Plant Diseases Classifier using FPGA-Accelerated CNN

ECNG 4981: Senior Project II
Fall 2020

Supervised by:
Dr. Yehea Ismail
Dr. Karim Seddik

Teaching Assistant: Ahmed M. Ibrahim

Hana Asal
Mahmoud Shamaa
Mostafa ElTobgy
Nour Mostafa
Osama El Farnawani
Salma Afifi

Abstract

Plant diseases are a crucial issue that threatens the field of agriculture. To facilitate the identification process as opposed to traditional methods, this thesis project developed a plant diseases classifier that implements deep learning on FPGA. Normally, embedded CNN applications, like classification, are applied using CPUs or GPUs, however, this results in reduced performance and power efficiency. Hence, and due to advancements in high level synthesis (HLS) tools and PYNQ development boards, the PYNQ-Z1 was used in our project. A 9-layer CNN was trained on the New Plant Diseases dataset and tested using Theano Framework, and an accuracy of 95.14% was achieved. The weights were then obtained and the RTL for the CNN was generated using HLS. Vivado HLX was then used for creating the block design and its wrapper, synthesis, generating the bitstream, and finally programming the FPGA. Communication between the Processing System (PS) and Programmable Logic (PL) on the board was mainly carried out through AXI4 and DMA protocols. This paper presents the overall design for the software and hardware implementations along with the general flow needed for using the hardware through inputting a leaf image from a webcam and through the HDMI cable to the board and finally, receiving a prediction indicating whether the plant is diseased or healthy.

Acronyms and Abbreviations

AMBA Arm Advanced Microcontroller Bus Architecture

AD Alzheimer's Disease

ANN Artificial Neural Network

ARM Advanced RISC Machines

AXI Advanced eXtensible Interface

CAN CSMA-CD/ASM protocol

CEC Consumer Electronics Control

CNN Convolutional Neural Network

CSMA/CD Carrier-sense Multiple Access with Collision Detection

CPU Central Processing Unit

CT Computed Tomography

DDC Direct Digital Controls

DDR Double Data Rate

DFE Design For Environment

DMA Direct Memory Access

DRAM Dynamic Random Access Memory

DSP Digital Signal Processing

FP32 Floating Point (32 bit)

FPGAs Field Programmable Gate Array

GPU Graphics Processing Unit

HDL Hardware description language

HDMI High Definition Multimedia Interface

HLS High Level Synthesis

I2C Inter-Integrated Circuit

IP Intellectual Property

LUTs Lookup Tables

ML Machine Learning

MMIO Memory Mapped Input/Output

NN Neural Network

OS Operating System

PHY Physical Layer

PL Programmable Logic

PS Processing System

RAM Random Access Memory

ReLU Rectified Linear Unit

RGB Red-Green-Blue

RNN Recurrent Neural Network

RTL Register Transfer Language

SDIO Secure Digital Input Output

SoC System on Chip

SPI Serial Peripheral Interface

TF TensorFlow

TMDS Transition-minimized Differential Signaling (TMDS)

UART Universal Asynchronous Receiver/Transmitter

USB Universal Serial Bus

WSI Whole Slide Tissue Images

YOLO You Only Look Once

List of Figures

Table of Contents

Introduction

World population growth is estimated to reach 9.7 billion by 2050 and 11.2 billion by 2100[16]. With such massive growth rates, demand for food production will subsequently rise greatly. Hence, it is becoming more crucial for technological advancements in the field of agriculture. As plant diseases impose a great threat and reduce greatly the production rates of food, an automated plant disease classifier using AI algorithms would facilitate greatly the process, save time and result in much better accuracy instead of traditional manual classifications.

Artificial Intelligence (AI) is a huge umbrella under which Deep Learning belongs. It is the study of the ability to create machines and computers that are able to perform and do tasks the same way as the humans would do. Since Deep learning is a subset of ML, it is used to automate and simplify the analysis of the inputted data to human beings. This is done by evaluating the data and fitting it into simpler models that can be easily understood. Even though Deep learning is considered to be one of the major computer fields nowadays, it operates differently from the conventional methods of problem solving using direct instructions in a code. Thus, automated decision making is being applied on the inputted data in deep learning.

Deep Learning is a subset of Machine Learning; both are the learning processes of teaching the computer to perform specific tasks for a certain application. The process includes a set of techniques, algorithms and mathematical models to teach the computers. Main objective of deep learning is to mimic how the brain operates; whereas the terminologies used are identical to the neurological ones. The basic building block in a NN is the Neuron, like the human nervous system, neurons here interact together in the form of layers. Similar to ML, deep learning also

can be supervised and unsupervised. Deep learning also outperforms humans in some basic cognitive tasks, one popular task is identifying objects that are not clear in a picture. Moreover, deep learning is dominating the video processing field, a prime example is Deep Fake technology; where videos of people are recreated with others' faces instead of theirs. Currently, deep learning algorithms are overshadowing other Computer Vision approaches like classic machine learning ones; which is why a lot of companies are shifting their attention to deep learning like Google, Facebook, Amazon and Tesla. For all the reasons mentioned above, deep learning was used in our project to efficiently classify whether a plant is diseased or not.

For our application, the"New- Plant diseases" dataset is used to train our Convolutional Neural Network. The CNN is trained to correctly classify the diseased and healthy plant leaves according to the input images. Since FPGA is the target hardware fabric that is used in our project along with the PS, PYNQ-Z1 is the perfect fit for our implementation. It helps designers exploit the benefits of PL and microprocessors to produce powerful designs and applications.

In the following sections, the structure of our project will be explained. Starting with the Literature Review, discussing firstly deep learning in more details, followed by popular frameworks used for implementation. Then, different applications of CNNs are discussed in the fields of biomedical, recycling and agriculture. An overview on the overall design of our implementation will be discussed starting with the HLS compiler to the FPGA deployment. Following this, the design of the individual system modules encompassing the hardware and the software aspects of our project will be explained in detail. Then, the hardware components that are used in our project will be listed along with their specifications and their functionality in our implementation. The standards involved in this project will be elaborated as well as the

validation techniques used to test the accuracy of our design. After considering the technical part of our design implementation, we will briefly discuss some societal and commercial aspects including: Feasibility and Economics, Societal and Environmental, Safety and Ethical Considerations. Lastly, we will end this paper by providing some recommendations for design improvement and future work.

.

Literature Review

Deep Learning and Neural Networks

Deep Learning

As previously mentioned, deep learning is a subgroup of machine learning; they are both the learning process of the computers to perform specific tasks. The history of this may go back to 1943 when a computer model was created by Walter Pitts and Warren McCulloch which was based on the neural networks of the human brain [1]. A set of techniques, algorithms and mathematical models are used for this learning process. At the beginning, the algorithm is fed a set of examples of a certain task in which the machine learning model should then be able to perform on its own. After it is predicted that the model has been trained and collected enough data, the model should be able to perform the assigned task, of course with a certain accuracy that may depend on the quality of the data presented to the model and the model itself. Deep learning as a branch of Machine Learning mainly focuses on the usage of Artificial Neural Networks (ANN) which is a layered structure of several algorithms. The biological neural network of the human brain is the model considered to be the architecture of the ANN. This trait is what makes deep learning to be considered as more capable than the normal machine learning models.

Artificial Neural Networks

In 1958, the first ANN was invented by psychologist Frank Rosenblatt [1]. Modelling how visual data is processed by the human brain and trained to recognize objects was its main intention; its name was Perceptron [1]. Later on it was realized that ANNs could be used in their own special

way, in which their learning capabilities and pattern-matching made them superior to the basic statistical and computational methods, thus by the late 1980s, ANNs were used for a wide range of reasons by numerous real-world institutes. Before getting into one of the most used algorithms of the ANN, we would like to elaborate more on the structure of the general neural network.

A standard neural network (NN) is a combination of "neurons"[2]; as a resemblance to the neuron that is in the human brain. These neurons have many inputs but have only one output. This output is calculated by simply multiplying all the inputs coming to the neuron by a specific value called the "weight". The learning process is about finding the correct weights that may make the model produce the output intended [2]. Deep Learning is about accurately assigning credit across many such stages. General neural networks are considered multi-layered, they may have an input layer, an output layer and one or more hidden layers as shown in Figure 1 [3]. The layers are formed by the neurons, each layer's inputs are connected by the outputs of the previous layer, while it is connected to the inputs of the next layer by its outputs; all the connections may be considered to be the weights.

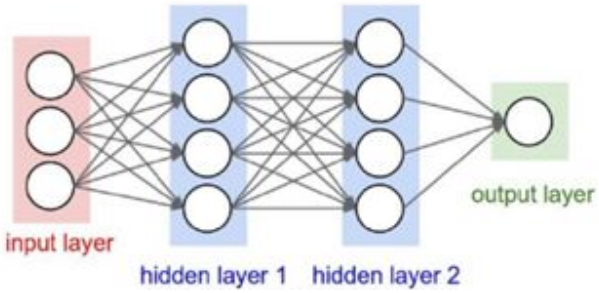


Figure 1: Neural Network [3]

Convolutional Neural Network

One of the most popular algorithms of the ANN is the Convolutional Neural Network (CNN). CNN specializes in image processing. Similar to the common neural network, the CNN has a basic algorithm, since for both the main building block for them is the concept of the weights, while the back and forward propagation are essential for the training and testing respectively. However, CNN uses the data inputted as a form of images; this is what makes it different from a common neural network. The forward propagation of CNN focuses on 2D convolution, which is used for acceleration since it implies useful parallelism. A typical CNN works sequentially and consists of many different layers. Each layer takes the inputs from the outputs of the previous layer. There are many different layers, which are mainly the convolution layer that performs 2D convolution, a fully-connected layer that multiplies with the weight, a Rectified Linear Unit (ReLU) layer performs activation thresholded at zero, and a pooling layer that does down-sampling by taking maximum or average. The last layer (the fully-connected layer) outputs an array of probabilities for the corresponding classes (results). The most frequently used CNN layers are convolution layers, pooling layers, ReLU layers and fully-connected layers [7].

Image Classification

Classification in ML is the learning process in which data is classified into numerous classes. There are several types of classification such as document classification, handwriting recognition, speech recognition, face recognition and finally image classification, which is what we are basing our project on. In the next segment, are a couple of studies made that used this type of classification while running their experiments. In the studies it has been proven how CNNs efficiently work on image classification.

For the first study, Computerised Tomography (CT) implementation of decision making for diagnosis of Alzheimer's disease (AD) has not been done yet, although it is the first tool ever for imaging and studying the human brain [4]. However, it still does provide enough features that may be used to be further studied. This is what the study aims to do, which is how deep learning, specifically CNNs, may be utilized to further classify the images obtained. The CNN architecture was advanced in which a hybrid was used between 2D and 3D CNN networks, since the images had a depth in the z direction ranging from 3-5 mms [4]. There are 3 categories that were defined, they were AD Figure 2a, lesion (tumor) Figure 2b and normal ageing Figure 2c. The main CNN's results were based on the average of results obtained from both fused networks that used the 2D images along spatial axial directions and 3D segmented blocks respectively. This CNN architecture provided accuracy results of 85.2% for AD, 80% for Lesion and 95.3% for normal ageing; average was 87.6% [4]. The results of the 2D CNN alone were also better than the usual approaches used providing accuracy rates of 86.3%, 85.6% and 85.2% for the different types of 2D CNNs [4]. More applications related will be discussed in the "Applications" section.

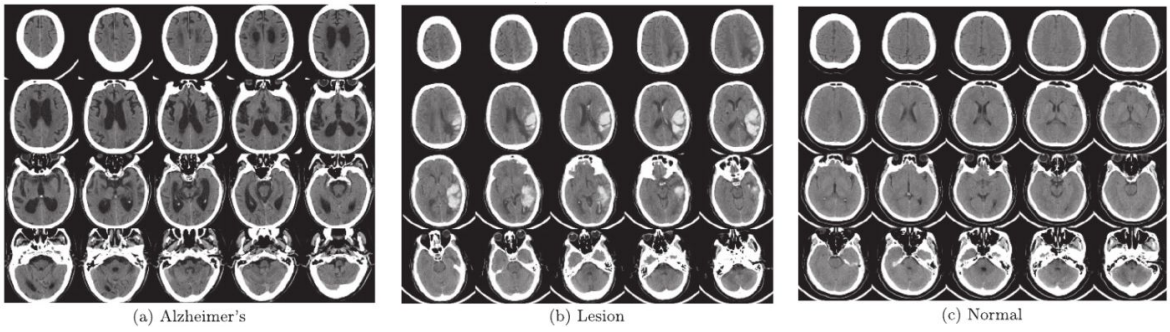


Figure 2: CT Scans for brain [4]

In this second study, the usage of CNN was different. Since Whole Slide Tissue Images (WSI) have a gigapixel resolution, it is impossible to use for training a CNN [5]. So to be able to recognize the difference between the cancer subtypes, the images are segmented to be used as patches rather than the whole image as seen in Figure 3; this made the features observable for the CNN to train. In Figure 3 the WSI image is clear with the patches magnified within frames, frames in red have the patches of the discriminative since they contain features that show grade IV tumor, while blue frames had patches that were non discriminative showing lower grade tumor. Thus, the main focus of the study was to prove that the image patch classifier would perform better than the standard image level classifier. It was challenging however from the combinations of the patches to obtain the final results. A decision fusion model was proposed to be able to decide from the patch level results obtained by the CNNs working on the patches. An Expectation-Maximization (EM) method which uses the relationship of the patches by one another is used to locate the discriminative patches [5]. The method was utilized to the classification of glioma and non-small-cell lung carcinoma cases into subtypes; the decision of the classifications is based on agreements between pathologists. For the dataset 80% were used for training while 20% were used for testing. The results delineated that the patch-level CNN

outperformed the image-level CNN obtaining higher grade classifications of 0.808 and 0.778 respectively [5].

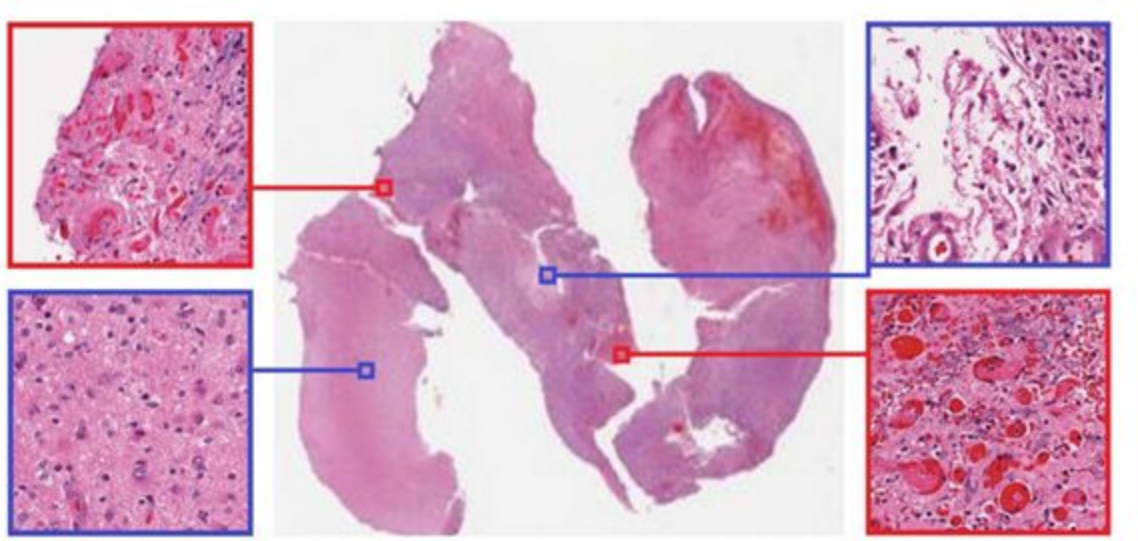


Figure 3: Whole Slide Tissue Image [5]

Field Programmable Gate Array (FPGA) CNN Frameworks

A deep learning framework is an interface, or a library for designing and training CNN models like Caffe, Theano, and TensorFlow, etc. They provide pre-built and optimised components to define models without dealing with the details of the underlying algorithms. It's a higher-level interface for the users to easily build and customise layers of different CNN models. Table (1) compares some of the most popular deep learning frameworks.

Software	Creator	Initial release	Open source	Interface	Written in	Has Pretrained models	Platform
Caffe	Berkeley Vision and Learning Center	2013	Yes	Python, Matlab, C++	C++	Yes	Linux, macOS,Windows
TensorFlow	Google Brain	2015	Yes	Python, C/C++, Java, Go	Python,C++, CUDA	Yes	Linux, macOS,Windows, Android
Theano	University of Montreal	2007	Yes	Python	Python	Yes	Cross-platform

Table (1): comparison of different deep learning frameworks[7].

Caffe

Caffe is one of the most popular open source frameworks geared towards the image processing field. It was processed by Yangqing Jia at the University of California, Berkeley. It is implemented in C++ with a useful Python interface. Caffe stands out for its strong image processing capabilities, speed, and optimisation of pre-existing models. It is among the fastest convolution networks implementations available; it's reported the ability to process over 60 million images on a daily basis with a single NVIDIA K40 GPU[6]. However, it was found less efficient in recurrent models and language models in comparison with the other frameworks. A major issue with Caffe is the difficulty of compilation and installation on Linux OS. As reported by many researchers, Caffe does not officially support Python3. The fact that Caffe built-in

libraries are written in C++ makes it harder to transmit data to Python-based boards like PYNQ [7]. This makes it harder for the developer to customise and upgrade[7].

TensorFlow

TensorFlow is a framework originally developed by Google Brain Team for research purposes in machine learning and deep learning. Unlike Caffe, it is written in Python Application Programming Interface over a C/C++ engine. TensorFlow has relatively good documentation and tutorials for beginners to help them implement the framework. Thus, the installation of TensorFlow on a PYNQ board is not hard as stated by several researchers and has existing codes for several complex deep learning models, such as RNNs (Recurrent Neural Networks)[8]. TensorFlow is popular in the fields of image recognition, such as face recognition, image captioning, and object detection, sound recognition, text-based application (like Google Translate), and video analysis. However, TensorFlow was criticised by researchers for the lack of flexibility. This is due to the declarative nature of TensorFlow; declarative programming gives users functions to use when trying to execute a command without a list of steps. As a result, researchers were not able to debug errors. For instance, when training a CNN model on TensorFlow, the syntax does not closely follow the learning process in the training process. Instead, the core operation is dictated by a simple command (yet its flow of the execution is not known); session.run()[9]. Another paper stated the same error when trying to execute a CNN[7]. Lastly, TensorFlow is considered slower than the other leading frameworks.

