDA Computation Capability Version	6.1
Total amount of global memory	16279 MBytes (17069309952 bytes)
Total CUDA Cores	2560
Multiprocessors	(20) Multiprocessors, (128) CUDA Cores
GPU Max Clock rate	1734 MHz (1.73 GHz)
Memory Clock rate	4513 Mhz

Memory Bus Width	256-bit
Maximum Texture Dimension Size (x,y,z)	1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
Device 1:	Tesla K40c
CUDA Driver Version / Runtime Version	10.0 / 10.0
Total amount of global memory	11441 MBytes (11996954624 bytes)
Total CUDA Cores	2880
Multiprocessors	(15) Multiprocessors, (192) CUDA Cores
GPU Max Clock rate	745 MHz (0.75 GHz)
Memory Clock rate	3004 Mhz
Memory Bus Width	384-bit
Maximum Texture Dimension Size (x,y,z)	1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Device 2:	Quadro K620
CUDA Driver Version / Runtime Version	10.0 / 10.0
Total amount of global memory	2000 MBytes (2096955392 bytes)
Total CUDA Cores	384
Multiprocessors	(3) Multiprocessors, (128) CUDA Cores
GPU Max Clock rate	1124 MHz (1.12 GHz)
Memory Clock rate	900 Mhz
Memory Bus Width	128-bit
Maximum Texture Dimension Size (x,y,z)	1D=(65536), 2D=(65536, 65536),

	3D=(4096, 4096, 4096)
Warp size (same for all)	32

Table. 3 Hardware configuration of GPUs

As shown in Table. 3, the machine is installed with CUDA 10.0.130 along with compatible cuDNN 7.5 for the TensorFlow and PyTorch.

The details process of installation of CUDA and cuDNN are mentioned in APPENDIX – A. Here we have mentioned the verification process after installation.

Recommended Actions for Installation Verifications

1. Check the .bashrc after reboot.
2. Verify the installed driver version. If driver is installed correctly it will be loaded by the below command.

\$ cat /proc/driver/nvidia/version

```
hpcmonster369@hpc369-Z10PE-D16-WS:~$ cat /proc/driver/nvidia/version
NVRM version: NVIDIA UNIX x86_64 Kernel Module 410.104 Tue Feb 5 22:58:30 CST 2019
GCC version: gcc version 7.3.0 (Ubuntu 7.3.0-27ubuntu1~18.04)
```

Figure.24 NVIDIA Driver Version

3. Verify the CUDA Toolkit Version by the below command.

\$ nvcc -V

```
hpcmonster369@hpc369-Z10PE-D16-WS:~$ nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2018 NVIDIA Corporation
Built on Sat_Aug_25_21:08:01_CDT_2018
Cuda compilation tools, release 10.0, V10.0.130
hpcmonster369@hpc369-Z10PE-D16-WS:~$
```

Figure.25 CUDA Toolkit Version

4. Compile the CUDA Examples

In order to modify, compile and run samples it must be installed with write permission. Please run the script which is already available in the CUDA installed directory.

```
cuda-install-samples-10.0.sh ~
```

which will copy the samples to the home directory. Once the copying is finished please run the below command to compile the samples.

```
cd ~/NVIDIA_CUDA-10.0_Samples/5_Simulations/nbody
```

```
make ./nbody
```

5. Run the Binaries

After compilation, run the deviceQuery under Samples folder, by below command.

```
./deviceQuery
```

If the CUDA software is installed and configured correctly the output of the deviceQuery would show pass statement as shown below.

```
deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.0, CUDA Runtime
Version = 10.0, NumDevs = 3
```

```
Result = PASS
```

Run the bandwidthTest for verification by below command.

```
./p2pBandwidthLatencyTest
```

If the CUDA software is able to connect to other GPU drivers then the below matrix will come in result page which validates the successful installation of CUDA.

P2P=Enabled Latency (P2P Writes) Matrix (us)

```
GPU  0   1   2
0  1.27 15.42 14.41
1 14.54  4.28 16.47
2 13.74 16.19  3.75
```

```
CPU  0   1   2
0  6.07 14.02 13.95
1 14.14  5.99 13.97
2 13.95 13.80  6.10
```

Test passed!

6. Verify the cuDNN validation test after successful installation of cuDNN.

To verify that cuDNN is running properly, compile the mnistCUDNN sample located in the `/usr/src/cudnn_samples_v7` directory in the debian file installation folder.

Steps:

1. Copy the cuDNN sample to a writable path.

Scp -r /usr/src/cudnn_samples_v7/ \$HOME

2. Go to the writable path.

\$ cd \$HOME/cudnn_samples_v7/mnistCUDNN

3. Compile the mnistCUDNN sample

\$ make clean && make

4. If face any issues, open the file /usr/include/cudnn.h & change below details & save it.

```
#include "driver_types.h" → #include <driver_types.h>
```

5. Run the mnistCUDNN sample.

\$./mnistCUDNN

6. If cuDNN is properly installed and running on you Linux machine you will see the similar message as above.

Test passed!

The above complete test results files are provided in the APPENDIX -A.

After above installation now the machine is compatible for running deep learning models but still clustering is ready to set.

There are basically two options how to do multi-GPU programming. First option to do it in CUDA and have a single thread and manage the GPUs directly by setting the current device and by declaring and assigning a dedicated memory-stream to each GPU or the other options is to use CUDA_Aware_MPI where a single thread is spawned for each GPU and all communication and synchronization is handled by MPI.

We have choose to go by the first option where the clustering is done based on cudaSetDevice query.

```

hpcmonster369@hpc369-Z10PE-D16-WS:~$ nvidia-smi topo -m
      GPU0    GPU1    GPU2    CPU Affinity
GPU00  X      PHB     SYS     0-7
GPU01  PHB    X       SYS     0-7
GPU02  SYS    SYS     X       8-15

Legend:
 X      = Self
 SYS    = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
 NODE   = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
 PHB    = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
 PXB    = Connection traversing multiple PCIe switches (without traversing the PCIe Host Bridge)
 PIX    = Connection traversing a single PCIe switch
 NV#    = Connection traversing a bonded set of # NVLinks

```

Figure.26 GPU Memory Array

The result of the NVIDIA Maximus cluster formation is shown below.

- < multiple host threads can use ::cudaSetDevice() with device simultaneously >
- > Peer access from Quadro P5000 (GPU0) -> Tesla K40c (GPU1) : Yes
- > Peer access from Quadro P5000 (GPU0) -> Quadro K620 (GPU2) : Yes
- > Peer access from Tesla K40c (GPU1) -> Quadro P5000 (GPU0) : Yes
- > Peer access from Tesla K40c (GPU1) -> Quadro K620 (GPU2) : Yes
- > Peer access from Quadro K620 (GPU2) -> Quadro P5000 (GPU0) : Yes
- > Peer access from Quadro K620 (GPU2) -> Tesla K40c (GPU1) : Yes

We have implemented the Synchronous All-reduce approach which is the default behavior of the distributed TensorFlow, MirroredStrategy API.

Both of these examples implement the All-reduce approach, however they can be easily extended to other approaches. Here 3 GPUs are working as actors to mirror the task which is taken care by TensorFlow in Figure. 27.


```

#-----
# Let's get serious and build the neural network
#-----
mirrored_strategy = tf.contrib.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1", "/gpu:2"])
with mirrored_strategy.scope():
    X = tf.placeholder(tf.float32, [
        None, config.n_steps, config.n_inputs], name="X")
    Y = tf.placeholder(tf.float32, [
        None, config.n_classes], name="Y")

    # is_train for dropout control:
    is_train = tf.placeholder(tf.bool, name="is_train")
    keep_prob_for_dropout = tf.cond(is_train,
        lambda: tf.constant(
            config.keep_prob_for_dropout,
            name="keep_prob_for_dropout"
        ),
        lambda: tf.constant(
            1.0,
            name="keep_prob_for_dropout"
        )
    )

    pred_y = LSTM_network(X, config, keep_prob_for_dropout)

```

Figure.27 Distributed TensorFlow API

5.2 Cluster of Raspberry Pis Setup

We have used Raspberry Pi 3 Model B+ in this experiment. There is a perfect reason to use raspberry pi in this research.

The raspberry pi board comprises a program memory (RAM), processor, graphics chip, CPU, GPU, Ethernet port, GPIO pins, Xbee socket, UART, power source connector and various other interfaces for other external devices. We have added a 32 GB flash memory SD card which could be used as storage in each pi. So that raspberry pi board will boot from this SD card similarly as a PC boots up into windows from its hard disk. This tiny computer having all the qualities with very cost effective price, a perfect candidate to build large clusters for research purpose.

Hardware configuration of Raspberry Pi is listed in the Table. 4.

There are 16 Pi's used to make this cluster. The Raspbian Stretch Kernel Version 4.14 is installed in each Pi. TensorFlow 1.8.0 the .whl file version is installed in each Pi. To create the sharing folder between the Pis NFS server and client model is implemented as it is supported by Linux terminal.

Processor	Broadcom BCM2837B0, Cortex-A53, 64-bit SoC @ 1.4GHz
Memory	1GB LPDDR2 SDRAM
Hard Drive 1	Samsung 32 GB Flash Drive
Power Supply	5V/2.5A DC via micro USB connector
Integrated Wi-Fi	2.4GHz and 5GHz
Ethernet speed	300Mbps

Table. 4 Hardware configuration of Raspberry Pi 3 Model B+

The details of creating the sharing server and client structure is described below.

Step-1:

Create NFS server in one of the pi which is known as master pi. Before setting up the NFS there is some prerequisites which is good to follow.

Login to the Pi configuration management file by the below command.

sudo raspi-config

1. Update the pi software.
2. Rename each pi from the default name to rpi# as per the nodes going to be used in the cluster. You can do that from the configuration file itself and restart the pi.
3. Enable the hostname in each pi.
4. Change the password of default to your own convenient one.
5. Change the assigned memory for GPU to minimum.

6. Change the assigned memory for CPU to maximum.
7. In raspi-config, change (3. Boot Options > B2 Wait for Network at Boot) from “No” to “Yes”. This will ensure that networking is available before the fstab file mounts the NFS client.
8. In raspi-config, enable the ssh mode.

Step-2: (NFS Server)

Install the NFS server in the master node by the below command.

```
sudo apt-get install nfs-common nfs-server -y
```

```
sudo mkdir /home/pi/Desktop/nfsserver
```

```
sudo chmod -R 777 /home/pi/Desktop/nfsserver
```

This will create a server folder named nfsserver in the master pi.

Step-3:

Validate the NFS Version by the below command.

```
rpcinfo -u localhost nfs
```

Step-4:

Add the nfsserver folder to the localhost so that when other Pi add something to the folder it will be automatically updated by the server. Please use below commands.

```
/home/pi/Desktop/nfsserver 192.168.1.1/26(rw, sync, no_subtree_check)
```

where nfsserver folder will read, write, sync with no sub tree check.

Step -5: (NFS Client)

Install the NFS client in each of the Raspberry Pi node so that we will communicate each other by the RPC protocol which NFS uses internally to communicate.

1. Install the NFS client by the command.

```
sudo apt-get install nfs-common -y
```

2. Make a client directory in the Pi.

```
sudo mkdir -p /home/pi/Desktop/nfs
```

3. Give permission to the directory.

```
sudo chown -R pi:pi /home/pi/Desktop/nfs
```

4. Mount the directory to the NFS server.

```
sudo mount 192.168.1.26:/home/pi/Desktop/nfs /home/pi/Desktop/nfs
```

```
sudo nano /etc/fstab 192.168.1.26:/home/pi/Desktop/nfs /home/pi/Desktop/nfs  
nfs rw 0 0
```

5. Verify the mount.

```
nfsstat -m
```

Step-6:

Restart the NFS client service so that the server will recognize the client.

```
sudo /etc/init.d/nfs-common restart
```

Step-7:

Once NFS client is installed in all the Raspberry Pi's restart the NFS server to verify that it is connecting to all the clients.

```
sudo /etc/init.d/nfs-kernel-server restart
```

The below snapshot shows after NFS client server model successfully installed in all the machines.

Figure.28 Raspberry Pis NFS connection

```
pi@rpi01:~$ netstat -a | grep nfs
tcp        0      0 0.0.0.0:nfs          0.0.0.0:*            LISTEN
tcp        0      0 192.168.1.26:917    192.168.1.26:nfs     ESTABLISHED
tcp        0      0 192.168.1.26:nfs    192.168.1.30:1004    ESTABLISHED
tcp        0      0 192.168.1.26:nfs    192.168.1.27:900     ESTABLISHED
tcp        0      0 192.168.1.26:nfs    192.168.1.26:917     ESTABLISHED
tcp        0      0 192.168.1.26:nfs    192.168.1.31:766     ESTABLISHED
tcp        0      0 192.168.1.26:nfs    192.168.1.19:966     ESTABLISHED
tcp        0      0 192.168.1.26:nfs    192.168.1.28:719     ESTABLISHED
tcp        0      0 192.168.1.26:nfs    192.168.1.25:961     ESTABLISHED
tcp        0      0 192.168.1.26:nfs    192.168.1.23:914     ESTABLISHED
tcp        0      0 192.168.1.26:nfs    192.168.1.24:webster ESTABLISHED
tcp        0      0 192.168.1.26:nfs    192.168.1.20:974     ESTABLISHED
tcp        0      0 192.168.1.26:nfs    192.168.1.29:928     ESTABLISHED
tcp        0      0 192.168.1.26:nfs    192.168.1.18:736     ESTABLISHED
tcp        0      0 192.168.1.26:nfs    192.168.1.16:917     ESTABLISHED
tcp        0      0 192.168.1.26:nfs    192.168.1.21:699     ESTABLISHED
tcp        0      0 192.168.1.26:nfs    192.168.1.17:1007    ESTABLISHED
tcp        0      0 192.168.1.26:nfs    192.168.1.22:744     ESTABLISHED
tcp6       0      0 [::]:nfs            [::]:*                LISTEN
udp        0      0 0.0.0.0:nfs         0.0.0.0:*
udp6       0      0 [::]:nfs            [::]:*
```

The directory is always available to all the Raspberry Pi workers along with master which having both client and server, the status of NFS is shown below in the snapshot.

```
pi@rpi01:~$ sudo /etc/init.d/nfs-kernel-server status
● nfs-server.service - NFS server and services
   Loaded: loaded (/lib/systemd/system/nfs-server.service; enabled; vendor preset: enabled)
   Active: active (exited) since Tue 2018-10-30 16:09:12 UTC; 11min ago
     Process: 503 ExecStart=/usr/sbin/rpc.nfsd $RPCNFSDARGS (code=exited, status=0/SUCCESS)
     Process: 501 ExecStartPre=/usr/sbin/exportfs -r (code=exited, status=0/SUCCESS)
    Main PID: 503 (code=exited, status=0/SUCCESS)
   CGroup: /system.slice/nfs-server.service

Oct 30 16:09:12 rpi01 systemd[1]: Starting NFS server and services...
Oct 30 16:09:12 rpi01 systemd[1]: Started NFS server and services.
```

Figure.29 NFS Status

Issues on NFS Setup:

NFS server on default only allows 15 Raspberry Pi nodes as client to connect to the server. This is the default NFS property. To increase the port please follow the below steps.

Step-1:

Go to the nfs-kernel-server file in command prompt and change to larger number as required.

sudo nano /etc/default/nfs-kernel-server

RPCNFSDCOUNT = 16

RPCMOUNTDOPTS= " --manage-gids --no-nfs-version 3"

Step-2:

Change the following things in nfs-utils file.

sudo nano /run/sysconfig/nfs-utils

RPCNFSDARGS = "16"

Step-3:

Create a directory named "**sunrpc.conf**" in the below location and add details as provided below.

1. Go to the directory: **/etc/modprobe.d**
2. Create a file named: **sunrpc.conf**
3. Add the contents in the above file to allow the clients # in the NFS server:

options sunrpc tcp_slot_table_entries=128

options sunrpc tcp_max_slot_table_entries=128

5.3 Simulation using Raspberry Pis Cluster

The simulation is done with 16 Raspberry Pi cluster where 15 nodes work as worker nodes and one works as master node as well worker node. In this cluster each task is associated with a server. This simulation is the Monte Carlo simulation which use 16 Raspberry Pi cluster distributed TensorFlow environment to give the result of the value of pi.

The program having two parts, one is server program **server.py** which is running in the

NFS server where each client having access and the other is the client part, **client.py** which calculates the value of Pi by using a Monte Carlo method. The program generates random points between (-1, -1) to (1, 1) in a circle of radius 1 inscribed in a square.

The source code of the Program is given in APPENDIX-E.

Distributed TensorFlow works a bit like a server-client model. The idea is that you create a whole bunch of workers that will perform the heavy lifting. You then create a session on one of those workers, and it will compute the graph, possibly distributing parts of it to other clusters on the server. In order to do this, the main worker or the master, needs to know about the other workers. This is done via the creation of a ClusterSpec as shown in Figure. 30, which you need to pass to all workers. A ClusterSpec is built using a dictionary, where the key is a “job name”, and each job contains many workers.

The code is taken from the simulation program where each Raspberry Pi node ip is entitled in the taskList and cluster creates a working cluster by using API **tf.train.ClusterSpec** where each job is specified as a sparse mapping from task indices to network addresses.

```

def getIpAddr():
    ni.ifaddresses("eth0")
    ip = ni.ifaddresses("eth0")[ni.AF_INET][0]["addr"]
    return ip

tf.app.flags.DEFINE_string("job_name", "", "Either 'ps' or 'worker'")
FLAGS = tf.app.flags.FLAGS

parameter_servers = ["192.168.1.26:1024"]

workers = ["192.168.1.16:1024", "192.168.1.17:1024", "192.168.1.18:1024", "192.168.1.19:1024",
           "192.168.1.20:1024", "192.168.1.21:1024", "192.168.1.22:1024", "192.168.1.23:1024",
           "192.168.1.24:1024", "192.168.1.25:1024", "192.168.1.26:1024", "192.168.1.27:1024",
           "192.168.1.28:1024", "192.168.1.29:1024", "192.168.1.30:1024", "192.168.1.31:1024"]

taskName = getIpAddr()+":1024"

try:
    taskNum = workers.index(taskName)
except ValueError:
    print(" Unable to find " + taskName + " in the worker group.")
    quit()

cluster = tf.train.ClusterSpec({"ps":parameter_servers, "worker":workers})
server = tf.train.Server(cluster, job_name=FLAGS.job_name, task_index=taskNum)

if FLAGS.job_name == "ps":
    server.join()
elif FLAGS.job_name == "worker":

```

Figure.30 TensorFlow Cluster API

In the Figure. 31, the TensorFlow API **tf.device** is used which is used to create a device context such that all the operations within that context will have the same device assignment instead of automatically selecting available devices by the program to participate in the computational process. It allows the user to select an user specified device for the operation.

```

with tf.device(tf.train.replica_device_setter(cluster=cluster_spec)):
    X = tf.placeholder(tf.float32, [
        None, config.n_steps, config.n_inputs], name="X")
    Y = tf.placeholder(tf.float32, [
        None, config.n_classes], name="Y")

```

Figure. 31 TensorFlow Device API

TensorFlow uses a dataflow graph to represent your computation in terms of the dependencies between individual operations. This leads to a low-level programming model in which you first define the dataflow graph, then create a TensorFlow session to run parts of the graph across a set of local and remote devices. As shown in Figure. 32, TensorFlow uses **tf.Session** API to create a session object which encapsulates the environment in which Operation objects are executed, and Tensor objects are evaluated. In simple terms, the session allocates memory to store the current value of the variable. **tf.global_variables_initializer()** initializes all the variables of the TensorFlow before using it in the operations.

```
with tf.Session("grpc://localhost:1024") as sess:
    sess.run(tf.global_variables_initializer())
    pi = sess.run(4*(total/size))
    print("pi:",pi)
```

Figure.32 TensorFlow Session API

In Figure. 33, the TensorFlow API **tf.train.Server()** is used as an in-process TensorFlow server, for use in distributed training. A **tf.train.Server** instance encapsulates a set of devices and a **tf.Session** target that can participate in distributed training. A server belongs to a cluster (specified by a **tf.train.ClusterSpec**), and corresponds to a particular task in a named job. The server can communicate with any other server in the same cluster.

```
cluster = tf.train.ClusterSpec({"ps":parameter_servers, "worker":workers})
server = tf.train.Server(cluster,job_name=FLAGS.job_name,task_index=taskNum)

if FLAGS.job_name == "ps":
    server.join()
elif FLAGS.job_name == "worker":
```

Figure. 33 TensorFlow Server API

Result

Sample Size	Time				
	Time 1	Time 2	Time 4	Time 8	16
10,000,000	1.495	1.18	1.32	1.961	2.34
20,000,000	2.594	1.704	1.572	2.087	2.48
30,000,000	3.673	2.284	1.848	2.299	2.57
40,000,000	4.758	2.818	2.09	2.35	2.63
50,000,000	6.561	3.373	2.391	2.48	2.79
60,000,000	7.713	3.91	2.648	2.628	2.593
70,000,000	N/A	4.451	2.923	2.782	2.421
80,000,000	N/A	4.998	3.218	2.953	2.561
90,000,000	N/A	5.678	3.482	3.147	2.842
100,000,000	N/A				
0		6.103	3.741	3.234	2.611

Table. 5 Raspberry Pi Cluster Monte Carlo Simulation

As the cluster size increases, the program is computing faster for larger sample sizes but slower for smaller sample sizes. For example, for sample size 100 million, the size 8 cluster is faster than the size 2 cluster (3.234s vs 6.103s). However, for sample size 10 million, the size 2 cluster is faster than the size 8 cluster (1.180s vs. 1.961s). The slow down for smaller sample sizes may due to overhead for the tasks to communicate with each other.

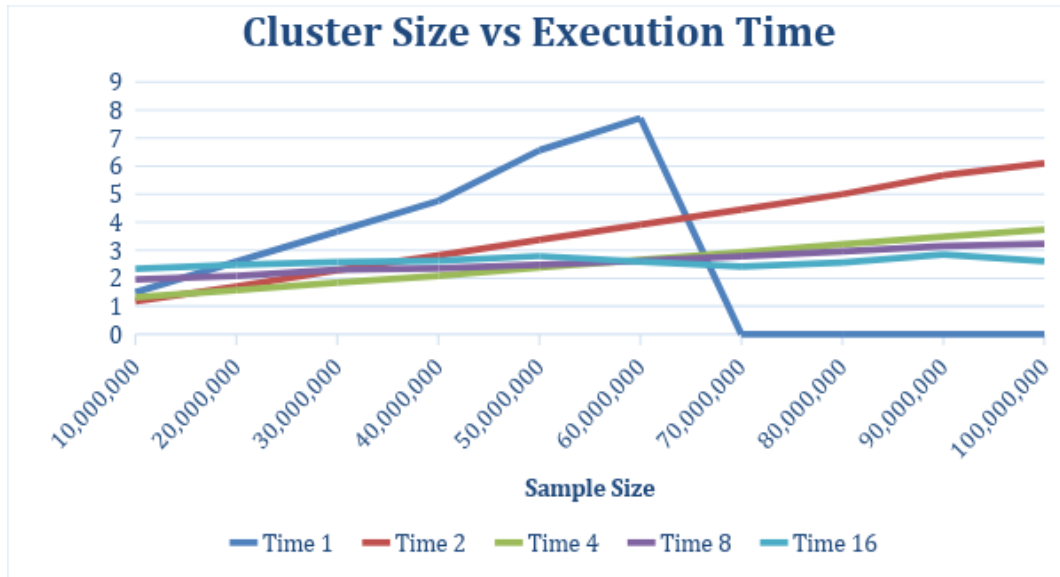


Figure. 34 Pi Cluster Execution Graph

5.4 Notes on TensorFlow Setup

The simulation program using TensorFlow built on the distributed TensorFlow APIs where GPU cluster is used. We have going to discuss import APIs used in this research work.

```
sessconfig = tf.ConfigProto(allow_soft_placement = True, log_device_placement=False)
sessconfig.gpu_options.allow_growth = True
with tf.Session(config=sessconfig) as sess:
    sess.run(tf.global_variables_initializer())
```

Figure. 35 TensorFlow GPU Growth API

By default, TensorFlow requests nearly all of the GPU memory of all GPUs to avoid memory fragmentation (since GPU has much less memory, it is more vulnerable to fragmentation). To avoid this issue, we have used the API

config.gpu_options.allow_growth = True as shown in Figure. 35, where TensorFlow can grow its memory gradually when desired. In Figure. 13 it's observed that while computing the 4 x 4 stacked residual bidirectional layer for dataset with 256 hidden layers which is the most complex iteration in our experiment it uses only one part of the GPU memory.

In the Figure. 35, we have use the API **allow_soft_placement= True** , it would let TensorFlow to automatically choose an existing and supported device to run the operations in case the specified one doesn't exist, we have set **allow_soft_placement** to True in the configuration option when creating the session. With this API, our program is compatible to run in machines without having GPU clusters without giving any errors.

StreamExecutor is a unified wrapper around the CUDA and OpenCL host-side programming models (runtimes). It lets host code target either CUDA or OpenCL devices with identically-functioning data-parallel kernels. StreamExecutor is currently used as the runtime for the vast majority of Google's internal GPGPU applications, and a snapshot of it is included in the open-source TensorFlow project, where it serves as the GPGPU runtime. As shown in Figure. 36 and 37, it inspects the capabilities of a GPU-like device at runtime and manages multiple devices.

```
2019-04-09 11:26:30.814754: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1511] Adding visible gpu devices: 0, 1
2019-04-09 11:26:31.791539: I tensorflow/core/common_runtime/gpu/gpu_device.cc:982] Device interconnect StreamExecutor with strength 1 edge matrix:
2019-04-09 11:26:31.791595: I tensorflow/core/common_runtime/gpu/gpu_device.cc:988]      0 1 2
2019-04-09 11:26:31.791607: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1001] 0:  N N N
2019-04-09 11:26:31.791615: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1001] 1:  N N N
2019-04-09 11:26:31.791622: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1001] 2:  N N N
2019-04-09 11:26:31.792480: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 15286
MB memory) -> physical GPU (device: 0, name: Quadro P5000, pci bus id: 0000:03:00.0, compute capability: 6.1)
2019-04-09 11:26:31.792933: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:1 with 10750
MB memory) -> physical GPU (device: 1, name: Tesla K40c, pci bus id: 0000:02:00.0, compute capability: 3.5)
```

Figure. 36 GPU StreamExecutor

```
2019-04-09 11:26:30.552467: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1432] Found device 0 with properties:
name: Quadro P5000 major: 6 minor: 1 memoryClockRate(GHz): 1.7335
pciBusID: 0000:03:00.0
totalMemory: 15.90GiB freeMemory: 15.78GiB
2019-04-09 11:26:30.690998: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1432] Found device 1 with properties:
name: Tesla K40c major: 3 minor: 5 memoryClockRate(GHz): 0.745
pciBusID: 0000:02:00.0
totalMemory: 11.17GiB freeMemory: 11.10GiB
2019-04-09 11:26:30.814664: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1432] Found device 2 with properties:
name: Quadro K620 major: 5 minor: 0 memoryClockRate(GHz): 1.124
pciBusID: 0000:81:00.0
totalMemory: 1.95GiB freeMemory: 1.42GiB
```

Figure. 37 GPU Device Selection

```

mirrored_strategy = tf.contrib.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1", "/gpu:2"])
with mirrored_strategy.scope():
    X = tf.placeholder(tf.float32, [
        None, config.n_steps, config.n_inputs], name="X")
    Y = tf.placeholder(tf.float32, [
        None, config.n_classes], name="Y")

```

Figure. 38 Distributed TF Multi GPU

As shown in Figure. 38, for distributed TensorFlow we have used the API, **tf.contrib.distribute.MirroredStrategy** in our program. This strategy uses one replica per device and sync replication for its multi-GPU version. When `cluster_spec` is given by the `configure` method, it turns into the multi-worker version that works on multiple workers with in-graph replication. Note: `configure` will be called by higher-level APIs if running in distributed environment.

