Dual Degree Project Report

Incremental Learning at the Edge: Design and applications

Anubhav Agarwal
17D070026
Guide: Prof. Siddharth Tallur

2021 - 2022

Department of Electrical Engineering
IIT Bombay

Dissertation Approval

The dissertation entitled

Incremental Learning at the Edge: Design and applications

by

Anubhav Agarwal

is approved for the degree of

Bachelor of Technology in Electrical Engineering and Master of Technology in Microelectronics

Digital Signature
Siddharth Tallur (i16333)
30-Jun-22 12:23:00 PM

Prof. Siddharth Tallur
Department of Electrical Engineering
(Supervisor)

Digital Signature
Sharayu Moharir (i16010)
30-Jun-22 12:28:50 PM

Prof. Sharayu Moharir
Department of Electrical Engineering
(Chairperson and Examiner)

Digital Signature
Laxmeesha Somappa (10002000)
01-Jul-22 08:30:27 AM

Prof. Laxmeesha Somappa
Department of Electrical Engineering
(Examiner)

Date: June 30, 2022
Place: Mumbai.

Declaration

I declare that this submission represents my own ideas. I have adequately cited any references from original sources where ideas/diagrams have been borrowed. I understand that violation of the above will result in disciplinary action from the institute.

Digital Signature
Anubhav Agarwal (17d070026)
~~30-Jun-22 12:16:32 PM~~

Anubhav Agarwal

(Roll No:17D070026)

June 30, 2022

Abstract

Edge devices are becoming an inseparable part of our everyday lives and there is tremendous opportunity to harness their resources to increase their capabilities even further. With the explosion of amount of data being collected everyday, the analytics is becoming a serious challenge. This DDP aims to develop edge based learning framework which is different from cloud based learning. This has been deployed on an FPGA and its performance is compared to other edge-based devices.

Contents

1. Introduction

1.1 Growth of Data in Everyday Lives

With the advent of smart devices in our daily lives, we are witnessing a boom in AI based applications and services. Deep Learning achieves state of the art performance in fields like healthcare, facial recognition, natural language processing, computer vision, condition monitoring, traffic surveillance and much more.

However, most of these applications are computationally expensive, posing certain hardware requirements to make them possible to execute. Although edge devices are becoming more capable everyday, it is still impossible to run many of these applications on edge devices.

Hence, they rely on cloud servers and internet connection. The data is collected and sent to the cloud for processing. The results are then sent back to the edge device. Some very popular examples are Amazon Alexa, Google Assistant, Apple Siri and Microsoft Cortana. However, the number of such devices is booming every year and so is the data being generated and collected. Sending everything to the cloud for processing poses enormous challenges.

This also brings about privacy concerns. Much of the data being collected is sensitive and personal. Sending it to the cloud makes it susceptible to hackers who can use it for malicious purposes.

Edge computing serves to bring the functionality of cloud server closer to the edge devices such as micro controllers, mobile phones and embedded devices.

Any AI application involves two aspects- training and inference. During training, large amount of data is fed to a model so that it can identify complex patterns necessary to perform the task at hand.

The learned weights are then stored and subsequently used during inference. During inference, the model is used to make predictions on unknown data.

1.2 Inference at Edge

There has been significant amount of research and development to bring inference to edge devices. There are various frameworks such as Tensorflow Lite, STM32Cube, CMSIS-NN, Vitis AI and many more that aim to bring inference to low power edge devices such as microcontrollers and FPGAs.

The first stage of Dual Degree Project involved exploring frameworks for inference at edge. Some of them are as follows:

1.2.1 Vitis AI

Figure 1.1: Vitis AI

Vitis AI is a framework developed by Xilinx to accelerate inference of ML tasks on Xilinx hardware platforms, such as Zynq UltraScale+ SoC.

1.2.2 STM 32 Cube AI

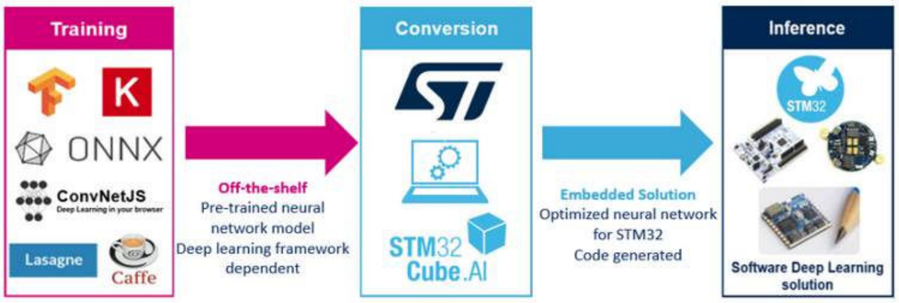

Training
ONNX
ConvNetJS
Deep Learning in your browser
Lasagne
Caffe
Off-the-shelf
Pre-trained neural network model
Deep learning framework dependent
Conversion
STM32 Cube.AI
Embedded Solution
Optimized neural network for STM32
Code generated
Inference
STM32
Software Deep Learning solution

Figure 1.2: STM32 Cube AI

STM32 Cube AI is a framework from STMicroelectronics to accelerate inference of Deep Neural Networks on ST Boards. Model is first created in existing framework such as Tensorlow, Keras or Caffe. After training, it is exported. The exported pre-trained model is converted to an optimized solution which can be deployed on any STMicroelectronics board for inference.

1.2.3 Tensorflow Lite

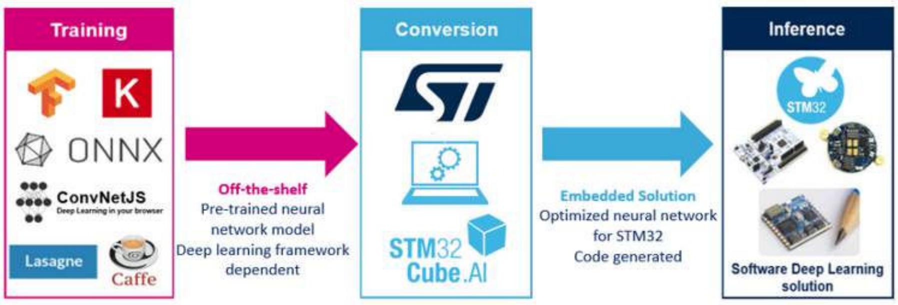

Training
ONNX
ConvNetJS
Deep Learning in your browser
Lasagne
Caffe
Off-the-shelf
Pre-trained neural network model
Deep learning framework dependent
Conversion
STM32 Cube.AI
Embedded Solution
Optimized neural network for STM32
Code generated
Inference
STM32
Software Deep Learning solution

Figure 1.3: STM32 Cube AI

1.3 Traditional Approaches to Incremental Learning

The traditional approach to incremental learning relies on a central server to do most heavy tasks. The edge devices collects data through sensor and transmits it to a server using network.

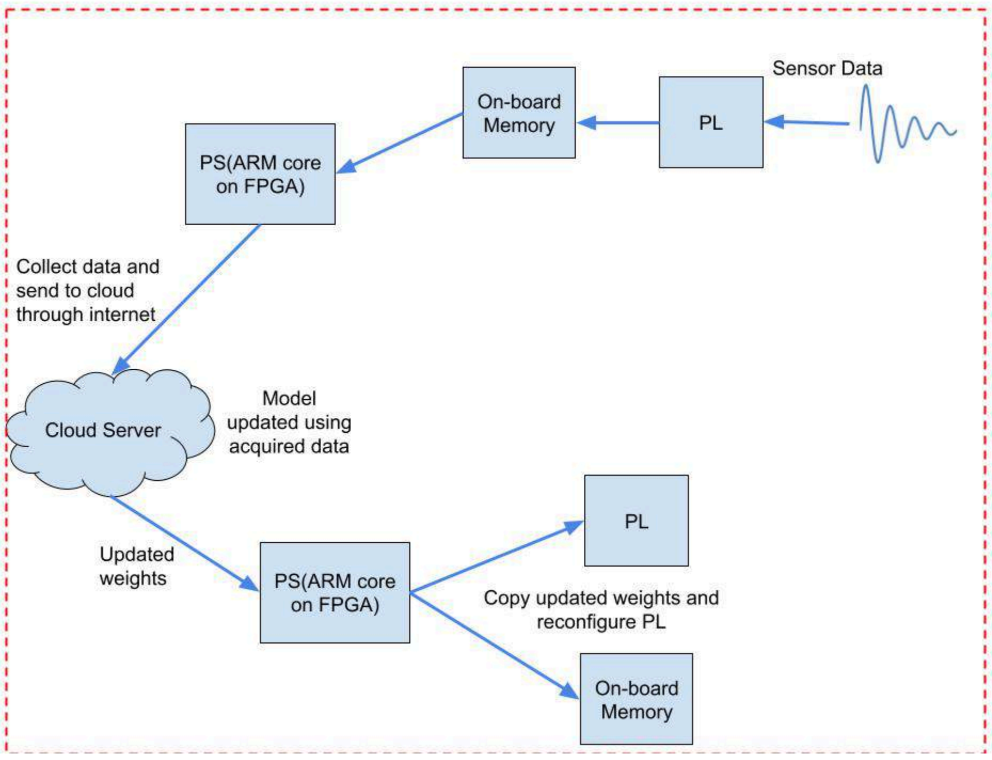

Sensor Data
PL
On-board Memory
PS(ARM core on FPGA)
Collect data and send to cloud through internet
Cloud Server
Model updated using acquired data
Updated weights
PS(ARM core on FPGA)
PL
Copy updated weights and reconfigure PL
On-board Memory

Figure 1.4: Cloud Based Approach to Incremental Learning

The steps shown in the above diagram are described as follows:

1. The sensor data is acquired using onboard peripherals on the FPGA board.
2. The data is sent from Programmable Logic to the onboard DDR memory.

3. The ARM CPU uploads the sensor data to cloud for processing.
4. The cloud server does training and calculates weights for the new model.
5. The weights are downloaded on to DDR on the FPGA.
6. The weights are written to Programmable Logic on FPGA.

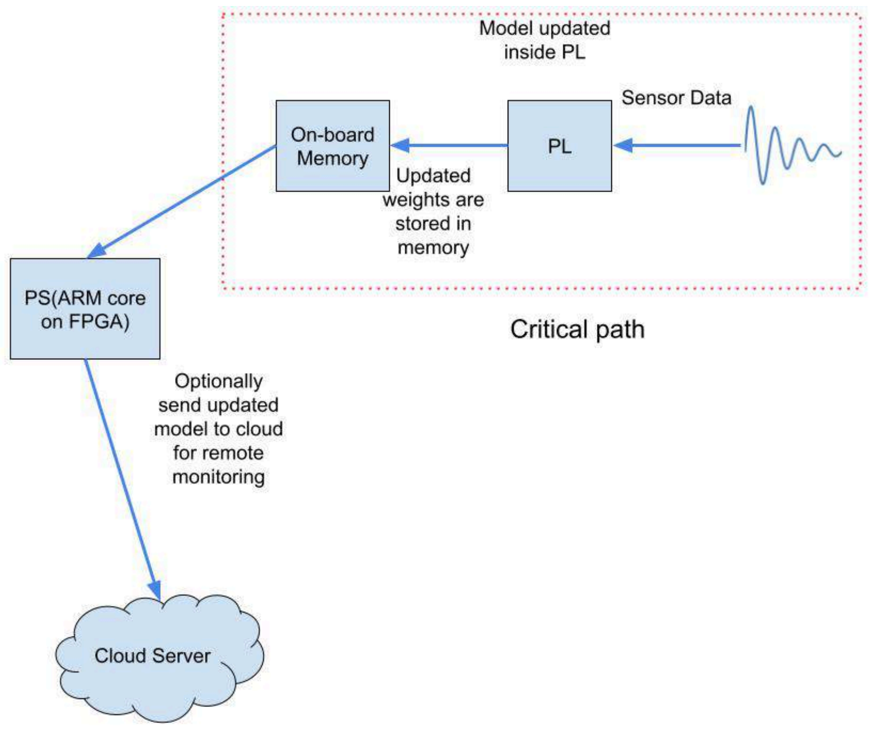

Model updated inside PL
Sensor Data
On-board Memory
PL
Updated weights are stored in memory
PS(ARM core on FPGA)
Critical path
Optionally send updated model to cloud for remote monitoring
Cloud Server

Figure 1.5: Edge Based Approach to Incremental Learning on FPGA

1. The sensor data is acquired using onboard peripherals on the FPGA board.
2. The Programmable Logic contains the design to read the incoming data and calculate the new weights.
3. The model is updated without having to store or transmit the data to a remote location for processing.
4. The new weights can be optionally sent to the cloud after regular intervals for monitoring purposes.

From the above we can see that the latter case is much more efficient than the first one due to the following reasons:

- The data acquired by the sensor doesn't need to be transmitted to a remote location for processing.
- A significant amount of energy is saved.

- The Critical Path for the second approach is much smaller, hence much faster and simpler.
- The edge device is no longer dependant on a remote server for processing.
- Since the critical path is smaller here, it has lesser chances of failure, making it more robust.

1.4 Case Study: Smart Factory

Sensor Fusion
Cloud Computing
Smart Sensors
IoT
Advanced Algorithms
Big Data
AI
Predictive Maintenance
Automation
SPECTRAL ENGINES

Figure 1.6: A smart factory

A smart factory contains several sensors for monitoring various aspects of production and maintenance. Typically, their data is collected and sent to a server/cloud facility via local network or internet.

In such a case, for a large factory, the collective rate of data transmitted by these sensors may reach GigaBytes per second(GB/s).

