Ministry of Higher Education and Scientific Research

University of Carthage

National Institute of Applied Sciences and Technology

End of Year Project (PFA)

Theme:
Building and Embedding a Diabetic Retinopathy Detection Deep Learning Model on Xilinx® Zynq-7000 SoC ZC702 Evaluation Board

Supervisor: Mr. Slim Ben Saoud

Authors:
Seifeddine Aouay | Berrima Rahma | Ben Achour Ghailen | Galaaoui Salma

Acknowledgements

We would like to express our gratitude to our professor, Mr. Slim BEN SAOUD. We would like to thank him for having supervised, guided, helped and advised us.

We send our sincere thanks to Mrs. Meriem DHOUIBI and Mrs. Afef SAIDI who by their words, their writings, their advice and their critics guided our reflections and who agreed to meet us and answer our questions during the whole period of execution of the project.

Finally, to all these contributors, we present our thanks, our respect and our gratitude.

Contents

List of Figures 3

List of Tables 4

1 Theoretical Study 1

1.1 General Introduction 1

1.1.1 Problematic 1

1.1.2 Deep Learning in Medical Applications 1

1.1.3 Deep Learning with Edge Computing 2

1.2 State of the art 3

1.2.1 Traditional Diagnosis 3

1.2.2 Limitations of Traditional Diagnosis 3

1.2.3 Automated Diabetic Retinopathy Detection 3

1.2.4 Automated Diabetic Retinopathy Detection Limits 4

2 Technical Study 4

2.1 Chosen Solution 4

2.2 Visualization tools 5

2.2.1 History Dictionary 5

2.2.2 Confusion Matrix 6

2.2.3 Netron 6

2.3 Data-set 7

2.4 Maximazing accuracy and minimizing loss 9

2.5 Encountered Problems 11

2.6 Optimization Techniques 12

2.6.1 Pruning 12

2.6.2 Quantization 18

2.7 Accelerator Compatibility 25

2.8 Implementation and Deployment on Hardware 27

2.8.1 Step 1 : Hardware Platform Creation 27

2.8.2 Step 2 : Build a linux OS Platform 35

2.8.3 Step 3 : Optimize model 39

2.8.4 Step 4 : Build an application 44

2.8.5 Step 5 : Boot the application 49

3 Obtained Results and Conclusion 51

3.1 Hardware Specifications: 51

3.2 Interpretations 53

3.3 Perspectives 54

Bibliography 55

List of Figures

1 5 patterns for Diabetic Retinopathy detection . . . 1
2 Kaggle Diabetic Retinopathy Contests . . . 3
3 Kaggle Diabetic Retinopathy Contests(*) . . . 4
4 ResNet architecture . . . 4
5 History Dictionary Example . . . 5
6 Confusion Matrix Example . . . 6
7 Architecture Visualization . . . 7
8 Distribution of the data-set . . . 8
9 Distribution of the data-set(*) . . . 8
10 Data-set sample . . . 9
11 Impact of Learning Rate . . . 10
12 Final History Dictionary . . . 10
13 Final Confusion Matrix . . . 11
14 Original Model Performance . . . 11
15 Layer parameters . . . 12
16 Mask structure and weights after applying it . . . 13
17 Number of parameters (Mask included) . . . 13
18 Number of parameters (Mask stripped) . . . 13
19 Number of zeros in pruned model . . . 14
20 Pruned model size after Compression . . . 14
21 The Channel Pruning Process . . . 14
22 Flow-chart of the Network Slimming procedure . . . 15
23 ResNet-18 architecture . . . 15
24 Basic Block Layers . . . 16
25 Model parameters before and after Compression(50%) . . . 16
26 ConvNet Input Shape[1] . . . 17
27 Model's layers' parameters . . . 17
28 Input shape of the Conv2D layers . . . 18
29 Quantization Outcomes[2] . . . 18
30 Weigh Quantization process . . . 19
31 Weights before and after Quantazation . . . 19
32 Model Performance after Dynamic Range Quantization . . . 20
33 Dynamic Range Quantization 32bit output . . . 20
34 Integer Quantization 8bit output . . . 21
35 Model Performance after Integer Quantization . . . 21
36 Model Performance after Rescaling . . . 21
37 Final Model Performance after reshaping data-set . . . 22
38 TensorFlow Lite quantization spec . . . 23
39 Weights before QAT . . . 23
40 Weights after QAT . . . 24
41 Model after quantization-aware training . . . 24
42 Edge TPU Workflow . . . 25
43 Edge TPU wrong compilation . . . 26
44 Xilinx® Implementation Steps . . . 27
45 Block Design . . . 28
46 PS IP configuration . . . 28
47 PS IP configuration(*) . . . 29
48 AXI_Interconnect IP configuration . . . 29

49 Address Editor . . . 30
50 USB_PULLUP configuration . . . 30
51 Clocking Wizard Output Clocks configuration . . . 31
52 Clocking Wizard configuration . . . 31
53 Power Summary . . . 33
54 Power Summary(*) . . . 34
55 Utilization . . . 34
56 Timing Summary . . . 35
57 Vivado® Hardware Platform requirements[3] . . . 35
58 PS IP Peripheral I/O Configuration . . . 36
59 Root filesystem type . . . 37
60 DECENT Pruning and Quantization Flow . . . 40
61 DNNC Components . . . 40
62 Importing main.cc and model .elf file . . . 45
63 TPU vs. GPU performance with larger batch sizes . . . 53

List of Listings

1 USB_PULLUP port constraint specification . . . 31
2 hier_dpu_clk Hierarchy Verilog description . . . 33
3 Petalinux Project Directory Tree . . . 39
4 freeze_session method . . . 41
5 freeze_model.py main . . . 42
6 Calibration images preprocessing python script . . . 43
7 /calibration images and calibration.txt . . . 43
8 calibration.txt contents . . . 43
9 decent_q.sh . . . 44
10 compile.sh . . . 44
11 DPU Task . . . 47
12 Copying and preprocessing inference images . . . 48
13 TopK method . . . 49

List of Tables

1 Model Performance summary . . . 51

1 Theoretical Study

1.1 General Introduction

1.1.1 Problematic

Diabetic retinopathy is the leading cause of blindness in the working-age population of the developed world. It is estimated to affect around 145 million people. In the process of Diabetic Retinopathy diagnosis, there are at least 5 patterns in the human eye to pay attention to:

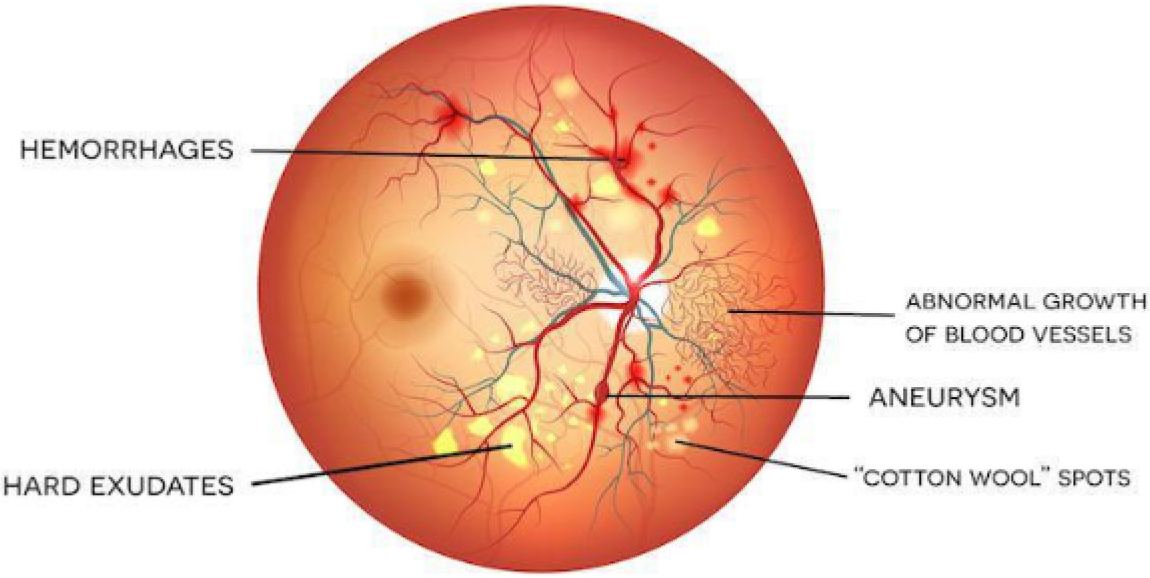


Figure 1: 5 patterns for Diabetic Retinopathy detection

Deep learning is an artificial intelligence function that imitates the workings of the human brain in processing data and creating patterns for use in decision making. Deep learning is a subset of machine learning in artificial intelligence (AI) that has networks capable of learning unsupervised from data that is unstructured or unlabeled.[4]

Deep Learning techniques have been of great assistance to researchers and in healthcare with the use of complex algorithms and software in order to emulate human cognition in the analysis of complicated medical data in order to approximate conclusion without direct human input.

So how can we make use of Deep Learning algorithms and the right devices for deployment in order to make Diabetic Retinopathy diagnosis faster and more accurate?

1.1.2 Deep Learning in Medical Applications

Any ailment in our organs can be visualized by using different modality signals and images, such as EEG, ECG, PCG, X-ray, magnetic resonance imaging, computerized tomography, Single photon emission computed tomography, Positron emission tomography, fundus and ultrasound images, etc., originating from various body parts to obtain useful information. Hospitals are encountering a massive influx of large multi-modality patient data to be analysed accurately and with context understanding.[5]

With their ability to process huge amounts of data in high speed and efficiently discern key features and hidden information in them, Deep Learning algorithms have helped medical staff in the diagnosis of diseases with great accuracy.

1.1.3 Deep Learning with Edge Computing

Deep learning's high accuracy comes at the expense of high computational and memory requirements for both the training and inference phases of deep learning. Training a deep learning model is space and computationally expensive due to millions of parameters that need to be iteratively refined over multiple time periods. Inference is computationally expensive due to the potentially high dimensionality of the input data (e.g., a high-resolution image) and millions of computations that need to be performed on the input data. High accuracy and high resource consumption are defining characteristics of deep learning.
To meet the computational requirements of deep learning, a common approach is to leverage cloud computing. To use cloud resources, data must be moved from the data source location on the network edge [e.g., from smartphones and Internet-of-Things (IoT) sensors] to a centralized location in the cloud. This potential solution of moving the data from the source to the cloud introduces several challenges, namely:

- Latency: Real-time inference is critical to many applications. For example, camera frames from an autonomous vehicle need to be quickly processed to detect and avoid obstacles or a voice-based-assistive application needs to quickly parse and understand the user's query and return a response. However, sending data to the cloud for inference or training may incur additional queuing and propagation delays from the network and cannot satisfy strict end-to-end low-latency requirements needed for real time, interactive applications; for example, real experiments have shown that offloading a camera frame to an Amazon Web Services server and executing a computer vision task take more than 200-ms end-to-end.

- Scalability: Sending data from the sources to the cloud introduces scalability issues, as network access to the cloud can become a bottleneck as the number of connected devices increases. Uploading all data to the cloud is also inefficient in terms of network resource utilization, particularly if not all data from all sources are needed by the deep learning. Bandwidth-intensive data sources, such as video streams, are particularly a concern.

- Privacy: Sending data to the cloud risks privacy concerns from the users who own the data or whose behaviors are captured in the data. Users may be wary of uploading their sensitive information to the cloud (e.g., faces or speech) and of how the cloud or application will use these data. For example, the recent deployment of cameras and other sensors in a smart city environment in New York City incurred serious concerns from privacy watchdogs.[6]

This is where the Edge Computing method comes in handy, it is a viable solution for the latency, scalability and privacy challenges. The principle of this method is to bring computing and information processing resources close to the end device. However while this method provides us with remarkable benefits in terms of speed and privacy, one of its major challenges is the choice of the device in terms of specifications and the deployment of the application (DL model) on the device. This will cause us to compromise between performance and cost.

1.2 State of the art

1.2.1 Traditional Diagnosis

Diabetic retinopathy is usually diagnosed with a comprehensive dilated eye exam. For this exam, drops placed in the patient's eyes widen (dilate) his pupils to allow the doctor to better view inside his eyes. There are two possible exams:

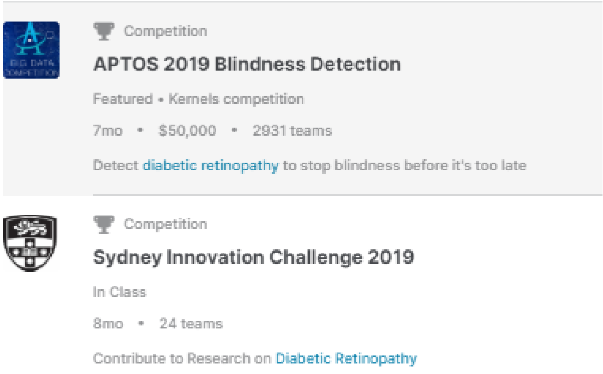
- *Fluorescein Angiography:* The doctor takes pictures of the inside of the patient's eyes. Then he injects a special dye into the patient's arm vein and takes more pictures as the dye circulates through his eyes' blood vessels. He can use the images to pinpoint blood vessels that are closed, broken down or leaking fluid.
- *Optical Coherence Tomography:* This imaging test provides cross-sectional images of the retina that show the thickness of the retina, which will help determine whether fluid has leaked into retinal tissue.[7]

1.2.2 Limitations of Traditional Diagnosis

This method has been the only option for patients but it is time consuming and not always accurate since it relies on the judgment of doctors. There are anomalies that can't be detected by the human eye Currently, detecting DR is a time-consuming and manual process that requires a trained clinician to examine and evaluate digital color fundus photographs of the retina. By the time human readers submit their reviews, often a day or two later, the delayed results lead to lost follow up, miscommunication, and delayed treatment. [8]

1.2.3 Automated Diabetic Retinopathy Detection

The need for a comprehensive and automated method of DR screening has long been recognized, so efforts have been made to use image classification, pattern recognition, and machine learning to give a diagnosis. With color fundus photography as input, the goal was to create an automated detection system to improve DR detection.
Many data scientists have built machine learning models to speed up disease detection and many competitions have been held on the Kaggle website to find more refined models. [8]



Competition

APTOS 2019 Blindness Detection

Featured • Kernels competition

7mo • $50,000 • 2931 teams

Detect diabetic retinopathy to stop blindness before it's too late

Competition

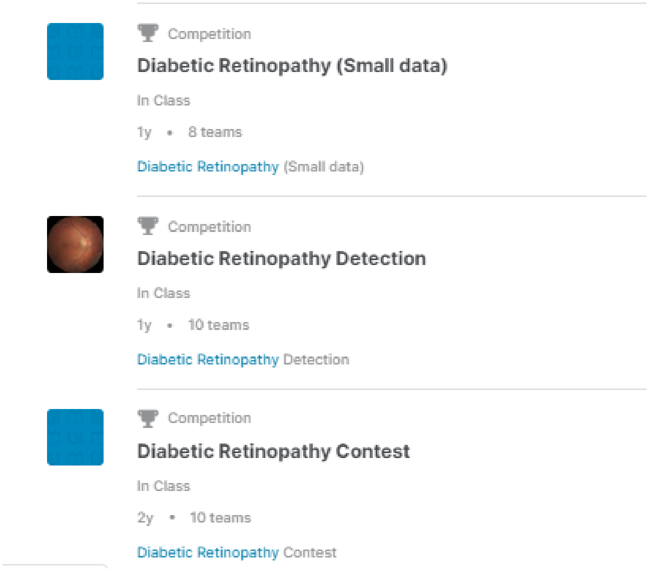
Sydney Innovation Challenge 2019

In Class

8mo • 24 teams

Contribute to Research on Diabetic Retinopathy

Figure 2: Kaggle Diabetic Retinopathy Contests



Competition

Diabetic Retinopathy (Small data)

In Class

1y • 8 teams

Diabetic Retinopathy (Small data)

Competition

Diabetic Retinopathy Detection

In Class

1y • 10 teams

Diabetic Retinopathy Detection

Competition

Diabetic Retinopathy Contest

In Class

2y • 10 teams

Diabetic Retinopathy Contest

Figure 3: Kaggle Diabetic Retinopathy Contests(*)

1.2.4 Automated Diabetic Retinopathy Detection Limits

Many of the existing models are not open source so not all doctors can have access to this service. Plus, this method is also time consuming because the clinician has to take pictures of the retina, manually input the pictures into the model and head back to the patient to explain the diagnosis. Instead, we suggest an FPGA board that has the trained model deployed on it. This will instantly provide the doctor with an answer.

2 Technical Study

2.1 Chosen Solution

Choosing the right architecture for a specific use is an important task.
In fact,in our case ResNet may seem the ideal choice since it is highly reliable when processing microscopic/complicated images.
We have chosen the ResNet-50 architecture since it deals with the Vanishing Gradient Problem. The core idea of ResNet is introducing a so-called “identity shortcut connection” that skips one or more layers to reduce the information loss.

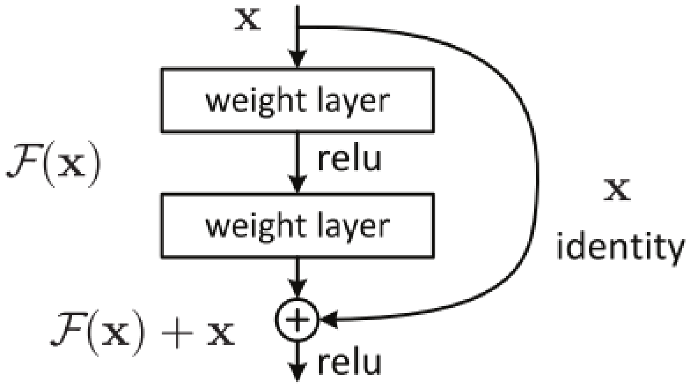

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
$\mathbf{x}$
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 4: ResNet architecture

ResNet-34 achieved a top-5 validation error of 5.71% better than BN-inception and VGG. ResNet-152 achieves a top-5 validation error of 4.49%. An ensemble of 6 models with different depths achieves a top-5 validation error of 3.57%. Winning the 1st place in ILSVRC-2015.

2.2 Visualization tools

2.2.1 History Dictionary

It generates an evolution plot using Matplotlib of train and validation loss/accuracy according to epochs.

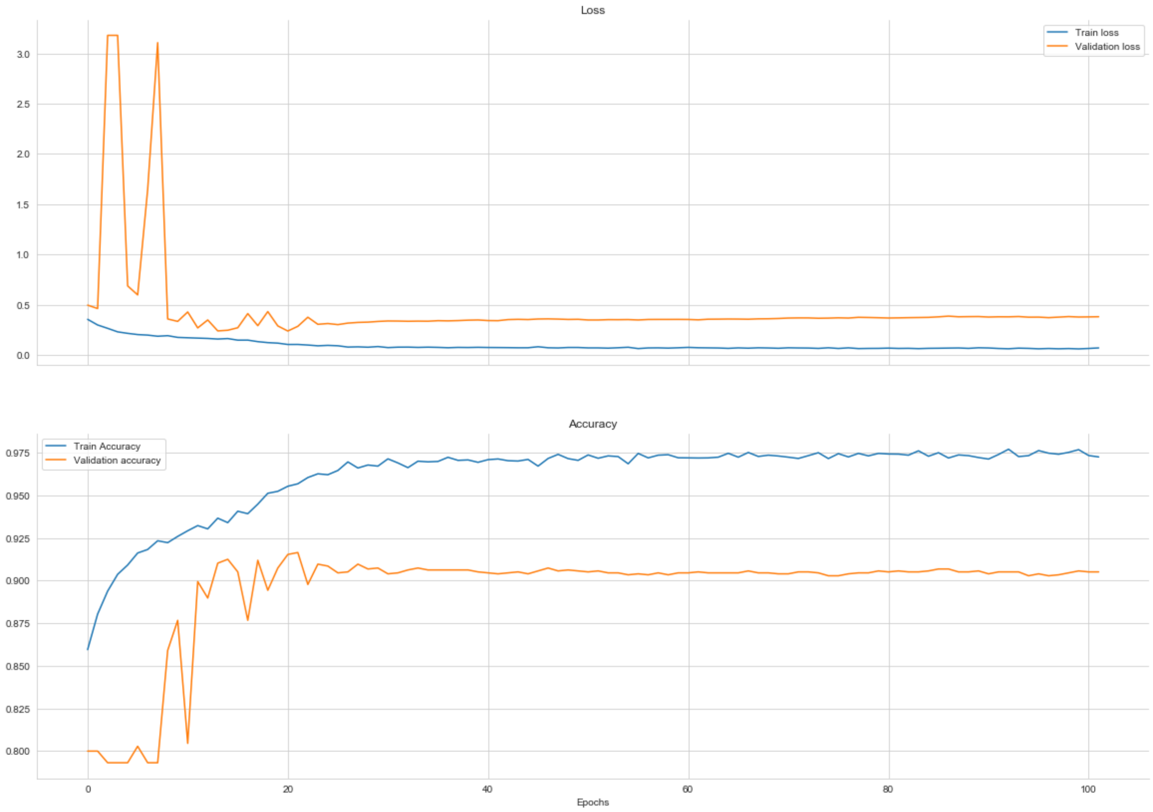


Figure 5: History Dictionary Example

2.2.2 Confusion Matrix

In the field of machine learning and specifically the problem of statistical classification, a confusion matrix also known as an error matrix is a specific table layout that allows visualization of the performance of an algorithm, typically a supervised learning one. Each row of the matrix represents the instances in an actual class while each column represents the instances in a predicted class.

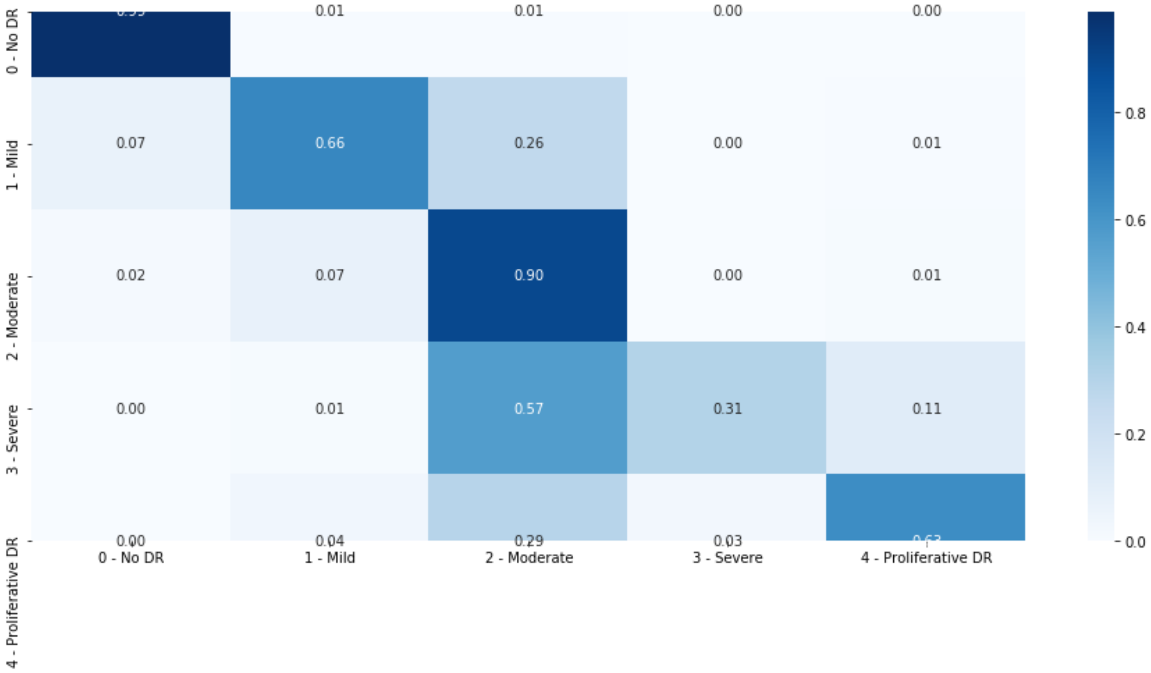
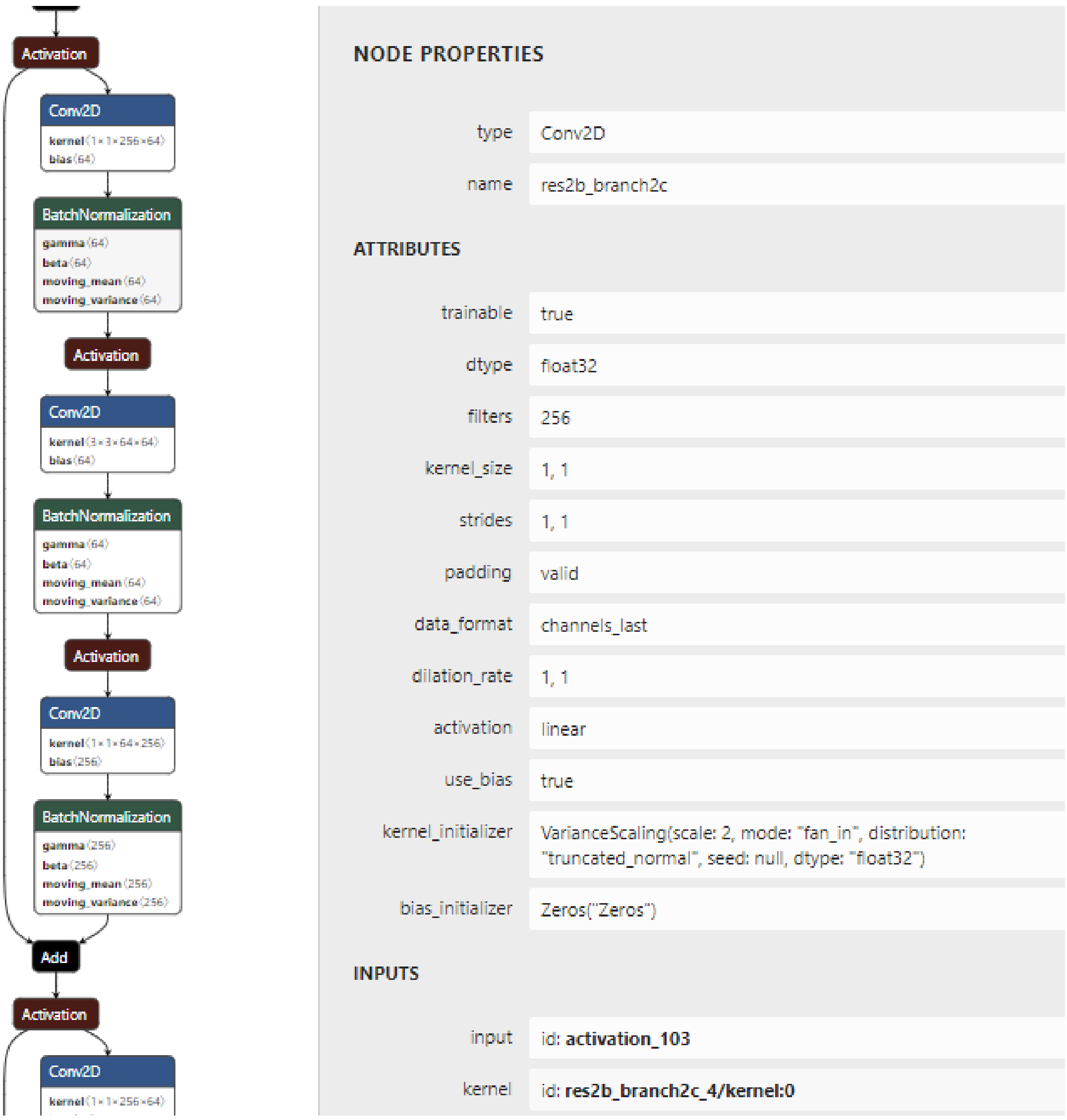


Figure 6: Confusion Matrix Example

2.2.3 Netron

Netron is a viewer for neural network, deep learning and machine learning models.
It enables us to visualize the architecture of our model as well as the attributes and the input of each layer.