In-graph replication: the client creates a single **tf.Graph** that specifies tasks for devices on all workers. The client then creates a client session which talks to the master service of a worker. Then the master will partition the graph and distribute the work to all participating workers.

Worker: A worker is a TensorFlow task that usually maps to one physical machine. We will have multiple workers with different task index. They all do similar things except for one worker checkpointing model variables, writing summaries, etc. in addition to its ordinary work.

The multi-worker version of this class maps one replica to one device on a worker. It mirrors all model variables on all replicas. For example, in our program we have two workers and each worker having single GPUs, it creates 2 copies of the model variables on these 2 GPUs. Then like in `MirroredStrategy`, each replica performs their computation with their own copy of variables unless in cross-replica model where variable or tensor reduction happens.

5.5 Notes on using PyTorch Setup

In PyTorch program we have used the API, **import torch.distributed as dist**.

PyTorch distributed currently only supports Linux. By default, the Gloo and NCCL backends are built and included in PyTorch distributed (NCCL only when building with CUDA). As Rule of thumb, we use the NCCL backend for distributed GPU training using CUDA.

```
train_loader, test_loader = data_preprocess.load(batch_size=BATCH_SIZE)
model = net.Network()
model = model.to(DEVICE)
torch.distributed.init_process_group(backend="nccl")
model = torch.nn.parallel.DistributedDataParallel(model)
optimizer = optim.SGD(params=model.parameters(), lr=LEARNING_RATE, momentum=0.9)
train(model, optimizer, train_loader, test_loader)
result = np.array(result, dtype=float)
```

Figure. 39 PyTorch Distributed API

In Figure. 39, The **torch.distributed** package provides PyTorch support and communication primitives for multiprocess parallelism across several computation nodes running on one or more machines. The class **torch.nn.parallel.DistributedDataParallel()** builds on this functionality to provide synchronous distributed training as a wrapper around any PyTorch model. This differs from the kinds of parallelism provided by Multiprocessing package - **torch.multiprocessing** and **torch.nn.DataParallel()** in that it supports multiple network-connected machines and in that the user must explicitly launch a separate copy of the main training script for each process.

```
train_set = data_loader(x_train, y_train, transform)
test_set = data_loader(x_test, y_test, transform)
train_loader = DataLoader(train_set, batch_size=batch_size, num_workers=8, pin_memory=True, shuffle=True, drop_last=True)
test_loader = DataLoader(test_set, batch_size=batch_size, num_workers=8, pin_memory=True, shuffle=False)
return train_loader, test_loader
```

Figure. 40 PyTorch Memory Shuffle

The task is distributed with 8 workers, and `pin_memory` is true so that the load of Dataset which is on CPU, would push it during training to the GPU, so that it can speed up the host to device transfer by enabling `pin_memory`.

This lets the DataLoader allocate the samples in page-locked memory, which speeds-up the transfer.

As our hardware is single-machine synchronous case, `torch.distributed` or the `torch.nn.parallel.DistributedDataParallel()` wrapper have below advantages over other approaches to data-parallelism.

1. Each process maintains its own optimizer and performs a complete optimization step with each iteration. While this may appear redundant, since the gradients have already been gathered together and averaged across processes and are thus the same for every process, this means that no parameter broadcast step is needed, reducing time spent transferring tensors between nodes which decreases the execution time of the deep learning model iteration.
2. Each process contains an independent Python interpreter, eliminating the extra interpreter overhead and “GIL-thrashing” that comes from driving several execution threads, model replicas, or GPUs from a single Python process. This is especially important for models that make heavy use of the Python runtime, including models with recurrent layers or many small components. As our program has recurrent LSTM layers with many hidden layers it gives an advantage point during deep model iterations.

CHAPTER VI

IMPLEMENTATION

6.1 Best Learning Rate

Deep Layers	Learning Rate	Hidden Layers	Execution Time	Prediction Accuracy	F1 Score
3	0.01	32	0:53:25	0.18052256	0.0552101
3	0.01	64	1:20:59	0.18052256	0.0552101
3	0.01	128	1:48:32	0.18052256	0.0552101
3	0.001	32	0:53:41	0.91177469	0.9116894
3	0.001	64	1:20:42	0.8995589	0.8998715
3	0.001	128	2:44:04	0.91923988	0.9191246
3	0.0001	32	0:53:39	0.89582628	0.8954641
3	0.0001	64	1:20:57	0.87580591	0.8761988
3	0.0001	128	3:58:56	0.89752293	0.8974544

Table. 6 Best Learning Rate

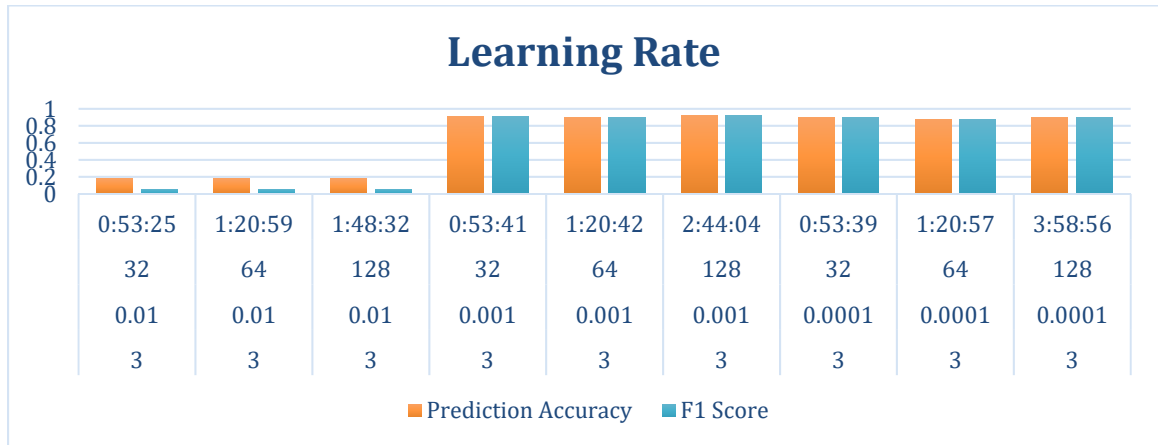


Figure. 41 Best Learning Rate

In the Figure 41. We have shown the prediction accuracy along with F1 score based on 3 different learning rates which are 0.01, 0.001 and 0.001. The F1 score is calculated based on confusion matrix which is an important parameter to verify the calculated accuracy. The learning rate 0.01 has shown the accuracy of 0.18 from Table. 2, which is bad so it can't be accepted as learning rate for the research. The learning rate 0.0001 has shown the accuracy in between 85% - 90 % which is okay but when the execution time is observed it's very high, this time is almost double as compared to 0.01 learning rate so it is discarded. The learning rate 0.001 has given the accuracy 0.9192 from Table. 6, which is considered the best accuracy from 3 learning rates along with best execution time for deep learning iterations.

6.2 CPU Execution Time between Layers

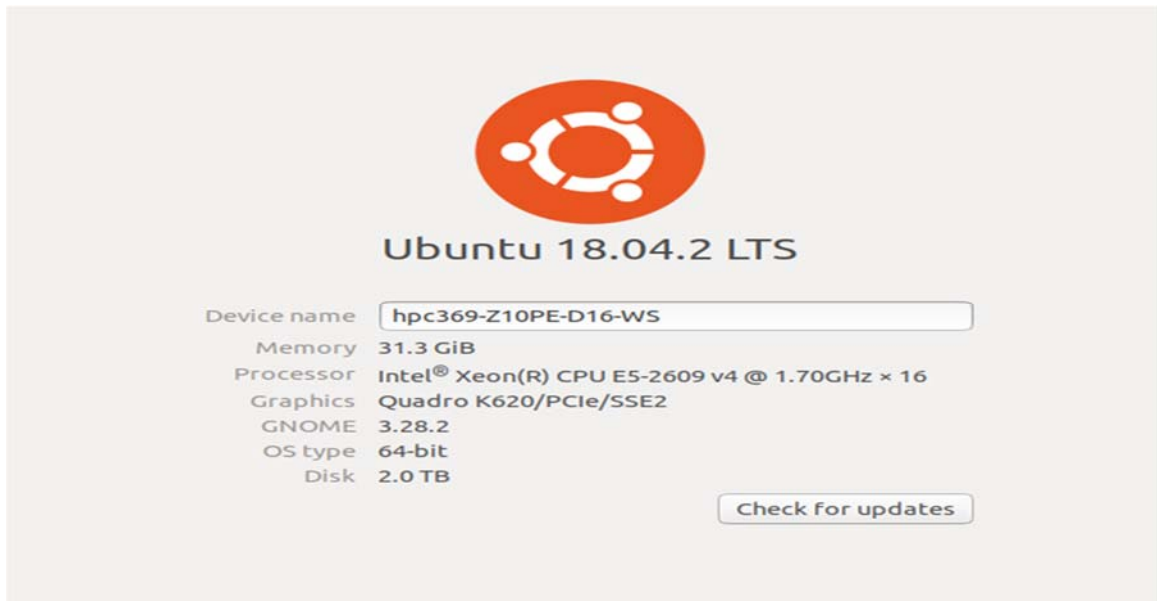


Figure. 42 Big Machine CPU Details

Deep Layers	Residual Layers	Hidden Layers			
		32	64	128	256
		Execution Time	Execution Time	Execution Time	Execution Time
2 x	2 Layers	1:42:21	2:09:22	3:04:51	5:49:40
3 x	3 Layers	5:18:24	9:34:22	6:44:26	16:00:54
4 x	4 Layers	8:35:14	9:14:42	11:18:50	20:50:11

Table. 7 Execution rate between Layers in CPU

In Table. 7 , the execution time of all the 3 layers which are 2 deep layers along with 2 residual layers, 3 deep layers along with 3 residual layers and 4 deep layers along with 4 residual layers are shown which are done by the computational power of CPU. The CPU hardware along with internal configuration is shown in Figure 42. The CPU uses its 16 core to do the computational analysis as shown in Figure. 50. Each layers having 4 types of hidden layers which are 32 layers, 64 layers, 128 layers and 256 layers which are having tensors to do the deep learning iterations. As shown in Figure. 43, the execution time increases as the hidden layers increases in the same layer as shown in the bubble chart. The smaller bubble means less execution time as compared to larger bubble which reflects longer execution time. As the layers scale increases the execution time increases as well. As we can see it from Figure. 44, as number of layers increases along with more hidden layers the execution time is the more longer than previous one.

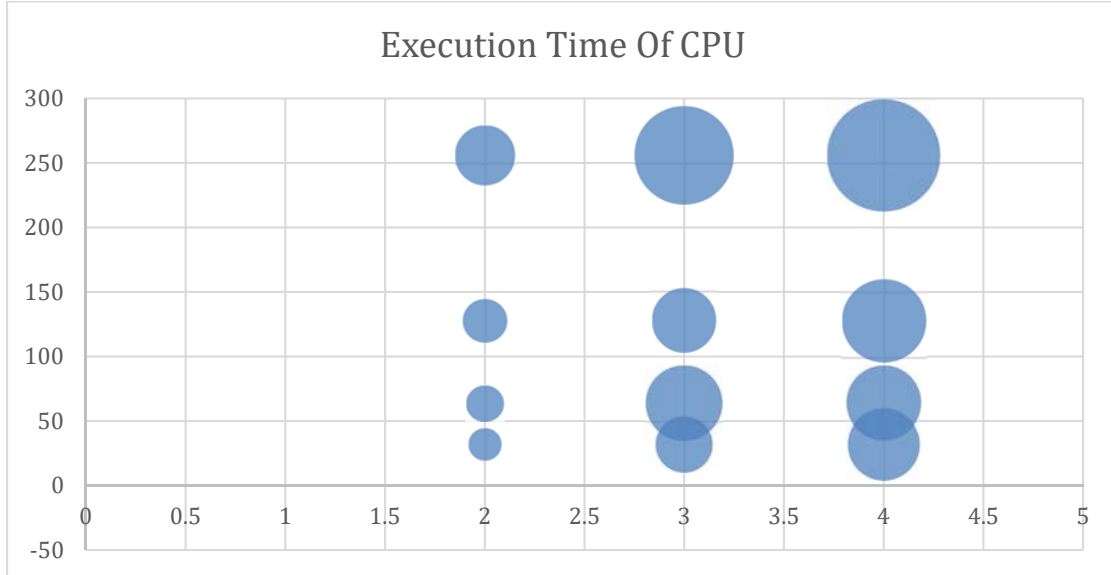


Figure. 43 Bubble Chart of CPU Execution

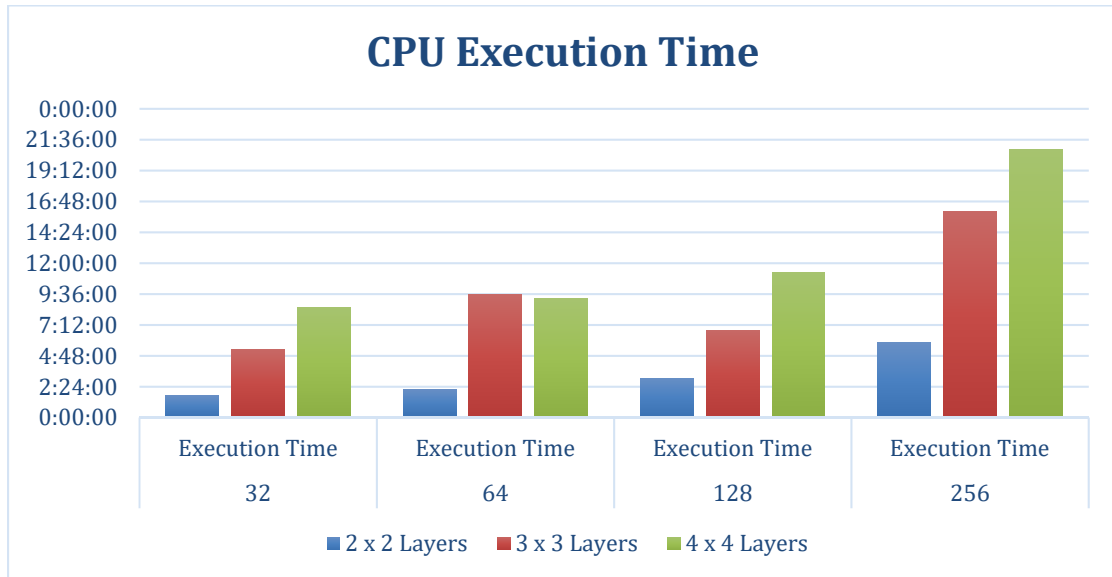


Figure. 44 Column Graph of CPU Execution between Layers

6.3 GPU Execution Time between Layers

Deep Layers	Residual Layers	Hidden Layers			
		32	64	128	256
		Execution Time	Execution Time	Execution Time	Execution Time
2 x	2 Layers	2:37:24	2:37:39	2:26:23	2:40:05
3 x	3 Layers	6:06:51	6:16:36	6:04:10	6:09:33
4 x	4 Layers	12:11:21	11:48:27	12:19:01	12:30:06

Table. 8 Execution rate between Layers in GPU

In the Table. 8, the execution time of 2 x 2 layers, 3 x 3 layers and 4 x 4 layers along with 32 hidden layers, 64 hidden layers, 128 hidden layers and 256 hidden layers are shown which are done by the GPU cluster.

The graphical representation of the execution time of each layer along with hidden layers are shown as bubble chart in Figure. 45 and as column bar graph in Figure. 46. Here we found an interesting thing. The deep layers of network takes a certain amount of GPU execution time which is irrelevant of the number of deep layers. As shown in Figure. 46, the execution time of 4 x 4 layers of network is approximately same for all the 4 hidden layers which can be referenced from the Figure. 45 as well.

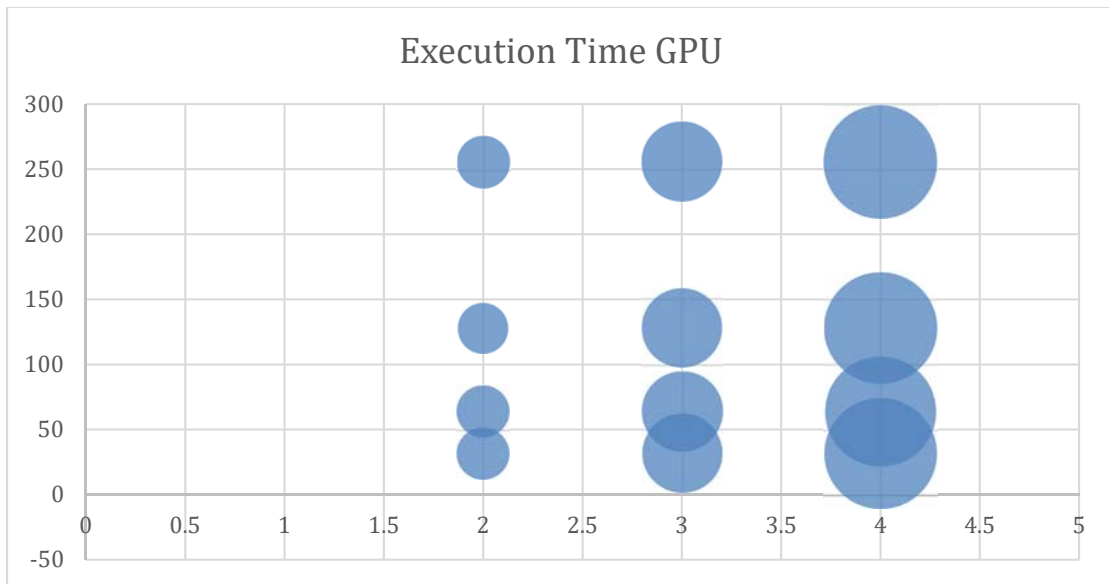


Figure. 45 Bubble Chart of GPU Execution

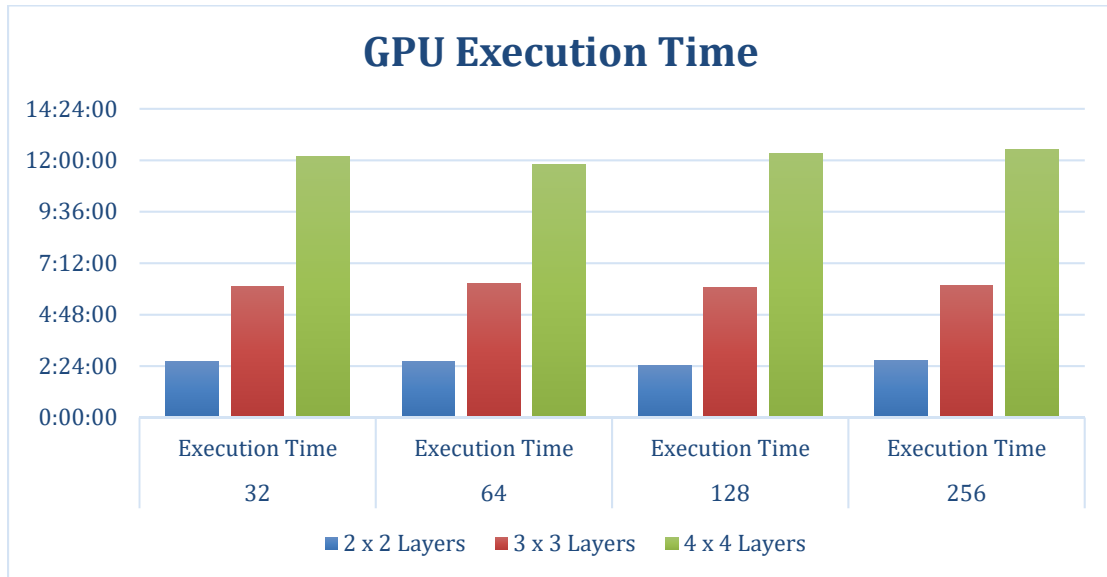


Figure. 46 Column Graph of GPU Execution between Layers

```

hpcmonster369@hpc369-Z10PE-D16-WS:~$ nvidia-smi
Mon Apr  8 12:58:58 2019

+-----+
| NVIDIA-SMI 410.104      Driver Version: 410.104      CUDA Version: 10.0      |
+-----+-----+
| GPU  Name                Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf    Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
+-----+-----+
|  0   Tesla K40c          Off          | 00000000:02:00:0 | Off      |
| 29%   61C   P0      69W / 235W | 124MiB / 11441MiB |  0%      | Default
+-----+-----+
|  1   Quadro P5000       Off          | 00000000:03:00:0 | Off      |
| 37%   58C   P0      45W / 180W | 15507MiB / 16278MiB | 18%      | Default
+-----+-----+
|  2   Quadro K620        Off          | 00000000:81:00:0 | On       |
| 34%   42C   P8       1W /  30W |  526MiB / 1999MiB |  0%      | N/A
+-----+-----+

+-----+
| Processes:                                     GPU Memory |
|  GPU       PID    Type   Process name                               Usage      |
+-----+-----+
|  0         28456   C     ...ter369/anaconda3/envs/tf_gpu/bin/python 113MiB    |
|  1         28456   C     ...ter369/anaconda3/envs/tf_gpu/bin/python 15495MiB  |
|  2          1340   G     /usr/lib/xorg/Xorg                          23MiB     |
|  2          1495   G     /usr/bin/gnome-shell                       53MiB     |
|  2          1703   G     /usr/lib/xorg/Xorg                          154MiB    |
|  2          1831   G     /usr/bin/gnome-shell                       201MiB    |
|  2          18491  G     /opt/google/chrome/chrome --              66MiB     |
|  2          28456   C     ...ter369/anaconda3/envs/tf_gpu/bin/python 18MiB     |
+-----+

```

Figure. 47 2 x 2 Layers GPU Utilization Snapshot

From Figure. 47, 48, 49 we found that the computational power of GPU is harnessed only by less than 1/3rd of the single GPU from the cluster. With the 2 x 2 layers, GPU utilization is 18%, where with 3 x 3 layers it increased a little to 22%, then with 4 x 4 layers it increased to 31%. The GPU architecture is working on the principle of SIMD vectorization.

SIMD processing exploits data-level parallelism. Data-level parallelism means that the operations required to transform a set of vector elements can be performed on all elements of the vector at the same time. That is, a single instruction can be applied to multiple data elements in parallel.

Support for SIMD operations is pervasive in the Cell Broadband Engine. In the PPE, they are supported by the Vector/SIMD Multimedia Extension instruction set. In the SPEs, they are supported by the SPU instruction set.

In both the PPE and SPEs, vector registers hold multiple data elements as a single vector. The data paths and registers supporting SIMD operations are 128 bits wide, corresponding to four full 32-bit words. This means that four 32-bit words can be loaded into a single register, and, for example, added to four other words in a different register in a single operation.

The process of preparing a program for use on a vector processor is called vectorization or SIMDization. It can be done manually by the programmer, or it can be done by a compiler that does auto-vectorization. Here GPU does the auto-vectorization process which is supported by 16 CPU core so that only 25 – 30 % computational power of GPU 0 is utilized where most of the CPU cores are utilizing 100% of their power which can be referenced from Figure. 50.


```

hpcmonster369@hpc369-Z10PE-D16-WS:~$ nvidia-smi
Wed Mar 27 23:54:17 2019

+-----+
| NVIDIA-SMI 410.104      Driver Version: 410.104      CUDA Version: 10.0      |
+-----+-----+-----+-----+-----+-----+
| GPU  Name                Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf    Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
+-----+-----+-----+-----+-----+-----+
|   0   Tesla K40c          Off          | 00000000:02:00.0 Off  |   0          0      |
| 27%   56C    P0         68W / 235W | 124MiB / 11441MiB |    0%      Default  |
+-----+-----+-----+-----+-----+-----+
|   1   Quadro P5000        Off          | 00000000:03:00.0 Off  |   0          0      |
| 32%   52C    P0         47W / 180W | 15507MiB / 16278MiB |   22%      Default  |
+-----+-----+-----+-----+-----+-----+
|   2   Quadro K620         Off          | 00000000:81:00.0 On   |   0          0      |
| 34%   37C    P8          1W / 30W | 374MiB / 1999MiB  |    3%      Default  |
+-----+-----+-----+-----+-----+-----+

+-----+
| Processes:                                                       GPU Memory |
|  GPU       PID    Type   Process name                               Usage      |
+-----+-----+-----+-----+-----+-----+
|    0       9024    C     ...ter369/anaconda3/envs/tf_gpu/bin/python  113MiB    |
|    1       9024    C     ...ter369/anaconda3/envs/tf_gpu/bin/python 15495MiB  |
|    2       1340    G     /usr/lib/xorg/Xorg                          23MiB     |
|    2       1495    G     /usr/bin/gnome-shell                        53MiB     |
|    2       1703    G     /usr/lib/xorg/Xorg                          105MiB    |
|    2       1831    G     /usr/bin/gnome-shell                        108MiB    |
|    2       2351    G     /opt/google/chrome/chrome --               48MiB     |
|    2       9024    C     ...ter369/anaconda3/envs/tf_gpu/bin/python  18MiB     |
+-----+-----+-----+-----+-----+-----+

```

Figure. 48 3 x 3 Layers GPU Utilization Snapshot

```

hpcmonster369@hpc369-Z10PE-D16-WS:~$ nvidia-smi
Fri Mar 29 14:43:41 2019

+-----+
| NVIDIA-SMI 410.104      Driver Version: 410.104      CUDA Version: 10.0      |
+-----+-----+-----+-----+-----+-----+
| GPU  Name                Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf    Pwr:Usage/Cap|     Memory-Usage | GPU-Util  Compute M. |
+-----+-----+-----+-----+-----+-----+
|   0   Tesla K40c          Off          | 00000000:02:00.0 Off  |   0%      Default  |
| 29%   60C    P0      69W / 235W | 124MiB / 11441MiB |           |           |
+-----+-----+-----+-----+-----+-----+
|   1   Quadro P5000       Off          | 00000000:03:00.0 Off  |  31%      Default  |
| 38%   60C    P0      48W / 180W | 15507MiB / 16278MiB |           |           |
+-----+-----+-----+-----+-----+-----+
|   2   Quadro K620        Off          | 00000000:81:00.0 On   |   0%      Default  |
| 34%   41C    P8       1W /  30W |  606MiB / 1999MiB |           |           |
+-----+-----+-----+-----+-----+-----+

+-----+
| Processes:                                                       GPU Memory |
|  GPU       PID    Type   Process name                               Usage    |
+-----+-----+-----+-----+-----+-----+
|    0       29328    C     ...ter369/anaconda3/envs/tf_gpu/bin/python  113MiB |
|    1       29328    C     ...ter369/anaconda3/envs/tf_gpu/bin/python 15495MiB|
|    2        1340    G     /usr/lib/xorg/Xorg                          23MiB   |
|    2        1495    G     /usr/bin/gnome-shell                        53MiB   |
|    2        1703    G     /usr/lib/xorg/Xorg                          215MiB  |
|    2        1831    G     /usr/bin/gnome-shell                        153MiB  |
|    2        2351    G     /opt/google/chrome/chrome --               133MiB  |
|    2       29328    C     ...ter369/anaconda3/envs/tf_gpu/bin/python  18MiB   |
+-----+-----+-----+-----+-----+-----+