Such high data rates consume enormous bandwidth and trigger additional latency. In this case, the server might overheat and shut down, thereby disrupting the functionality of the entire network.

This can be solved by placing devices near the edge for heavy data processing. This considerably reduces the load on the central server, improves latency and also makes the system more robust, less prone to failure.

This process is called Decentralization.

1.5 DDP Objective

The objective of Dual Degree Project is to highlight the importance of edge training over the conventional server-based approach. An FPGA-based system is developed which is capable of learning weights in a Multi-Layer Perceptron. This system is then used to detect faults using vibration data in an industrial machine. Several challenges associated with training at edge have been addressed.

2. Training at Edge

Since the rise of Internet of Things, Edge Machine Learning(Edge ML) has become a very hot topic of discussion. There has been a huge surge in the amount of devices connected to the cloud, however, the current network infrastructure is not yet ready to support this advancement. There are various issues associated with sending data to the cloud for analysis. This can be solved at least partially by Edge ML. Edge training is a learning procedure that aims to learn the optimal values of weights and bias for a model deployed at edge device. As opposed to traditional approach, the training occurs on edge servers or edge device, which are less powerful than centralized servers.

This has the following advantages:

- Reliance on network stability is reduced. System can be used in remote areas where internet connectivity is not very good.
- Latency is significantly reduced. Since processing takes place in proximity of source, much of the time spent in data transmission is saved. This makes real time data processing possible.
- Decentralization: Since much of the processing is now done on the edge devices, the dependence on central server is eliminated. This will prevent a catastrophic impact in case the server fails. Hence, this makes the system more robust.
- End devices with idle resources can communicate among themselves to collaboratively finish a task.

2.1 Challenges

The key challenges in edge training are as follows:

2.1.1 Model Development

Developing a model which can fit on the edge device/server. Size and memory resources are key factors that affect training efficiency. Some techniques to enable model to fit on edge device are as follows:

- Knowledge Distillation: This is based on transfer learning, which trains a neural network of smaller size with distilled knowledge from a larger model. First, a large and complex model(Teacher) is trained from scratch. A small(student) model is trained using logits obtained from the larger model.

- Quantization: A deep neural network involves millions of parameters, consuming a large amount of storage on edge device. However, we don't always require high precision for the weights to achieve decent accuracy. We can achieve reasonable accuracy by using fixed point precision instead of float
- Network Pruning: The main idea here is to delete unimportant parameters. Thus, neurons with smaller weights are removed.

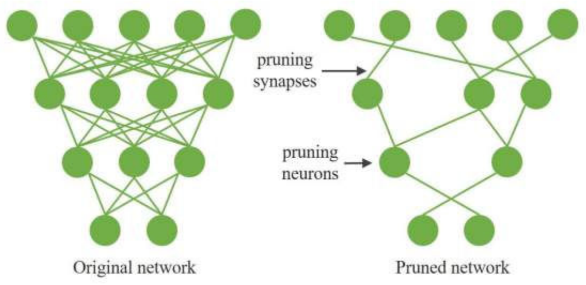


Figure 2.1: Pruning

2.1.2 Optimize Training

Making the training faster: Training a deep Neural Network is a computationally expensive procedure which leads to low training efficiency on edge devices due to their limited computational resources.

2.1.3 Optimal Hardware Usage

Since edge devices have much fewer resources compared to central server systems, sustainable usage of resources is essential to perform training. The various resources present in an FPGA are as follows:

- Look Up Tables(LUTs): These are used to implement much of the combinatorial logic in the FPGA as truth tables in LUT memory.
- DDR: This is a large memory of the order of Gigabytes. However, this is much slower compared to Block RAM, requiring several clock cycles to fetch data.
- Block RAM: This memory lies in the Programmable Logic of the FPGA but is very small in size. Since it is located in the PL, it is very fast and has small access time. It can be used to store variables that are frequently used by the logic blocks in PL. If an IP needs to access data from DDR, it is first transferred to a local buffer made of BRAM. The processed data is written back from BRAM to DDR.
- Multipliers and DSP: These are special units used to perform arithmetic operations.

2.2 Available Platforms

2.2.1 Raspberry Pi

Figure 2.2: Raspberry Pi 4

The Raspberry Pi is an affordable single board computer. It consists of BroadCom BCM2837 SoC with a 64-bit quad core ARM Cortex A72 processor running at 1.5GHz.

It supports various peripherals such as ethernet, HDMI, Audio Jack, GPIOs and many more.

2.2.2 Jetson Nano

Figure 2.3: Nvidia Jetson Nano

The NVidia Jetson Nano is a small computer just like Raspberry Pi which allows us to run multiple neural networks in parallel for various applications. It contains a 128-core Nvidia Maxwell GPU alongwith Quad Core ARM A57 Core.

2.2.3 FPGA

Figure 2.4: PYNQ-Z2 FPGA

We have used PYNQ-Z2 FPGA which consists of Zynq-7020 SoC by Xilinx. It has a dual core ARM Cortex-A9 processor with 1GB DDR3 RAM and a rich set of peripherals.

3. Application and Dataset

An algorithm might obtain good prediction performances during a period of time followed by poor ones later on as a result of an unexpected drift. Hence, it needs to be updated at regular intervals for robust performance at all times.

This type of scenario is quite common in systems that are used to detect fault in industrial machines. Convolution Neural Networks are widely used for this purpose. However, they largely rely on the assumption that the test data and training data have the same distribution. However, a practical industrial environment is dynamic, this is rarely the case. The changes in load conditions and ambient environment result in changes in the data distribution, leading to performance degradation in the model. This phenomenon is called domain shift.

Incremental learning is very useful in dynamic scenarios where the input data distribution(P(X)) and Conditional distribution changes with time and environment.

3.1 Fault Detection Using Vibration Data

We aim to build an efficient system to demonstrate learning capabilities at the edge. We will demonstrate the practical usage of our setup by deploying a Condition Based Monitoring System to detect faults in an industrial appliance.

We are using lab-generated data with lower precision sensor. This comprises of vibration data acquired from a motor running at 3000 rpm using STMicrolelectronics wireless industrial sensor node. Vibration data was recorded using an on-board 3-axis vibration sensor with 6KHz bandwidth and 26.7KHz data rate. We have used only the data acquired on the z-axis for our purpose. The anomalies were introduced using the following two ways:

- By coupling a bearing with a faulty bearing to motor shaft
- Using a healthy bearing but with eccentric weight on the motor shaft.

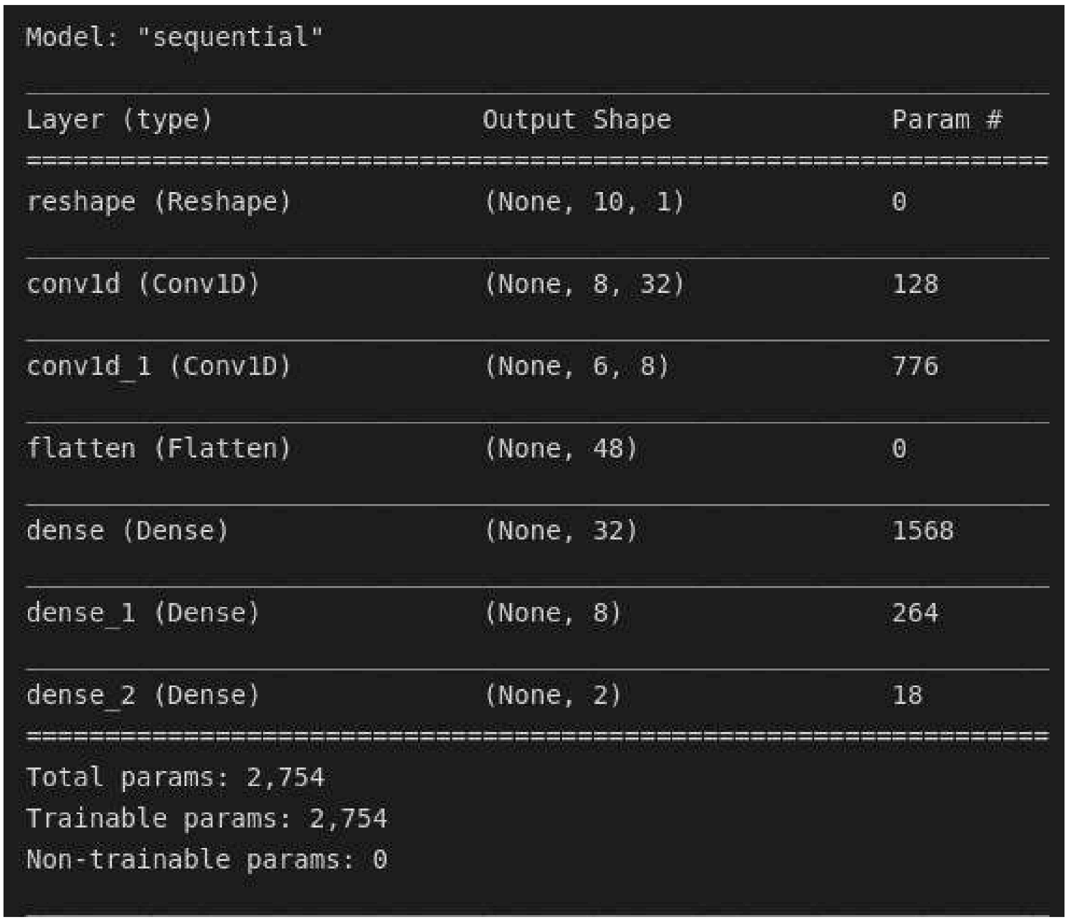
The raw data we obtain is a time series data which is not fit to be utilized directly by a neural network. We first need to extract features from the data and feed them into the network.

We have extracted 10 features from the raw time domain data which are as follows:

- Mean : $\frac{1}{n}\sum_{i=1}^{n} x_i$
- Median : $\begin{cases} X[\frac{n+1}{2}], & n \text{ is odd} \\ \frac{X[\frac{n}{2}]+X[\frac{n}{2}+1]}{2}, & n \text{ is even} \end{cases}$
- Mean absolute deviation: $\frac{1}{n}\sum_{i=1}^{n} |x_i - \mu|$
- Variance: $\frac{\sum_{i=1}^{n}(x_i-\mu)^2}{n}$

- Standard deviation: $\sqrt{\frac{\sum_{i=1}^{n}(x_i-\mu)^2}{n}}$
- Kurtosis: $\frac{1}{n}\sum_{i=1}^{n}(\frac{x_i-\mu}{\sigma})^4$
- Skew: $\frac{3(\mu-\mu_{1/2})}{\sigma}$
- Crest factor: $max(|x_i|) \div \sqrt{\frac{\sum_{i=1}^{n} {x_i}^2}{n}}$
- Impulse factor: $max(|x_i|) \div \frac{\sum_{i=1}^{n} {x_i}^2}{n}$
- Shape factor: $\sqrt{\frac{\sum_{i=1}^{n} {x_i}^2}{n}} \div \frac{\sum_{i=1}^{n} {x_i}^2}{n}$

3.1.1 The Model



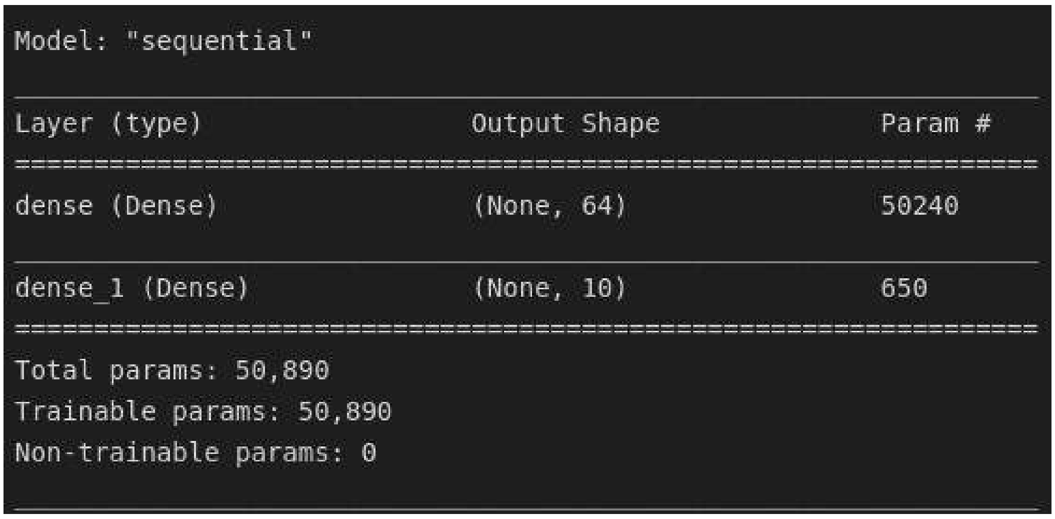
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
reshape (Reshape)            (None, 10, 1)             0
_________________________________________________________________
conv1d (Conv1D)              (None, 8, 32)             128
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 6, 8)              776
_________________________________________________________________
flatten (Flatten)            (None, 48)                0
_________________________________________________________________
dense (Dense)                (None, 32)                1568
_________________________________________________________________
dense_1 (Dense)              (None, 8)                 264
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 18
=================================================================
Total params: 2,754
Trainable params: 2,754
Non-trainable params: 0
_________________________________________________________________
```

Figure 3.1: Model architecture to classify vibration data

3.2 MNIST

MNIST is a very popular dataset widely used as a Hello World in Machine Learning. We have used this dataset in order to demonstrate training capabilities of our FPGA design and benchmarking with other edge devices.

Figure 3.2: MNIST Dataset



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 64)                50240
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0
_________________________________________________________________
```

Figure 3.3: Model to classify MNIST dataset

4. Neural Networks

4.1 Introduction

Neural networks are mathematical models that use learning algorithms inspired by the brain to store information.