Below is a part of our model's architecture visualized using the Netron tool:



NODE PROPERTIES

type	Conv2D
name	res2b_branch2c

ATTRIBUTES

trainable	true
dtype	float32
filters	256
kernel_size	1, 1
strides	1, 1
padding	valid
data_format	channels_last
dilation_rate	1, 1
activation	linear
use_bias	true
kernel_initializer	VarianceScaling(scale: 2, mode: "fan_in", distribution: "truncated_normal", seed: null, dtype: "float32")
bias_initializer	Zeros("Zeros")

INPUTS

input	id: **activation_103**
kernel	id: **res2b_branch2c_4/kernel:0**

Figure 7: Architecture Visualization

2.3 Data-set

The data-set is provided by Kaggle and it contains 3662 images used for training, 1992 for testing and a “train.csv” file containing the training labels.

A clinician has rated each image for the severity of diabetic retinopathy on a scale of 0 to 4, explained below:

- 0 - No DR
- 1 - Mild DR
- 2 - Moderate DR
- 3 - Severe DR
- 4 - Proliferate DR

Our Data is distributed as shown in the graph below:

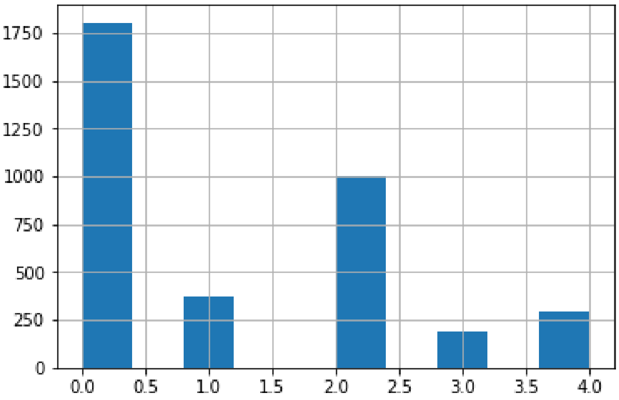
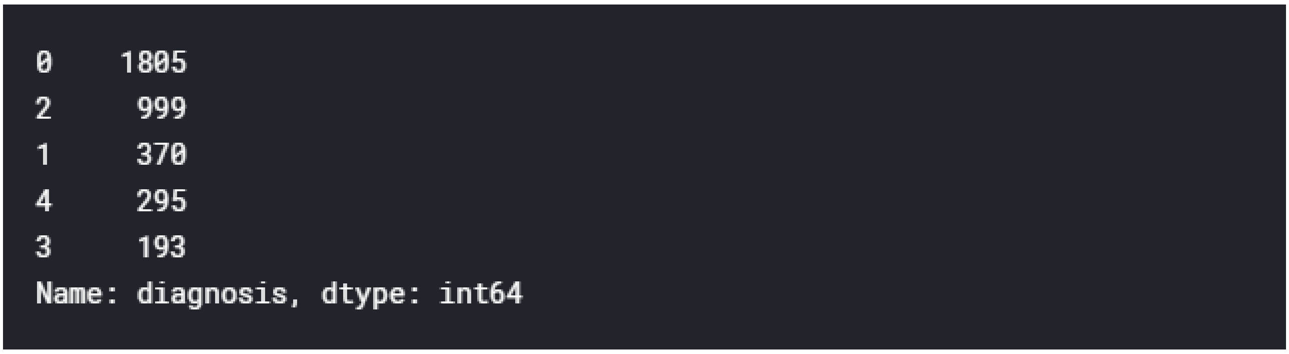


Figure 8: Distribution of the data-set



```
0    1805
2     999
1     370
4     295
3     193
Name: diagnosis, dtype: int64
```

Figure 9: Distribution of the data-set(*)

the data-set should be split into:

- Training Data: A set of examples used for learning, that is to fit the parameters of the classifier.
- Validation Data: Is a sample of data held back from training our model that is used to give an estimate of model skill while tuning model's hyper-parameters.
- Test Data: A set of examples used only to assess the performance of a fully-specified classifier.

Here is a sample of the data-set:

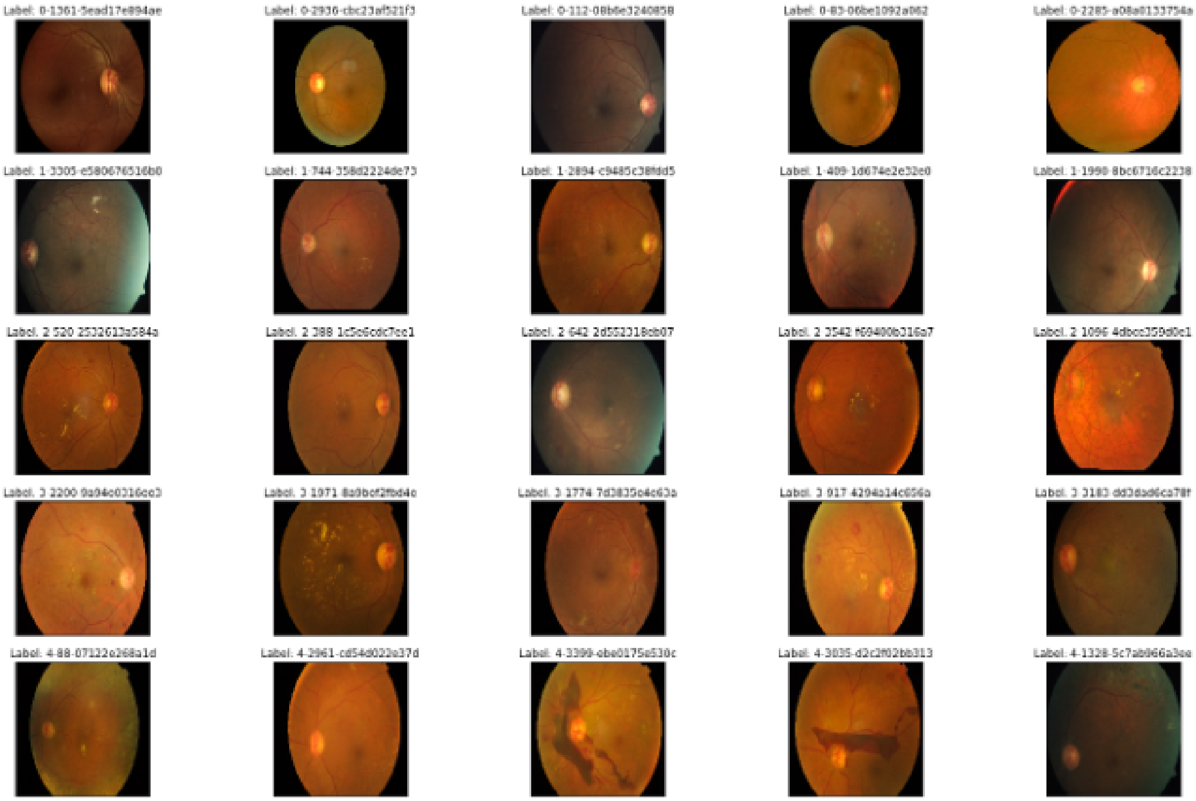

Label: 0-1361-5ead17e894ae
Label: 0-2936-cbc23af521f3
Label: 0-112-08b6e3240858
Label: 0-83-06be1092a062
Label: 0-2285-a08a0133754a
Label: 1-3305-e580676516b0
Label: 1-744-358d2224de73
Label: 1-2894-c9485c38fdd5
Label: 1-409-1d674e2e32e0
Label: 1-1990-8bc6716c2238
Label: 2 520 2532613a584a
Label: 2 388 1c5c6cdc7ee1
Label: 2 642 2d552318cb07
Label: 2 3542 f69400b316a7
Label: 2 1096 4dbcc359d0c1
Label: 3 2200 9a94c0316ec3
Label: 3 1971 8a9bcf2fbd4c
Label: 3 1774 7d3835a4c63a
Label: 3 917 4294a14c656a
Label: 3 3183 dd3dad6ca78f
Label: 4-88-07122e268a1d
Label: 4-2961-cd54d022a37d
Label: 4-3399-ebe0175e530c
Label: 4-3035-d2c2f02bb313
Label: 4-1328-5c7ab066a3ee

Figure 10: Data-set sample

2.4 Maximazing accuracy and minimizing loss

The architectures are not complete. Indeed they are missing the output layers which vary according to our use and it is our role during the training phase to choose the best disposition by manipulating the layers.
A Neural Network (NN) also has several parameters:

- Epochs: One epoch is when an entire data-set is passed forward and backward through a neural network only once.
  Since one epoch is too big to feed to the model at once we divide it into several smaller batches.

- Batch: Total number of training examples present in a single batch. It impacts learning significantly.

- Learning Rate: Learning Rate determines how fast or slow we will move towards the optimal weights -A low learning rate means more training time and more time results in increased GPU cost -A high learning rate could result in a model that might not predict anything accurate.

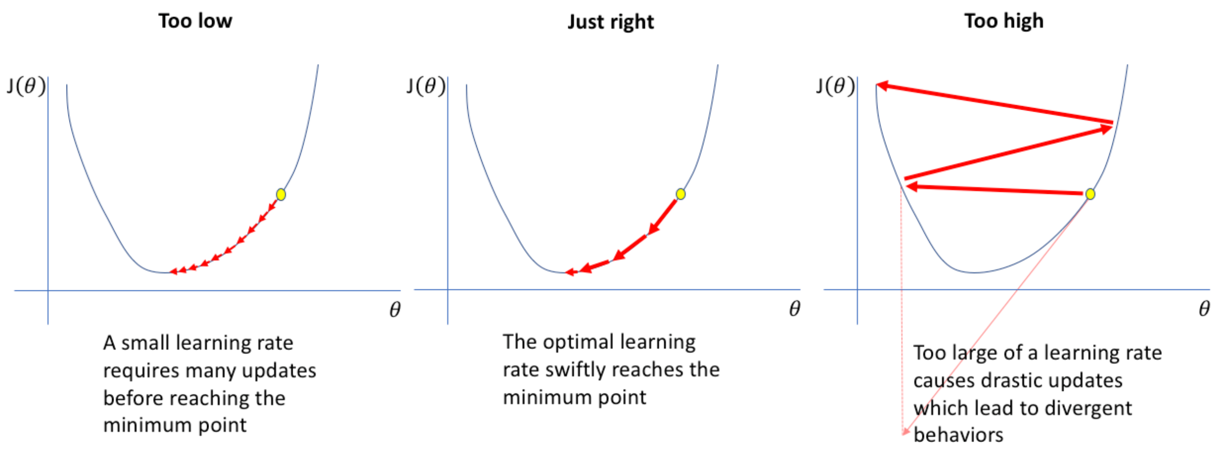

Too low
Just right
Too high
J(θ)
J(θ)
J(θ)
θ
θ
θ
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors

Figure 11: Impact of Learning Rate

Tweaking each of the parameters mentioned above affects the model's final accuracy. Below are the History Dictionary and Confusion Matrix of our model:

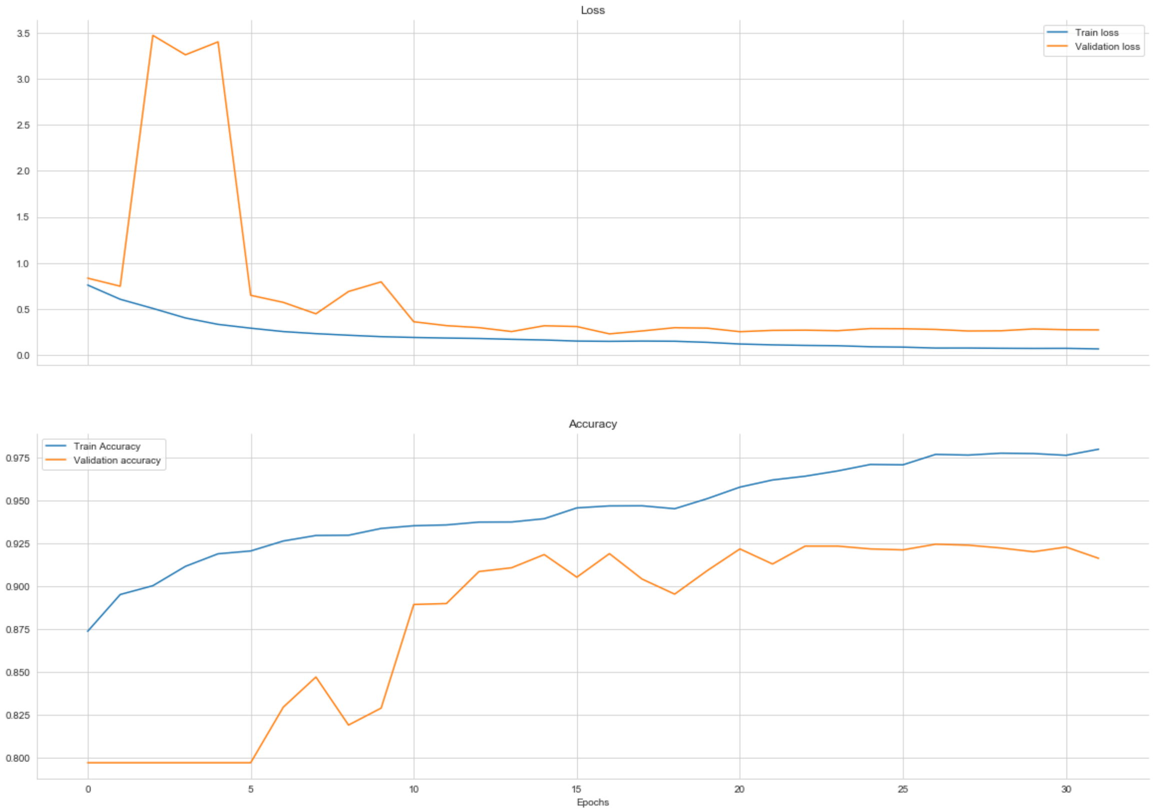


Figure 12: Final History Dictionary

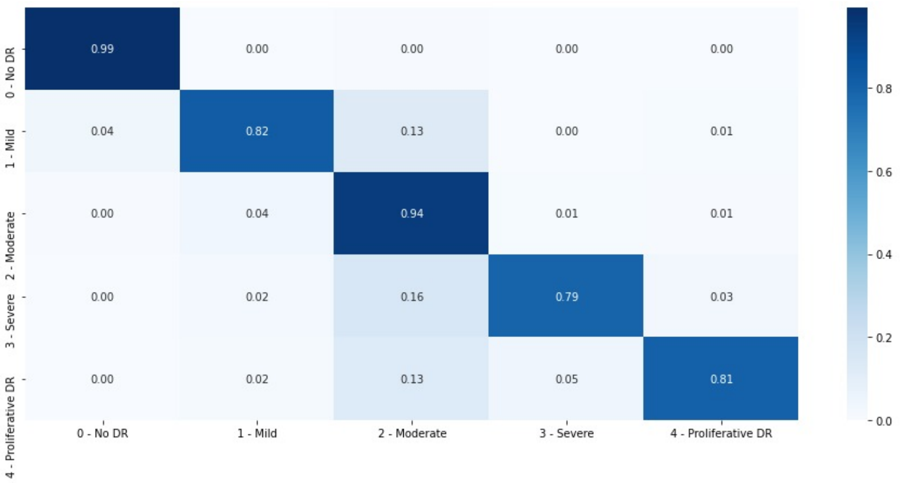
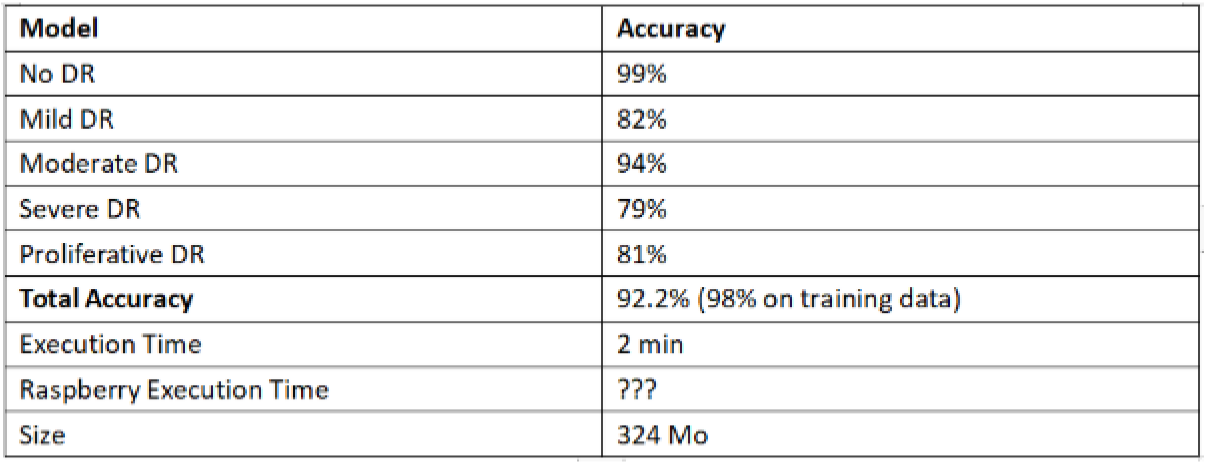


Figure 13: Final Confusion Matrix

2.5 Encountered Problems

We notice that our model has trouble detecting some classes:

- Mild DR: 82% accuracy
- Severe DR: 79% accuracy
- Proliferate DR: 81% accuracy



Model	**Accuracy**
No DR	99%
Mild DR	82%
Moderate DR	94%
Severe DR	79%
Proliferative DR	81%
Total Accuracy	92.2% (98% on training data)
Execution Time	2 min
Raspberry Execution Time	???
Size	324 Mo

Figure 14: Original Model Performance

This result may be explained by the lack of data in these specific three classes (only 193 images for the Severe DR class) as shown in Figure 9 in the Data-set subsection.

- Huge time consumption for prediction.
- Great usage of memory to do the necessary operations(28 Million parameters)
- Important size which will make the implementation of the model on hardware difficult.

2.6 Optimization Techniques

Optimization is an important step since the trained model may present several problems such as time consumption, memory usage and large size.
Here it is important to introduce two of the most important optimization techniques.

2.6.1 Pruning

Pruning is a technique in deep learning that aids in the development of smaller and more efficient neural networks. It is a model optimization technique that involves eliminating unnecessary values in the weight tensor. This results in compressed neural networks that run faster, reducing the computational cost involved in training the networks. This is even more crucial when deploying models to mobile phones or other edge devices, as it is in our case.[9]

Pruning requires Fine-Tuning it means that our model needs further training for a few epochs which makes this solution time consuming.

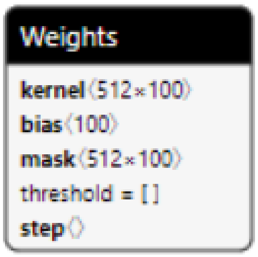
- **1st Technique:** Magnitude-based weight pruning

This technique works by removing parameters within a model that have only a minor impact on its predictions. We are practically setting neural network parameters values to zero to remove low-weight connections between the layers of a neural network.[2]

We used a function that requires the Keras layer to be pruned and other parameters to configure the pruning algorithm during training.The parameters are:

1. **Sparsity:** It is used across the whole training process. X% sparsity means that X% of the weight tensor is going to be pruned away.
2. **Schedule:** Contains the pruning frequency.

The idea of this technique is to apply a mask that gets rid of the unuseful weights.
Using Netron we can visualize the layer pruned as well as the mask we applied to this layer.