```

Figure. 49 4 x 4 Layers GPU Utilization Snapshot

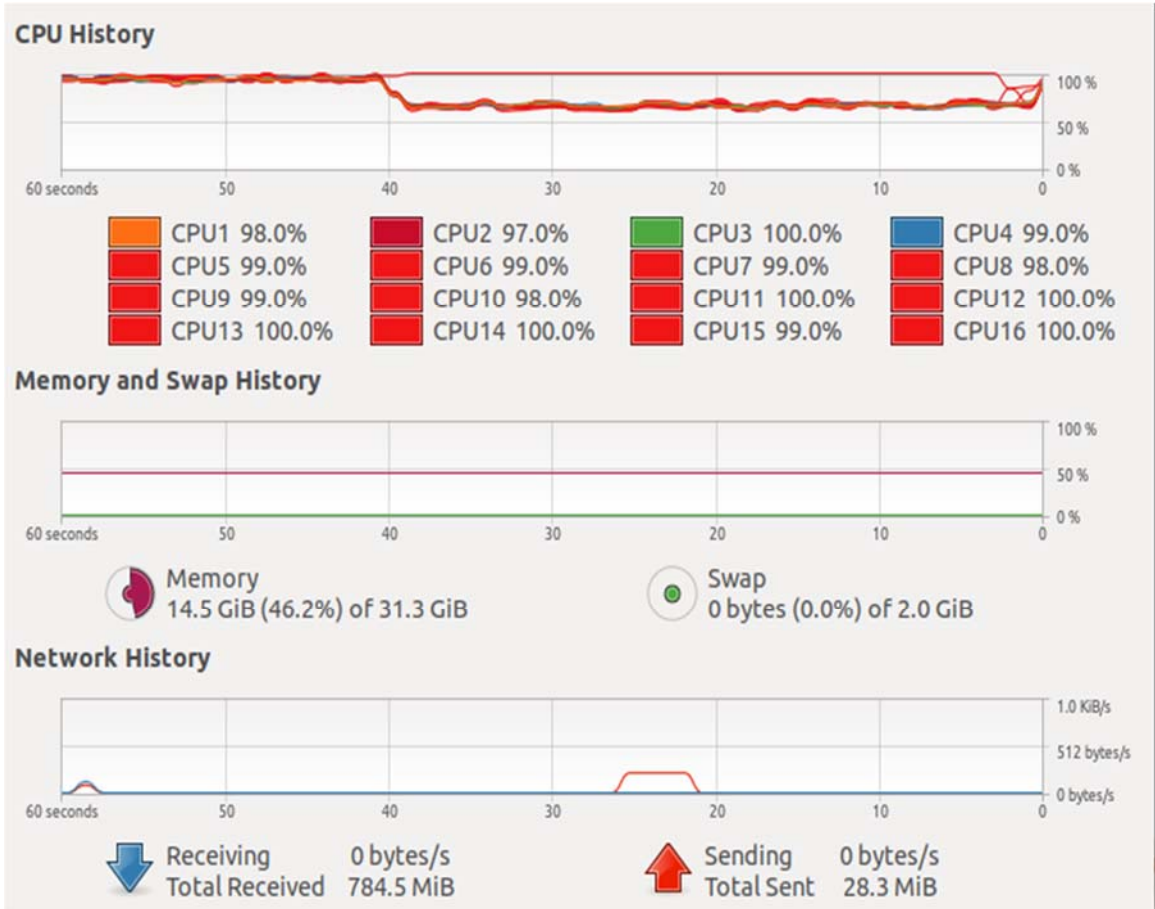


Figure. 50 3 x 3 Layers CPU Utilization Snapshot

6.4 Bidirectional Vs Non-Bidirectional Execution between Layers

De ep La yer s	Resi dual Lay ers	Bidire ctional	Hidden Layers							
			32		64		128		256	
			Predic tion Accur acy	Exec ution Time	Predic tion Accur acy	Exec ution Time	Predic tion Accur acy	Execu tion Time	Predic tion Accur acy	Exec ution Time
3	3	TRUE	0.910 07805	5:18: 24	0.911 43537	9:34: 22	0.913 13201	6:44:2 6	0.182 21921	16:0 0:55
3	3	FALS E	0.932 13439	2:47: 35	0.883 94976	3:52: 52	0.920 93652	6:03:3 8	0.182 21921	21:5 1:49

Table. 9 Bidirectional vs Non-bidirectional layers execution time

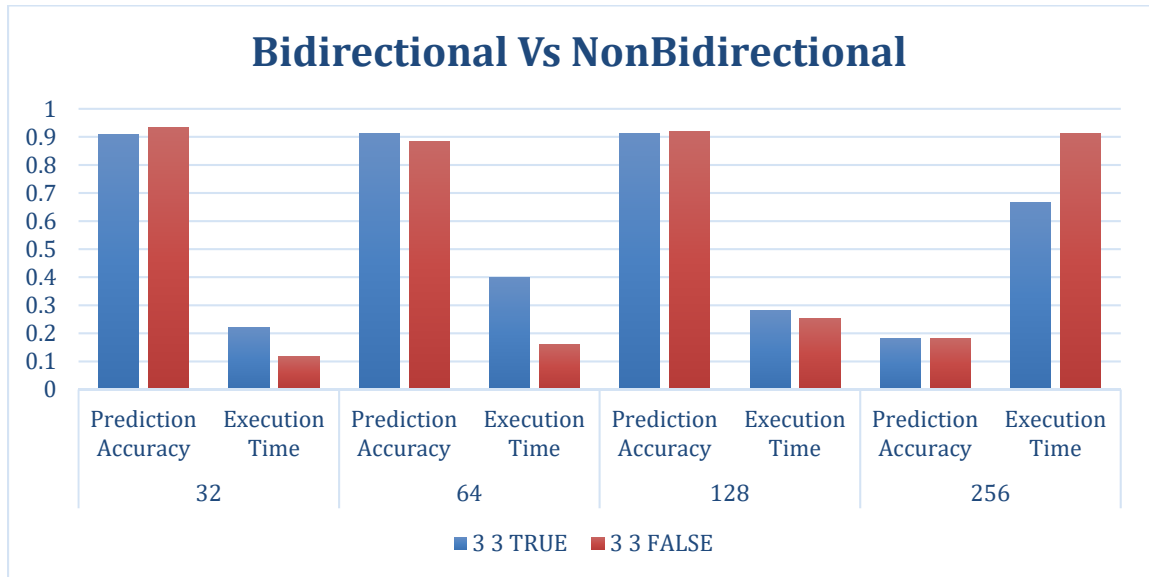


Figure. 51 Column Graph of Execution time between bidirectional and non-bidirectional

In the Table. 9, execution time along with prediction accuracy is shown in 3 x 3 deep residual layers where one learning is by bidirectional while the other learning is by non-bidirectional or unidirectional. When we observe the result of prediction accuracy in Figure. 51, with 32 layers of hidden layer unidirectional layers gave better prediction accuracy but as deep layers increased to 64 layers and 128 layers the bidirectional layers gave better prediction accuracy but on the cost of higher execution time as it's almost twice the LSTM layers as compared to unidirectional LSTM. As the deep layers with hidden layers increased to a certain point of threshold tensors both the iterations failed to give the expected prediction accuracy. So from this result we found out for each experiment there is a high efficiency level above which it doesn't matter to the architectural level.

6.5 Stack Bidirectional Vs Stack Non-Bidirectional Execution between Layers

Depth Layers	Residual Layers	Bidirectional	Hidden Layers							
			32		64		128		256	
			Prediction Accuracy	Execution Time	Prediction Accuracy	Execution Time	Prediction Accuracy	Execution Time	Prediction Accuracy	Execution Time
3	0	TRUE	0.88768238	1:30:33	0.91754329	2:12:40	0.90295213	3:18:13	0.892772317	9:28:29
3	0	FALSE	0.91177469	0:53:41	0.8995589	1:20:42	0.91923988	2:44:05	0.182219207	7:59:22

Table. 10 Stack Bidirectional vs Stack Non-bidirectional execution time

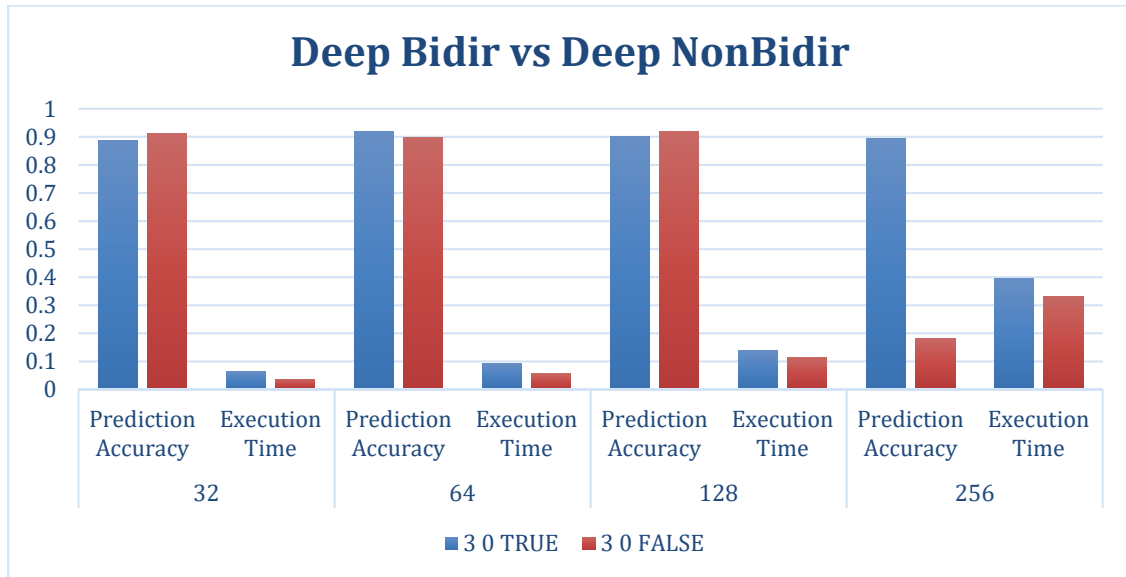


Figure. 52 Deep Bidirectional vs Deep Non-bidirectional Execution time

In the Table. 10, the experiment is carried out between stack layers or deep layers with bidirectional and non-bidirectional communications. The Figure. 52 show that, the prediction accuracy is better with unidirectional communication between layers with lesser hidden layers, as the network goes more and more with more hidden layers the bidirectional accuracy degrades. The bidirectional communication gives much better result as compared to single directional communications in 256 hidden layers which is a very good prediction accuracy.

6.6 Best Accuracy between all Layers

Deep Layers	Residual Layers	Bidirectional	Hidden Layers							
			32		64		128		256	
			Prediction Accuracy	Execution Time	Prediction Accuracy	Execution Time	Prediction Accuracy	Execution Time	Prediction Accuracy	Execution Time
3	0	TRUE	0.887 68238	1:30: 33	0.917 54329	2:12: 40	0.902 95213	3:18:1 3	0.8927 72317	9:28: 29
3	0	FALSE	0.911 77469	0:53: 41	0.899 5589	1:20: 42	0.919 23988	2:44:0 5	0.1822 19207	7:59: 22
3	3	FALSE	0.932 13439	2:47: 35	0.883 94976	3:52: 52	0.920 93652	6:03:3 8	0.1822 19207	21:5 1:49
3	3	TRUE	0.910 07805	5:18: 24	0.911 43537	9:34: 22	0.913 13201	6:44:2 6	0.1822 19207	16:0 0:55

Table. 11 Best Accuracy in 3 deep layers

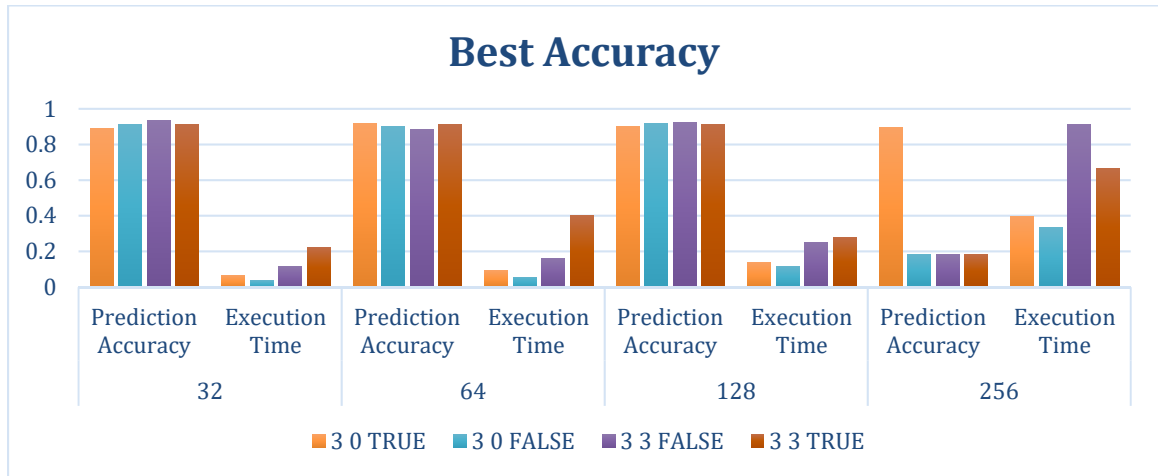


Figure. 53 Best Accuracy among all types of 3 stacked layers

In Figure. 53, the prediction accuracy along with execution time of each 3 deep layers with residual layer or without residual layer, with bidirectional or without bidirectional layers is calculated to give an overview of better layer for this experiment. If we observe the 32 hidden layers, 3 x 3 bidirectional false gives the best prediction accuracy with good execution time, with 64 hidden layers 3 x 3 bidirectional gives better prediction accuracy than any of them as proved in previous section, with 128 hidden layers, 3 x 3 non-bidirectional gives best accuracy but with higher execution time where 3 x 0 gives almost similar prediction accuracy with half of the execution time, with 256 hidden 3 x 0 bidirectional layer is the clear winner as compare to others in terms of prediction accuracy and execution time.

6.7 Between Deep Residual Bidirectional between 3 x3 and 4 x4

Deep Layers	Residual Layers	Bidirectional	Hidden Layers							
			32		64		128		256	
			Prediction Accuracy	Execution Time	Prediction Accuracy	Execution Time	Prediction Accuracy	Execution Time	Prediction Accuracy	Execution Time
3	3	TRUE	0.910 07805	5:18: 24	0.911 43537	9:34: 22	0.913 13201	6:44: 26	0.1822 19207	16:0 0:55
4	4	TRUE	0.914 48933	8:35: 14	0.875 80591	9:14: 42	0.904 30945	11:1 8:50	0.1822 19207	20:5 0:11

Table. 12 Deep Residual 3 x3 vs 4 x 4 layers

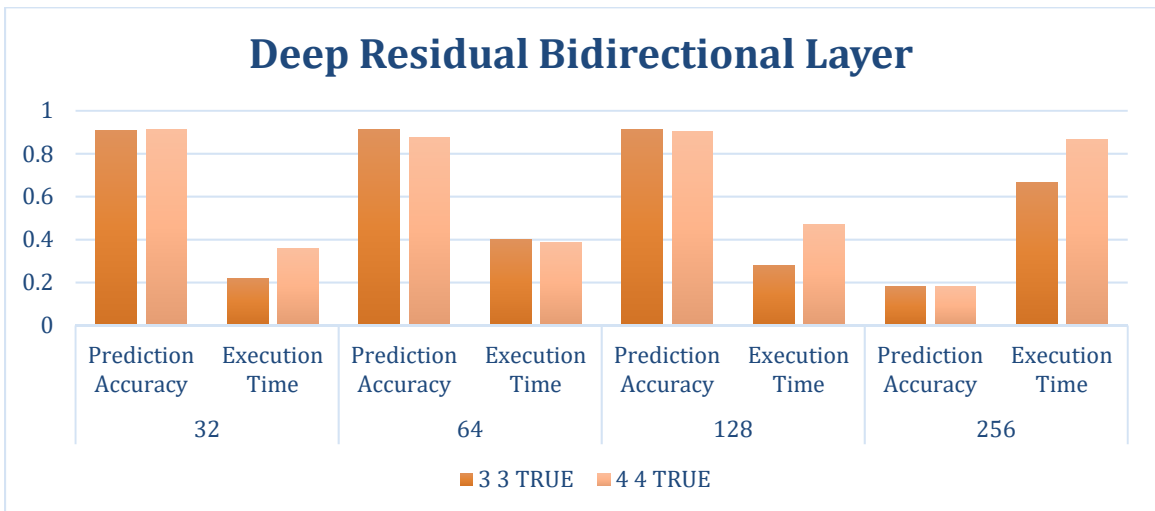


Figure. 54 Column graph of 3 x 3 vs 4 x 4 Deep Residual Layers

In Figure. 54, we are trying to show how deep residual bidirectional layers going to behave in a more complex network for this it is used all over the researches. We found that with our test data and higher layers of bidirectional communication the execution increases rapidly but the prediction accuracy dropped with giving the same test result as 3 x 3 layers. We might need more datasets with more complex architecture to test this feature.

6.8 Between Deep Layer vs Prediction Accuracy vs Exe Time in CPU

Deep Layer s	Hidden Layers							
	32		64		128		256	
	Predicti on Accuracy	Execut ion Time	Predicti on Accuracy	Execut ion Time	Predicti on Accuracy	Execut ion Time	Predicti on Accuracy	Execut ion Time
2 Layer s	0.92229 3842	1:42:2 1	0.92466 9147	2:09:2 2	0.87750 2561	3:04:5 1	0.18221 9207	5:49:4 0
3 Layer s	0.91007 8049	5:18:2 4	0.91143 5366	9:34:2 2	0.91313 2012	6:44:2 6	0.18221 9207	16:00: 54
4 Layer s	0.91448 9329	8:35:1 4	0.87580 5914	9:14:4 2	0.90430 9452	11:18: 50	0.18221 9207	20:50: 11

Table. 13 Execution matrix of all layers by CPU

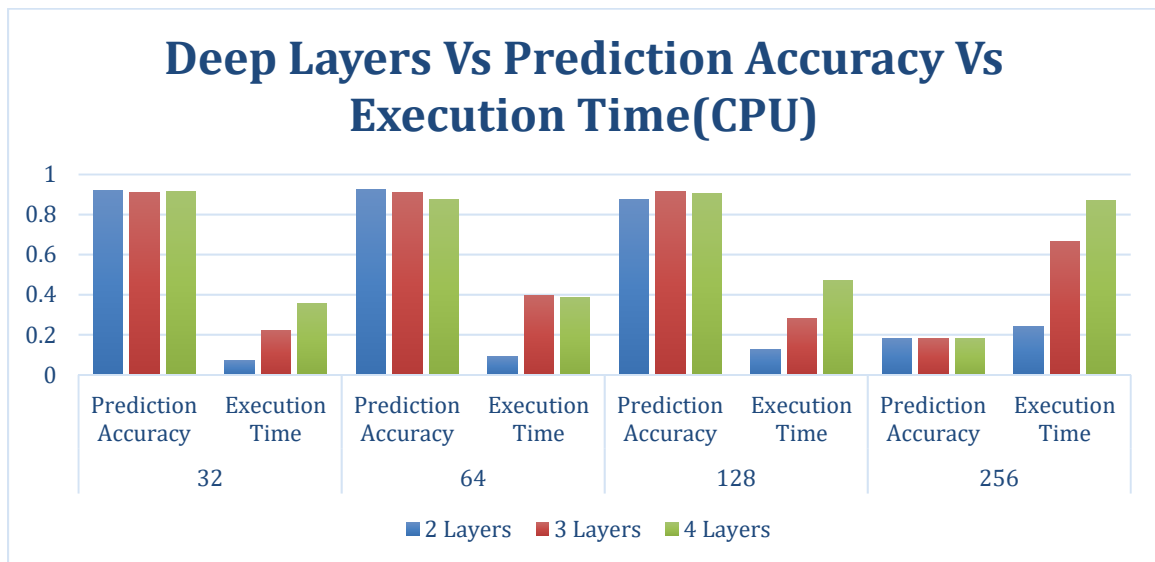


Figure. 55 CPU Execution Graph for all Layers

In the Figure. 55, we have shown the prediction accuracy along with execution time of different layers which are 2 x 2 layers, 3 x 3 layers and 4 x 4 layers. With 32 hidden layers, the 2 x 2 architecture performed better with higher prediction accuracy than other layers, with 64 layers we observed 2 x 2 layers performance improved with good prediction accuracy of 0.9246, with 128 hidden layers the 3 x 3 architecture started performing better than others with 0.9131 prediction accuracy, with 256 hidden layers 4 x 4 architecture started giving better prediction accuracy than other layers which is 0.2143 but it's still a low prediction accuracy. The execution of 4 x 4 layer is always higher as it needs more hidden layers to iterate over deep learning model. This experiment concluded that our research data performs expectedly well with 3 x 3 architecture. We might need a bigger dataset to check the feasibility over 4 x 4 layer.

6.9 Between Deep Layer vs Prediction Accuracy vs Exe Time in GPU

Deep Layer s	Hidden Layers		Hidden Layers		Hidden Layers		Hidden Layers	
	32		64		128		256	
	Predicti on Accuracy	Executi on Time	Predicti on Accuracy	Executi on Time	Predicti on Accuracy	Executi on Time	Predict ion Accuracy	Execu tion Time
2 Layer s	0.90125 5488	2:37:24	0.93281 3048	2:37:39	0.90464 8781	2:26:23	0.1822 19207	2:40:0 5
3 Layer s	0.89345 0975	6:06:51	0.92534 7805	6:16:36	0.88327 1098	6:04:10	0.1822 19207	6:09:3 4
4 Layer s	0.90770 2744	12:11:2 2	0.92025 7866	11:48:2 7	0.18221 9207	12:19:0 1	0.1822 19207	12:30: 06

Table. 14 Execution matrix of all layers by GPU

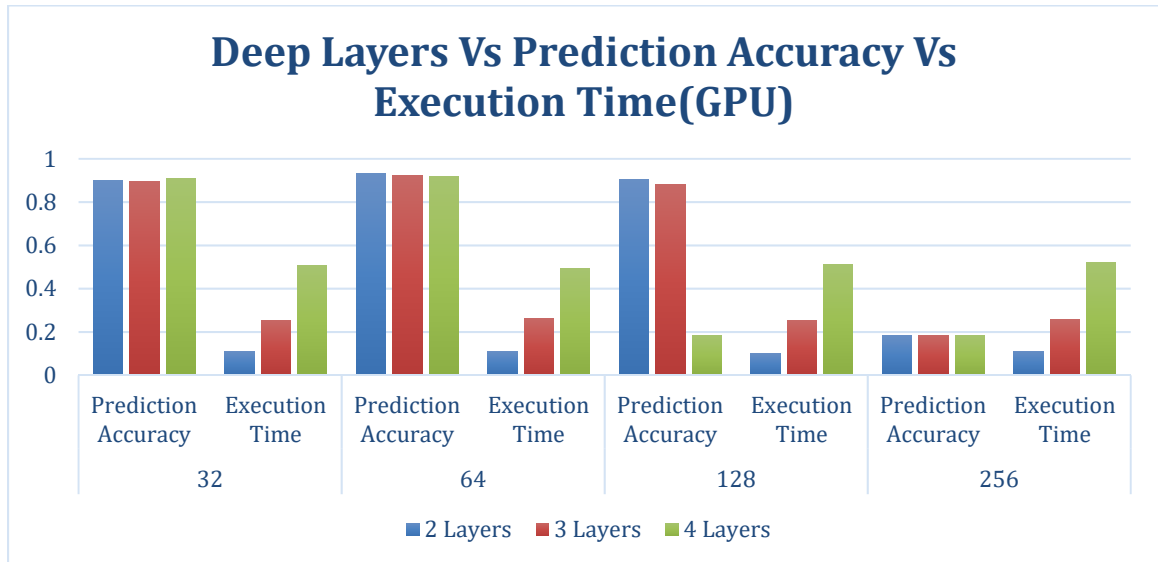


Figure. 56 GPU Execution Graph for all Layers

The Figure. 56, has shown the prediction accuracy and execution time between different layers such as 2 x 2, 3 x 3, and 4 x 4 with 32, 64, 128 and 256 hidden layers with the computational power of the GPU cluster. With GPU the 4 x 4 layer started performing with higher prediction accuracy of 0.9077 than other layers in 32 layers hidden network, with 64 hidden layers the 2 x 2 layer gives better prediction accuracy of 0.9338 than other layers, with 128 hidden layers the 2 x 2 results much better with good prediction accuracy of 0.9046 and outstanding execution time which is overall same as explained in previous section, with 256 layers of hidden network all layers failed to give an expected prediction accuracy but the execution improved drastically over CPU. The execution time just dropped to half with GPU cluster as compared to CPU.

6.10 Deep Layer CPU Execution

D ee p La ye rs	Re sid ual La yer s	Har dw are	Hidden Layers											
			8		16		32		64		128		256	
			Pre dict ion Acc ura cy	Ex ecu tio n Ti me	Pre dic tio n Acc ura cy	Ex ecu tio n Ti me	Pre dict ion Acc ura cy	Ex ecu tio n Ti me	Pre dic tio n Acc ura cy	Ex ecu tio n Ti me	Pre dict ion Acc ura cy	Ex ecu tio n Ti me	Pre dict ion Acc ura cy	Ex ecu tio n Ti me
4	4	CP U	0.9 182 218 91	7:5 8:2 1	0.9 04 98 81 1	7:5 8:4 6	0.9 144 893 29	8:3 5:1 4	0.8 75 80 59 1	9:1 4:4 2	0.9 043 094 52	11: 18: 50	0.1 822 192 07	20: 50: 11

Table. 15 4 x 4 deep Layers CPU execution matrix

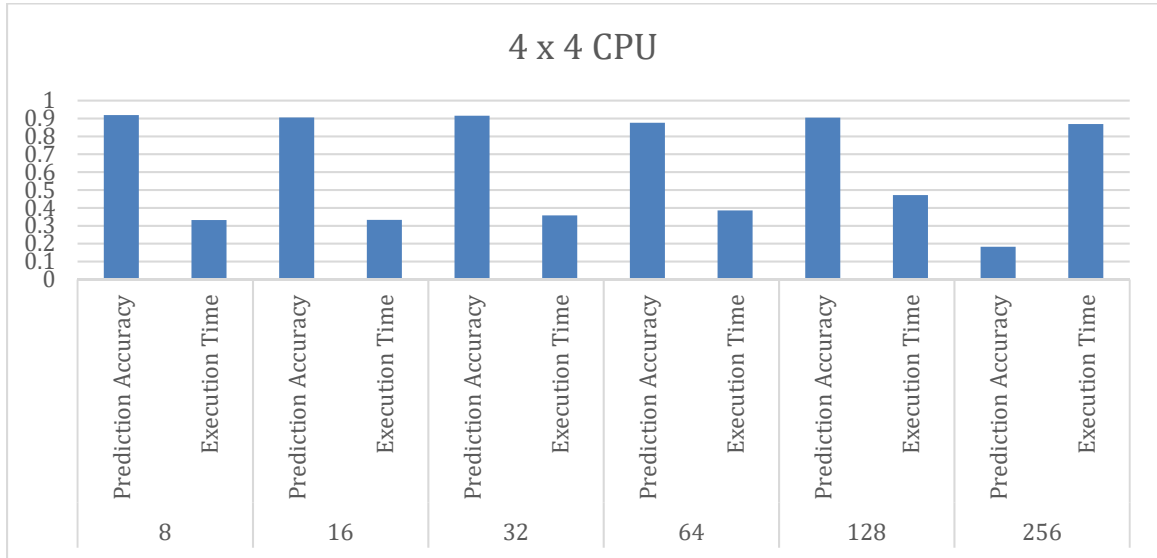


Figure. 57 Column Graph of 4 x 4 deep Layers CPU execution

In Figure. 57, we have performed the experiment of 4 x 4 deep layers with very low hidden layers to higher hidden layers to check the prediction accuracy. With 8 layers of hidden layers, the accuracy is better as compared to other hidden layers network. As the layers increase over time, the prediction accuracy started to drop. In this experiment, we found that 16 layers and 128 layers the prediction accuracy is almost equivalent but the execution time difference is increased to 25% more.