Theano with Lasagne

Theano is a Python library and compiler which helps users develop, optimise, and efficiently evaluate mathematical expressions involving multi-dimensional arrays efficiently[10]. It was

developed by Montreal Institute for learning Algorithms, University of Montreal. Theano uses NumPy library for efficient computations on n-dimensional array data types. However, Theano requires a higher-level user interface software package to be installed on top of it for easier CNN deployment on FPGAs. Lasagne is one of the most popular libraries used with Theano making it easier to express the architecture of deep learning models, and training algorithms, as mathematical expressions to be evaluated by Theano[10]. It's a lightweight library for building and training neural networks in Theano. Lasagne provides a more compact and intuitive code representation than Theano[7]. Using Lasagne, a user can declare a CNN with simple and understandable Python codes. This helps in customising and tailoring CNNs for different applications and sizes by adding or removing layers from the CNNs. Users can easily use Lasagne to customise CNN layers and make use of Theano's symbolic differentiation algorithm to define cost functions on iPython Notebook scripts, such as Jupyter. Lasagne allows users to build neural networks at the layer level using blocks like "Conv2DLayer" and "DropoutLayer". "Conv2DLayer" and "DropoutLayer" are lasagne classes that perform 2D convolution on the inputs. Additionally, Lasagne published Recipes, a collection of tutorials and examples demonstrating how to easily deploy pre-trained CNN models and the installation steps are very simple and easy compared to the other frameworks.

DarkNet

Darknet is an open source deep learning framework written in C and CUDA. As mentioned in [49], darknet is fast and easy to install. However, when looking for documentations and tutorials online for this framework, there were not enough sources. YOLO (You Only Look Once) is one of the applications used in darknet with a tiny darknet that's only 4.0MB. Thus, such a

framework could be useful in applications that require smaller networks due to the limited hardware resources.

Applications

Recycling

Furthermore, a field that has recently shifted towards deep learning and image classification is recycling. A lot of work is being conducted in the aim of being able to implement automated waste segregation systems into recycling units or new housing development projects. Solid waste management represents a foundational issue worldwide, and manual segregation of solid waste results in numerous health hazards for the workers, as well as inaccurate results and a very hectic and time-consuming process[48]. Accordingly, the need of separate waste segregation systems in householdings or recycling units is becoming more and more needed nowadays. This would facilitate the process greatly resulting in a greener and healthier environment. Numerous research and projects are being done as well in this field. In [48] for example, the model utilizes OpenCV and fast R-CNN algorithms. The following results shown in figure 4 were achieved.

S.no.	Type of waste	Accuracy
1	Glass	99
2	Wood	67
3	Paper	97
4	Textile	70
5	Metal	70
6	Plastics	97

Figure 4: Results of waste segregation system[48]

Another study in the University of South Africa, implemented a waste classification system using deep learning. [13]They used the 50-layer residual (ResNet-50) CNN model which was pre-trained. Transfer Learning, a technique that enables developers to create an accurate network for a new dataset without a long training process, was then conducted. The fully connected layer was removed and replaced by Support Vector Machine (SVM) to train their small dataset that consisted of 1989 images, and divided into four different categories; glass, paper, plastic and metal. The accuracy obtained at the end was 87%.

Agriculture

Agriculture, as well, is a vast field that deep learning has recently had a huge impact on. Although meeting the demand to provide enough food for more than 7 billion people is not an easy task, modern technologies have made human society able to act accordingly and be able to cope with such high demand. However, several factors still threaten food security such as climate change, decline in pollinators and plant diseases.

In previous work conducted by Toda et al., PlantVillage dataset was used, which is classified into 38 labels (54,306 images, 26 diseases, 14 crop species), and the images were split into training, validation, and test datasets with a ratio of 6:2:2. The network used in this method was a modified CNN based on InceptionV3. Training of the CNN was then conducted using Keras Python library with TensorFlow backend. An accuracy of 97.15% was reached by the model[14].

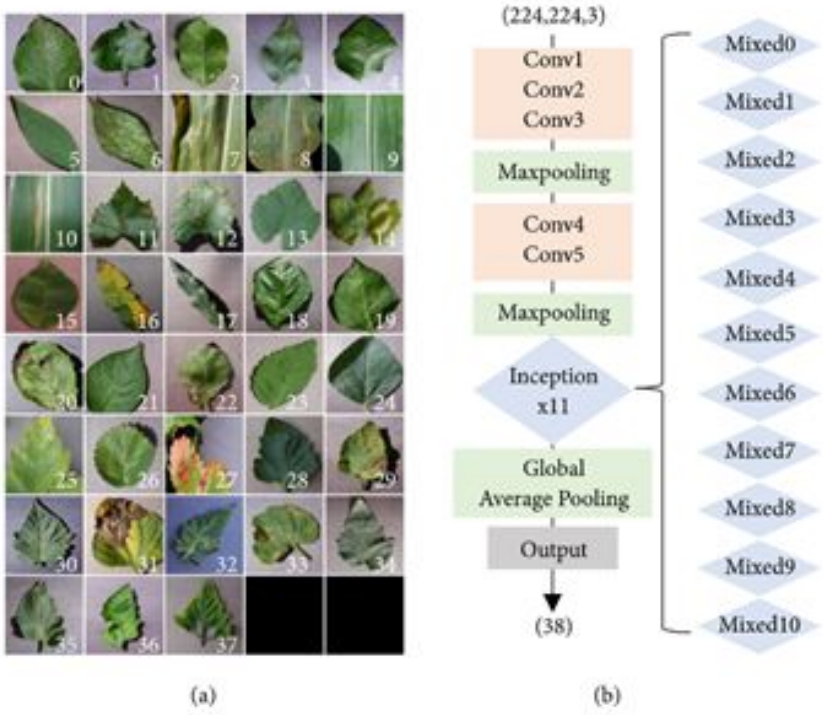


Figure 5: Image-based disease diagnosis training using CNN[14]

Another work done in this field also used the PlantVillage dataset, however, the dataset was used in three different versions which are; colored images (as it is), a gray-scaled version and a version of the dataset where the leaves were segmented, thus removing all the extra background information which might have the potential to introduce some inherent bias in the dataset due to the regularized process of data collection in case of PlantVillage dataset. This research mainly focused on two specific CNNs that are AlexNet and GoogLeNet. For the training mechanism, transfer learning and training from scratch were both implemented, one at a time. Moreover, several training-testing set distributions were tried, and the performance based on each of the above-mentioned parameters was noted. The best performance at the end was achieved with GoogLeNet, using Transfer Learning and the normal colored dataset with 80% of the images used for training and 20% used for testing[15]. When implementing a plant diseases classifier in

real life however, an unmanned aerial vehicle (UAV) is usually needed to monitor the agricultural field by its own. In fact, various models have been already implemented, tested and are being used by various stakeholders. According to Tetila et al., one project included a Phantom 3 Professional UAV, equipped with a 1/2.3-inch Sony EXMOR sensor and 12.3megapixel resolution. However, the UAV needed to be at a height of 2 meters to effectively capture pictures of the leaves in the field. In this particular example, and with the pictures captured from the drone, an accuracy of 99.04% was reached, using the Inception-v3 CNN, with transfer learning and fine tuning based on a customized dataset[16]. In addition to that, images acquired by the RGB camera produce good accuracy with a resolution of 0.2 mm. In fact, it is recommended to use UAV acquisitions with a resolution not worse than 1 mm at the leaf level to produce high accuracy[17].

The Standards Involved in Our Work

In embedded systems, communication between the system and other peripherals must follow certain protocols and standards. Serial communication protocols have become one the most important aspects of embedded systems [1]. USB is one of the most popular serial communication protocols [4]. In this protocol, data is transferred from/to the USB host to/from the device in a form of packets. On the other hand, UART is a different protocol. It is a hardware circuitry that acts as a bridge between the process of the SoC and the serial communication protocol (USB). The typical features of UART on the PYNQ-Z1 board are 115200 baud rate, 8 data bits, 1 stop bit, and no parity [2].

UART and USB

The PYNQ-Z1 board supports many types of serial communication protocols and interfaces including SPI, UART, CAN, I2C, USB 2.0, HDMI, and SDIO as shown in figure 6 [2]. For high band-width communication, 1G Ethernet, USB 2.0, SDIO are the controllers provided by the PYNQ-Z1 board. On the other hand, for high band-width communication, SPI, UART, CAN, I2C are the controllers available. The board has an external connection port for the Ethernet, USB, HDMI and UART. UART is the protocol used for the serial communication between the board and the PC over USB. The host computer used for programming the board is connected via USB cable with a MicroUSB connector. The connector is plugged into the MicroUSB Port of the board. The MicroUSB Port of the PYNQ-Z1 is named PROG/UART and a USB-UART bridge is attached to it [3]. This port is used to set up the board and program it as illustrated in figure [7].

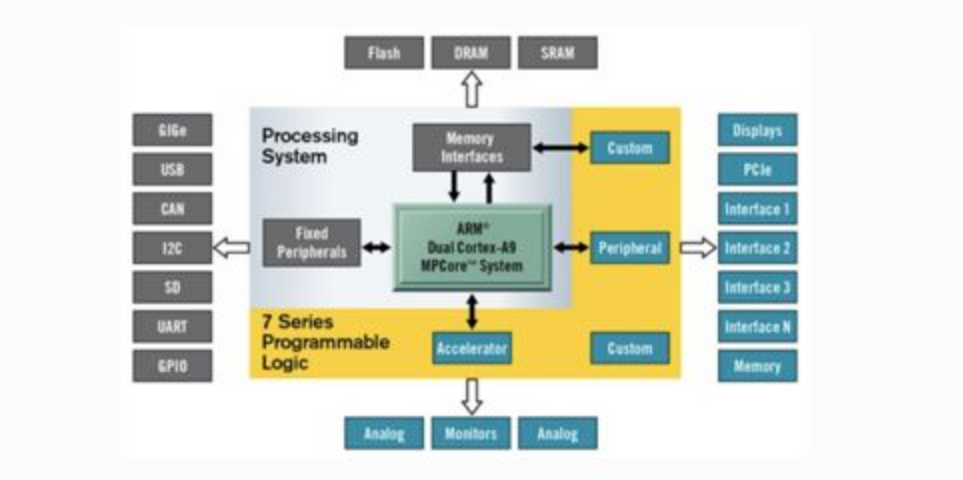


Figure 6: Serial Communication Interfaces on PYNQ-Z1 (5)

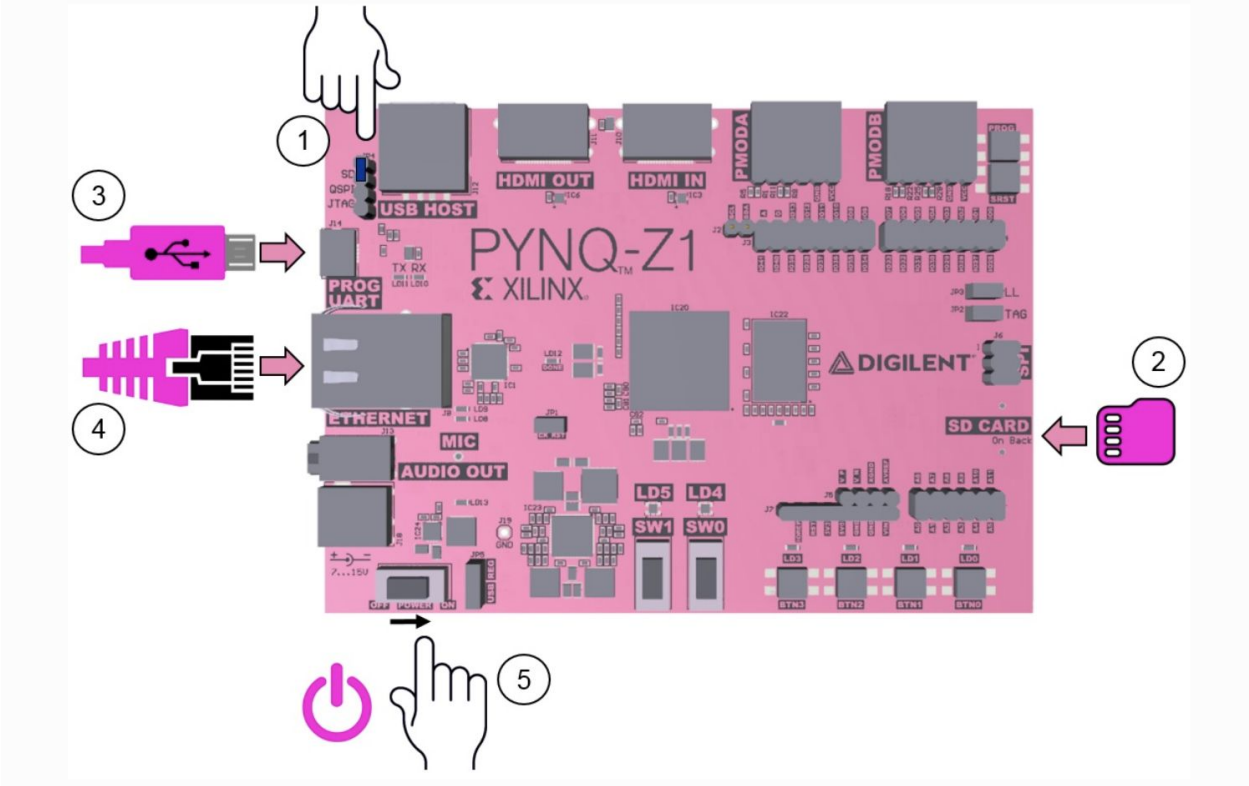
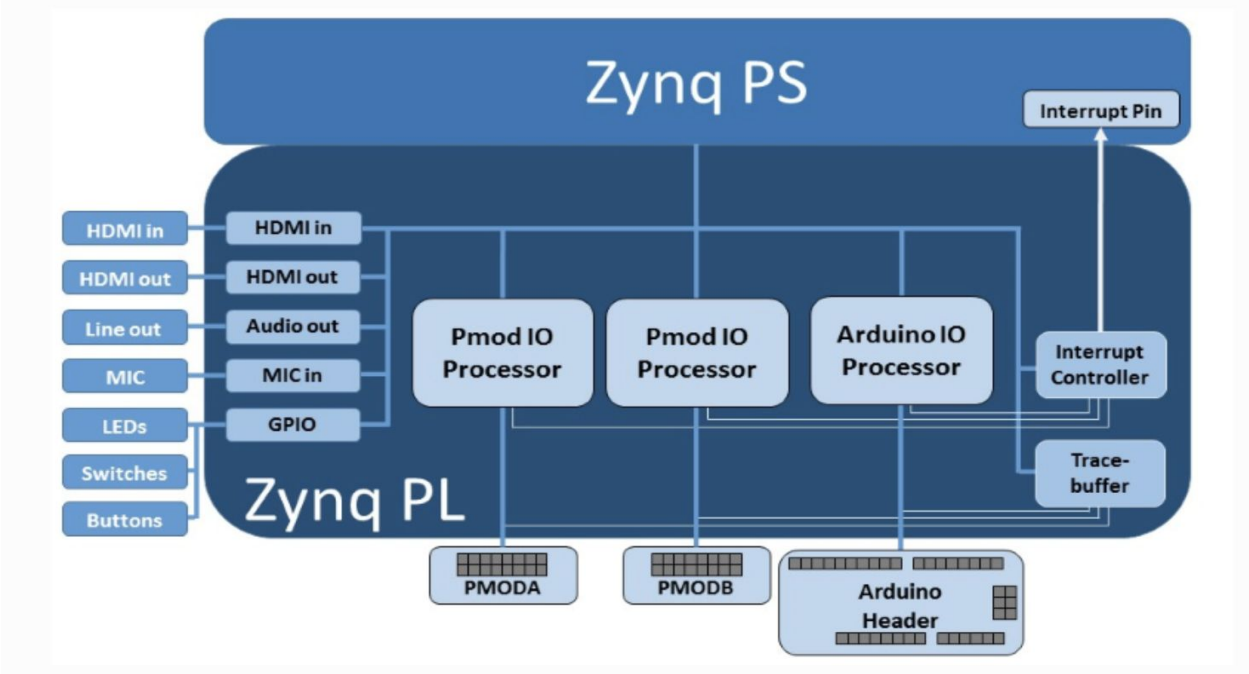


Figure [7]: UART/PROG port on PYNQ-Z1 [2].

HDMI

HDMI or High-Definition Multimedia Interface is a standard used in high bandwidth digital media transmission. The HDMI standard was originally developed by large electronics companies as Hitachi, Panasonic, and Silicon Image to name a few [1]. As mentioned in High-Definition Multimedia Interface, HDMI employs three communication protocols: DDC, TMDS, and CEC. TMDS is the method used for encoding the data transferred through an HDMI cable. It's used to allow a transfer of high-quality images and videos over large distances. The cable includes a twisted pair cable where the signal(image) and its inverse are transmitted. This allows the TDMS protocol to calculate the difference between them in a certain way and improve the quality of the transferred signals[1]. DDC is another communication protocol that is a subset of the I2C protocol [2]. DDC follows the rule of I2C; it dictates the communication protocol between the master (PC sending the image) and the slave (PYNQ). PYNQ-Z1 has a base overlay for HDMI interface, which will be furtherly mentioned in section "List of Components". Base overlays are hardware libraries included in the PYNQ PL design [3]. They allow the user to customise the hardware platform for a specific application. HDMI is one of the base overlays that are automatically linked to the PL as shown in figure [8]. Overlays allow the user to use HDMI conveniently with the PYNQ board.