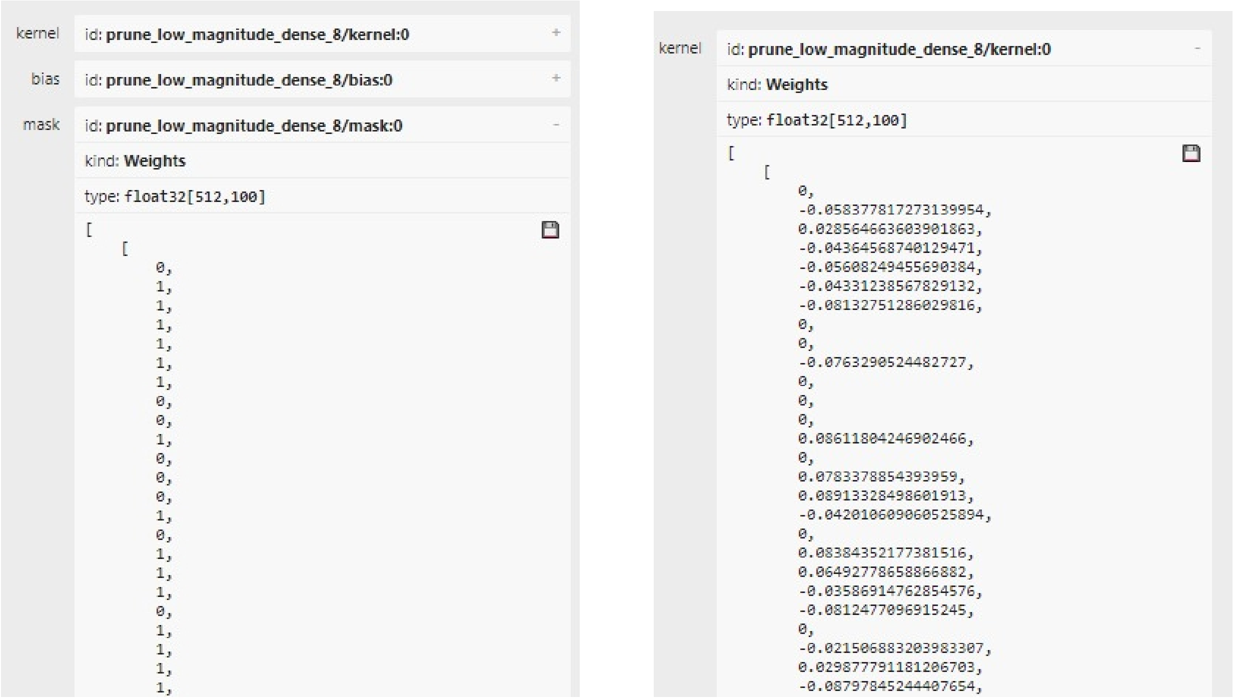


Weights

kernel⟨512×100⟩
bias⟨100⟩
mask⟨512×100⟩
threshold = [ ]
step⟨⟩

Figure 15: Layer parameters

this mask allows to prune 20% of the weights



```
kernel  id: prune_low_magnitude_dense_8/kernel:0
bias    id: prune_low_magnitude_dense_8/bias:0
mask    id: prune_low_magnitude_dense_8/mask:0
        kind: Weights
        type: float32[512,100]
        [
            [
                0,
                1,
                1,
                1,
                1,
                1,
                1,
                0,
                0,
                1,
                0,
                0,
                0,
                1,
                0,
                1,
                1,
                1,
                0,
                1,
                1,
                1,
                1,
```

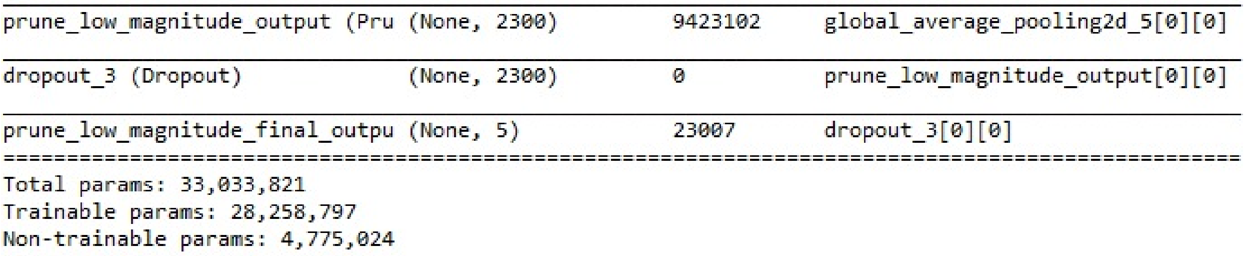
(a) Mask structure

```
kernel  id: prune_low_magnitude_dense_8/kernel:0
        kind: Weights
        type: float32[512,100]
        [
            [
                0,
                -0.058377817273139954,
                0.028564663603901863,
                -0.04364568740129471,
                -0.05608249455690384,
                -0.04331238567829132,
                -0.08132751286029816,
                0,
                0,
                -0.0763290524482727,
                0,
                0,
                0,
                0.08611804246902466,
                0,
                0.0783378854393959,
                0.08913328498601913,
                -0.042010609060525894,
                0,
                0.08384352177381516,
                0.06492778658866882,
                -0.03586914762854576,
                -0.0812477096915245,
                0,
                -0.021506883203983307,
                0.029877791181206703,
                -0.08797845244407654,
```

(b) Weighs after applying mask

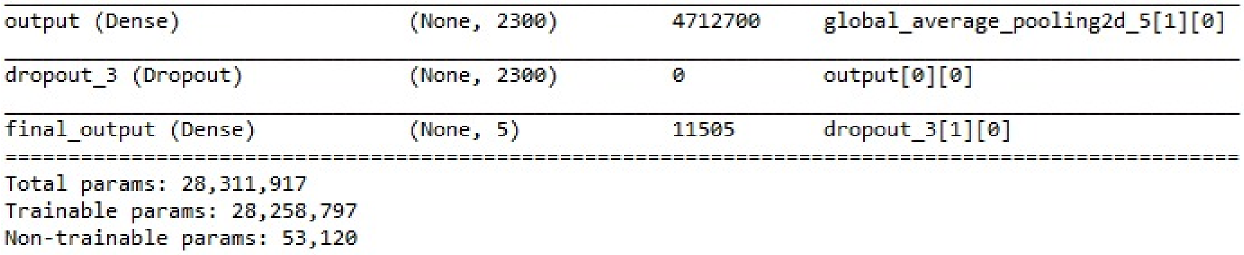
Figure 16: Mask structure and weights after applying it

Thanks to this method we managed to build a 108 Mo model while maintaining a high accuracy which is 82% but the number of parameters remain the same.



```
prune_low_magnitude_output (Pru (None, 2300)        9423102      global_average_pooling2d_5[0][0]

dropout_3 (Dropout)             (None, 2300)        0            prune_low_magnitude_output[0][0]

prune_low_magnitude_final_outpu (None, 5)           23007        dropout_3[0][0]
==================================================================================================
Total params: 33,033,821
Trainable params: 28,258,797
Non-trainable params: 4,775,024
```

Figure 17: Number of parameters (Mask included)

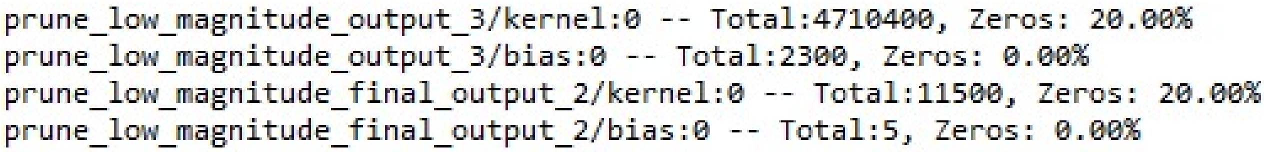


```
output (Dense)                  (None, 2300)        4712700      global_average_pooling2d_5[1][0]

dropout_3 (Dropout)             (None, 2300)        0            output[0][0]

final_output (Dense)            (None, 5)           11505        dropout_3[1][0]
==================================================================================================
Total params: 28,311,917
Trainable params: 28,258,797
Non-trainable params: 53,120
```

Figure 18: Number of parameters (Mask stripped)

The number dropped because we removed the mask used to prune the weights.

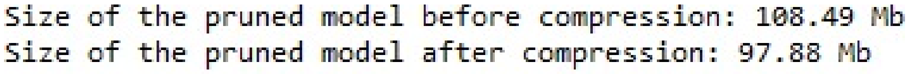
We can check the number of zeros in each layer to confirm our result.



```
prune_low_magnitude_output_3/kernel:0 -- Total:4710400, Zeros: 20.00%
prune_low_magnitude_output_3/bias:0 -- Total:2300, Zeros: 0.00%
prune_low_magnitude_final_output_2/kernel:0 -- Total:11500, Zeros: 20.00%
prune_low_magnitude_final_output_2/bias:0 -- Total:5, Zeros: 0.00%
```

Figure 19: Number of zeros in pruned model

The pruned model can be further improved by compressing it into a '.zip' file. The advantage of compressing an already pruned model lies in higher compression and decompression speeds in comparison to a non pruned model.



```
Size of the pruned model before compression: 108.49 Mb
Size of the pruned model after compression: 97.88 Mb
```

Figure 20: Pruned model size after Compression

- **2nd Technique:** Network Slimming

The network slimming approach takes wide and large networks as input models to identify and prune insignificant channels, yielding thin and compact models with comparable accuracy. Its purpose is to:

1. Reduce the model size.
2. Decrease the run-time memory footprint.
3. Lower the number of computing operations.

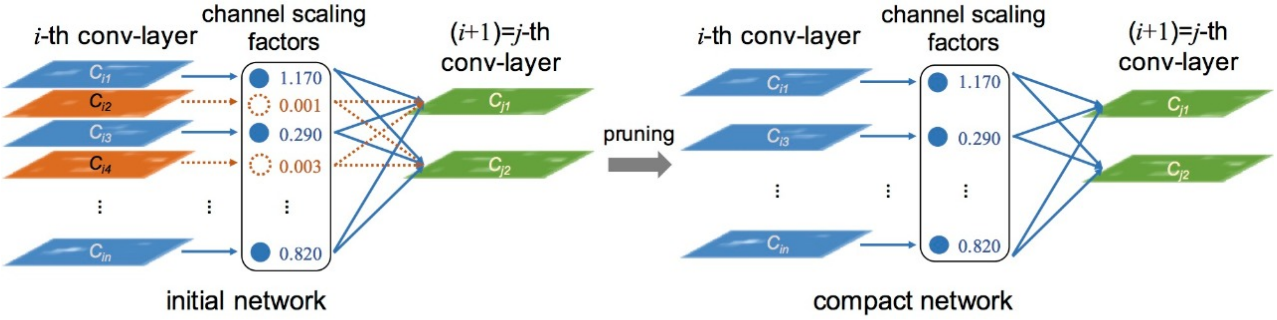


Figure 21: The Channel Pruning Process

The idea of this technique is to associate a scaling factor with each channel in convolutional layers. Sparsity regularization is imposed on these scaling factors to automatically identify unimportant channels. The channels with small scaling factor values (in orange color) will be pruned. After pruning, we obtain compact models, which are then fine-tuned to achieve comparable accuracy as normally trained full network.[10]

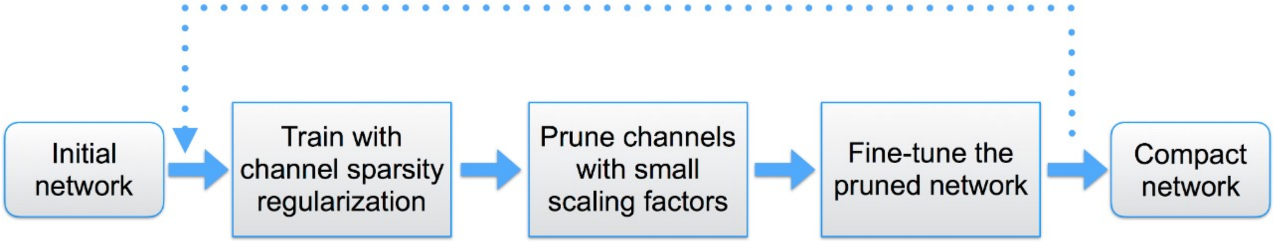
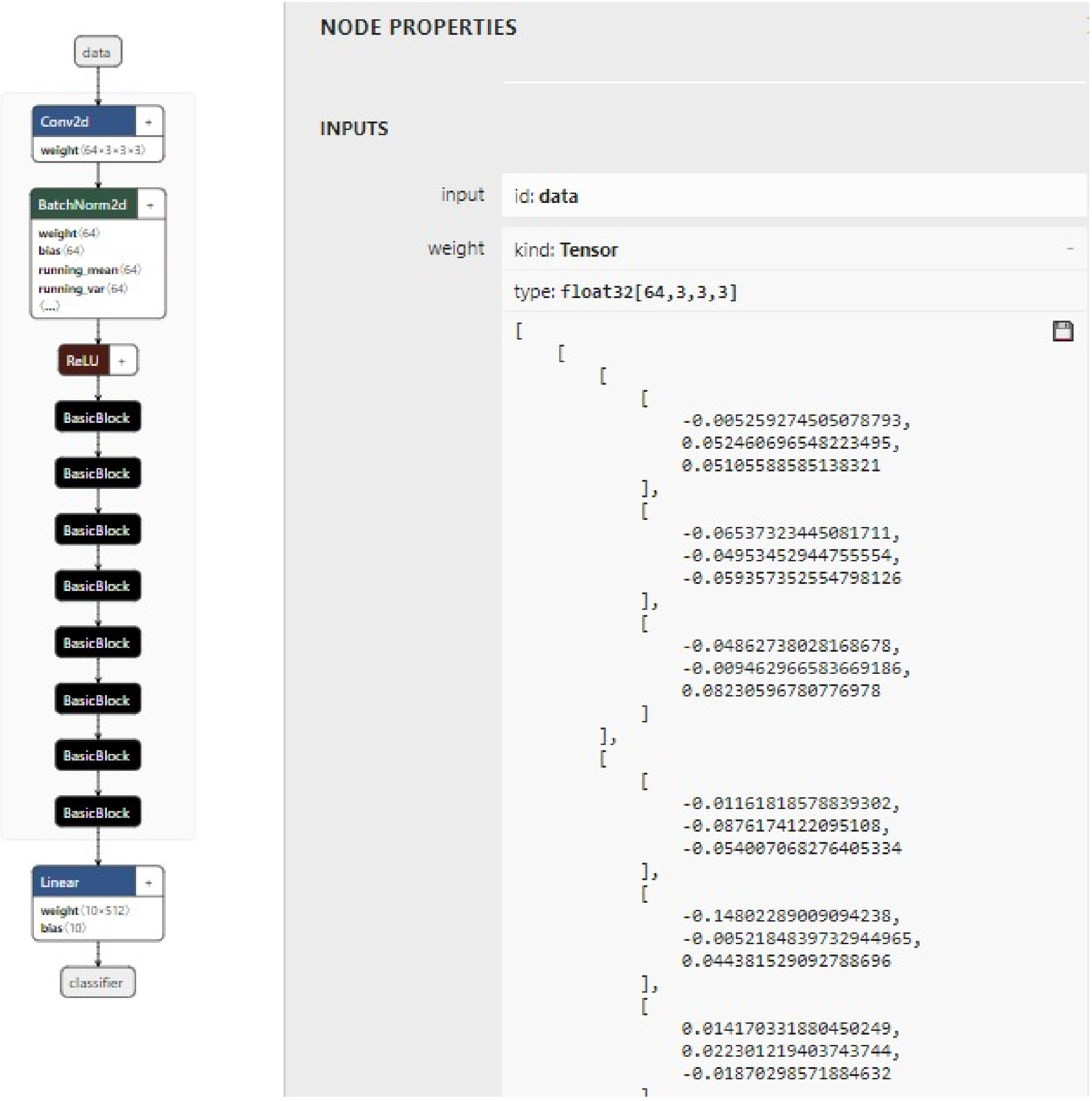

Initial network
Train with channel sparsity regularization
Prune channels with small scaling factors
Fine-tune the pruned network
Compact network

Figure 22: Flow-chart of the Network Slimming procedure

Our initial network is a Resnet-18 which is a smaller version of Resnet-50 with fewer layers and parameters.
We provided our model with CIFAR-10 dataset consisted of 60000 colour images in 10 different classes.

Using Netron we can visualize the architecture of our model.



NODE PROPERTIES

INPUTS

input: id: **data**

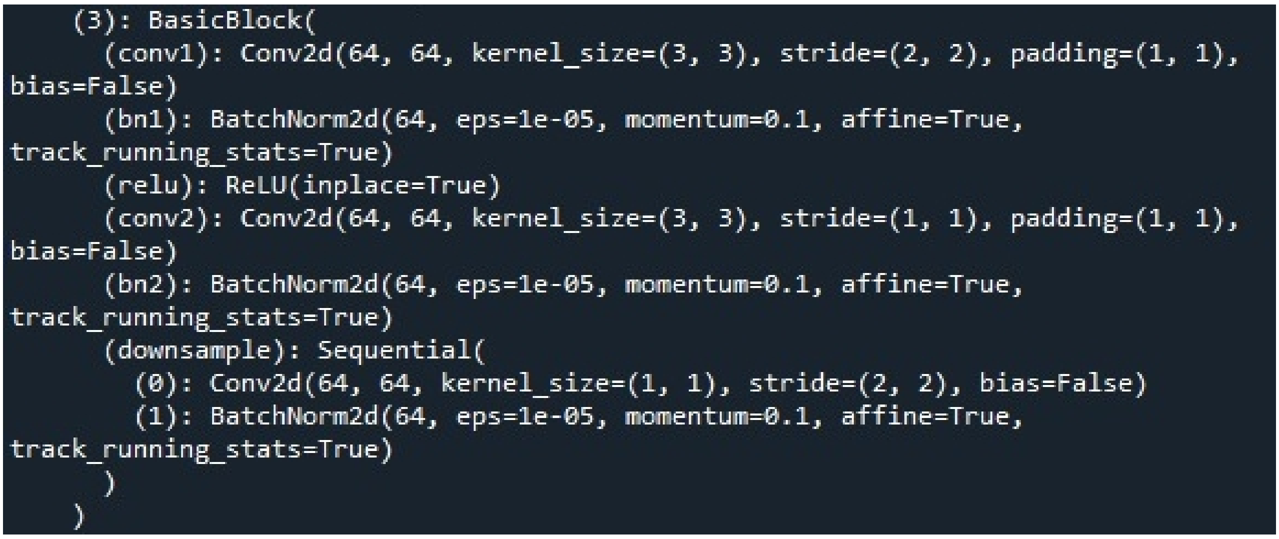
weight: kind: **Tensor**

type: `float32[64,3,3,3]`

```
[
    [
        [
            [
                -0.005259274505078793,
                0.052460696548223495,
                0.05105588585138321
            ],
            [
                -0.06537323445081711,
                -0.04953452944755554,
                -0.059357352554798126
            ],
            [
                -0.04862738028168678,
                -0.009462966583669186,
                0.08230596780776978
            ]
        ],
        [
            [
                -0.011618185788393O2,
                -0.0876174122095108,
                -0.054007068276405334
            ],
            [
                -0.14802289009094238,
                -0.005218483973294496 5,
                0.04438152909278869 6
            ],
            [
                0.014170331880450249,
                0.022301219403743744,
                -0.01870298571884632
            ]
```

Figure 23: ResNet-18 architecture

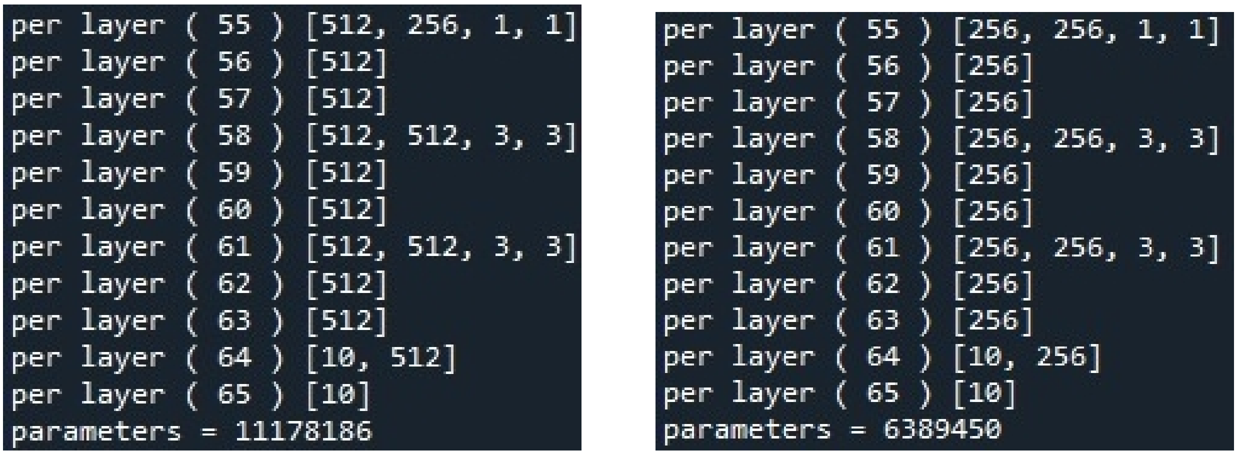
Each basic block is a sequential of Conv2D layer followed by a Batch_Normalization layer and a ReLu activation function all that added to an identity shortcut at the end which makes it similar to the ResNet-50 architecture.



```
    (3): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 64, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
```

Figure 24: Basic Block Layers

Our initial model here has a total of 65 layers and 11 Million parameters. The pruning technique can either compress the model by reducing the number of channels of the Conv2d layers or delete a whole layer by setting its channels to 0:



```
per layer ( 55 ) [512, 256, 1, 1]
per layer ( 56 ) [512]
per layer ( 57 ) [512]
per layer ( 58 ) [512, 512, 3, 3]
per layer ( 59 ) [512]
per layer ( 60 ) [512]
per layer ( 61 ) [512, 512, 3, 3]
per layer ( 62 ) [512]
per layer ( 63 ) [512]
per layer ( 64 ) [10, 512]
per layer ( 65 ) [10]
parameters = 11178186
```

(a)

```
per layer ( 55 ) [256, 256, 1, 1]
per layer ( 56 ) [256]
per layer ( 57 ) [256]
per layer ( 58 ) [256, 256, 3, 3]
per layer ( 59 ) [256]
per layer ( 60 ) [256]
per layer ( 61 ) [256, 256, 3, 3]
per layer ( 62 ) [256]
per layer ( 63 ) [256]
per layer ( 64 ) [10, 256]
per layer ( 65 ) [10]
parameters = 6389450
```

(b)

Figure 25: Model parameters before and after Compression(50%)

To avoid any kind of confusion here is a look at the input data shape of the CNN:

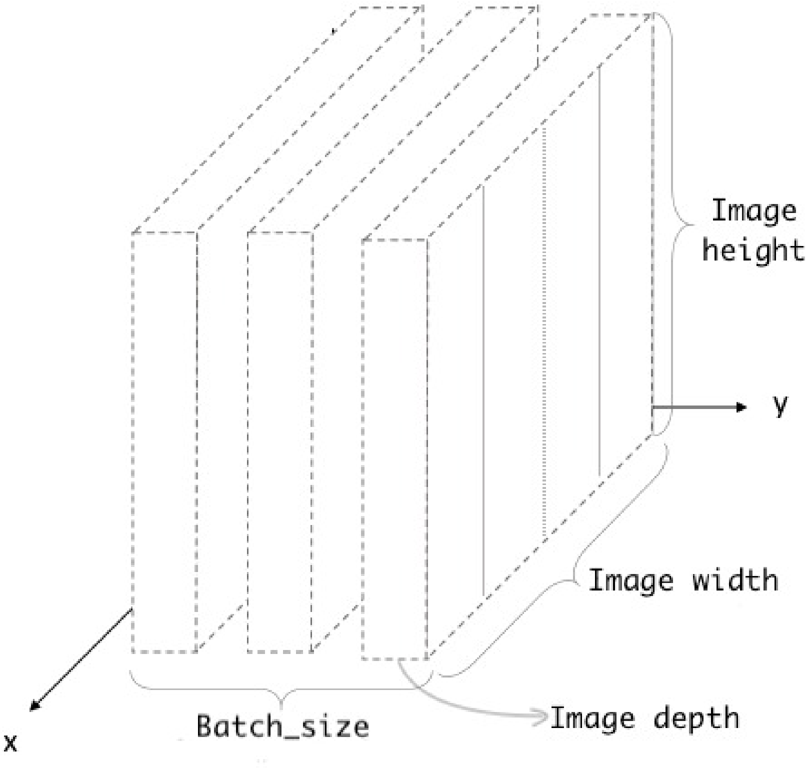
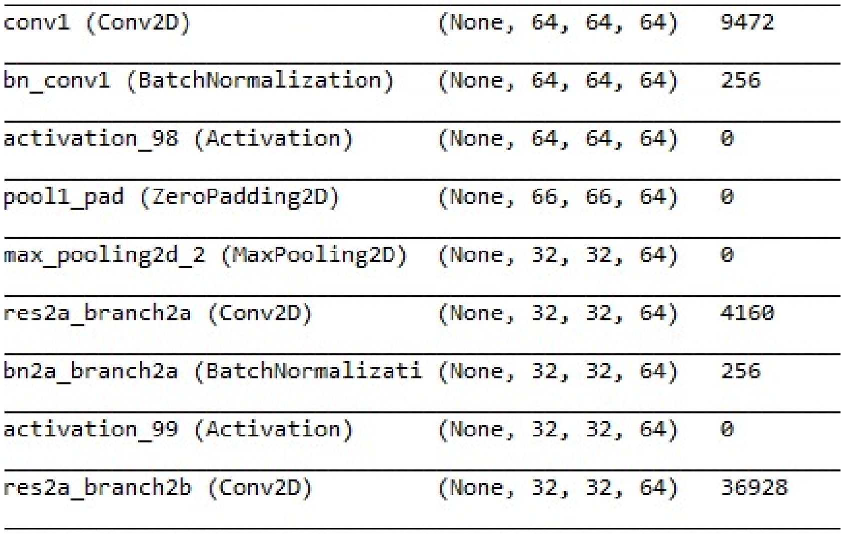


Figure 26: ConvNet Input Shape[1]

- **Input Shape:**

We always have to give a 4D array as input to the CNN. So input data has a shape of *(batch_size, height, width, depth)*.

Going back to our ResNet-50 model that we are working on. We applied the function summary() that can help us verifying the input shape of each layer.

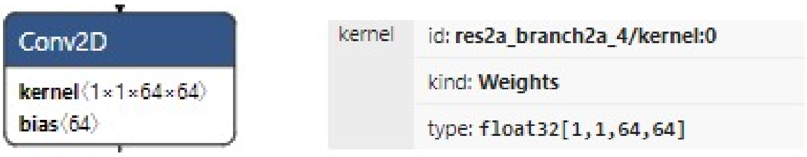


```
conv1 (Conv2D)                (None, 64, 64, 64)   9472

bn_conv1 (BatchNormalization)  (None, 64, 64, 64)   256

activation_98 (Activation)     (None, 64, 64, 64)   0

pool1_pad (ZeroPadding2D)      (None, 66, 66, 64)   0

max_pooling2d_2 (MaxPooling2D) (None, 32, 32, 64)   0

res2a_branch2a (Conv2D)        (None, 32, 32, 64)   4160

bn2a_branch2a (BatchNormalizati (None, 32, 32, 64)  256

activation_99 (Activation)     (None, 32, 32, 64)   0

res2a_branch2b (Conv2D)        (None, 32, 32, 64)   36928
```

Figure 27: Model's layers' parameters

We can notice that our batch is *None* because ResNet-50 architecture does not know in advance the batch size we are using. As a result this value will be replaced while fitting our data.

Since we are feeding our model a single image at a time (the model's input is 1 image) we can notice thanks to Netron that the batch got the value 1 which is the minimum value that assures the lowest number of parameters per layer.