6.11 Lower GPU vs Higher CPU

Deep Layer s	Residual Layer s	Hard ware	Hidden Layers							
			32		64		128		256	
			Prediction	Execution	Prediction	Execution	Prediction	Execution	Prediction	Execution

			Accur acy	Time	Accur acy	Time	Accu racy	Time	Accur acy	Time
3 x	3	GPU	0.893 4509 75	6:06: 51	0.925 3478 05	6:16: 36	0.88 3271 1	6:04: 10	0.182 2192 07	6:09: 33
4 x	4	CPU	0.914 4893 29	8:35: 14	0.875 8059 14	9:14: 42	0.90 4309 45	11:18 :50	0.182 2192 07	12:30 :06

Table. 16 3 x 3 Layer GPU vs 4 x4 Layer CPU execution matrix

In Table. 16, it has shown the data of execution with 3 x 3 deep layers of GPU and 4 x 4 layers with CPU to evaluate the beneficial power of GPU over CPU for more complex network with bigger dataset. We found that the prediction accuracy of 4 x 4 CPU is more with 32 layers, then with 64 layers the 3 x 3 GPU gives better prediction accuracy than CPU of 4 x 4, with 128 hidden layers the CPU performs better with higher prediction accuracy of 0.9043 as compared to GPU with 2 times execution time taken from GPU which is not good. In Figure. 58, by adding 256 layers the execution speed of 4 x 4 is increased to double but failed to give an expected prediction accuracy on both the CPU and GPU models.

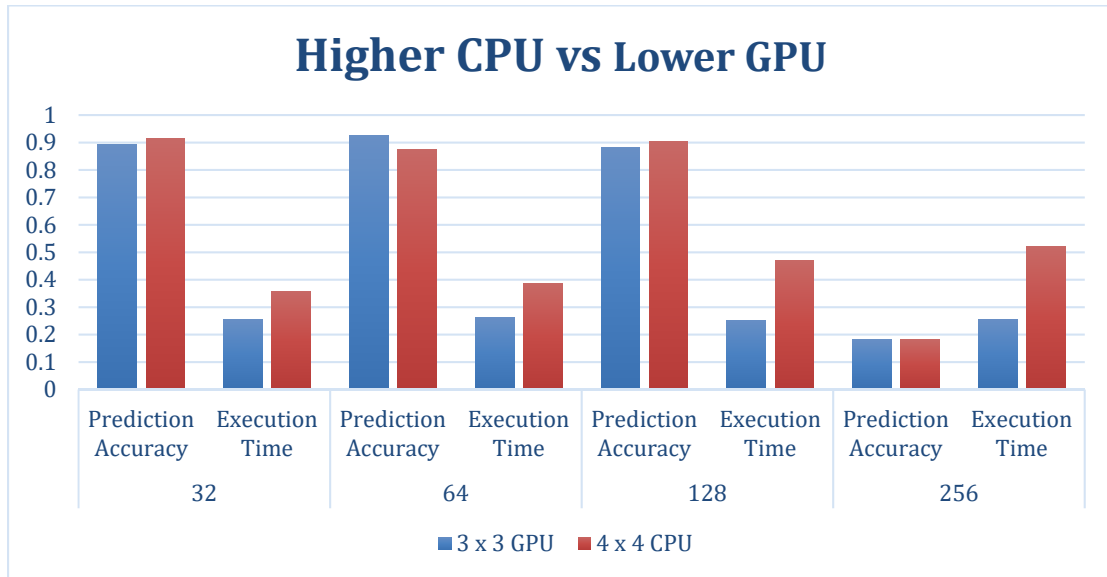


Figure. 58 Column Graph of 3 x 3 GPU vs 4 x 4 CPU Execution Result

6.12 4 x 4 CPU vs 4 x 4 GPU Layers

Deep Layer s	Residual Layer s	Hard ware	Hidden Layers							
			32		64		128		256	
			Prediction Accuracy	Execution Time	Prediction Accuracy	Execution Time	Prediction Accuracy	Execution Time	Prediction Accuracy	Execution Time
4 x	4	CPU	0.914489329	8:35:14	0.875805914	9:14:42	0.90430945	11:18:50	0.182219207	20:50:11
4x	4	GPU	0.9077027	12:11:22	0.9202578	11:48:27	0.182219	12:19:01	0.1822192	12:30:06

			44		66		2		07	
--	--	--	----	--	----	--	---	--	----	--

Table.17 4 x 4 Layers CPU vs GPU Execution

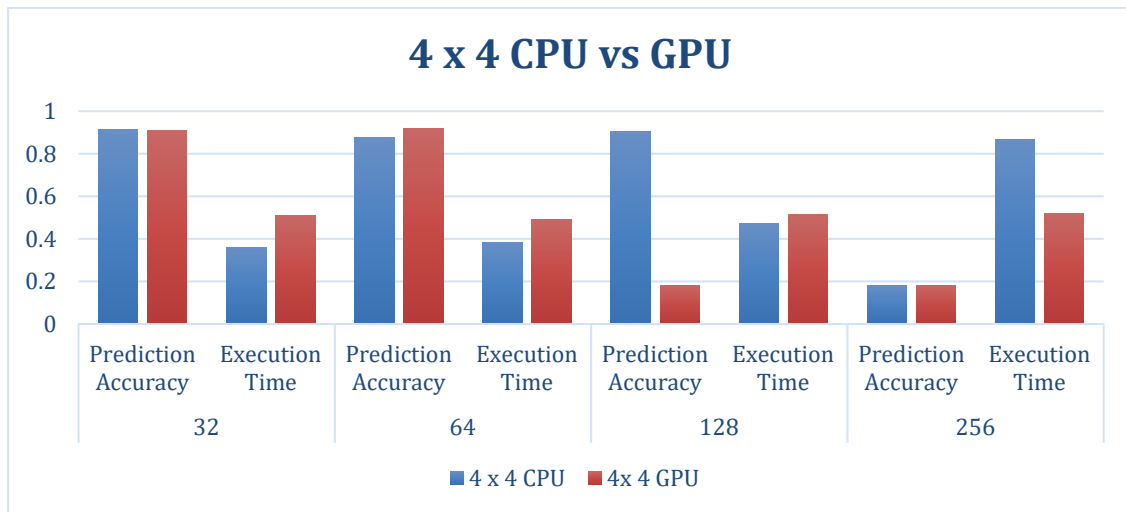


Figure. 59 Graph of 4 x 4 deep layers CPU vs GPU Execution

In Figure. 59, we have shown the computational power of both CPU and GPU of a complex deep learning model with 4 x 4 architecture. With 32 nodes of hidden layers, the CPU performs better with higher prediction accuracy of 0.9144, with 64 nodes of hidden layers the GPU is giving better result with 0.9205 with same execution of previous hidden layers, with 128 layers CPU performance is 0.9043 which is outstanding as compared to GPU which is just 0.18. When we went to 256 hidden layers, the GPU and CPU in 4 x 4 failed to get expected results as both gave the same prediction accuracy of 0.18, but the execution time of CPU is almost 100% more as compared to GPU execution time. So in this experiment, 64 hidden layers GPU is the selected result which is 92%.

6.13 Bidirectional Lower vs Stack Higher Layers

D ee p L ay er s	Resid ual Layer	Bidire ctiona l	Hidden Layers		Hidden Layers		Hidden Layers		Hidden Layers	
			Predi ction Accu racy	Exec ution Time	Predic tion Accur acy	Execu tion Time	Predi ction Accu racy	Exec ution Time	Predict ion Accura cy	Exec ution Time
			32		64		128		256	
2 x	2	TRU E	0.922 2938 4	1:42: 21	0.924 66914 7	2:09: 22	0.87 7502 56	3:04: 51	0.1822 19207	5:49: 40
3 x	3	FALS E	0.932 1343 9	2:47: 35	0.883 94975 7	3:52: 52	0.92 0936 52	6:03: 38	0.1822 19207	21:51 :49

Table. 18 2 x 2 Bidirectional Stack Layer vs 3 x 3 Stack Layer

In the Figure. 60, we have shown the 2 x 2 bidirectional layers with 3 x 3 non-bidirectional layers which having almost same computational powers as 2 x 2 with bidirectional gives 8 times of complex network over single LSTM cells where 3 x 3 non-bidirectional gives 9 times of complex network over single LSTM cell. With 32 hidden layers, 3 x3 layers gives better accuracy over 2 x 2 layers where 64 layers, 2 x 2 layers gives better result. When we consider the 128 hidden layers over 2 x 2 and 3 x 3 stacked

layers we found out that initial lower layers are used to learn the model and higher layers are used to calculate the accuracy of the model in this case 3 x 3 having more initial layers which gives a better learning to the model than 2 x 2 layers. In 256 layers, both the models failed to perform but when you see the execution time 3 x 3 layers execution time is very high as compared to 2 x 2 model which is almost 4 times of 2 x 2 models. So the conclusion we draw that with higher layers the model will learn much faster but with increase of complexity of hidden layers it will fail to pass the learning to the higher layers which takes more time as it becomes very slow to pass the information.

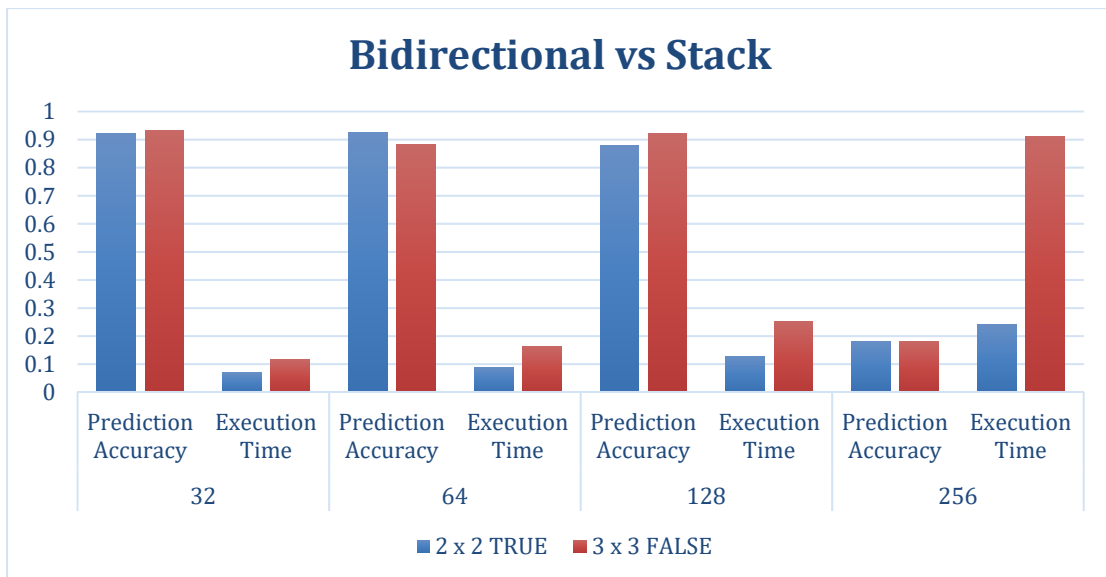


Figure. 60 Graph of 2 x 2 Bidirectional Stack Layer vs 3 x 3 Stack Layer

6.14 Stack vs Hidden layer on Execution Time and Prediction Accuracy

Layers	Hidden Layers	Execution Time	Prediction Accuracy
2 x 2	32	1:42:21	0.922293842
2 x 2	64	2:09:22	0.924669147
2 x 2	128	3:04:51	0.877502561
2 x 2	256	5:49:40	0.182219207
4 x 4	8	7:58:20	0.918221891
4 x 4	16	7:58:46	0.90498811
4 x 4	32	8:35:14	0.914489329
4 x 4	64	9:14:42	0.875805914

Table. 19 2 x 2 stacked hidden layers vs 4 x 4 stacked hidden layers

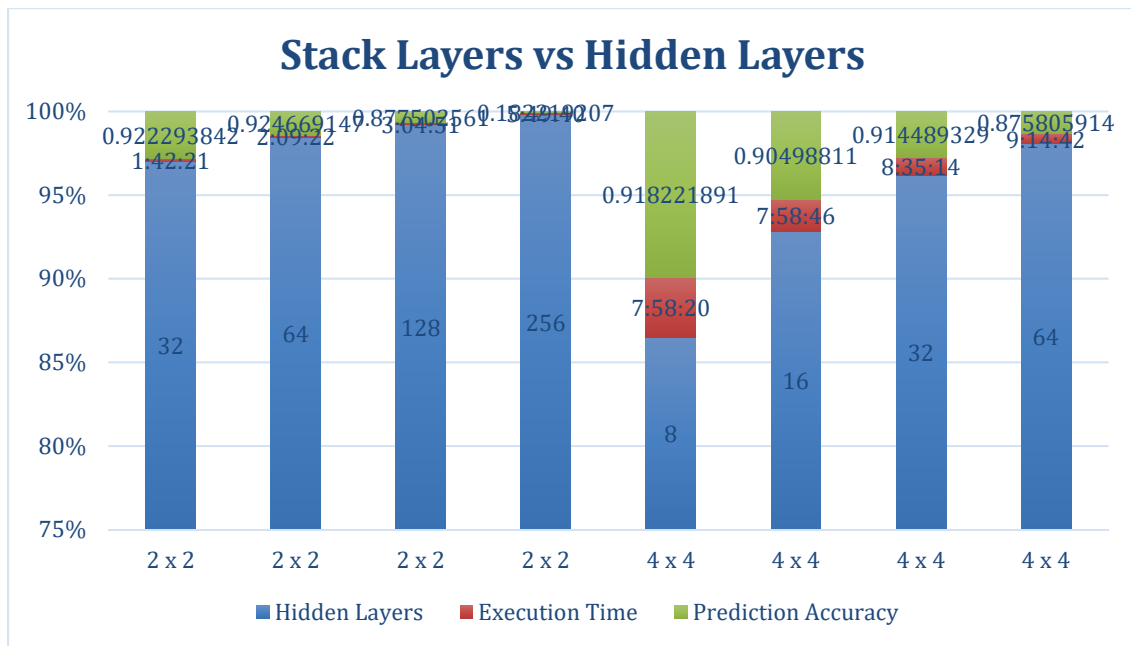


Figure. 61 Execution time Graph of 2 x2 vs 4x 4 stacked layers

In the Table 19, we have shown the matrix of 2 x 2 stack layers with higher hidden layers and compared with 4 x 4 stack layers with lower hidden layers which is having almost same computational power over single layer. We have used 2 x 2 layers with 32, 64, 128 and 256 hidden layers and 4 x 4 layers with 8, 16, 32 and 64 hidden layers. As shown in Figure 61, the biggest thing with higher layers is the execution time. The execution time is very high with high layers with lower hidden layers. With 4 x 4 layers the execution time is almost doubled as compared to the 2 x 2 layers. With time efficient, 2 x 2 layers are the clear winners in this part of research as shown in Figure. 62. For prediction accuracy, the 2 x 2 stacked layers with 32, 64 hidden layers gave better prediction accuracy as compared to 4 x 4 stacked layers with 8, 16 layers of hidden layers. So we concluded that lower models with higher hidden nodes provides better test results and execution time as compared to higher layers with less hidden nodes.

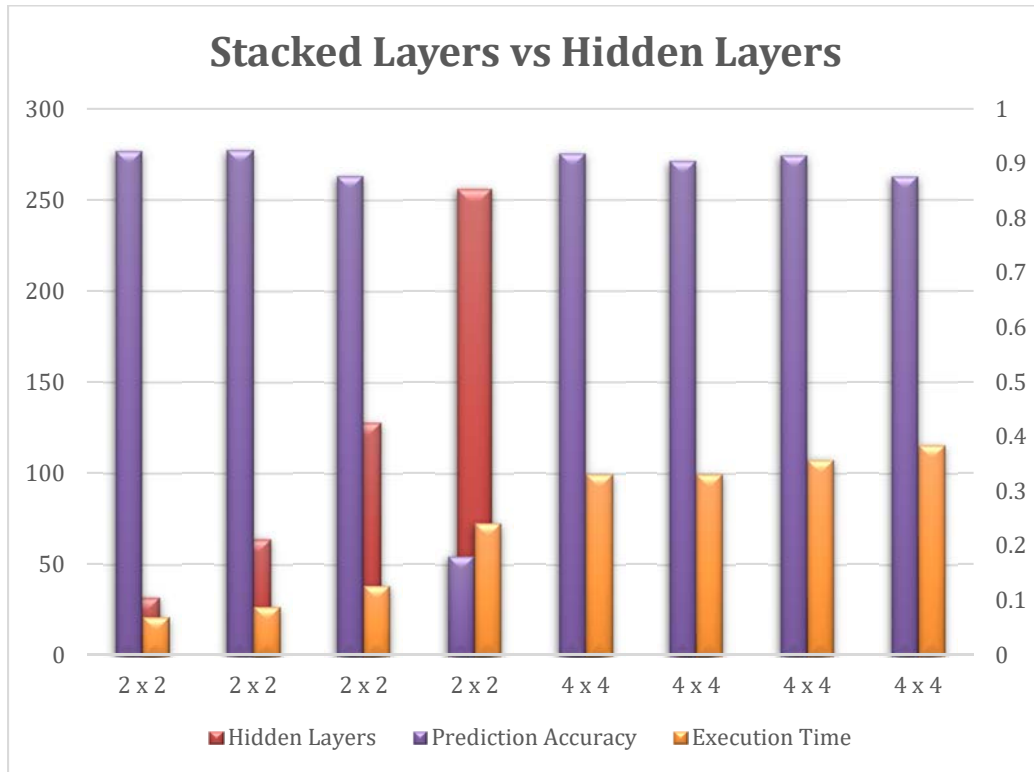


Figure. 62 Execution time graph with stack layers vs hidden layers

6.15 PyTorch vs TensorFlow Efficiency Comparison

Deep Layers	Language	Hidden Layers							
		32		64		128		256	
		Prediction Accuracy	Execution Time	Prediction Accuracy	Execution Time	Prediction Accuracy	Execution Time	Prediction Accuracy	Execution Time
3 Layers	PyTorch	0.9114 87694	0:40:53	0.9022 57844	0:51:39	0.9182 66254	2:21:13	0.1822 18305	6:58:31
3 Layers	TensorFlow	0.9117 74695	0:53:41	0.8995 58902	1:20:42	0.9192 39879	2:44:05	0.1822 19207	7:59:22

Table.20 Efficiency between PyTorch and TensorFlow

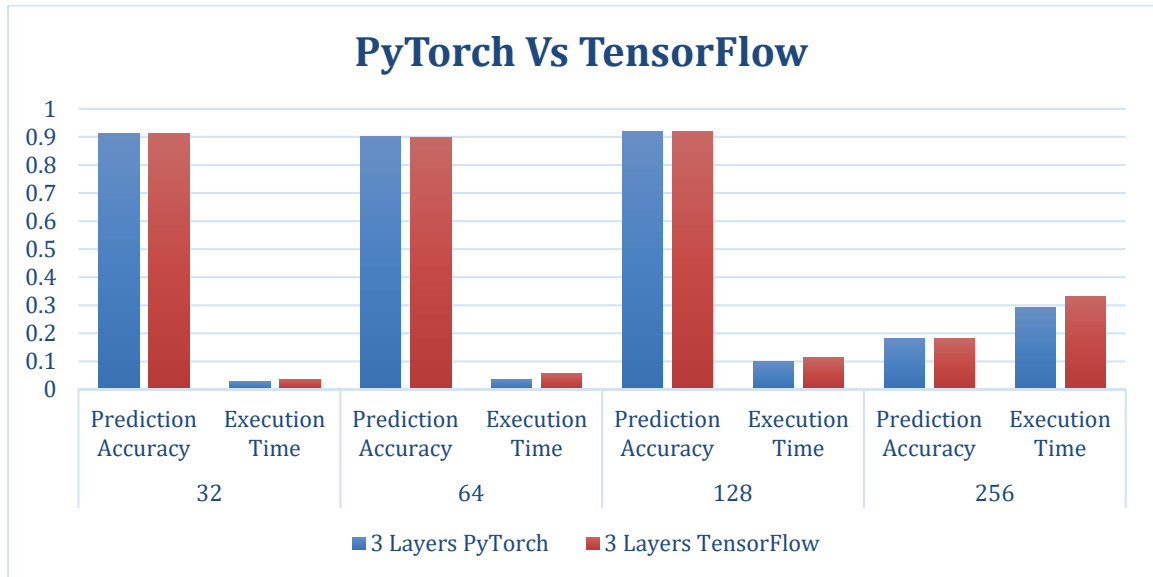


Figure. 63 Execution Graph between PyTorch and TensorFlow

In Table. 20, it shows the experiment result of 3 deep layered network with 32, 64, 128 and 256 hidden layers programmed using TensorFlow API and PyTorch API.

TensorFlow and PyTorch both are very good frameworks used by machine learning researchers for building deep neural networks. The major difference is TensorFlow core APIs are built using C++ and Python is used as wrapper on core to communicate with data where PyTorch is built on top of Torch framework with python wrapper. The best way to compare two frameworks is to code something up in both of them.

In Figure. 63, it displays our graphical representation of the Table. 20 data. We found out PyTorch is executing much faster than TensorFlow. The execution time is always lower than TensorFlow in all the hidden layers. The prediction accuracy is similar with TensorFlow. During the whole experiment, in the 128 hidden layer network PyTorch results marginally better prediction accuracy and less execution time than TensorFlow framework so PyTorch is the winner.

6.16 Raspberry PI Cluster vs Intel Xeon CPU Efficiency Comparison

Deep Layers	Hardware	Hidden Layers							
		32		64		128		256	
		Prediction Accuracy	Execution Time	Prediction Accuracy	Execution Time	Prediction Accuracy	Execution Time	Prediction Accuracy	Execution Time
3 Layers	PI Cluster	0.921709823	1:43:51	0.904981205	2:45:39	0.922189017	4:14:43	0.182218305	10:19:31
3 Layers	Intel Xeon CPU	0.911774695	0:53:41	0.899558902	1:20:42	0.919239879	2:44:05	0.182219207	7:59:22

Table.21 Efficiency between Raspberry Pi Cluster and Intel Xeon CPU

In this section, we did another experiment where we executed our LSTM deep learning model with same UCI dataset but with different hardware. In the previous section, we used the same dataset using same hardware but with different frameworks. One hardware would be a 16 threads multicore Intel Xeon CPU processor with 32 GB of memory and another hardware is 16 Raspberry Pi nodes cluster each having 1 GB RAM working in a cluster fashion made by parameter server architecture.

In the Table.21, we noted all our experiment results. Figure 64 is the graphical representation of our experimental result. We observed that in distributed machine learning, the accuracy of model is improved with more execution time. With 32 hidden layers, PI cluster give 92% accuracy in 1hour 43 minutes execution time where Intel CPU give 91% accuracy with 53 minutes execution time. The cluster is giving better prediction accuracy with high execution time than multicore Intel CPU. This is might be due to execution throughput with 16 PIs connecting together during the execution. With 256 layers, the accuracy is equivalent on both the hardware which just 18% but the execution time of cluster is higher than single Intel Xeon CPU. So we can draw the conclusion that with high power GPU clustered distributed machines this could be an efficient performance improvement which needs further research.

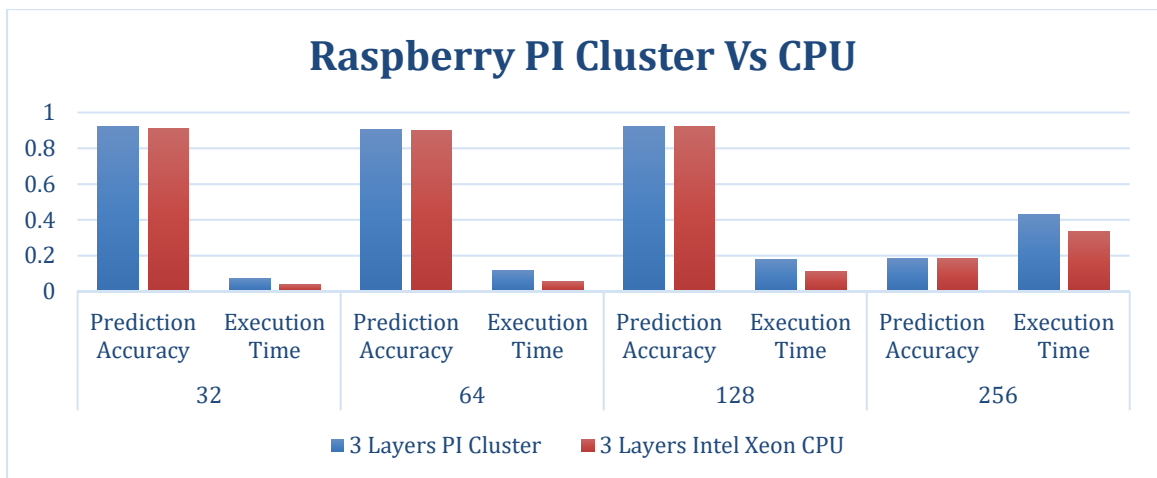


Figure.64 Execution graph between Pi Cluster vs Intel Xeon CPU

CHAPTER VII

CONCLUSION

7.1 Summary

In this thesis, we proposed a distributed deep learning model to solve a Human Activity Recognition (HAR) problem. We focused on the deep learning model using asynchronous parameter server architecture as well synchronous all-reduce approach. For this purpose, we have created the Raspberry Pi cluster using 16 Raspberry Pi nodes and the NVIDIA GPU cluster having 3 NVIDIA GPUs where both the systems are tested with distributed approach by using distributed TensorFlow API and PyTorch API. To work on the HAR problem, we have created a Residual Bidirectional LSTM to simulate HAR by using this distributed system. There are several points that we tested in this research thesis. First we created a multilayer deep learning model with 2 x 2, 3 x 3 and 4 x 4 architecture where stacked layers are compared with non-stacked layers, stacked layers are compared with residual bidirectional layers, residual bidirectional layers are compared with residual non-bidirectional layers. Those layers with different variations were executed over CPUs and GPUs to benchmark the performance. The small stacked layers with heavy hidden layers are compared with high stacked layers with less hidden layers on both CPU and GPU.

In this experiment, when we evaluated the execution times along with prediction accuracy over the multicore CPU and the GPU cluster using different programming APIs: TensorFlow and PyTorch. Note that the execution time of PyTorch over 3 x3 layer is faster than TensorFlow over the same layer. We have also tested the distributed TensorFlow framework in Raspberry Pi cluster to benchmark the CPU performance of 16- node Raspberry Pi cluster, where each Pi having 1 GB RAM, all-together 16 GB RAM equivalent with 32 GB Octacore Intel Xeon CPU. When the result of CPU computation of 2 x 2 layers over the multicore CPU is compared with the Raspberry Pi cluster, we found that in distributed network the execution time is almost twice than the multicore system due to high latency and low throughput.

After comparing all experiments, the result show that the implementation of distributed TensorFlow on the GPU cluster works much faster than on the multicore CPU for high number of stacked layers with multi hidden layers. But it takes approximately same time for less stacked layers and dense stacked layers. For lesser stacked layers, GPU computation is less efficient. The CPU computation gives better prediction accuracy with 3 x 3 stacked layers but execution time is 3 times slower than the GPU cluster

7.2 Future Work

There will be continuous efforts on developing PyTorch and TensorFlow programming frameworks with different computing structures. How PyTorch will behave with the GPU clusters would be another interesting study. There are multiple distributed architectures that need to be tested. This would not only require changes in the programming structure, but would also need more sophisticated multi GPU cluster hardware machines. For achieving efficiency in terms of optimal data movement this attempt would require multiple GPU units physically connected to each other like Raspberry Pi cluster or connected over the internet where bandwidth would be another parameter to do research. Due to limitation of GPUs with more machines, multi machine GPU cluster could be a

future work. With a Network Attached Storage (NAS) server along with Raspberry Pi cluster would become a more effective solution for storage problems. The distributed deep learning is just the beginning of a new dimension of research with massive scales datasets from different geographical areas.