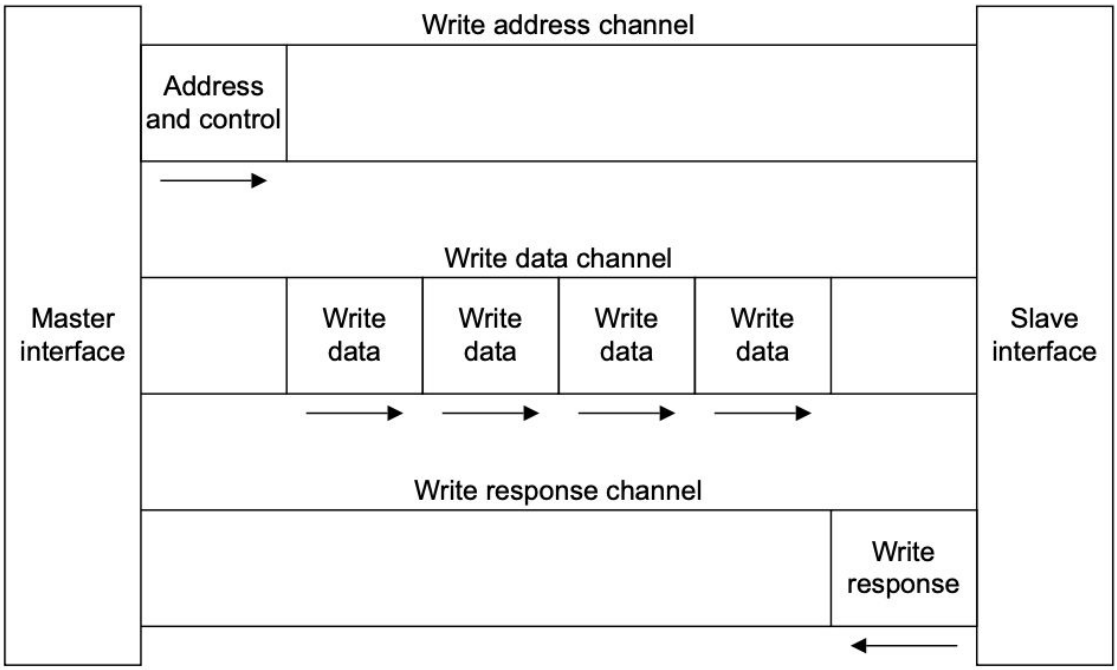
Figure[8]: HDMI- Base overlay

ETHERNET

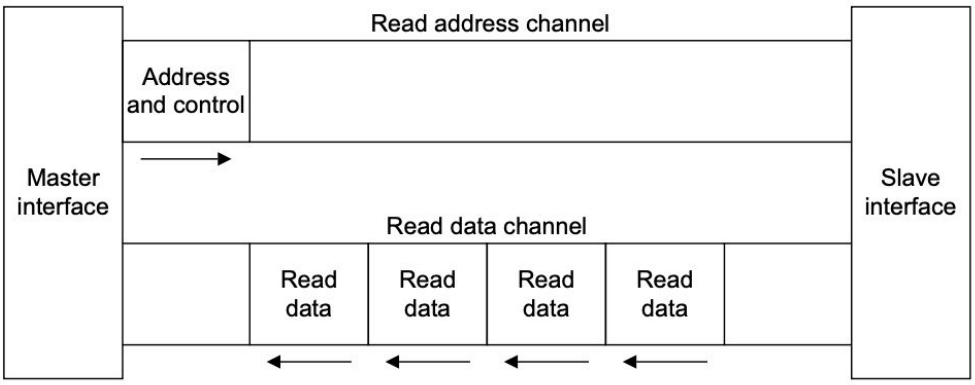
Ethernet is one of the most famous networking protocols for connecting devices over a network. PYNQ-Z1 supports Gigabit Ethernet PHY[4] which allows fast transmission of data. As per Vijay Moorthy in [5], Ethernet is another word for IEEE 802.3 standard. It uses CSMA/CD protocol to control data transmission between the connected devices and avoid data corruption. In our implementation, an Ethernet cable enables us to use the PYNQ hardware/software interface.

AXI

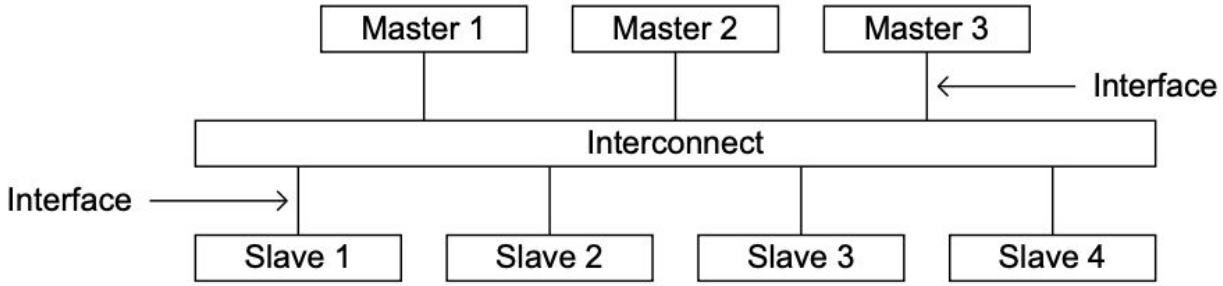
AXI is one of the bus protocols of ARM Advanced Microcontroller Bus Architecture for microcontrollers. It connects multiple masters and slaves on an interconnect as illustrated in figure[11]. AXI typically has 5 channels for data transmission according to "AMBA® AXI™ and ACE™ Protocol Specification"[6]: read address, read data, write address, write data, and write response. In a write data transaction, data is transferred from the master to the slave as in figure[9] on the write data channel. The slave acknowledges the transaction by sending a signal on the write response channel. On the other hand, in a read data translation, data is transferred from the slave to the master on the read data channel[7]. The address channels are used to define the nature of the transferred data. The data exchange for the AXI protocol is described in figures[9] and [10]. PYNQ-Z1 uses AXI to connect different modules within a design and the IPs to the processor[7].



Figure[9]: AXI Write transaction



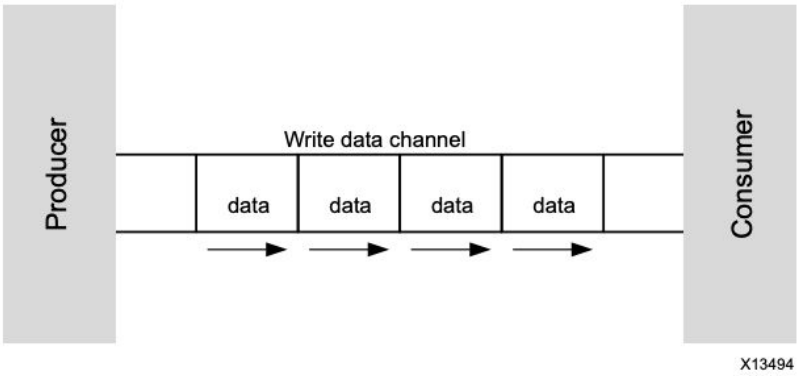
Figure[10]: AXI Read transaction



Figure[11]: AXI Interconnect

AXI4 is an advanced AXI protocol that provides more flexibility in one's design. In PYNQ, AXI4 communication can be divided into two main categories: AXI4-Lite and AXI4 stream[1]. AXI4-Lite is a memory mapped bus connection that is used by the PYNQ ARM processor(PS-master) to access the IPs(slave in the PL design[1]. This type of AXI4 connection is much simpler than the AXI4 Stream. AXI4 Stream is a point-to-point communication bus between IPs on the PLdesign. Data is sent between the "producer" and the "consumer" as long as there's an available space. The "consumer" accepts data as long as the data channel is not empty; see figure[12]. This streaming interface allows a high bandwidth for data transmission and is more complex than an AXI-Lite communication[1]. To control AXI4 Stream masters and slaves,

a set of controller IPs are provided. AXI4 Stream Switch and AXI4 Stream Interconnect are two of the controllers used that support up to 16 masters and slaves[3]. They are used to manage the routing and switching between different AXI streams for better control and higher data transmission rate.

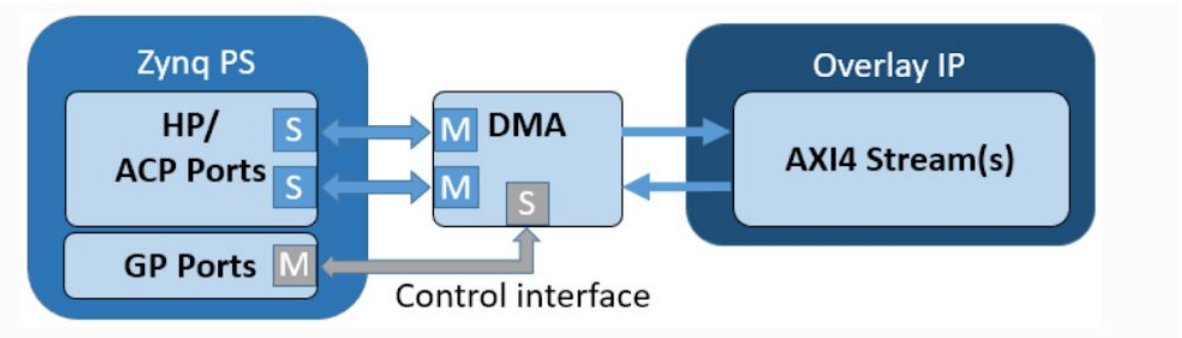


Figure[12]: AXI4 Stream

DMA

DMA is a protocol for efficient memory access. DMA is used to connect peripherals directly to the memory bus through a DMA Controller without the help of the processor. It improves the speed of memory access and allows for high data throughput[8]. Also, DMA takes the load off the processor when a peripheral needs to access the memory. This allows the processor to work on other tasks while the peripherals access the memory.

Xilinx has a ready-packaged IP for DMA. It is mainly composed of an AXI control interface and read and write channels. The read channel is used to read data from the PS DRAM through an AXI-Lite master port and then write the data read to an AXI Stream port connected to an IP in the PL. On the other hand, the write channel reads the data from a stream using the stream port connected to the IP, then writes it back to PS DRAM using the AXI master port.



Figure[13]: DMA in PYNQ

Overall Design

This section describes the overall image classification flow using PYNQ SoC. Our Hardware-Software interface is the Jupyter notebook that is used to interact with the processing system on the PYNQ. Our CNN is trained using GPU, and the obtained weights from the Lasagne trained network are, then, deployed on the hardware . The flow is then divided into two parts: PS and PL that are communicating with each other using DMA. Firstly, a communication is established between the HDMI interface on the board to be able to transfer the image to it. After the image is captured, it is saved on the board's SD card. The next step is to preprocess the image as a numpy format to match the images in our npz dataset. This preprocessing takes place on the board's processing system after loading the image to the DDR. The image is transferred from the PS side to the PL side from the DDR via DMA; and a classification is obtained. Then, the CNN output (image classification) will be passed to the PS side again via the DMA showing the obtained results.

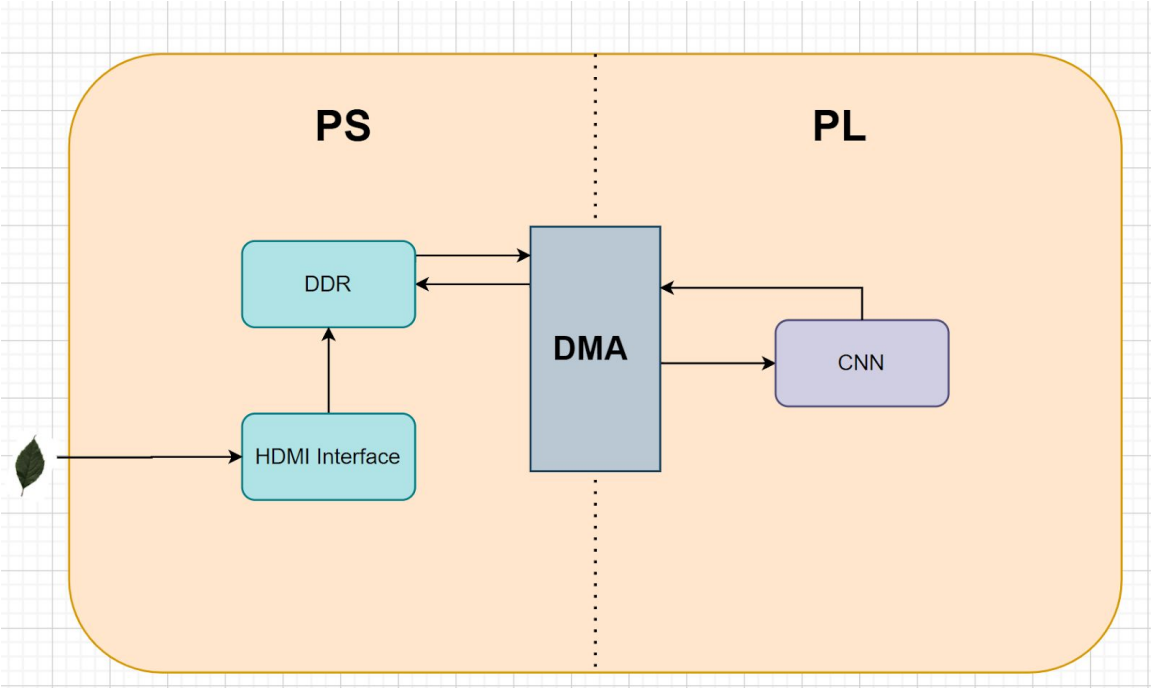


Figure [14]: Overall Image Flow

Design of Individual System Modules

Software

Dataset

In our project, we have used the “New Plant Diseases” dataset which we have obtained from Kaggle datasets. “New Plant Diseases” dataset consists of about 86000 jpg images of 10 different types of plants’ leaves. In our project, we have divided this dataset into 3 categories: training, testing and verification. About 58000 images (about 67.4 %) were allocated for training, 18000 images (about 21 %) were allocated for testing and 10000 images (about 11.6 %) were

allocated for validation. We have also classified this dataset into 2 classes: diseased and healthy leaves in order to make it suitable for our application. The following paragraph will discuss how these jpg images are transformed into an npz file to be used by our CNN.

Dataset Size Transformation

To transform the dataset in hand to meet the dimension of 32x32 of the architecture settled on, Python was used to achieve this task. The library used for image processing and manipulation is called Pillow, along with a package called "Image" the task was easily done. In order to make a generic code to resize any number of images inside any directory, the library "OS" was used to do that. So, the Python script created is to be given a directory, new dimensions, and an extension. Figure [15] shows the Python script used to transform the size of the dataset.

```
from PIL import Image
import os
#constants
directory = r"C:/Users/OElFarna/Desktop/Dataset"
NEW_PIXEL_SIZE1 = 32
NEW_PIXEL_SIZE2 = 32
EXTENSION = 'jpg'
os.chdir(directory)
for i, f in enumerate (os.listdir(directory)):
    f_name, f_ext = os.path.splitext(f)
    new_name = "image_" + str(i) + "." + EXTENSION
    os.rename(f,new_name)
    image = Image.open(new_name)
    image = image.resize((NEW_PIXEL_SIZE1,NEW_PIXEL_SIZE2))
    image.save(new_name)
    print(image.size)
```

Figure [15]: Python Script To Transform The Dataset's Dimensions

Dataset Format Transformation

When preparing the format of our dataset, NumPy was used which is the core library for scientific computing in Python. Through it, we were able to have a high-performance multidimensional array object, and tools for working with these arrays. Accordingly, the Dataset was transformed into numpy arrays. A numpy array is a grid of values, all of the same type, and indexed by an ordered pair of nonnegative integers [7]. Numpy arrays are also, generally, faster than other formats. They consume less memory to store the data and are more efficient and flexible to use. Accordingly, we transformed our dataset into numpy arrays along with its indices for diseased or healthy. The numpy arrays were then archived into .npz format. Figure [16] shows the python script used for the transformation.

The transformation was carried out 2 times, one with data normalization and one without. Substantially, Normalization does not cause any distortions in the differences between the ranges of values, however, it alters the values in the numeric columns in the dataset to a common scale[6]. Normally, normalization results in better accuracy and more efficient training[8].

dataset_transform.py > ...

```python
import numpy
import keras
import os
import ctypes
import tensorflow as tf

def images_to_array(dataset_dir, image_size):
    dataset_array = []
    dataset_labels = []

    class_counter = 0

    classes_names = os.listdir(dataset_dir)
    for current_class_name in classes_names:
        class_dir = os.path.join(dataset_dir, current_class_name)
        images_in_class = os.listdir(class_dir)

        print("Class index", class_counter, ", ", current_class_name, ":" , len(images_in_class))

        i = 0
        for image_file in images_in_class:
            i = i+1

            if image_file.endswith(".jpg"):
                image_file_dir = os.path.join(class_dir, image_file)

                img = keras.preprocessing.image.load_img(image_file_dir, target_size=(image_size, image_size))
                #print("here" + str(i))
                img_array = keras.preprocessing.image.img_to_array(img)
                # img_array = img_array.astype(int)
                # print(type(img_array))
                #img_array = int(img_array)
                img_array = img_array/255.0
                # print(img_array)
                dataset_array.append(img_array)
                dataset_labels.append(class_counter)
        class_counter = class_counter + 1
        # print(class_counter)
    dataset_array = numpy.array(dataset_array)
    dataset_labels = numpy.array(dataset_labels)
    return dataset_array, dataset_labels

#zip_file = tf.keras.utils.get_file(origin="https://drive.google.com/drive/u/0/folders/0AOdLiYzohYgzUk9PVA",fname="dataset_32.zip", extract=True)
#base_dir, _ = os.path.splitext(zip_file)

train_dir = "Full_Dataset_No_Norm/train"
image_size = 32
train_dataset_array, train_dataset_array_labels = images_to_array(dataset_dir=train_dir, image_size=image_size)
print("Training Data Array Shape :", train_dataset_array.shape)
numpy.save("train_dataset_array.npy", train_dataset_array)
numpy.save("train_dataset_array_labels.npy", train_dataset_array_labels)

test_dir = "Full_Dataset_No_Norm/test"
test_dataset_array, test_dataset_array_labels = images_to_array(dataset_dir=test_dir, image_size=image_size)
print("Test Data Array Shape :", test_dataset_array.shape)
numpy.save("test_dataset_array.npy", test_dataset_array)
numpy.save("test_dataset_array_labels.npy", test_dataset_array_labels)
```

Figure [16]: Code for dataset transformation to npz file

TensorFlow

TensorFlow was the first framework to be used for training the CNN for this application. Firstly, we trained a network on the CIFAR-10 dataset with a similar architecture to the one used in our implementation with minor differences. The classification and testing results were similar to the ones on the CIFAR-10 TF tutorials [11].The testing accuracy and classification results are shown in figure[18] and [17], respectively.