Machine learning is the scientific discipline that is concerned with the design and development of algorithms that allow computers to learn complex patterns using data.

A major focus of machine-learning research is to automatically learn to recognize complex patterns and make intelligent decisions based on data. Hence, machine learning is closely related to fields such as statistics, data mining, pattern recognition, and artificial intelligence.

Similar to the brain, neural networks are built up of many neurons with many connections between them. Multi Layer Perceptron is one of the most popular form of Neural Networks.Mathematically, they are capable of learning any mapping function and have been proven to be a universal approximation algorithm.

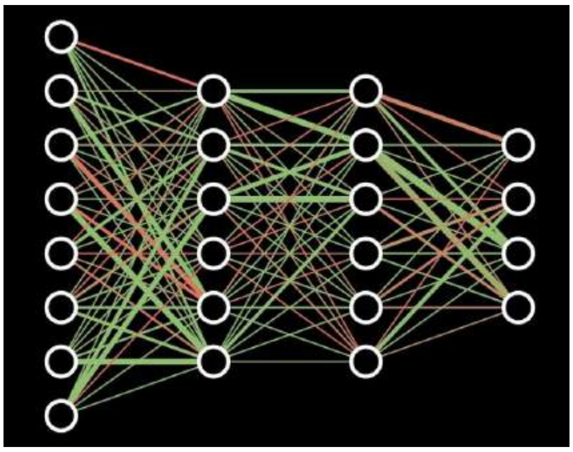


Figure 4.1: A Multi Layer Perceptron

4.2 Layers in Neural Networks

The basic building blocks of Neural Networks are neurons. These are simple computational units having weighted input signal and producing an output signal using an activation function.

A row of Neurons is called a layer. A Neural network consists of multiple layers stacked on top of each other. The architecture of neurons in a network is called network topology.

We will now discuss the most popular types of layers in a neural network:

4.2.1 Fully Connected(Dense) Layer

Fully Connected layers connect every neuron in one layer to every neuron in the next layer. They are typically used in the final stages of a Convolutional Neural network to extract the features for classification.

These layers are computationally expensive. As the number of input neurons grows, the number of weights to be stored and corresponding computations increases proportionally.

The number of neurons, activation function used are the typical hyperparameters for a fully connected layer.

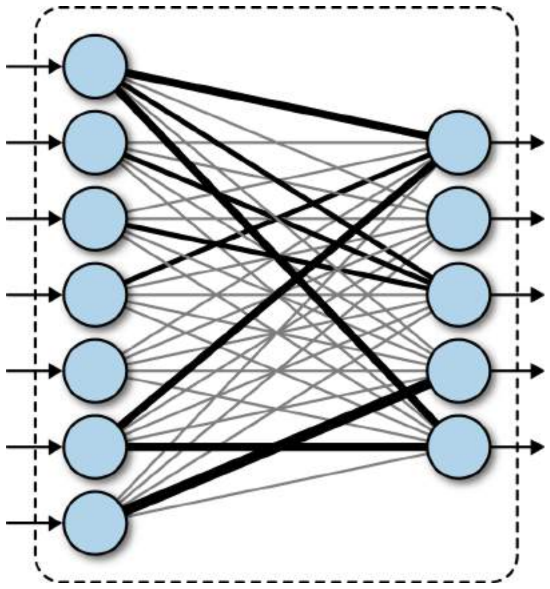


Figure 4.2: A Fully Connected Layer

Suppose we have a dense layer containing **d** inputs and **h** outputs.
The various symbols are represented as follows:

- Input : $\mathbf{x} \in \mathbb{R}^d$
- Weights: $\mathbf{W} \in \mathbb{R}^{h \times d}$
- Bias: $\mathbf{b} \in \mathbb{R}^h$
- Output: $\mathbf{y} \in \mathbb{R}^h$

During forward propagation, the output **y** is computed as follows:

$$\mathbf{y} = \mathbf{W^T x} + \mathbf{b} \tag{4.1}$$

While doing back propagation, suppose the incoming gradient from the output layer is **dy**.
The weight gradients **dw** are calculated as follows:

$$\mathbf{dw} = \mathbf{dy} \cdot \mathbf{x} \tag{4.2}$$

The bias gradients **dw** are calculated as follows:

$$\mathbf{db} = \mathbf{dy} \tag{4.3}$$

The input gradients **dx** are calculated as follows:

$$\mathbf{dx} = \mathbf{W^T dy} \tag{4.4}$$

4.2.2 Convolution Layer

Convolution Layer is an important type of layer in Convolutional Neural Networks(CNNs). It consists of a filter(also known as kernel) that sweeps through the input layer, getting multiplied one at a time to the corresponding input. The filter weights describe the feature being extracted from the input.

Multiplication is systematically repeated over the entire dimensions of the input, resulting in a feature map being generated one unit at a time.

The stride indicates the number of pixels by which the filter moves for the next iteration.

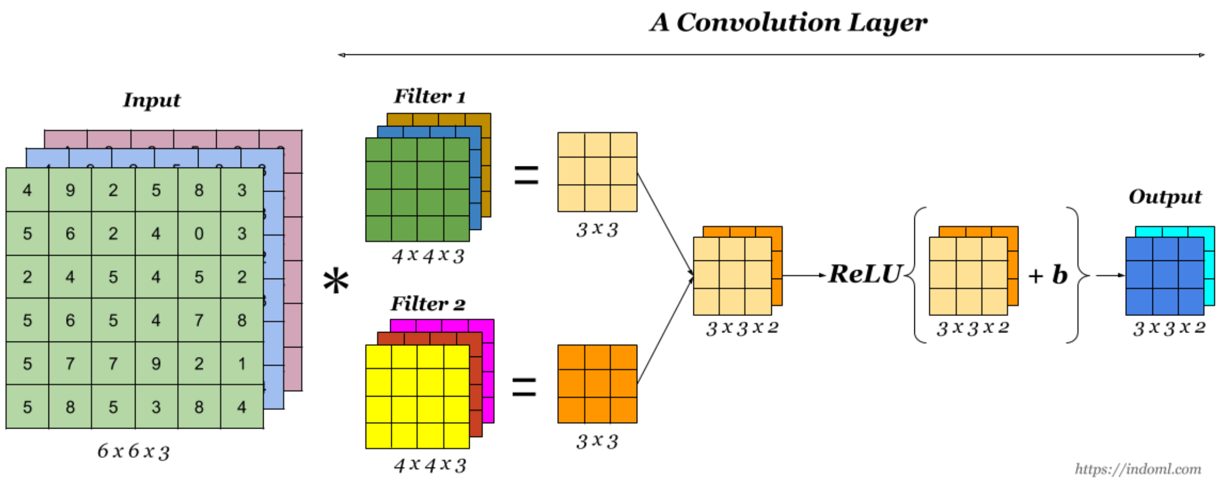


Figure 4.3: convolution Layer

We are using the following notation to display the various hyper parameters in convolution:

- **C** : The number of input channels
- **H**: The height of input
- **W**: The width of input
- **F**: The number of output channels
- **outH**: The height of output. This is expressed as outH = H - FH + 1.
- **outW**: The height of output. This is expressed as outW = W - FW + 1.

The various symbols in a convolution layer are represented as follows:

- Input : $\mathbf{X} \in \mathbb{R}^{C \times H \times W}$

- Weights: $\mathbf{W} \in \mathbb{R}^{C \times F \times FH \times FW}$
- Bias: $\mathbf{b} \in \mathbb{R}^{F}$
- Output: $\mathbf{y} \in \mathbb{R}^{F \times outH \times outW}$

During forward propagation, the output $\mathbf{y}$ is computed as follows:

$$\mathbf{y} = \mathbf{X} * \mathbf{W} + \mathbf{b} \tag{4.5}$$

While doing back propagation, suppose the incoming gradient from the output layer is $\mathbf{dy}$. The weight gradients $\mathbf{dw}$ are calculated as follows:

$$\mathbf{dx} = \mathbf{dy} * \mathbf{w} \tag{4.6}$$

The bias gradients $\mathbf{dw}$ are calculated as follows:

$$\mathbf{db} = \mathbf{dy} \tag{4.7}$$

The input gradients $\mathbf{dx}$ are calculated as follows:

$$\mathbf{dw} = \mathbf{dy} * \mathbf{X} \tag{4.8}$$

4.2.3 ReLU Activation

In order for neural networks to learn complex patterns, we need to introduce non-linearity between layers. The function must avoid easy saturation and should be sensitive to the input. ReLU Activation or Rectified Linear Activation Unit is a piecewise linear function that will output the input directly if it is positive, 0 otherwise. This activation has the following advantages:

- It overcomes vanishing gradient problem which exists in sigmoid and tanh activations hence allowing models to learn faster and perform better.
- It is computationally very simple to implement and doesn't require much resources.

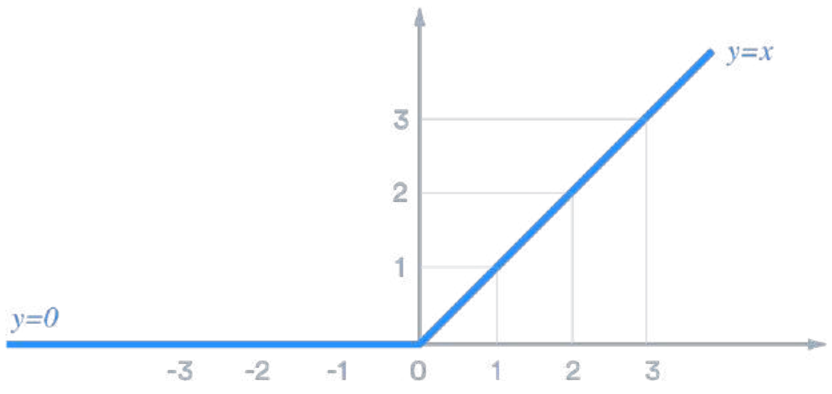


Figure 4.4: ReLU Activation

Forward pass equation:

$$y = \begin{cases} x & x >= 0 \\ 0 & x < 0 \end{cases} \tag{4.9}$$

Backward pass equation:

$$dx = \begin{cases} dy & x >= 0 \\ 0 & x < 0 \end{cases} \tag{4.10}$$

4.2.4 Loss Functions

The applications selected for demonstration are based on classification of data. Categorical Cross-entropy loss is the most popular loss function used for this

Setup is as follows:

- Input logits are denoted by vector **X** for which we need to calculate the gradients and the loss.
- The label is denoted by **y** which is a integer signifying the class label.
- The probability distribution **p** is first calculated.

$$p(i) = \frac{e^{x(i)}}{\sum_{k=0}^{n-1} e^{x(k)}} \tag{4.11}$$

- The cross entropy loss is given by $-log(p(y))$
- The gradients for the logits are given as follows:

$$dX(i) = \begin{cases} p(i) & i \neq y \\ p(i) - 1 & i = y \end{cases} \tag{4.12}$$

5. Implementation Approach 1

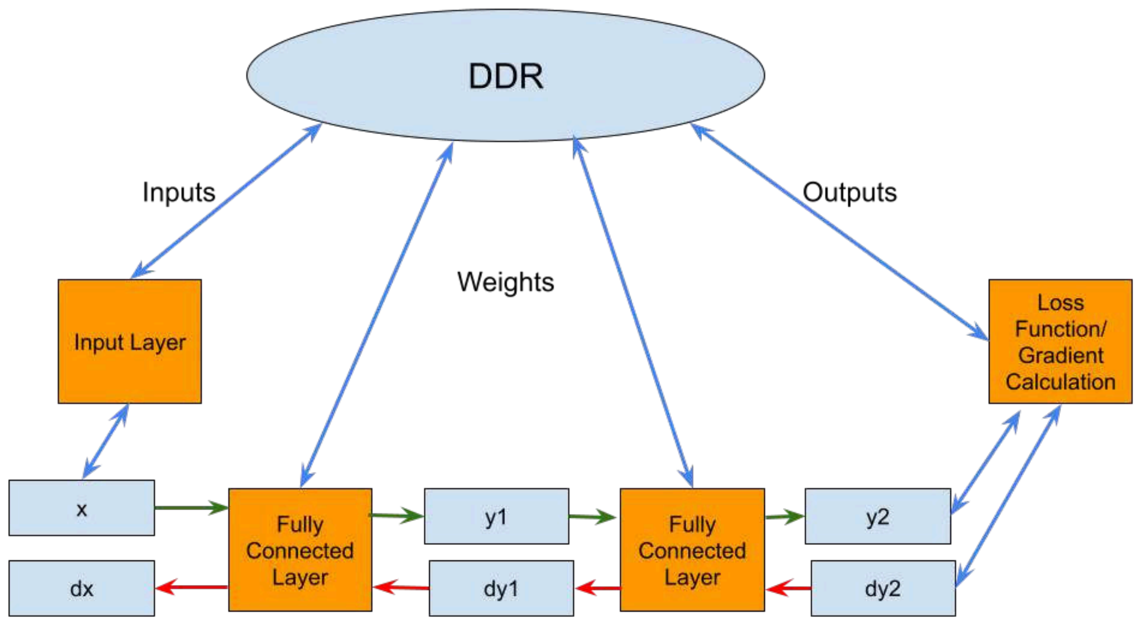


Figure 5.1: Flow diagram for Implementation 1

5.1 Features

- All computations are performed in fixed point precision. The weights and activations are stored in 16-bit signed fixed point format out of which three bits are used to represent integer part while 13 bits represent fractional part.
- Weights are stored in the DDR. They are transferred to Block RAM inside IP when inference or backpropagation takes place. The gradients are written back to DDR after backpropagation.
- Activation and their gradients are stored in Dual Port Block RAMs for faster access.