APPENDIX A – TESTBED ARCHITECTURE

A.1 NVIDIA GPU MACHINE SETUP

In this single machine cluster, there are 3 NVIDIA GPU cards used. These GPUs are taken from different old machines and put together in this machine for clustering purpose.

1. NVIDIA Tesla K40c
2. NVIDIA Quadro p5000
3. NVIDIA K640

Different GPUs might be installed with different drivers in different machines. Those should be under one driver in a single machine cluster which should support all the GPUs.

The command to check if any driver already installed in your machine.

```
$ ubuntu-drivers devices
```

The command to show all NVIDIA drivers in your machine.

```
$ lspci -v | grep NVIDIA
```

Step-1 : Remove previous installations

The command removes if any older driver already installed.

```
$ sudo apt-get purge nvidia*
```

The command removes CUDA installation along with drivers as well.

```
$ sudo apt-get autoremove
```

The command checks what NVIDIA GPU cards the machine has as shown in Figure.

```
$ sudo lshw -c display
```



```
hpcmonster369@hpc369-Z10PE-016-MS:~$ sudo lshw -c display
*-display
  description: 3D controller
  product: GK110BGL [Tesla K40c]
  vendor: NVIDIA Corporation
  physical id: 0
  bus info: pci@0000:02:00.0
  version: a1
  width: 64 bits
  clock: 33MHz
  capabilities: pm nsi pcieexpress bus_master cap_list
  configuration: driver=nouveau latency=0
  resources: irq:40 memory:c6000000-c6ffffff memory:b0000000-bfffffff memory:c0000000-c1ffffff
*-display
  description: VGA compatible controller
  product: GP104GL [Quadro P5000]
  vendor: NVIDIA Corporation
  physical id: 0
  bus info: pci@0000:03:00.0
  version: a1
  width: 64 bits
  clock: 33MHz
  capabilities: pm nsi pcieexpress vga_controller bus_master cap_list rom
  configuration: driver=nouveau latency=0
  resources: irq:48 memory:c4000000-c4ffffff memory:90000000-9fffffff memory:a0000000-a1ffffff ioport:6000(size=128) memory:c5000000-c507ffff
*-display
  description: VGA compatible controller
  product: GM107GL [Quadro K620]
  vendor: NVIDIA Corporation
  physical id: 0
  bus info: pci@0000:01:00.0
  version: a2
  width: 64 bits
  clock: 33MHz
  capabilities: pm nsi pcieexpress vga_controller bus_master cap_list rom
  configuration: driver=nouveau latency=0
  resources: irq:54 memory:fa000000-faffffff memory:e0000000-efffffff memory:f0000000-f1ffffff ioport:f000(size=128) memory:fb000000-fb07ffff
```

Figure.65 NVIDIA GPU Cards

You can see in the Figure.65, those are default “driver= nouveau” that means NVIDIA driver is not installed in this machine.

There are 2 ways to install NVIDIA driver in machine.

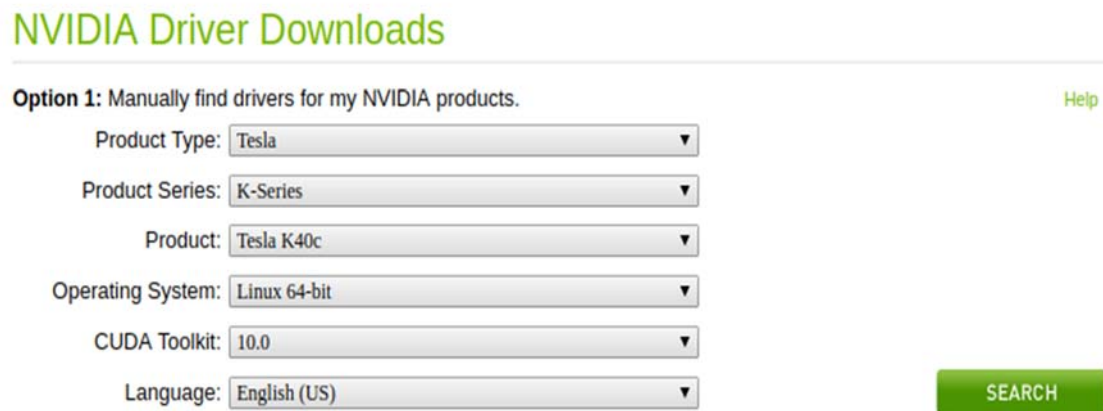
First one is to install from PPA drivers which is third party compatible with all NVIDIA GPUs. The Second one is installing from NVIDIA website by manually checking each GPU model with driver compatibility.

The advantage of ppa is easy and it automatically keeps updating if the creator adds new versions. Ubuntu integrates video into kernel with dpkg. If you install directly from NVIDIA, you still have to manually rerun that part of install task with each kernel update otherwise video stops working. With PPA it’s automatic. That’s why you don’t see it in synaptic nor dpkg commands.

Step-2: Download the Driver (With NVIDIA Driver)

https://www.nvidia.com/content/DriverDownload-March2009/confirmation.php?url=/tesla/410.104/NVIDIA-Linux-x86_64-410.104.run&lang=us&type=Tesla

In the website, you need to choose the required driver for the installed GPUs in your machine by the below page as shown in Figure.66.



The screenshot shows the 'NVIDIA Driver Downloads' page. Under the heading 'Option 1: Manually find drivers for my NVIDIA products.', there are six dropdown menus for selection: Product Type (Tesla), Product Series (K-Series), Product (Tesla K40c), Operating System (Linux 64-bit), CUDA Toolkit (10.0), and Language (English (US)). A green 'SEARCH' button is located to the right of the Language dropdown. A 'Help' link is visible in the top right corner.

Figure.66 NVIDIA Driver Repository

As I have 3 different GPUs, Tesla K40c is compatible with NVIDIA-Linux-x86_64-410.104. Quadro p5000 & Quadro K620 are compatible with NVIDIA-Linux-x86_64-418.56. For all 3 GPUs, I am taking the 410.104 as base driver version.

Step-3: Build Essential Dependencies

1. build-essentials – For building drivers

2. dkms – For providing dkms support, DKMS is for packages that provide a kernel module in source form (or binary with a source wrapper), so they don't have to update the module for every kernel rebuild.
3. gcc-multilib – For providing 32-bit support
4. xorg and xorg-dev – For graphic display on a workstation with GUI (If not installed)

Check with command: **\$ sudo X -version**

```

hpcmonster369@hpc369-Z10PE-D16-WS:~$ sudo X -version
[sudo] password for hpcmonster369:
X.Org X Server 1.20.1
X Protocol Version 11, Revision 0
Build Operating System: Linux 4.4.0-140-generic x86_64 Ubuntu
Current Operating System: Linux hpc369-Z10PE-D16-WS 4.18.0-16-generic #17-18.04.1-Ubuntu SMP Tue Feb 12 13:35:51 UTC 2019 x86_64
Kernel command line: BOOT_IMAGE=/boot/vmlinuz-4.18.0-16-generic root=UUID=57d45905-8e2a-4520-9077-7bf714fc2c10 ro recovery nomodeset
Build Date: 27 November 2018 05:27:12PM
xorg-server-hwe-18.04 2:1.20.1-3ubuntu2.1~18.04.1 (For technical support please see http://www.ubuntu.com/support)
Current version of pixman: 0.34.0
    Before reporting problems, check http://wiki.x.org
    to make sure that you have the latest version.
hpcmonster369@hpc369-Z10PE-D16-WS:~$

```

Figure.67 Graphics Display

Please run the command: **\$ sudo apt-get install build-essential gcc-multilib dkms**

Step-4: Disable default nouveau

Please note that nouveau drivers manual removal is required only if you are going to install the proprietary NVIDIA drivers. If not after NVIDIA driver installation, nouveau may cause blurry screens. As we have NVIDIA GPUs, we need to remove it before installing NVIDIA drivers.

1. Please create a file. Please follow the command below.

\$ sudo gedit /etc/modprobe.d/blacklist-nouveau.conf

2. Please add below contents in it

blacklist nouveau

blacklist lbm-nouveau

options nouveau modeset=0

alias nouveau off

alias lbm-nouveau off

Please verify the file with contents by below command

cat /etc/modprobe.d/blacklist-nouveau.conf

Step-5: Update the initramfs

It needs to update the initramfs which might be configured to load the nouveau drivers.

The update-initramfs script manages your initramfs images on your local box. It keeps track of the existing initramfs archives in /boot. There are three modes of operation create, update or delete. You must at least specify one of those modes.

Please run the command below.

\$ sudo update-initramfs -u

It will give confirmation with below line.

update-initramfs: Generating /boot/initrd.img-4.18.0-15-generic

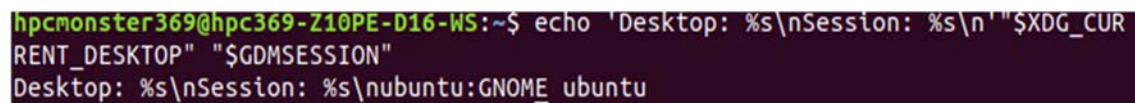
Please reboot the machine to proceed further.

Step-6: Stop Desktop Manager

After computer is rebooted, we need to stop the desktop manager before executing the runfile to install the driver. lightdm is the default manager in Ubuntu. If GNOME or KDE desktop environment is used, then desktop manager would be gdm or kdm.

To find the running session in your machine please use below command.

```
$ echo 'Desktop: %s\nSession: %s\n'"$XDG_CURRENT_DESKTOP"  
"$GDMSESSION"
```



```
hpcmonster369@hpc369-Z10PE-D16-WS:~$ echo 'Desktop: %s\nSession: %s\n'"$XDG_CUR  
RENT_DESKTOP" "$GDMSESSION"  
Desktop: %s\nSession: %s\nubuntu:GNOME ubuntu
```

Figure.68 GDM Session

Please run the command to stop gdm service.

```
$ sudo service gdm stop
```

In order to install new NVIDIA driver we need to stop the current display server. The easiest way to do this is to change into runlevel 3 using telinit command. After this command, the display server will stop, therefore make sure to save all current work before proceed.

Please run the command below.

```
$ sudo telinit 3
```

Step-6: Install the driver

```
cd $HOME
```

```
sudo chmod +x NVIDIA-Linux-x86_64-410.104.run
```

```
sudo ./NVIDIA-Linux-x86_64-410.104.run --dkms -s
```

Step-7: Check Installation by using below command.

\$ nvidia-smi

As shown in Figure.68, after successful installation, it will report all CUDA capable devices in your system.

```
hpcmonster369@hpc369-Z10PE-D16-WS:~$ nvidia-smi
Wed Mar 27 18:20:40 2019

+-----+
| NVIDIA-SMI 410.104      Driver Version: 410.104      CUDA Version: 10.0      |
+-----+-----+
| GPU  Name                Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf    Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
+-----+-----+
| 0   Tesla K40c           Off         | 00000000:02:00.0 Off  |   0B / 11441MiB | 0%      Default  |
| 23%   33C    P8      21W / 235W | 0MiB / 11441MiB |           |             |
+-----+-----+
| 1   Quadro P5000         Off         | 00000000:03:00.0 Off  |   2MiB / 16278MiB | 0%      Default  |
| 26%   33C    P8       6W / 180W | 2MiB / 16278MiB |           |             |
+-----+-----+
| 2   Quadro K620          Off         | 00000000:81:00.0 On   | 320MiB / 1999MiB | 0%      Default  |
| 34%   34C    P8       1W /  30W | 320MiB / 1999MiB |           |             |
+-----+-----+

+-----+
| Processes:                                     GPU Memory |
|  GPU       PID    Type   Process name                               Usage      |
+-----+-----+
|    2       3130    G     /usr/lib/xorg/Xorg                          23MiB     |
|    2       3292    G     /usr/bin/gnome-shell                       53MiB     |
|    2       3515    G     /usr/lib/xorg/Xorg                          156MiB    |
|    2       3656    G     /usr/bin/gnome-shell                       81MiB     |
+-----+-----+
```

Figure.69 NVIDIA Driver Successful Installation Snapshot

Step -2: (Alternative of above with PPA)

1. Add the Official NVIDIA PPA to Ubuntu and update it.

\$ sudo add-apt-repository ppa:graphics-drivers/ppa

\$ sudo apt update

2. Please check with below command which driver is required to install.

\$ ubuntu-drivers devices

```
hpcmonster369@hpc369-Z10PE-D16-WS:~$ ubuntu-drivers devices
== /sys/devices/pci0000:00/0000:00:02.0/0000:02:00.0 ==
modalias : pci:v000010DEd00001024sv000010DEsd00000983bc03sc02i00
vendor   : NVIDIA Corporation
model    : GK110BGL [Tesla K40c]
manual_install: True
driver   : nvidia-driver-410 - third-party free
driver   : nvidia-driver-418 - third-party free recommended
driver   : nvidia-340 - distro non-free
driver   : nvidia-driver-390 - distro non-free
driver   : nvidia-driver-415 - third-party free
driver   : nvidia-driver-396 - third-party free
driver   : xserver-xorg-video-nouveau - distro free builtin
```

Figure.70 Ubuntu Driver Display

In Figure. 70, it clearly recommends nvidia-driver-418, but for hassle free environment, we have installed 410.

3. Install the recommended NVIDIA Driver.

\$ sudo apt install nvidia-driver-410

Step-3: Install CUDA Toolkit

Pre-Installation Actions

1. Please verify whether you have a CUDA capable GPU.

\$ lspci | grep -i nvidia

2. Please verify whether you have a supported version of Linux.

\$ uname -m && cat /etc/*release

3. Please verify the system has gcc installed.

```
$ gcc --version
```

4. Please verify if the system has correct kernel header installed

```
$ uname -r
```

5. Please run the command to install updated kernel header.

```
$ sudo apt-get install linux-headers-$(uname -r)
```

6. Please select below link to download the CUDA as in Figure.71.

<https://developer.nvidia.com/cuda-downloads>

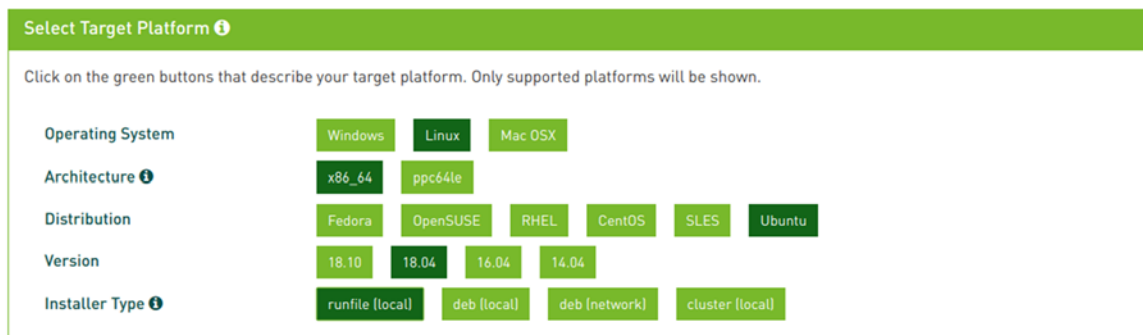


Figure. 71 CUDA Toolkit

7. Install repository meta-data

```
$ sudo dpkg -i cuda-repo-<distro>_<version>_<architecture>.deb
```

8. Installing the CUDA public GPG key (Installing the local repo)

```
$ sudo apt-key add /var/cuda-repo-<version>/7fa2af80.pub
```

9. Update the Apt repository cache

\$ sudo apt-get update

10. Install CUDA

\$ sudo apt-get install cuda

11. Set Environment path (Post Installation)

1. Take backup of existing bashrc file.
2. Go to the home directory.

cd \$HOME

3. Open the .bashrc file

sudo gedit .bashrc

4. Add following two commands in .bashrc file.

```
export PATH=/usr/local/cuda-10.0/bin${PATH:+:${PATH}}
```

```
export LD_LIBRARY_PATH=/usr/local/cuda-10.0/lib64${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

5. Save and close the .bashrc file .
6. Restart the machine.

Verification Action already mentioned in the Implementation section.

Step-4: Install cuDNN

1. Go to the cuDNN download page (need registration) and select the latest cuDNN 7.5 version made for CUDA 10.0.

Please use the link below.

<https://developer.nvidia.com/rdp/cudnn-download>

2. Download all 3 .deb files: the runtime library, the developer library, and the code samples library for Ubuntu 18.04.

3. Install them in the same order:

```
sudo dpkg -i libcudnn7_7.5.0.56-1+cuda10.0_amd64.deb (the runtime library)
```

```
sudo dpkg -i libcudnn7-dev_7.5.0.56-1+cuda10.0_amd64.deb (the developer library)
```

```
sudo dpkg -i libcudnn7-doc_7.5.0.56-1+cuda10.0_amd64.deb (the code samples)
```

4. The verification process is mentioned in the Implementation section.

Step-5: Install lipcupty-dev

1. Please use the below command.

```
sudo apt-get install lipcupty-dev
```

2. Please add the below line in the bashrc file for environment setup. Use below command. (Please take a backup of bashrc file)

```
echo 'export
```

```
LD_LIBRARY_PATH=/usr/local/cuda/extras/CUPTI/lib64:$LD_LIBRARY_PATH'
```

```
>> ~/.bashrc
```

Please follow the given link for more details.

<https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html>

A.2 PROGRAM MACHINE SETUP

The HAR program contains multiple files. The 3 important files are

1. lstm_architecture.py
2. Config_Dataset_HAR.py
3. Config_Dataset_HAR.ipynb (Jupyter Notebook)
4. download_datasets.py

The “data” directory needs to be created manually with 775 access inside the environment which is mentioned in the Config_Dataset_HAR as path for training and testing data samples of HAR. The folder structure mentioned in the Figure. 8 will automatically established by the Config file once it gets the data folder. The download_dataset.py will load the UCI repository file for the first time from website and put in the data directory. This file is places inside the data folder.

The first section of Config_Dataset_HAR.ipynb file will set the path of data, call the download_datasets.py script to load the data and will create necessary directory structure for the program. It needs to run only once for the whole program.

The second section of Config_Dataset_HAR.ipynb file on running creates X_train_signals_paths and X_test_signal_paths with proper folder structure.

The load_X and load_y methods takes the input parameters of signal_paths and returns ndarray which is tensor of features of both training and testing. Basically it prepares the datasets for training and testing by the deep learning model.

The file lstm_architecture.py contains all the different types of LSTM functions which

are fed by the dataset from the `Config_Dataset_HAR.py` file with a window of 128 timesteps. The input of HAR should be a time series, and the basic structure of the LSTM guarantees that it can preserve the characteristics on the temporal dimension.

The below input parameters used for different features.

`self.training_epochs` is the number of iterations the model will run.

`self.learning_rate` is the parameter which decides what would be the learning rate of the model.

`self.n_hidden` is the parameter which decides how many hidden layers will be developed by the model for experiment.

`self.use_bidirectionnal_cells` is the parameter which decided cells will do bidirectional communication or not.

`n_layers_in_highway` parameter decides how many residual layers would be there in the model.

`n_stacked_layers` parameter decides how many deep-stacked layers would be there in the model.

OneHotEncoder

A one hot encoding is a representation of categorical variables as binary vectors. This first requires that the categorical values be mapped to integer values. Then, each integer value is represented as a binary vector that is all zero values except the index of the integer, which is marked with a 1. We have not used any API for this, we have used the manual process. The function `one_hot(y)` converts labels from dense to one hot layer. For example it takes `[[5], [0], [3]]` as input array and returns `[[0, 0, 0, 0, 0, 1], [1, 0, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0]]` as output.

L2 Regularization

We have used L^2 regularization in the context of Stochastic Gradient Descent in Neural Network.

$$J_{regularized} = \underbrace{-\frac{1}{m} \sum_{i=1}^m (y^{(i)} \log(a^{[L](i)}) + (1 - y^{(i)}) \log(1 - a^{[L](i)}))}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m} \frac{\lambda}{2} \sum_l \sum_k \sum_j W_{kj}^{(l)2}}_{\text{L2 regularization cost}}$$

Figure.72 An L2-regularized version of the cost function used in SGD of RNN

Generally in Machine Learning, when we fit our model we search the solution space for the most fitting solution; In the context of Neural Networks, the solution space can be thought of as the space of all functions our network can represent. We know that the size of this space depends on the depth of the network and the activation functions used. We also know that with at least one hidden layer followed by an activation layer using a “squashing” function, this space is very large, and that it grows exponentially with the depth of the network (e.g the universal approximation theorem).

When we are using Stochastic Gradient Descent (SGD) to fit our network’s parameters to the learning problem at hand, we take, at each iteration of the algorithm, a step in the solution space towards the gradient of the loss function $J(\theta; X, y)$ in respect to the network’s parameters θ . Since the solution space of deep neural networks is very rich, this method of learning might overfit to our training data. This overfitting may result in significant generalization error and bad performance on test data, in the context of model development, if no counter-measure is used. Those counter-measures are called regularization techniques.

Additionally, a large network can be optimized correctly for a problem with sufficient

regularization, such as L2 weight decay and dropout. However, if no regularization is used, results trend to overfitting and bad operations on the test set. Complexity is good but only if countered with regularization. Too many layers and cells per layer will increase the computational complexity and waste computational resources. When the layer number and cell number reach a certain scale, the recognition accuracy will remain at a certain scale instead of increasing. By adding more depth, regularization is then needed to avoid overfitting while still improving accuracy. The L2 norm of the weights for weight decay is added in the loss function in our deep learning model.

Our deep LSTM neural network is limited in terms of how many data points it can access: it has access to only 128 time steps when making its predictions. Especially when deepened, the next forward/backward duo will see output from the other pass “in advance”, because, logically, a backward pass for our bidirectional LSTM reverses the input and the output before the concatenation. Thus, the Bidir-LSTM has the same input and output shape as the baseline LSTM. But at a given time step, it has access to more information in advance because of the backward passes.

Activation Function

In our network, the activity function is unified with ReLU, because it always outperforms with deep networks to counter gradient vanishing. Using it's recommended to use RELU/leaky RELU as the activation function, as it is relatively robust to the vanishing/exploding gradient issue (especially for networks that are not too deep). Although the output is a tensor for a given time window, the time axis has been crunched by the neural network. That is, we need only the last element of the output and can discard the others. Thus, only the gradient from the prediction at the last time step is applied. This also causes a LSTM cell to be unnecessary: the uppermost backward LSTM in the bidirectional pass. Hopefully, this is not of great concern because TensorFlow should

evaluate what to compute and what not to compute. Additionally, the training dataset should be shuffled during the training process. The state of the neural network is reset at each new window for each new prediction. In our experiment, 3 x 3 residual bidirectional LSTM out-performing other LSTM models with 2 x 2 and 4 x 4 architecture. The 3 x 3 could be thought of 18 LSTM cells working in a network.

Adam Optimizer

Adam is an adaptive learning rate optimization algorithm that's been designed specifically for training deep neural networks. First published in 2014, Adam was presented at ICLR 2015 conference for deep learning practitioners. Adam is an adaptive learning rate method, which means, it computes individual learning rates for different parameters. Its name is derived from adaptive moment estimation, and the reason it's called that is because Adam uses estimations of first and second moments of gradient to adapt the learning rate for each weight of the neural network.

Dropout

self.keep_prob_for_dropout is the parameter which specifies the dropout in the model. dropout is applied between each layer on the depth axis or, sometimes, just at the output, depending on what is specified in the configuration file, which is another hyper-parameter. Dropout refers to the fact that parts of tensors that are output by the hidden layer are shut down to a zero value to a certain probability for each value in each training epoch, while other values scale up accordingly to keep the same geometric norm of the tensor's values. The inoperative nodes can be regarded as dead nodes (or neurons) that are temporarily not in the network, which means that the weights and biases behind these dead notes temporarily neither learns nor contributes to the predictions during that training step for a batch. The weights are kept intact.

APPENDIX B – SOURCE CODE

B.1 TensorFlow Code

download_dataset.py

```
# !wget "https://archive.ics.uci.edu/ml/machine-learning-databases/00240/UCI HAR
Dataset.zip"
# !wget "https://archive.ics.uci.edu/ml/machine-learning-databases/00240/UCI HAR
Dataset.names"

# import copy
import os
from subprocess import call

print("")

print("Downloading UCI HAR Dataset...")
if not os.path.exists("UCI HAR Dataset.zip"):
    call(
        'wget "https://archive.ics.uci.edu/ml/machine-learning-databases/00240/UCI HAR
Dataset.zip"',
        shell=True
    )
    print("Downloading done.\n")
else:
```



```

print("Dataset already downloaded. Did not download twice.\n")

print("Extracting...")
extract_directory = os.path.abspath("UCI HAR Dataset")
if not os.path.exists(extract_directory):
    call(
        'unzip -nq "UCI HAR Dataset.zip"',
        shell=True
    )
    print("Extracting successfully done to {}".format(extract_directory))
else:
    print("Dataset already extracted. Did not extract twice.\n")

```

lstm_architecture.py

```

__author__ = 'jk_ranbir'

import tensorflow as tf
from sklearn import metrics
from sklearn.utils import shuffle
import numpy as np
from datetime import datetime
import time

def one_hot(y):
    """convert label from dense to one hot
    argument:

```

```

    label: ndarray dense label ,shape: [sample_num,1]
return:
    one_hot_label: ndarray one hot, shape: [sample_num,n_class]
"""
# e.g.: [[5], [0], [3]] --> [[0, 0, 0, 0, 0, 1], [1, 0, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0]]

y = y.reshape(len(y))
n_values = np.max(y) + 1
return np.eye(n_values)[np.array(y, dtype=np.int32)] # Returns FLOATS

```

```

def batch_norm(input_tensor, config, i):
    # Implementing batch normalisation: this is used out of the residual layers
    # to normalise those output neurons by mean and standard deviation.

    if config.n_layers_in_highway == 0:
        # There is no residual layers, no need for batch_norm:
        return input_tensor

    with tf.variable_scope("batch_norm") as scope:
        if i != 0:
            # Do not create extra variables for each time step
            scope.reuse_variables()

        # Mean and variance normalisation simply crunched over all axes
        axes = list(range(len(input_tensor.get_shape())))

        mean, variance = tf.nn.moments(input_tensor, axes=axes, shift=None, name=None,

```

```

keep_dims=False)
    stdev = tf.sqrt(variance+0.001)

    # Rescaling
    bn = input_tensor - mean
    bn /= stdev
    # Learnable extra rescaling

    # tf.get_variable("relu_fc_weights", initializer=tf.random_normal(mean=0.0,
stddev=0.0)
    bn *= tf.get_variable("a_noreg", initializer=tf.random_normal([1], mean=0.5,
stddev=0.0))
    bn += tf.get_variable("b_noreg", initializer=tf.random_normal([1], mean=0.0,
stddev=0.0))
    # bn *= tf.Variable(0.5, name=(scope.name + "/a_noreg"))
    # bn += tf.Variable(0.0, name=(scope.name + "/b_noreg"))

return bn

```

```

def relu_fc(input_2D_tensor_list, features_len, new_features_len, config):
    """make a relu fully-connected layer, mainly change the shape of tensor
both input and output is a list of tensor
argument:
    input_2D_tensor_list: list shape is [batch_size,feature_num]
    features_len: int the initial features length of input_2D_tensor
    new_feature_len: int the final features length of output_2D_tensor
    config: Config used for weights initializers
return:

```