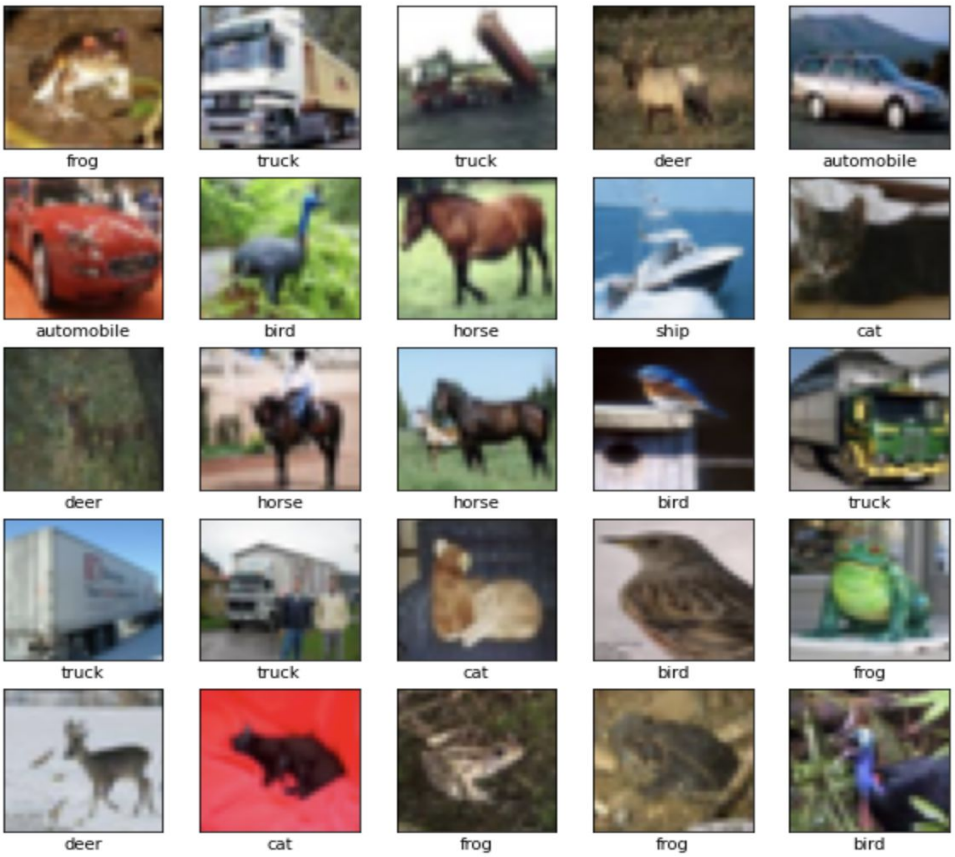
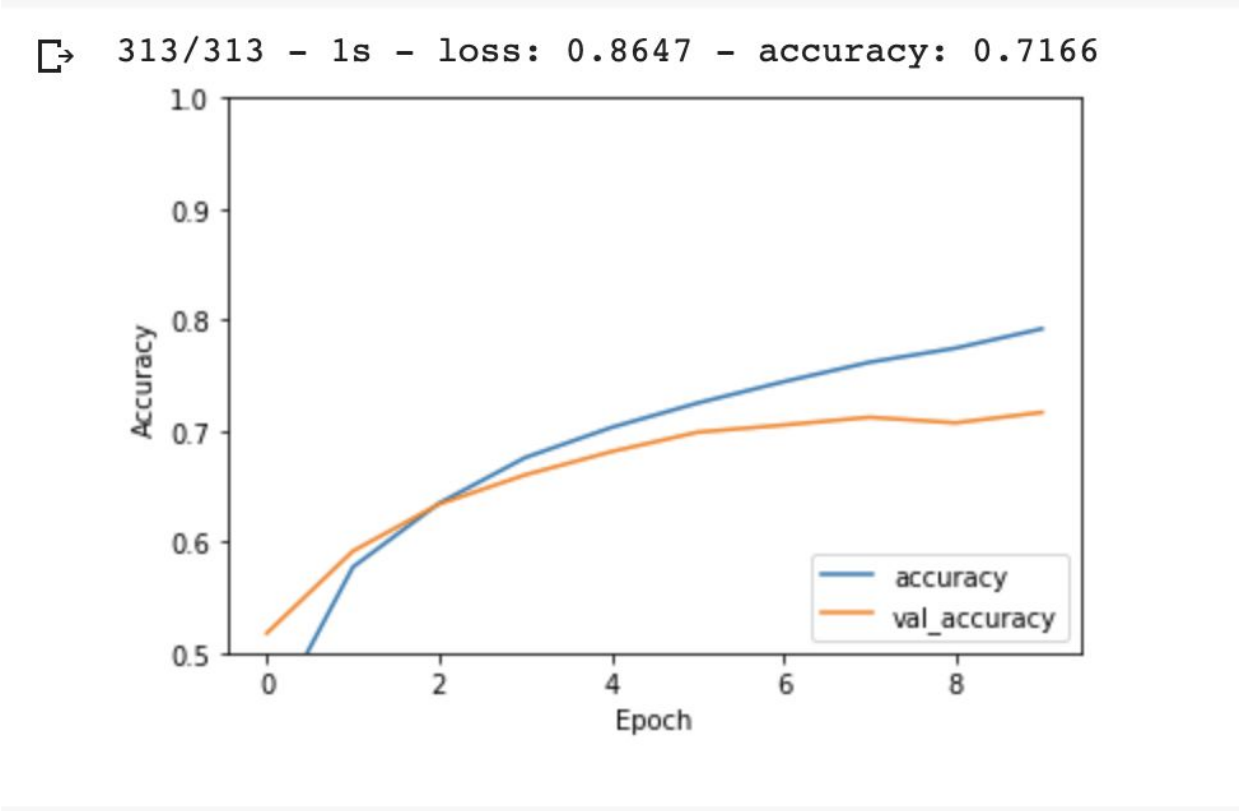


Figure [17]: Classification results of CIFAR-10 using TF



Figure[18]: CIFAR-10 Validation accuracy using TF

The next step was to train a CNN using TF on our "New Plant Diseases" dataset with the same CNN architecture as the one used with the CIFAR-10 dataset. The summary of the CNN training is shown in figure [19]. One of the advantages of TF is that no transformation of the dataset is needed. The zipped dataset folder was simply used in the TF model, unlike Lasagne framework. After uploading the dataset and building the CNN, the model was trained for approximately half an hour on Google Colab. The batch size was set to 30 while the epochs were set to 20. Additionally, Adam optimizer was used to train our model. An optimizer is an extended class that helps in training the model. Its function is to improve the speed and performance of the model training [12]. Afterwards, the accuracy and the validation loss of our model was monitored. In figure [20], the accuracy of the model started with 83.56% and it increased to 98.16% at the 16th epoch. However, the validation accuracy adapted a bizarre pattern. It did not increase in a consistent pattern; it fluctuated as the training accuracy increased as illustrated in figure [21]. The reason for this is that our CNN model experienced an over-fitting. Over-fitting happens due to the low

bias but high variance of the model [14]. TensorFlow was not the easiest framework to install on the PYNQ-Z1 board according to [7]. Therefore, we decided to continue software implementation using Theano with Lasagne as our framework with a new CNN architecture.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 212, 212, 16)      448
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 106, 106, 16)      0
_________________________________________________________________
dropout (Dropout)            (None, 106, 106, 16)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 106, 106, 32)      4640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 53, 53, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 53, 53, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 26, 26, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 26, 26, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 26, 26, 128)       73856
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 13, 13, 128)       0
_________________________________________________________________
dropout_2 (Dropout)          (None, 13, 13, 128)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 13, 13, 128)       147584
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 6, 6, 128)         0
_________________________________________________________________
dropout_3 (Dropout)          (None, 6, 6, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 4608)              0
_________________________________________________________________
dense (Dense)                (None, 512)               2359808
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 1026
=================================================================
Total params: 2,605,858
Trainable params: 2,605,858
Non-trainable params: 0
_________________________________________________________________
```

Figure [19]: CNN architecture summary on TF

```
Epoch 1/20
2079/2079 [==============================] - 98s 47ms/step - loss: 0.3512 - accuracy: 0.8356 - val_loss: 0.4332 - val_accuracy: 0.7746
Epoch 2/20
2079/2079 [==============================] - 96s 46ms/step - loss: 0.2211 - accuracy: 0.9088 - val_loss: 0.2877 - val_accuracy: 0.8709
Epoch 3/20
2079/2079 [==============================] - 94s 45ms/step - loss: 0.1713 - accuracy: 0.9323 - val_loss: 0.2686 - val_accuracy: 0.8848
Epoch 4/20
2079/2079 [==============================] - 96s 46ms/step - loss: 0.1447 - accuracy: 0.9432 - val_loss: 0.4451 - val_accuracy: 0.8232
Epoch 5/20
2079/2079 [==============================] - 96s 46ms/step - loss: 0.1242 - accuracy: 0.9522 - val_loss: 0.2168 - val_accuracy: 0.9093
Epoch 6/20
2079/2079 [==============================] - 95s 46ms/step - loss: 0.1076 - accuracy: 0.9593 - val_loss: 0.1948 - val_accuracy: 0.9222
Epoch 7/20
2079/2079 [==============================] - 96s 46ms/step - loss: 0.0963 - accuracy: 0.9630 - val_loss: 0.2340 - val_accuracy: 0.9007
Epoch 8/20
2079/2079 [==============================] - 96s 46ms/step - loss: 0.0893 - accuracy: 0.9667 - val_loss: 0.2222 - val_accuracy: 0.9111
Epoch 9/20
2079/2079 [==============================] - 97s 46ms/step - loss: 0.0831 - accuracy: 0.9697 - val_loss: 0.1751 - val_accuracy: 0.9330
Epoch 10/20
2079/2079 [==============================] - 97s 47ms/step - loss: 0.0759 - accuracy: 0.9724 - val_loss: 0.4246 - val_accuracy: 0.8628
Epoch 11/20
2079/2079 [==============================] - 97s 47ms/step - loss: 0.0661 - accuracy: 0.9753 - val_loss: 0.1939 - val_accuracy: 0.9323
Epoch 12/20
2079/2079 [==============================] - 95s 46ms/step - loss: 0.0619 - accuracy: 0.9772 - val_loss: 0.2196 - val_accuracy: 0.9214
Epoch 13/20
2079/2079 [==============================] - 95s 46ms/step - loss: 0.0625 - accuracy: 0.9775 - val_loss: 0.2551 - val_accuracy: 0.9069
Epoch 14/20
2079/2079 [==============================] - 95s 46ms/step - loss: 0.0585 - accuracy: 0.9789 - val_loss: 0.2396 - val_accuracy: 0.9216
Epoch 15/20
2079/2079 [==============================] - 95s 46ms/step - loss: 0.0580 - accuracy: 0.9797 - val_loss: 0.2136 - val_accuracy: 0.9265
Epoch 16/20
2079/2079 [==============================] - 95s 46ms/step - loss: 0.0518 - accuracy: 0.9816 - val_loss: 0.4493 - val_accuracy: 0.8725
```

Figure [20]: Accuracy and Validation accuracy of our Model in TF

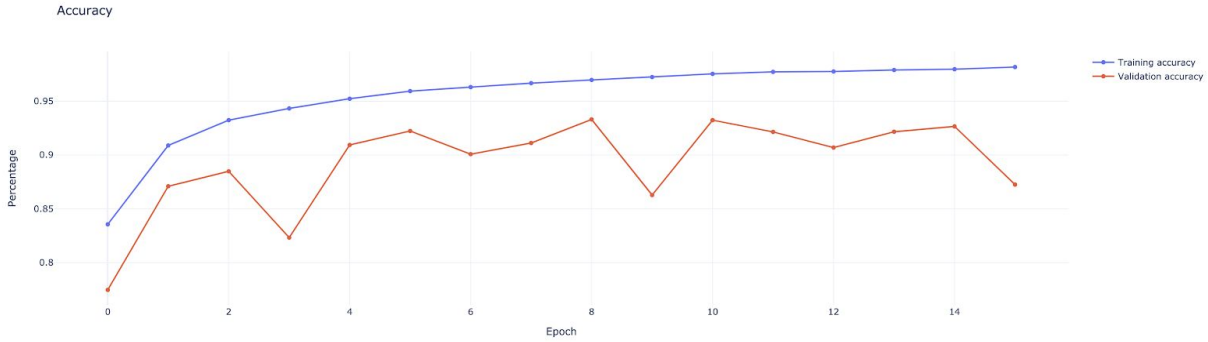


Figure [21]: Accuracy and Validation accuracy vs epochs

Theano Framework

Theano along with Lasagne was extremely helpful and simple for CNN implementations and other deep learning applications. Lasagne offers helpful tutorials and examples that were helpful for creating CNNs as well as training and even testing. Furthermore, the installation as well as the configuration of Theano with Lasagne was simple, unlike other frameworks that were tried before like Caffe and DarkNet. Thus, after settling on the framework it was time to start implementing the new CNN in mind using Lasagne.

CNN Architecture

A vital step in the project was the decision of the CNN architecture to be used. That is because for a CNN the parameters of convolutional, pool and fully connected layers are all salient for the training process and attaining good results. After research in the literature studies it was concluded that a decent starting architecture might be the Cifar-10 one; to be clearer, this architecture is the one used for the majority of Cifar-10 dataset-related applications. Consequently, a decision was made of testing this architecture as a starting point, then fine tuning the hyperparameters present such as filter sizes, number of layers and others in order to achieve better results. Additionally, if fine tuning did not achieve any improvements, new architectures in other literature studies would have been needed. This implementation had a total of nine layers: three convolutional layers, three pool layers, one flattening layer, one input layer and one output layer; figure [22] shows a layout for the layers in the CNN and figure [23] shows the summary of the layers. The input layer was of dimensions 32x32x3 as the input images are in the RGB format. Besides, this architecture has a constant filter size for convolution along the three layers of 2x2, and the three layers have a padding of 2x2 as well. For the first convolutional layer, it has 32 filters, and the same for the second layer; but for the third one, it has 64 filters. What was different about the second and third layer, however, was having the nonlinear function ReLU. Moreover, most of the implementations that adapt this CNN design had its pool layers as Max Pool ones, however, it was found in the literature [7] that Average Pool accomplished more reliable results for our network architecture; thus, average pool was used instead of max for two of the three pool layers in the design. Also, all the pool layers had a stride of 2x2 as well as a pool size of 2x2. As mentioned, this architecture was made for the Cifar-10 dataset, which has ten different classes to be outputted from the softmax layer. Nonetheless, this final layer was left

as it is, and in case the architecture produced good results it will be left to its state. The reason for that, is that the implementation found in [7] paper was targeting the Cifar-10 dataset; and to avoid changing this part in the RTL design and adapt the one found in the study, the softmax layer was left with ten outputs as an initiative. Finally, the CNN was written in code using Lasagne in Python code with a simple and understandable format that represents the network exactly; the code is shown in figure [24].

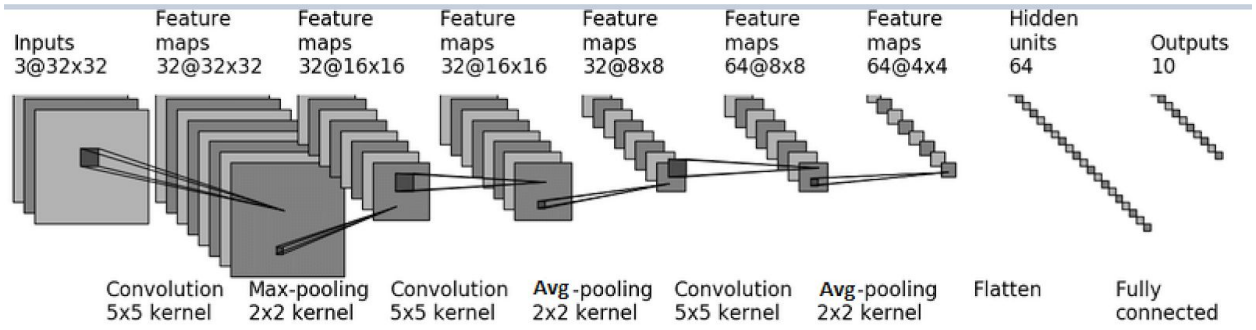


Figure [22]: CNN Architecture

Layer Type	Number	Functionality
Convolution	3	Extract the main features from the input dataset using feature maps
MaxPool	1	Used to reduce the number of features to be used in classifying the input dataset using a 5x5 filter
Average Pool	2	Used to reduce the number of features to be used in classifying the input dataset using a 2x2 filter
Fully-Connected	1	Used to classify the input according to the specified classes in the designed network and produces the final output

Figure [23]: CNN Layers Summary

```
def build_cnn(input_var=None):

    #Input
    network = lasagne.layers.InputLayer         (shape=(None, 3, 32, 32),input_var=input_var)
    #Conv
    network = lasagne.layers.Conv2DLayer        (network, num_filters=32, filter_size=(5, 5),pad = 2,nonlinearity= None)
    #MaxPool
    network = lasagne.layers.MaxPool2DLayer     (network, pool_size=(2, 2), stride = (2,2), ignore_border = False)
    #NonLinear
    network = lasagne.layers.NonlinearityLayer  (network, nonlinearity=lasagne.nonlinearities.rectify)
    #Conv
    network = lasagne.layers.Conv2DLayer        (network, num_filters=32, filter_size=(5, 5),pad = 2 ,nonlinearity=lasagne.nonlinearities.rectify)
    #AvgPool
    network = lasagne.layers.Pool2DLayer        (network, pool_size=(2, 2), stride = (2,2),ignore_border=False, mode = 'average_exc_pad')
    #Conv
    network = lasagne.layers.Conv2DLayer        (network, num_filters=64, filter_size=(5, 5),pad = 2 ,nonlinearity=lasagne.nonlinearities.rectify)
    #AvgPool
    network = lasagne.layers.Pool2DLayer        (network, pool_size=(2, 2), stride = (2,2),ignore_border=False, mode = 'average_exc_pad')
    #Dense
    network = lasagne.layers.DenseLayer         (network, num_units=64,nonlinearity=None)
    #Dense
    network = lasagne.layers.DenseLayer         (network, num_units=10,nonlinearity=None)
    #Output Layer
    network = lasagne.layers.NonlinearityLayer  (network, nonlinearity=lasagne.nonlinearities.softmax)

    return network
```

Figure [24]: CNN Implementation With Lasagne

Training

After having the CNN architecture ready and the dataset format in npz ready, it was time to train the network. Training is the most critical part of any deep learning implementation, and it is the most time consuming. Luckily, we had the chance of a powerful computer available to us, equipped with the state of the art GPU Nvidia GeForce RTX 2080 Ti that accelerated the training time immensely. As stated above, the training set was about 70% of the dataset, which was more than enough. For the choice of hyperparameters, it was salient to look carefully at them before deciding which ones to use; also, several trials of training were done with different hyperparameters in order to achieve the best results possible. The initial choices for the hyperparameters were as follows: 256 epochs, 128 images per batch, learning rate of 0.01 and momentum of 0.9. After several trials, it was conducted that the best results came when this selection of hyperparameters was agreed upon: 300 epochs, batch size of 128, 0.0001 as the learning rate and same momentum of 0.9. The old choice of learning rate in specific made the

gradient descent fail, making it go away from the local minima; thus, decrementing the rate a couple of times solved the issue, which is known as the NaN issue. In addition, various trials of training were done using the normalized dataset and the unnormalized one, which gave slightly different results.