- Two successive layer IPs share the same Block RAM storage for input and output.
- A separate InputLayer IP is used to write the input vector from DDR to input BRAM.

We have tested the implemented design on two different applications- MNIST and Promethean dataset.

5.2 Design Methodology

Any Xilinx FPGA project consists of the following steps:

5.2.1 High Level Synthesis

Using Vitis High Level Synthesis tool we can create blocks to implement any function written in C++. HLS converts C++ code into IP blocks written in verilog which can be used in our FPGA design.

5.2.2 Vivado

Vivado is used to create the block design for our hardware. The IP blocks synthesized in previous step are imported and added to the block design.

The created design is synthesized and implemented. The implemented design can now be exported as a bitstream file which can be used to program the FPGA.

5.2.3 PYNQ/Vitis

In this step, the drivers are written to run the hardware. In Zynq-based platforms, we have an onboard dual core ARM Cortex-A9 processor. There are two primary approaches for this:

- Baremetal Approach: The drivers are written in C in Vitis IDE. The code is flashed to the board. Since there is no linux kernel running on the arm core, this approach is faster. However, it is harder to debug and write large applications using this approach.
- The PYNQ Approach: PYNQ is a framework developed by Xilinx to make FPGA development much more easier. It has various libraries written in Python for ease of use and faster development times. However, this runs on a linux based operating system, hence, there is an overhead while running code on PYNQ.

I have used the PYNQ based approach in my implementation to provide flexibility and ease of use.

5.3 IP Blocks Used

The IPs used in this design are described as follows:

5.3.1 Block RAM Generator

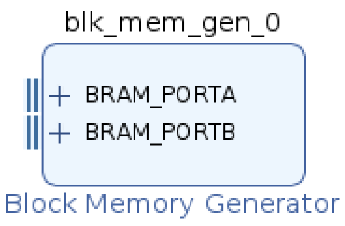
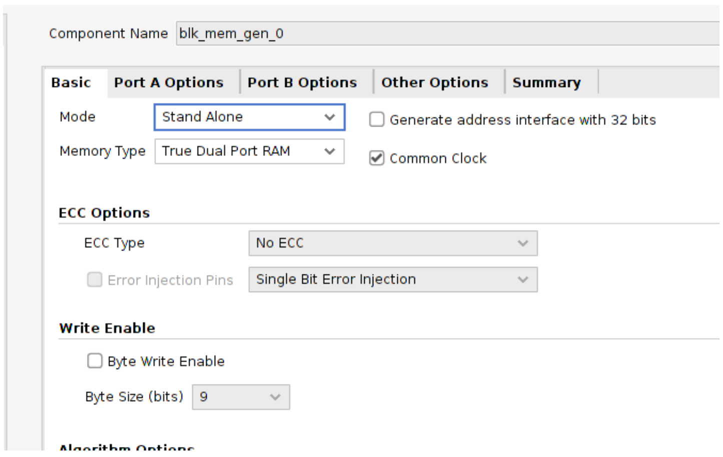


Figure 5.2: Dual Port Block RAM Cell



Component Name blk_mem_gen_0

Basic | Port A Options | Port B Options | Other Options | Summary

Mode: Stand Alone

Memory Type: True Dual Port RAM

☐ Generate address interface with 32 bits

☑ Common Clock

ECC Options

ECC Type: No ECC

☐ Error Injection Pins: Single Bit Error Injection

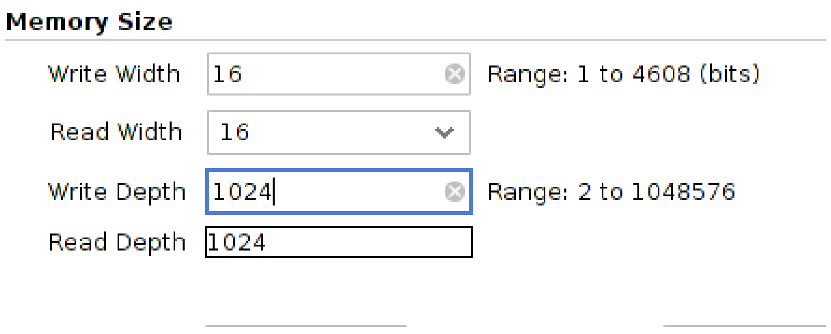
Write Enable

☐ Byte Write Enable

Byte Size (bits): 9

Algorithm Options

Figure 5.3: Block RAM Cell Configuration



Memory Size

Write Width: 16 — Range: 1 to 4608 (bits)

Read Width: 16

Write Depth: 1024 — Range: 2 to 1048576

Read Depth: 1024

Figure 5.4: Block RAM Cell Configuration

- The Block RAM Generator is used as a local storage for frequently used variables.
- It can have either a single port or dual ports.

- It operates either in standalone mode or in AXI BRAM Controller mode. In standalone mode, we can specify arbitrary bitwidth for our data. In the latter, the bitwidth must be in multiples of 32.
- The depth indicates the number of variables to store while the width specifies the bitwidth of the data.
- For a Dual Port BRAM, the read/write latency is 2 cycles. This must be taken care of while designing HLS IPs.

5.3.2 InputLayer

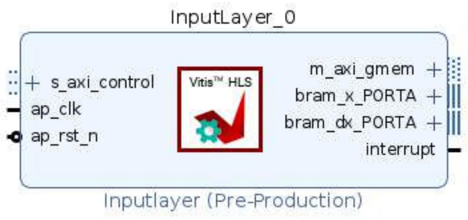


Figure 5.5: Input Layer IP

- This IP is used to transfer the input data vector from DDR to Block RAM connected to first layer of our network.
- AXI BRAM Controller is not used for this purpose because we want to use a bit width of 16.
- The `m_axi_gmem` port is connected to the DDR. This is used to fetch input vector from DDR.
- The `bram_x_PORTA` and `bram_dx_PORTA` ports are connected to Block RAM cells. The IP transfers relevant data from DDR to BRAM connected to these ports.
- The `s_axi_control` is connected to a General Purpose Master port on the ARM. Using this we can control the operations of our IP. The DDR address from which data needs to be fetched is also specified here.

5.3.3 Fully Connected layer IP

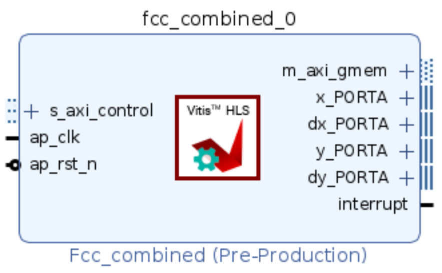


Figure 5.6: Fully Connected Layer IP

- This IP is used to perform operations related to a Dense layer. `x_PORTA` is connected to BRAM cell containing input activations. `dx_PORTA` is connected to BRAM cell containing outgoing gradients of layers below. `y_PORTA` is connected to BRAM cell containing output activations. `dy_PORTA` is connected to BRAM cell containing incoming gradients from higher layers.

- The `m_axi_gmem` port is connected to DDR through High Performance(HP) Slave Port on Zynq for fetching the weights. This port operates in Full AXI mode.

- The `s_axi_control` slave port is connected to General Purpose Master Port on Zynq. This port contains various registers. By writing to these registers, we can provide instructions to start/stop the operations, provide address for fetching data from memory.

A dense layer typically contains weights in the order of millions. It will not be possible to accommodate such a large number of weights in the limited Block RAM available in an FPGA. To solve this problem, the IP computes output activations partially in an iterative manner. This is displayed as follows:

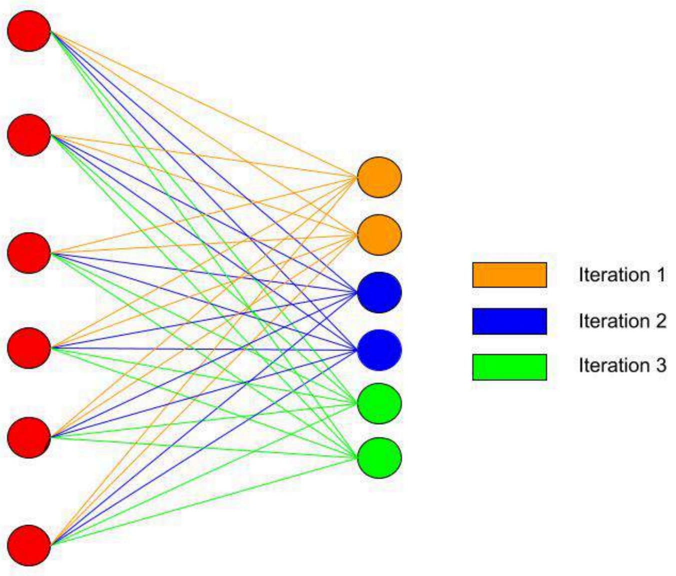


Figure 5.7: Computation of Output Nodes in Fully Connected layer after successive iterations

In the above figure, weights corresponding to neurons in orange are fetched first and the orange nodes are computed. The gradients are written back to DDR memory.

The process is repeated for blue and green nodes.

5.3.4 Convolution Layer IP

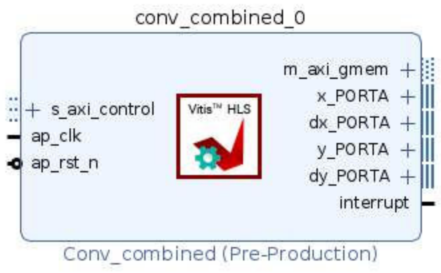


Figure 5.8: Convolutional Layer IP

- This IP serves to perform forward and backward propagation operations associated with a convolution layer. `x_PORTA` is connected to BRAM cell containing input activations. `dx_PORTA` is connected to BRAM cell containing outgoing gradients of layers below. `y_PORTA` is connected to BRAM cell containing output activations. `dy_PORTA` is connected to BRAM cell containing incoming gradients from higher layers.

- The `m_axi_gmem` port is connected to DDR through High Performance(HP) Slave Port on Zynq for fetching the weights. This port operates in Full AXI mode.
- The `s_axi_control` slave port is connected to General Purpose Master Port on Zynq. This port contains various registers. By writing to these registers, we can provide instructions to start/stop the operations, provide address for fetching data from memory.
- This IP contains local buffers for storing the fetched weights. These buffers are multidimensional arrays made of BRAM cells.
- The weights are fetched only once. They are fetched again only when they are updated in the DDR after a batch is processed. Whether or not to fetch weights is indicated by a flag on `s_axi_control` port.

5.3.5 Relu Activation IP

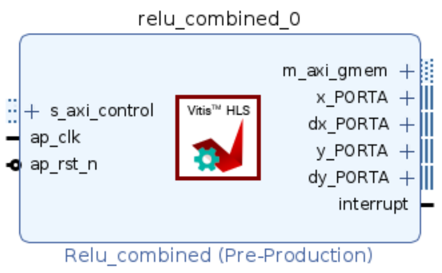


Figure 5.9: Relu Activation IP

- This IP is used to implement the ReLU Activation function.
- Since no weights are required, this IP doesn't have `m_axi_gmem` port to fetch data from DDR.
- `x_PORTA` is connected to BRAM cell containing input activations. `dx_PORTA` is connected to BRAM cell containing outgoing gradients of layers below. `y_PORTA` is connected to BRAM cell containing output activations. `dy_PORTA` is connected to BRAM cell containing incoming gradients from higher layers.

5.3.6 Loss Calculation IP

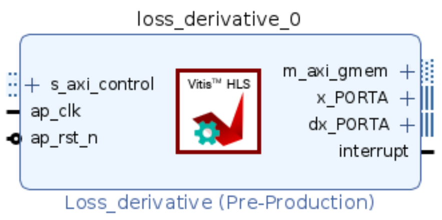


Figure 5.10: Loss Calculation IP

- This IP calculates the cross entropy loss associated with the inputs and labels provided.
- The HLS code uses `ap_fixed` library to perform various arithmetic operations such as logarithm, exponential, etc. This is available only for fixed point precision, not for floating point precision. In case of floating point precision, alternate ways need to be considered to do the arithmetic.
- It is connected to the last layer of our network and calculates gradients for the outermost layer.
- `x_PORTA` is connected to BRAM cell containing output logits for last layer. `dx_PORTA` is connected to BRAM cell where the calculated gradients are stored.
- The `m_axi_gmem` port is connected to DDR through High Performance(HP) Slave Port on Zynq. This is used for writing the output logits and gradients to DDR for debugging purposes. The user only has access to the data in DDR. We cannot directly read data from Block RAM.

5.3.7 Weight Update IP

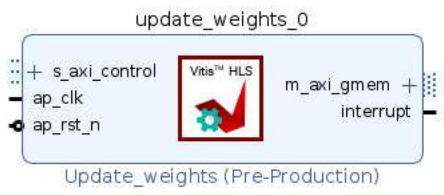


Figure 5.11: Weight Update IP

- Batch gradient optimizer is being used to train the model.
- During batch training, gradients are calculated and accumulated. After the batch is processed, this IP is used to update the weights using the gradients.
- This IP fetches the weights and their gradients from the DDR. The new weights are calculated and written back to the DDR.
- The `m_axi_gmem` port is connected to DDR through High Performance(HP) Slave Port on Zynq.
- The `s_axi_control` slave port is connected to General Purpose Master Port on Zynq. This port contains various registers. By writing to these registers, we can provide instructions to start/stop the operations, provide address for fetching data from memory.