```

        output_2D_tensor_list list shape is [batch_size,new_feature_len]
"""

W = tf.get_variable(
    "relu_fc_weights",
    initializer=tf.random_normal(
        [features_len, new_features_len],
        mean=0.0,
        stddev=float(config.weights_stddev)
    )
)
b = tf.get_variable(
    "relu_fc_biases_noreg",
    initializer=tf.random_normal(
        [new_features_len],
        mean=float(config.bias_mean),
        stddev=float(config.weights_stddev)
    )
)

# intra-timestep multiplication:
output_2D_tensor_list = [
    tf.nn.relu(tf.matmul(input_2D_tensor, W) + b)
    for input_2D_tensor in input_2D_tensor_list
]

return output_2D_tensor_list

```

```

def single_LSTM_cell(input_hidden_tensor, n_outputs):
    """ define the basic LSTM layer
    argument:
        input_hidden_tensor: list a list of tensor,
                               shape: time_steps*[batch_size,n_inputs]
        n_outputs: int num of LSTM layer output
    return:
        outputs: list a time_steps list of tensor,
                 shape: time_steps*[batch_size,n_outputs]
    """
    with tf.variable_scope("lstm_cell"):
        lstm_cell = tf.nn.rnn_cell.LSTMCell(n_outputs, state_is_tuple=True,
forget_bias=0.999)
        outputs, _ = tf.nn.static_rnn(lstm_cell, input_hidden_tensor, dtype=tf.float32)
    return outputs

def bi_LSTM_cell(input_hidden_tensor, n_inputs, n_outputs, config):
    """build bi-LSTM, concatenating the two directions in an inner manner.
    argument:
        input_hidden_tensor: list a time_steps series of tensor, shape: [sample_num,
n_inputs]
        n_inputs: int units of input tensor
        n_outputs: int units of output tensor, each bi-LSTM will have half those internal
units
        config: Config used for the relu_fc
    return:

```

```

        layer_hidden_outputs: list a time_steps series of tensor, shape: [sample_num,
n_outputs]
        """
        n_outputs = int(n_outputs/2)

        print ("bidir:")

        with tf.variable_scope('pass_forward') as scope2:
            hidden_forward = relu_fc(input_hidden_tensor, n_inputs, n_outputs, config)
            forward = single_LSTM_cell(hidden_forward, n_outputs)

        print (len(hidden_forward), str(hidden_forward[0].get_shape()))

        # Backward pass is as simple as surrounding the cell with a double inversion:
        with tf.variable_scope('pass_backward') as scope2:
            hidden_backward = relu_fc(input_hidden_tensor, n_inputs, n_outputs, config)
            backward = list(reversed(single_LSTM_cell(list(reversed(hidden_backward)),
n_outputs)))

        with tf.variable_scope('bidir_concat') as scope:
            # Simply concatenating cells' outputs at each timesteps on the innermost
            # dimension, like if the two cells acted as one cell
            # with twice the n_hidden size:
            layer_hidden_outputs = [
                tf.concat([f, b], len(f.get_shape()) - 1)
                for f, b in zip(forward, backward)]

        return layer_hidden_outputs

```

```

def residual_bidirectional_LSTM_layers(input_hidden_tensor, n_input, n_output,
layer_level, config, keep_prob_for_dropout):
    """This architecture is only enabled if "config.n_layers_in_highway" has a
    value only greater than int(0). The arguments are same than for bi_LSTM_cell.
    arguments:
        input_hidden_tensor: list a time_steps series of tensor, shape: [sample_num,
n_inputs]
        n_inputs: int units of input tensor
        n_outputs: int units of output tensor, each bi-LSTM will have half those internal
units
        config: Config used for determining if there are residual connections and if yes, their
number and with some batch_norm.
    return:
        layer_hidden_outputs: list a time_steps series of tensor, shape: [sample_num,
n_outputs]
    """
    with tf.variable_scope('layer_{}'.format(layer_level)) as scope:

        if config.use_bidirectionnal_cells:
            get_lstm = lambda input_tensor: bi_LSTM_cell(input_tensor, n_input, n_output,
config)
        else:
            get_lstm = lambda input_tensor: single_LSTM_cell(relu_fc(input_tensor,
n_input, n_output, config), n_output)

        def add_highway_redisual(layer, residual_minilayer):
            return [a + b for a, b in zip(layer, residual_minilayer)]

```

```

hidden_LSTM_layer = get_lstm(input_hidden_tensor)
# Adding K new (residual bidir) connections to this first layer:
for i in range(config.n_layers_in_highway - 1):
    with tf.variable_scope('LSTM_residual_{}'.format(i)) as scope2:
        hidden_LSTM_layer = add_highway_redisual(
            hidden_LSTM_layer,
            get_lstm(input_hidden_tensor)
        )

    if config.also_add_dropout_between_stacked_cells:
        hidden_LSTM_layer = [tf.nn.dropout(out, keep_prob_for_dropout) for out in
hidden_LSTM_layer]

    return [batch_norm(out, config, i) for i, out in enumerate(hidden_LSTM_layer)]

def LSTM_network(feature_mat, config, keep_prob_for_dropout):
    """model a LSTM Network,
    it stacks 2 LSTM layers, each layer has n_hidden=32 cells
    and 1 output layer, it is a full connet layer
    argument:
        feature_mat: ndarray fature matrix, shape=[batch_size,time_steps,n_inputs]
        config: class containing config of network
    return:
        : ndarray output shape [batch_size, n_classes]
    """

```



```

with tf.variable_scope('LSTM_network') as scope: # TensorFlow graph naming

    feature_mat = tf.nn.dropout(feature_mat, keep_prob_for_dropout)

    # Exchange dim 1 and dim 0
    feature_mat = tf.transpose(feature_mat, [1, 0, 2])
    print (feature_mat.get_shape())
    # New feature_mat's shape: [time_steps, batch_size, n_inputs]

    # Temporarily crush the feature_mat's dimensions
    feature_mat = tf.reshape(feature_mat, [-1, config.n_inputs])
    print (feature_mat.get_shape())
    # New feature_mat's shape: [time_steps*batch_size, n_inputs]

    # Split the series because the rnn cell needs time_steps features, each of shape:
    hidden = tf.split(feature_mat, config.n_steps, 0)
    print (len(hidden), str(hidden[0].get_shape()))
    # New shape: a list of length "time_step" containing tensors of shape [batch_size,
n_hidden]

    # Stacking LSTM cells, at least one is stacked:
    print ("\nCreating hidden #1:")
    hidden = residual_bidirectional_LSTM_layers(hidden, config.n_inputs,
config.n_hidden, 1, config, keep_prob_for_dropout)
    print (len(hidden), str(hidden[0].get_shape()))

    for stacked_hidden_index in range(config.n_stacked_layers - 1):
        # If the config permits it, we stack more lstm cells:

```

```

        print ("\nCreating hidden #{}:".format(stacked_hidden_index+2))
        hidden = residual_bidirectional_LSTM_layers(hidden, config.n_hidden,
config.n_hidden, stacked_hidden_index+2, config, keep_prob_for_dropout)
        print (len(hidden), str(hidden[0].get_shape()))

    print ("")

    # Final fully-connected activation logits
    # Get the last output tensor of the inner loop output series, of shape [batch_size,
n_classes]
    last_hidden = tf.nn.dropout(hidden[-1], keep_prob_for_dropout)
    last_logits = relu_fc(
        [last_hidden],
        config.n_hidden, config.n_classes, config
    )[0]
    return last_logits

def run_with_config(Config, X_train, y_train, X_test, y_test):
    start_time = datetime.now()
    print ("Start Time: ",time.ctime())
    print ("")
    tf.reset_default_graph() # To enable to run multiple things in a loop

    #-----
    # Define parameters for model
    #-----
    config = Config(X_train, X_test)

```

```

print("Some useful info to get an insight on dataset's shape and normalisation:")
print("features shape, labels shape, each features mean, each features standard
deviation")

print(X_test.shape, y_test.shape,
      np.mean(X_test), np.std(X_test))
print("the dataset is therefore properly normalised, as expected.")

#-----
# Let's get serious and build the neural network
#-----

with tf.device("/cpu:0"): # Remove this line to use GPU. If you have a too small GPU,
it crashes.

#with tf.device('/gpu:0'):
#with tf.device('/gpu:1'):
#mirrored_strategy = tf.contrib.distribute.MirroredStrategy(devices=["/gpu:0",
"/gpu:1"])
#with mirrored_strategy.scope():
    X = tf.placeholder(tf.float32, [
        None, config.n_steps, config.n_inputs], name="X")
    Y = tf.placeholder(tf.float32, [
        None, config.n_classes], name="Y")

# is_train for dropout control:
is_train = tf.placeholder(tf.bool, name="is_train")
keep_prob_for_dropout = tf.cond(is_train,
    lambda: tf.constant(
        config.keep_prob_for_dropout,
        name="keep_prob_for_dropout"

```

```

    ),
    lambda: tf.constant(
        1.0,
        name="keep_prob_for_dropout"
    )
)

pred_y = LSTM_network(X, config, keep_prob_for_dropout)

# Loss, optimizer, evaluation

# Softmax loss with L2 and L1 layer-wise regularisation
print ("Unregularised variables:")
for unreg in [tf_var.name for tf_var in tf.trainable_variables() if ("noreg" in
tf_var.name or "Bias" in tf_var.name)]:
    print (unreg)
l2 = config.lambda_loss_amount * sum(
    tf.nn.l2_loss(tf_var)
    for tf_var in tf.trainable_variables()
    if not ("noreg" in tf_var.name or "Bias" in tf_var.name)
)
# first_weights = [w for w in tf.all_variables() if w.name ==
'LSTM_network/layer_1/pass_forward/relu_fc_weights:0'][0]
# l1 = config.lambda_loss_amount * tf.reduce_mean(tf.abs(first_weights))
loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits_v2(logits=pred_y,labels=Y)) + l2 # +

```

11

```

# Gradient clipping Adam optimizer with gradient noise
optimize = tf.contrib.layers.optimize_loss(
    loss,
    global_step=tf.Variable(0),
    learning_rate=config.learning_rate,
    optimizer=tf.train.AdamOptimizer(learning_rate=config.learning_rate),
    clip_gradients=config.clip_gradients,
    gradient_noise_scale=config.gradient_noise_scale
)

correct_pred = tf.equal(tf.argmax(pred_y, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, dtype=tf.float32))

#-----
# Hooray, now train the neural network
#-----
# Note that log_device_placement can be turned of for less console spam.

#sessconfig = tf.ConfigProto(log_device_placement=False)
sessconfig = tf.ConfigProto(allow_soft_placement =
True,log_device_placement=False)
#sessconfig.gpu_options.allow_growth = True
with tf.Session(config=sessconfig) as sess:
    #init = tf.global_variables_initializer()
    sess.run(tf.global_variables_initializer())

best_accuracy = (0.0, "iter: -1")
best_f1_score = (0.0, "iter: -1")

```

```

# Start training for each batch and loop epochs

worst_batches = []

for i in range(config.training_epochs):

    # Loop batches for an epoch:
    shuffled_X, shuffled_y = shuffle(X_train, y_train, random_state=i*42)
    for start, end in zip(range(0, config.train_count, config.batch_size),
                          range(config.batch_size, config.train_count + 1, config.batch_size)):

        _, train_acc, train_loss, train_pred = sess.run(
            [optimize, accuracy, loss, pred_y],
            feed_dict={
                X: shuffled_X[start:end],
                Y: shuffled_y[start:end],
                is_train: True
            }
        )

        worst_batches.append(
            (train_loss, shuffled_X[start:end], shuffled_y[start:end])
        )

        worst_batches = list(sorted(worst_batches))[-5:] # Keep 5 poorest

# Train F1 score is not on boosting
train_f1_score = metrics.f1_score(

```

```
    shuffled_y[start:end].argmax(1), train_pred.argmax(1), average="weighted"  
)
```

```
# Retrain on top worst batches of this epoch (boosting):
```

```
# a.k.a. "focus on the hardest exercises while training":
```

```
for _, x_, y_ in worst_batches:
```

```
    _, train_acc, train_loss, train_pred = sess.run(  
        [optimize, accuracy, loss, pred_y],  
        feed_dict={  
            X: x_,  
            Y: y_,  
            is_train: True  
        }  
    )
```

```
# Test completely at the end of every epoch:
```

```
# Calculate accuracy and F1 score
```

```
pred_out, accuracy_out, loss_out = sess.run(  
    [pred_y, accuracy, loss],  
    feed_dict={  
        X: X_test,  
        Y: y_test,  
        is_train: False  
    }  
)
```

```
# "y_test.argmax(1)": could be optimised by being computed once...
```

```

f1_score_out = metrics.f1_score(
    y_test.argmax(1), pred_out.argmax(1), average="weighted"
)

print (
    "iter: {}, ".format(i) + \
    "train loss: {}, ".format(train_loss) + \
    "train accuracy: {}, ".format(train_acc) + \
    "train F1-score: {}, ".format(train_f1_score) + \
    "test loss: {}, ".format(loss_out) + \
    "prediction accuracy: {}, ".format(accuracy_out) + \
    "test F1-score: {}".format(f1_score_out)
)

best_accuracy = max(best_accuracy, (accuracy_out, "iter: {}".format(i)))
best_f1_score = max(best_f1_score, (f1_score_out, "iter: {}".format(i)))

print("")
print("final prediction accuracy: {}".format(accuracy_out))
print("best epoch's prediction accuracy: {}".format(best_accuracy))
print("final F1 score: {}".format(f1_score_out))
print("best epoch's F1 score: {}".format(best_f1_score))
print("")
end_time = datetime.now()
print("End Time: ",time.ctime())
print("Exec Duration: {}".format(end_time - start_time))
print("")

```



```
# returning both final and bests accuracies and f1 scores.  
return accuracy_out, best_accuracy, f1_score_out, best_f1_score
```

Config Dataset HAR.py

```
#!/usr/bin/env python  
# coding: utf-8  
  
# In[ ]:  
__author__ = 'jkranbir'  
  
# Note: Linux bash commands start with a "!" inside those "ipython notebook" cells  
import os  
DATA_PATH = "data/"  
get_ipython().system('pwd && ls')  
os.chdir(DATA_PATH)  
get_ipython().system('pwd && ls')  
get_ipython().system('python download_datasets.py')  
get_ipython().system('pwd && ls')  
os.chdir("..")  
get_ipython().system('pwd && ls')  
DATASET_PATH = DATA_PATH + "UCI HAR Dataset/"  
print("\n" + "Dataset is now located at: " + DATASET_PATH)  
  
# In[ ]:  
__author__ = 'jkranbir'
```

```

from lstm_architecture import one_hot, run_with_config
import numpy as np
import os
#os.environ["CUDA_VISIBLE_DEVICES"]="0,1"

#-----
# Neural net's config.
#-----

class Config(object):
    """
    define a class to store parameters,
    the input should be feature mat of training and testing
    """

    def __init__(self, X_train, X_test):
        # Data shaping
        self.train_count = len(X_train) # 7352 training series
        self.test_data_count = len(X_test) # 2947 testing series
        self.n_steps = len(X_train[0]) # 128 time_steps per series
        self.n_classes = 6 # Final output classes

        # Training
        self.learning_rate = 0.001
        self.lambda_loss_amount = 0.005
        self.training_epochs = 250 #5
        self.batch_size = 100
        self.clip_gradients = 15.0

```

```

self.gradient_noise_scale = None
# Dropout is added on inputs and after each stacked layers (but not
# between residual layers).
self.keep_prob_for_dropout = 0.85 # **(1/3.0)

# Linear+relu structure
self.bias_mean = 0.3
# I would recommend between 0.1 and 1.0 or to change and use a xavier
# initializer
self.weights_stddev = 0.2

#####
# NOTE: I think that if any of the below parameters are changed,
# the best is to readjust every parameters in the "Training" section
# above to properly compare the architectures only once optimised.
#####

# LSTM structure
# Features count is of 9: three 3D sensors features over time
self.n_inputs = len(X_train[0][0])
self.n_hidden = 256 # nb of neurons inside the neural network
# Use bidir in every LSTM cell, or not:
self.use_bidirectionnal_cells = True #False

# High-level deep architecture
self.also_add_dropout_between_stacked_cells = False #True
# NOTE: values of exactly 1 (int) for those 2 high-level parameters below totally
disables them and result in only 1 starting LSTM.

```

```

    # self.n_layers_in_highway = 1 # Number of residual connections to the LSTMs
(highway-style), this is did for each stacked block (inside them).
    # self.n_stacked_layers = 1 # Stack multiple blocks of residual
    # layers.

#-----
# Dataset-specific constants and functions + loading
#-----

# Useful Constants

# Those are separate normalised input features for the neural network
INPUT_SIGNAL_TYPES = [
    "body_acc_x_",
    "body_acc_y_",
    "body_acc_z_",
    "body_gyro_x_",
    "body_gyro_y_",
    "body_gyro_z_",
    "total_acc_x_",
    "total_acc_y_",
    "total_acc_z_"
]

# Output classes to learn how to classify
LABELS = [
    "WALKING",
    "WALKING_UPSTAIRS",

```

```

"WALKING_DOWNSTAIRS",
"SITTING",
"STANDING",
"LAYING"
]

DATA_PATH = "data/"
DATASET_PATH = DATA_PATH + "UCI HAR Dataset/"

TRAIN = "train/"
TEST = "test/"

# Load "X" (the neural network's training and testing inputs)

def load_X(X_signals_paths):
    """
    Given attribute (train or test) of feature, read all 9 features into an
    np ndarray of shape [sample_sequence_idx, time_step, feature_num]
    argument: X_signals_paths str attribute of feature: 'train' or 'test'
    return: np ndarray, tensor of features
    """
    X_signals = []

    for signal_type_path in X_signals_paths:
        file = open(signal_type_path, 'r')
        # Read dataset from disk, dealing with text files' syntax
        X_signals.append(

```

```

        [np.array(serie, dtype=np.float32) for serie in [
            row.replace(' ', '').strip().split(' ') for row in file
        ]]
    )
    file.close()
    return np.transpose(np.array(X_signals), (1, 2, 0))
X_train_signals_paths = [
    DATASET_PATH + TRAIN + "Inertial Signals/" + signal + "train.txt" for signal in
INPUT_SIGNAL_TYPES
]
X_test_signals_paths = [
    DATASET_PATH + TEST + "Inertial Signals/" + signal + "test.txt" for signal in
INPUT_SIGNAL_TYPES
]
X_train = load_X(X_train_signals_paths)
X_test = load_X(X_test_signals_paths)
# Load "y" (the neural network's training and testing outputs)

def load_y(y_path):
    """
    Read Y file of values to be predicted
    argument: y_path str attribute of Y: 'train' or 'test'
    return: Y ndarray / tensor of the 6 one_hot labels of each sample
    """
    file = open(y_path, 'r')
    # Read dataset from disk, dealing with text file's syntax
    y_ = np.array(
        [elem for elem in [

```

```

        row.replace(' ', ' ').strip().split(' ') for row in file
    ]],
    dtype=np.int32
)
file.close()

# Subtract 1 to each output class for friendly 0-based indexing
return one_hot(y_ - 1)

y_train_path = DATASET_PATH + TRAIN + "y_train.txt"
y_test_path = DATASET_PATH + TEST + "y_test.txt"
y_train = load_y(y_train_path)
y_test = load_y(y_test_path)

#-----
# Training (maybe multiple) experiment(s)
#-----

n_layers_in_highway = 4
n_stacked_layers = 4
trial_name = "{}x{}".format(n_layers_in_highway, n_stacked_layers)

for learning_rate in [0.001]: # [0.01, 0.001, 0.0001]:
    for lambda_loss_amount in [0.005]:
        for clip_gradients in [15.0]:
            print ("learning_rate: {}".format(learning_rate))
            print ("lambda_loss_amount: {}".format(lambda_loss_amount))
            print ("")

```

```

class EditedConfig(Config):
    def __init__(self, X, Y):
        super(EditedConfig, self).__init__(X, Y)

        # Edit only some parameters:
        self.learning_rate = learning_rate
        self.lambda_loss_amount = lambda_loss_amount
        self.clip_gradients = clip_gradients
        # Architecture params:
        self.n_layers_in_highway = n_layers_in_highway
        self.n_stacked_layers = n_stacked_layers

    ## Useful catch upon looping (e.g.: not enough memory)
    # try:
    #   accuracy_out, best_accuracy = run_with_config(EditedConfig)
    # except:
    #   accuracy_out, best_accuracy = -1, -1
    accuracy_out, best_accuracy, fl_score_out, best_fl_score = (
        run_with_config(EditedConfig, X_train, y_train, X_test, y_test)
    )
    print (accuracy_out, best_accuracy, fl_score_out, best_fl_score)

    with open('{}_result_HAR_6.txt'.format(trial_name), 'a') as f:
        f.write(str(learning_rate) + '\t' + str(lambda_loss_amount) + '\t' +
str(clip_gradients) + '\t' + str(
            accuracy_out) + '\t' + str(best_accuracy) + '\t' + str(fl_score_out) + '\t' +
str(best_fl_score) + '\n\n')

```



```
print ("_____")
print ("")
print ("Done.")
```

```
# In[ ]:
```

B.2 PyTorch Code

Script.py

```
__author__ = 'jkranbir'
```

```
# Note: Linux bash commands start with a "!" inside those "ipython notebook" cells
```

```
import os
```

```
DATA_PATH = "data/"
```

```
!pwd && ls
```

```
os.chdir(DATA_PATH)
```

```
!pwd && ls
```

```
!python download_datasets.py
```

```
!pwd && ls
```

```
os.chdir("../")
```

```
!pwd && ls
```

```
DATASET_PATH = DATA_PATH + "UCI HAR Dataset/"
print("\n" + "Dataset is now located at: " + DATASET_PATH)
```

network 1.py

```
# encoding=utf-8
"""
    Created on 12:48 2019/03/10
    @author: Jagadish Kumar Ranbirsingh
"""

import torch.nn as nn
import torch.nn.functional as F

class Network(nn.Module):
    def __init__(self):
        super(Network, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels=9, out_channels=32, kernel_size=(1, 9)),
            # nn.BatchNorm1d()
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(1, 2), stride=2)
        )
        self.conv2 = nn.Sequential(
```

```

        nn.Conv2d(in_channels=32, out_channels=64, kernel_size=(1, 9)),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=(1, 2), stride=2)
    )
    self.fc1 = nn.Sequential(
        nn.Linear(in_features=64 * 26, out_features=1000),
        nn.ReLU()
    )
    self.fc2 = nn.Sequential(
        nn.Linear(in_features=1000, out_features=500),
        nn.ReLU()
    )
    self.fc3 = nn.Sequential(
        nn.Linear(in_features=500, out_features=6)
    )

def forward(self, x):
    out = self.conv1(x)
    out = self.conv2(out)
    out = out.reshape(-1, 64 * 26)
    out = self.fc1(out)
    out = self.fc2(out)
    out = self.fc3(out)
    out = F.softmax(out, dim=1)
    return out

```

network.py

```

# encoding=utf-8
"""
Created on 12:48 2019/03/10
@author: Jagadish Kumar Ranbirsingh
"""
import torch.nn as nn
import torch.nn.functional as F

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels=9, out_channels=32, kernel_size=(1, 9)),
            # nn.BatchNorm1d()
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(1, 2), stride=2)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=(1, 9)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(1, 2), stride=2)
        )
        self.conv2_drop = nn.Dropout2d()

        self.fc1 = nn.Sequential(
            nn.Linear(in_features=64 * 26, out_features=1000),
            nn.ReLU()

```

```

)
self.fc2 = nn.Sequential(
    nn.Linear(in_features=1000, out_features=500),
    nn.ReLU()
)
self.fc3 = nn.Sequential(
    nn.Linear(in_features=500, out_features=6)
)
def forward(self, x):
    out = self.conv1(x)
    out = self.conv2_drop(self.conv2(out))
    out = out.view(-1, 64 * 26)
    out = self.fc1(out)
    out = self.fc2(out)
    out = self.fc3(out)
    return out

```

```

class Network(nn.Module):
    def __init__(self):
        super(Network, self).__init__()
        self.cnn = CNN()
        self.rnn = nn.LSTM(64 * 26, 6, 2)

```

```

def forward(self, x):
    out = self.cnn(x)
    out = self.rnn(out)
    out = F.softmax(out, dim=1)
    return out

```

data_preprocess.py

```
# encoding=utf-8
"""
    Created on 07:51 2019/03/10
    @author: Jagadish Kumar Ranbirsingh
"""
import numpy as np
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms

# This is for parsing the X data, you can ignore it if you do not need preprocessing
def format_data_x(datafile):
    x_data = None
    for item in datafile:
        item_data = np.loadtxt(item, dtype=np.float)
        if x_data is None:
            x_data = np.zeros((len(item_data), 1))
        x_data = np.hstack((x_data, item_data))
    x_data = x_data[:, 1:]
    print(x_data.shape)
    X = None
    for i in range(len(x_data)):
        row = np.asarray(x_data[i, :])
        row = row.reshape(9, 128).T
        if X is None:
            X = np.zeros((len(x_data), 128, 9))
```

```

        X[i] = row
    print(X.shape)
    return X

# This is for parsing the Y data, you can ignore it if you do not need preprocessing
def format_data_y(datafile):
    data = np.loadtxt(datafile, dtype=np.int) - 1
    YY = np.eye(6)[data]
    return YY

# This for processing the dataset from scratch
# After script downloading the dataset, program put it in the DATA_PATH folder

def load_data():
    DATA_PATH = 'data/'
    DATASET_PATH = DATA_PATH + 'UCI HAR Dataset/'
    TRAIN = 'train/'
    TEST = 'test/'

    INPUT_SIGNAL_TYPES = [
        "body_acc_x_",
        "body_acc_y_",
        "body_acc_z_",
        "body_gyro_x_",
        "body_gyro_y_",
        "body_gyro_z_",
        "total_acc_x_",
        "total_acc_y_",

```

```

        "total_acc_z_"
    ]
    str_train_files = [DATASET_PATH + TRAIN + 'Inertial Signals/' + item + 'train.txt'
for item in
        INPUT_SIGNAL_TYPES]
    str_test_files = [DATASET_PATH + TEST + 'Inertial Signals/' + item + 'test.txt' for
item in INPUT_SIGNAL_TYPES]
    str_train_y = DATASET_PATH + TRAIN + 'y_train.txt'
    str_test_y = DATASET_PATH + TEST + 'y_test.txt'

    X_train = format_data_x(str_train_files)
    X_test = format_data_x(str_test_files)
    Y_train = format_data_y(str_train_y)
    Y_test = format_data_y(str_test_y)
    return X_train, onehot_to_label(Y_train), X_test, onehot_to_label(Y_test)
def onehot_to_label(y_onehot):
    a = np.argwhere(y_onehot == 1)
    return a[:, -1]

class data_loader(Dataset):
    def __init__(self, samples, labels, t):
        self.samples = samples
        self.labels = labels
        self.T = t

    def __getitem__(self, index):
        sample, target = self.samples[index], self.labels[index]
        return self.T(sample), target

```



```

def __len__(self):
    return len(self.samples)

def load(batch_size=100):
    x_train, y_train, x_test, y_test = load_data()
    x_train, x_test = x_train.reshape((-1, 9, 1, 128)), x_test.reshape((-1, 9, 1, 128))
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean=(0,0,0,0,0,0,0,0), std=(1,1,1,1,1,1,1,1))
    ])
    train_set = data_loader(x_train, y_train, transform)
    test_set = data_loader(x_test, y_test, transform)
    train_loader = DataLoader(train_set, batch_size=batch_size, num_workers=8,
pin_memory=True, shuffle=True, drop_last=True)
    test_loader = DataLoader(test_set, batch_size=batch_size, num_workers=8,
pin_memory=True, shuffle=False)
    return train_loader, test_loader

```

Config Dataset HAR.py