```
def main(model='cnn', num_epochs=300):
    #constants
    TRAIN_SET_CONSTANT      = 56544
    TEST_SET_CONSTANT       = 17572
    VAL_SET_CONSTANT        = 10000
    BATCH_CONSTANT          = 128
    LEARNING_RATE_CONSTANT  = 0.0001
    MOMENTUM_CONSTANT       = 0.9
    # Load the dataset
```

Figure [25]: Final Choice Of Hyperparameters

Testing

To test the performance of the CNN that was trained, as mentioned above, the test set was about 20% of the dataset. Plus, a validation set of about 10% was used as well for better performance analysis. The example code tutorial provided by Lasagne was used for testing and outputting the results of each epoch and the final result in an organized manner. The accuracy for our model was in the 70-80 percentiles, nevertheless, after a couple of modifications to the hyperparameters the accuracy became stable in the 90s region. There was a difference, however, in the testing for normalized and unnormalized datasets; and that was expected as normalization boosts training as well as results. Even though the accuracy achieved by the normalized dataset is better than that of the unnormalized dataset, we have used the unnormalized dataset to avoid an issue we have faced "DMA timeout error" on the board while using the normalized dataset.

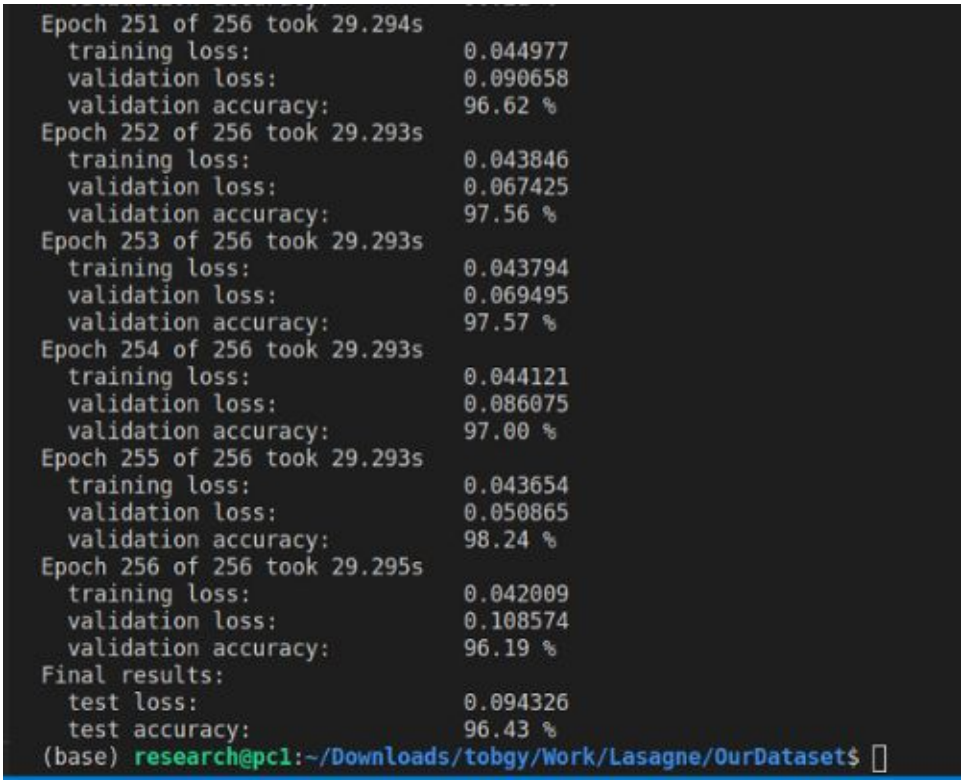


Figure [26]: Last Six Epochs Using Normalized Dataset

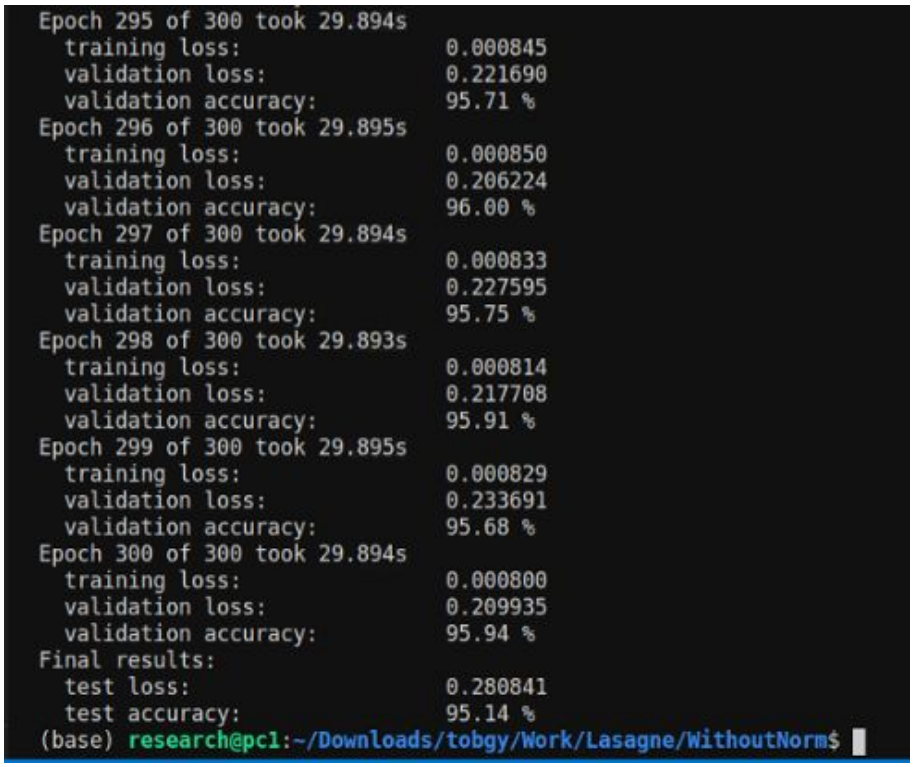


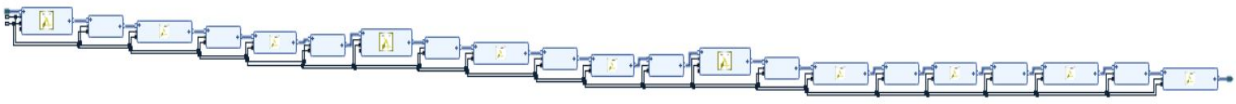
Figure [27]: Last Six Epochs Using Unnormalized Dataset

Hardware

PS

In this part, the PS side is represented by the ARM processor on the PYNQ board. This ARM processor will be used mainly for the processing of the leaf image before entering the CNN on our PL side. First of all, the inputted leaf image to the processor will be resized into 32x32 dimensions. Then, the image needed to be transformed from jpg to NumpyArray format [Height, Width, Channel] the same way the whole dataset was, also, transformed. This step is considered to be important as this is the image format that CNN needs from the input layer. In addition to this, the numpy formatted image is associated with indices for diseased or healthy.

PL



Figure[28]: Cascaded CNN layers IP blocks

Why Vivado HLS

The implementation of the CNN on the FPGA required some prerequisite steps; mainly creating a Vivado HLS Block Design. Vivado HLS is a programming environment which allows software engineers to develop a RTL code targeting FPGA as the execution fabric, written in C++ rather than HDL[1]. However, developing an HDL design is a demanding task as it requires a lot of time to create and validate [1]. Figure [29] shows the difference in developing a processor-targeted code and a FPGA targeted code, namely RTL. Developing and optimising an FPGA design flow with RTL definitely demands more time and that's not always available for software engineers and developers. Xilinx Vivado HLS has solved this time and complexity problem for software engineers. As per "Introduction to FPGA Design with Vivado HLS", the Vivado HLS software has facilitated the task of developing an FPGA targeted design by providing the required tool as illustrated in figure [30].

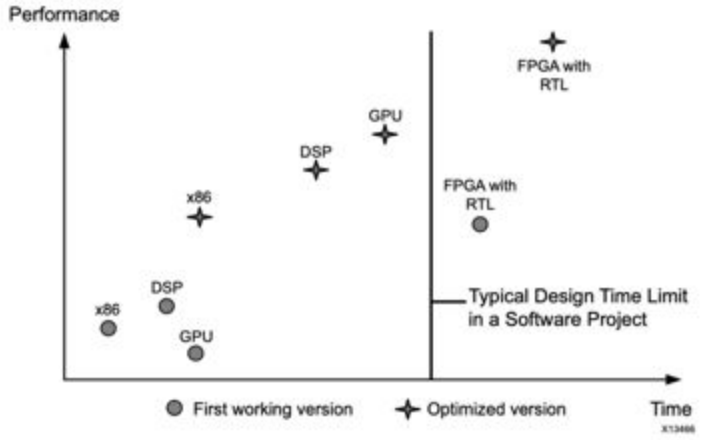


Figure [29]: Performance vs Time (Before HLS)

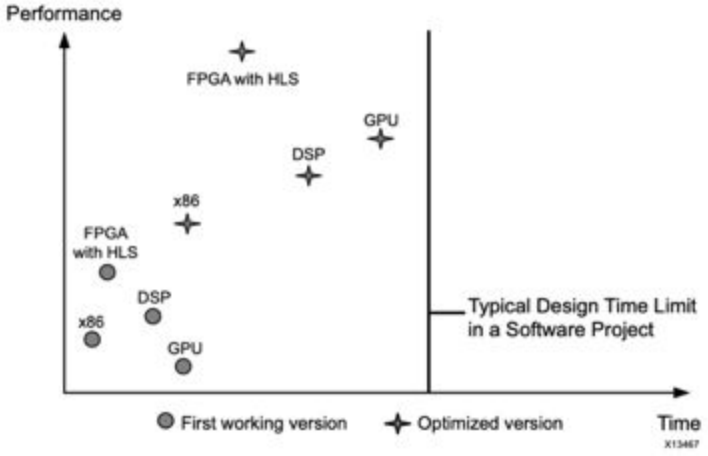


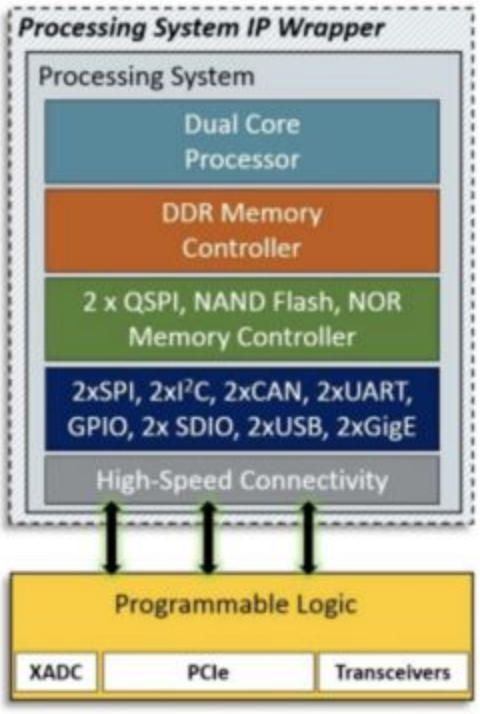
Figure [30]: Performance vs Time (After HLS)

FPGA implementation steps with Vivado HLS

HLS allows users to program FPGAs using high-level languages. It synthesises C++ Codes into RTL implementation. Also, HLS compilers generate IPs for the different design blocks for the developers' code. In PYNQ Classification - Python on Zynq FPGA for Neural Networks, Erwell Wang has developed a framework providing IPs for the CNN architecture that was used in our

project. This has aided in creating the Vivado HLX top level Block design to be mapped onto the PYNQ-Z1's PL. The first step was to regenerate the IPs by synthesising the C++ codes provided by Wang for the different CNN layer into HLS. An IP for each CNN layer was created: Convolution, Pooling, and Fully connected. The ReLU layer was included in each layer IP as the output of each layer should be rectified [2]. A modified implementation of the convolution layer was implemented by Wang in [2]. As described in [2], the layer was divided into 2 sub-block: "sliding window" and normal matrix multiplication. The sliding window technique is basically converting the input feature map into columns to be able to perform matrix multiplication. This implementation increased the efficiency of the normal convolution adopted in CNNs[2]. Lastly, the layers' IPs were cascaded to form the top CNN block IP as shown in figure [28].

The next step was to create the top Block Diagram for the FPGA mapping in Vivado HLX. The CNN top block IP "cifar_10_0" was connected to DMA controllers, AXI4 interconnects, an AXI4 stream switch, and the Xilinx PS IP. The PS IP is basically a wrapper as described in figure [31] that acts as a connection between the SoC's PL and PS [3]. This IP facilitates the integration between the PL Blocks added in the top design and the PS. DMA controllers were added to map the DMA connection between the PS's DDR and the PL. AXI4 interconnects were needed in the top design to support the AXI-DMA protocol between the FPGA (PL) and the DDR in the PS side of the PYNQ. Finally, a bit stream file is generated and is ready to be deployed on the FPGA.



Figure[31]: Xilinx PS Wrapper IP

PS-PL Communication

The flow of the entire project depends on the communication between the PS and PL. As mentioned in the “Standards" section, AXI4 protocol and DMA represent the vital components for the success of the communication between the two sides. Figure [32] illustrates the block diagram of our project and how the PS is connected to the PL. As shown, the ZYNQ processing system (PS) connects to the CNN (cifar_10_0), in the PL side, through an AXI peripheral controller, which supports data transfers between the AXI4 Interface and external synchronous and/or asynchronous peripheral devices[15], an AXI4 interconnect, which connects one or more AXI memory-mapped master devices to one or more slave devices, an AXI4 switch which

allows multiple masters and slave to be connected by using the TDEST signal to route transfers to different slaves[16], and two DMA controllers that. Looking closely at our application; firstly, the image is transferred via the HDMI peripheral to the DDR in the PS side as described above. The image, in numpy format, is then transferred from the DDR to the DMA controller through the AXI4 interconnect. The DMA controller then transfers the image to the CNN through the AXI4 switch. Following the forward propagation, and the network's computation of the output, this output, which is the prediction, is then propagated back to the PS in a similar manner. The output of the CNN is first transferred to the DMA controller through the AXI4 switch. The DMA controller sends it to the PS through the AXI4 interconnect and the prediction is finally processed and displayed notifying the user whether the leaf image is diseased or healthy.

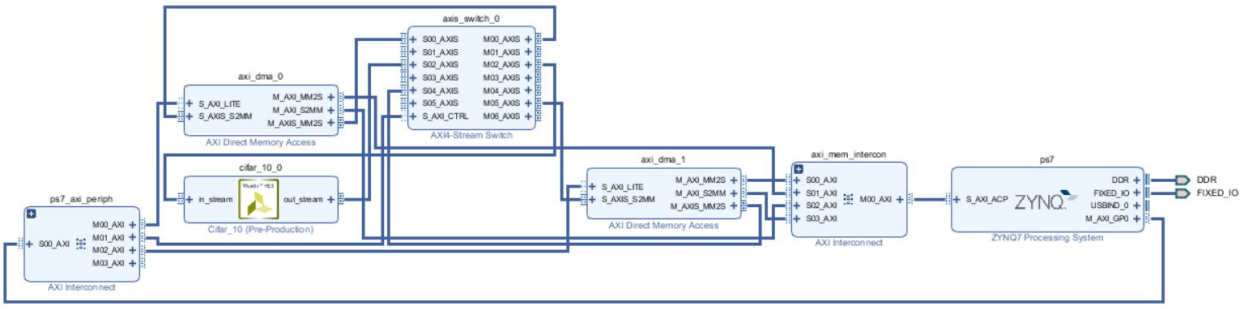


Figure [32]: Top block design

Software-Hardware Python Interface

The interface is written in Python for transferring the information from the processor to the FPGA cores. Initially, we import the required libraries such as Lasagne, numpy, PYNQ drivers, and conv_fpga, which is a script that includes the DMA wrapper; which is responsible for handling the DMA requests. Then, we set up the HDMI driver by using the built-in HDMI functions from the PYNQ overlay. The test image is taken from an HDMI camera and saved on the SD card of the PYNQ board. Additional image processing steps need to be done on the test image taken from a camera since its original dimensions are not compatible with our neural network. Thus, the image gets transposed and reshaped using numpy.

An empty CNN gets instantiated with the exact same parameters as the CNN created on the FPGA in order to load the FPGA's CNN with our trained weights. We used a built-in function from lasagne called "set_all_param_values" to fill the empty CNN with the weights generated by our software training algorithm in npz format. All the parameters are then copied to the FPGA via a function called "FPGAWeightLoader" originating from conv_fpga python script found in Wang's publication [7]. This function takes a certain layer's weights vector and the filter dimensions, creates kernel variable that contains the filter size, prepares this variable to be sent through the DMA with some numpy manipulations, and it calls the hardware execution function from the DMA wrapper class to actually send the weights to the FPGA.

For testing with a test set, the test images and labels are loaded using numpy. The numpy array carrying the images gets transposed for relocating the channels parameter. The function "FPGA_CIFAR10", obtained from Wang's project, takes the input images along with the input

layer dimensions through a Lasagne function called “get_output” [7]. Then, it transfers the input data to the FPGA via the DMA wrapper function. The input data gets processed in the FPGA cores that include the CNN, and the output prediction returns to a variable called “prob”. We use the function “argmax” from numpy that calculates which output class has the maximum probability of having the features of the input data. The accuracy is then determined by comparing the predictions with the test labels stored in a numpy array.

Finally, we deploy the CNN on the ARM processor and run the same test to compare between the elapsed time and accuracy on the FPGA and the processor.

List of Components

FPGA Board

PYNQ-Z1v2.5

The PYNQ-Z1v2.5 board has ZYNQ 7020 FPGA which offers 630 KB of Block RAMS, 220 DSPs, 8 DMA channels and 53200 LUTs. In addition to this, it offers Low-bandwidth peripheral controllers as: SPI, UART, CAN and I2C ;moreover, High-bandwidth peripheral controllers as: 1G Ethernet and USB 2.0. These on-chip resources can accommodate the implementation of our project's CNN which made this board the best fit for the implementation of our project. Moving on from the on-chip resources,generally, the PYNQ-Z1v2 board was considered to be a best fit as it incorporates the PYNQ project that was launched a few years ago by Xilinx. The PYNQ project is a creative framework that allows the instantiation of synthesized FPGA IPs or overlays using Python scripts without the use of RTL languages that complicate the CNN design much more. Since this user-friendly environment runs Python on Linux Ubuntu OS, the CNN layers can be dispatched in parallel to become synthesized PYNQ overlays that can be instantiated in the regular Theano syntax. Furthermore, it is capable of running Jupyter Notebooks that contain built-in functions for the communication through DMA and MMIO, and it uses overlays to configure the hardware of the PL.

NVIDIA GeForce RTX 2080 Ti

There are 544 tensor cores which help improve the speed of machine learning applications which makes it suitable for training our CNN. The card also has 68 ray tracing acceleration cores. NVIDIA has paired 11 GB GDDR6 memory with the GeForce RTX 2080 Ti, which are connected using a 352-bit memory interface. The GPU is operating at a frequency of 1350 MHz, which can be boosted up to 1545 MHz, memory is running at 1750 MHz (14 Gbps effective).

HDMI to HDMI Cable

HP HDMI to HDMI Cable (2ux04aa)

This HDMI to HDMI Cable Supports up to 4K Ultra High Definition signal as well as High Speed 100Mbps, full-duplex ethernet traffic. It is used in our project to connect the FPGA with a laptop, in order to prevent the use of an external camera and use the laptop's camera instead for testing our CNN. In the PYNQ project, a built in overlay called the "Base Overlay" allowed us to use the HDMI Input/Output controllers easily. As stated, this overlay allowed us to capture an image then process it and let it be used by our CNN. This was implemented by running the cells that are shown in figure [33] below. This code ensures that the HDMI is set on to capture mode. Following this, when the capture mode is on, three frames will be captured, by default, and then we can access one frame by providing the frame index either (0 , 1 or 2). Then, this image will be provided as an input image to CNN to detect whether it is diseased or healthy.

HDMI Interface

In [2]:
```
#Initialzing