5.4 Application 1: MNIST Classification

5.4.1 Block Diagram and Synthesis

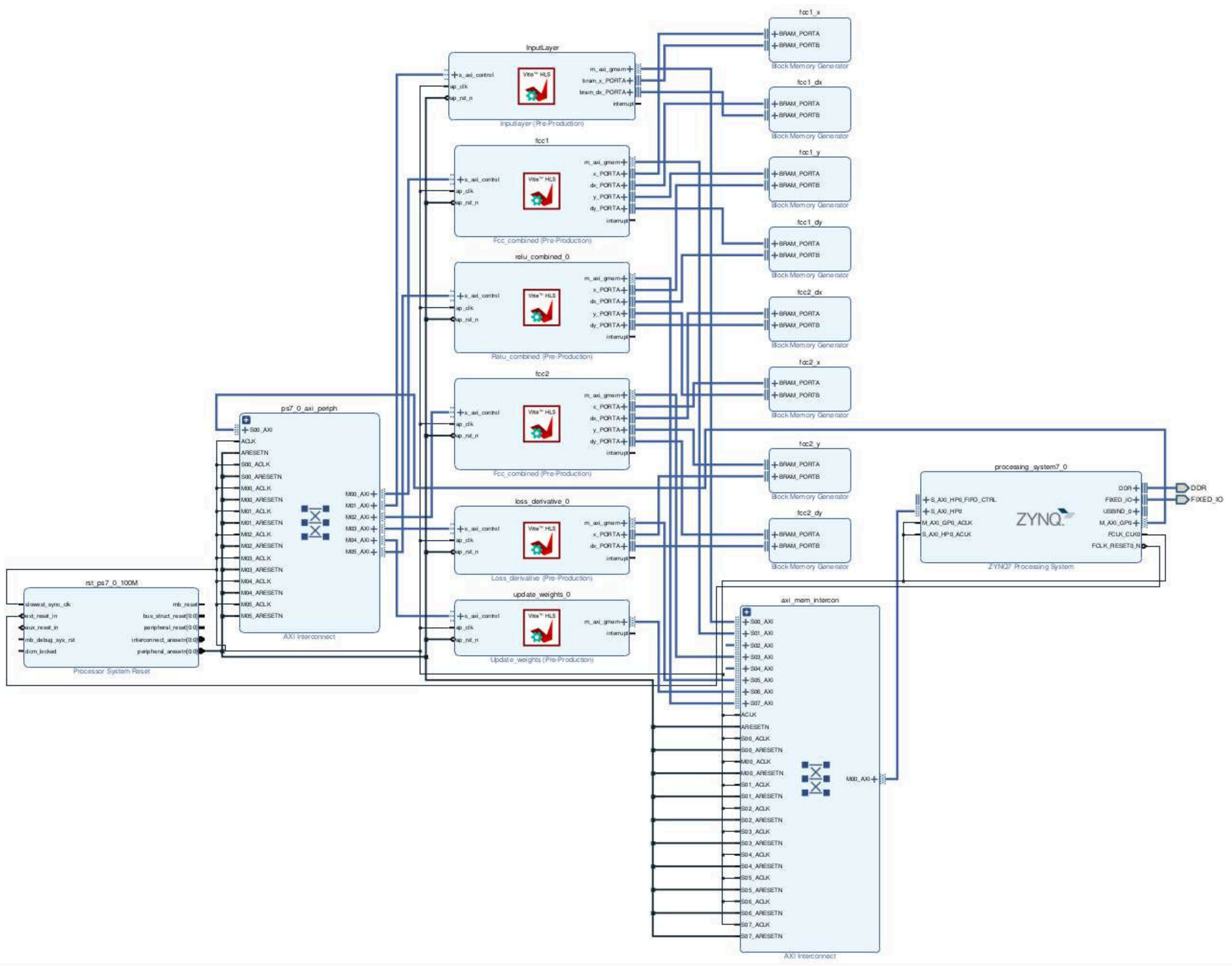


Figure 5.12: Block Diagram for MNIST classifier

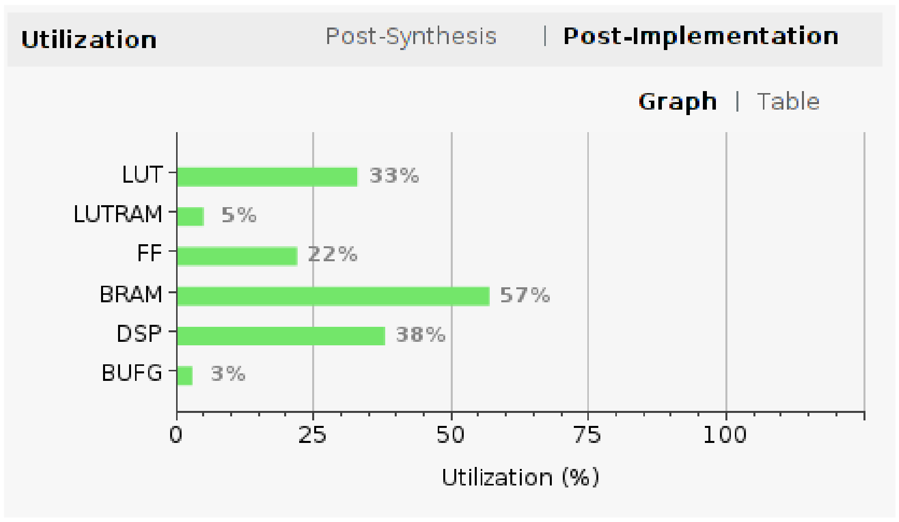


Figure 5.13: Utilization Report for Design Implementing MNIST Classifier

The above design was exported as a bitstream file after synthesis and implementation.

5.4.2 Results

For evaluation purposes, we measured the time taken for various steps involved in training process. A typical training loop involves the following stages:

- Forward Propagation
- Calculation of loss and gradients for last layer
- Backpropagation and calculating gradients for layer weights
- Updating weights after `batch_size` samples.

In the following we compare the time taken to execute various steps in a training loop for MNIST dataset. The batch size was chosen to be 16 and the learning rate was kept 0.001.

Step	CPU	Jetson-Nano	Raspberry Pi	FPGA
Training Loop Time	3.9ms	453ms	290ms	25.2ms
Forward Prop Time	0.9ms	33.9ms	28.8ms	1.6ms
Gradient Calculation	1.9ms	188.7ms	138ms	2.7ms
Loss Calculation	1.01ms	163.4ms	49ms	1.4ms
Weight Update	1.2ms	67.8ms	73.7ms	14.2ms
Power Consumption	50W	5-10W	4.5W	1.8W

Table 5.1: Timing Benchmarks for Single Training Loop

From the above table we can see that our FPGA implementation performs significantly faster than the other edge devices(Raspberry Pi and Jetson Nano). CPU performance is better than what we get on all edge devices, however at the cost of much higher power consumption and price.

5.4.3 Limitations

This implementation gives excellent results when compared to other edge platforms. However, this brings along some disadvantages as well.

- Only one IP is activate at a given time during operation. Hence, the design doesn't account for optimal usage of resources.
- Since we need a separate IP for every layer in the model, the design is not scalable. The FPGA would simply run out of resources after a certain number of layers.
- The design is not reusable for different models. If the model architecture changes, the hardware design needs to be modified as well.

5.5 Application 2: Classification Using Vibration Data

5.5.1 Block Diagram

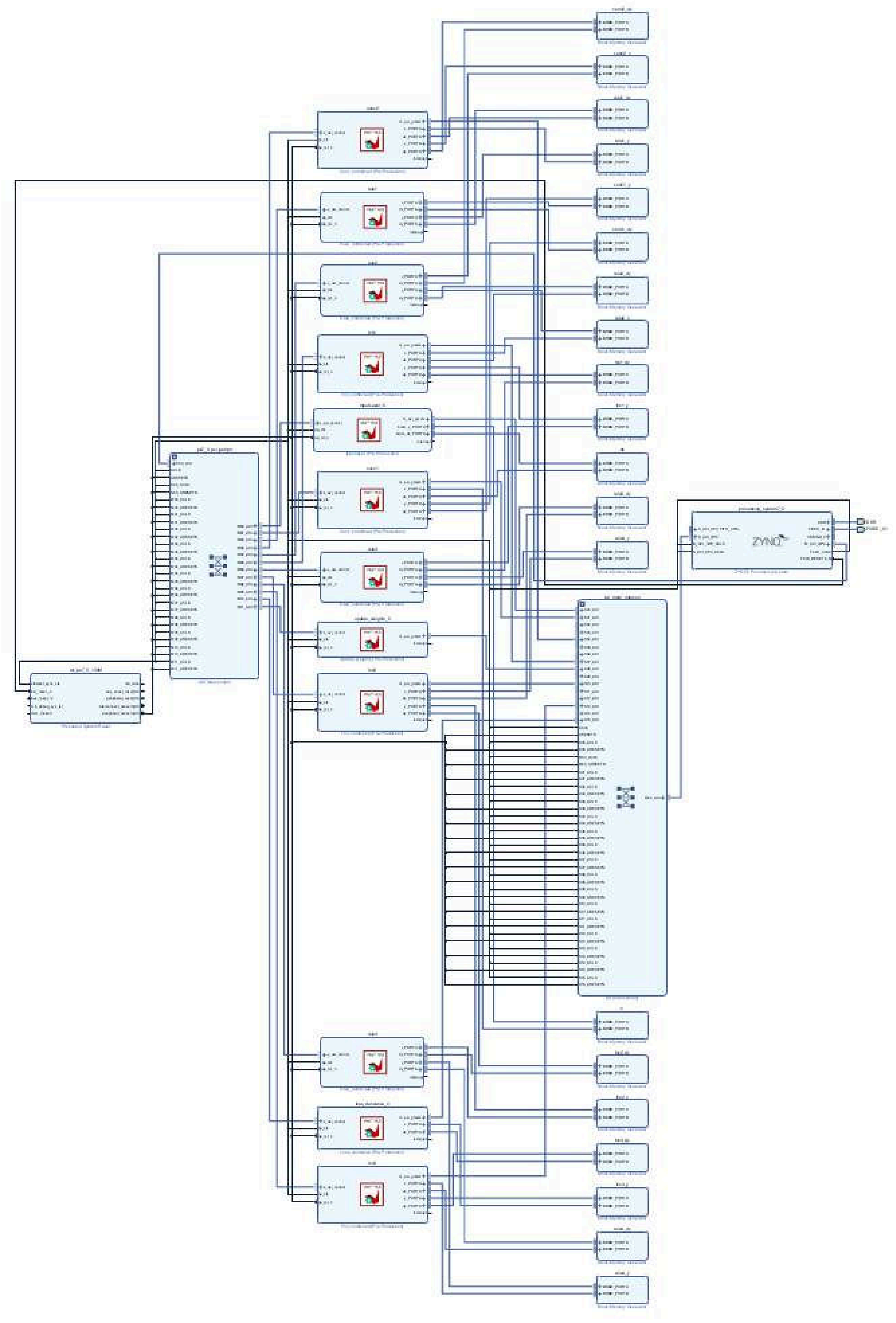


Figure 5.14: Block Diagram for Promethean classifier

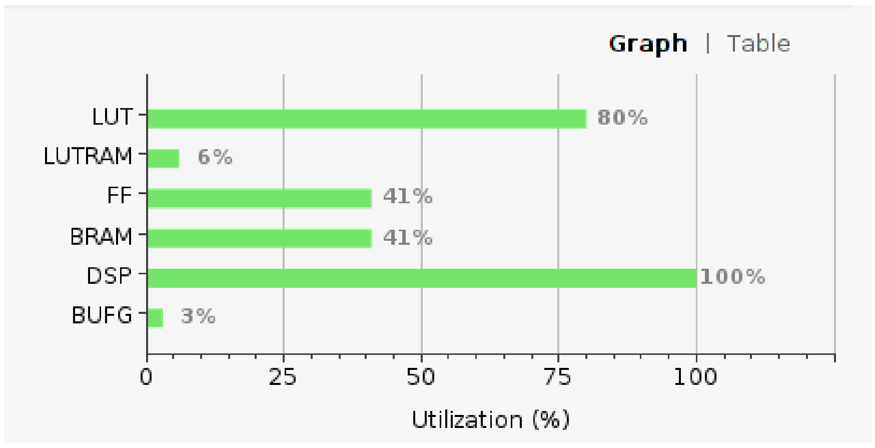


Figure 5.15: Utilization Report for Design Implementing Promethean Classifier

5.5.2 Discussion

From the above figure, we can see that the model utilizes all the DSP resources and most of the LUT.

Moreover, it took around 4 hours to synthesize the above design due to its complexity. There were several timing errors associated with its implementation.

Hence, we can conclude that this approach is not scalable for large models. An alternative approach has been introduced in the next chapter where these issues have been taken care of.

6. Software Layer- PYNQ

6.1 Introduction

PYNQ is an open-source project from Xilinx to make it easier to use Xilinx FPGA's.

It provides a Python interface to exploit the benefits of Programmable Logic, Processing System and the various peripherals.