Conv2D
kernel⟨1×1×64×64⟩
bias⟨64⟩

(a)

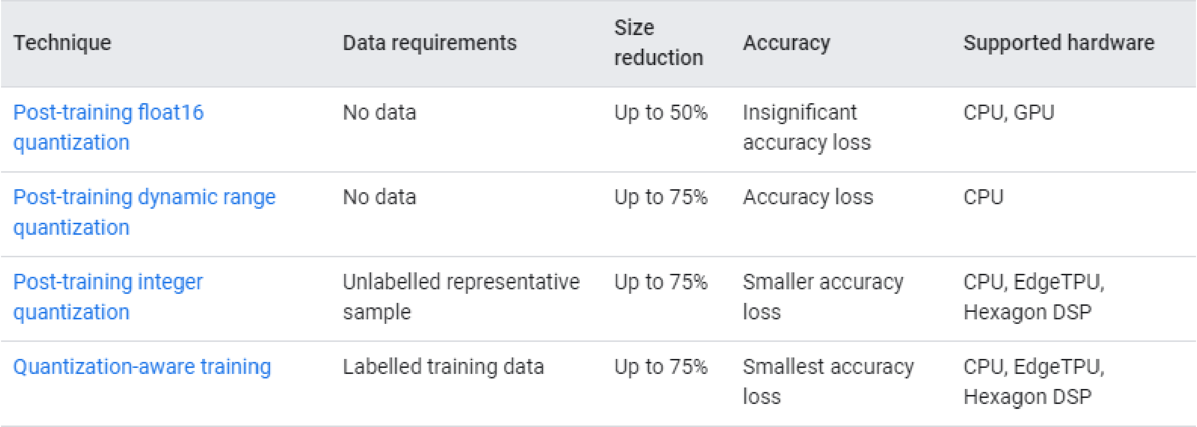
kernel
id: res2a_branch2a_4/kernel:0
kind: Weights
type: float32[1,1,64,64]

(b)

Figure 28: Input shape of the Conv2D layers

2.6.2 Quantization

Post-training quantization is a conversion technique that can reduces model size while also improving CPU and hardware accelerator latency, with little degradation in model accuracy. We can perform these techniques using an already-trained float TensorFlow model when we convert it to TensorFlow Lite format.



Technique	Data requirements	Size reduction	Accuracy	Supported hardware
Post-training float16 quantization	No data	Up to 50%	Insignificant accuracy loss	CPU, GPU
Post-training dynamic range quantization	No data	Up to 75%	Accuracy loss	CPU
Post-training integer quantization	Unlabelled representative sample	Up to 75%	Smaller accuracy loss	CPU, EdgeTPU, Hexagon DSP
Quantization-aware training	Labelled training data	Up to 75%	Smallest accuracy loss	CPU, EdgeTPU, Hexagon DSP

Figure 29: Quantization Outcomes[2]

There are different approaches in quantization and we have experimented with some of them:

- **1st Technique:** Post-training dynamic range quantization

The first technique that we applied for quantizing our model is Post-training Dynamic Range Quantization:

The simplest form of post-training Quantization statically quantizes only the weights from 32-bit floating point to 8-bits of precision. This technique is enabled as an option in the Tensorflow Lite converter:[2]

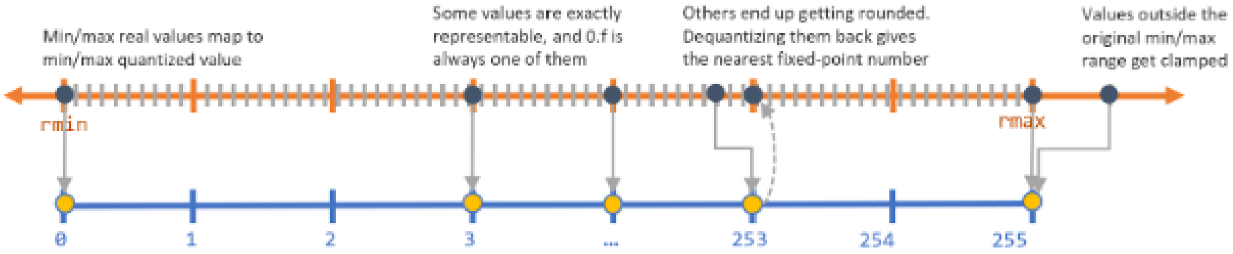
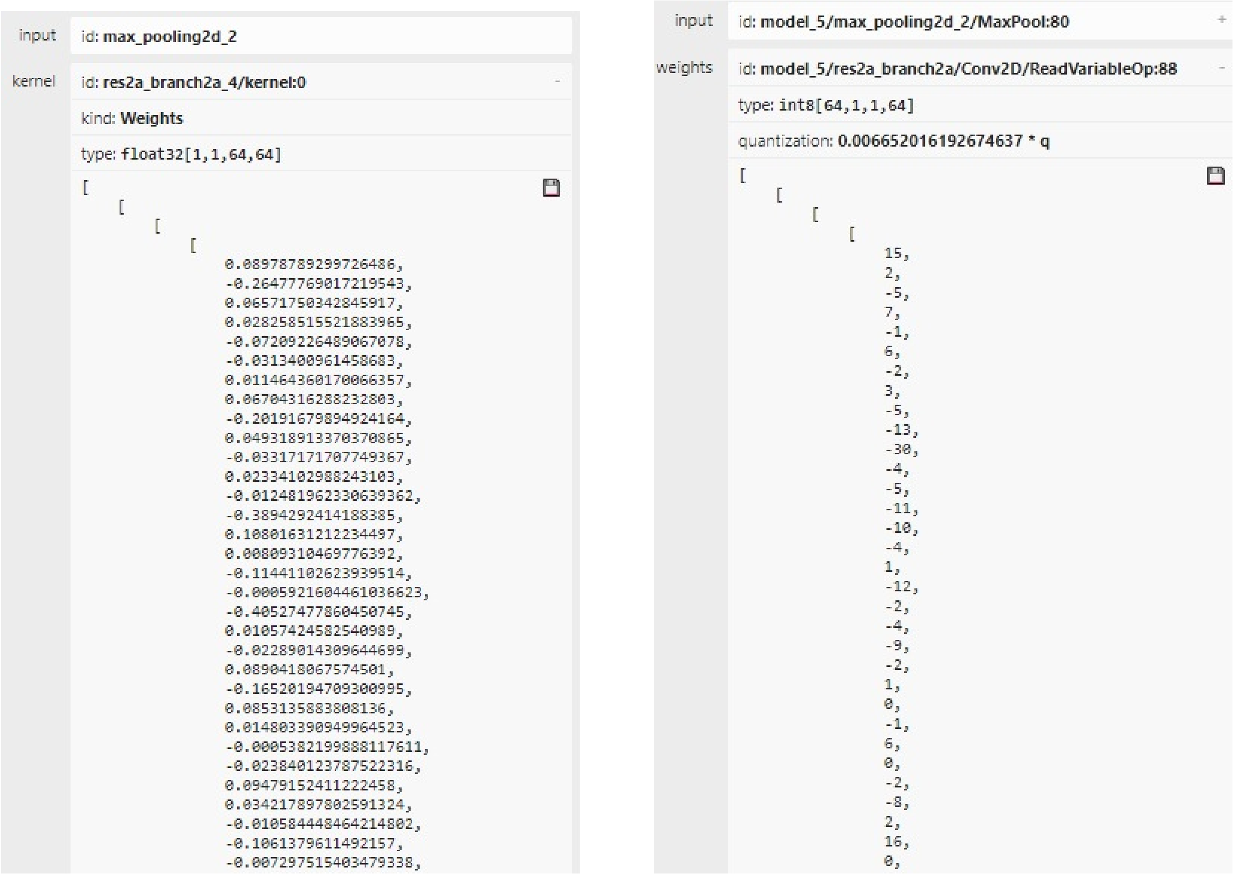

Min/max real values map to min/max quantized value
Some values are exactly representable, and 0.f is always one of them
Others end up getting rounded. Dequantizing them back gives the nearest fixed-point number
Values outside the original min/max range get clamped
rmin
rmax
0
1
2
3
...
253
254
255

Figure 30: Weigh Quantization process

Using Netron we can visualize our weights before and after Quantization.
Before Quantization: Weights are 32-bit floats.
After Quantization: All weights are 8-bit integers.



input id: max_pooling2d_2

kernel id: res2a_branch2a_4/kernel:0

kind: Weights

type: float32[1,1,64,64]

```
[
    [
        [
            [
                0.08978789299726486,
                -0.26477769017219543,
                0.06571750342845917,
                0.028258515521883965,
                -0.07209226489067078,
                -0.0313400961458683,
                0.011464360170066357,
                0.06704316288232803,
                -0.20191679894924164,
                0.049318913370370865,
                -0.03317171707749367,
                0.02334102988243103,
                -0.012481962330639362,
                -0.3894292414188385,
                0.10801631212234497,
                0.00809310469776392,
                -0.11441102623939514,
                -0.0005921604461036623,
                -0.40527477860450745,
                0.01057424582540989,
                -0.02289014309644699,
                0.0890418067574501,
                -0.16520194709300995,
                0.0853135883808136,
                0.014803390949964523,
                -0.0005382199888117611,
                -0.023840123787522316,
                0.09479152411222458,
                0.034217897802591324,
                -0.010584448464214802,
                -0.1061379611492157,
                -0.007297515403479338,
```

(a) Weighs Before Quantization

input id: model_5/max_pooling2d_2/MaxPool:80

weights id: model_5/res2a_branch2a/Conv2D/ReadVariableOp:88

type: int8[64,1,1,64]

quantization: 0.006652016192674637 * q

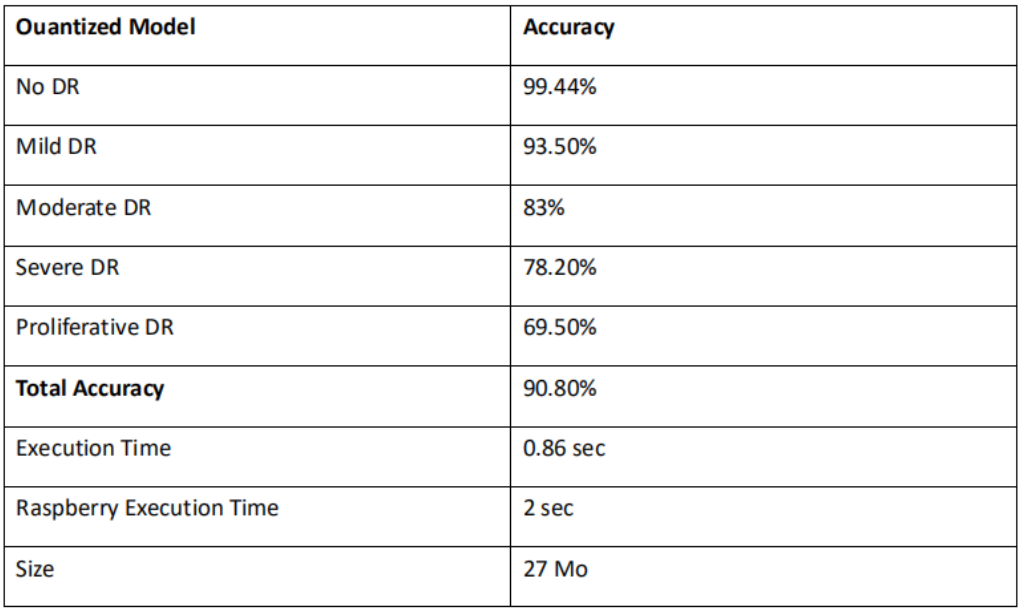
```
[
    [
        [
            [
                15,
                2,
                -5,
                7,
                -1,
                6,
                -2,
                3,
                -5,
                -13,
                -30,
                -4,
                -5,
                -11,
                -10,
                -4,
                1,
                -12,
                -2,
                -4,
                -9,
                -2,
                1,
                0,
                -1,
                6,
                0,
                -2,
                -8,
                2,
                16,
                0,
```

(b) Weighs After Quantization

Figure 31: Weights before and after Quantazation

Because the structure of the layer is changed after the quantization process from [1,1,64,64] to [64,1,1,64], the weights are not displayed in the same order and therefore it is not possible to directly compare the weights before and after quantization.

Results:



Ouantized Model	**Accuracy**
No DR	99.44%
Mild DR	93.50%
Moderate DR	83%
Severe DR	78.20%
Proliferative DR	69.50%
Total Accuracy	90.80%
Execution Time	0.86 sec
Raspberry Execution Time	2 sec
Size	27 Mo

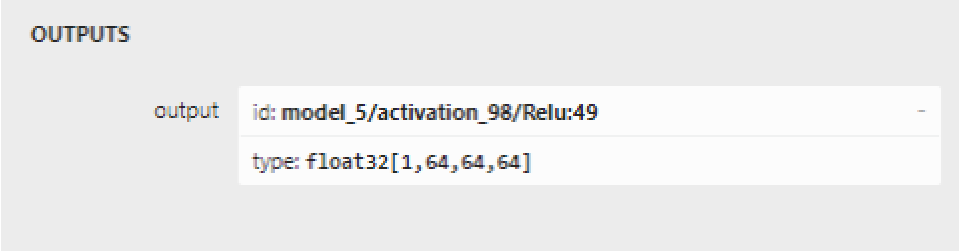
Figure 32: Model Performance after Dynamic Range Quantization

- **2nd Technique:** Post-training integer quantization

We can apply a second technique to quantize our model which is Post-training integer Quantization.
In contrast to the last technique that we applied which stores only the weights as 8-bit integers,this technique statically quantizes all weights and activations during model conversion. However, the conversion process requires that we supply a representative data-set. This representative data-set allows the Quantization process to measure the dynamic range of activations and inputs, which is critical to finding an accurate 8-bit representation of each weight and activation value.

Using Netron we can visualize the output of each layer (output of the activation functions) and compare it to the first technique.

Post-training Dynamic Range Quantization: Type float32

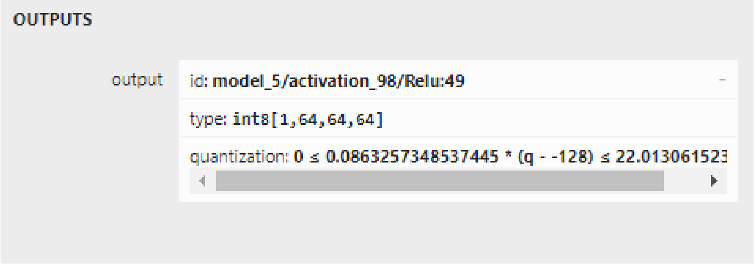


OUTPUTS

output
id: model_5/activation_98/Relu:49
type: float32[1,64,64,64]

Figure 33: Dynamic Range Quantization 32bit output

Post-training integer Quantization: Type int8



OUTPUTS

output

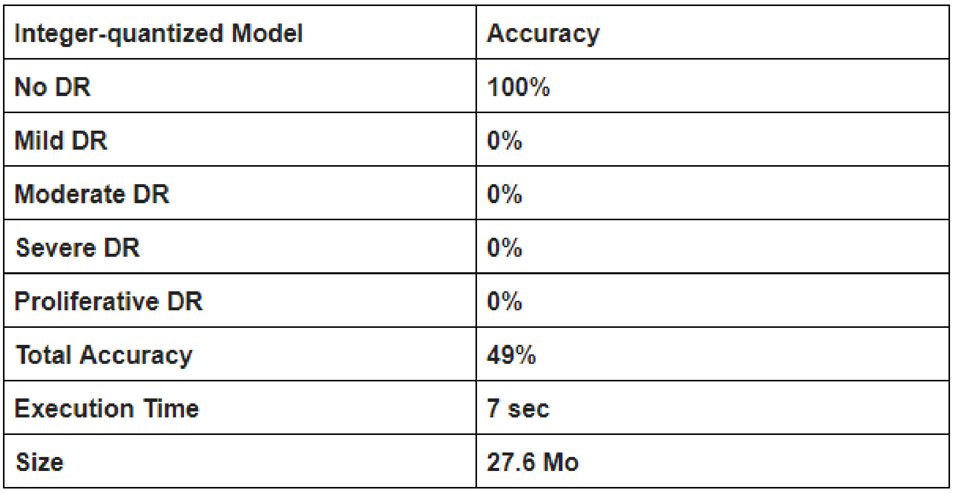
id: model_5/activation_98/Relu:49

type: int8[1,64,64,64]

quantization: 0 ≤ 0.0863257348537445 * (q - -128) ≤ 22.013061523

Figure 34: Integer Quantization 8bit output

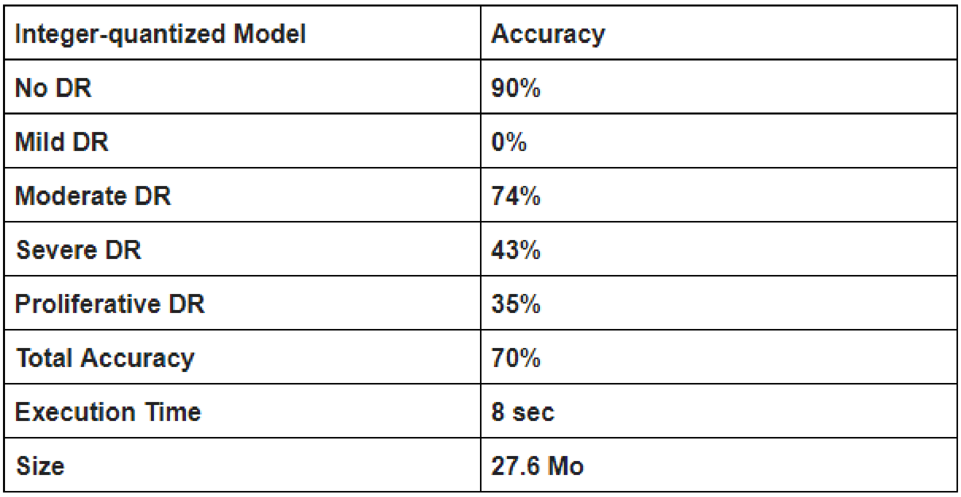
Results:



Integer-quantized Model	Accuracy
No DR	100%
Mild DR	0%
Moderate DR	0%
Severe DR	0%
Proliferative DR	0%
Total Accuracy	49%
Execution Time	7 sec
Size	27.6 Mo

Figure 35: Model Performance after Integer Quantization

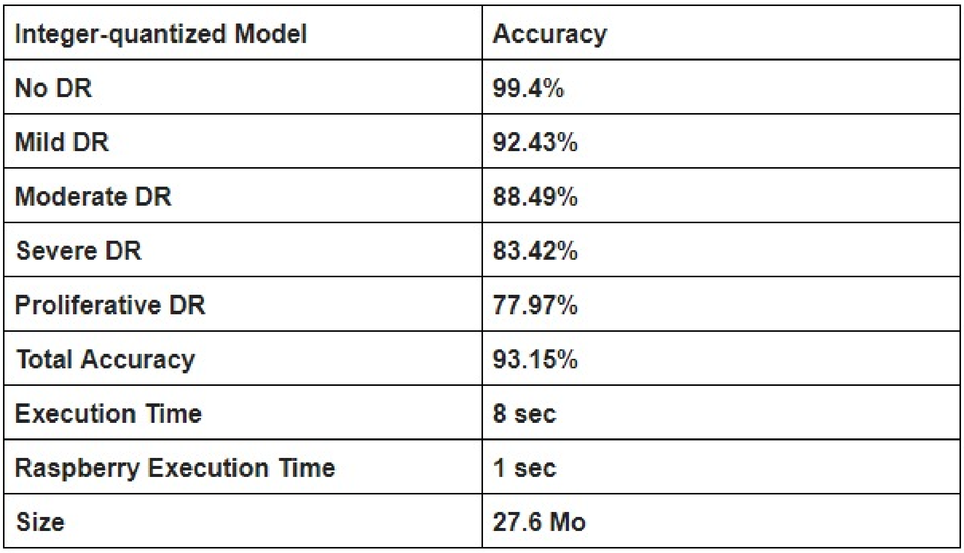
Our first model performs very badly. In fact, it predicts every input as a No DR and that is why we need a representative data-set that is formatted the same as the original training data-set (uses the same data range). In our case we need to rescale every input as a float-32 ranging between 0 and 1 before feeding it to our model.
After dealing with this problem we obtained better results:



Integer-quantized Model	Accuracy
No DR	90%
Mild DR	0%
Moderate DR	74%
Severe DR	43%
Proliferative DR	35%
Total Accuracy	70%
Execution Time	8 sec
Size	27.6 Mo

Figure 36: Model Performance after Rescaling

The problem is that accuracy is still low and our model cannot detect the Mild DR class so we need a representative data-set equally split between all classes to assure an accurate 8-bit representation of each weight and activation value.



Integer-quantized Model	Accuracy
No DR	99.4%
Mild DR	92.43%
Moderate DR	88.49%
Severe DR	83.42%
Proliferative DR	77.97%
Total Accuracy	93.15%
Execution Time	8 sec
Raspberry Execution Time	1 sec
Size	27.6 Mo

Figure 37: Final Model Performance after reshaping data-set

- Interpretation

-The result is satisfying in term of accuracy but it can be improved.
-The execution requires less time on Raspberry pi 3 b+.

- Perspective on performance improvement

Providing different representative data-sets to our model.
While converting a model to Tensorflow Lite format some optimizations may come at the cost of accuracy (DEFAULT, OPTIMIZE_FOR_SIZE, OPTIMIZE_FOR_LATENCY).

- **3rd Technique:** Quantization-aware training

The core idea is that QAT simulates low-precision inference-time computation in the forward pass of the training process.This introduces the quantization error as noise during the training and as part of the overall loss, which the optimization algorithm tries to minimize (the errors become part of the loss metric). Hence, the model learns parameters that are more robust to quantization.[2]

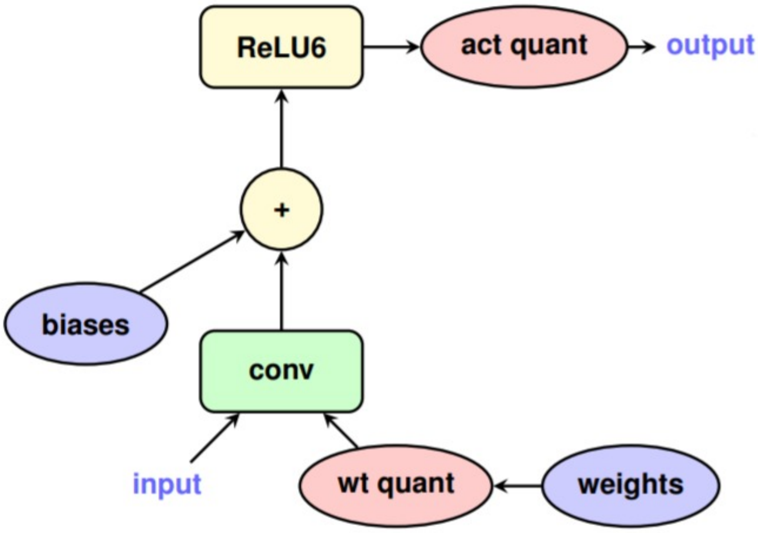
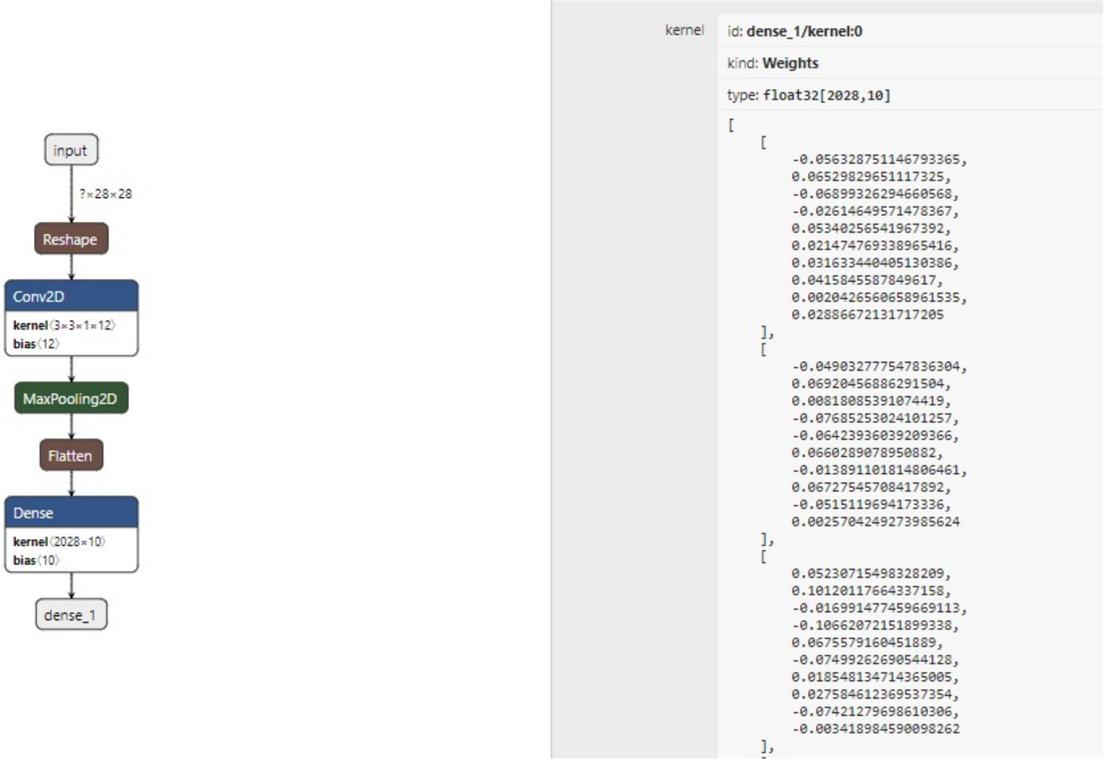


Figure 38: TensorFlow Lite quantization spec

The quantization error is modeled using **fake quantization** nodes to simulate the effect of quantization in the forward and backward passes. The forward-pass models quantization, while the backward-pass models quantization as a straight-through estimator. Both the forward- and backward-pass simulate the quantization of weights and activations. During back propagation, the parameters are updated at high precision as this is needed to ensure sufficient precision in accumulating tiny adjustments to the parameters.