#Download
Overlay("base.bit").download()
#Initialize HDMI as input
hdmi_in = HDMI("in")
#start connection
hdmi_in.start()
print("Connection succeeded" if (hdmi_in.state() == 1) else "Connection Failed")
print("Dimension of captured frane is: " + str(hdmi_in.frame_width()) + " x " + str(hdmi_in.frame_height()))
```

```
Connection succeeded
Dimension of captured frane is: 1920 x 1080
```

In [5]:
```
#Take frame

frame = hdmi_in.frame()
orig_img_path = r'/home/xilinx/testleaf25Dec_health3.jpg'
frame.save_as_jpeg(orig_img_path)
Image(filename = orig_img_path)
```

Figure [33]: HDMI Code

Design Validation

One of the most vital parts of any project is validating the results several times with various methods. Consequently, validation was a huge part of our project and took a considerable amount of time. Firstly, there was the test set alongside with the validation set verifications done after training the CNN and great results were obtained. More importantly, however, is the validation phase for the project on the SoC itself, the PYNQ. This phase consisted of testing on individual images from the test set, a batch of images and images taken from the HDMI interface. Not only this, but also testing was done on both the accelerated approach and the ARM processor alone; in order to produce the observable impact of the acceleration done. As expected, as found in several literature sources that the accuracy was to decrease a bit, and that happened. Also, the expectation of exponential acceleration in the time of classification was met when the FPGA was used, and the forward propagation for either one image or the batch of 500 images was significantly different between the classical processor-only approach and the FPGA-aided

one. A surprising result, however, was having a higher accuracy for the accelerated FPGA test than the processor one; which was not expected at all as a myriad of sources stated the opposite.

As mentioned, the results of the FPGA accelerated approach proved its effectiveness in comparison to the processor-only approach. Figure [34] shows the results when testing with one diseased image from the test set. Notice the elapsed time how lower it is with a factor of 12,556; which is extremely high, and for other applications such difference in classification might be crucial. As for the batch of 500 images, the result was not as strong when compared to the one image, nevertheless, it was still decent with an acceleration of a factor of 24.65. Nonetheless, the accuracy, surprisingly, was higher with 2.2% as seen in figure [36] and figure [37]. Lastly, was the verification using the HDMI interface in the PYNQ; and for that part, a couple of test leaves were obtained from AUC to experiment with. The interface used was as follows: the HDMI cable is connected from the PYNQ to a laptop and its webcam is used to capture the leaves images. Figures [38] and [39] show the images captured for a diseased leaf and a healthy one. In addition, for both cases the test was done using the FPGA accelerated design and the ARM only; showing significant improvement in time of classification as before. Moreover, the utilization reports for our design were generated in order to evaluate the hardware's performance. Adding to this, the power report showed that the usage of FPGA resulted in low power consumption of 2.189 W.

FPGA Deployment

In [6]:
```
FPGA_net = {}
FPGA_net['input'] = InputLayer((None, 3, 32, 32))
FPGA_net['cifar10'] = FPGA_CIFAR10(FPGA_net['input'])
FPGA_net['prob'] = NonlinearityLayer(FPGA_net['cifar10'], softmax)
```

In [7]:
```
#FPGA Test Time
batch_size = 1

%time prob = lasagne.layers.get_output(FPGA_net['cifar10'], floatX(image*64), deterministic=True)#.eval()
FPGA_predicted = np.argmax(prob, 1)

print("Plant is Healthy" if FPGA_predicted == 0 else "Plant is Diseased")
```

```
Elapsed Test Time:  0.0002640180000028636
CPU times: user 1.31 s, sys: 880 ms, total: 2.19 s
Wall time: 2.26 s
Plant is Diseased
```

Figure [34]: Testing A Diseased Image From Test Set On FPGA

ARM Deployment

In [8]:
```
batch_size = 1
import time
start_time = time.process_time()
prob = np.array(lasagne.layers.get_output(net['prob'], floatX(image), deterministic=True).eval())
predicted = np.argmax(prob, 1)
end_time = time.process_time()
print("Elapsed Test Time: ", end_time-start_time)
print("Plant is Healthy" if predicted == 0 else "Plant is Diseased")
```

```
Elapsed Test Time:  3.315167199000001
Plant is Diseased
```

Figure [35]: Testing A Diseased Image From Test Set On ARM Only

FPGA Deployment

In [6]:
```
FPGA_net = {}
FPGA_net['input'] = InputLayer((None, 3, 32, 32))
FPGA_net['cifar10'] = FPGA_CIFAR10(FPGA_net['input'])
FPGA_net['prob'] = NonlinearityLayer(FPGA_net['cifar10'], softmax)
```

In [7]:
```
batch_size = 500

%time prob = lasagne.layers.get_output(FPGA_net['cifar10'], floatX(data['raw'][0:batch_size]*64), deterministic=True)#.eval()
FPGA_predicted = np.argmax(prob, 1)
```

```
Elapsed Test Time:  3.1895250819999994
CPU times: user 7.61 s, sys: 320 ms, total: 7.93 s
Wall time: 9.37 s
```

In [8]:
```
#Accuracy
FPGA_accuracy = np.mean(FPGA_predicted == data['labels'][0:batch_size])
print("Accuracy on " + str(batch_size) + " images is: "+str(FPGA_accuracy * 100) + "%")
```

```
Accuracy on 500 images is: 92.4%
```

Figure [36]: Testing 500 Images From Test Set On FPGA

ARM Deployment

In [9]:
```
batch_size = 500
import time
start_time = time.process_time()
prob = np.array(lasagne.layers.get_output(net['prob'], floatX(data['raw'][0:batch_size]), deterministic=True).eval())
predicted = np.argmax(prob, 1)
end_time = time.process_time()
print("Elapsed Test Time: ", end_time-start_time)
```

```
Elapsed Test Time:  78.41888247
```

In [10]:
```
#Accuracy
accuracy = np.mean(predicted == data['labels'][0:batch_size])
print("Accuracy on " + str(batch_size) + " images is: "+str(accuracy * 100) + "%")
```

```
Accuracy on 500 images is: 90.2%
```

Figure [37]: Testing 500 Images From Test Set On ARM Only

```
In [3]: #Take frame

        frame = hdmi_in.frame()
        orig_img_path = r'/home/xilinx/testleaffinalDec_health5.jpg'
        frame.save_as_jpeg(orig_img_path)
        Image(filename = orig_img_path)
```

Out[3]:

Figure [38]: Diseased Leaf Captured From Laptop Webcam

```
In [3]: #Take frame

        frame = hdmi_in.frame()
        orig_img_path = r'/home/xilinx/healthyorange_2333.jpg'
        frame.save_as_jpeg(orig_img_path)
        Image(filename = orig_img_path)
```

Out[3]:

Figure [39]: Healthy Leaf Captured From Laptop Webcam

FPGA Deployment

In [6]:
```
FPGA_net = {}
FPGA_net['input'] = InputLayer((None, 3, 32, 32))
FPGA_net['cifar10'] = FPGA_CIFAR10(FPGA_net['input'])
FPGA_net['prob'] = NonlinearityLayer(FPGA_net['cifar10'], softmax)
```

In [7]:
```
#FPGA Test Time
batch_size = 1

%time prob = lasagne.layers.get_output(FPGA_net['cifar10'], floatX(image*64), deterministic=True)#.eval()
FPGA_predicted = np.argmax(prob, 1)

print("Plant is Healthy" if FPGA_predicted == 0 else "Plant is Diseased")
```

```
Elapsed Test Time:  0.0002611530000038442
CPU times: user 1.37 s, sys: 940 ms, total: 2.31 s
Wall time: 2.76 s
Plant is Diseased
```

Figure [40]: Testing A Captured Diseased Image From Test Set On FPGA

ARM Deployment

In [8]:
```
batch_size = 1
import time
start_time = time.process_time()
prob = np.array(lasagne.layers.get_output(net['prob'], floatX(image), deterministic=True).eval())
predicted = np.argmax(prob, 1)
end_time = time.process_time()
print("Elapsed Test Time: ", end_time-start_time)
print("Plant is Healthy" if predicted == 0 else "Plant is Diseased")
```

```
Elapsed Test Time:  3.306409359
Plant is Diseased
```

Figure [41]: Testing A Captured Diseased Image From Test Set On ARM Only

FPGA Deployment ¶

In [6]:
```
FPGA_net = {}
FPGA_net['input'] = InputLayer((None, 3, 32, 32))
FPGA_net['cifar10'] = FPGA_CIFAR10(FPGA_net['input'])
FPGA_net['prob'] = NonlinearityLayer(FPGA_net['cifar10'], softmax)
```

In [9]:
```
#FPGA Test Time
batch_size = 1

%time prob = lasagne.layers.get_output(FPGA_net['cifar10'], floatX(image*64), deterministic=True)#.eval()
FPGA_predicted = np.argmax(prob, 1)

print("Plant is Healthy" if FPGA_predicted == 0 else "Plant is Diseased")
```

```
Elapsed Test Time:  0.0004635290000101727
CPU times: user 220 ms, sys: 190 ms, total: 410 ms
Wall time: 209 ms
Plant is Healthy
```

Figure [42]: Testing A Captured Healthy Image From Test Set On FPGA

ARM Deployment

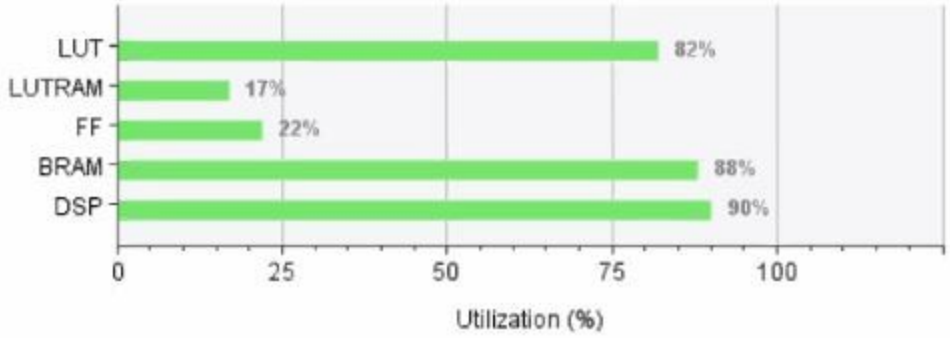
In [8]:
```
batch_size = 1
import time
start_time = time.process_time()
prob = np.array(lasagne.layers.get_output(net['prob'], floatX(image), deterministic=True).eval())
predicted = np.argmax(prob, 1)
end_time = time.process_time()
print("Elapsed Test Time: ", end_time-start_time)
print("Plant is Healthy" if predicted == 0 else "Plant is Diseased")
```