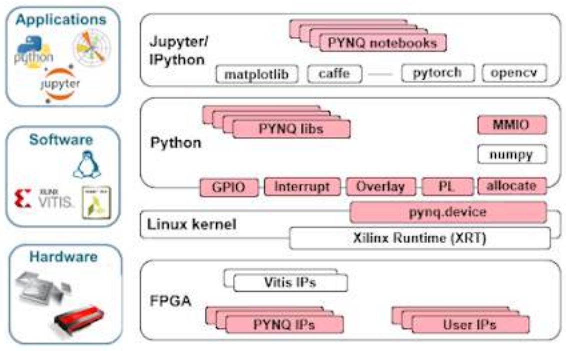


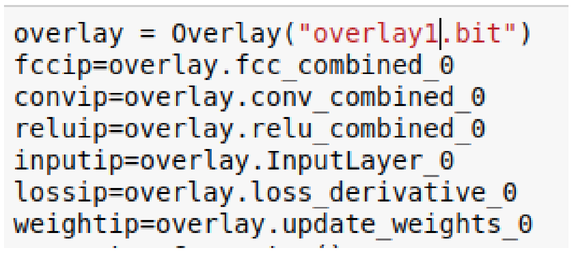
Figure 6.1: Overview of PYNQ framework

6.2 Setup

- First, a PYNQ image is downloaded and flashed to a micro SD card. Pre-built images for popular boards can be found online on the PYNQ website.
- The micro SD card is then inserted into the FPGA board. The board is set to boot from SD card and then powered on. This causes the linux kernel to boot up.
- The PYNQ is connected to local network via ethernet cable. We can then ssh into the board using another computer.
- By default, a jupyter notebook is also activated at port 9090. One can simply connect to `hhtp://ip_address:9090` to access the jupyter notebook server.
- The hardware bitstream file and Hardware Handoff file generated using Vivado needs to be placed in the same folder as the Jupyter Notebook.

6.3 PYNQ Library and Usage

The PYNQ drivers and code has been explained in this section.



```
overlay = Overlay("overlay1.bit")
fccip=overlay.fcc_combined_0
convip=overlay.conv_combined_0
reluip=overlay.relu_combined_0
inputip=overlay.InputLayer_0
lossip=overlay.loss_derivative_0
weightip=overlay.update_weights_0
```

Figure 6.2: Loading Overlay and IPs

6.3.1 Overlay

In the above figure, we can see that Overlay function has been used to load the hardware design of PL. It provides a Python interface to control the PL using Python running on PS.

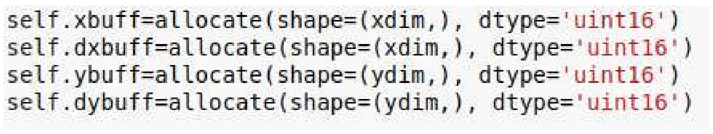
6.3.2 DefaultIP

We declare objects of the type DefaultIP using the overlay. Every object corresponds to an HLS IP in the block design.

Using these objects, we can modify the contents of registers on axi-lite interface of the IP. This can be used to start/stop the operations, passing memory address from where data needs to be fetched.

6.3.3 Allocate

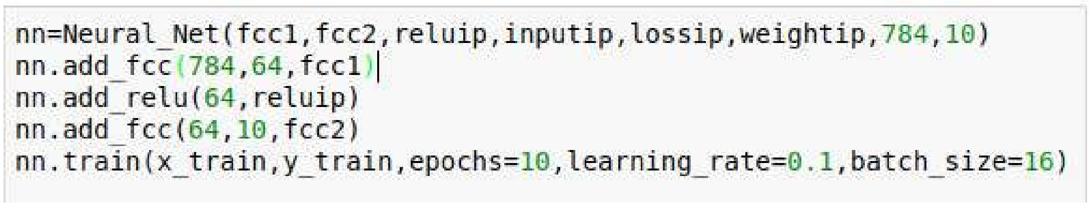
This function is used to allocate space inside the DDR. The physical memory address is then obtained which is required by HLS IPs to fetch data from DDR.



```
self.xbuff=allocate(shape=(xdim,), dtype='uint16')
self.dxbuff=allocate(shape=(xdim,), dtype='uint16')
self.ybuff=allocate(shape=(ydim,), dtype='uint16')
self.dybuff=allocate(shape=(ydim,), dtype='uint16')
```

Figure 6.3: Using allocate function

6.3.4 The Neural Network Class



```
nn=Neural_Net(fcc1,fcc2,reluip,inputip,lossip,weightip,784,10)
nn.add_fcc(784,64,fcc1)
nn.add_relu(64,reluip)
nn.add_fcc(64,10,fcc2)
nn.train(x_train,y_train,epochs=10,learning_rate=0.1,batch_size=16)
```

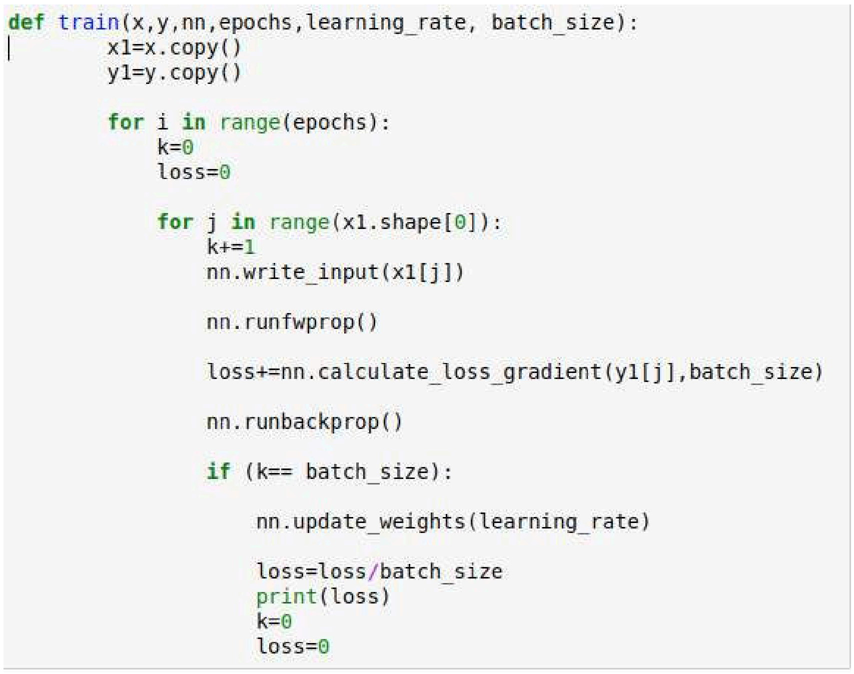
Figure 6.4: Using the Neural Network Class

The above figure demonstrates the usage of the Neural Network class created by us.

First, a `Neural_Net` object is created. The `InputLayer`, `LossIP` and `Update_Weight` IP objects are passed as arguments.

Then, layers and their dimensions are successively added as shown. The IPs responsible for performing their computations are also specified.

Finally, the train function is called with respective arguments to start the training process.



```python
def train(x,y,nn,epochs,learning_rate, batch_size):
        x1=x.copy()
        y1=y.copy()

        for i in range(epochs):
            k=0
            loss=0

            for j in range(x1.shape[0]):
                k+=1
                nn.write_input(x1[j])

                nn.runfwprop()

                loss+=nn.calculate_loss_gradient(y1[j],batch_size)

                nn.runbackprop()

                if (k== batch_size):

                    nn.update_weights(learning_rate)

                    loss=loss/batch_size
                    print(loss)
                    k=0
                    loss=0
```

Figure 6.5: Training Loop

7. Implementation 2

This approach aims to solve the issues encountered in the previous implementation. The overview of the design in this case is as follows:

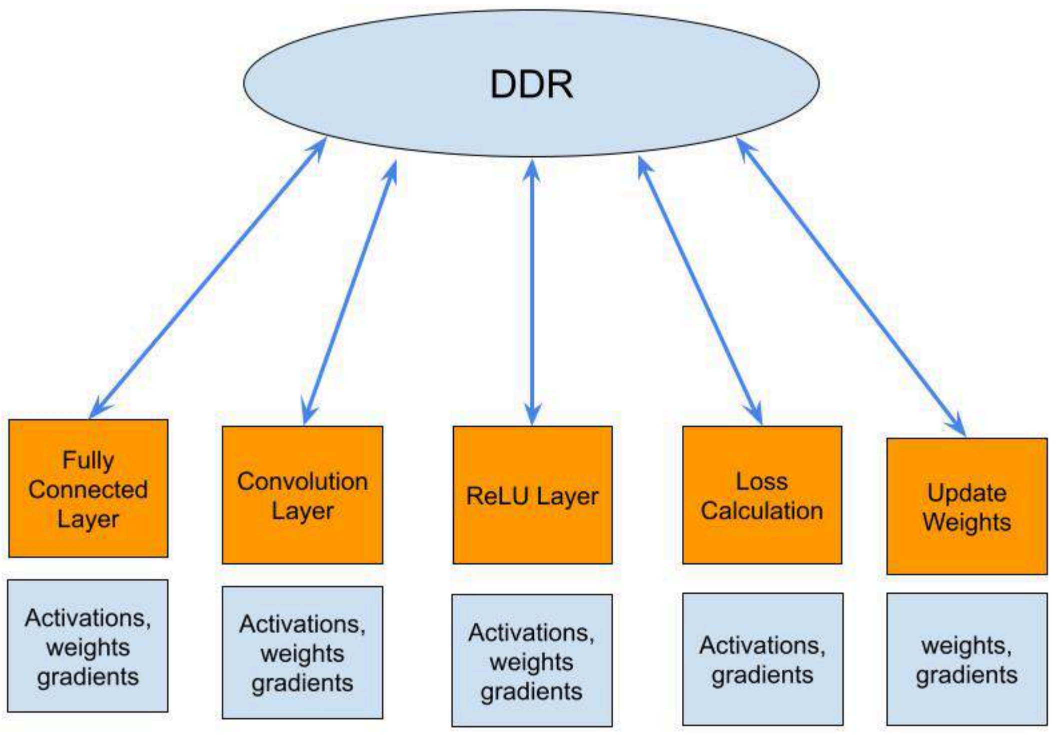


Figure 7.1: Flow Diagram for Implementation 2

7.1 Features

The features of this implementation are as follows:

- The weights and activations for the layers are stored in the DDR, unlike the previous implementation where the activations were stored only in BRAM.

- Whenever a layer operation needs to be performed, the corresponding weights, activations and gradients are fetched from DDR and stored in local Block RAM. After the results are computed, the modified values are written back to the DDR.
- The new weights are fetched only when update operation has taken place. Else, the weight fetch operation is not carried out as it would be redundant.
- The design contains an IP for every type of layer. This IP is used every time an operation needs to be performed for that type of layer.
- Since the amount of Block RAM resource inside every IP is fixed, it limits the maximum dimension of input and output vector supported by the IP. The maximum dimensions supported are enlisted as follows:

7.2 IPs Used

The IPs used in this design are described as follows:

7.2.1 Fully Connected layer IP

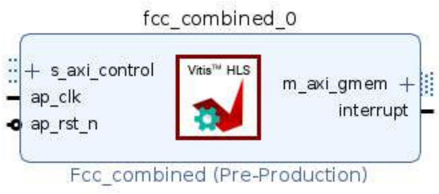


Figure 7.2: Fully Connected Layer IP

- This IP is used to perform operations related to a Dense layer
- The `m_axi_gmem` port is connected to DDR through High Performance(HP) Slave Port on Zynq for fetching the weights, activations and gradients. This port operates in Full AXI mode.
- The `s_axi_control` slave port is connected to General Purpose Master Port on Zynq. This port contains various registers. By writing to these registers, we can provide instructions to start/stop the operations, provide address for fetching data from memory.
- Much of the implementation and optimizations in this implementation for Dense layer is similar to previous implementation.

7.2.2 Convolution Layer IP

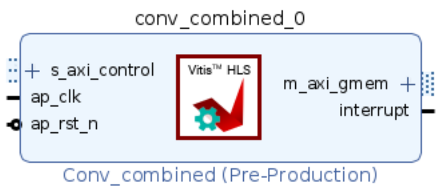


Figure 7.3: Convolutional Layer IP

- This IP serves to perform forward and backward propagation operations associated with a convolution layer.
- The `m_axi_gmem` port is connected to DDR through High Performance(HP) Slave Port on Zynq for fetching the weights, activations and gradients. This port operates in Full AXI mode.
- The `s_axi_control` slave port is connected to General Purpose Master Port on Zynq. This port contains various registers. By writing to these registers, we can provide instructions to start/stop the operations, provide address for fetching data from memory.
- This IP contains local buffers for storing the fetched weights, activations and gradients. These buffers are multidimensional arrays made of BRAM cells.

7.2.3 Relu Activation IP

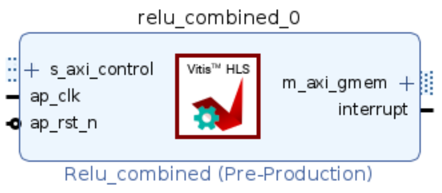


Figure 7.4: Relu Activation IP

- This IP is used to implement the ReLU Activation function.
- The `m_axi_gmem` port is connected to DDR through High Performance(HP) Slave Port on Zynq for fetching the activations and gradients. This port operates in Full AXI mode.

7.2.4 Loss Calculation IP

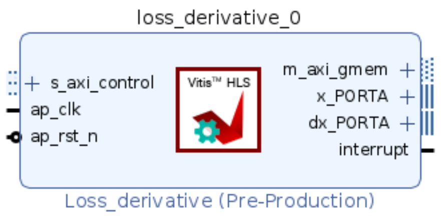


Figure 7.5: Loss Calculation IP

- This IP calculates the cross entropy loss associated with the inputs and labels provided.
- The HLS code uses `ap_fixed` library to perform various arithmetic operations such as logarithm, exponential, etc. This is available only for fixed point precision, not for floating point precision. In case of floating point precision, alternate ways need to be considered to do the arithmetic.
- It is connected to the last layer of our network and calculates gradients for the outermost layer.
- The `m_axi_gmem` port is connected to DDR through High Performance(HP) Slave Port on Zynq. This is used for writing the output logits and gradients to DDR for debugging purposes. The user only has access to the data in DDR. We cannot directly read data from Block RAM.