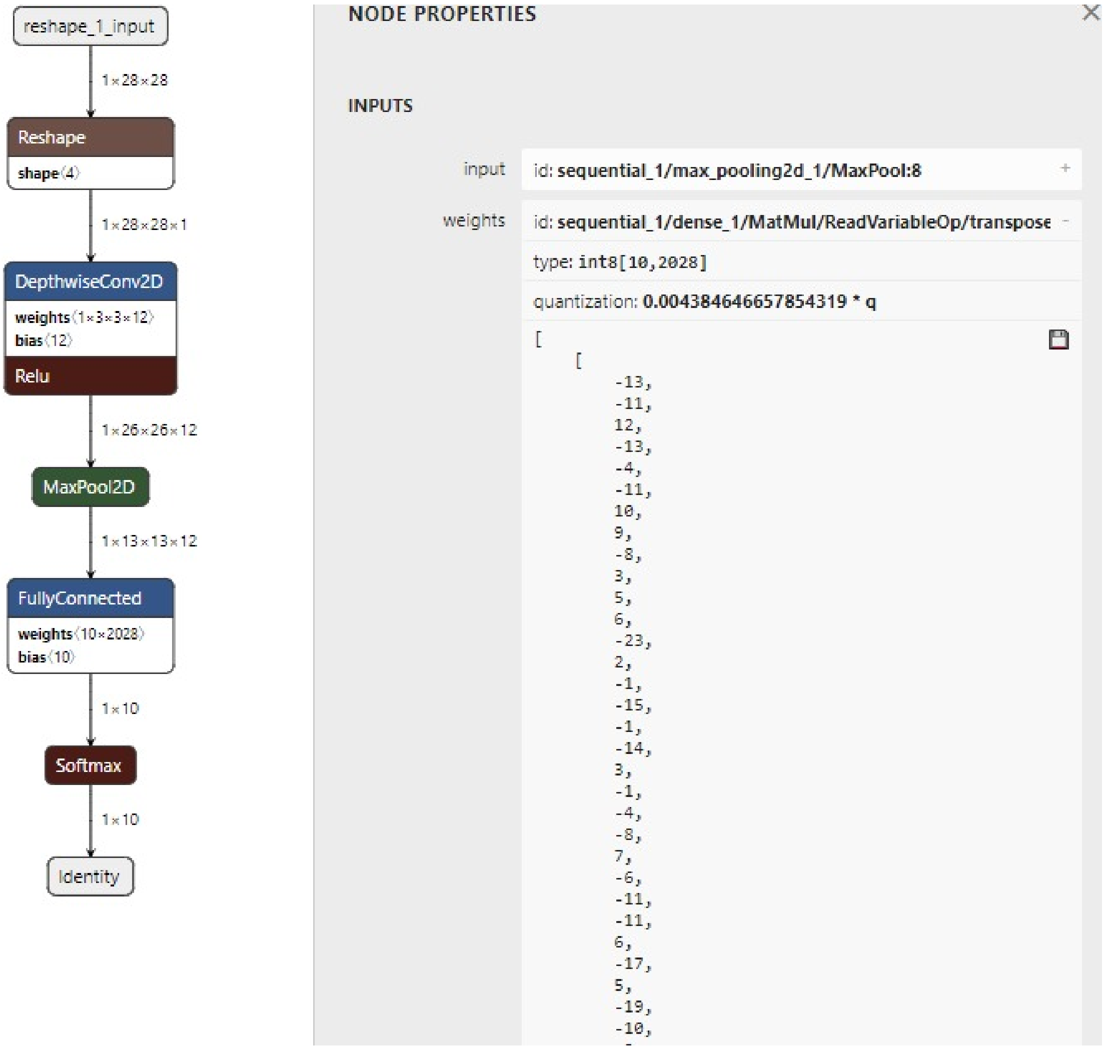
To try out this method we used a basic model with only two layers and trained it on MNIST data-set.
Here is a look at our model's parameters:



```
kernel    id: dense_1/kernel:0
          kind: Weights
          type: float32[2028,10]
          [
              [
                  -0.056328751146793365,
                  0.06529829651117325,
                  -0.06899326294660568,
                  -0.02614649571478367,
                  0.05340256541967392,
                  0.021474769338965416,
                  0.031633440405130386,
                  0.0415845587849617,
                  0.0020426560658961535,
                  0.02886672131717205
              ],
              [
                  -0.049032777547836304,
                  0.06920456886291504,
                  0.00818085391074419,
                  -0.0768525302410257,
                  -0.06423936039209366,
                  0.0660289078950882,
                  -0.013891101814806461,
                  0.06727545708417892,
                  -0.0515119694173336,
                  0.0025704249273985624
              ],
              [
                  0.05230715498328209,
                  0.10120117664337158,
                  -0.016991477459669113,
                  -0.10662072151899338,
                  0.0675579160451889,
                  -0.07499262690544128,
                  0.018548134714365005,
                  0.027584612369537354,
                  -0.07421279698610306,
                  -0.003418984590098262
              ],
```

Figure 39: Weights before QAT

after using the Quantization-aware training technique we ended up with this model.



NODE PROPERTIES

INPUTS

input id: sequential_1/max_pooling2d_1/MaxPool:8

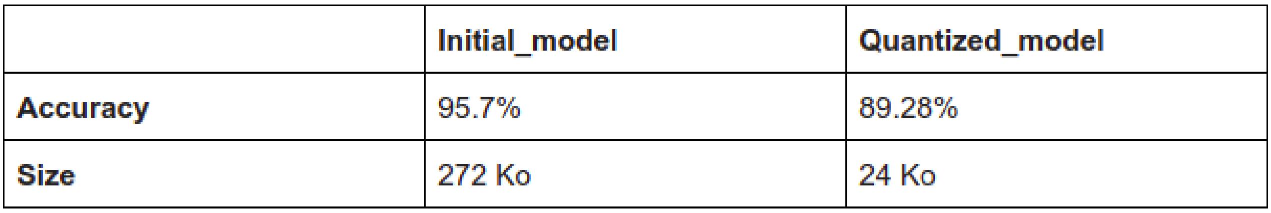
weights id: sequential_1/dense_1/MatMul/ReadVariableOp/transpose

type: int8[10,2028]

quantization: 0.004384646657854319 * q

```
[
    [
        -13,
        -11,
        12,
        -13,
        -4,
        -11,
        10,
        9,
        -8,
        3,
        5,
        6,
        -23,
        2,
        -1,
        -15,
        -1,
        -14,
        3,
        -1,
        -4,
        -8,
        7,
        -6,
        -11,
        -11,
        6,
        -17,
        5,
        -19,
        -10,
```

Figure 40: Weights after QAT



	Initial_model	**Quantized_model**
Accuracy	95.7%	89.28%
Size	272 Ko	24 Ko

Figure 41: Model after quantization-aware training

- **Comparison between the three quantization techniques:**

Post-training quantization and Quantization-aware training transform all weights as well as activation functions to 8-bit integers which makes our model compatible with some microprocessors and microcomputers that support only 8-bit integer operations.

Besides,our model can be further improved by using EDGE TPU which works only on full-integer models (using Post-training quantization or Quantization-ware training).

Post-training quantization schemes often require a calibration step to determine scaling factors, but it may be error-prone if a good representative sample is not used. QAT improves on this process by maintaining statistics needed to choose good scaling factors.

2.7 Accelerator Compatibility

This new model now uses quantized input and output, making it compatible with more accelerators, such as the Coral Edge TPU.

- What is the Edge TPU?

The Edge TPU is a small ASIC designed by Google that provides high performance Deep Learning inferencing for low-power devices.

- What is the Edge TPU's processing speed?

An individual Edge TPU can perform 4 trillion (fixed-point) operations per second, using only 2 Watts of power.[11]

- Creating a TensorFlow Lite model for the Edge TPU

We need to convert our model to TensorFlow Lite and it must be quantized using either Quantization-aware training or full integer post-training Quantization to set both the input and output type to uint8.

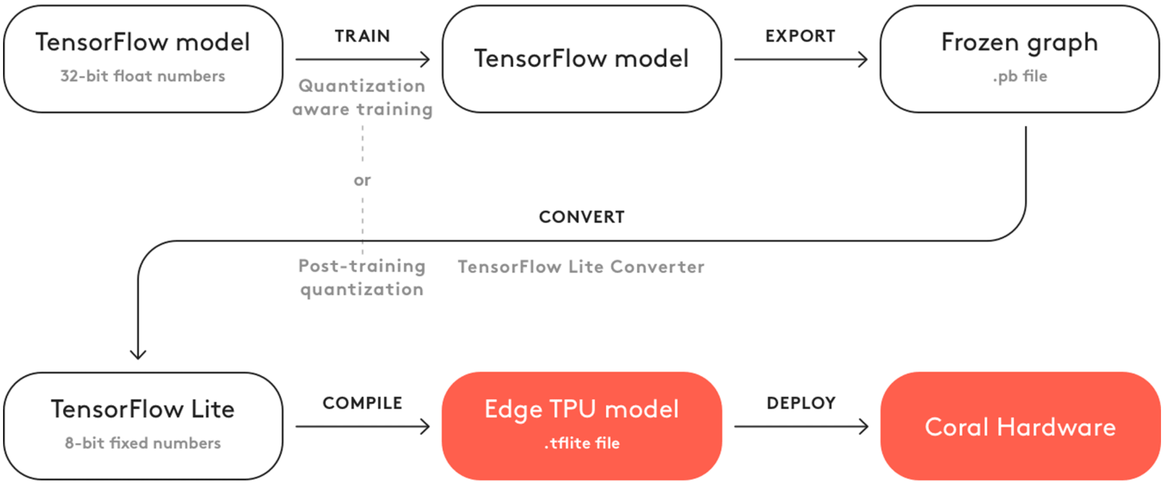


Figure 42: Edge TPU Workflow

- Compiling

After we train and convert our model to TensorFlow Lite (with quantization) the final step is to compile it with the Edge TPU Compiler.
If our model does not meet all the requirements listed at the top of this section, it can still compile, but only a portion of the model will execute on the Edge TPU. At the first point in the model graph where an unsupported operation occurs, the compiler partitions the graph into two parts. The first part of the graph that contains only supported operations is compiled into a custom operation that executes on the Edge TPU, and everything else executes on the CPU

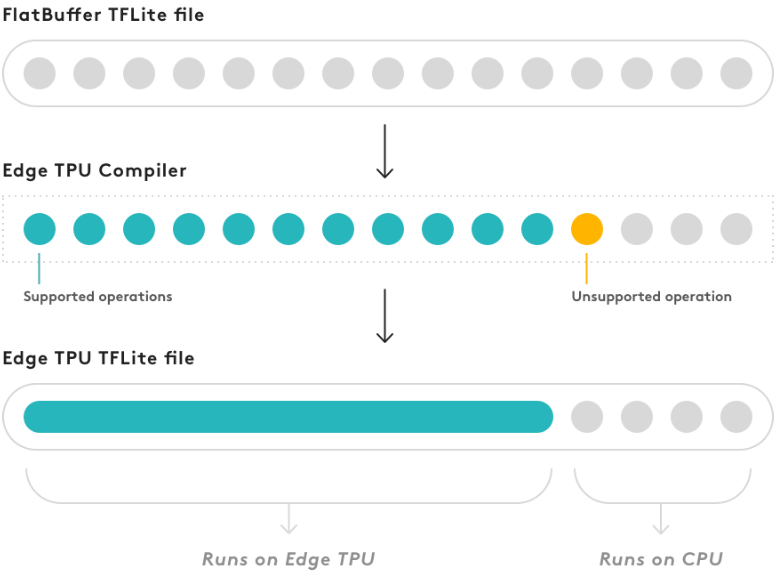


Figure 43: Edge TPU wrong compilation

2.8 Implementation and Deployment on Hardware

To implement our model, we are using the Zynq-7000 SoC ZC702 Evaluation Kit. This system on chip includes all the basic components of hardware, design tools, IPs, and pre-verified reference designs including a targeted design, enabling a complete embedded processing platform. We made this choice because Xilinx® boards offer the DPU feature:
The Xilinx® Deep Learning Processor Unit (DPU) is a programmable engine dedicated for convolutional neural network. The unit contains register configure module, data controller module, and convolution computing module. There is a specialized instruction set for DPU, which enables DPU to work efficiently for many convolutional neural networks. The deployed convolutional neural network in DPU includes VGG, ResNet, GoogLeNet, YOLO, SSD, MobileNet, FPN, etc.
Along with the board, we are using many Xilinx® services listed below

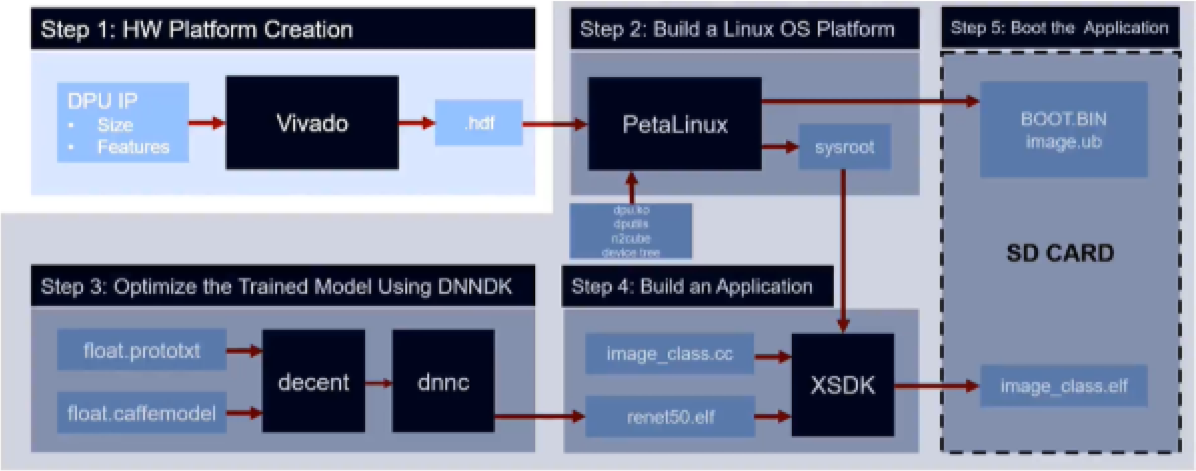


Figure 44: Xilinx® Implementation Steps

2.8.1 Step 1 : Hardware Platform Creation

Vivado® Design Suite is a software suite produced by Xilinx® for synthesis and analysis of HDL designs. We use it to generate a bit file (.hdf) containing the hardware description file.
Below is the Hardware Design for our system which uses the DPU v1.4.0 to accelerate our Deep Learning model.

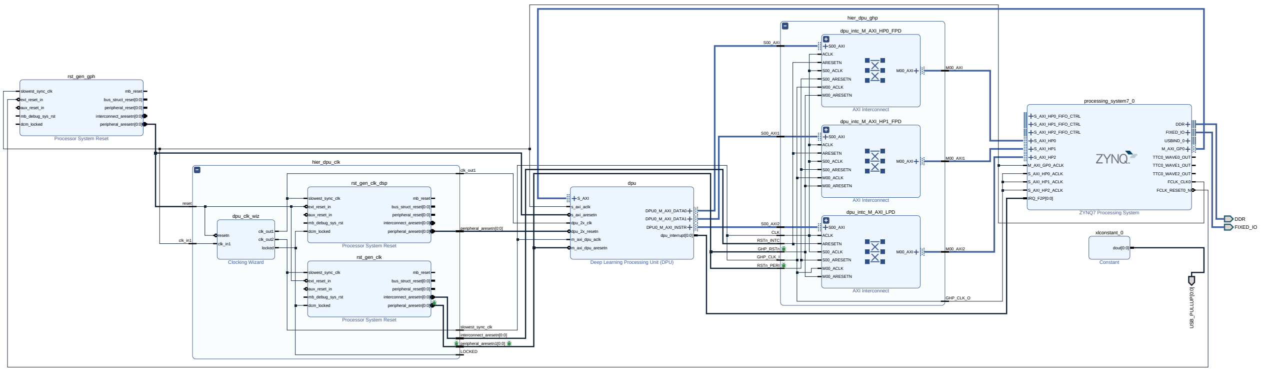


Figure 45: Block Design

- IP configuration and Port creation

We must be very careful when customizing the IP blocks, because any misconfiguration will generate errors when we try to validate the design.
Here are the configurations for the Processing System IP:

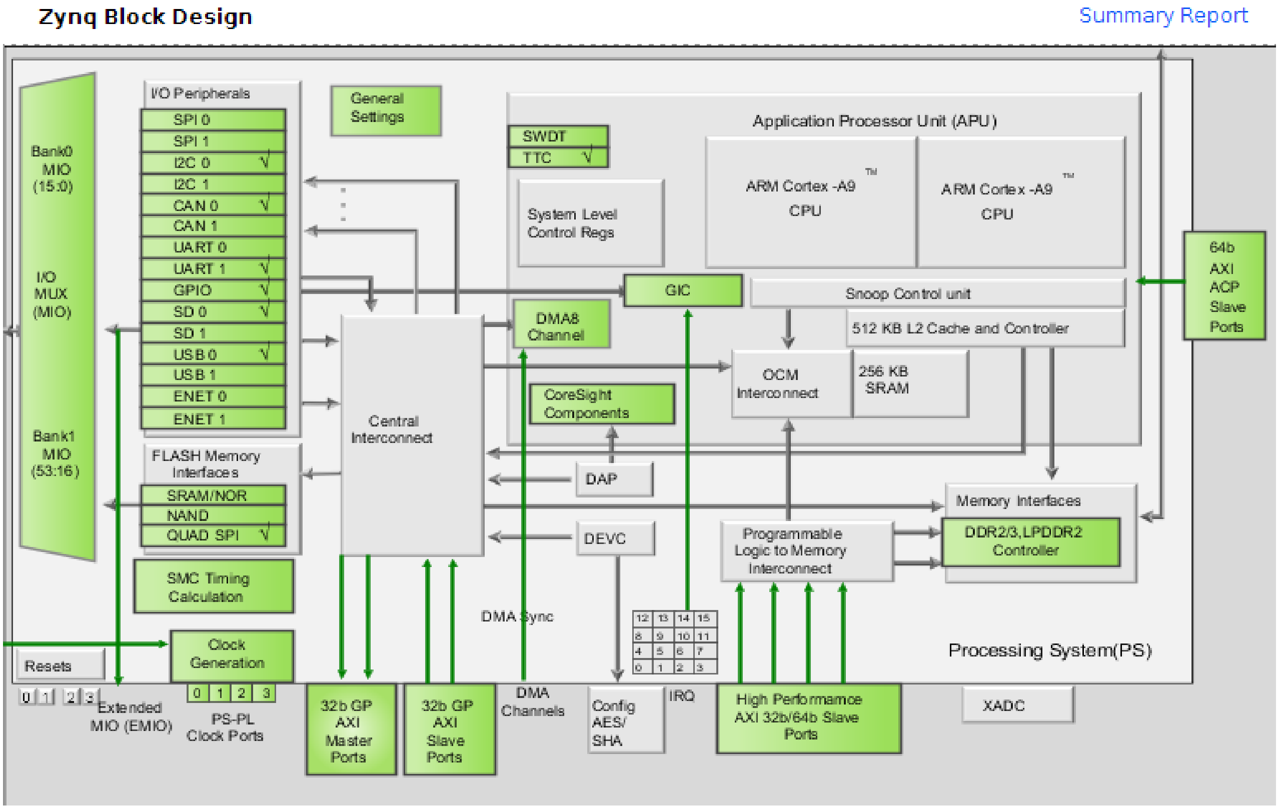
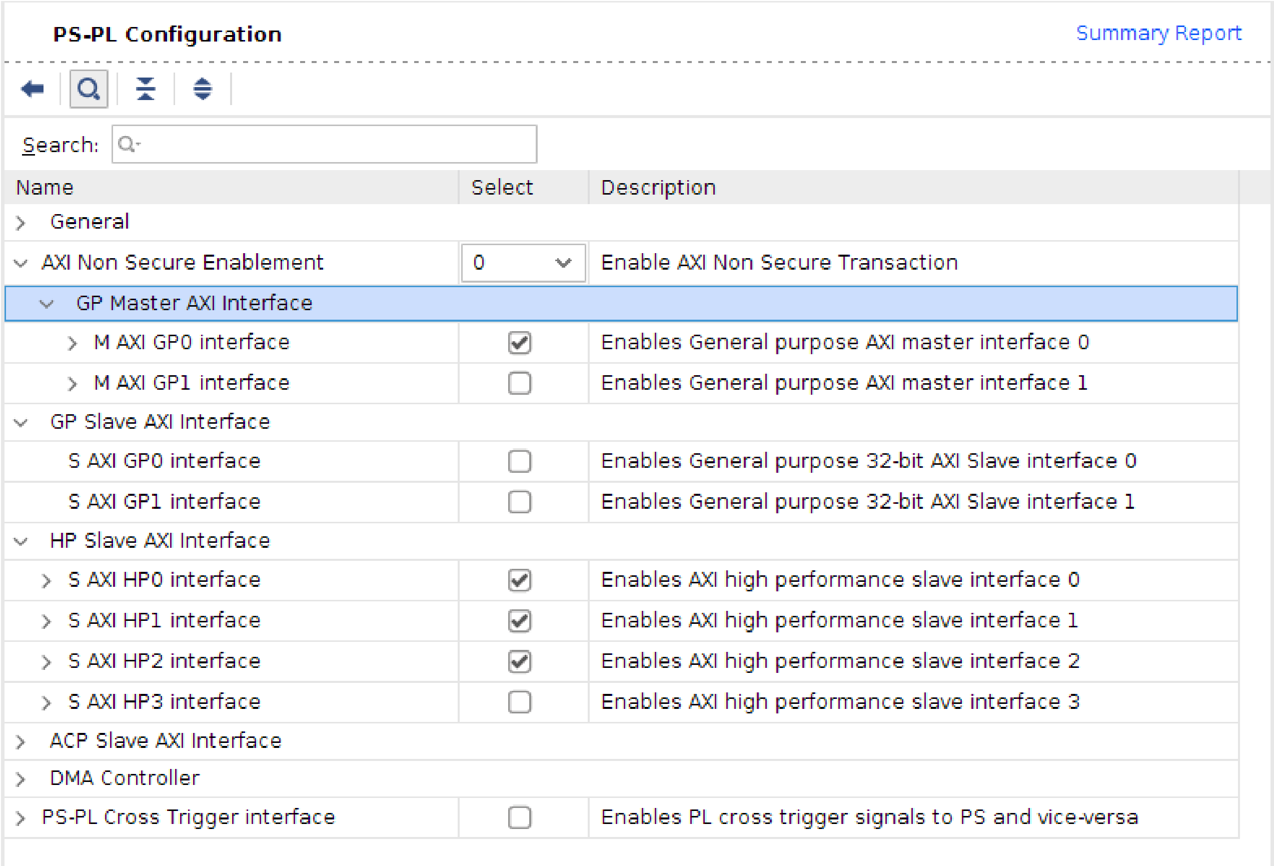


Figure 46: PS IP configuration

From the PS/PL section in the PS IP customization dialog box we must select only the slave and master interfaces we need. We must also specify the data width mode (32 bits or 64 bits) for the interfaces we select to avoid any width mismatching.



PS-PL Configuration

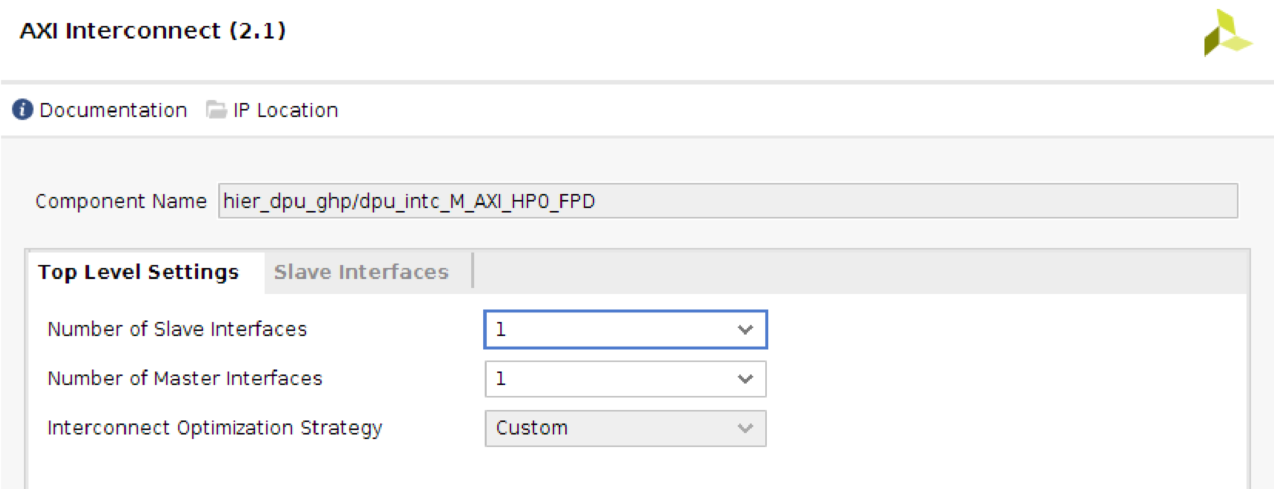
Search:

Name	Select	Description
General		
AXI Non Secure Enablement	0	Enable AXI Non Secure Transaction
GP Master AXI Interface		
M AXI GP0 interface	☑	Enables General purpose AXI master interface 0
M AXI GP1 interface	□	Enables General purpose AXI master interface 1
GP Slave AXI Interface		
S AXI GP0 interface	□	Enables General purpose 32-bit AXI Slave interface 0
S AXI GP1 interface	□	Enables General purpose 32-bit AXI Slave interface 1
HP Slave AXI Interface		
S AXI HP0 interface	☑	Enables AXI high performance slave interface 0
S AXI HP1 interface	☑	Enables AXI high performance slave interface 1
S AXI HP2 interface	☑	Enables AXI high performance slave interface 2
S AXI HP3 interface	□	Enables AXI high performance slave interface 3
ACP Slave AXI Interface		
DMA Controller		
PS-PL Cross Trigger interface	□	Enables PL cross trigger signals to PS and vice-versa

Figure 47: PS IP configuration(*)

Below are the customizations of the other IPs we had to configure like the AXI_Interconnect IPs and the clocking wizard.