```
Elapsed Test Time:  3.311977517999999
Plant is Healthy
```

Figure [43]: Testing A Captured Healthy Image From Test Set On ARM Only

Summary

Resource	Utilization	Available	Utilization %
LUT	43706	53200	82.15
LUTRAM	2919	17400	16.78
FF	23565	106400	22.15
BRAM	122.50	140	87.50
DSP	198	220	90.00



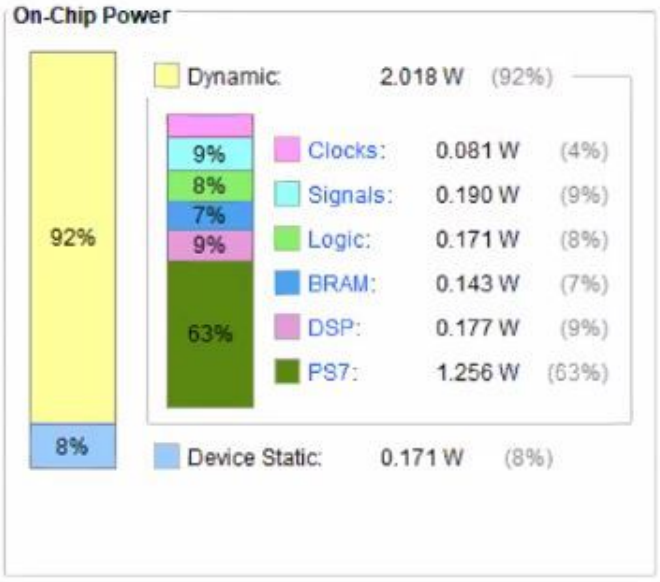
Figure[44]: Utilization Summary After Synthesis in Vivado HLX

Summary

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**2.189 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**50.3°C**
Thermal Margin:	34.7°C (2.9 W)
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity



Figure[45]: Power Summary After Synthesis in Vivado HLX

Feasibility and Economics

A feasibility study was needed to be done to ensure that our product is viable in numerous aspects to be discussed, and to finally determine whether the investments to be made would be worth it or not.

The first type of feasibility needed to be studied would be the technical feasibility, which is an assessment that focuses on the technical resources available. This study delineates whether the ideas that were set would be successfully converted using the technical resources provided. In our project, this study is obviously successful since through the overall design using all the resources that we had the opportunity to acquire have been efficiently used to obtain the results that were aspired as previously shown in the design validation section.

Moreover, another type of feasibility study is the economic feasibility. Through this study, a systematic cost/benefit analysis is done for the project, to aid in calculating the actual economic benefits and whether it deserves the financial funds that would be allocated for it. This study is extremely crucial for future investors to know how credible the project is and whether they should invest in this product or not. Our product excels greatly in this study as well, since the only costly component that is used is the PYNQ-Z1v2.5 board. Other components such as the HDMI cable, laptop camera and processor are components that may be considered to already belong to most of those who will aspire to invest in the product. Thus, it will bring all the benefits as discussed before at a relatively low cost.

Thirdly, there is the legal feasibility, this type scrutinizes mainly the legal requirements that the product may conflict, such as zoning laws, social media laws or data protection acts. Our product does not currently interfere with any legal issues, the only thing is that the images that are to be taken of a certain plant or area should be subject to the approval of the owner. However, for

future work, if such a product is enhanced for it to be installed on a drone, there is a lot of legal paperwork that would need to be done to officially use it.

The following type of feasibility study is the operational type, in this type the study that is made is to ensure that the product does satisfy the needs and goals that were set. The product's goals were set from the start, and they were fulfilled. The product as shown in the design validation section classify whether a leaf is healthy or diseased, which was the goal from the start to design an FPGA accelerated CNN that classifies whether a leaf is healthy or diseased to aid in the identification process of plant diseases that may prove to be a huge threat in field agriculture.

Furthermore, the fifth and final type of feasibility study was the scheduling feasibility. This assessment is extremely vital since it designs the whole timeline that will decide whether the project will be completed in the specific timeframe given or not. For our product, from last semester we had proposed a timeline that we would follow, however alterations were made to accommodate for the situation that we were in during this pandemic, and the project has successfully been finished on time. The specific timeline may be seen in the Gantt Chart section.

The feasibility study for the product was imperative to identify any constraints the product may face ranging from technical aspects, to financial limitations or to legal restrictions. It sparks a crucial factor for potential sponsors which is the credibility of the product.

Societal and Environmental Considerations

One of the main aims which we focused on was to ensure the sustainability for our product in terms of the Social and Environmental aspects. Sustainability in electronics encompasses many

aspects, as it entails both the entire system and the low level “design” perspective. Most designers now are trying to always follow the Design For Environment (DFE) Framework which entails eight main eight axioms[10]:

1. Manufacture without producing hazardous waste
2. Use clean technologies
3. Reduce product chemical emissions
4. Reduce product energy consumption
5. Use non-hazardous recyclable materials
6. Use recyclable material and reuse components
7. Design for ease of disassembly
8. Product reuse or recycle at end of life.

When designing our product, we wanted to ensure that we covered as many of the above axioms as possible. We managed not to include any materials with hazardous waste or chemical emissions. Moreover, the usage of FPGA contributes to the DFE design massively due to the possibility of its firmware reuse.

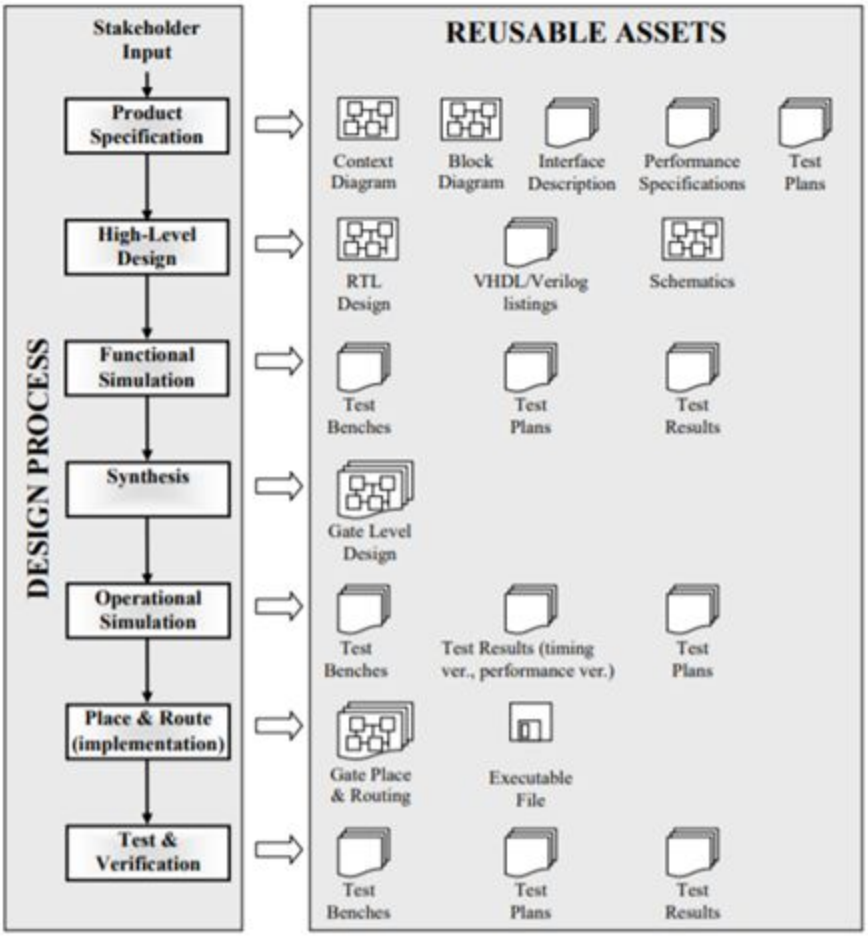


Figure [46]: Reusable Assets of FPGA

As shown in figure (46), FPGA has numerous assets that can be reused which consequently saves a lot of power that is lost in developing and testing new firmware and assets[9]. Furthermore, the software used as well, will be reused without any need for modifications which would also save a lot of power and energy such as training the neural network using the GPU which ran continuously for two and half hours. Moreover, the PYNQ Z1 board does not require, in general, high power and is relatively power and energy efficient. It normally uses a 12 VDC supplied through the Micro-USB port (J8), an external power supply, or a battery.

In addition to the hardware and design specification, our project aims to deliver a sustainable and environment-friendly product that would cater to the well-being of agriculture fields. This would consequently cater enormously to the well being of the environment and reduce the need of using harmful chemicals that prevent pests and plant diseases. Furthermore, our project with extra enhancements, would remove the need for humans to carry out the hectic job of ensuring the lack of diseases among crops themselves and efficiency after having our product implemented, would increase greatly. Thus, not only will this product ensure higher productivity, but it will also enhance numerous social aspects such as human health and human welfare due to the early and accurate detection of plants' diseases. Accordingly having our project sustainable both, socially and environmentally was one of our top priorities.

Operational and Safety Considerations

Another focus that the team needed to shed light on were the operational and safety considerations for the project. These considerations mainly revolve around identifying the safety risks on both humans and the board that may appear as a result of the operational environment of the product.

The first consideration that needs to be taken in regards to the PYNQ-Z1 board is its power supply. The Digilent USB-JTAG-UART port (J14) can provide the power supply needed to the board; it may also use another source of power such as an external power supply or a battery. However, there are some certain specifications that need to be followed. Near the power switch there is a jumper (JP5) that regulates the type of source to be used.

0.5A is the maximum that the USB 2.0 port can provide, this would be perfect for designs that are of low complexity. However, there are some more demanding functions that may use more than what the port can carry, so what occurs is that the consumption of power would increase until the limit of the current. Following that, when the voltage decreases beneath the nominal value, the power-on reset signal is activated as a form of safety so that the ZYNQ is reset.

There are applications that may need to run without being connected to the PC USB port, thus it is best to use an external power supply. To do this, power jack (J18) is plugged in and jumper JP5 is set to "REG". The used plug should deliver, in DC voltage, 7V to 15 V, and must be coax of 2.1 mm center positive internal diameter. Using any voltage above 15V can cause permanent damage, thus it is imperative to consider this.

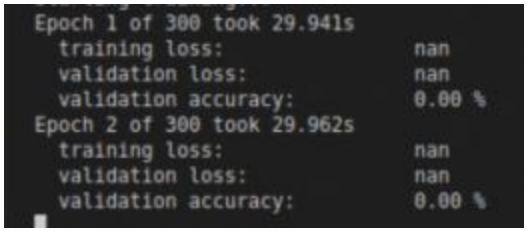
Moreover, another source is by using a battery. It should be connected to the shield connector, and jumper JP5 is also set to "REG", on J7 pin "VIN" is connected to the battery's positive terminal and on J7 pin "GND" is connected to the negative terminal. All the methods stated above need to be followed carefully to ensure that there are no risks on the board due to the power supply.

Another aspect that needs to be well taken care of is the temperature of the PYNQ Z1 board. The maximum operating temperature of the board is 85 degrees Celsius, thus it is not the easiest for the board to be damaged due to overheating, but it is still possible. Usually, chips are best protected by using fans and heat sinks. These are the best options to ensure the safety of the board, but it is extremely vital to make sure that what is installed is installed properly to avoid damaging the chip. The temperature sensor graph that is on board may be used to keep an eye on the temperature so that if it is needed to reset the board. It is important to store the board in a

moderate room temperature, since if it is stored in a hot weather already before powering it up will increase the risk on it. In our application, agricultural weathers are very high in temperature especially in the summer, so it is essential to install a fan or heat sink to protect the board as much as possible and to utilize the temperature sensor graph to monitor the temperature.

Ethical Considerations

As our team was testing the CNN and its accuracy in detecting the diseased leaves and the healthy leaves,the ethical standards that the team should abide by, were taken into consideration during the implementation phase of our project. For this reason, all the results from our CNN were documented in this paper even though some of these results showed the inaccuracy of our CNN's Classification during some of the testing phases. Some of these results are, also, shown in the figure below presenting the achieved accuracies of our network in some of the performed tests. This is a crucial part that needs to be added to our paper as it shows that ethical considerations need to be taken into account regardless of the type of the field of any ongoing research



Figure[47]: NAN Results

Implementation Gantt Chart

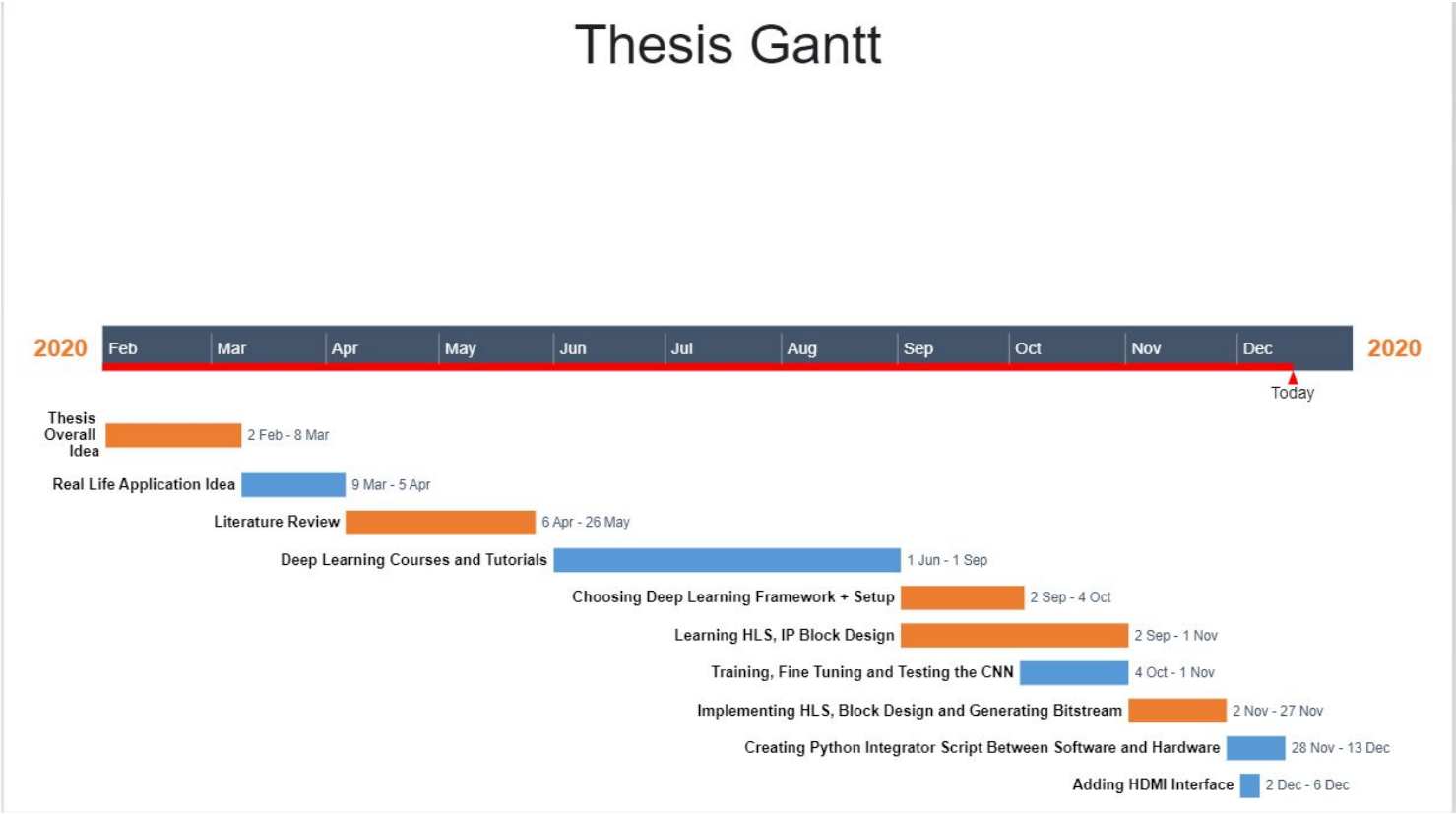


Figure [48]: Gantt Chart of the Project

Conclusion

In this project, an application is developed in order to classify the leaves into diseased leaves or healthy leaves. This is achieved by utilizing the PS and PL sides of the PYNQ board. As the CNN is deployed on the FPGA inorder to accelerate the CNN as well as boosting its performance. For the implementation of this project, different models were being trained on different frameworks such as TensorFlow, at first, then Theano (used in our project). This allowed us to explore the use of different frameworks. Furthermore, the report shows how there are different standards of communication as HDMI, Ethernet, UART, AXI and DMA , for instance, that our team came across and facilitated the communication between the PS and the PL as well as capturing images and inputting these images to the CNN. At last, the CNN is validated using testing images as well as validation images. As for the PL side, HLS is used to generate some of the RTL blocks. Then, these RTL blocks are wrapped and packaged to be used in this project. This is eventually considered to act as an initial step in producing a commercial hardware used in agricultural fields.

Future Work and Recommendations

There is no project with the possibility of enhancements to it; and this project is no exception. In the upcoming lines a few suggestions will be stated for improvement to the project.

Further Training

For such an application, extensive training is required in order to be practical. Like facial recognition systems it needs to be trained with datasets that are enormous in size. In our case, as mentioned, the training was done with only 58000 images; thus, the classifier is not practical to be used as a classifier system in a farm or a crop field and needs more training. In addition, the CNN needs training on different types of leaves datasets as well to have the CNN trained even better for the practical world.

Bigger CNN

Another vital improvement is using new CNN architectures with more number of layers. The reason for that is when more layers are implemented in the design, accuracy usually increases. Popular CNN architectures like ResNet, GoogleNet and AlexNet might be a decent fit for a classifier like ours. However, in our case, we could not use any of these or even bigger CNN architecture from scratch; because of the board sources limitation in the PYNQ. Furthermore, the CNN architecture can be enhanced to accommodate more classes and it could also label the leaf's type; which would be a greater help for landowners and farmers.

Better Board

As stated in the previous recommendation, not using a big CNN was because of the sources limitation of the PYNQ board. Consequently, using a bigger and more expensive board might have made greater results in terms of classification delay and accuracy. That is because the PYNQ had sources such as number of LUTs for example significantly lower than a lot of boards used in the practical systems. Thus, using expensive boards like Xilinx Virtex-7 FPGA VC7215 will definitely lead to exponentially better results in terms of accuracy and delay. The Virtex-7, which is priced at 12,000$, that price is 60 times more expensive than our PYNQ priced at 200$ shows the difference in the performance of each.

Camera

In order to achieve a more practical system, a camera connected to the board which takes the images instead of the HDMI interface would be better indeed. PYNQ supports USB interfaces; so, a USB connected camera might be an improvement. Also, a HDMI connected camera might be even better as it will be replaced with our already-implemented interface with the laptop's webcam.

Drone

This system is required to be working in a crop field categorizing whether plants are healthy or not with nearly real-time delay. Capturing the images of the leaves using a camera will be better using a drone roaming the field and feeding in the image data to CNN. This improvement to the system might be the key for commercializing it as a product to be used heavily in farmlands; however, it needs further work.

References

[1] Schmidhuber, Jürgen. “Deep Learning in Neural Networks: An Overview.” *Neural Networks*, vol. 61, 2015, pp. 85–117., doi:10.1016/j.neunet.2014.09.003.

[2] Kay, Alexx. “Artificial Neural Networks.” *Computerworld*, Computerworld, 12 Feb. 2001, www.computerworld.com/article/2591759/artificial-neural-networks.html.

[3] Krishna, M Manoj, et al. “Image Classification Using Deep Learning.” *International Journal of Engineering & Technology*, vol. 7, no. 2.7, 2018, p. 614., doi:10.14419/ijet.v7i2.7.10892.

[4] Q. Li, W. Cai, X. Wang, Y. Zhou, D. D. Feng and M. Chen, "Medical image classification with convolutional neural network," *2014 13th International Conference on Control Automation Robotics & Vision (ICARCV)*, Singapore, 2014, pp. 844-848, doi: 10.1109/ICARCV.2014.7064414.

[5] Hou, Le, et al. “Patch-Based Convolutional Neural Network for Whole Slide Tissue Image Classification.” *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, doi:10.1109/cvpr.2016.266.

[6] Sharma, Pulkit. “Deep Learning Frameworks: Best Deep Learning Frameworks.” *Analytics Vidhya*, 12 May 2020, www.analyticsvidhya.com/blog/2019/03/deep-learning-frameworks-comparison/

[7] Wang, Erwei, et al. “A PYNQ-Based Framework for Rapid CNN Prototyping.” 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2018, doi:10.1109/fccm.2018.00057.

[8] Murphy, John. “Deep Learning Frameworks: A Survey of TensorFlow, Torch, Theano, Caffe, Neon, and the IBM Machine Learning Stack.” *Microway*, 8 Aug. 2019, www.microway.com/hpc-tech-tips/deep-learning-frameworks-survey-tensorflow-torch-theano-caffe-neon-ibm-machine-learning-stack/.

[9] Jimenez, Nico. *Tensorflow* , 8 Oct. 2017, nicodjimenez.github.io/2017/10/08/tensorflow.html.

[10] Al-Rfou, Rami, et al. "Theano: A Python framework for fast computation of mathematical expressions." *arXiv preprint, 2016, arXiv:1605.02688.*

[11] Monkam, Patrice, et al. "CNN models discriminating between pulmonary micro-nodules and non-nodules from CT images." BioMed Eng OnLine 17, 96, 2018. https://doi.org/10.1186/s12938-018-0529-x

[12] Tan, Jen Hong, et al. "Age-Related Macular Degeneration Detection Using Deep Convolutional Neural Network." *Future Generation Computer Systems*, North-Holland, 29 May 2018, www.sciencedirect.com/science/article/abs/pii/S0167739X17319167.

[13] Adedeji, Olugboja, and Zenghui Wang. "Intelligent Waste Classification System Using Deep Learning Convolutional Neural Network." *Procedia Manufacturing*, Elsevier, 14 Aug. 2019, www.sciencedirect.com/science/article/pii/S2351978919307231.

[14] Toda, Yosuke, and Fumio, Okura. "How Convolutional Neural Networks Diagnose Plant Disease." *Plant Phenomics*, AAAS, 26 Mar. 2019, spj.sciencemag.org/plantphenomics/2019/9237136/.

[15] Mohanty, Sharada, et al. "Using Deep Learning for Image-Based Plant Disease Detection." Frontiers, Frontiers, 6 Sept. 2016, www.frontiersin.org/articles/10.3389/fpls.2016.01419/full.

[16] Tetila, Everton, et al. "Automatic R[ecognition of Soybean Leaf Diseases Using UAV Images and Deep Convolutional Neural Networks." *IEEE Geoscience and Remote Sensing Letters*, 2019. PP. 1-5. 10.1109/LGRS.2019.2932385.

[17] Bouroubi, Yacine, et al. "Pest Detection on UAV Imagery using a Deep Convolutional Neural Network." *In Proceedings of the 14th International Conference on Precision Agriculture*. Montreal, Quebec, Canada, June 24-27, 2018, https://www.ispag.org/proceedings/?action=download&item=5136

[18] S. I. Venieris and C. Bouganis, "fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs," *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Washington, DC, 2016, pp. 40-47, doi: 10.1109/FCCM.2016.22.

[19] Umuroglu, Yaman, et al. “Finn.” *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA 17*, 2017, doi:10.1145/3020078.3021744.

[20] Qiu, Jiantao, et al. “Going Deeper with Embedded FPGA Platform for Convolutional Neural Network.” *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA 16*, 2016, doi:10.1145/2847263.2847265.

[21] Stone, John E., et al. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems.” *Computing in Science & Engineering*, vol. 12, no. 3, 2010, pp. 66–73., doi:10.1109/mcse.2010.69.

[22] Xilinx. Pynq.

[23] Ben Cope et al. Implementation of 2d convolution on fpga, gpu and cpu.

[24] “Arty Z7-20: SoC Zynq®-7000 Development Board for Makers and Hobbyists.” Xilinx.

[25] “Jetson TK1 Development Pack 1.0.” NVIDIA Developer, 25 Apr. 2017.

[26] Boncelet, Charles, et al. “PYNQ-Z1: Python Productivity for Zynq-7000 ARM/FPGA SoC.” Digilent.

[27] “Python Productivity for Zynq.” PYNQ.

[28] “PYNQ Introduction.” PYNQ Introduction - Python Productivity for Zynq (Pynq).

[29] Le, Tung D., et al. “Involving CPUs into Multi-GPU Deep Learning.” *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, doi:10.1145/3184407.3184424.

[30] Yangqing Jia. Learning semantic image representations at a large scale. 2014

[31] Li, Jingjun, et al. “An Experimental Study on Deep Learning Based on Different Hardware Configurations.” *2017 International Conference on Networking, Architecture, and Storage (NAS)*, 2017, doi:10.1109/nas.2017.8026843.

[32] Krizhevsky, Alex, et al. “ImageNet Classification with Deep Convolutional Neural Networks.” *Communications of the ACM*, vol. 60, no. 6, 2017, pp. 84–90., doi:10.1145/3065386.

[33] Nurvitadhi, Eriko, et al. “Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC.” *2016 International Conference on Field-Programmable Technology (FPT)*, 2016, doi:10.1109/fpt.2016.7929192.

[34] Tsai, Tsung-Han, and Shih-Wei Chen. “Single-Chip Design for Intelligent Surveillance System.” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 9, 2018, pp. 1637–1646., doi:10.1109/tvlsi.2018.2827385.

[35] Andri, Renzo, et al. “YodaNN: An Ultra-Low Power Convolutional Neural Network Accelerator Based on Binary Weights.” *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2016, doi:10.1109/isvlsi.2016.111.

[36] William Dally. High-Performance Hardware for Machine Learning. Tutorial, NIPS, 2015

[37] Xilinx. Deep learning with int8 optimization on xilinx devices. 2016.

[38] Warden, Pete. “Pete Warden`s Blog.” *Pete Warden`s Blog*, 20 Apr. 2015, https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning

[39] Zhang, Li, et.al. “Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks”. 2015

[40] Chetlur, CliffWoolley, et.al. “cudnn: Efficient primitives for deep learning”. 2014

[41] Meloni, Deriu, et.al. “International Conference on ReConFigurable Computing and FPGAs (ReConFig).” *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2016.

[42] Hong, Park, et.al. "A 1.9nJ/pixel embedded deep neural network processor for high speed visual attention in a mobile vision recognition SoC." *IEEE Asian Solid-State Circuits Conference (A-SSCC)*, 2015.

[43] Sasagawa, Yukihiro and Mori, Atsuhiro. "High-level video analytics PC subsystem using SoC with heterogeneous multi-core architecture." *Symposium on VLSI Circuits (VLSI Circuits), Kyoto, 2015*.

[44] “Caffe.” *Caffe*, caffe.berkeleyvision.org/.

[45] Varma, Subir, and Sanjiv Das. “Deep Learning.” *SCU Web Page of Sanjiv Ranjan Das*, 27 Sept. 2018, srdas.github.io/DLBook/ConvNets.html#introduction.

[46] Zhou, Yiren, et al. “On Classification of Distorted Images with Deep Convolutional Neural Networks.” *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017, doi:10.1109/icassp.2017.7952349.

[47] Rosebrock, Adrian, et al. “ImageNet: VGGNet, ResNet, Inception, and Xception with Keras.” *PyImageSearch*, 18 Apr. 2020, www.pyimagesearch.com/2017/03/20/imagenet-vggnet-resnet-inception-xception-keras/.

[48] Brintha, V. P., et al. “Automatic Classification of Solid Waste Using Deep Learning.” *SpringerLink*, Springer, Cham, 3 Jan. 2019, link.springer.com/chapter/10.1007/978-3-030-24051-6_83.

[49] “Serial Communication,” Serial Communication - an overview | ScienceDirect Topics. [Online].Available:https://www.sciencedirect.com/topics/computer-science/serial-communication. [Accessed: 19-Dec-2020].

[50] C. Boncelet, G. Bockari, Unknown, L. Rodrígues-Flores, N. Truong, A. Luna, Tim, M. Abdelshakour, and D. Buskirk, “PYNQ-Z1: Python Productivity for Zynq-7000 ARM/FPGA SoC,” *Digilent*. [Online]. Available: https://store.digilentinc.com/pynq-z1-python-productivity-for-zynq-7000-arm-fpga-soc/. [Accessed: 19-Dec-2020].

[51] “PYNQ-Z2 Reference Manual v1.0.” 17-May-2018. https://d2m32eurp10079.cloudfront.net/Download/pynqz2_user_manual_v1_0.pdf. [Accessed: 19-Dec-2020].

[52] *USB Protocol*. [Online]. Available: https://www.keil.com/pack/doc/mw/USB/html/_u_s_b__protocol.html. [Accessed: 19-Dec-2020].

[53] “Introduction to Overlays,” *Introduction to Overlays - Python productivity for Zynq (Pynq) v1.0*. [Online]. Available: https://pynq.readthedocs.io/en/v1.4/6_overlays.html. [Accessed: 19-Dec-2020].

[54] T. A. I. Team, “How, When, and Why Should You Normalize / Standardize / Rescale Your Data?,” *Towards AI - The Best of Tech, Science, and Engineering*, 29-May-2020. [Online]. Available: https://towardsai.net/p/data-science/how-when-and-why-should-you-normalize-standardize-rescale-your-data-3f083def38ff. [Accessed: 19-Dec-2020].

[55] *Python Numpy Tutorial (with Jupyter and Colab)*. [Online]. Available: https://cs231n.github.io/python-numpy-tutorial/. [Accessed: 19-Dec-2020].

[56] Shalabi, Luai. (2006). Coding and Normalization: The Effect of Accuracy, Simplicity, and Training Time.

[57] N. Baradaran, J. Park, and P. C. Diniz, "Data Reuse in Configurable Architectures with RAM Blocks," *Field Programmable Logic and Application Lecture Notes in Computer Science*, pp. 1113–1115, 2004.

[58] "DMA," *DMA - Python productivity for Zynq (Pynq) v1.0*. [Online]. Available: https://pynq.readthedocs.io/en/v2.4/pynq_libraries/dma.html. [Accessed: 19-Dec-2020].

[59] E. Staff, "Introduction to direct memory access," *Embedded.com*, 14-Oct-2003. [Online]. Available: https://www.embedded.com/introduction-to-direct-memory-access/. [Accessed: 19-Dec-2020].

[60] R. Jain, *Gigabit Ethernet*. [Online]. Available: https://www.cse.wustl.edu/~jain/cis788-97/ftp/gigabit_ethernet/index.html. [Accessed: 19-Dec-2020].

[61] C. Boncelet, G. Bockari, Unknown, L. Rodrígues-Flores, N. Truong, A. Luna, Tim, M. Abdelshakour, and D. Buskirk, "PYNQ-Z1: Python Productivity for Zynq-7000 ARM/FPGA SoC," *Digilent*. [Online]. Available: https://store.digilentinc.com/pynq-z1-python-productivity-for-zynq-7000-arm-fpga-soc/. [Accessed: 19-Dec-2020].

[61] "Introduction to Overlays," *Introduction to Overlays - Python productivity for Zynq (Pynq) v1.0*. [Online]. Available: https://pynq.readthedocs.io/en/v1.4/6_overlays.html. [Accessed: 19-Dec-2020].

[62] S. Rockowitz, *DDC vs I2C - ddcutil Documentation*. [Online]. Available: https://www.ddcutil.com/ddc_vs_i2c/. [Accessed: 19-Dec-2020].

[63] "HDMI," *Mepits*. [Online]. Available: https://www.mepits.com/tutorial/142/communication/hdmi. [Accessed: 19-Dec-2020].

[64] "Ecodesign," Ecodesign - an overview | ScienceDirect Topics. [Online]. Available: https://www.sciencedirect.com/topics/engineering/ecodesign. [Accessed: 19-Dec-2020].

[65] "Convolutional Neural Network (CNN) : TensorFlow Core," *TensorFlow*. [Online]. Available: https://www.tensorflow.org/tutorials/images/cnn. [Accessed: 19-Dec-2020].

[66] "TensorFlow - Optimizers," *Tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/tensorflow/tensorflow_optimizers.htm. [Accessed:19-Dec-2020].

[67] "tf.keras.optimizers.Adam : TensorFlow Core v2.4.0," TensorFlow. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam. [Accessed:19-Dec-2020].

[68] "Machine Learning Lesson of the Day – Overfitting and Underfitting," All About Statistics.[Online].Available:https://web.archive.org/web/20161229170858/http://www.statsblogs.com/2014/03/20/machine-learning-lesson-of-the-day-overfitting-and-underfitting/. [Accessed: 19-Dec-2020].

[69] "AXI External Peripheral Controller," Xilinx. [Online]. Available: https://www.xilinx.com/products/intellectual-property/axi_epc.html. [Accessed:19-Dec-2020].

[70] "AXI4-Stream Infrastructure IP Suite v3 - Xilinx." [Online]. Available:

https://www.xilinx.com/support/documentation/ip_documentation/axis_infrastructure_ip_suite/v1_1/pg085-axi4stream-infrastructure.pdf. [Accessed: 19-Dec-2020].

[71] Xilinx.com. 2020. [online] Available at: <https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf> [Accessed 19 December 2020].

[72] Gstitt.ece.ufl.edu. 2020. [online] Available at:

<http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf>

[Accessed 19 December 2020].

[73] Xilinx.com. 2020. [online] Available at: <https://www.xilinx.com/support/documentation/ip_documentation/axis_infrastructure_ip_suite/v1_1/pg085-axi4stream-infrastructure.pdf> [Accessed 19 December 2020].

[74]*Xilinx.com*, 2020. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf. [Accessed: 19- Dec- 2020]

[75] "Zynq-7000 Processing System IP", Xilinx, 2020. [Online]. Available: https://www.xilinx.com/products/intellectual-property/processing_system7.html#:~:text=The%2

0Processing%20System%20IP%20is,the%20Zynq%2D7000%20Processing%20System.&text=The%20Processing%20System%20IP%20Wrapper%20acts%20as%20a%20logic%20connection,the%20Vivado%C2%AE%20IP%20integrator. [Accessed: 19- Dec- 2020]

Appendices

Appendix A: 1 Image Python Script