7.2.5 Weight Update IP

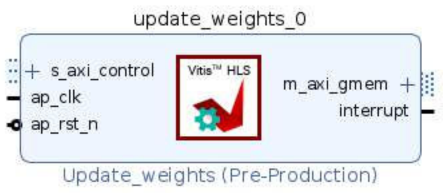


Figure 7.6: Weight Update IP

- Batch gradient optimizer is being used to train the model.
- During batch training, gradients are calculated and accumulated. After the batch is processed, this IP is used to update the weights using the gradients.
- This IP fetches the weights and their gradients from the DDR. The new weights are calculated and written back to the DDR.
- The `m_axi_gmem` port is connected to DDR through High Performance(HP) Slave Port on Zynq.

- The `s_axi_control` slave port is connected to General Purpose Master Port on Zynq. This port contains various registers. By writing to these registers, we can provide instructions to start/stop the operations, provide address for fetching data from memory.

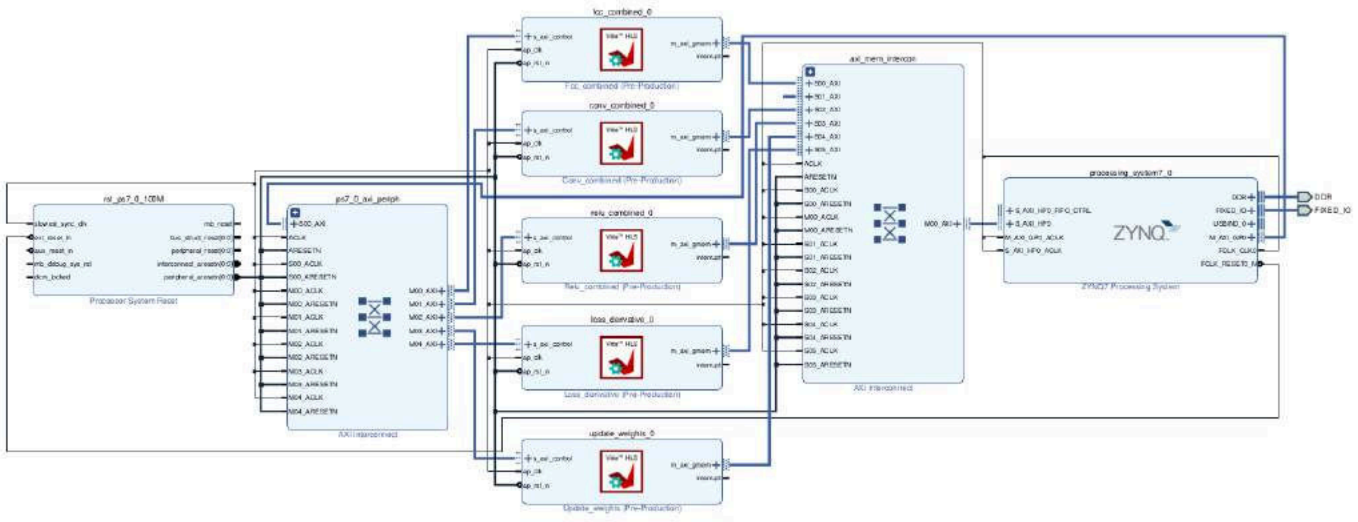


Figure 7.7: Block Design for Implementation 2

This implementation overcomes the following drawbacks from previous design:

- It is scalable. Since a particular IP is reused for a given type of layer, we don't need a separate IP for every layer in our model, unlike the first implementation.
- The hardware is reusable. We don't need to modify the hardware for different models as the same design can be used for all of them. This saves alot of development time.

This design has the following drawbacks:

- Since the activations need to be fetched from DDR every time, it makes the execution slower in comparison to earlier implementation.
- Even in this case, only one IP is being used at a time. Hence, the hardware resources are not being utilized in the most optimum manner. This can be solved by pipelining the flow.

7.3 Application 1: Classification of Vibration Data

Due to scalability, we were able to implement the model to detect faults using vibration data which had not been possible in the earlier case. The hardware design is as shown in the previous figure.

The timing results are as follows:

Step	CPU	Jetson-Nano	Raspberry Pi	FPGA
Training Loop Time	3.9ms	57ms	401ms	25.2ms
Forward Prop Time	0.9ms	15.43ms	109.9ms	1.6ms
Gradient Calculation	1.9ms	19.03ms	149.5ms	2.7ms
Loss Calculation	1.01ms	9.08ms	62.1ms	0.89ms
Weight Update	1.2ms	12.96ms	79.38ms	7.22ms
Power Consumption	50W	5-10W	4-5W	1.8W

Table 7.1: Timing Benchmarks for Single Training Loop

7.4 Application 2: MNIST Dataset

In the following table, we compare the timing reports for both the implementations on MNIST dataset:

Step	CPU	Jetson-Nano	Raspberry Pi	Impl. 1	Impl. 2
Training Loop Time	3.9ms	453ms	290ms	25.2ms	97ms
Forward Prop Time	0.9ms	33.9ms	28.8ms	1.6ms	4.13ms
Gradient Calculation	1.9ms	188.7ms	138ms	2.7ms	4.4ms
Loss Calculation	1.01ms	163.4ms	49ms	1.4ms	1.1ms
Weight Update	1.2ms	67.8ms	73.7ms	14.2ms	69.4ms
Power Consumption	50W	5-10W	4.5W	1.8W	1.8W

Table 7.2: Timing Benchmarks for Single Training Loop

7.5 Discussion

From the table in previous section, we can see that second Implementation is slower than the first one.

This is expected because of the fact that weights and activations are now fetched from DDR everytime an operation needs to be performed. In the first approach, the activations were fetch from Block RAM. The weights were fetched from DDR only after batch update took place. In the second approach, both weights and activations are fetched from DDR memory in every iteration.

The first approach is a good option where the model is small and low latency and high speed is a priority.

For larger models, the second approach is more appropriate as it is scalable.

8. Concluding Remarks

8.1 Conclusion

In this project, we highlighted the importance of processing and training closer to the sensor nodes, the edge. We have raised issues which can arise while depending on a central server for much of the processing. An FPGA-based framework was developed capable of learning weights in a multi-layer perceptron. The results show that the our framework performs considerably better compared to other edge-based platforms commonly available. The application has been demonstrated in a system that detects faults in a machine using vibration data collected from a sensor. The performance for this application has been benchmarked on our system and compared with other platforms.

8.2 Future Work

The following optimizations can be implemented to further improve the performance of our implementation:

- **Pipelining:** A major shortcoming in both implementations is that only one resource is being utilized at a time. This can be overcome by pipelining the various stages of our neural network. This way, more IPs can be functional at a given time, hence achieving better parallelism and improving throughput.
- **Distributed Learning:** Multiple edge servers can interact and distribute workload among idle devices to achieve low-latency and reliability.

9. References

1. Edge Intelligence: Architectures, Challenges, and Applications,Xu, Dianlei and Li, Tong and Li, Yong and Su, Xiang and Tarkoma, Sasu and Jiang, Tao and Crowcroft, https://arxiv.org/abs/2003.12172,

2. S. Samarakoon, M. Bennis, W. Saad and M. Debbah, ”Distributed Federated Learning for Ultra-Reliable Low-Latency Vehicular Communications,” in IEEE Transactions on Communications, vol. 68, no. 2, pp. 1146-1159, Feb. 2020, doi: 10.1109/TCOMM.2019.2956472.

3. http://www.pynq.io/

10. Appendix

10.1 Classification in Genomics

One of the key issues with genomics is sequence classification. The size of NCBI RefSeq bacterial database is growing exponentially every year as sequencing techniques are becoming cheaper.

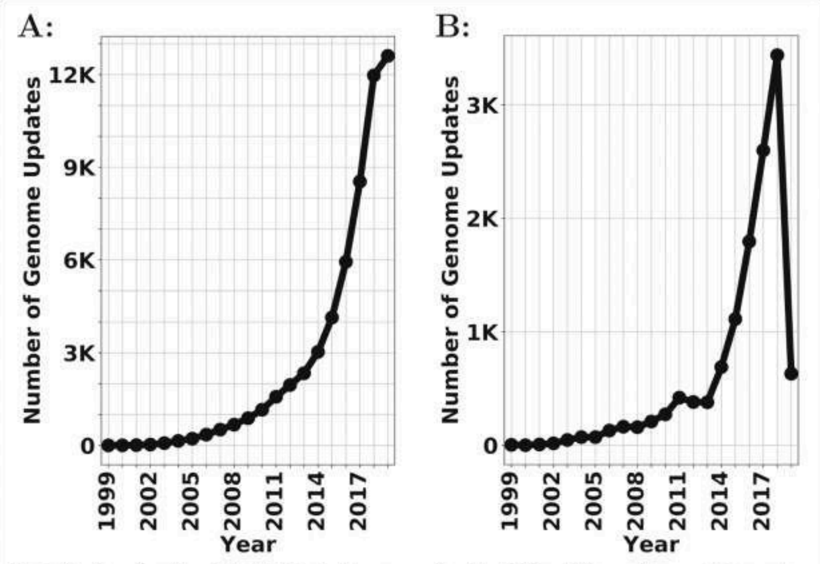


Fig. 1 Number of updates in the NCBI bacterial genomes from the RefSeq database **a** Accumulative number of genome updates per year; **b** compared with previous year, the number of new updates per year

Figure 10.1: Annual Updates to NCBI Refseq Database

Sequence classification is an important data mining task in genomics. It can be used to identify the organism by observing a DNA sequence from conservative region of a particular gene.

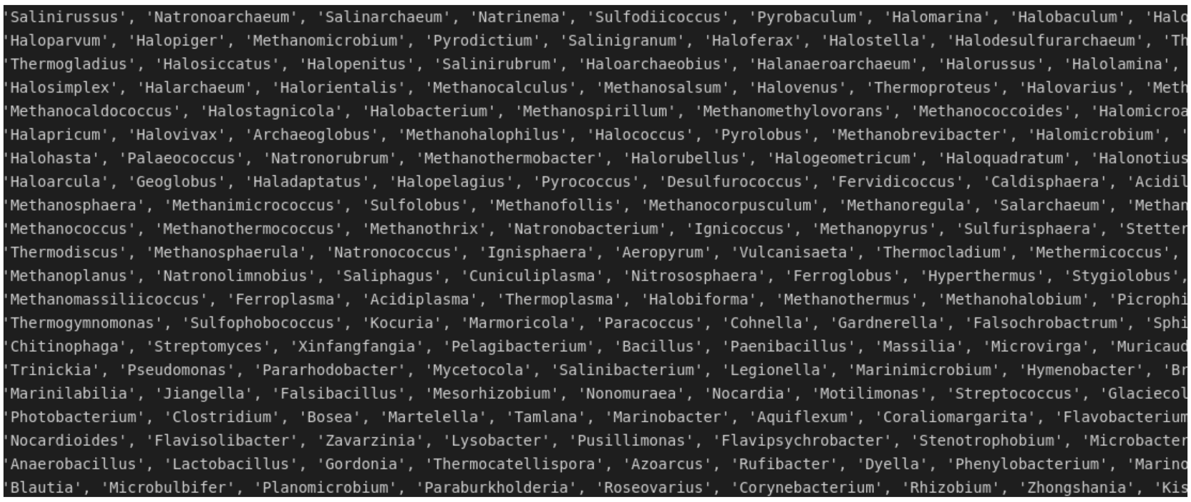
Sequence Classification presents the following challenges:

- High dimensionality of input.
- The same model achitecture should be able to generalize to a wide variety of data.
- One should be able to modify the architecture to introduce additional classes. Since retraining the model is an expensive task, it is desirable that the new model should be able to adapt to additional classes without having to retrain from scratch. Information from old model should be preserved.