The AXI_Interconnect IP is an IP that manages Masters and Slaves therefore we must specify the number of Master Interfaces and Slave interfaces to match our needs.



AXI Interconnect (2.1)

Component Name: hier_dpu_ghp/dpu_intc_M_AXI_HP0_FPD

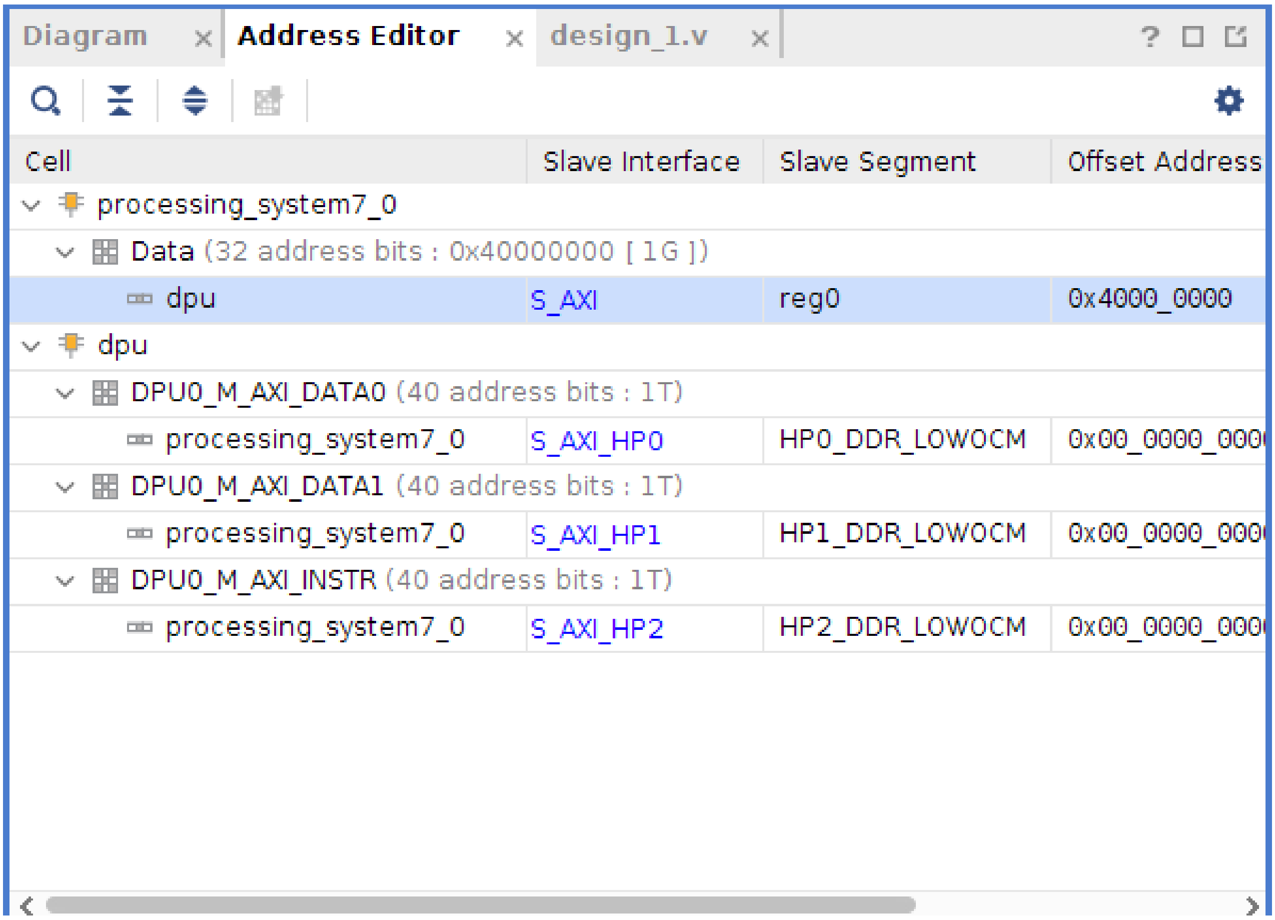
Number of Slave Interfaces: 1

Number of Master Interfaces: 1

Interconnect Optimization Strategy: Custom

Figure 48: AXI_Interconnect IP configuration

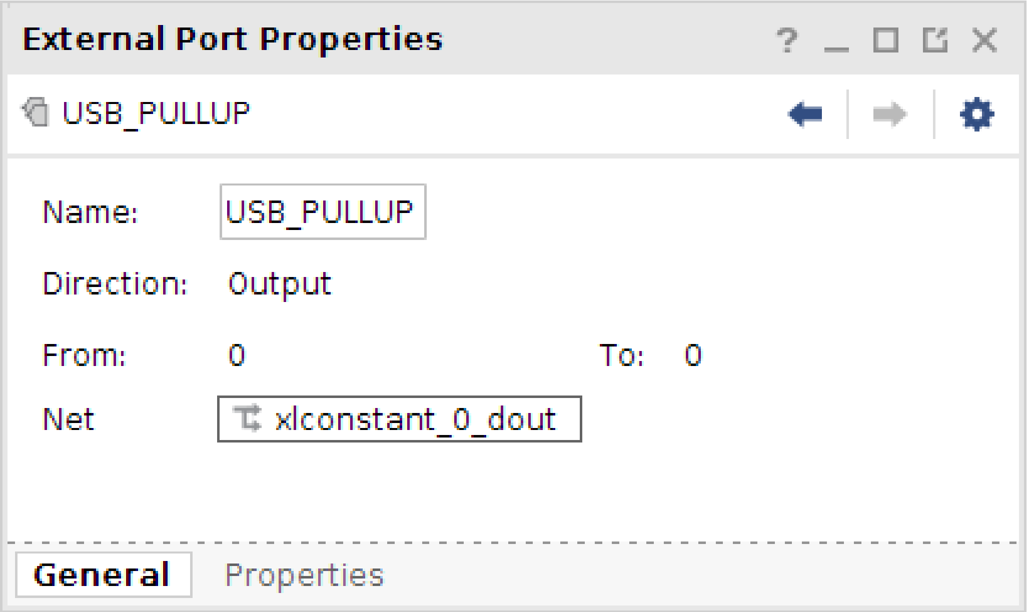
We must also pay attention to the offset addresses of AXI Slaves and AXI Masters because any overlapping between slave addresses or mismatching between slave and master address range or overlapping between slave addresses will generate errors during the validation process. For that purpose we must run the **"Auto Assign Address"** feature in the Address Editor.



Cell	Slave Interface	Slave Segment	Offset Address
processing_system7_0			
Data (32 address bits : 0x40000000 [1G])			
dpu	S_AXI	reg0	0x4000_0000
dpu			
DPU0_M_AXI_DATA0 (40 address bits : 1T)			
processing_system7_0	S_AXI_HP0	HP0_DDR_LOWOCM	0x00_0000_000
DPU0_M_AXI_DATA1 (40 address bits : 1T)			
processing_system7_0	S_AXI_HP1	HP1_DDR_LOWOCM	0x00_0000_000
DPU0_M_AXI_INSTR (40 address bits : 1T)			
processing_system7_0	S_AXI_HP2	HP2_DDR_LOWOCM	0x00_0000_000

Figure 49: Address Editor

For the ports we have manually created like the USB_PULLUP port we must specify the Name, Type and Direction as well the Vector range and make it an external port by selecting **Make External** in the Context Menu.



USB_PULLUP

Name: USB_PULLUP

Direction: Output

From: 0 To: 0

Net xlconstant_0_dout

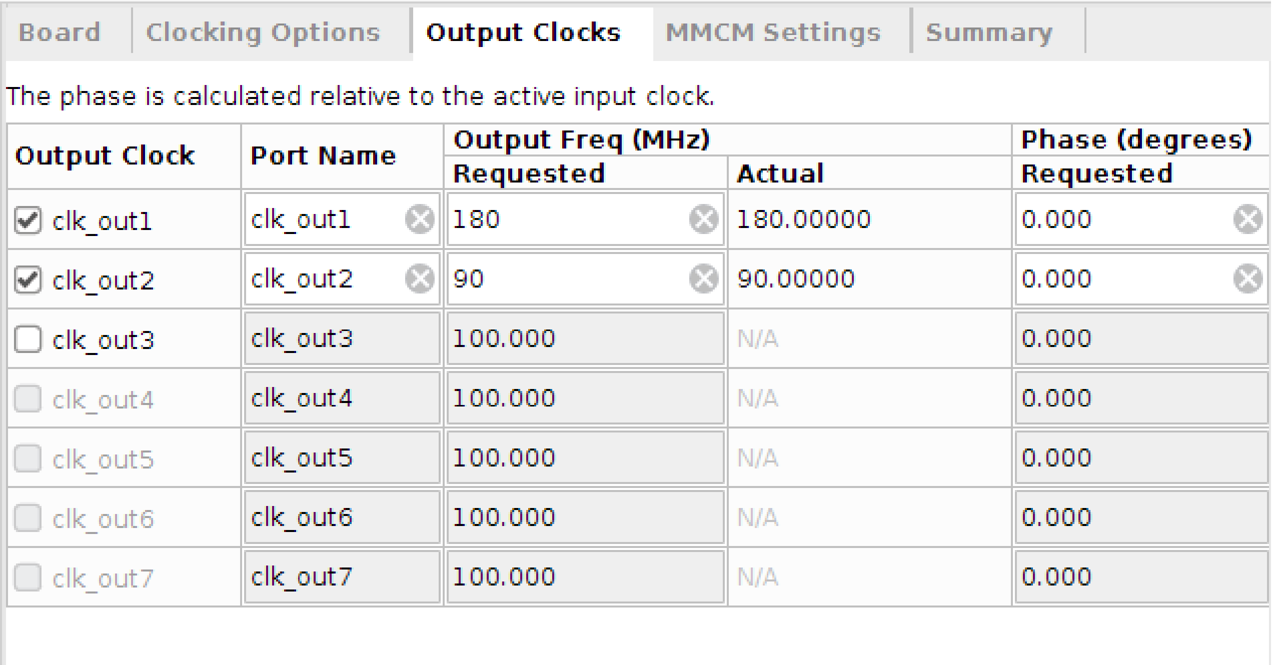
Figure 50: USB_PULLUP configuration

Next we must specify the physical constraint for the USB_PULLUP port by creating a new constraint and writing the following lines of code in the .xdc file. This informs the software what physical pins on the FPGA we plan on using or connecting to:

```
set_property LOC G17 [get_ports USB_PULLUP]
set_property IOSTANDARD LVCMOS25 [get_ports USB_PULLUP]
```

Listing 1: USB_PULLUP port constraint specification

For the Clocking Wizard in the Output Clocks section we must configure the output clocks **"clk_out1"** and **"clk_out2"** as such:

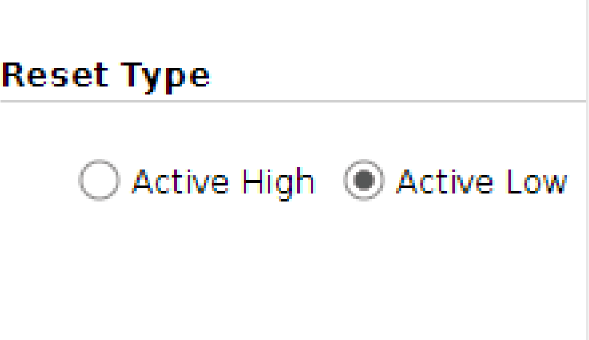


The phase is calculated relative to the active input clock.

Output Clock	Port Name	Output Freq (MHz)		Phase (degrees)
		Requested	Actual	Requested
☑ clk_out1	clk_out1	180	180.00000	0.000
☑ clk_out2	clk_out2	90	90.00000	0.000
□ clk_out3	clk_out3	100.000	N/A	0.000
□ clk_out4	clk_out4	100.000	N/A	0.000
□ clk_out5	clk_out5	100.000	N/A	0.000
□ clk_out6	clk_out6	100.000	N/A	0.000
□ clk_out7	clk_out7	100.000	N/A	0.000

Figure 51: Clocking Wizard Output Clocks configuration

In the same section we must switch from the **"reset"** port to the **"resetn"** port by selecting **"ACTIVE_LOW"** because it is linked to other active low pins like the **"peripheral_aresetn[0:0]"** pin in the **"rst_gen_gph"** PS Reset IP and the **"ext_reset_in"** pins in the **"rst_gen_clk_dsp"** and the **"rst_gen_clk"** PS Reset IPs.



Reset Type

○ Active High ● Active Low

Figure 52: Clocking Wizard configuration

- Creating the HDL Wrapper

Once our design in successfully validated, we new create the HDL wrapper which will generate the Verilog script that describes our Block Design.
Here is the Verilog code snippet describing the ”hier_dpu_clk” Hierarchy.

```
module hier_dpu_clk_imp_RQUEN2
   (LOCKED,
    clk_in1,
    clk_out1,
    interconnect_aresetn,
    peripheral_aresetn,
    peripheral_aresetn1,
    reset,
    slowest_sync_clk);
  output LOCKED;
  input clk_in1;
  output clk_out1;
  output [0:0]interconnect_aresetn;
  output [0:0]peripheral_aresetn;
  output [0:0]peripheral_aresetn1;
  input reset;
  output slowest_sync_clk;

  wire Net;
  wire clk_wiz_0_clk_out1;
  wire clk_wiz_0_clk_out2;
  wire clk_wiz_0_locked;
  wire [0:0]rst_ps7_0_50M1_peripheral_aresetn;
  wire [0:0]rst_ps7_0_50M2_interconnect_aresetn;
  wire [0:0]rst_ps7_0_50M2_peripheral_aresetn;
  wire rst_ps7_0_50M_peripheral_aresetn;

  assign LOCKED = clk_wiz_0_locked;
  assign Net = clk_in1;
  assign clk_out1 = clk_wiz_0_clk_out1;
  assign interconnect_aresetn[0] = rst_ps7_0_50M2_interconnect_aresetn;
  assign peripheral_aresetn[0] = rst_ps7_0_50M1_peripheral_aresetn;
  assign peripheral_aresetn1[0] = rst_ps7_0_50M2_peripheral_aresetn;
  assign rst_ps7_0_50M_peripheral_aresetn = reset;
  assign slowest_sync_clk = clk_wiz_0_clk_out2;
  design_1_clk_wiz_0_1 dpu_clk_wiz
       (.clk_in1(Net),
        .clk_out1(clk_wiz_0_clk_out1),
        .clk_out2(clk_wiz_0_clk_out2),
        .locked(clk_wiz_0_locked),
        .resetn(rst_ps7_0_50M_peripheral_aresetn));
  design_1_rst_ps7_0_50M_5 rst_gen_clk
       (.aux_reset_in(1'b1),
        .dcm_locked(clk_wiz_0_locked),
        .ext_reset_in(rst_ps7_0_50M_peripheral_aresetn),
        .interconnect_aresetn(rst_ps7_0_50M2_interconnect_aresetn),
        .mb_debug_sys_rst(1'b0),
        .peripheral_aresetn(rst_ps7_0_50M2_peripheral_aresetn),
        .slowest_sync_clk(clk_wiz_0_clk_out2));
  design_1_rst_ps7_0_50M_4 rst_gen_clk_dsp
       (.aux_reset_in(1'b1),
        .dcm_locked(clk_wiz_0_locked),
        .ext_reset_in(rst_ps7_0_50M_peripheral_aresetn),
        .mb_debug_sys_rst(1'b0),
        .peripheral_aresetn(rst_ps7_0_50M1_peripheral_aresetn),
        .slowest_sync_clk(clk_wiz_0_clk_out1));
endmodule
```

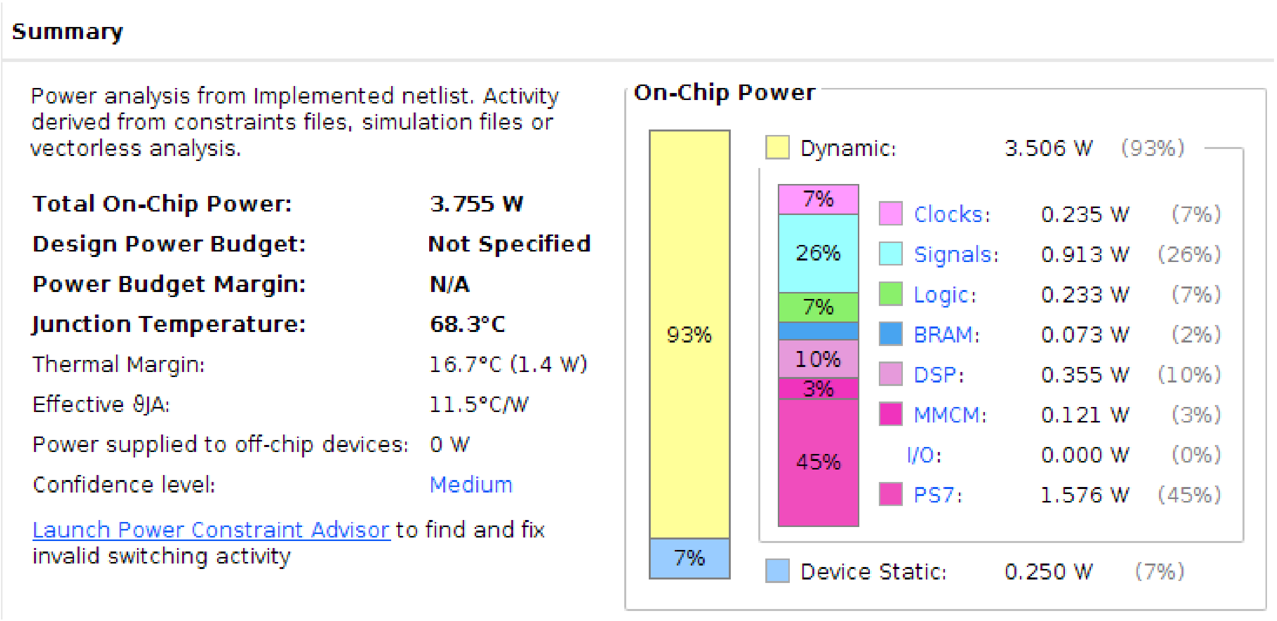
Listing 2: hier_dpu_clk Hierarchy Verilog description

- Synthesis, Implementation and Bitstream generation

After successfully creating the HDL Wrapper, we now go on to running synthesis in order to create the Synthesized Design and the Implemented Design.These steps will generate checkpoints in Project Mode.

The synthesis step converts the RTL code into the netlist, and afterwards when the synthesis is complete without any errors we run implementation in order to get the Implemented Design. The Implementation step takes the netlist as an input and uses it to do the Optimization, Placement and Routing. Once all of the previous steps have been completed without any errors we can open the Implemented Design to have an overview of the power consumption and timing reports.

In the Power section we can view the power summary report:



Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**3.755 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**68.3°C**
Thermal Margin:	16.7°C (1.4 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

On-Chip Power

Dynamic:	3.506 W	(93%)
Clocks:	0.235 W	(7%)
Signals:	0.913 W	(26%)
Logic:	0.233 W	(7%)
BRAM:	0.073 W	(2%)
DSP:	0.355 W	(10%)
MMCM:	0.121 W	(3%)
I/O:	0.000 W	(0%)
PS7:	1.576 W	(45%)
Device Static:	0.250 W	(7%)

Figure 53: Power Summary

A similar statistic can be viewed post-implementation in the Project Summary section:

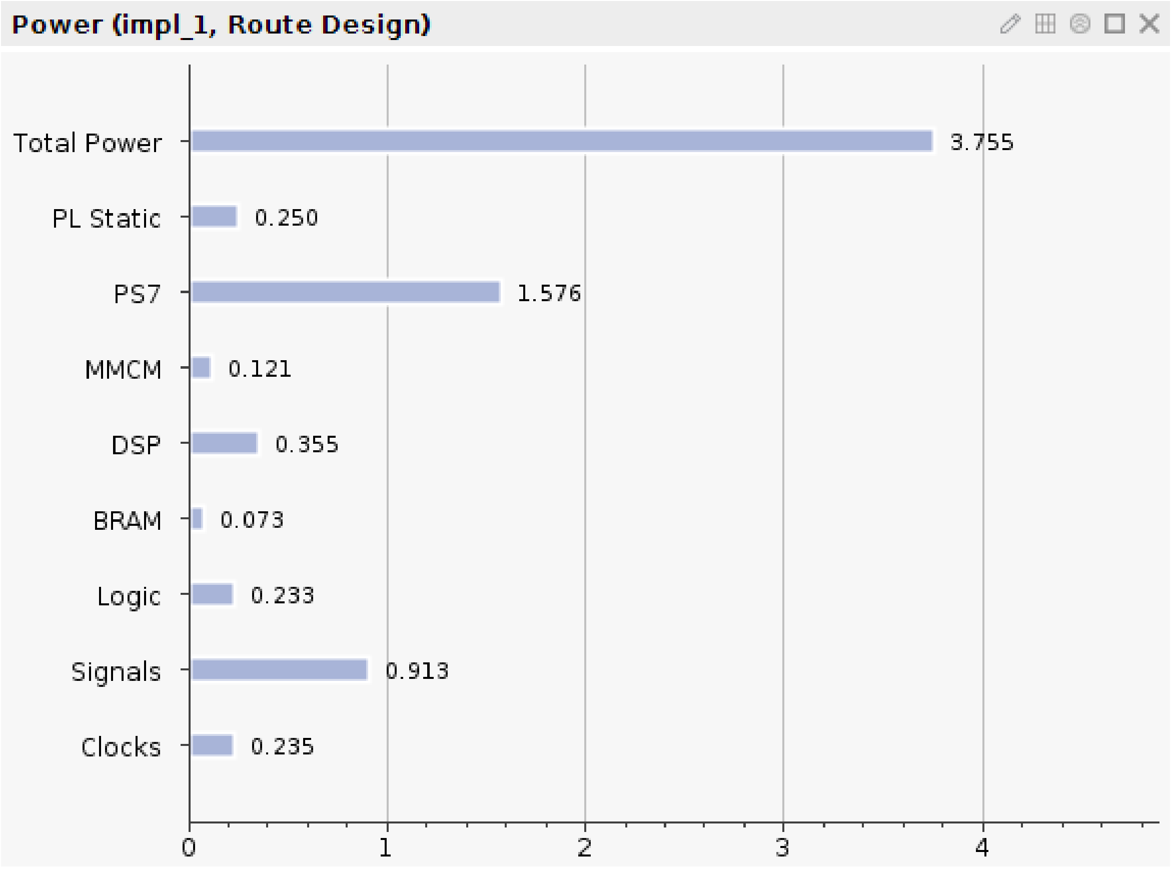


Figure 54: Power Summary(*)

In the same section we can view the utilization in percentages of each block in our FPGA.

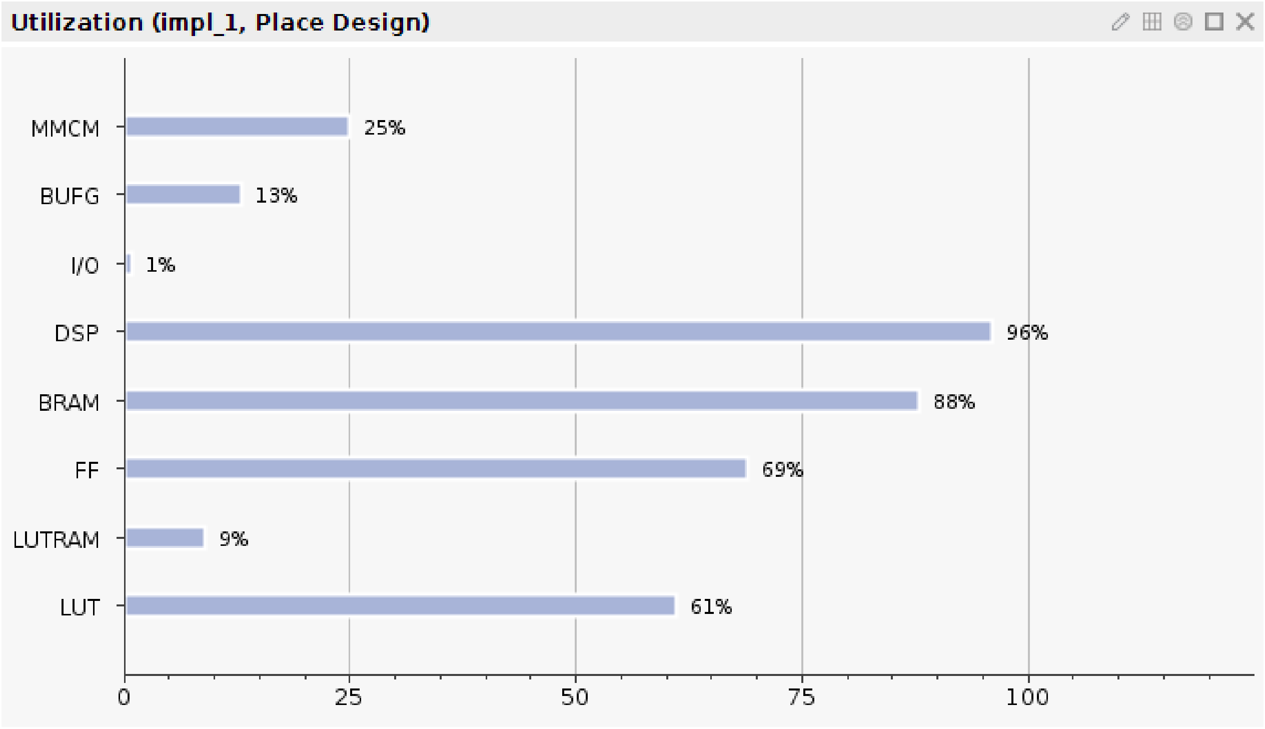
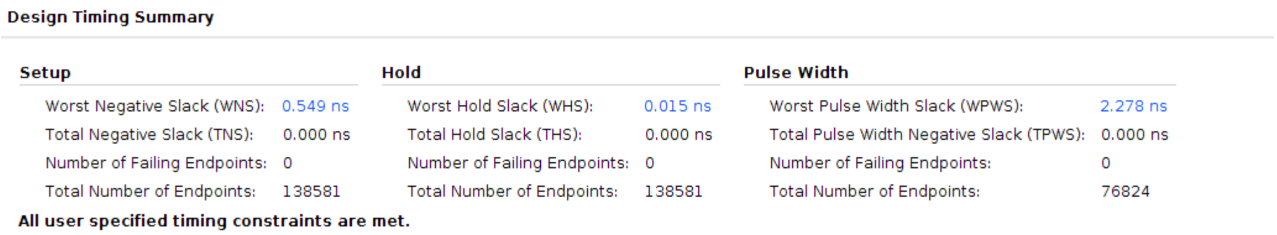


Figure 55: Utilization

In the Implemented Design we can check the Timing section for the timing summary to see if whether or not the timing constraints have been met:



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.549 ns	Worst Hold Slack (WHS):	0.015 ns	Worst Pulse Width Slack (WPWS):	2.278 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	138581	Total Number of Endpoints:	138581	Total Number of Endpoints:	76824

All user specified timing constraints are met.

Figure 56: Timing Summary

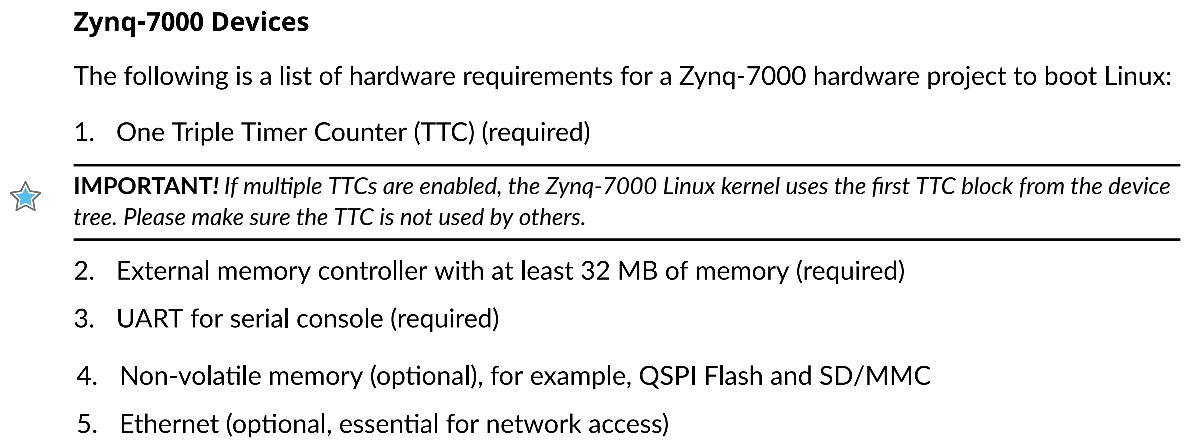
The last step is to export the hardware specification, which we will use in the next steps, by clicking **File** > **Export** > **Export Hardware** and selecting the Include Bitstream check box.

2.8.2 Step 2 : Build a linux OS Platform

We are creating a software platform using Petalinux. Petalinux is a tool you use to customise, build and deploy Embedded Linux solutions on Xilinx® processing systems. It is an Embedded Linux Systems Development Kit that targets Xilinx® SoC designs. Petalinux is only compatible with Linux distributions (CentOS, Ubuntu or Redhat)[3] so it requires setting up a virtual machine on a Windows.

1. **Checking Requirements:**

Before we get started with this step we must check that our Hardware Platform created using Vivado® is Linux ready. Below are the hardware requirements for a Zynq-7000 hardware project to boot Linux according to the Xilinx® Petalinux User Guide:



Zynq-7000 Devices

The following is a list of hardware requirements for a Zynq-7000 hardware project to boot Linux:

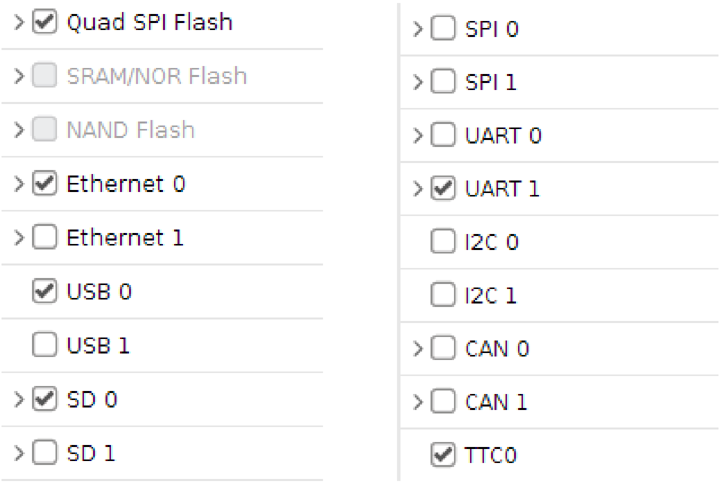
1. One Triple Timer Counter (TTC) (required)

IMPORTANT! *If multiple TTCs are enabled, the Zynq-7000 Linux kernel uses the first TTC block from the device tree. Please make sure the TTC is not used by others.*

2. External memory controller with at least 32 MB of memory (required)
3. UART for serial console (required)
4. Non-volatile memory (optional), for example, QSPI Flash and SD/MMC
5. Ethernet (optional, essential for network access)

Figure 57: Vivado® Hardware Platform requirements[3]

To make sure our Vivado® Hardware Platform was Linux ready we opened our Vivado® and in the Block Design checked if our PS IP customization matched the requirements. In the Peripheral I/O Configuration tab of the customization menu we could see that our configuration was valid:



- ☑ Quad SPI Flash
- □ SRAM/NOR Flash
- □ NAND Flash
- ☑ Ethernet 0
- □ Ethernet 1
- ☑ USB 0
- □ USB 1
- ☑ SD 0
- □ SD 1

(a)

- □ SPI 0
- □ SPI 1
- □ UART 0
- ☑ UART 1
- □ I2C 0
- □ I2C 1
- □ CAN 0
- □ CAN 1
- ☑ TTC0

(b)

Figure 58: PS IP Peripheral I/O Configuration

2. **Creating PetaLinux Project:**

The first step for building the Linux OS Platform is to create a new PetaLinux project under our directory for Vivado® projects with the following command:

```
user@user:~/Vivado_Projects$ petalinux-create -t project -n Peta
linux_HW_DR --template zynq
```

This command creates a new PetaLinux project at the specified location of type ”project” specified with the -t option with a specified name ”Petalinux_HW_DR” specified with the -n option, the –template option assumes the specified CPU architecture which is Zynq in our case.

3. **Configuring PetaLinux Project**

But before we can run the configuration command we must add the packages we will use in this project, this is done by modifying the *user-rootfsconfig* and adding the following lines:

```
#Note: Mention Each package in individual line
#These packages will get added into rootfs menu entry

CONFIG_gpio-demo
CONFIG_peekpoke
CONFIG_autostart
CONFIG_dpu
CONFIG_python-cffi
CONFIG_python-pycparser
CONFIG_libffi
```

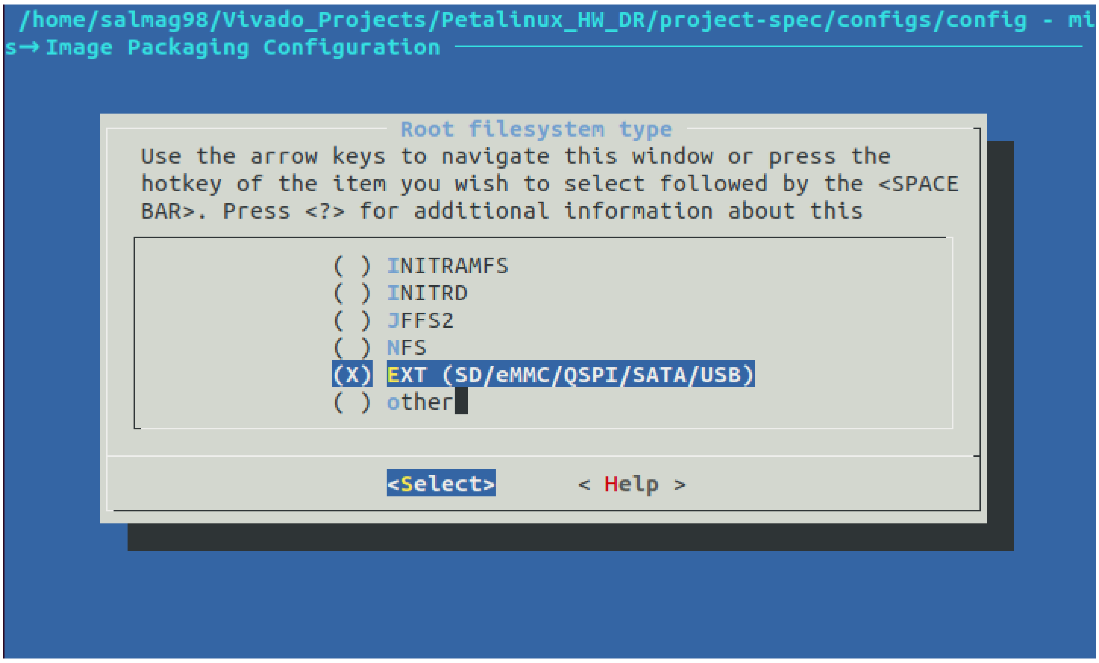
then we have to configure our PetaLinux project in accordance with our Vivado® Hardware Platform this is done by running the following command in the PetaLinux project directory:

```
user@user:~/Vivado_Projects/Petalinux_HW_DR$ petalinux-config --
get-hw-description=Vivado_Projects/Diabetic_Rethinopathy_HW
_description
```

This launches the top system configuration menu when petalinux-config –get-hw- description runs the first time for the PetaLinux project or the tool detects there is a change in the system primary hardware candidates.

We have to configure the Image Packaging by selecting the Root filesystem type. Under **Image Packaging Configuration** > we have to select Root filesystem type.

Then we have to set the filesystem type to **EXT** by selecting the option **EXT (SD,eMMC, QSPI,SATA,USB)** and the exiting two times to return to the main menu:



```
 /home/salmag98/Vivado_Projects/Petalinux_HW_DR/project-spec/configs/config - mi
s→ Image Packaging Configuration

                          Root filesystem type
   Use the arrow keys to navigate this window or press the
   hotkey of the item you wish to select followed by the <SPACE
   BAR>. Press <?> for additional information about this

                  ( ) INITRAMFS
                  ( ) INITRD
                  ( ) JFFS2
                  ( ) NFS
                  (X) EXT (SD/eMMC/QSPI/SATA/USB)
                  ( ) other

                  <Select>      < Help >
```

Figure 59: Root filesystem type

Finally in the main menu we must enter the **DTG Settings** menu to specify the device name by replacing ”template” in the **MACHINE_NAME** field by ”zc702”.

After adding the autostart, dnndk and recipes-modules to the project directory that will help us build libraries and drivers we ca now start configuring the **root file-system** by executing the following command ; the *-c* or - *-component* option is mutually exclusive with the - *-get-hw-description* option that configures the entire project, because the *-c* option tells the config command to configure a specific component, which means that in the following step we will be configuring the **Root File-System**:

```
user@user:~/Vivado_Projects/Petalinux_HW_DR$
petalinux-config -c rootfs
```

While configuring the root file-system we must make sure under the **Petalinux Package Groups** to enable the **petalinuxgroup-petalinux-opencv** package that is used for the preprossessing of inference images and the **packagegroup-petalinux-x11**, we also need to enable **autostart** under **Apps**, **dpu** under **Modules**, and **dnndk** under **User Packages**.

X11 *is a windowing system for bitmap displays, that is used for drawing and moving windows on the display device and interacting with a mouse and keyboard, later on in our SDK application x11 will enable us to display the images that are being run through the model in the inference by using cpp x11 member functions in the main.cc file.*

Next we have to configure the **kernel** by executing the following command and expand the available memory by changing the **Kernel Features -> Maximum zone order** from 11 to 15.

Again the *-c* option here specifies that we are going to configure a specific component which is the **Kernel**.

```
user@user:~/Vivado_Projects/Petalinux_HW_DR$
petalinux-config -c kernel
```

4. **Building PetaLinux Project:**

After making sure that the DPU module is set to auto load then we can configure it using the *-c* option. The *-x* or *- -execute* option is used to run a specific build step, in our case it is the *do_install* step. Next we can build or **PetaLinux** project:

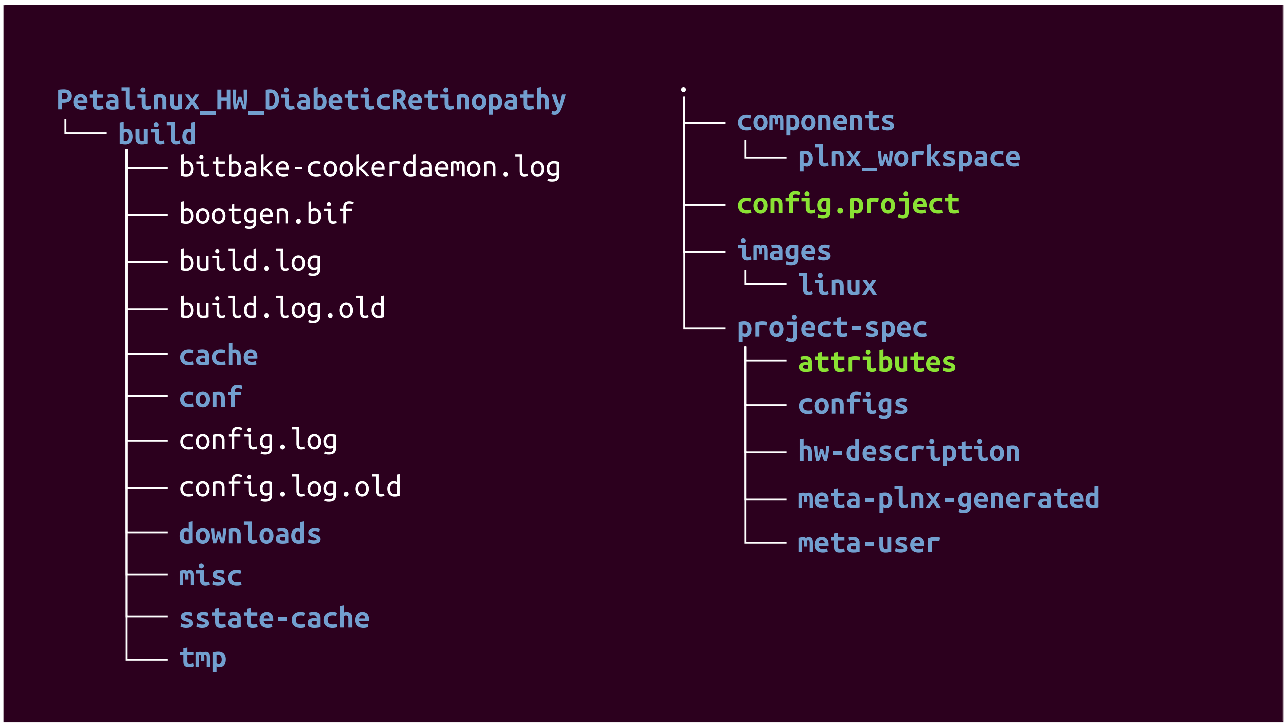
```
user@user:~/Vivado_Projects/Petalinux_HW_DR$
petalinux-build -c dpu -x do_install
user@user:~/Vivado_Projects/Petalinux_HW_DR$ petalinux-build
```

5. **Creating the Boot Image:**

Now we have to build the boot image, meaning we have to generate under the **/Petalinux_HW_DR/images/linux** the **BOOT.bin** and **image.ub** as well as the **rootfs.tar.gz** which will later be used to generate the **/ROOTFS** of our Operating System:

```
user@user:~/Vivado_Projects/Petalinux_HW_DR$
petalinux-package --boot --fsbl --u-boot --fpga
```

Here is the PetaLinux Project root directory tree obtained after the project creation and configuration:



```
Petalinux_HW_DiabeticRetinopathy
└── build
    ├── bitbake-cookerdaemon.log
    ├── bootgen.bif
    ├── build.log
    ├── build.log.old
    ├── cache
    ├── conf
    ├── config.log
    ├── config.log.old
    ├── downloads
    ├── misc
    ├── sstate-cache
    └── tmp
```

```
.
├── components
│   └── plnx_workspace
├── config.project
├── images
│   └── linux
└── project-spec
    ├── attributes
    ├── configs
    ├── hw-description
    ├── meta-plnx-generated
    └── meta-user
```

Listing 3: Petalinux Project Directory Tree

2.8.3 Step 3 : Optimize model

Embedding an AI application on a board like Xilinx® ZedBoard, requires the model to be at its optimal state, with the minimum number of variables and thus low memory usage, this is where this step comes in. Here we optimize the trained model using DNNDK (Deep Neural Network Development Kit).

Deep Neural Network Development Kit (DNNDK) is a full-stack deep learning SDK for the Deep-learning Processor Unit (DPU). It provides a unified solution for deep neural network inference applications by providing pruning, quantization, compilation, optimization, and run time support.

For this step we must also have the latest DNNDK compatible version with PetaLinux and our DPU, and with that we must be careful with the Python version and to specify the board for which we are installing DNNDK, in our case it is the **ZebBoard**. Alternatively, if the computer we are running DNNDK on is equipped with an Nvidia GPU, we can download the version that works with CUDA instead of the CPU version for better and thus faster performance.

Here is a summary of installation steps:

- Downloading and extracting the DNNDK package.
- Installing dependency libraries (Caffe).
- Installing DNNDK TensorFlow packages with Anaconda.

- Creating an Anaconda virtual environment named **decent**.
- Installing the appropriate TensorFlow DNNDK package inside the **decent** environment.
- installing the required python modules (numpy, opencv-python, sklearn, scipy, progressbar2) inside the environment using the **pip** command.
- Installing the DNNDK Host Tools by running the *install.sh* inside the **host_x86**.

The DNNDK Framework comes with the following tools:[12]

- **DECENT**

With world-leading model compression technology, we can reduce model complexity by 5x to 50x with minimal accuracy impact. Deep Compression takes the performance of our AI inference to the next level.

DECENT includes two capabilities: Coarse-Grained Pruning and quantization. These reduce the number of required operations and quantize the weights. The whole working flow of DECENT is shown below:

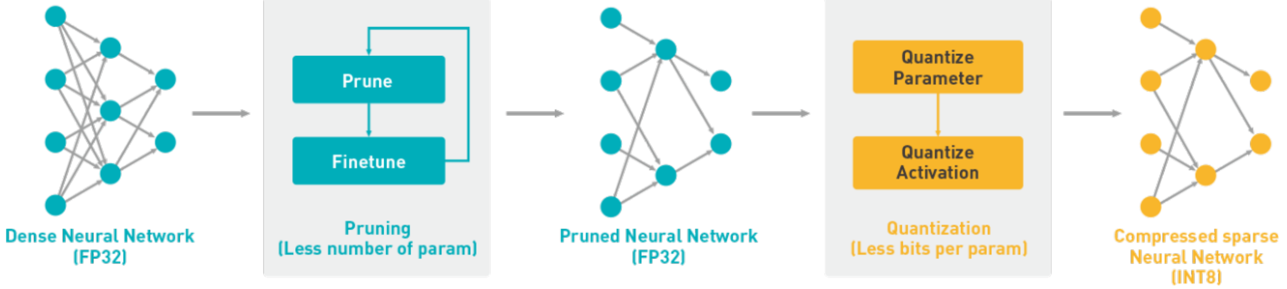


Figure 60: DECENT Pruning and Quantization Flow

- **DNNC**

The architecture of the Deep Neural Network Compiler (DNNC) compiler is shown in the following figure. The front-end parser is responsible for parsing the Caffe/TensorFlow model and generates an intermediate representation (IR) of the input model. The optimizer handles optimizations based on the IR, and the code generator maps the optimized IR to DPU instructions.

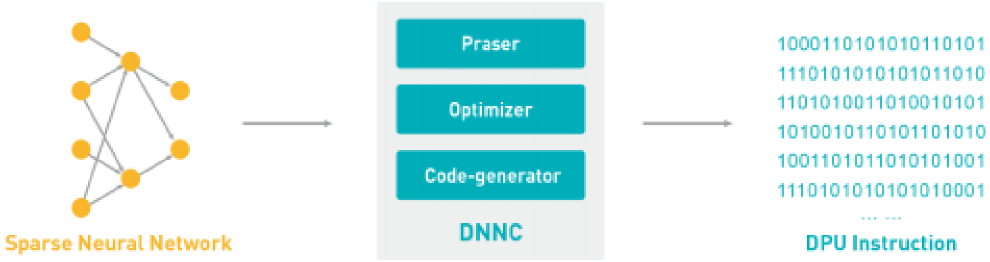


Figure 61: DNNC Components

- **DLet**

  DLet is DNNDK host tool designed to parse and extract various DPU configuration parameters from DPU Hardware Handoff file HWH generated by Vivado.

1. **Generating DCF file**

Before we can go any further with the DNNDK we must generate the DPU Configuration File of our application using the DLet tool by executing the following command:

```
user@user:~$ dlet -f /home/user/Vivado_Projects/Diabetic_Rethinopathy_HW_description/Diabetic_Rethinopathy_HW_description.srcs/sources_1/bd/design_1/hw_handoff/design_1.hwh
```

This will generate a .dcf file with the specified DCF file, DNNC compiler automatically produces DPU code instructions suited for the DPU configuration parameters.

2. **Activating Anaconda environment**

First we have to activate the *decent* Anaconda environment we created by running the following command, once inside the *decent* environment we can run python scripts that use the numpy, opencv-python, sklearn, scipy, progress-bar2 modules:

```
user@user:~$ conda activate decent
```

3. **Generate Frozen Graph**

This is done using a python script *freeze_model.py* that takes the .h5 model as an input and generates the .pb frozen graph, this is the method that freezes the graph:

```python
def freeze_session(session, keep_var_names=None, output_names=None, clear_devices=True):
    graph = session.graph
    with graph.as_default():
        freeze_var_names = list(set(v.op.name for v in tf.global_variables()).difference(keep_var_names or []))
        output_names = output_names or []
        output_names += [v.op.name for v in tf.global_variables()]
        input_graph_def = graph.as_graph_def()
        if clear_devices:
            for node in input_graph_def.node:
                node.device = ""
        frozen_graph = tf.graph_util.convert_variables_to_constants(
            session, input_graph_def, output_names, freeze_var_names)
    return frozen_graph
```

Listing 4: freeze_session method

Then in the main function of the script, the .h5 model is loaded and the frozen graph .pb file is written using these lines of code:

```python
loaded_model= keras.models.load_model('./resnet3.h5', compile=False)

# make list of output and input node names
(...)

frozen_graph = freeze_session(keras.backend.get_session(), output_names=output_names)

tf.train.write_graph(frozen_graph, "./", "frozen_graph.pb", as_text=False)
```

Listing 5: freeze_model.py main

4. **Model Compression: DECENT**

The DNNDK folder contains sample bash scripts for TensorFlow ResNet50 and Inception models we can use and customize to use the Deep Neural Network Development Kit to optimize our model.

Using the DECENT tool, we can compress our model by running the *decent_q.sh* shell script that takes the frozen graph and preprocessed calibration images.

Above is the image calibration script *"classification_input_fn"* that is in charge of preprocessing the calibration images and then calibrating the compressed model using those same images. The preprocessing is done using the python cv2 (OpenCv) module and can be seen in lines *21* to *25*, the previously mentioned section resizes the images to the *128*128* input tensor size using a cubic interpolation and rescales their pixels to the *(0,1)* range. The picture are the appended to the *images* NumPy array.

```python
1  #! /usr/bin/python3
2  # coding=utf-8
3  
4  import tensorflow as tf
5  import numpy as np
6  import cv2
7  
8  IMG_SIZE = 128
9  
10 tf.enable_eager_execution()
11 
12 calib_image_dir = "./calibration/"
13 calib_image_list = "./calibration.txt"
14 calib_batch_size = 5
15 def calib_input(iter):
16     images = []
17     line = open(calib_image_list).readlines()
18     for index in range(0, calib_batch_size):
19         curline = line[iter * calib_batch_size + index]
20         [calib_image_name, label_id] = curline.split(' ')
21         image = cv2.imread(calib_image_name)
```

```
        image = cv2.resize(image,dsize=(IMG_SIZE,IMG_SIZE), interpolation=cv2.INTER_CUBIC)
        cv2.imshow("test image", image)
        cv2.waitKey(300)
        image = image / 255.0
        images.append(image)
    images = np.array(images).reshape(-1, IMG_SIZE, IMG_SIZE, 3)
    return {"input_3": images}
```

Listing 6: Calibration images preprocessing python script

The calibration folder contains 50 images (10 images of each class) that were extracted and copied from the *validation* folder from our data-set using a short python script we wrote to copy the images and the path of each image next to its class label.

```
import os, shutil

filee = open('/home/usr/Vivado_Projects/DNNDK_HW_DR/calibration.txt','w')
given_dir = "/home/usr/Documents/PFA/data_organized/val"

for i in os.listdir(given_dir):
    for j in os.listdir(os.path.join(given_dir,i))[:10]:
        filee.write(os.path.join("./calibration",j)+' '+ i +'\n')
        shutil.copy(os.path.join(os.path.join(given_dir,i),j),"/home/usr/Vivado_Projects/DNNDK_HW_DR/calibration")
```

Listing 7: /calibration images and calibration.txt

Here are a few lines of the *calibration.txt* file that have been automaticaly written by the python script above:

```
./calibration/9a7bd084395e.png 2
./calibration/5eb311bcb5f9.png 2
./calibration/ca25745942b0.png 1
./calibration/ca1036496659.png 1
./calibration/7bf981d9c7fe.png 1
./calibration/9eaf735cf01f.png 1
./calibration/3bf3085ac167.png 0
./calibration/e04f3c6619a3.png 0
```

Listing 8: calibration.txt contents

Below is the *decent_q.sh* shell script that takes the forzen graph, and calibration images as an input and triggers the *calib_input* function in the Calibration python script:

```
decent_q quantize \
    --input_frozen_graph ./frozen_graph.pb \
    --input_nodes input_3 \
    --input_shapes ?,128,128,3 \
    --output_nodes final_output/Softmax \
    --input_fn classification_input_fn.calib_input \
    --method 1 \
    --gpu 0 \
    --calib_iter 10 \
    --output_dir ./quantize_results
```

Listing 9: decent_q.sh

This creates the *quatize_results* folder as an output in which are stored the ProtoBuf files for the quantized graph of the model that will be later on used to compile it.

5. **Model Compilation: DNNC**

The DNNC tool compiles when we execute the *compile.sh* script that takes the *deploy_model.pb* generated by the DECENT tool and the .dcf DPU Configuration File we generated using the DLet tool as an input:

```
#!/bin/bash

dnnc \
    --mode=debug \
        --parser=tensorflow \
    --frozen_pb=./quantize_results/deploy_model.pb \
    --dcf=dpu-06229-229-202006229-9-45.dcf \
    --cpu_arch=arm32\
    --output_dir=dnnc_output \
    --net_name=PFA_Model_optimized \
```

Listing 10: compile.sh

Once successfully run this script generates the Executable and Linkable File PFA_Model _optimized.elf in the **dnnc_output** folder which will later be used for the application SDK creation.

2.8.4 Step 4 : Build an application

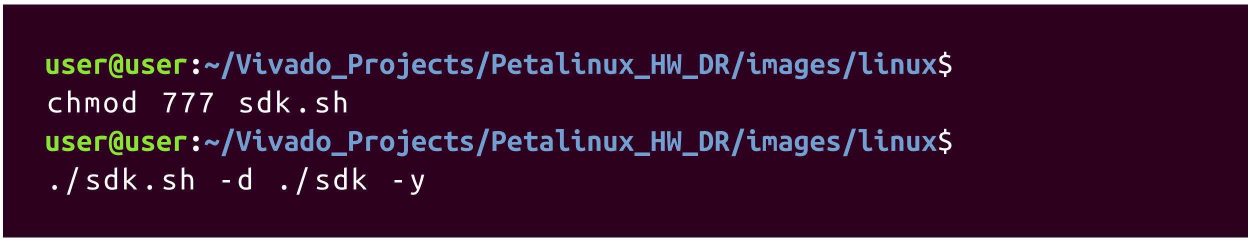
The Xilinx® Software Development Kit (XSDK) is an Integrated Development Environment (IDE) for development of embedded software applications targeted towards Xilinx® embedded processors. SDK works with hardware designs created with Vivado® Design Suite. SDK is based on the Eclipse open source standard. This section requires installing the XSDK 2019.1 which is the compatible version with our PetaLinux and Vivado platforms.

- **Generate new SDK**
  Now that the PetaLinux project has been successfully built we can run this command under the PetaLinux project directory in order to generate a new SDK:

```
user@user:~/Vivado_Projects/Petalinux_HW_DR$ petalinux-build -s
```

- **Extract SDK**

  First we have to give the *sdk.sh* file 777 permissions, meaning to make it globally readable writable and executable, then execute the shell script to generate the *sdk* folder containing the *sysroots* directory for the SDK application.



```
user@user:~/Vivado_Projects/Petalinux_HW_DR/images/linux$
chmod 777 sdk.sh
user@user:~/Vivado_Projects/Petalinux_HW_DR/images/linux$
./sdk.sh -d ./sdk -y
```

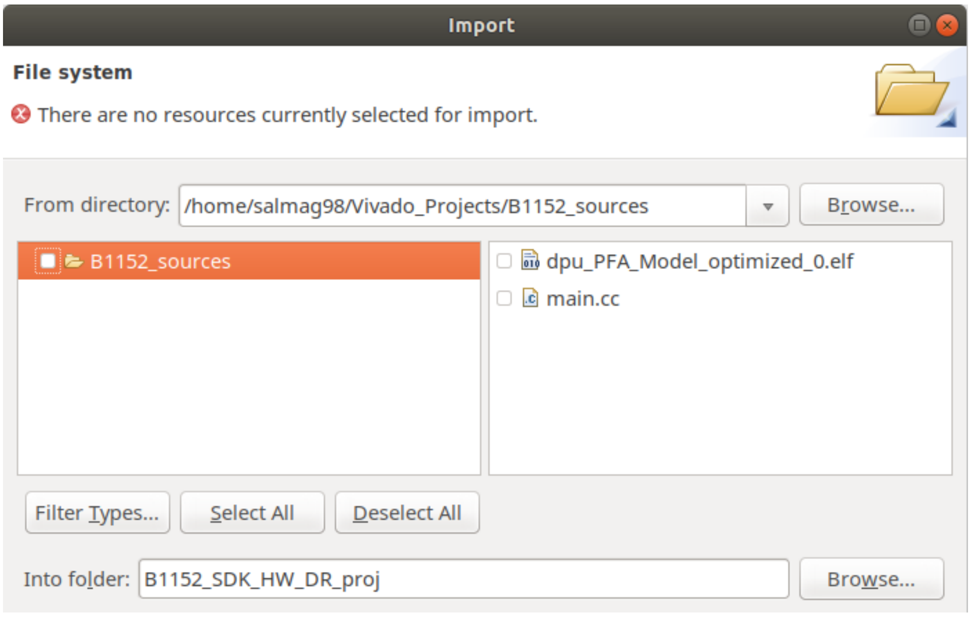
- **Create a New Application Project on XSDK**

  Below are the parameters of our application:

  - **OS Platform:** Linux
  - **Processor Type:** ps7_cortexa9
  - **Language:** C+
  - **Compiler:** 64-bit

- **Import Source Files and Model .elf Files**

  Now we must customize this new project by importing a sample *main.cc* application and the model *dpu_PFA_Model_optimized_0.elf* file generated by the DNNC in the *dnnc_output* directory after running the *compile.sh* script.



Import

File system

There are no resources currently selected for import.

From directory: /home/salmag98/Vivado_Projects/B1152_sources

□ B1152_sources

□ dpu_PFA_Model_optimized_0.elf
□ main.cc

Into folder: B1152_SDK_HW_DR_proj

Figure 62: Importing main.cc and model .elf file

- **Update the Application Build Settings and Building Project**

1. Open **C/C++ Build Settings**
2. Add **SYSROOT** to **C/C++ Build Environment** and point it to the *cortexa9t2hf-neon-xilinx-linux-gnueabi* directory under **Petalinux_HW_DR/images/linux/sdk /sysroots**
3. Point **G++ compiler** and **G++ Linker** to **SYSROOT**
4. Add the following libraries to the **G++ Linker Libraries**:
   - n2cube
   - dputils
   - opencv_core
   - opencv_imgcodecs
   - opencv_highgui
5. Add the model .elf file to the **G++ Linker Miscellaneous** in **Other Objects**
6. Exit the settings and Build the project

The main.cc script loads the inference images and the word list from the directory specified by the developer (./test) in our case rearranges the pixels in each vertically, re-scales them and feeds them to the input Tensor of the model, then runs the DPU task, finally it calls the **TopK** to output the prediction percentages for each class. Below is the DPU task code:

```
void runPFA(DPUTask *taskpfa) {
assert(taskpfa);

// number of correctly predicted images
long long Rright = 0;

/* Mean value for Diabetic Retinopathy Classification specified in
 ↪  Caffe prototxt */
vector<string> kinds, images;

/* Load all image names.*/
ListImages(baseImagePath, images);
if (images.size() == 0) {
    cerr << "\nError: No images existing under " << baseImagePath << endl;
    return;
}

/* Load all kinds words.*/
LoadWords(baseImagePath + "word_list.txt", kinds);
if (kinds.size() == 0) {
    cerr << "\nError: No words exist in file words.txt." << endl;
    return;
}

/* Get channel count of the output Tensor for Diabetic Retinopathy
 ↪  Classification Task  */
int channel = dpuGetOutputTensorChannel(taskpfa, OUTPUT_NODE);
float *softmax = new float[channel];
27      float *FCResult = new float[channel];
28      float scale_input = dpuGetInputTensorScale(taskpfa, INPUT_NODE);
29      printf("scale_input = %f;\n\r", scale_input);
30      DPUTensor *dpu_in = dpuGetInputTensor(taskpfa, INPUT_NODE);
31      int8_t *data = dpuGetTensorAddress(dpu_in);
32
33      for (auto &imageName : images) {
34          cout << "\nLoad image : " << imageName << endl;
35          /* Load image and Set image into DPU Task for Diabetic Retinopathy
             ↪  Classification */
36          Mat image = imread(baseImagePath + imageName);
37          normalize_image(image, data, scale_input);
38
39          /* Launch RetNet50 Task */
40          cout << "\nRun DPU Task  ..." << endl;
41          dpuRunTask(taskpfa);
42
43          /* Get DPU execution time (in us) of DPU Task */
44          long long timeProf = dpuGetTaskProfile(taskpfa);
45          cout << "  DPU Task Execution time: " << (timeProf * 1.0f) << "us\n";
46          float prof = (RESNET50_WORKLOAD / timeProf) * 1000000.0f;
47          cout << "  DPU Task Performance: " << prof << "GOPS\n";
48
49          /* Get FC result and convert from INT8 to FP32 format */
50          dpuGetOutputTensorInHWCFP32(taskpfa, OUTPUT_NODE, FCResult, channel);
51
52          /* Calculate softmax on CPU and display TOP-5 classification
             ↪  results */
53          CPUCalcSoftmax(FCResult, channel, softmax);
54          TopK(softmax, channel, 5, kinds, imageName.c_str(), Rright);
55
56          /* Display the impage */
57          cv::imshow("PFA classification", image);
58          cv::waitKey(1000);
59      }
60
61
62      cout << "  Number of correct images: " << Rright << " images\n";
63      cout << "  Accuracy %" << fixed << setprecision(2) <<
         ↪  (Rright/(double)365)*100 << endl;
64
65      delete[] softmax;
66      delete[] FCResult;
67  }
```

Listing 11: DPU Task

We have made a few changes to the main.cc script in order to manually calculate the model's accuracy at the end of the inference. We have declared a new *long integer* variable *Rright* (line 5) that is going to be incremented inside the TopK method each time the first digit in the image that has been fed to the network matches the Top predicted class.

This is possible because the inference images in the **/test** directory have been copied from the validation directory of our data-set and resized using this python script that we wrote and ran inside the *decent* environment after installing the *pillow*, *os* and *shutils* modules:

```python
import os, shutil
from PIL import Image

given_dir = "/home/user/Documents/PFA/data_organized/val"

for i in os.listdir(given_dir):
    for j in os.listdir(os.path.join(given_dir,i)):
        shutil.copy(os.path.join(os.path.join(given_dir,i),j),"/home/user/Documents/PFA/test")
        dst_file = os.path.join("/home/user/Documents/PFA/test", j)
        new_dst_file_name = os.path.join("/home/user/Documents/PFA/test", i+j)
        os.rename(dst_file,new_dst_file_name)

given_inference_dir = "/home/user/Documents/PFA/test"

for i in os.listdir(given_inference_dir):
    image = Image.open(os.path.join(given_inference_dir,i))
    image = image.resize((128,128))
    image.save(os.path.join(given_inference_dir,i));
```

Listing 12: Copying and preprocessing inference images

The TopK function has been modified to take two additional parameters: ImageName and Rright:

```cpp
    void TopK(const float *d, int size, int k, vector<string> &vkinds, string Imname, long long &right) {
    assert(d && size > 0 && k > 0);
    priority_queue<pair<float, int>> q;

    for (auto i = 0; i < size; ++i) {
        q.push(pair<float, int>(d[i], i));
    }

    //Increment right if the top class matches the Image class (first digit in image name)
    pair<float, int> ktop = q.top();
    string cls = to_string(ktop.second);
    if (cls[0] == Imname[0]) right++;

    for (auto i = 0; i < k; ++i) {
        pair<float, int> ki = q.top();
        printf("top[%d] prob = %-8f  name = %s\n", i, d[ki.second],
        vkinds[ki.second].c_str());
        q.pop();
    }
}
```

Listing 13: TopK method

Once all the images have been fed to the network we tell the script to print the number of correctly predicted images (line 62 DPU taks) and calculate the model's accuracy by dividing that number by the total number of inference images (line 63 DPU Task).

This method is truly a manual calculation of accuracy and it's downside is that we have to manually change line 63 each time the number of inference images changes. With some work we can improve the script to automatically retrieve the number of images in the inference (./test) directory by modifying the `ListImages(string const &path, vector<string> &images)` method.

Once the application is successfully built we can find the generated *SDK_HW_DR_proj.elf* application under the **/Debug** directory of our SDK project.

2.8.5 Step 5 : Boot the application

- **Create the Boot Image**

```
user@user:~/Vivado_Projects/Petalinux_HW_DR/images/linux$
petalinux-package --boot --fsbl --u-boot --fpga
```

  This will generate the BOOT.bin, image.ub and rootfs.tar.gz files under the **/image/linux** directory that will be used later on to create the bootable SD Card.

- **Partitioning SD Card**

  Using cp BOOT.bin image.ub /media/user/BOOT we must format or delete any existing partitions on the SD Card and create a new **BOOT** primary partition of size **1024 MiB** and of **FAT32** file-system, the remaining space on the SD Card is used for the **ROOTFS** primary partition of **Ext4** file-system.

- **Bootable SD Card**
  Once the two partitions are mounted, the last step is to extract the rootfs.tar.gz in the SD Card **ROOTFS** partition:

```
user@user:~$
sudo tar xzf rootfs.tar.gz -C /media/{user}/ROOTFS/
user@user:~$ sync
```

  Now we have to copy the BOOT.bin and image.ub in the **BOOT** partition, and our application is ready to be run on the Zynq ZC702 Board.

```
user@user:~$ cp BOOT.bin image.ub /media/{user}/BOOT
```

- **Boot the board with the created Image**

  Connect the board to the computer and insert the SD card.

  We need to set the boot mode to SD card with the right configuration of the jumpers on the board, otherwise it will try to read from the Jtag.

  Using a **Serial** session on **MobaXterm** we can extract the board's I.P address using the *ifconfig* command. Now we have to create a new **SSH** session specifying the board's I.P, once connected we will be prompted for the login and password which is root for both. Now all we have to do is run the *SDK_HW_DR_proj.elf*.

  Here is the output for one image of class 4 in MobaXterm:

```
Load image : 4d2c2f02bb313.png

Run DPU Task ...
DPU Task Execution time: 26214us
DPU Task Performance: 96.3989GOPS
top[0] prob = 0.999997 name = 4
top[1] prob = 0.000002 name = 2
top[2] prob = 0.000001 name = 1
top[3] prob = 0.000000 name = 3
top[4] prob = 0.000000 name = 0
```

3 Obtained Results and Conclusion

For a fair comparison of performances we chose to evaluate each trained model based on the same set of validation data.

The table below shows the obtained results using the different technologies and optimization methods:

Performance	Accuracy %	PC CPU	Google CPU	Raspberry Pi	Google TPU	Google GPU	DPU (ZC702)	Size(Mo)
Model								
Initial Model (.h5)	92.2	120 s	1.45 s	N/A	0.89 s	0.75 s	N/A	324
Dynamic Range Quantized Model (.tflite)	90.8	0.86 s	0.074 s	2 s	0.06 s	0.055 s	N/A	27
Integer Quantized Model (.tflite)	93.15	8 s	0.064 s	1 s	0.057 s	0.052 s	N/A	27.6
Magnitude Based Weight Pruning (.h5)	82	61.4 s	1.03 s	N/A	1.02 s	0.89s	N/A	108
DNNDK Compressed & Compiled Model (.elf)	80.27	N/A	N/A	N/A	N/A	N/A	26200 us	26.9

Table 1: Model Performance summary

3.1 Hardware Specifications:

- **Personal Computer:** 16 GB RAM.

```
Product name : Intel Core i7-6700 Processor
# of Cores : 4
# of Threads : 8
Processor Base Frequency : 2.60 GHz
Max Turbo Frequency : 3.50 GHz
Cache : 6 MB Intel® Smart Cache
Bus Speed : 8 GT/s
Intel® Turbo Boost Technology 2.0 Frequency‡ : 3.50 GHz
TDP : 45 W Configurable TPD-down : 35 W
```

- **Google CoLaboratory CPU:** Google CoLab offers 12 GB RAM with two CPUs @ 220 GHz here are their specs according to the platform:

```
vendor_id : GenuineIntel
cpu family : 6
model : 63
model name : Intel(R) Xeon(R) CPU @ 2.30GHz
stepping : 0
microcode : 0x1
cpu MHz : 2300.000
cache size : 46080 KB
physical id : 0
```

```
siblings : 2
core id : 0
cpu cores : 1
apicid : 0
initial apicid : 0
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
bogomips : 4600.00
clflush size : 64
cache_alignment : 64
address sizes : 46 bits physical, 48 bits virtual
```

- **Google Colaboratory GPU:** 120 TFLOPS

```
Product Name : NVIDIA TESLA K80 4992 NVIDIA CUDA cores with a dual-GPU
design
Up to 2.91 teraflops double-precision performance with NVIDIA GPU Boost
Up to 8.73 teraflops single-precision performance with NVIDIA GPU Boost
24 GB of GDDR5 memory
480 GB/s aggregate memory bandwidth
ECC protection for increased reliability
```

- **Google Colaboratory TPU(v2):** 64 GB High Bandwidth Memory, 180 TFLOPS

```
# of Cores : 8
8 GiB of HBM for each TPU core
One MXU for each TPU core
Up to 512 total TPU cores and 4 TiB of total memory in a TPU Pod
```

- **Raspberry Pi 3 B+:**

```
SoC: Broadcom BCM2837B0 quad-core A53 (ARMv8) 64-bit @ 1.4GHz
RAM: 1GB LPDDR2 SDRAM
Networking: Gigabit Ethernet (via USB channel), 2.4GHz and 5GHz 802.11b/-
g/n/ac Wi-Fi
```

- **DPU v3.0 on Xilinx ZC702 Board:**

```
Board Specs:

Processing System: ARM Cortex-A9 Based

Application Processor Unit (APU):
• 2.5 DMIPS/MHz per CPU
• CPU frequency: Up to 1 GHz
Caches:

• 32 KB Level 1 4-way set-associative instruction and data caches (inde-
pendent for each CPU)
• 512 KB 8-way set-associative Level 2 cache (shared between the CPUs)

On-Chip Memory:
• On-chip boot ROM
• 256 KB on-chip RAM (OCM)

DPU:
Architecture : B1152
Kernel Name : PFA _Model_optimized_0
Workload MACs : 2527.60 MOPS
DPU Task Performance (Average) : 96.3500 GOPS
```

3.2 Interpretations

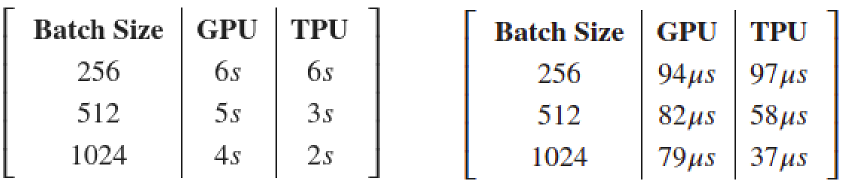
Of all the results we obtained, a few were a bit unexpected:

The inference of the initial model on RaspberryPi was not successful as the board heats up excessively when we run our model on it which made us switch off the board to prevent damage, this may be due to the fact that the initial model's size is very heavy for the RaspberryPi.

The Integer Quantized Model inference took surprisingly **more** time that Dynamic Range Quantized Model on PC CPU and we think that is may be due to the fact that this quantization method is still under development and has some compatibility issues with certain architectures.

The steep drop of accuracy in the DNNDK compressed and optimized model in comparison to the initial model and other quantization methods performance can be explained by the fact that the calibration data was not the same for the model optimized with DNNDK and Tensorflow.

Although Google CoLab boasts that it's TPU performs 15 to 30 times faster than contemporary GPUs and CPUs in inferencing, and deliveres a 30–80 times improvement in TOPS/Watt measure, in our measures the Google CoLaboratory GPU performed better in every case.



Batch Size	GPU	TPU
256	$6s$	$6s$
512	$5s$	$3s$
1024	$4s$	$2s$

(a) Epoch Time

Batch Size	GPU	TPU
256	$94\mu s$	$97\mu s$
512	$82\mu s$	$58\mu s$
1024	$79\mu s$	$37\mu s$

(b) Step Time

Figure 63: TPU vs. GPU performance with larger batch sizes

We think that this is due to the fact that TPU setup takes some time when compiling the model and distributing the data in the clusters, so the first epoch always takes more time, similarly in inferencing, since in our case we only fed one image to the network in order to test different Google CoLab Processing Units performances with our model because inferencing with all the validation data requires uploading 365 images to Google Drive and with slow internet connection it was not an option for us, eventhough it would have shown a better performance with TPU.

3.3 Perspectives

- We have yet to experiment with other DPU architectures like the B512 and B800 and B1024 because DPU architectures higher than B1152 require more DSP blocks than there are on the Zynq ZC702 board.
- We can also try to calibrate our model in the DECENT step on different calibration images and see whether the final accuracy improves.
- Try model inference on Google TPU v3.
- Try to work on different ResNet architectures that require more manual implementation and support image segmentation in order to be able to further experiment with the Network Slimming Pruning method.

Bibliography

[1] Shiva Verma. Understanding input output shapes in convolution neural network | keras. *towards data science*, 2019.

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

[3] Xilinx. *PetaLinux Tools Documentation, Reference Guide UG1144*.

[4] Marshall Hargrave. Deep learning. *Investopedia*, 2019.

[5] Oliver Faust Enas Abdulhay Victor Hugo Costa de Albuquerque, Gustavo Ramirez and Blaz Zupan. Deep learning methods for medical applications. *Artificial Intelligence in Medicine, ELSEVIER*, 2019.

[6] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, PP:1–20, 07 2019.

[7] Mayo Clinic Staff. Diabetic retinopathy. *Mayo Clinic, Patient Care & Health Information, Diseases & Conditions*, 2018.

[8] Kaggle. Diabetic retinopathy detection, overview.

[9] Derrick Mwiti. Research guide: Pruning techniques for neural networks, Nov 2019.

[10] Zhuang Liu. Learning efficient convolutional networks through network slimming, in iccv 2017. https://github.com/liuzhuang13/slimming, 2019.

[11] Coral. Frequently asked questions.

[12] Xilinx. *DNNDK User Guide UG1327*.