```
# coding: utf-8

# # Includes

# In[1]:

#-----------------------------------------------------------------Includes-----------------------------------------------
-----------------
import lasagne
from lasagne.layers import InputLayer, DenseLayer, NonlinearityLayer
from lasagne.layers import Conv2DLayer as ConvLayer
from lasagne.layers import Pool2DLayer as PoolLayer
from lasagne.nonlinearities import softmax, rectify, linear
import conv_fpga
from conv_fpga import FPGA_CIFAR10
from conv_fpga import FPGAQuickTest
from conv_fpga import FPGAWeightLoader as FPGALoadW
from lasagne.utils import floatX
import numpy as np
import gzip
import _pickle as cPickle
import matplotlib.pyplot as plt
import time
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
from PIL import Image
from pynq import Overlay
from pynq.drivers.video import HDMI
from IPython.display import Image


# # HDMI Interface
```

```
# In[2]:

#Initialzing

#Download
Overlay("base.bit").download()
#Initialize HDMI as input
hdmi_in = HDMI("in")
#start connection
hdmi_in.start()
print("Connection succeeded" if (hdmi_in.state() == 1) else "Connection Failed")
print("Dimension of captured frane is: " + str(hdmi_in.frame_width()) + " x " + str(hdmi_in.frame_height()))


# In[3]:

#Take frame

frame = hdmi_in.frame()
orig_img_path = r'/home/xilinx/healthyorange_2333.jpg'
frame.save_as_jpeg(orig_img_path)
Image(filename = orig_img_path)


# # Show input image and Resize

# In[2]:

from PIL import Image
directory = r"/home/xilinx/healthyOrange999"
directory2 = r"/home/xilinx/"
NEW_PIXEL_SIZE = 32
EXTENSION = 'jpg'

image = Image.open(directory + "." + EXTENSION)
image = image.resize((NEW_PIXEL_SIZE, NEW_PIXEL_SIZE))
image.save(directory2 + "healthyOrange999." + EXTENSION)

image = mpimg.imread('/home/xilinx/healthyOrange999.jpg')
```

```
plt.figure()
plt.imshow(image)
plt.show()

#One Image Array
image = np.transpose(image, (2,0,1))
image = image.reshape(1,3,32,32)
print(np.shape(image))

#normalizing one image
mean_image = np.full((1,3,32,32), 1.0)
image = image / mean_image
print(image)


# In[14]:

print(image.size)


# # Create the CNN

# In[3]:

net = {}
net['input'] = InputLayer((None, 3, 32, 32))
net['conv1'] = ConvLayer(net['input'], num_filters=32, filter_size=5, pad=2, nonlinearity=None)
net['pool1'] = PoolLayer(net['conv1'], pool_size=2, stride=2, mode='max', ignore_border=False)
net['relu1'] = NonlinearityLayer(net['pool1'], rectify)
net['conv2'] = ConvLayer(net['relu1'], num_filters=32, filter_size=5, pad=2, nonlinearity=rectify)
net['pool2'] = PoolLayer(net['conv2'], pool_size=2, stride=2, mode='average_exc_pad', ignore_border=False)
net['conv3'] = ConvLayer(net['pool2'], num_filters=64, filter_size=5, pad=2, nonlinearity=rectify)
net['pool3'] = PoolLayer(net['conv3'], pool_size=2, stride=2, mode='average_exc_pad', ignore_border=False)
net['ip1'] = DenseLayer(net['pool3'], num_units=64, nonlinearity = None)
net['ip2'] = DenseLayer(net['ip1'], num_units=10, nonlinearity = None)
net['prob'] = NonlinearityLayer(net['ip2'], softmax)


# # Load the Weights
```

```
# In[4]:

with np.load('/home/xilinx/model_NoNorm.npz') as f:
    param_values = [f['arr_%d' % i] for i in range(len(f.files))]
lasagne.layers.set_all_param_values(net['prob'], param_values)


# # Copy the parameters to the FPGA

# In[5]:

#FPGALoadW(weight, status, IFDim, OFDim, PadDim)
weight = net['conv1'].W.get_value()
FPGALoadW(weight, 1, 32, 32, 2)
weight = net['conv2'].W.get_value()
FPGALoadW(weight, 2, 16, 16, 2)
weight = net['conv3'].W.get_value()
FPGALoadW(weight, 3, 8, 8, 2)
weight = net['ip1'].W.get_value()
weight = np.transpose(weight)
weight = weight.reshape(64, 64, 4, 4)
FPGALoadW(weight, 4, 4, 1, 0, flip_filters=False)
weight = net['ip2'].W.get_value()
weight = np.transpose(weight)
weight = weight.reshape(10, 64, 1, 1)
FPGALoadW(weight, 5, 1, 1, 0, flip_filters=False)


# # FPGA Deployment

# In[6]:

FPGA_net = {}
FPGA_net['input'] = InputLayer((None, 3, 32, 32))
FPGA_net['cifar10'] = FPGA_CIFAR10(FPGA_net['input'])

# In[7]:
```

```
#FPGA Test Time
batch_size = 1

get_ipython().magic("time prob = lasagne.layers.get_output(FPGA_net['cifar10'], floatX(image*64),
deterministic=True)#.eval()")
FPGA_predicted = np.argmax(prob, 1)

print("Plant is Healthy" if FPGA_predicted == 0 else "Plant is Diseased")


# # ARM Deployment

# In[8]:

batch_size = 1
import time
start_time = time.process_time()
prob = np.array(lasagne.layers.get_output(net['prob'], floatX(image), deterministic=True).eval())
predicted = np.argmax(prob, 1)
end_time = time.process_time()
print("Elapsed Test Time: ", end_time-start_time)
print("Plant is Healthy" if predicted == 0 else "Plant is Diseased")
```

Appendix B: 500 Images Python Script

```
# coding: utf-8

# # Includes

# In[1]:
```

```
#--------------------------------------------------------------------Includes-------------------------------------------------
-----------------
import lasagne
from lasagne.layers import InputLayer, DenseLayer, NonlinearityLayer
from lasagne.layers import Conv2DLayer as ConvLayer
from lasagne.layers import Pool2DLayer as PoolLayer
from lasagne.nonlinearities import softmax, rectify, linear
import conv_fpga
from conv_fpga import FPGA_CIFAR10
from conv_fpga import FPGAQuickTest
from conv_fpga import FPGAWeightLoader as FPGALoadW
from lasagne.utils import floatX
import numpy as np
import gzip
import _pickle as cPickle
import matplotlib.pyplot as plt
import time
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
from PIL import Image
from pynq import Overlay
from pynq.drivers.video import HDMI
from IPython.display import Image


# # Load Test Set

# In[2]:

#Loading full dataset
data= {}
data['raw'] = np.load('/home/xilinx/test_dataset_array_noNorm.npy')
data['raw'] = np.transpose(data['raw'],(0,3,1,2))
data['labels'] = np.load('/home/xilinx/test_dataset_array_labels_noNorm.npy')


# # Create the CNN

# In[3]:
```

```
net = {}
net['input'] = InputLayer((None, 3, 32, 32))
net['conv1'] = ConvLayer(net['input'], num_filters=32, filter_size=5, pad=2, nonlinearity=None)
net['pool1'] = PoolLayer(net['conv1'], pool_size=2, stride=2, mode='max', ignore_border=False)
net['relu1'] = NonlinearityLayer(net['pool1'], rectify)
net['conv2'] = ConvLayer(net['relu1'], num_filters=32, filter_size=5, pad=2, nonlinearity=rectify)
net['pool2'] = PoolLayer(net['conv2'], pool_size=2, stride=2, mode='average_exc_pad', ignore_border=False)
net['conv3'] = ConvLayer(net['pool2'], num_filters=64, filter_size=5, pad=2, nonlinearity=rectify)
net['pool3'] = PoolLayer(net['conv3'], pool_size=2, stride=2, mode='average_exc_pad', ignore_border=False)
net['ip1'] = DenseLayer(net['pool3'], num_units=64, nonlinearity = None)
net['ip2'] = DenseLayer(net['ip1'], num_units=10, nonlinearity = None)
net['prob'] = NonlinearityLayer(net['ip2'], softmax)


# # Load the Weights

# In[4]:

with np.load('/home/xilinx/model_NoNorm.npz') as f:
    param_values = [f['arr_%d' % i] for i in range(len(f.files))]
lasagne.layers.set_all_param_values(net['prob'], param_values)


# # Copy the parameters to the FPGA

# In[5]:

#FPGALoadW(weight, status, IFDim, OFDim, PadDim)
weight = net['conv1'].W.get_value()
FPGALoadW(weight, 1, 32, 32, 2)
weight = net['conv2'].W.get_value()
FPGALoadW(weight, 2, 16, 16, 2)
weight = net['conv3'].W.get_value()
FPGALoadW(weight, 3, 8, 8, 2)
weight = net['ip1'].W.get_value()
weight = np.transpose(weight)
weight = weight.reshape(64, 64, 4, 4)
FPGALoadW(weight, 4, 4, 1, 0, flip_filters=False)
```

```
weight = net['ip2'].W.get_value()
weight = np.transpose(weight)
weight = weight.reshape(10, 64, 1, 1)
FPGALoadW(weight, 5, 1, 1, 0, flip_filters=False)



# # FPGA Deployment

# In[6]:

FPGA_net = {}
FPGA_net['input'] = InputLayer((None, 3, 32, 32))
FPGA_net['cifar10'] = FPGA_CIFAR10(FPGA_net['input'])

# In[7]:

batch_size = 500

get_ipython().magic("time prob = lasagne.layers.get_output(FPGA_net['cifar10'],
floatX(data['raw'][0:batch_size]*64), deterministic=True)#.eval()")
FPGA_predicted = np.argmax(prob, 1)



# In[8]:

#Accuracy
FPGA_accuracy = np.mean(FPGA_predicted == data['labels'][0:batch_size])
print("Accuracy on " + str(batch_size) + " images is: "+str(FPGA_accuracy * 100) + "%")



# # ARM Deployment

# In[9]:

batch_size = 500
import time
start_time = time.process_time()
prob = np.array(lasagne.layers.get_output(net['prob'], floatX(data['raw'][0:batch_size]),
deterministic=True).eval())
```

```
predicted = np.argmax(prob, 1)
end_time = time.process_time()
print("Elapsed Test Time: ", end_time-start_time)


# In[10]:

#Accuracy
accuracy = np.mean(predicted == data['labels'][0:batch_size])
print("Accuracy on " + str(batch_size) + " images is: "+str(accuracy * 100) + "%")
```