```

#!/usr/bin/env python
# coding: utf-8

# In[ ]:
__author__ = 'jkranbir'

# Note: Linux bash commands start with a "!" inside those "ipython notebook" cells
import os

```

```

DATA_PATH = "data/"
get_ipython().system('pwd && ls')
os.chdir(DATA_PATH)
get_ipython().system('pwd && ls')
get_ipython().system('python download_datasets.py')
get_ipython().system('pwd && ls')
os.chdir("..")
get_ipython().system('pwd && ls')

DATASET_PATH = DATA_PATH + "UCI HAR Dataset/"
print("\n" + "Dataset is now located at: " + DATASET_PATH)

# In[ ]:
# encoding=utf-8
"""
    Created on 09:41 2019/03/10
    @author: Jagadish Kumar Ranbirsingh
"""
import data_preprocess
import matplotlib.pyplot as plt
import network as net
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import tqdm
from datetime import datetime

```

```

import time

BATCH_SIZE = 100 #256
N_EPOCH = 10 * 250 #250 In dataset 7352 training series
LEARNING_RATE = 0.001
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Device :',DEVICE)
print('Device Count',torch.cuda.device_count())
print('Current Device:',torch.cuda.get_device_name(torch.cuda.current_device()))
result = [ ]

def train(model, optimizer, train_loader, test_loader):
    n_batch = len(train_loader.dataset) // BATCH_SIZE
    print('n_batch',n_batch)
    criterion = nn.CrossEntropyLoss()

    for e in range(N_EPOCH):
        model.train()
        correct, total_loss = 0, 0
        total = 0
        for index, (sample, target) in enumerate(train_loader):
            sample, target = sample.to(DEVICE).float(), target.to(DEVICE).long()
            sample = sample.view(-1, 9, 1, 128)
            output = model(sample)
            loss = criterion(output, target)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

```

```

total_loss += loss.item()
_, predicted = torch.max(output.data, 1)
total += target.size(0)
correct += (predicted == target).sum()

if index % 20 == 0:
    tqdm.tqdm.write('Epoch: [{} / {}], Batch: [{} / {}], loss: {:.4f}'.format(e + 1,
N_EPOCH, index + 1, n_batch, loss.item()))
    acc_train = float(correct) * 100.0 / (BATCH_SIZE * n_batch)
    tqdm.tqdm.write(
        'Epoch: [{} / {}], loss: {:.4f}, train acc: {:.2f}%'.format(e + 1, N_EPOCH,
total_loss * 1.0 / n_batch, acc_train))

```

```

# Testing
model.train(False)
with torch.no_grad():
    correct, total = 0, 0
    for sample, target in test_loader:
        sample, target = sample.to(DEVICE).float(), target.to(DEVICE).long()
        sample = sample.view(-1, 9, 1, 128)
        output = model(sample)
        _, predicted = torch.max(output.data, 1)
        total += target.size(0)
        correct += (predicted == target).sum()
    acc_test = float(correct) * 100 / total
    tqdm.tqdm.write('Epoch: [{} / {}], test acc: {:.2f}%'.format(e + 1, N_EPOCH,

```

```

float(correct) * 100 / total))
    result.append([acc_train, acc_test])
    result_np = np.array(result, dtype=float)
    np.savetxt('result.csv', result_np, fmt='%0.2f', delimiter=',')

def plot():
    data = np.loadtxt('result.csv', delimiter=',')
    plt.figure()
    plt.plot(range(1, len(data[:, 0]) + 1), data[:, 0], color='blue', label='train')
    plt.plot(range(1, len(data[:, 1]) + 1), data[:, 1], color='red', label='test')
    plt.legend()
    plt.xlabel('Epoch', fontsize=14)
    plt.ylabel('Accuracy (%)', fontsize=14)
    plt.title('Training and Prediction Accuracy', fontsize=20)

if __name__ == '__main__':
    torch.cuda.manual_seed_all(10)
    start_time = datetime.now()
    print ("Start Time: ",time.ctime())
    print ("")
    train_loader, test_loader = data_preprocess.load(batch_size=BATCH_SIZE)
    model = net.Network()
    model = model.to(DEVICE)
    #torch.distributed.init_process_group(backend="nccl")
    #model = torch.nn.parallel.DistributedDataParallel(model)
    optimizer = optim.SGD(params=model.parameters(), lr=LEARNING_RATE,
momentum=0.9)
    train(model, optimizer, train_loader, test_loader)

```

```

result = np.array(result, dtype=float)
np.savetxt('result.csv', result, fmt='%.2f', delimiter=',')
plot()
print("")
end_time = datetime.now()
print("End Time: ",time.ctime())
print("Exec Duration: {}".format(end_time - start_time))
print("")

# In[ ]:

```

data/download_datasets.py : It's same as TensorFlow.

B.3 Raspberry PI Cluster – Monte Carlo Simulation

server.py

```

import sys
import tensorflow as tf
import netifaces as ni

def getIpAddr():
    ni.ifaddresses("eth0")
    ip = ni.ifaddresses("eth0")[ni.AF_INET][0]["addr"]
    return ip

```

```
taskList =  
["192.168.1.16:1024","192.168.1.17:1024","192.168.1.18:1024","192.168.1.19:1024",  
  
"192.168.1.20:1024","192.168.1.21:1024","192.168.1.22:1024","192.168.1.23:1024",  
  
"192.168.1.24:1024","192.168.1.25:1024","192.168.1.26:1024","192.168.1.27:1024",  
  
"192.168.1.28:1024","192.168.1.29:1024","192.168.1.30:1024","192.168.1.31:1024"]
```

```
taskName = getIpAddr()+":1024"
```

```
try:
```

```
    taskNum = taskList.index(taskName)
```

```
except ValueError:
```

```
    print(" Unable to find " + taskName + " in the task List.")
```

```
    quit()
```

```
cluster = tf.train.ClusterSpec({"local":taskList})
```

```
server = tf.train.Server(cluster,job_name="local",task_index=taskNum)
```

```
server.join()
```

client.py

```
import tensorflow as tf
```

```
import numpy as np
```

```
import math
```

```
import time
```

```
start = time.time()
```

```
size = int(1*math.pow(10,6))
```

```

taskList =
["192.168.1.16:1024","192.168.1.17:1024","192.168.1.18:1024","192.168.1.19:1024",

"192.168.1.20:1024","192.168.1.21:1024","192.168.1.22:1024","192.168.1.23:1024",

"192.168.1.24:1024","192.168.1.25:1024","192.168.1.26:1024","192.168.1.27:1024",

"192.168.1.28:1024","192.168.1.29:1024","192.168.1.30:1024","192.168.1.31:1024"]

taskCount = len(taskList)
n = size//taskCount
r = size % taskCount
cluster = tf.train.ClusterSpec({"local":taskList})
total = tf.Variable(0,dtype=tf.float32)

for i in range(0,taskCount):
    if (i==0):
        sampleSize = n + r
    else:
        sampleSize = n

    deviceName = "/job:local/task:"+str(i)
    with tf.device(deviceName):
        pointList = tf.random_uniform(shape=[sampleSize,2],minval=-
1,maxval=1,dtype=tf.float32)
        distanceList = tf.sqrt(tf.reduce_sum(tf.pow(pointList,2),1))
        boolList = tf.less(distanceList,1)
        circleCount = tf.reduce_sum(tf.cast(boolList,tf.float32))

```



```

total = total + circleCount
print("task:",i," sampleSize: ",sampleSize)

with tf.Session("grpc://localhost:1024") as sess:
    sess.run(tf.global_variables_initializer())
    pi = sess.run(4*(total/size))
    print("pi:",pi)

end = time.time()
totalTime = end - start
print("Time: {:,3f}".format(totalTime))

```

B.4 Raspberry PI Cluster Code

[lstm_architecture.py](#)

```

__author__ = 'jk_ranbir'

import tensorflow as tf
from sklearn import metrics
from sklearn.utils import shuffle
import numpy as np
from datetime import datetime
import time
import sys
#import tensorflow as tf
import netifaces as ni

```

```

def getIpAddr():
    ni.ifaddresses("eth0")
    ip = ni.ifaddresses("eth0")[ni.AF_INET][0]["addr"]
    return ip

tf.app.flags.DEFINE_string("job_name", "", "Either 'ps' or 'worker'")
FLAGS = tf.app.flags.FLAGS

parameter_servers = ["192.168.1.26:1024"]

workers =
["192.168.1.16:1024","192.168.1.17:1024","192.168.1.18:1024","192.168.1.19:1024",
"192.168.1.20:1024","192.168.1.21:1024","192.168.1.22:1024","192.168.1.23:1024",
"192.168.1.24:1024","192.168.1.25:1024","192.168.1.26:1024","192.168.1.27:1024",
"192.168.1.28:1024","192.168.1.29:1024","192.168.1.30:1024","192.168.1.31:1024"]

taskName = getIpAddr()+":1024"
try:
    taskNum = workers.index(taskName)
except ValueError:
    print(" Unable to find " + taskName + " in the worker group.")
    quit()

cluster = tf.train.ClusterSpec({"ps":parameter_servers, "worker":workers})

```

```

server = tf.train.Server(cluster,job_name=FLAGS.job_name,task_index=taskNum)

if FLAGS.job_name == "ps":
    server.join()
elif FLAGS.job_name == "worker":

    def one_hot(y):
        """convert label from dense to one hot
        argument:
            label: ndarray dense label ,shape: [sample_num,1]
        return:
            one_hot_label: ndarray one hot, shape: [sample_num,n_class]
        """
        # e.g.: [[5], [0], [3]] --> [[0, 0, 0, 0, 0, 1], [1, 0, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0]]
        y = y.reshape(len(y))
        n_values = np.max(y) + 1
        return np.eye(n_values)[np.array(y, dtype=np.int32)] # Returns FLOATS

    def batch_norm(input_tensor, config, i):
        # Implementing batch normalisation: this is used out of the residual layers
        # to normalise those output neurons by mean and standard deviation.

        if config.n_layers_in_highway == 0:
            # There is no residual layers, no need for batch_norm:
            return input_tensor
        with tf.variable_scope("batch_norm") as scope:
            if i != 0:

```

```

        # Do not create extra variables for each time step
        scope.reuse_variables()
        # Mean and variance normalisation simply crunched over all axes
        axes = list(range(len(input_tensor.get_shape())))

        mean, variance = tf.nn.moments(input_tensor, axes=axes, shift=None,
name=None, keep_dims=False)
        stdev = tf.sqrt(variance+0.001)
        # Rescaling
        bn = input_tensor - mean
        bn /= stdev
        # Learnable extra rescaling
        # tf.get_variable("relu_fc_weights",
initializer=tf.random_normal(mean=0.0, stddev=0.0)
        bn *= tf.get_variable("a_noreg", initializer=tf.random_normal([1],
mean=0.5, stddev=0.0))
        bn += tf.get_variable("b_noreg", initializer=tf.random_normal([1],
mean=0.0, stddev=0.0))
        # bn *= tf.Variable(0.5, name=(scope.name + "/a_noreg"))
        # bn += tf.Variable(0.0, name=(scope.name + "/b_noreg"))
    return bn
def relu_fc(input_2D_tensor_list, features_len, new_features_len, config):
    """make a relu fully-connected layer, mainly change the shape of tensor
    both input and output is a list of tensor
    argument:
        input_2D_tensor_list: list shape is [batch_size,feature_num]
        features_len: int the initial features length of input_2D_tensor
        new_feature_len: int the final features length of output_2D_tensor

```

```

        config: Config used for weights initializers
    return:
        output_2D_tensor_list list shape is [batch_size,new_feature_len]
    """
    W = tf.get_variable(
        "relu_fc_weights",
        initializer=tf.random_normal(
            [features_len, new_features_len],
            mean=0.0,
            stddev=float(config.weights_stddev)
        )
    )
    b = tf.get_variable(
        "relu_fc_biases_noreg",
        initializer=tf.random_normal(
            [new_features_len],
            mean=float(config.bias_mean),
            stddev=float(config.weights_stddev)
        )
    )
    # intra-timestep multiplication:
    output_2D_tensor_list = [
        tf.nn.relu(tf.matmul(input_2D_tensor, W) + b)
        for input_2D_tensor in input_2D_tensor_list
    ]
    return output_2D_tensor_list
def single_LSTM_cell(input_hidden_tensor, n_outputs):
    """ define the basic LSTM layer

```

```

    argument:
        input_hidden_tensor: list a list of tensor,
                               shape: time_steps*[batch_size,n_inputs]
        n_outputs: int num of LSTM layer output
    return:
        outputs: list a time_steps list of tensor,
                shape: time_steps*[batch_size,n_outputs]
    """
    with tf.variable_scope("lstm_cell"):
        lstm_cell = tf.nn.rnn_cell.LSTMCell(n_outputs, state_is_tuple=True,
forget_bias=0.999)
        outputs, _ = tf.nn.static_rnn(lstm_cell, input_hidden_tensor,
dtype=tf.float32)
    return outputs
def bi_LSTM_cell(input_hidden_tensor, n_inputs, n_outputs, config):
    """build bi-LSTM, concatenating the two directions in an inner manner.
    argument:
        input_hidden_tensor: list a time_steps series of tensor, shape:
[sample_num, n_inputs]
        n_inputs: int units of input tensor
        n_outputs: int units of output tensor, each bi-LSTM will have half those
internal units
        config: Config used for the relu_fc
    return:
        layer_hidden_outputs: list a time_steps series of tensor, shape:
[sample_num, n_outputs]
    """
    n_outputs = int(n_outputs/2)

```

```

print ("bidir:")
with tf.variable_scope('pass_forward') as scope2:
    hidden_forward = relu_fc(input_hidden_tensor, n_inputs, n_outputs,
config)
    forward = single_LSTM_cell(hidden_forward, n_outputs)

print (len(hidden_forward), str(hidden_forward[0].get_shape()))

# Backward pass is as simple as surrounding the cell with a double inversion:
with tf.variable_scope('pass_backward') as scope2:
    hidden_backward = relu_fc(input_hidden_tensor, n_inputs, n_outputs,
config)
    backward =
list(reversed(single_LSTM_cell(list(reversed(hidden_backward)), n_outputs)))

with tf.variable_scope('bidir_concat') as scope:
    # Simply concatenating cells' outputs at each timesteps on the innermost
    # dimension, like if the two cells acted as one cell
    # with twice the n_hidden size:
    layer_hidden_outputs = [
        tf.concat([f, b], len(f.get_shape()) - 1)
        for f, b in zip(forward, backward)]
    return layer_hidden_outputs
def residual_bidirectional_LSTM_layers(input_hidden_tensor, n_input, n_output,
layer_level, config, keep_prob_for_dropout):
    """This architecture is only enabled if "config.n_layers_in_highway" has a
value only greater than int(0). The arguments are same than for bi_LSTM_cell.
arguments:

```

```

        input_hidden_tensor: list a time_steps series of tensor, shape:
[sample_num, n_inputs]
        n_inputs: int units of input tensor
        n_outputs: int units of output tensor, each bi-LSTM will have half those
internal units
        config: Config used for determining if there are residual connections and
if yes, their number and with some batch_norm.
    return:
        layer_hidden_outputs: list a time_steps series of tensor, shape:
[sample_num, n_outputs]
        """
        with tf.variable_scope('layer_{}'.format(layer_level)) as scope:

            if config.use_bidirectionnal_cells:
                get_lstm = lambda input_tensor: bi_LSTM_cell(input_tensor, n_input,
n_output, config)
            else:
                get_lstm = lambda input_tensor:
single_LSTM_cell(relu_fc(input_tensor, n_input, n_output, config), n_output)
            def add_highway_redisual(layer, residual_minilayer):
                return [a + b for a, b in zip(layer, residual_minilayer)]

            hidden_LSTM_layer = get_lstm(input_hidden_tensor)
            # Adding K new (residual bidir) connections to this first layer:
            for i in range(config.n_layers_in_highway - 1):
                with tf.variable_scope('LSTM_residual_{}'.format(i)) as scope2:
                    hidden_LSTM_layer = add_highway_redisual(
                        hidden_LSTM_layer,

```



```

        get_lstm(input_hidden_tensor)

    if config.also_add_dropout_between_stacked_cells:
        hidden_LSTM_layer = [tf.nn.dropout(out, keep_prob_for_dropout) for
out in hidden_LSTM_layer]
        return [batch_norm(out, config, i) for i, out in
enumerate(hidden_LSTM_layer)]
def LSTM_network(feature_mat, config, keep_prob_for_dropout):
    """model a LSTM Network,
    it stacks 2 LSTM layers, each layer has n_hidden=32 cells
    and 1 output layer, it is a full connet layer
    argument:
        feature_mat: ndarray fature matrix,
shape=[batch_size,time_steps,n_inputs]
        config: class containing config of network
    return:
        : ndarray output shape [batch_size, n_classes]
    """
    with tf.variable_scope('LSTM_network') as scope: # TensorFlow graph
naming
        feature_mat = tf.nn.dropout(feature_mat, keep_prob_for_dropout)
        # Exchange dim 1 and dim 0
        feature_mat = tf.transpose(feature_mat, [1, 0, 2])
        print (feature_mat.get_shape())
        # New feature_mat's shape: [time_steps, batch_size, n_inputs]

        # Temporarily crush the feature_mat's dimensions
        feature_mat = tf.reshape(feature_mat, [-1, config.n_inputs])

```

```

print (feature_mat.get_shape())
# New feature_mat's shape: [time_steps*batch_size, n_inputs]

# Split the series because the rnn cell needs time_steps features, each of
shape:

hidden = tf.split(feature_mat, config.n_steps, 0)
print (len(hidden), str(hidden[0].get_shape()))
# New shape: a list of length "time_step" containing tensors of shape
[batch_size, n_hidden]

# Stacking LSTM cells, at least one is stacked:
print ("\nCreating hidden #1:")
hidden = residual_bidirectional_LSTM_layers(hidden, config.n_inputs,
config.n_hidden, 1, config, keep_prob_for_dropout)
print (len(hidden), str(hidden[0].get_shape()))

for stacked_hidden_index in range(config.n_stacked_layers - 1):
    # If the config permits it, we stack more lstm cells:
    print ("\nCreating hidden #{}:".format(stacked_hidden_index+2))
    hidden = residual_bidirectional_LSTM_layers(hidden, config.n_hidden,
config.n_hidden, stacked_hidden_index+2, config, keep_prob_for_dropout)
    print (len(hidden), str(hidden[0].get_shape()))
print ("")
# Final fully-connected activation logits
# Get the last output tensor of the inner loop output series, of shape
[batch_size, n_classes]
last_hidden = tf.nn.dropout(hidden[-1], keep_prob_for_dropout)
last_logits = relu_fc(

```

```

        [last_hidden],
        config.n_hidden, config.n_classes, config
    )[0]
    return last_logits

def run_with_config(Config, X_train, y_train, X_test, y_test):
    start_time = datetime.now()
    print ("Start Time: ",time.ctime())
    print ("")
    tf.reset_default_graph() # To enable to run multiple things in a loop

    #-----
    # Define parameters for model
    #-----
    config = Config(X_train, X_test)
    print("Some useful info to get an insight on dataset's shape and normalisation:")
    print("features shape, labels shape, each features mean, each features standard
deviation")
    print(X_test.shape, y_test.shape,
          np.mean(X_test), np.std(X_test))
    print("the dataset is therefore properly normalised, as expected.")

    #-----
    # Let's get serious and build the neural network
    #-----
    #with tf.device("/cpu:0"): # Remove this line to use GPU. If you have a too
small GPU, it crashes.
    with tf.device(tf.train.replica_device_setter(cluster=cluster)):
    #with tf.device('/gpu:0'):

```

```

#with tf.device('/gpu:1'):
#mirrored_strategy = tf.contrib.distribute.MirroredStrategy(devices=["/gpu:0",
"/gpu:1"])
#with mirrored_strategy.scope():
    X = tf.placeholder(tf.float32, [
        None, config.n_steps, config.n_inputs], name="X")
    Y = tf.placeholder(tf.float32, [
        None, config.n_classes], name="Y")
# is_train for dropout control:
is_train = tf.placeholder(tf.bool, name="is_train")
keep_prob_for_dropout = tf.cond(is_train,
    lambda: tf.constant(
        config.keep_prob_for_dropout,
        name="keep_prob_for_dropout"
    ),
    lambda: tf.constant(
        1.0,
        name="keep_prob_for_dropout"
    )
)
pred_y = LSTM_network(X, config, keep_prob_for_dropout)

# Loss, optimizer, evaluation
# Softmax loss with L2 and L1 layer-wise regularisation
print ("Unregularised variables:")
for unreg in [tf_var.name for tf_var in tf.trainable_variables() if ("noreg"
in tf_var.name or "Bias" in tf_var.name)]:
    print (unreg)

```

```

l2 = config.lambda_loss_amount * sum(
    tf.nn.l2_loss(tf_var)
    for tf_var in tf.trainable_variables()
    if not ("noreg" in tf_var.name or "Bias" in tf_var.name)
)
# first_weights = [w for w in tf.all_variables() if w.name ==
'LSTM_network/layer_1/pass_forward/relu_fc_weights:0']
# l1 = config.lambda_loss_amount * tf.reduce_mean(tf.abs(first_weights))
loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits_v2(logits=pred_y, labels=Y))
+ l2 # + l1

# Gradient clipping Adam optimizer with gradient noise
optimize = tf.contrib.layers.optimize_loss(
    loss,
    global_step=tf.Variable(0),
    learning_rate=config.learning_rate,
    optimizer=tf.train.AdamOptimizer(learning_rate=config.learning_rate),
    clip_gradients=config.clip_gradients,
    gradient_noise_scale=config.gradient_noise_scale
)
correct_pred = tf.equal(tf.argmax(pred_y, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, dtype=tf.float32))

#-----
# Hooray, now train the neural network
#-----
# Note that log_device_placement can be turned off for less console spam.

```

```

#sessconfig = tf.ConfigProto(log_device_placement=False)
sessconfig = tf.ConfigProto(allow_soft_placement =
True,log_device_placement=False)

#sessconfig.gpu_options.allow_growth = True
#with tf.Session(config=sessconfig) as sess:
with tf.Session("grpc://localhost:1024") as sess:
    #init = tf.global_variables_initializer()
    sess.run(tf.global_variables_initializer())
    best_accuracy = (0.0, "iter: -1")
    best_f1_score = (0.0, "iter: -1")
    # Start training for each batch and loop epochs
    worst_batches = []
    for i in range(config.training_epochs):

        # Loop batches for an epoch:
        shuffled_X, shuffled_y = shuffle(X_train, y_train, random_state=i*42)
        for start, end in zip(range(0, config.train_count, config.batch_size),
                             range(config.batch_size, config.train_count + 1,
config.batch_size)):
            _, train_acc, train_loss, train_pred = sess.run(
                [optimize, accuracy, loss, pred_y],
                feed_dict={
                    X: shuffled_X[start:end],
                    Y: shuffled_y[start:end],
                    is_train: True
                }
            )
            worst_batches.append(

```

```

        (train_loss, shuffled_X[start:end], shuffled_y[start:end])
    )
    worst_batches = list(sorted(worst_batches))[-5:] # Keep 5 poorest

# Train F1 score is not on boosting
train_f1_score = metrics.f1_score(
    shuffled_y[start:end].argmax(1), train_pred.argmax(1),
average="weighted"
)
# Retrain on top worst batches of this epoch (boosting):
# a.k.a. "focus on the hardest exercises while training":
for _, x_, y_ in worst_batches:
    _, train_acc, train_loss, train_pred = sess.run(
        [optimize, accuracy, loss, pred_y],
        feed_dict={
            X: x_,
            Y: y_,
            is_train: True
        }
    )

# Test completely at the end of every epoch:
# Calculate accuracy and F1 score
pred_out, accuracy_out, loss_out = sess.run(
    [pred_y, accuracy, loss],
    feed_dict={
        X: X_test,
        Y: y_test,

```

```

        is_train: False
    }
)
# "y_test.argmax(1)": could be optimised by being computed once...
f1_score_out = metrics.f1_score(
    y_test.argmax(1), pred_out.argmax(1), average="weighted"
)
print (
    "iter: {}, ".format(i) + \
    "train loss: {}, ".format(train_loss) + \
    "train accuracy: {}, ".format(train_acc) + \
    "train F1-score: {}, ".format(train_f1_score) + \
    "test loss: {}, ".format(loss_out) + \
    "test accuracy: {}, ".format(accuracy_out) + \
    "test F1-score: {}".format(f1_score_out)
)
best_accuracy = max(best_accuracy, (accuracy_out, "iter:
{}".format(i)))
best_f1_score = max(best_f1_score, (f1_score_out, "iter: {}".format(i)))
print("")
print("final test accuracy: {}".format(accuracy_out))
print("best epoch's test accuracy: {}".format(best_accuracy))
print("final F1 score: {}".format(f1_score_out))
print("best epoch's F1 score: {}".format(best_f1_score))
print("")
end_time = datetime.now()
print("End Time: ",time.ctime())
print("Exec Duration: {}".format(end_time - start_time))

```



```
print("")
# returning both final and bests accuracies and f1 scores.
return accuracy_out, best_accuracy, f1_score_out, best_f1_score
```

Config_Dataset_HAR.py

```
#!/usr/bin/env python
# coding: utf-8
```

```
# In[ ]:
__author__ = 'jkranbir'
```

```
# Note: Linux bash commands start with a "!" inside those "ipython notebook" cells
import os
```

```
DATA_PATH = "data/"
get_ipython().system('pwd && ls')
os.chdir(DATA_PATH)
get_ipython().system('pwd && ls')
```

```
get_ipython().system('python download_datasets.py')
```

```
get_ipython().system('pwd && ls')
os.chdir("..")
get_ipython().system('pwd && ls')
```

```
DATASET_PATH = DATA_PATH + "UCI HAR Dataset/"
print("\n" + "Dataset is now located at: " + DATASET_PATH)
```

```

# In[ ]:

__author__ = 'jkranbir'

from lstm_architecture import one_hot, run_with_config
import numpy as np
import os
#os.environ["CUDA_VISIBLE_DEVICES"]="0,1"

#-----
# Neural net's config.
#-----

class Config(object):
    """
    define a class to store parameters,
    the input should be feature mat of training and testing
    """
    def __init__(self, X_train, X_test):

        workers =
["192.168.1.16:1024","192.168.1.17:1024","192.168.1.18:1024","192.168.1.19:1024",
"192.168.1.20:1024","192.168.1.21:1024","192.168.1.22:1024","192.168.1.23:1024",
"192.168.1.24:1024","192.168.1.25:1024","192.168.1.26:1024","192.168.1.27:1024",

```

```

"192.168.1.28:1024","192.168.1.29:1024","192.168.1.30:1024","192.168.1.31:1024"]
    taskCount = len(workers)
    cluster = tf.train.ClusterSpec({"worker":workers})

# Data shaping
self.train_count = len(X_train)/taskCount # 7352/16 training series
self.test_data_count = len(X_test)/taskCount # 2947/16 testing series
self.n_steps = len(X_train[0]) # 128 time_steps per series
self.n_classes = 6 # Final output classes

# Training
self.learning_rate = 0.001
self.lambda_loss_amount = 0.005
self.training_epochs = 250 #5
self.batch_size = 100
self.clip_gradients = 15.0
self.gradient_noise_scale = None
# Dropout is added on inputs and after each stacked layers (but not
# between residual layers).
self.keep_prob_for_dropout = 0.85 # *(1/3.0)

# Linear+relu structure
self.bias_mean = 0.3
# I would recommend between 0.1 and 1.0 or to change and use a xavier
# initializer
self.weights_stddev = 0.2

#####

```

```

# NOTE: I think that if any of the below parameters are changed,
# the best is to readjust every parameters in the "Training" section
# above to properly compare the architectures only once optimised.
#####

# LSTM structure
# Features count is of 9: three 3D sensors features over time
self.n_inputs = len(X_train[0][0])
self.n_hidden = 256 # nb of neurons inside the neural network
# Use bidir in every LSTM cell, or not:
self.use_bidirectionnal_cells = True #False

# High-level deep architecture
self.also_add_dropout_between_stacked_cells = False #True
# NOTE: values of exactly 1 (int) for those 2 high-level parameters below totally
disables them and result in only 1 starting LSTM.
# self.n_layers_in_highway = 1 # Number of residual connections to the LSTMs
(highway-style), this is did for each stacked block (inside them).
# self.n_stacked_layers = 1 # Stack multiple blocks of residual
# layers.