10.2 The Dataset

We are using curated NCBI- 16s-18s RefSeq Ribosomal dataset for classification task.
The characteristics of this dataset are as follows:

- The dataset is stored in a fasta file.
- The file contains 24855 sequences, each row representing a data entry.
- Every row contains a sequence ID, description and the sequence itself. The information about the organism is in the sequence description.
- The target is to classify the organism into its species/genus. This information is available in the sequence description.
- There are a total of 4283 Genus and 18825 Species in the dataset.
- There is not enough data to classify the sequences into the species. Hence, we try to classify a sequence into its genus.



```
'Salinirussus', 'Natronoarchaeum', 'Salinarchaeum', 'Natrinema', 'Sulfodiicoccus', 'Pyrobaculum', 'Halomarina', 'Halobaculum', 'Halo
'Haloparvum', 'Halopiger', 'Methanomicrobium', 'Pyrodictium', 'Salinigranum', 'Haloferax', 'Halostella', 'Halodesulfurarchaeum', 'Th
'Thermogladius', 'Halosiccatus', 'Halopenitus', 'Salinirubrum', 'Haloarchaeobius', 'Halanaeroarchaeum', 'Halorussus', 'Halolamina',
'Halosimplex', 'Halarchaeum', 'Halorientalis', 'Methanocalculus', 'Methanosalsum', 'Halovenus', 'Thermoproteus', 'Halovarius', 'Meth
'Methanocaldococcus', 'Halostagnicola', 'Halobacterium', 'Methanospirillum', 'Methanomethylovorans', 'Methanococcoides', 'Halomicroa
'Halapricum', 'Halovivax', 'Archaeoglobus', 'Methanohalophilus', 'Halococcus', 'Pyrolobus', 'Methanobrevibacter', 'Halomicrobium', '
'Halohasta', 'Palaeococcus', 'Natronorubrum', 'Methanothermobacter', 'Halorubellus', 'Halogeometricum', 'Haloquadratum', 'Halonotius
'Haloarcula', 'Geoglobus', 'Haladaptatus', 'Halopelagius', 'Pyrococcus', 'Desulfurococcus', 'Fervidicoccus', 'Caldisphaera', 'Acidil
'Methanosphaera', 'Methanimicrococcus', 'Sulfolobus', 'Methanofollis', 'Methanocorpusculum', 'Methanoregula', 'Salarchaeum', 'Methan
'Methanococcus', 'Methanothermococcus', 'Methanothrix', 'Natronobacterium', 'Ignicoccus', 'Methanopyrus', 'Sulfurisphaera', 'Stetter
'Thermodiscus', 'Methanosphaerula', 'Natronococcus', 'Ignisphaera', 'Aeropyrum', 'Vulcanisaeta', 'Thermocladium', 'Methermicoccus',
'Methanoplanus', 'Natronolimnobius', 'Saliphagus', 'Cuniculiplasma', 'Nitrososphaera', 'Ferroglobus', 'Hyperthermus', 'Stygiolobus',
'Methanomassiliicoccus', 'Ferroplasma', 'Acidiplasma', 'Thermoplasma', 'Halobiforma', 'Methanothermus', 'Methanohalobium', 'Picrophi
'Thermogymnomonas', 'Sulfophobococcus', 'Kocuria', 'Marmoricola', 'Paracoccus', 'Cohnella', 'Gardnerella', 'Falsochrobactrum', 'Sphi
'Chitinophaga', 'Streptomyces', 'Xinfangfangia', 'Pelagibacterium', 'Bacillus', 'Paenibacillus', 'Massilia', 'Microvirga', 'Muricaud
'Trinickia', 'Pseudomonas', 'Pararhodobacter', 'Mycetocola', 'Salinibacterium', 'Legionella', 'Marinimicrobium', 'Hymenobacter', 'Br
'Marinilabilia', 'Jiangella', 'Falsibacillus', 'Mesorhizobium', 'Nonomuraea', 'Nocardia', 'Motilimonas', 'Streptococcus', 'Glaciecol
'Photobacterium', 'Clostridium', 'Bosea', 'Martelella', 'Tamlana', 'Marinobacter', 'Aquiflexum', 'Coraliomargarita', 'Flavobacterium
'Nocardioides', 'Flavisolibacter', 'Zavarzinia', 'Lysobacter', 'Pusillimonas', 'Flavipsychrobacter', 'Stenotrophobium', 'Microbacter
'Anaerobacillus', 'Lactobacillus', 'Gordonia', 'Thermocatellispora', 'Azoarcus', 'Rufibacter', 'Dyella', 'Phenylobacterium', 'Marino
'Blautia', 'Microbulbifer', 'Planomicrobium', 'Paraburkholderia', 'Roseovarius', 'Corynebacterium', 'Rhizobium', 'Zhongshania', 'Kis
```

Figure 10.2: Genus Present in the dataset

10.3 Encoding the Sequence

The DNA sequence is a string of letters. In order to perform mathematical analysis on the same, we need to convert it into a mathematical representation. We have the following ways to convert a DNA sequence into mathematical representation:

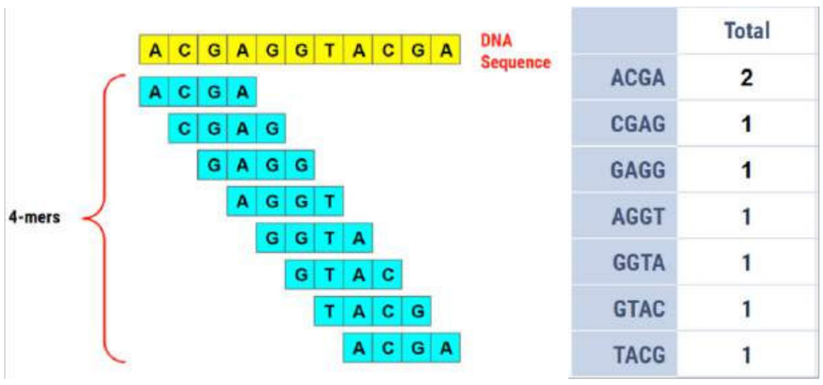
- One hot encoding based
- Kmer based

10.3.1 One Hot Encoding Based

In this method, every base is converted into a one-hot encoded vector of size 4. The entire sequence is hence encoded as a two-dimensional vector.

The disadvantage is that this creates a very high dimensional input matrix.

10.3.2 Kmer Based



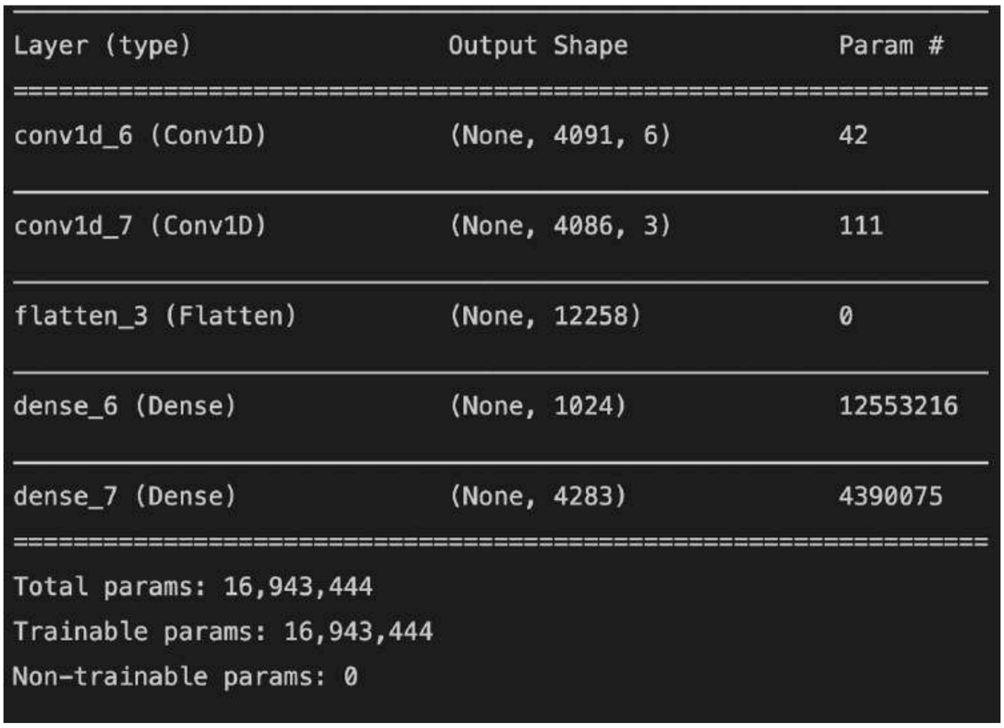
	Total
ACGA	2
CGAG	1
GAGG	1
AGGT	1
GGTA	1
GTAC	1
TACG	1

Figure 10.3: Kmer Approach for Classification

- A window of size k is slid from left to right across the DNA sequence, shifting one base pair at a time.
- We maintain a table recording how many times we have encountered every a kmer in the process. For example, if k = 4, we have a table with $4^4 = 256$ rows.
- The entries in this table are considered as mathematical representation of the DNA sequence.

10.4 Classification Model

We have designed a convolutional neural network for classifying the input sequence into various genus.



```
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_6 (Conv1D)            (None, 4091, 6)           42
_________________________________________________________________
conv1d_7 (Conv1D)            (None, 4086, 3)           111
_________________________________________________________________
flatten_3 (Flatten)          (None, 12258)             0
_________________________________________________________________
dense_6 (Dense)              (None, 1024)              12553216
_________________________________________________________________
dense_7 (Dense)              (None, 4283)              4390075
=================================================================
Total params: 16,943,444
Trainable params: 16,943,444
Non-trainable params: 0
```

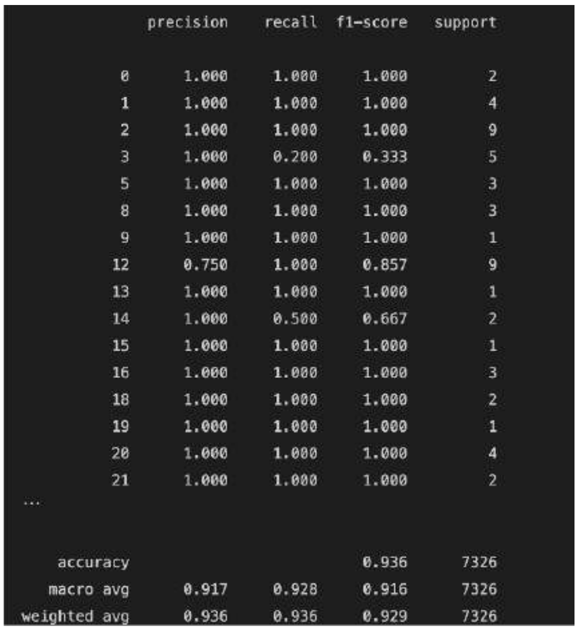
Figure 10.4: Tensorflow model for classification

The model contains convolutional layers followed by dense layers, each with ReLU activation function. There are 16.9 Million parameters, majority of which lie in dense layers.

We are considering k=6, due to which the input is of size $4^6 = 4096$. We are classifying the data into the respective genus. After analyzing the dataset, we discover that there are total 4283 genus present. Hence, the last fully connected layer has 4283 nodes.

10.5 Results

The training process was run for 10 epochs with a batch size of 32. We obtained decent accuracy of 93%.



```
              precision    recall  f1-score   support

           0      1.000     1.000     1.000         2
           1      1.000     1.000     1.000         4
           2      1.000     1.000     1.000         9
           3      1.000     0.200     0.333         5
           5      1.000     1.000     1.000         3
           8      1.000     1.000     1.000         3
           9      1.000     1.000     1.000         1
          12      0.750     1.000     0.857         9
          13      1.000     1.000     1.000         1
          14      1.000     0.500     0.667         2
          15      1.000     1.000     1.000         1
          16      1.000     1.000     1.000         3
          18      1.000     1.000     1.000         2
          19      1.000     1.000     1.000         1
          20      1.000     1.000     1.000         4
          21      1.000     1.000     1.000         2
...

    accuracy                          0.936      7326
   macro avg      0.917     0.928     0.916      7326
weighted avg      0.936     0.936     0.929      7326
```

Figure 10.5: Classification Results

F1-score represents the ability of a classifier to distinguish between the different classes. The value 1 represents perfect ability of the classifier to classify an input into the different classes.

We get F1-score 0.9 which is a good value.

10.6 Class Incremental Learning

Since the classifier networks can be large, containing large number of parameters in the order of millions, it takes significant resources and time to train these networks.

As we have seen in the previous section, the amount of data in NCBI database is increasing exponentially every day. This also introduces new species and genus which we may want to classify.

In order to add new classes to an existing neural network, the nodes in the final dense layer are increased. It is desirable that the previously learnt information is retained, so that the new network need not be trained from scratch.

After updating the model and training for the new classes, it is very much possible that previously learnt information is lost. This phenomenon is known as **catastrophic forgetting**.

Under class-incremental learning, the objective is to learn for newly added classes without catastrophic forgetting. There are various approaches to do so. Some of them are as follows:

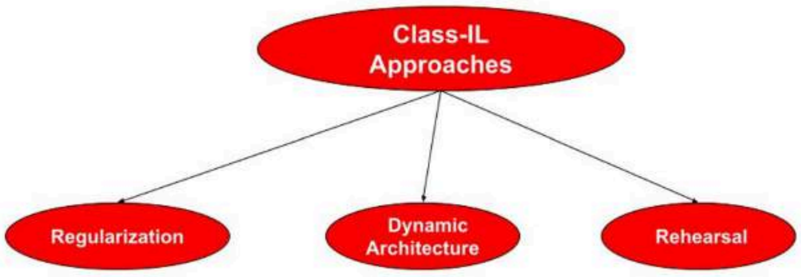


Figure 10.6: Approaches to Class Incremental Learning

We will be discussing these approaches one by one in the following sections.

10.7 Regularization

This class of approach focuses on preventing weight drift relevant to previously learnt tasks. We estimate the prior importance of each parameter to penalize changes to them. Following regularization loss is introduced in addition to cross entropy loss:

$$L_{reg}(\theta^t) = \frac{1}{2} \sum_{i=1}^{|\theta^{t-1}|} \Omega_i ({\theta_i}^{t-1} - {\theta_i}^{t})^2 \tag{10.1}$$

Here, ${\theta_i}^t$ is the weight i of the network currently being trained on task t. Ω_i contains the importance for each network weight.

The values of Ω_i can be calculated using Elastic Weight Consolidation(EWC) in which Ω_i is calculated as a diagonal approximation of Fisher Information Matrix.

10.8 Rehearsal Based

In this approach, a small number of exemplars from previous tasks are preserved to prevent forgetting of previous tasks. Incremental Classifier and Representation Learning (iCaRL) first proposed this type of approach.

10.9 Incremental Learning in Genomics

We have used Rehearsal-based approach for class-incremental learning in Genomics.

Since the architecture is dynamic, it is better suited to use PyTorch framework instead of Tensorflow. We have demonstrated Class Incremental Learning on NCBI RefSeq database.

The results are as follows:

Number of Classes	**Training Accuracy**	**Test Accuracy**
4	99.79	97.3
8	97.33	74.82
10	99.7	96.5
14	100	97.46
18	99.54	97.84
22	99.57	95.22
26	99.43	97.65
30	99.62	98.5

Table 10.1: Accuracy