#-----
# Dataset-specific constants and functions + loading
#-----

# Useful Constants

```

```
# Those are separate normalised input features for the neural network
```

```
INPUT_SIGNAL_TYPES = [
```

```
    "body_acc_x_",
```

```
    "body_acc_y_",
```

```
    "body_acc_z_",
```

```
    "body_gyro_x_",
```

```
    "body_gyro_y_",
```

```
    "body_gyro_z_",
```

```
    "total_acc_x_",
```

```
    "total_acc_y_",
```

```
    "total_acc_z_"
```

```
]
```

```
# Output classes to learn how to classify
```

```
LABELS = [
```

```
    "WALKING",
```

```
    "WALKING_UPSTAIRS",
```

```
    "WALKING_DOWNSTAIRS",
```

```
    "SITTING",
```

```
    "STANDING",
```

```
    "LAYING"
```

```
]
```

```
DATA_PATH = "data/"
```

```
DATASET_PATH = DATA_PATH + "UCI HAR Dataset/"
```

```
TRAIN = "train/"
```

```
TEST = "test/"
```

```

# Load "X" (the neural network's training and testing inputs)
def load_X(X_signals_paths):
    """
    Given attribute (train or test) of feature, read all 9 features into an
    np ndarray of shape [sample_sequence_idx, time_step, feature_num]
    argument: X_signals_paths str attribute of feature: 'train' or 'test'
    return: np ndarray, tensor of features
    """
    X_signals = []
    for signal_type_path in X_signals_paths:
        file = open(signal_type_path, 'r')
        # Read dataset from disk, dealing with text files' syntax
        X_signals.append(
            [np.array(serie, dtype=np.float32) for serie in [
                row.replace(' ', '').strip().split(' ') for row in file
            ]
            )
        )
        file.close()
    return np.transpose(np.array(X_signals), (1, 2, 0))
X_train_signals_paths = [
    DATASET_PATH + TRAIN + "Inertial Signals/" + signal + "train.txt" for signal in
    INPUT_SIGNAL_TYPES
]
X_test_signals_paths = [
    DATASET_PATH + TEST + "Inertial Signals/" + signal + "test.txt" for signal in
    INPUT_SIGNAL_TYPES
]

```

```

X_train = load_X(X_train_signals_paths)
X_test = load_X(X_test_signals_paths)

# Load "y" (the neural network's training and testing outputs)
def load_y(y_path):
    """
    Read Y file of values to be predicted
    argument: y_path str attribute of Y: 'train' or 'test'
    return: Y ndarray / tensor of the 6 one_hot labels of each sample
    """
    file = open(y_path, 'r')
    # Read dataset from disk, dealing with text file's syntax
    y_ = np.array(
        [elem for elem in [
            row.replace(' ', '').strip().split(' ') for row in file
        ]],
        dtype=np.int32
    )
    file.close()

    # Subtract 1 to each output class for friendly 0-based indexing
    return one_hot(y_ - 1)

y_train_path = DATASET_PATH + TRAIN + "y_train.txt"
y_test_path = DATASET_PATH + TEST + "y_test.txt"
y_train = load_y(y_train_path)
y_test = load_y(y_test_path)
#-----

```

```

# Training (maybe multiple) experiment(s)
#-----
n_layers_in_highway = 4
n_stacked_layers = 4
trial_name = "{}x{}".format(n_layers_in_highway, n_stacked_layers)
for i in range(0, taskCount):
    for learning_rate in [0.001]: # [0.01, 0.001, 0.0001]:
        for lambda_loss_amount in [0.005]:
            for clip_gradients in [15.0]:
                print ("learning_rate: {}".format(learning_rate))
                print ("lambda_loss_amount: {}".format(lambda_loss_amount))
                print ("")

class EditedConfig(Config):
    def __init__(self, X, Y):
        super(EditedConfig, self).__init__(X, Y)

    # Edit only some parameters:
    self.learning_rate = learning_rate
    self.lambda_loss_amount = lambda_loss_amount
    self.clip_gradients = clip_gradients

    # Architecture params:
    self.n_layers_in_highway = n_layers_in_highway
    self.n_stacked_layers = n_stacked_layers

## Useful catch upon looping (e.g.: not enough memory)
# try:
#     accuracy_out, best_accuracy = run_with_config(EditedConfig)

```



```

# except:
# accuracy_out, best_accuracy = -1, -1
accuracy_out, best_accuracy, fl_score_out, best_fl_score = (
    run_with_config(EditedConfig, X_train, y_train, X_test, y_test)
)
print (accuracy_out, best_accuracy, fl_score_out, best_fl_score)

with open('{}_result_HAR_6.txt'.format(trial_name), 'a') as f:
    f.write(str(learning_rate) + '\t' + str(lambda_loss_amount) + '\t' +
str(clip_gradients) + '\t' + str(
        accuracy_out) + '\t' + str(best_accuracy) + '\t' + str(fl_score_out) + '
\t' + str(best_fl_score) + '\n\n')

    print ("_____")
print ("")
print ("Done.")
# In[ ]:

```

data/download_datasets.py : Same as TensorFlow Data

APPENDIX C – TENSORFLOW SETUP

C.1 TensorFlow Installation

In the previous APPENDIX-A, we have already installed NVIDIA GPUs in the big machine. Then we have successfully installed NVIDIA driver for the GPUs along with CUDA and cuDNN in the machine. We are starting this section, with the prerequisite of all previous installations.

Step-1: Check if your machine having conda installed previously by the command.

```
hpcmonster369@hpc369-Z10PE-D16-WS:~$ conda --version
```

If not installed, Please download the Anaconda from the website.

<https://www.anaconda.com/download/#linux>

Step-2: Go to the download folder and verify the md5sum value of the downloaded Anaconda copy with the link given below.

```
md5sum Anaconda3-5.3.0-Linux-x86_64.sh  
4321e9389b648b5a02824d4473cfdb5f Anaconda3-5.3.0-Linux-x86_64.sh
```

Verify with below link as having same md5sum #

http://docs.anaconda.com/anaconda/install/ hashes/Anaconda3-5.3.0-Linux-x86_64.sh-hash/

If both having same md5sum values, then you have downloaded the software correctly.

Step-3: Install Anaconda3.

```
bash Anaconda3-5.3.0-Linux-x86_64.sh
```

Step-4: Go to the installation page, accept the Anaconda3 license after installation.

Step-5: It is recommended to select yes to prepend Anaconda3 install location to the path in your bashrc file. You can create a backup of your bashrc file before clicking on yes for safety purpose.

Step-6: Activate the installation by using below command.

```
source ~/.bashrc
```

Step-7: Verify Installation → conda list

C.2 TensorFlow Environment Setup

Step-1: Create the environment

```
hpcmonster369@hpc369-Z10PE-D16-WS:~$ conda create --name  
rnn_lstm_har_tensorflow tensorflow-gpu
```

Step-2: Activate the environment

```
hpcmonster369@hpc369-Z10PE-D16-WS:~$ conda activate rnn_lstm_har_tensorflow
```

Step-3: Add Dependencies

```
conda install numpy
```

```
conda install keras
```

```
conda install pandas
```

```
conda install matplotlib
```

```
conda install scipy scikit-learn
```

```
conda install nb_conda
```

Step-4: Check Available Jupyter Kernel

hpcmonster369@hpc369-Z10PE-D16-WS:~\$ jupyter kernelspec list

Step-5: Validate the environment

conda info --envs

Once all steps completed successfully, the environment would display as Figure. 73

```
hpcmonster369@hpc369-Z10PE-D16-WS:~$ conda info --envs
# conda environments:
#
base                * /home/hpcmonster369/anaconda3
multicore_cpu       /home/hpcmonster369/anaconda3/envs/multicore_cpu
multicore_gpu       /home/hpcmonster369/anaconda3/envs/multicore_gpu
rnn_lstm_har_pytorch /home/hpcmonster369/anaconda3/envs/rnn_lstm_har_pytorch
rnn_lstm_har_tensorflow /home/hpcmonster369/anaconda3/envs/rnn_lstm_har_tensorflow
tf_gpu              /home/hpcmonster369/anaconda3/envs/tf_gpu
hpcmonster369@hpc369-Z10PE-D16-WS:~$
```

Figure. 73 TensorFlow Project Screen

APPENDIX D – PYTORCH SETUP

D.1 PyTorch Installation

In the previous APPENDIX-A, we have already installed NVIDIA GPUs in the big machine. Then we have successfully installed NVIDIA driver for the GPUs along with CUDA and cuDNN in the machine. In the APPENDIX-C, we have already installed Anaconda3 in the machine. We are starting this section, with the prerequisite of all previous installations.

D.2 PyTorch Environment Setup

Step-1: Create the environment

```
hpcmonster369@hpc369-Z10PE-D16-WS:~$ conda create -n rnn_lstm_har_pytorch python=3.6
```

Step-2: Activate the environment

```
hpcmonster369@hpc369-Z10PE-D16-WS:~$ conda activate rnn_lstm_har_pytorch
```

Step-3: Add Dependencies

```
conda install pytorch=0.4.1 cuda90 -c pytorch
```

```
conda install torchvision -c pytorch
```

```
conda install matplotlib
```

```
conda install -c conda-forge tqdm
```

```
conda install nb_conda
```

Step-4: Validate the environment

```
conda info --envs
```

Once all steps completed successfully, the environment would display as Figure. 74

```
hpcmonster369@hpc369-Z10PE-D16-WS:~$ conda info --envs
# conda environments:
#
base                * /home/hpcmonster369/anaconda3
multicore_cpu       /home/hpcmonster369/anaconda3/envs/multicore_cpu
multicore_gpu       /home/hpcmonster369/anaconda3/envs/multicore_gpu
rnn_lstm_har_pytorch /home/hpcmonster369/anaconda3/envs/rnn_lstm_har_pytorch
rnn_lstm_har_tensorflow /home/hpcmonster369/anaconda3/envs/rnn_lstm_har_tensorflow
tf_gpu              /home/hpcmonster369/anaconda3/envs/tf_gpu

hpcmonster369@hpc369-Z10PE-D16-WS:~$ █
```

Figure. 74 PyTorch Project Screen

APPENDIX E – RASPBERRY PI CLUSTER SETUP

E.1 Raspberry Pi Parts

1. Raspberry Pi 3 Model B+ motherboard
2. Samsung 32 GB Class 10 MicroSD card
3. 2.5A Power Adapter
4. 2 Heat sinks
5. MicroSD USB Reader (Optional)
6. Premium Case (Optional)
7. Premium HDMI Cable (Optional)

E.2 Individual Raspberry Pi Installation

Step-1: 1. Install “Raspbian Stretch with desktop” Kernel Version 4.14 from the official Raspberry PI website link given below.

<https://www.raspberrypi.org/downloads/raspbian/>

Step-2: Download and Install the Etcher (Linux x86 version) which will burn the Raspbian image to the Micro SD card. The link given below.

<https://www.balena.io/etcher/>

Step-3: Get a microSD card adapter and fire up the Etcher so that all the microSD cards having Kernel version 4.14.

Step-4: To enable SSH remote access from the Pi, open the “boot” drive on the microSD card and create an empty file named ssh with no extension. Open the folder in a shell and run below command.

\$ touch ssh

Step-5: To build a package that supports all Raspberry Pi devices—including the Pi 1 and Zero use the below command in your Linux machine which will build a .whl package for installation in Raspberry pi.

```
$ tensorflow/tools/ci_build/ci_build.sh PI \  
  
tensorflow/tools/ci_build/pi/build_raspberry_pi.sh PI_ONE
```

For updated version, please use the below link

https://www.tensorflow.org/install/source_rpi

Step-6: Copy the wheel file to the Raspberry Pi and install with pip with appropriate version number.

```
$ pip install tensorflow-version-cp34-none-linux_armv7l.whl
```

Step-7: Connect all the Raspberry Pi to a switch which is connected to a network.

Step-8: Login to Pi using the default password **raspberrypi**.

Step-9: Go to the raspi-config to do the rest of the setup.

```
pi@raspberrypi~$ sudo raspi-config
```

1. Change the password of default to your own convenient one.
2. Set the locale and timezone.
3. Rename each pi from the default name to rpi# as per the nodes going to be used in the cluster. You can do that from the configuration file itself and restart the pi.
3. Set the hostname in each pi.

```
sudo hostname node01    # whatever name you chose
```

```
sudo nano /etc/hostname # change the hostname here too
```

```
sudo nano /etc/hosts   # change "raspberrypi" to "node01"
```


4. In raspi-config, check whether the ssh mode is enabled or not. If not enable it.
5. Change the assigned memory for GPU to minimum.
6. Change the assigned memory for CPU to maximum.
7. In raspi-config, change (3. Boot Options > B2 Wait for Network at Boot) from “No” to “Yes”. This will ensure that networking is available before the fstab file mounts the NFS client.
8. Restart the Pi by below command.

sudo reboot

9. Repeat the process for all Pis.
10. For password less entry into each Pi, you can generate SSH keys for all nodes are distributing the public keys of each node to the rest of the nodes. Please the link to generate SSH keys. After that update */etc/hosts* of each node with ip address of rest node.
<https://www.raspberrypi.org/documentation/remote-access/ssh/passwordless.md>

Step-10: To work on a cluster we need to set up the NFS server and client on master node and NFS client set up on all the worker nodes which is already mentioned in 5.2 Cluster of Raspberry Pis Setup section.

E.3 Installation of Program

- Step-1: Run the server.py program on the each Raspberry Pi node.
- Step-2: The master node should contain both the files lstm_architecture.py and Config_Dataset_HAR.py with proper dataset folder inside the NFS server directory.
- Step-3: The folder structure of the dataset is already mentioned in APPENDIX-A.
- Step-4: Run the Config_Dataset_HAR.py on the master node to start the deep model iterating.

REFERENCES

- [1] Yann LeCun, Yoshua Bengio, Geoffrey Hinton (2015), “Deep learning” , Nature, May 28, Volume 521(7553), p.436–444, doi:10.1038/nature14539.
- [2] “The Next Generation of Machine Learning Chips”, Deloitte Global, December, 2017
- [3] Wojciech Zaremba, Ilya Sutskever, Oriol Vinyals (2015), “Recurrent Neural Network Regularization” , Conference paper at ICLR, arXiv:1409.2329
- [4] Sepp Hochreiter, Jürgen Schmidhuber (1997), “Long short-term memory”, Neural Computation Volume 9, Issue 8, November 15, p.1735-1780, doi:10.1162/neco.1997.9.8.1735.
- [5] Samuel, Arthur L. (1959). "Some Studies in Machine Learning Using the Game of Checkers". IBM Journal of Research and Development. 44: 206–226. CiteSeerX 10.1.1.368.2254. doi:10.1147/rd.441.0206.
- [6] https://en.wikipedia.org/wiki/GeForce_256#cite_note-2
- [7] <https://ai.googleblog.com/2017/04/federated-learning-collaborative.html>
- [8] Li, Mu and Andersen, David G. and Park, Jun Woo and Smola, Alexander J. and Ahmed, Amr and Josifovski, Vanja and Long, James and Shekita, Eugene J. and Su, Bor-Yiing (2014), “Scaling Distributed Machine Learning with the Parameter Server”, Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, Broomfield, CO, p.583-598, <http://dl.acm.org/citation.cfm?id=2685048.2685095>

- [9] Uthayasankar Sivarajah, Muhammad Mustafa Kamal, Zahir Irani, Vishanth Weerakkody (2017), "Critical analysis of Big Data challenges and analytical methods", *Journal of Business Research*, Volume 70, January, Pages 263-286.
- [10] Dobre, Ciprian, Xhafa, Fatos (2014), "Intelligent services for Big Data science", *Future Generation Computer Systems*, vol. 37, p.267-281, doi:10.1016/j.future.2013.07.014
- [11] Mayer-Schönberger, V., & Cukier, K. (2013) "Big data: A revolution that will transform how we live, work, and think. Boston", MA, Houghton Mifflin Harcourt, PsycINFO Database Record
- [12] Alex Krizhevsky and Sutskever, Ilya and Hinton, Geoffrey E (2012), "ImageNet Classification with Deep Convolutional Neural Networks", *Advances in Neural Information Processing Systems 25 NIPS 2012*, p. 1097-1105, <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>
- [13] Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow (2016), "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems", *CoRR*, vol. abs/1603.04467
- [14] Ketkar, Nikhil (2017). "Introduction to PyTorch". *Deep Learning with Python*. Apress, Berkeley, CA. pp. 195–208, doi: 10.1007/978-1-4842-2766-4_12. ISBN 9781484227657.
- [15] Tianqi Chen, Mu Li, Yutian Li (2015), "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems", *CoRR*, vol. abs/1512.01274, <http://arxiv.org/abs/1512.01274>
- [16] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, Joseph M. Hellerstein (2012), "Distributed GraphLab: A Framework for Machine

Learning and Data Mining in the Cloud”, Proc. VLDB Endow, Vol 5, No. 8, doi =10.14778/2212351.2212354

[17] Joseph Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, Carlos Guestrin (2012). "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs." Proceedings of Operating Systems Design and Implementation (OSDI).

[18] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin and J. Hellerstein. "GraphLab: A New Framework for Parallel Machine Learning", In the 26th Conference on Uncertainty in Artificial Intelligence (UAI), Catalina Island, USA, 2010

[19] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng. "Large scale distributed deep networks." In Neural Information Processing Systems, 2012.

[20] Dean LG, Kendal RL, Schapiro SJ, Thierry B, Laland KN (2012), "Identification of the social and cognitive processes underlying human cumulative culture.", Science. 2012; 335(6072):1114–1118. doi:10.1126/science.1213969

[21] Yujun Lin, Song Han, Huizi Mao, Yu Wang, William J. Dally (2018), "Deep gradient compression: Reducing the communication bandwidth for distributed training", ICLR 2018

[22] Bengio Y, Simard P, Frasconi P.(1994),"Learning long-term dependencies with gradient descent is difficult",IEEE Trans Neural Netw. 1994; 5(2):157-66., PMID: 18267787 DOI: 10.1109/72.279181

[23] Karanbir Singh Chahal, Manraj Singh Grover, Kuntal Dey (2018),"A Hitchhiker's Guide On Distributed Training of Deep Neural Networks",CoRR, vol.abs/1810.11787, <http://arxiv.org/abs/1810.11787>

[24] Ioannis Mitliagkas, Ce Zhang, Stefan Hadjis, Christopher Ré, "Asynchrony begets Momentum, with an Application to Deep Learning", NIPS 2016, arXiv: 1605.09774

- [25] J. Yang, J. Lee, and J. Choi (2011), "Activity Recognition Based on RFID Object Usage for Smart Mobile Devices," *J. Comput. Sci. Technol.*, vol. 26, no. 2, pp. 239–246, Mar. 2011.
- [26] Tim Salimans, Diederik P. Kingma (2016), "Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks", NIPS 2016
- [27] S. Lasecki, Walter & Chol Song, Young & Kautz, Henry & P. Bigham, Jeffrey. (2013), "Real-time crowd labeling for deployable activity recognition", *Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW*. 1203-1212. 10.1145/2441776.2441912.
- [28] Sumit Majumder, Emad Aghayi, Moein Noferesti, Hamidreza Memarzadeh-Tehran, Tapas Mondal, Zhibo Pang, M. Jamal Deen (2017), "Smart Homes for Elderly Healthcare—Recent Advances and Research Challenges" , *Sensors (Basel)*. 2017 Nov; 17(11): 2496, doi: 10.3390/s17112496.
- [29] L. Chen, C. D. Nugent, and H. Wang (2012), "A KnowledgeDriven Approach to Activity Recognition in Smart Homes," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 6, pp. 961–974, Jun. 2012.
- [30] Y.-J. Chang, S.-F. Chen, and J.-D. Huang (2011), "A Kinect-based system for physical rehabilitation: a pilot study for young adults with motor disabilities," *Res. Dev. Disabil.*, vol. 32, no. 6, pp. 2566–2570, 2011.
- [31] N. Alshurafa, W. Xu, J. Liu, M.-C. Huang, B. Mortazavi (2013), C. Roberts, and M. Sarrafzadeh, "Designing a Robust Activity Recognition Framework for Health and Exergaming using Wearable Sensors.," *IEEE J. Biomed. Heal. Informatics*, no. c, pp. 1–11, Oct. 2013.
- [32] Fish, Ram David Adva; Messenger, Henry; Baryudin, Leonid; Dardashti, Soroush Salehian; Goldshtein, Evgenia , "Fall detection system using a combination of

accelerometer, audio input and magnetometer", Patent No. 9648478 , Filing Date: 21 August 2014

[33] B. Lange, C.-Y. Chang, E. Suma, B. Newman, A. S. Rizzo (2011), and M. Bolas, "Development and evaluation of low cost game-based balance rehabilitation tool using the Microsoft Kinect sensor.," Conf. Proc. IEEE Eng. Med. Biol. Soc., vol. 2011, pp. 1831–4, Jan. 2011.

[34] K. Yoshimitsu, Y. Muragaki, T. Maruyama, M. Yamato, and H. Iseki (2014), "Development and Initial Clinical Testing of 'OPECT': An Innovative Device for Fully Intangible Control of the Intraoperative Image-Displaying Monitor by the Surgeon," Neurosurgery, vol. 10.

[35] B. Mirmahboub, S. Samavi, N. Karimi, and S. Shirani (2012), "Automatic Monocular System for Human Fall Detection based on Variations in Silhouette Area.," IEEE Trans. Biomed. Eng., no. c, pp. 1–10, Nov. 2012.

[36] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra and Jorge L. Reyes-Ortiz (2013), "A Public Domain Dataset for Human Activity Recognition Using Smartphones" , ESANN 2013 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium), 24-26 April 2013.

[37] UCI Human Activity Recognition Using Smartphones Data Set, "https://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones"

[38] W. Ong, L. Palafox, and T. Koseki (2013), "Investigation of Feature Extraction for Unsupervised Learning in Human Activity Detection," Bull. Networking, Computer. System Software, vol. 2, no. 1, pp. 30–35, 2013.

- [39] Jimmy Lei Ba, Jamie Ryan Kiros, Geoffrey E. Hinton (2016), “Layer Normalization”, NIPS 2016
- [40] Sergey Ioffe, Christian Szegedy (2015), “Batch normalization: Accelerating deep network training by reducing internal covariate shift.” In International Conference on Machine Learning, pages 448–456, 2015.
- [41] David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams (1986), “Learning representations by back-propagating errors”, *Nature*, volume 323, pages 533–536 (1986), <https://doi.org/10.1038/323533a0>
- [42] Hava T. Siegelmann (1996), “Recurrent neural networks and finite automata”, *Computational Intelligence*, Volume 12, Number 4, 1996
- [43] Minsky, Marvin L. (1967), “Computation: Finite and Infinite Machines”, Prentice-Hall, Inc., ISBN = 0-13-165563-9
- [44] Siegelmann H.T. Sontag E.D. (1995), “On the Computational Power of Neural Nets”, *Journal of Computer and System Sciences*, Volume 50, Issue 1, February 1995, p.132-150
- [45] Hava T Siegelmann, Eduardo D Sontag (1994), “Analog computation via neural networks*”, *Theoretical Computer Science*, Volume 131, Issue 2, 12 September 1994, p.331-360, [https://doi.org/10.1016/0304-3975\(94\)90178-3](https://doi.org/10.1016/0304-3975(94)90178-3)
- [46] Hava T. Siegelmann and Eduardo D. Sontag (1991), “Turing Computability with Neural Nets”, *Applied Mathematics Letters*, Vol. 4, p.77-80
- [47] Mozer, M. C. (1995). "A Focused Backpropagation Algorithm for Temporal Pattern Recognition". In Chauvin, Y.; Rumelhart, D. “Backpropagation: Theory, architectures, and applications.” ResearchGate. Hillsdale, NJ: Lawrence Erlbaum Associates. pp. 137–169. Retrieved 2017-08-21.

- [48] Bengio, Y & Frasconi, Paolo & Simard, Patrice (1993), "Problem of learning long-term dependencies in recurrent networks", 1993 IEEE International Conference on Neural Networks. 1183 - 1188 vol.3, doi:10.1109/ICNN.1993.298725.
- [49] Cho, Kyunghyun; van Merriënboer, Bart; Gulcehre, Caglar; Bahdanau, Dzmitry; Bougares, Fethi; Schwenk, Holger; Bengio, Yoshua (2014). "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". arxiv: 1406.1078 [cs.CL].
- [50] S. Vosoughi, P. Vijayaraghavan, and D. Roy (2016), "Tweet2vec: Learning tweet embeddings using character-level CNN-LSTM encoder-decoder," CoRR , vol. abs/1607.07514.
- [51] Michaela Blott, Ling Liu, Kimon Karras, Kees A Vissers (2015), "Scaling Out to a Single-Node 80Gbps Memcached Server with 40Terabytes of Memory", In HotStorage '15.
- [52] Priya Goyal, Piotr Doll, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, Kaiming He (2017), "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", CoRR, vol abs/1706.02677, <http://arxiv.org/abs/1706.02677>
- [52] Peter H. Jin, Qiaochu Yuan, Forrest N. Iandola, Kurt Keutzer (2016), "How to scale distributed deep learning?", CoRR, vol.abs/1611.04581
- [53] Yujun Lin, Song Han, Huizi Mao, Yu Wang, William J. Dally, "Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training", CoRR ,vol.abs/1712.01887, 2017
- [54] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," arXiv preprint arxiv: 1512.03385, 2015.

- [55] Jaeyoung Kim, Mostafa El-Khamy, Jungwon Lee (2017), “Residual LSTM: Design of a Deep Recurrent Architecture for Distant Speech Recognition”, INTERSPEECH 2017 August 20–24, 2017, Stockholm, Sweden,
- [56] A. Emin Orhan and Zachary Pitkow (2017), “Skip Connections Eliminate Singularities”, Orhan2017SkipCE
- [57] Y.-y. Chen, Y. Lv, Z. Li, and F.-Y. Wang (2016), “Long short-term memory model for traffic congestion prediction with online open data,” in Intelligent Transportation Systems (ITSC), 2016 IEEE 19th International Conference on. IEEE, 2016, pp. 132–137.
- [58] Zhiyong Cui, Student Member, Ruimin Ke, Student Member, Yinhai Wang (2018), “Deep Stacked Bidirectional and Unidirectional LSTM Recurrent Neural Network for Network-wide Traffic Speed Prediction” , CoRR, vol.abs/1801.02143,2018
- [59] Yu Zhao and Rennong Yang and Guillaume Chevalier and Maoguo Gong (2017), “Deep Residual Bidir-LSTM for Human Activity Recognition Using Wearable Sensors”, CoRR , doi:abs/1708.08989
- [60] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi (2016), "Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation.