

UNDERGRADUATE THESIS

Classification of images using neural network and its implementation on a local FPGA board

by

Ayush Mall

A thesis submitted in partial fulfillment for the
BITS F421T Research Thesis

under

Dr. Olivier Berder

IRISA



BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI
CAMPUS

December 2019

Declaration of Authorship

I, Ayush Mall, declare that this thesis titled, ‘Classification of images using neural network and its implementation on FPGA’ and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Ayush Mall

Date: 8th December 2019

Certificate

This is to certify that the thesis entitled, “ *Classification of images using neural network and its implementation on FPGA*” and submitted by Ayush Mall ID No. 2016A3TS0163G in partial fulfillment of the requirements of BITS F421T Thesis, embodies the work done by him under my supervision.

Supervisor

Dr. Olivier Berder

IRISA

Date:

University of Rennes, CNRS, IRISA

Abstract

Dr. Olivier Berder
IRISA

Research Thesis

by Ayush Mall

This report gives an in-depth explanation of all the components of a neural network used for image classification and explains tuning and optimization of said classifiers. That includes Logistic Regression, Shallow Neural Network, Deep Neural Network and Convolutional Neural network. Using the techniques discussed, an image classifier is developed capable of operating with a high accuracy. It then expands on the procedure of implementing the classifier on an FPGA board using SensAI and discusses the complications faced and suggests possible solutions to those problems.

Acknowledgements

I would like to express my sincere gratitude to Dr. Olivier Berder, Dr. Matthieu Gautier, Mr. Mickael Le Gentil and Dr. Antoine Courtay for their constant support with the project. For any problem that i had, i got the proper guidance and suggestions. The entire lab has been a great experience for me, with the best instruments and facilities that ensured a smooth progress of my project. I have also like to extend my gratitude to my on-campus supervisor Dr. Amalin Prince who has guided me for a long time now. I would not have made it to this lab if not for him. I would lastly like to thank my parents who have always stood by my side and supported me with all my decisions emotionally and financially.

Contents

Declaration of Authorship	i
Certificate	ii
Abstract	iii
Acknowledgements	iv
List of Figures	vii
Abbreviations	viii
1 Classification using Neural Networks	1
1.1 Logistic Regression: Binary Classification	1
1.1.1 Regression Model	2
1.1.2 Loss function	2
1.1.3 Gradient Descent	2
1.1.4 Mini-batch Gradient Descent	4
1.2 Shallow Neural Networks	5
1.2.1 Neurons	5
1.2.2 Hidden Layer	6
1.2.3 Output Layer	7
1.2.4 Forward Propagation	7
1.2.5 Weight Initialization	7
1.2.6 Backward Propagation	7
1.3 Deep Neural Networks	9
1.3.1 Forward Propagation	9
1.3.2 Backward Propagation	10
1.3.3 Prediction	10
2 Hyper-parameters , Regularization and Optimization	11
2.1 Data set	11
2.1.1 Dataset Split	12
2.2 Prediction Errors	12

2.2.1	Irreducible Error	12
2.2.2	Bias Error	13
2.2.3	Variance Error	13
2.3	Regularisation	14
2.3.1	L1 and L2 Regularisation	14
2.3.2	Dropout	14
2.3.3	Early Stopping	15
2.4	Optimization	16
2.4.1	Exponentially weighted Average	16
2.4.2	Gradient Descent with Momentum	17
2.4.3	RMSprop	17
2.4.4	Adam optimizer	18
2.4.5	Batch Normalization	18
2.4.6	Multiclass Classification	18
3	Implementation of Neural Network on ICE40 UltraPlus FPGA	20
3.1	Description of the hardware	20
3.1.1	Initial Setup	20
3.1.2	Modified Goal	22
3.2	Use of Convolutional Neural Network	22
3.2.1	Convolutional Layer	22
3.2.2	Pooling Layer	23
3.2.3	Fully Connected Layer	24
3.3	Modelling and Training of the neural network	24
3.3.1	Constraints in implementation	25
3.3.2	Keras Implementation	27
3.3.3	Tensorflow Implementation	27
3.4	Implementation of Neural Network into the FPGA	29
3.4.1	Loading model into SensAI	30
3.4.2	Analysis and Compilation	31
3.4.3	Downloading the bit-stream into the FPGA	33
3.4.4	Possible Solutions	33
3.5	Conclusion	34

List of Figures

1.1	Handwriting Sample	1
1.2	Gradient Descent	3
1.3	SGD Path	4
1.4	Gradient Descent Comparison	4
1.5	Example of Shallow Neural Network	5
1.6	Node in a hidden layer	5
1.7	Examples of various Activation Functions	6
1.8	Computation path for Gradient Descent in a Shallow Neural Network	8
1.9	Example of a Deep Neural Network	9
1.10	Computation path of Deep neural network	10
2.1	Split of Dataset	12
2.2	Example of underfitting in Classification	13
2.3	Example of overfitting in Classification	13
2.4	Example of one iteration of Dropout	15
2.5	Early Stopping	15
2.6	Plot smoothened using EWA	16
2.7	Affect of Momentum on SGD	17
3.1	Snippet of the code in Raspberry	21
3.2	Original Workflow	21
3.3	Example of CNN	22
3.4	Example of convolution	23
3.5	Example of Pooling	24
3.6	Fully Connected Layer Layer	24
3.7	Implementation of the neural network on keras	27
3.8	Implementation of the neural network on Tensorflow	28
3.9	Model of neural network used.	29
3.10	Project window in SensAI	30
3.11	Project Implementation Setting Window	30
3.12	Analysis and Simulation window in SensAI	31
3.13	Flow Diagram	32
3.14	Error while downloading the bit-stream	33
3.15	Persisting Error while downloading the bit-stream	33

Abbreviations

NN	N eural N etwork
GD	G radient D escent
SGD	S tochastic G radient D escent
LR	L ogistic R egression
DNN	D eep N eural N etwork
RMS	R oot M ean S quare
Adam	A daptive M omentum
BN	B atch N ormalization
CNN	C onvolutional N eural N etwork
FC	F ully C onnected
FPGA	F ield P rogrammable G ate A rray
SC	S emi C onductor
LUT	L ook U p T able
EBR	E mbded B lock R AM

Dedicated to my parents

Chapter 1

Classification using Neural Networks

Neural networks are computer algorithms modeled to detect and recognize patterns. They get their name from the fact that their design is loosely based on the working of a neuron in the human brain. A simple neural network requires 'learning' on a training set before it can be used to recognize patterns. This training set is usually a large dataset of patterns, very similar to the pattern the neural network is expected to recognize. To draw parallels to the human brain, consider the following image[1]:

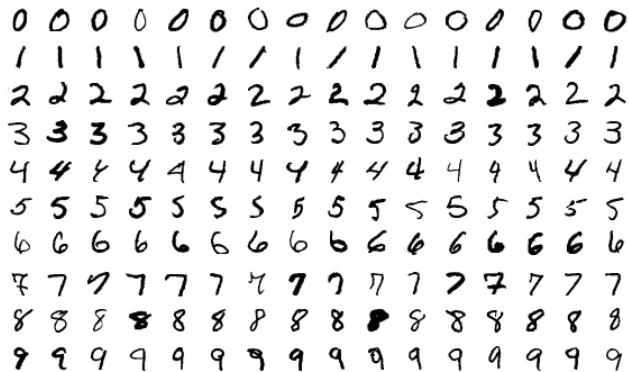


FIGURE 1.1: Handwriting Sample

If the brain is presented with this image and told all the characters in the first row come under the category of 'zero', the brain will be able to recognize a 'Zero' whenever any character that has a resemblance to the characters in the first row, is presented to it.

1.1 Logistic Regression: Binary Classification

Logistic Regression is a machine-learning classification algorithm to predict the probability of categorical dependent variables[2]. The output possible categories could be only

two. The classifier instead of drawing a straight line between the data in the data space, it finds an estimate, for all possible data points, of the conditional probability that the point belongs to the positive category.

1.1.1 Regression Model

Since the model gives a probability as the output, a normal linear model wouldn't suffice as the output can only be between 0 and 1. For this very reason, the linear model is passed in a function that has output only between 0 and 1, called the activation function. The most often used activation function in logistic regression is the Sigmoid activation function (σ), and so we have used that in our model too.

$$\hat{y} = \sigma(w^T x + b)$$

where,

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

Classification is done based on probability calculated for each input, which is assumed to be equal to the output of $\sigma(z)$. It is classified as follows:

$$\sigma(z_i) = \begin{cases} \geq 0.5, & y_i = 1 \\ \leq 0.5, & y_i = 0 \end{cases}$$

1.1.2 Loss function

Once the probabilities have been calculated, the next step is to design a loss function that would calculate how much does the model deviate from actual results, so that it can make changes in the model and improve its performance. The loss function used in the logistic regression model is the 'log loss' function[3]:

$$\text{Log loss} = \sum_{(x,y) \in D} (-y \log(\hat{y}) - (1-y) \log(1-\hat{y})),$$

where (x,y) belong from the data set, and \hat{y} is calculated from the model.

1.1.3 Gradient Descent

For the model given above, gradient descent helps in finding the right W and b matrices to minimize the following loss function:

$$J(W, b) = -\frac{1}{m} \sum_{i=1}^m (-y \log(\hat{y}) - (1-y) \log(1-\hat{y}))$$

,where m is the number of training examples

Gradient descent is a recursive optimization algorithm to find the minima of a function. To find the minima, steps equal to the negative of the gradient of that function at that point concerning the model parameters[4].

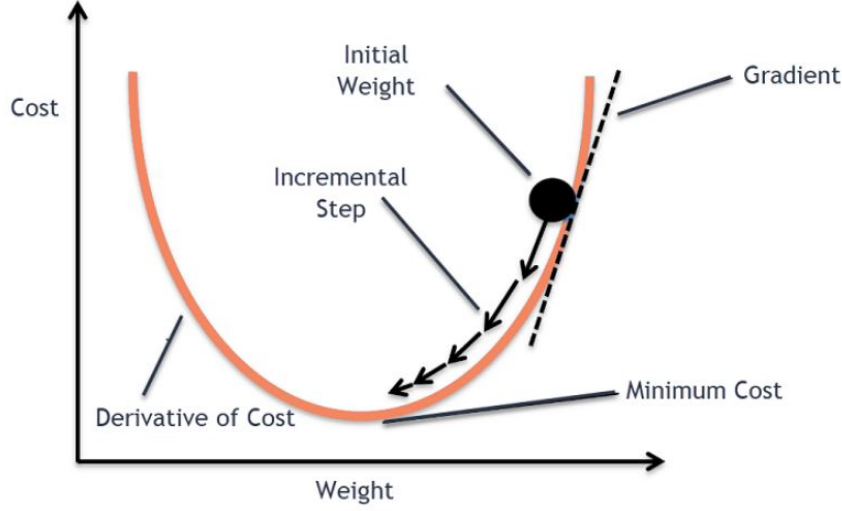


FIGURE 1.2: Gradient Descent

That is,

$$W = W - \alpha \frac{\partial J}{\partial W}$$

$$b = b - \alpha \frac{\partial J}{\partial b}$$

,where α = learning rate

It can be shown using simple calculus that for a loss function with a parameter matrix θ , the gradient at any point can be found as:

$$\frac{\partial J(\theta)}{\partial \theta_j} = -\frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^i$$

This method of Gradient Descent is called Batch Gradient Descent. If the number of training set (m) is too big, the summation of these many numbers is too computationally demanding as the algorithm requires the entire training set to be in the memory. One way to solve this problem is to update the parameters after each data input as shown below:

for i in the range of (m):

$$\theta_j = \theta_j - \alpha \cdot (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^i$$

This method of gradient descent is called Stochastic gradient descent(SGD)[5]. SGD has a less smooth path to the minima when compared to the Batch GD.

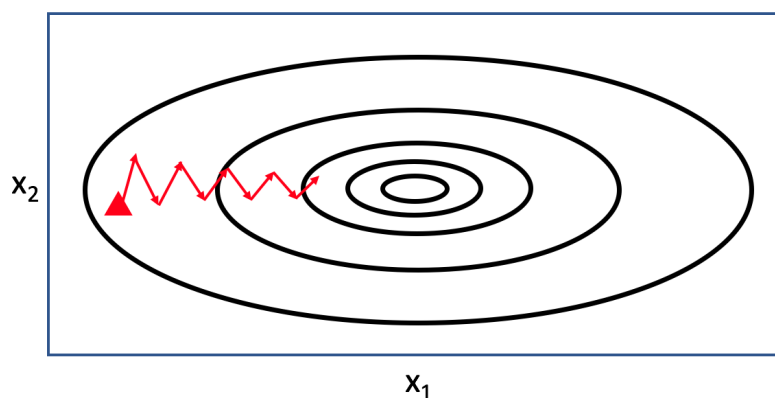


FIGURE 1.3: SGD Path

That enables it to miss some local minima owing to its noisy updates, but also poses some problems like its inability to settle on a point of minimum error close to the minima and also can get computationally slow for large datasets, as it makes gradient calculations for each input data. These problems can be solved to some extent by striking a middle ground between SGD and Batch GD, using mini-batch GD.

1.1.4 Mini-batch Gradient Descent

Mini-batch Gradient descent divides the training set into small batches that are used to find losses and update the parameters[6]. It shares the robustness of SGD and the efficiency of Batch GD. It adds the parameter 'mini-batch size' to the model.

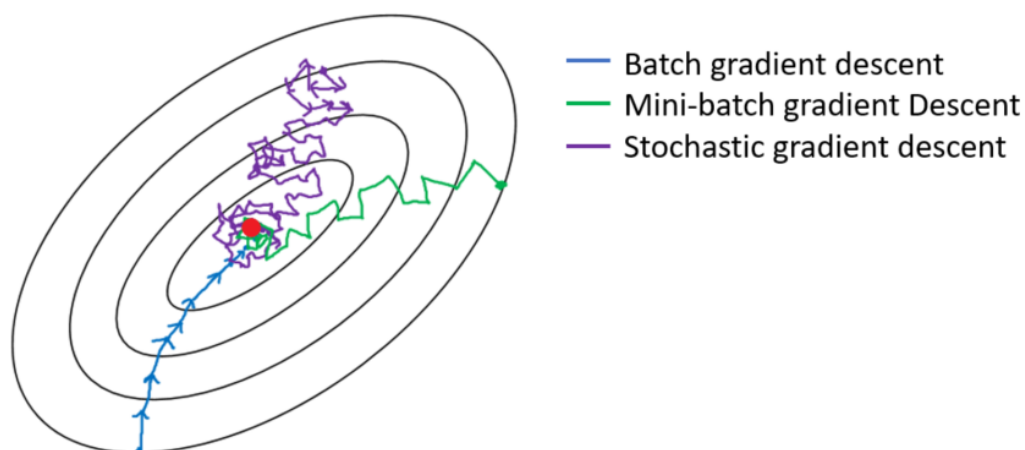


FIGURE 1.4: Gradient Descent Comparison

1.2 Shallow Neural Networks

As the name suggests, a Shallow neural Network is a neural network with one input layer, one hidden layer, and one output layer. The hidden layer is the layer where the artificial neurons take in a set of weighted inputs and produce an output through an activation function.

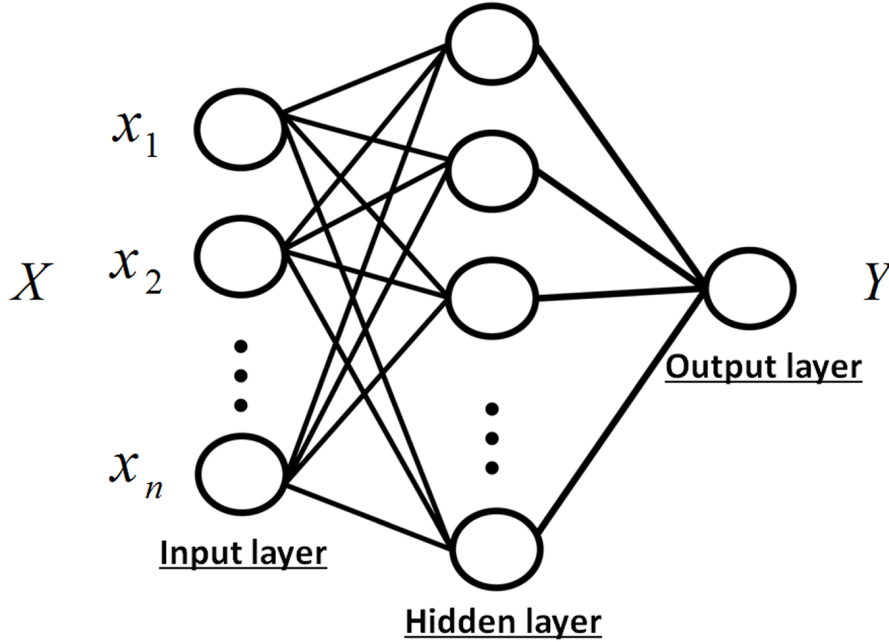
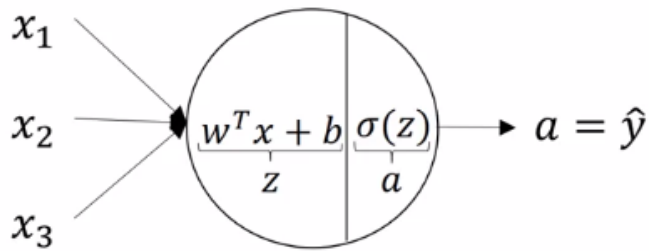


FIGURE 1.5: Example of Shallow Neural Network

A very simple example of a shallow neural network is Logistic Regression which was discussed in the previous section. A shallow NN can have more than one inputs and more than one neuron in the hidden layer.

1.2.1 Neurons

Each neuron has two halves, the head and the tail[7]:



$$z = w^T x + b$$

$$a = \sigma(z)$$

FIGURE 1.6: Node in a hidden layer

The tail of the neuron contains the weights for the model. If the input is an array of $N \times 1$, and then node has W and b that are also $N \times 1$. The head contains the activation function for that particular neuron. These activation functions all have a salient feature of having a range of $(0,1)$. There are a few functions that serve the purpose and can be used in a NN[8]. For example:

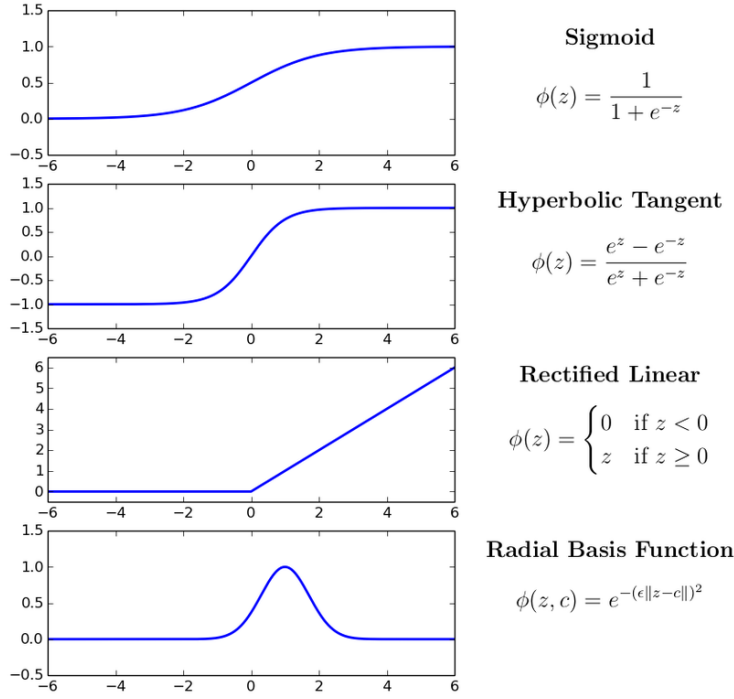


FIGURE 1.7: Examples of various Activation Functions

1.2.2 Hidden Layer

For a hidden layer with 4 neurons, the following equations calculate the output:

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, \quad a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, \quad a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, \quad a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, \quad a_4^{[1]} = \sigma(z_4^{[1]})$$

The superscript denotes the layer number and the subscript number denotes the node number in a particular layer. X is the matrix with the input set. $w_i^{[j]}$ and $b_i^{[j]}$ are the weight and bias matrices in node j that's present in layer i . $z_i^{[j]}$ is the intermediate output and $a_i^{[j]}$ is the final output from node j present in the layer i . $\text{Sigma}(\sigma)$ denotes the activation function for that node.

1.2.3 Output Layer

The output layer has only one node and the value that comes out of its activation function is considered as the output value and is used to compute the loss:

$$\hat{y} = A^{[2]} = \sigma(w^{[2]T} A^{[1]} + b^{[2]})$$

1.2.4 Forward Propagation

The forward propagation multiplies the inputs and weights of the hidden layer, passes them through their activation functions, and then passes through the output layer to calculate the loss using the loss function. The following equations summarise the path of forward propagation:

$$Z^{[1]} = W^{[1]T} X + b^{[1]}$$

$$A^{[1]} = \sigma(Z^{[1]})$$

$$Z^{[2]} = W^{[2]T} A^{[1]} + b^{[2]}$$

$$\hat{y} = A^{[2]} = \sigma(Z^{[2]})$$

1.2.5 Weight Initialization

Suppose the weight matrices for all neurons in a hidden layer are initialized with the same value. That would mean that in the first iteration for all inputs, each neuron would give the same output and eventually the correctional value to minimize the loss be the same for each neuron in that layer. Therefore for the next iteration also, each neuron will have the same weight. As it can be observed, there will be no significance of having more than one neuron in a particular layer then. Therefore to have each neuron work as a unique function, the weights are initialized randomly[9]. The values should be small and close to zero as the slope of activation functions at these values as large. This leads to faster learning while using gradient descent. The bias matrix can be initialized as all zeroes.

1.2.6 Backward Propagation

Much like logistic regression, for accurate predictions the loss function should be minimized by making updates to the weight. This is done through the method of Gradient Descent. The procedure is a little more complicated than LR.

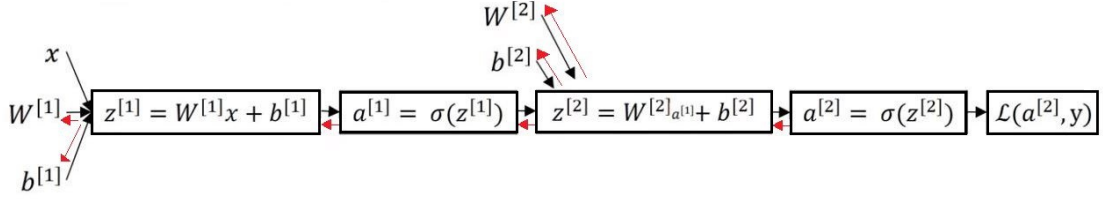


FIGURE 1.8: Computation path for Gradient Descent in a Shallow Neural Network

In the above figure, the forward propagation is shown with black arrows. The red lines are used to denote the path of backward propagation, which updates the weights and biases to reach a minimum value of the loss function. Using the loss function discussed in the previous section and taking sigmoid as the activation functions for all neurons, the gradient descent can be easily calculated using simple calculus as follows[10]:

$$\begin{aligned}
 dA^{[2]} &= \frac{\partial L(A^{[2]}, Y)}{\partial A^{[2]}} = \frac{-Y}{A^{[2]}} + \frac{1-Y}{1-A^{[2]}} \\
 dZ^{[2]} &= \frac{\partial L(A^{[2]}, Y)}{\partial Z^{[2]}} = \frac{\partial L(A^{[2]}, Y)}{\partial A^{[2]}} * \frac{\partial A^{[2]}}{\partial Z^{[2]}} = A^{[2]} - Y \\
 dW^{[2]} &= \frac{\partial L(A^{[2]}, Y)}{\partial W^{[2]}} = \frac{\partial L(A^{[2]}, Y)}{\partial Z^{[2]}} * \frac{\partial Z^{[2]}}{\partial W^{[2]}} = \frac{1}{m} dZ^{[2]} A^{[2]T} \\
 db^{[2]} &= \frac{\partial L(A^{[2]}, Y)}{\partial b^{[2]}} = \frac{\partial L(A^{[2]}, Y)}{\partial Z^{[2]}} * \frac{\partial Z^{[2]}}{\partial b^{[2]}} = \frac{1}{m} dZ^{[2]} \\
 dA^{[1]} &= \frac{\partial L(A^{[2]}, Y)}{\partial A^{[1]}} = \frac{\partial L(A^{[2]}, Y)}{\partial Z^{[2]}} * \frac{\partial Z^{[2]}}{\partial A^{[1]}} = dZ^{[2]} W^{[2]T} \\
 dZ^{[1]} &= \frac{\partial L(A^{[2]}, Y)}{\partial Z^{[1]}} = \frac{\partial L(A^{[2]}, Y)}{\partial A^{[1]}} * \frac{\partial A^{[1]}}{\partial Z^{[1]}} = dZ^{[2]} W^{[2]T} * \sigma'(Z^{[2]}) \\
 dW^{[1]} &= \frac{\partial L(A^{[2]}, Y)}{\partial W^{[1]}} = \frac{\partial L(A^{[2]}, Y)}{\partial Z^{[1]}} * \frac{\partial Z^{[1]}}{\partial W^{[1]}} = \frac{1}{m} dZ^{[1]} X^T \\
 db^{[1]} &= \frac{\partial L(A^{[2]}, Y)}{\partial b^{[1]}} = \frac{\partial L(A^{[2]}, Y)}{\partial Z^{[1]}} * \frac{\partial Z^{[1]}}{\partial b^{[1]}} = \frac{1}{m} dZ^{[1]}
 \end{aligned}$$

For all $W^{[i]}$ and $b^{[i]}$, the new values are :

$$W^{[i]} = W^{[i]} - \alpha * dW^{[i]}$$

$$b^{[i]} = b^{[i]} - \alpha * db^{[i]}$$

1.3 Deep Neural Networks

A Deep Neural Network(DNN) differs from a shallow NN as it has more than one hidden layer. A DNN is capable of drawing out more complex patterns and relations between input and output, at the cost of larger memory requirement and higher computation time for training the model.

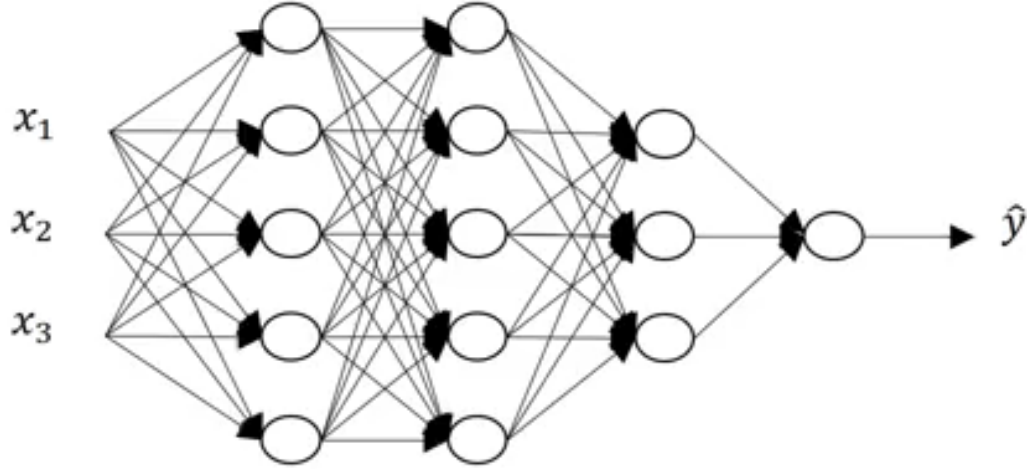


FIGURE 1.9: Example of a Deep Neural Network

Each layer i has $n^{[i]}$ nodes. The superscript for all parameters denotes the layer it is present in. The size of weights depend on the number nodes in the layer it is present and the layer before that. For example, for any layer l :

$$w^{[l]} : (n^{[l]}, n^{[l-1]})$$

$$b^{[l]} : (n^{[l]}, 1)$$

$$A^{[l]}, Z^{[l]} : (n^{[l]}, 1)$$

1.3.1 Forward Propagation

The concepts used in the previous sections can be extrapolated and used here to explain the forward propagation for a DNN. For any hidden layer, the output of the activation function from the previous layer acts as the input and is used to calculate its own weighted output. For the first layer, it uses the input to model as its own inputs. The output from the activation function of the output layer is used to calculate the loss function. The loss function is the same as used Shallow NN. This concept can be simplified as follows for a n - layered model:

$$A^{[0]} = X$$

$$Z^{[l]} = W^{[l]T} A^{[l-1]} + b^{[l]}$$

$$A^{[l]} = \sigma(Z^{[l]})$$

$$A^{[n]} = \hat{y}$$

$$\text{Log loss} = \sum_{(x,y) \in D} (-y \log(\hat{y}) - (1-y) \log(1-\hat{y}))$$

1.3.2 Backward Propagation

The back propagation concepts are very similar to those discussed for Shallow NN. For the calculations for gradient descent, a few values from the forward propagation are also necessary and therefore during forward propagation, these values are stored as cache at each layer. The following equations explain the methodology[10]:

$$dA^{[l-1]} = \frac{\partial L(\hat{Y}, Y)}{\partial A^{[l-1]}} = \frac{\partial L(\hat{Y}, Y)}{\partial Z^{[l]}} * \frac{\partial Z^{[l]}}{\partial A^{[l-1]}} = dZ^{[l]} W^{[l]T}$$

$$dZ^{[l]} = \frac{\partial L(\hat{Y}, Y)}{\partial Z^{[l]}} = \frac{\partial L(\hat{Y}, Y)}{\partial A^{[l]}} * \frac{\partial A^{[l]}}{\partial Z^{[l]}} = dA^{[l]} W^{[l]T} * \sigma'(Z^{[l]})$$

$$dW^{[l]} = \frac{\partial L(\hat{Y}, Y)}{\partial W^{[l]}} = \frac{\partial L(\hat{Y}, Y)}{\partial Z^{[l]}} * \frac{\partial Z^{[l]}}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$$

$$db^{[l]} = \frac{\partial L(\hat{Y}, Y)}{\partial b^{[l]}} = \frac{\partial L(\hat{Y}, Y)}{\partial Z^{[l]}} * \frac{\partial Z^{[l]}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^m dZ^{[l][i]}$$

The entire pathway of training the model can be summarised by this diagram:

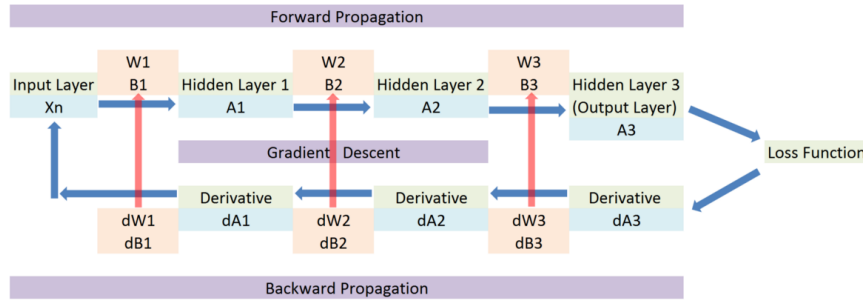


FIGURE 1.10: Computation path of Deep neural network

1.3.3 Prediction

Once the model has been trained well, such that it has very small losses, the model can be used for classification. An input can be provided to the model and the output can be interpreted as shown below, to classify the data:

$$\hat{y} = \begin{cases} \geq 0.5, & y_i = 1 \\ \leq 0.5, & y_i = 0 \end{cases}$$

Chapter 2

Hyper-parameters , Regularization and Optimization

A neural network has various variables that determine the performance of the model. These variables are called Hyper-Parameters and these are usually fixed for a model before the training starts. Some of these variables like the number of layers and the learning rate have already been discussed in the previous sections. This chapter discusses various other techniques used to improve the performance of the neural network and the other hyper-parameters introduced.

2.1 Data set

To improve training, it is imperative to divide the available dataset into right sets. The dataset is usually split into three parts: Training set, Development/Validation(Dev) set, and Test set:

The **training set** is what the model uses to 'learn'. Depending on the aimed accuracy and availability of data, the size of the training set is determined. The training set should be representative of all possible cases as the model will 'learn' only what it will 'see' and therefore should cover all classes equally. For a Binary Classifier, it should have almost equal instances of that class being present and absent.

The **Development/Validation set** is used while training, to tune all the hyperparameters. The model doesn't train using this data set. It is used only to measure the accuracy, and the parameters are changed accordingly. This set doesn't have to be very big or very carefully chosen.

The **test set**, as the name suggests, is used finally test the model when the training is finally done. The test set should be carefully chosen as it should be representative of the entire data set and is big enough to yield statistically useful data. It should

cover all possible cases the model would ever face in the real world to gauge its actual performance.

2.1.1 Dataset Split

The entire size of the data set and the split of the data set is dependent on the availability of data and aimed accuracy of the model. For models aimed at very high accuracy or very complex models, require a large training set. The size of the validation depends on the number of hyper-parameters to be tuned. If the parameters are not too many or the model is already performing well, there is no need for a dev set. The test set is usually aimed to be big enough to cover all cases and be able to be diverse enough to mirror the training set and the real world. For a data set with millions of data, the validation and testing set doesn't have to be a big proportion of the total set. Follow could be a realistic split of data:

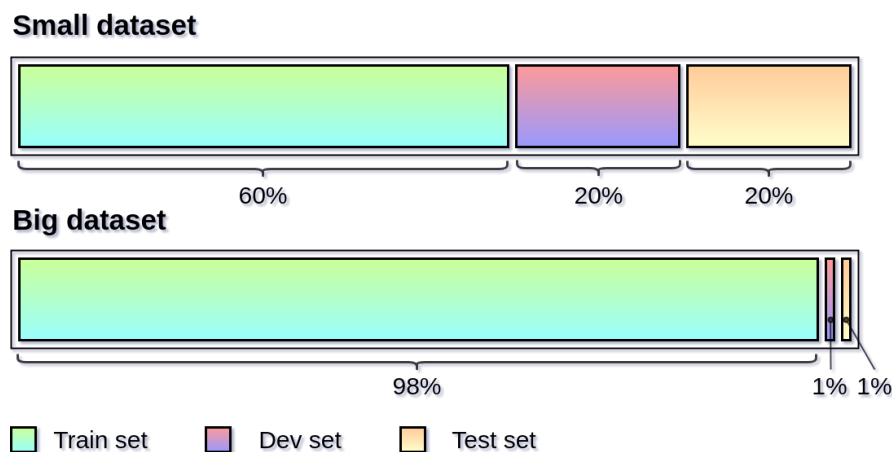


FIGURE 2.1: Split of Dataset

2.2 Prediction Errors

The main aim of a neural network is to make an intelligent guess of a mapping F , from X to Y such that for any given \hat{X} , $F(\hat{X})$ would predict an accurate value \hat{Y} . This process has scope for a lot of errors and the sources of these errors will be discussed in this section:

2.2.1 Irreducible Error

These errors can't be reduced for a model selected by tuning the model. This problem arises when the function mapping is a factor of another variable, that's not considered in the model. It could also be limited by the limitation of the model used.

2.2.2 Bias Error

Bias errors arise from erroneous assumptions made to simplify the model and make the mapping easier to learn. This results in faster and easier training but fails to incorporate the complex functions and miss out on some mappings from the input to output, thus reducing its predictive capability[11]. This results in underfitting. The concept of underfitting can be better understood from the following diagram:

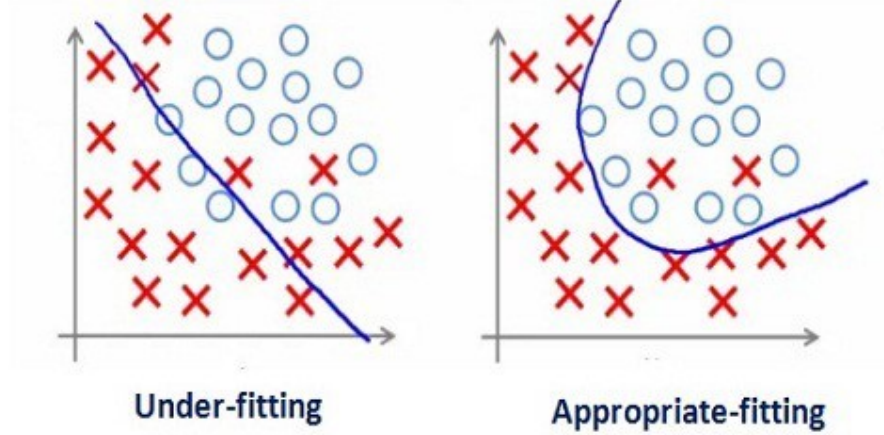


FIGURE 2.2: Example of underfitting in Classification

2.2.3 Variance Error

Variance errors arise when the model is very susceptible to change in the data set. The model will make mappings based on the outliers and random noise in the input[11]. This results in a less robust model and also results in overfitting. With overfitting, the accuracy of the training dataset would increase but with the test set, the accuracy would go down. The following diagram explains the concept of overfitting:

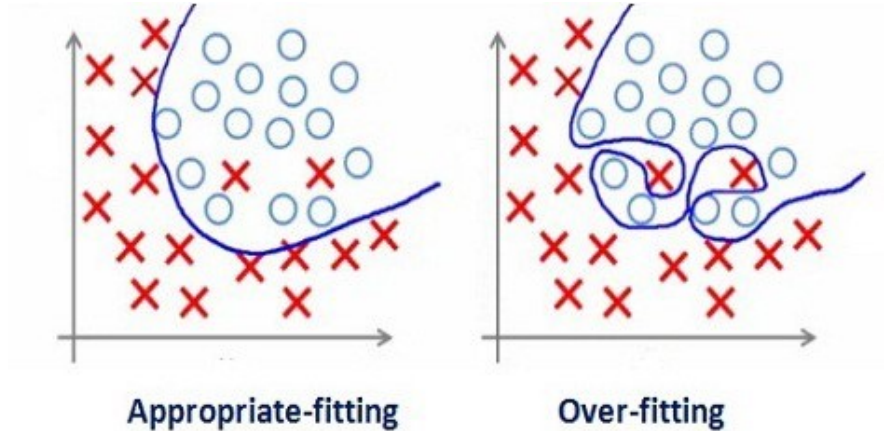


FIGURE 2.3: Example of overfitting in Classification

2.3 Regularisation

For a model to give a desirable result, the model should find the right fit. Underfitting can be avoided by making a complex model and making the training set bigger. The real problem arises in controlling over-fitting as it is not feasible to remove all outliers and noise from the data set. For helping generalize the mapping better, Regularisation is used. These are techniques used while training the model to avoid overfitting.

It has been noticed that overfitting can almost always be associated with large coefficients in forward propagation, as small changes in the input result in large changes to the loss functions and the model tries to accommodate that change more drastically. Most regularisation techniques revolve around penalizing large weights, thus driving down their values. Some of the techniques are discussed in the following section:

2.3.1 L1 and L2 Regularisation

Both these techniques aim at solving the problem of overfitting by penalizing the weight matrix and driving down the value of the coefficients. This is implemented by adding a regularisation term which is a function of the weights of the model, to the loss function. L1 and L2 differ in the regularisation term added.

L2 Regularisation:

In L2 regularisation we have[\[12\]](#),

$$Cost\ Function = Loss + \frac{\lambda}{2m} \sum \|W\|^2$$

Here, λ is called the Regularisation parameter and is a hyperparameter used to control the regularisation. L2 regularisation drives the values of weights towards zero but doesn't make them zero. This is the more regularisation used most often.

L1 Regularisation:

In L1 regularisation we have[\[13\]](#),

$$Cost\ Function = Loss + \frac{\lambda}{2m} \sum \|W\|$$

Unlike L2 regularisation, L1 regularisation has a penalty on the absolute value of W . This can drive the value of weights to zero, thus is also beneficial in compressing the size of the model.

2.3.2 Dropout

This regularisation method reduces overfitting by making the model more robust[\[14\]](#). For each iteration of training, in every layer, some nodes are randomly removed. The

probability that a node will be removed or dropped out is another hyper-parameter to the model.

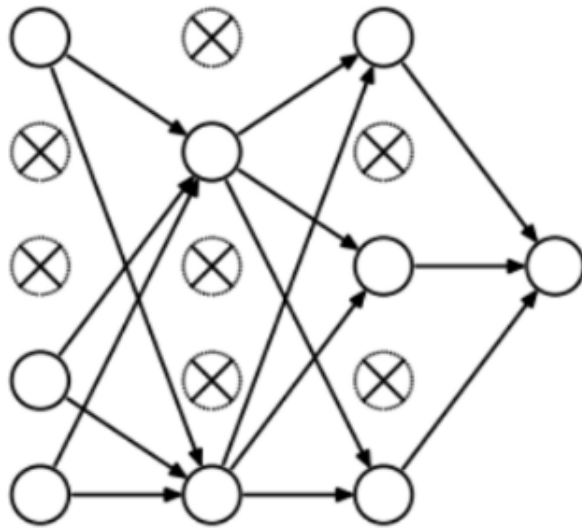


FIGURE 2.4: Example of one iteration of Dropout

2.3.3 Early Stopping

In this technique, while training, the performance of the model on validation is monitored. The training is stopped when the accuracy on validation starts to decrease due to overfitting.

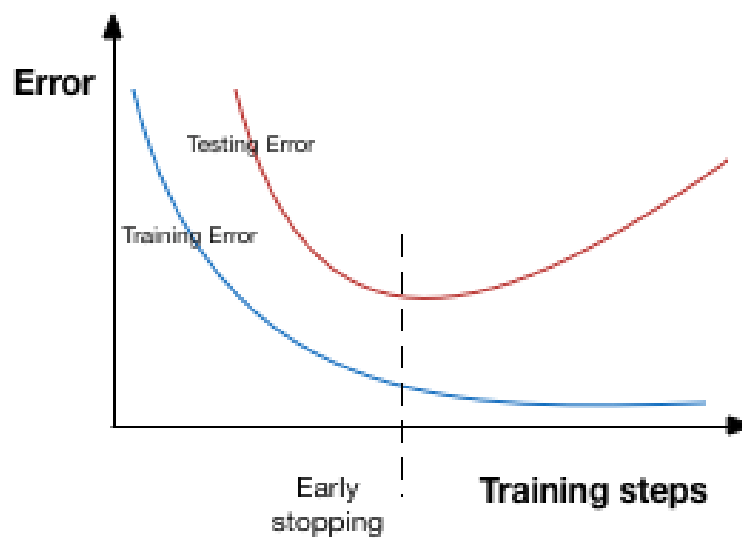


FIGURE 2.5: Early Stopping

2.4 Optimization

Optimization refers to finding the minima for the loss function of the model. This is usually done using the method of gradient descent as discussed in the previous chapter. Gradient descent has a lot of variations that make the optimization faster and more computationally efficient. In the previous sections, we have already discussed Stochastic, Batch and Mini-Batch Gradient Descent. In this section, we will discuss some other optimization techniques.

2.4.1 Exponentially weighted Average

This is not a type of optimizer but it is used in other optimizing techniques. It uses the concept of moving average to average out the noise in the data. For a sequence of data θ_n , EWA will produce a sequence S_n as follows:

$$S_n = \beta S_{n-1} + (1 - \beta)\theta_n$$

This introduces a new hyper-parameter β to the model. The value of β can be approximated to the number of points the average is done on, as:

$$\text{Number of points averaged over} \simeq \frac{1}{1 - \beta}$$

The following plot shows the effect of EWA on scattered data:

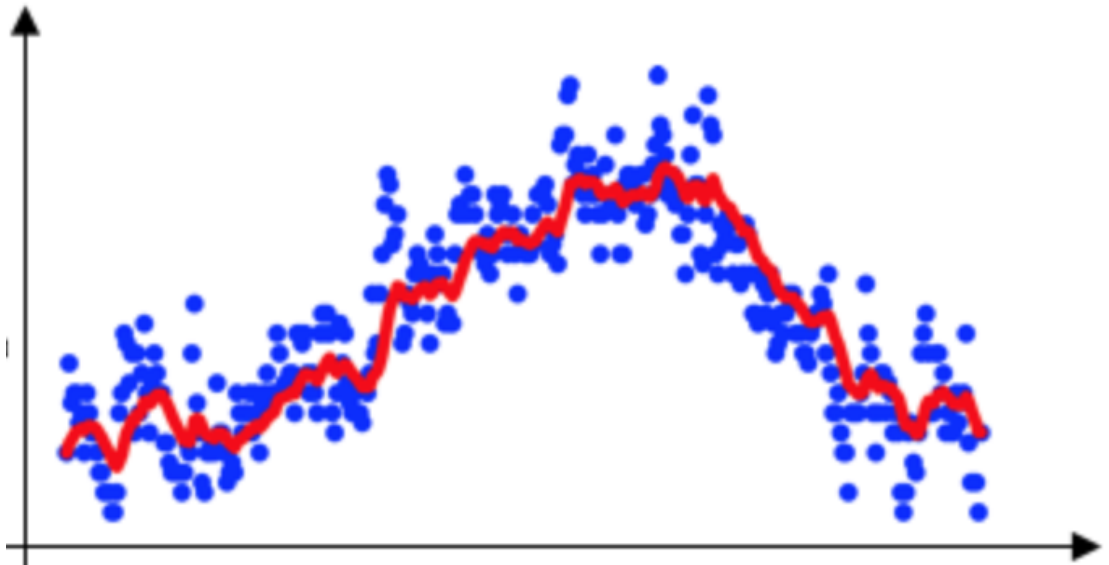


FIGURE 2.6: Plot smoothened using EWA

2.4.2 Gradient Descent with Momentum

Unlike the SGD, this optimizing technique uses a moving average of the gradient to update the weights instead of the gradient from that particular step. This solves the problem of SGD, and ensures that the derivative is always in the right direction. It can be implemented in the following way:

$$V_t^{[W]} = \beta V_{t-1}^{[W]} + (1 - \beta) \nabla_W L(W, b, X, Y)$$

$$V_t^{[b]} = \beta V_{t-1}^{[b]} + (1 - \beta) \nabla_b L(W, b, X, Y)$$

$$W = W - \alpha V_t^{[W]}$$

$$b = b - \alpha V_t^{[b]}$$

The following diagram exhibits how momentum moves drive the gradient descent in the right direction.

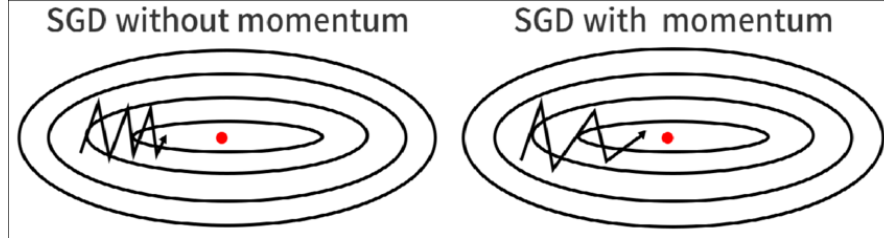


FIGURE 2.7: Affect of Momentum on SGD

2.4.3 RMSprop

Much like Gradient Descent with Momentum, Gradient Descent with RMS prop reduces oscillations and thus allows the model to use larger learning rates and converge to the minima faster[15]. The difference in the two models is how the gradient descent is applied.

$$V_t^{[W]} = \beta V_{t-1}^{[W]} + (1 - \beta) (\nabla_W L(W, b, X, Y))^2$$

$$V_t^{[b]} = \beta V_{t-1}^{[b]} + (1 - \beta) (\nabla_b L(W, b, X, Y))^2$$

$$W = W - \alpha \frac{\nabla_W L(W, b, X, Y)}{\sqrt{V_t^{[W]} + \epsilon}}$$

$$b = b - \alpha \frac{\nabla_b L(W, b, X, Y)}{\sqrt{V_t^{[b]} + \epsilon}}$$

Sometimes the value of V_t could approach, blowing up the correctional term. To avoid this, ϵ is added to the denominator of that term and set to a very small value.

2.4.4 Adam optimizer

Adaptive moment optimizer or Adam Optimizer uses the concepts from both GD with Momentum and GD with RMSprop. It uses the moving average of both, the gradients and their squares, to find the new values of weights[16].

$$\begin{aligned}
m_t^{[W]} &= \beta_1 m_{t-1}^{[W]} + (1 - \beta_1) \nabla_W L(W, b, X, Y) \\
m_t^{[b]} &= \beta_1 m_{t-1}^{[b]} + (1 - \beta_1) \nabla_b L(W, b, X, Y) \\
V_t^{[W]} &= \beta_2 V_{t-1}^{[W]} + (1 - \beta_2) (\nabla_W L(W, b, X, Y))^2 \\
V_t^{[b]} &= \beta_2 V_{t-1}^{[b]} + (1 - \beta_2) (\nabla_b L(W, b, X, Y))^2 \\
W &= W - \alpha \frac{m_t^{[W]}}{\sqrt{V_t^{[W]} + \epsilon}} \\
b &= b - \alpha \frac{m_t^{[b]}}{\sqrt{V_t^{[b]} + \epsilon}}
\end{aligned}$$

This introduces two hyperparameters β_1 and β_2 . Usually, they are set to 0.9 and 0.999 respectively.

2.4.5 Batch Normalization

While back-propagation, each layer individually tries to rectify the error in prediction and changes the weights based on the input it gets from the previous layer. The problem arises when the output from the previous layer has a huge variance and changes drastically with the updated value of weights in the previous layer. This phenomenon is called Internal Covariate Shift. This causes the present layer to wait for the previous layer to update its weights to the new updated value before it can find its new weight. This makes the process of learning very slow. This problem is solved by normalizing the input mini-batch to have a mean equal to 0 and a variance equal to 1 so that the change of input to a layer is not affected drastically by changing the weights of the previous layer. This way all layers can compute the new weight through back-propagation simultaneously for a mini-batch. The mini-batch for any layer k is normalized as follows:

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

2.4.6 Multiclass Classification

The neural network can be used to classify more than two classes. This can be done using making some changes to the output layer and loss function. The output layer should have the same number of nodes as the classes and each node has a SoftMax

activation function instead of relu function. For an n-class classifier, the output node gives 'n' outputs. The following equation describes the softmax function:

$$y_i(z_i) = \frac{e^{z_i}}{\sum_{k=1}^n e^{z_k}}$$

The cost function is also changed to a loss function called Cross Entropy function.

$$L(\hat{Y}, Y) = - \sum_{i=1}^n Y_i \log(\hat{Y}_i)$$

For prediction, the class with the highest output value is considered as the detected class.

Chapter 3

Implementation of Neural Network on ICE40 UltraPlus FPGA

The basic aim of this project is to implement a CNN to detect human faces on the ICE40 FPGA board provided by Lattice semiconductors. The software used to implement neural networks is SensAI, also a software provided by Lattice SC. iCE40 UltraPlus FPGA with 5K lookup tables (LUTs) is more than capable to implement Neural Networks with great efficiency in speed and power consumption. The neural network is implemented and trained on the TensorFlow 1.12/Keras framework. The entire model is saved as (.pb/.h5) for TensorFlow, a Tensorflow inference frozen model file which contains both graph and parameter values, to be fed to the SensAI. For Keras, the model is saved as a (.h5) file. This is converted to a bitstream (.bin) file which can then loaded into the FPGA board.

3.1 Description of the hardware

3.1.1 Initial Setup

The initial plans of the project were a little different from the final implementation. The project involved data retrieval from a gas sensor using a Raspberry Pi 3 board which would be passed through an ADC and then be supplied to the FPGA board with a neural network model downloaded into it which would have been trained to classify the gas from the data provided by the sensor. The sensor setup has an array of sensors, and all the values are passed through a Multiplexer (Mux). The select lines are controlled by the Raspberry Pi board and select which sensor's data it wants depending on how the Raspberry board is programmed. A python code is written to control all these

operations in the Raspberry board. The following is a snippet of the code that runs a loop on the different sensors from which the data is to be extracted:

```
arg=2
if(len(sys.argv)>1):
    arg=int(sys.argv[1])
for nb_total in range(0,arg):

    for nb_loop in range(0,100):

        t_mes.append(time.time()-t0)
        freq=0
        v.standby()
        v.start_sweep()

        while not(v.flag_sweep==1):

            for col in range(1,2): #selects the column of sensor array

                for lig in range(0,1):#selects the row of sensor array
                    mux.set(c.rfb[lig],col,lig)

                    for nb in range(0,Nb_repeat):
                        v.repeat_measure()
                        re[nb],im[nb]=v.read_measure()

                    |
                    reMed = np.median(re)
                    imMed = np.median(im)
                    if(freq==0):
                        c.push_sens(reMed,imMed,freq,col,lig,nb_loop)
                        val_tmp=c.capteur[col,lig,2*freq,nb_loop]
                        if(nb_loop==0 and nb_total==0):
                            val_init=c.capteur[col,lig,2*freq,nb_loop]
                        if(val_tmp>1.2*val_init):
                            print('Infinite Loop !!!!')

                    freq+=1
                print(nb_loop)
                v.inc_sweep()
```

FIGURE 3.1: Snippet of the code in Raspberry

The Raspberry Pi board uses the GPIO pins for all input/output operations. The data from the ADC is transferred to the bus using an I2C bus integrated into the GPIO pins. Once the data has been acquired, it is stored in an excel sheet. This data is to be used to extract important features from the data and noise corrected before it is sent to the FPGA board for classification. The following diagram explains the workflow better:

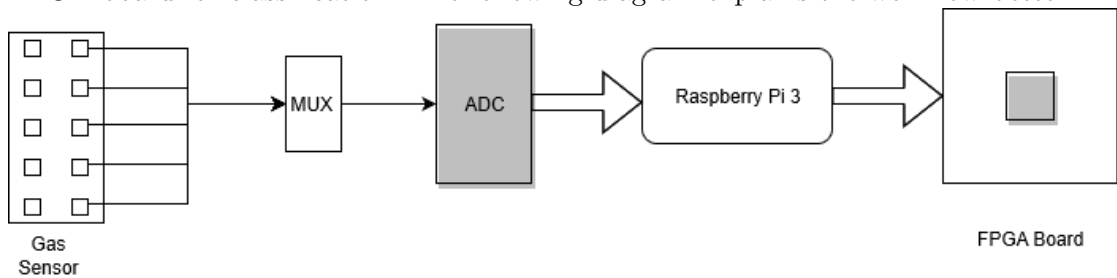


FIGURE 3.2: Original Workflow

The eventual goal of the project was to integrate all these components into one embedded board, eliminating the Raspberry board. However, this initial plan had to be modified a little bit due to the fact that there wasn't enough data from the sensor for different gases with different concentrations to train the classifier.

3.1.2 Modified Goal

As the previously proposed model couldn't be developed due to a lack of data, it was decided to model the classifier to classify images. More specifically, images of handwritten digits from 0 to 9. The MNIST data set was used to train the model. The goal this project set to achieve was to develop and train a model that would classify the data set with high accuracy. The second half of this project was to then implement this neural network in a local FPGA and understand the complication behind the implementation.

3.2 Use of Convolutional Neural Network

As the new implementation involved images, it was best to use a Convolutional Neural Network for classification as it increases the accuracy of classification while reducing the computational complexity. Convolutional Neural network (CNN) is a deep learning algorithm to extract features out of the input image and make the process of classification a lot more efficient. A usual 32x32 RGB image has 3072 inputs. Turning them into a vector and feeding it to the neural network is not a feasible option. CNN helps extract useful spatial features that can be used to classify the image. A CNN usually involves a few filters that convolute over the image matrix and extract these features. A general CNN looks something like this:

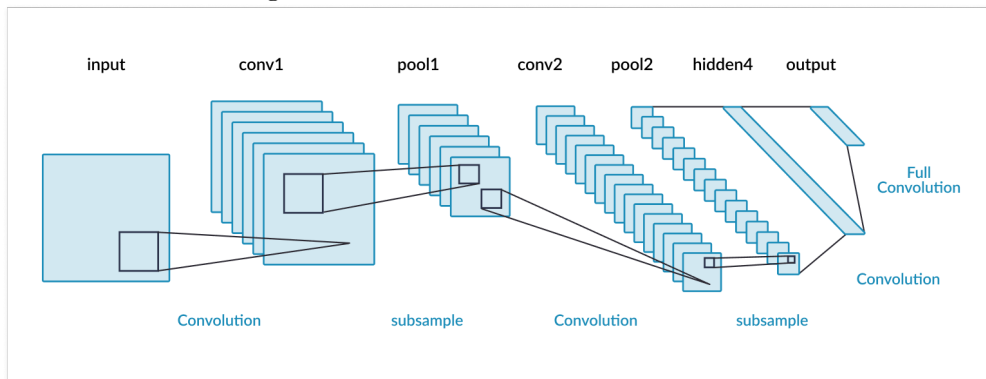


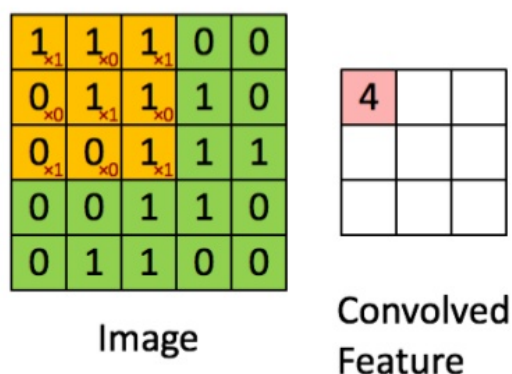
FIGURE 3.3: Example of CNN

3.2.1 Convolutional Layer

The convolutional layer consists of filters that extract spatial features from the image. These filters are matrices with values depending on their function. These filters have the same number of channels as the input data. In one layer, there could be multiple

filters and the output of these filters are stacked one over the other. The initial filters extract very basic features like vertical and horizontal edges, but as we go deeper into the network, these filters can extract more complex features like faces or objects[17]. The values of these matrices are updated during back-propagation and changed to minimize the loss function, much like in DNN. Convolution of filters with the image involves superimposing the filter matrix over the input matrix and multiplying all the overlapping numbers and taking their sum as the value of the first index of the output matrix. The filter matrix is then slid across the input matrix and the same procedure is repeated to form the entire output matrix. This leads to a smaller matrix with just one channel. To keep the dimensions of the output matrix the same as the input matrix, the input matrix should be padded with values. The following image elucidates the process further:

Convolution



http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

FIGURE 3.4: Example of convolution

3.2.2 Pooling Layer

The pooling layer helps in reducing the size of the convolved matrix without losing out on important features. This increases computational efficiency. It involves sliding a window over the matrix to form sub-matrices and taking one value from it that best captures the characteristic of that sub-matrix. There are two kinds of pooling: Max pooling and Average pooling. As the name suggests, Max pooling takes the maximum value of the sub-matrix as the value corresponding to that sub-matrix in the output matrix. The Average pooling takes the average of all elements instead of taking the maximum value. The following image will explain this concept better:

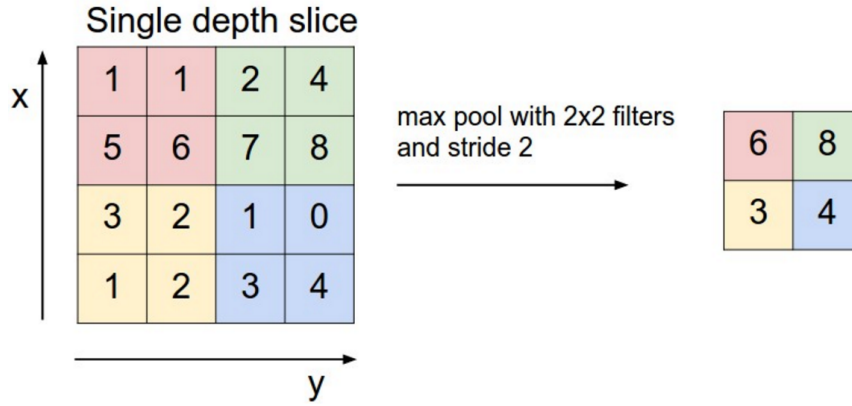


FIGURE 3.5: Example of Pooling

3.2.3 Fully Connected Layer

Once the features have been extracted and the output matrix is made small enough to make efficient computations, the output matrix is laid as a vertical matrix called the Fully Connected layer and this layer is then fed into a Neural Net as the input layer and this neural network could be like any neural network discussed in the previous sections depending on computational power at hand and expected accuracy. This layer can be noticed in this example of a CNN:

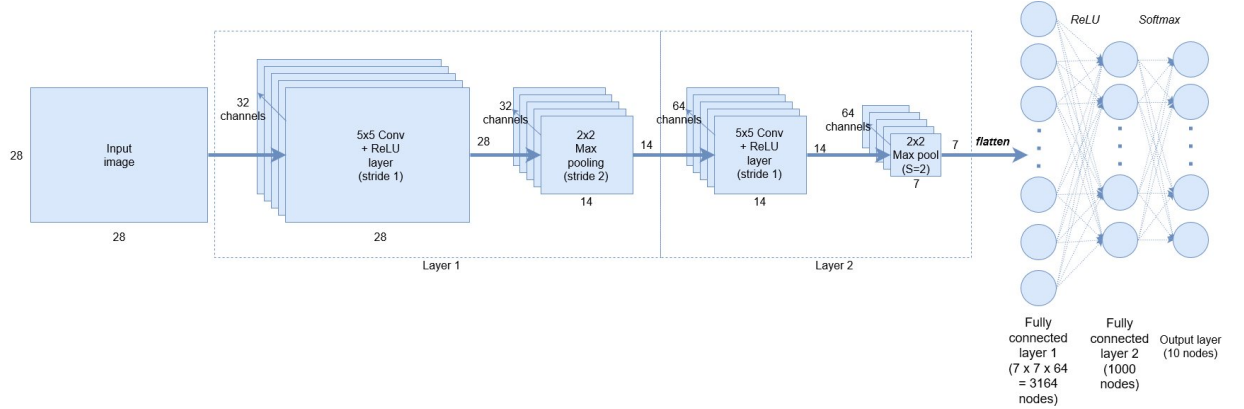


FIGURE 3.6: Fully Connected Layer Layer

3.3 Modelling and Training of the neural network

The framework used to implement the neural network is TensorFlow and keras. The model is coded in python. The model was trained for both frameworks to analyze the implementation complexity of the output file from both frameworks into the embedded system. The following TensorFlow and keras layers are supported by the SensAI:

- Convolution
- Matrix Multiplication

- Placeholder
- Batch Normalization
- Pooling
- Relu
- Element-wise add
- Max-pooling

3.3.1 Constraints in implementation

Keeping in mind that the model later has to be implemented into an FPGA, some constraints are laid down by the software that helps convert the model into a bitstream that can be downloaded into an FPGA:

- Data pre or post-processing will be ignored
- Only one placeholder allowed with its size explicitly declared
- Training to inference optimization conversion must be done for any training model you wish to use with SensAI
- Data post-processing operations (such as softmax, sigmoid, etc.) are not supported. Supported output layers are Conv2D, Matmul (for Inner Product and Full Connect) and Global Average.
- Maximum convolutional kernel size can be 9x9.

The model used for the following project is as follows :

$$\begin{aligned}
 & CONV(3 \times 3, 512) \rightarrow BN \rightarrow MaxPooling((2, 2), stride = 2) \rightarrow \\
 & CONV(3 \times 3, 64) \rightarrow BN \rightarrow MaxPooling((2, 2), stride = 2) \rightarrow \\
 & CONV(3 \times 3, 64) \rightarrow BN \rightarrow MaxPooling((2, 2), stride = 2) \rightarrow \\
 & Flatten \rightarrow output - layer
 \end{aligned}$$

Number of Convolutional layer	Optimizer	Learning Rate	Accuracy
1	ADAM	0.0001	97.1%
2	ADAM	0.0001	97.5%
3	SGD	0.0001	95.89%
3	ADAM	0.001	91.97%
3	ADAM	0.0001	98.1%

The following model was chosen after training the data set with a lot of different variations of the model as it gave the best result as demonstrated in the table above.

The model is trained on the MNIST data set, which is a repository of handwritten digits for 0 to 9. 60000 images are used to train the model and 1000 to test the model with no validation set. The model is trained to recognize the digit shown in a 28x28 jpg file. The optimizer used for the code is ADAM optimizer with $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

3.3.2 Keras Implementation

Following is the snippet of the code used for training the model using the keras framework:

```
import keras as keras
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten
from keras.layers import Dense
from keras.models import model_from_json
from keras import optimizers
import numpy
import os
import h5py

path = "utf-8'mnist.npz"

mnist = keras.datasets.mnist
(training_images, training_labels), (test_images, test_labels) = mnist.load_data(path=path)

training_images=training_images.reshape(60000, 28, 28, 1)
test_images = test_images.reshape(10000, 28, 28, 1)
training_images, test_images = training_images / 255.0, test_images / 255.0

model = Sequential()
model.add(Conv2D(512, (3, 3), input_shape=(28, 28, 1), padding='same',))
keras.layers.BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001,
center=True, scale=True, beta_initializer='zeros', gamma_initializer='ones',
moving_mean_initializer='zeros', moving_variance_initializer='ones')
model.add(MaxPooling2D(2,2))
model.add(Conv2D(64, (3, 3), input_shape=(28, 28, 1), padding='same',))
keras.layers.BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001,
center=True, scale=True, beta_initializer='zeros', gamma_initializer='ones',
moving_mean_initializer='zeros', moving_variance_initializer='ones')
model.add(MaxPooling2D(2,2))
model.add(Conv2D(64, (3, 3), input_shape=(28, 28, 1), padding='same',))
keras.layers.BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001,
center=True, scale=True, beta_initializer='zeros', gamma_initializer='ones',
moving_mean_initializer='zeros', moving_variance_initializer='ones')
model.add(MaxPooling2D(2,2))
model.add(Flatten())
model.add(Dense(10, activation='softmax'))

adam = optimizers.Adam(learning_rate = 0.0001)
model.compile(loss='sparse_categorical_crossentropy', optimizer=adam, metrics=['accuracy'])

model.fit(training_images, training_labels, epochs=1, batch_size = 2, verbose=0)

scores = model.evaluate(training_images, training_labels, verbose=0)
print("s: %.2f%%" % (model.metrics_names[1], scores[1]*100))

model_json = model.to_json()
with open("model1.json", "w") as json_file:
    json_file.write(model_json)

model.save("model1.h5")
print("Saved model to disk")
```

FIGURE 3.7: Implementation of the neural network on keras

The following model was trained for 1 epoch with a batch size of 2 and a learning rate of 0.0001 to reach the accuracy of **98.1%** The model and the weights are saved as a (.h5) file. This file saves the trained model along with all the weights.

3.3.3 Tensorflow Implementation

The Tensorflow implementation has two major functions: one function that defines the structure of the NN used, defining the layers. The other function calls the first function and trains the images on the defined NN using an optimizer defined in this function

itself. The following code snippet shows the function used to execute the training. The function has the following inputs:

- Features: Input to the training models
- Labels: labels for each input
- Mode: can be set to TRAINING or TESTING
- Param: It is a dictionary of parameters for hyper-parameters for optimizers, number of epochs, batch sizes, etc.

```
def my_model_fn(features, labels, mode, params):
    is_training = mode == tf.estimator.ModeKeys.TRAIN

    logits = mymodel.createModel(features, is_training=is_training, num_classes=10, inputResolution=28)

    if mode == tf.estimator.ModeKeys.PREDICT:
        return tf.estimator.EstimatorSpec(mode=mode, predictions={
            "classes": tf.argmax(logits, 1),
            "logits": logits,
            "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
        })
    if(WITHDUMMYCLASS):
        labels=tf.subtract(labels,1)
    loss = tf.reduce_mean(
        tf.losses.sparse_softmax_cross_entropy(logits=logits, labels=labels))

    global_batch_size = params["num_shards"] * params["batch_size"]

    decayRateConst=int (NUMBERPEREPOCH*0.8)# 1300 * 1000
    decay_steps =decayRateConst * params["num_epochs"] // global_batch_size
    learning_rate = tf.train.polynomial_decay(
        params["lr"],
        global_step=tf.train.get_or_create_global_step(),
        end_learning_rate=params["min_lr"],
        decay_steps=decay_steps,
        power=0.01, #1.0 Linear Decay
        cycle=False)

    lr_repeat = tf.reshape(
        tf.tile(tf.expand_dims(learning_rate, 0), [params["batch_size"],]),
        [params["batch_size"], 1])

    if params["optimizer"] == "adam":
        optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
    elif params["optimizer"] == "rmsprop":
        optimizer = tf.train.RMSPropOptimizer(
            learning_rate=learning_rate,
            momentum=params["momentum"],
            epsilon=0.0000000001)
    else:
        optimizer = tf.train.MomentumOptimizer(
            learning_rate=learning_rate,
            momentum=params["momentum"],
            use_nesterov=True)

    train_op = optimizer.minimize(loss, tf.train.get_global_step())
```

FIGURE 3.8: Implementation of the neural network on Tensorflow

The code shown above calls the function 'createModel' which defines the structure of the model. The function is shown below:

```
import numpy as np

from modelutil import *

def CBRP_module(inputs, depth, name, withPool,is_training=True ):

    with tf.variable_scope(name, "ConvBatchPool", [inputs], reuse=tf.AUTO_REUSE):
        conv = conv2d(inputs, depth, name="conv")
        epsBN=1e-3
        meanVarBetaGamma=tf.constant(np.full((4,depth),epsBN,dtype=np.float32), dtype=tf.float32 )
        bn = tf.nn.batch_normalization(conv, meanVarBetaGamma[0], meanVarBetaGamma[1], meanVarBetaGamma[2],
            meanVarBetaGamma[3], epsBN,name='batchnorm' )
        relu = tf.nn.relu(bn , name="relu" )
        if(withPool):
            maxPool=tf.nn.max_pool(relu, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME',
                name="maxPooling" )
            return maxPool
        else:
            return relu

def createModel(images, is_training=True, num_classes=2,inputResolution=32 ):

    net = CBRP_module (images, 512 , "CBR1",True,is_training)
    net = CBRP_module (net, 64 , "CBR2",True,is_training)
    net = CBRP_module (net, 64 , "CBR2",True,is_training)
    net = tf.layers.flatten(net,name='Flatten')
    net = tf.layers.dense(net,10,name='FC', reuse= tf.AUTO_REUSE)
    return net
```

FIGURE 3.9: Model of neural network used.

After running the training model, the model and the weights are saved as (.pb) files. This file is later used to embed the NN into the FPGA.

3.4 Implementation of Neural Network into the FPGA

The following section will give a general overview of the work-flow of implementing the neural network into the FPGA. This is done by taking the help of a software called SensAI that converts the neural network model into a bitstream that can be downloaded into an FPGA.

3.4.1 Loading model into SensAI

Once the weights and the trained model have been stored in a file, this is to be loaded into SensAI. This software converts the model and the weights into a bitstream that can be used to program the FPGA. The programming frameworks supported are Keras, TensorFlow, and Caffe. The first step involves loading the model file into this program. The following image demonstrates loading a tensor flow (.pb) file into the software:

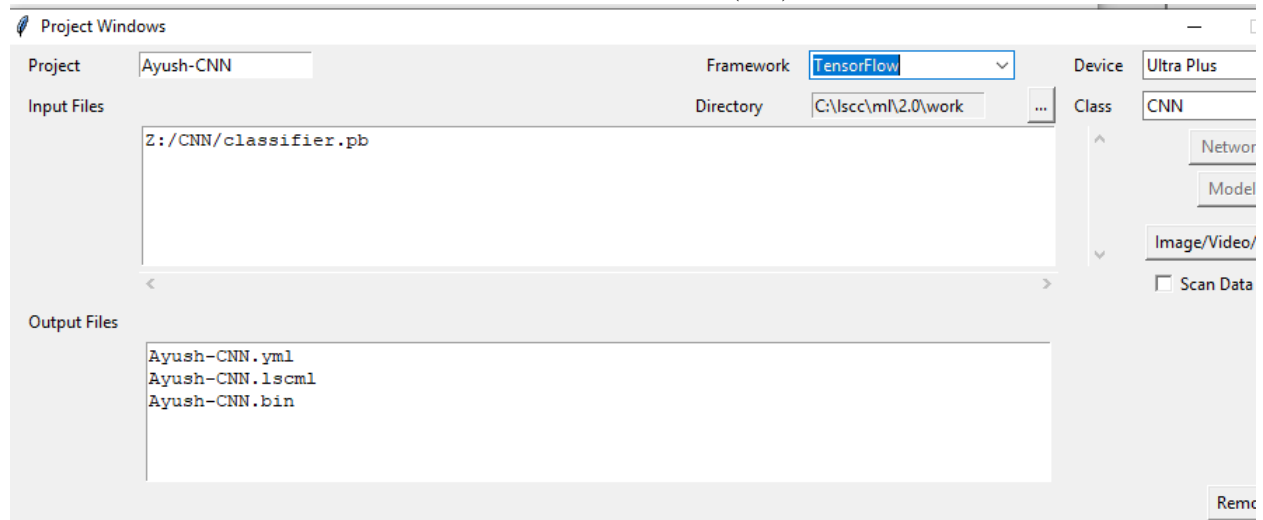


FIGURE 3.10: Project window in SensAI

Next follows the project implementation settings window :

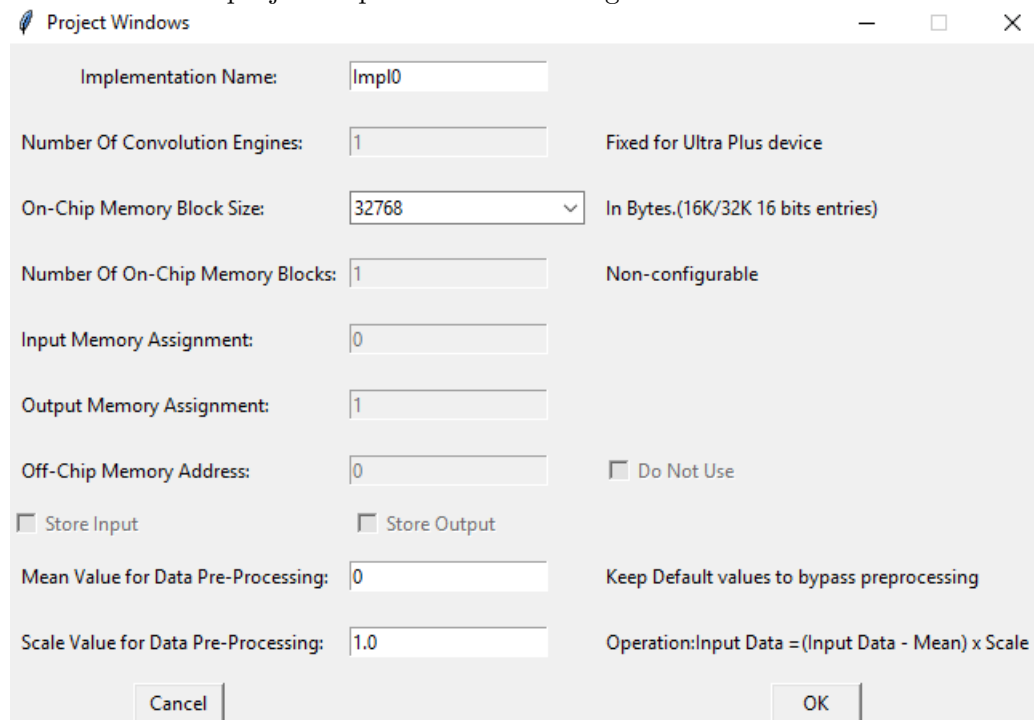


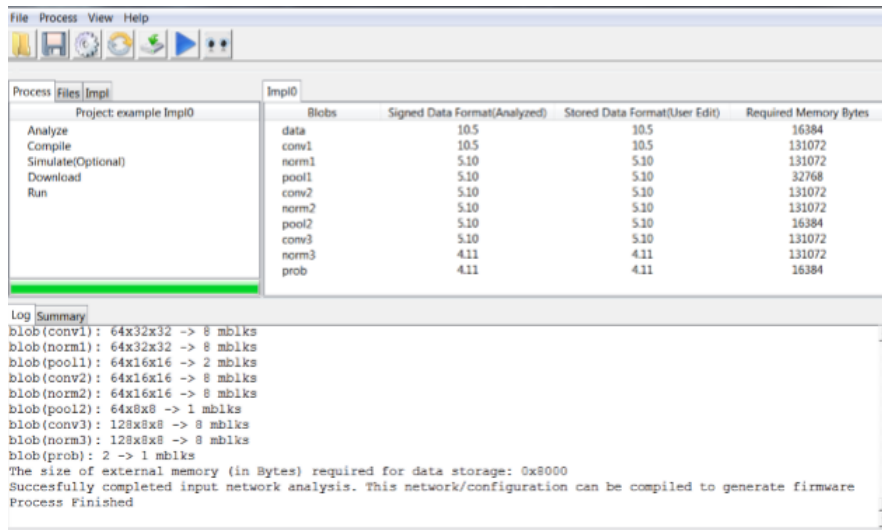
FIGURE 3.11: Project Implementation Setting Window

The program gives certain flexibility to the settings for implementation on the board:

- The number of convolution engines is fixed for the board in use (ICE40 UP) while can be varied from 1-8 on a board like EC5P.
- On-chip memory block size is the storage space used by each block implemented in the Embedded Block RAM(EBR), which is a lot faster than accessing DRAM. Higher the value, higher the operation frequency but it also means the use of a larger device.
- It gives the flexibility to normalize the data to a certain mean and variance.

3.4.2 Analysis and Compilation

This stage analyses the code to verify compatibility with the Lattice DNN project Compiler and. The 'Analyse' function must be run on the project before we can progress to the Compile or Simulate stages. Once the model has been analyzed, it lists down all the layers in the model and estimates the memory used by each layer.



Process	Files	Impl	Blobs	Signed Data Format(Analyzed)	Stored Data Format(User Edit)	Required Memory Bytes
Analyze		Project: example Impl0	data	10.5	10.5	16384
Compile			conv1	10.5	10.5	131072
Simulate(Optional)			norm1	5.10	5.10	131072
Download			pool1	5.10	5.10	32768
Run			conv2	5.10	5.10	131072
			norm2	5.10	5.10	131072
			pool2	5.10	5.10	16384
			conv3	5.10	5.10	131072
			norm3	4.11	4.11	131072
			prob	4.11	4.11	16384

Log	Summary
blob(conv1): 64x32x32 -> 8 mblks	
blob(norm1): 64x32x32 -> 8 mblks	
blob(pool1): 64x16x16 -> 2 mblks	
blob(conv2): 64x16x16 -> 8 mblks	
blob(norm2): 64x16x16 -> 8 mblks	
blob(pool2): 64x8x8 -> 1 mblks	
blob(conv3): 128x8x8 -> 8 mblks	
blob(norm3): 128x8x8 -> 8 mblks	
blob(prob): 2 -> 1 mblks	
The size of external memory (in Bytes) required for data storage: 0x8000	
Successfully completed input network analysis. This network/configuration can be compiled to generate firmware	
Process Finished	

FIGURE 3.12: Analysis and Simulation window in SensAI

This window also shows the data format of each layer. SensAI only supports unsigned 8-bit and signed 8- and 16-bit formats. The number preceding the period is the number of bits used to represent the integer component of the number, while the number following it is the number of bits used in the fractional component. For signed-data, the total number of bits is one less than the total number of bits used, as one bit is always used for signage. Therefore for 10.5 would mean a total of 5 bit for an integral part, 10 bit for fraction and 1 bit for sign = 16bits.

After running the analysis, the software also makes a flow diagram of the data path:

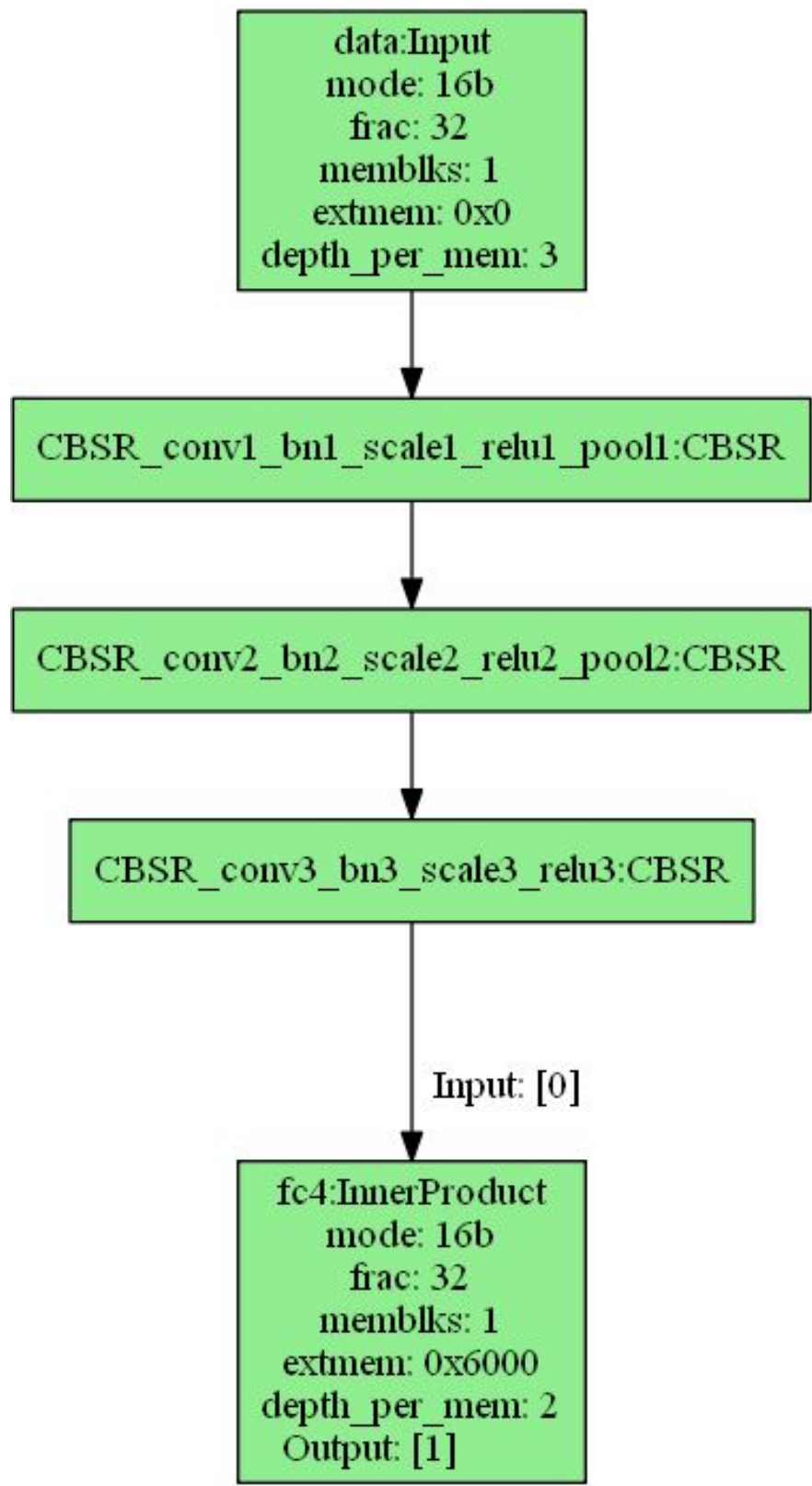


FIGURE 3.13: Flow Diagram

Once the analysis is completed, the next step is to compile the project. This generates a firmware file. It also generates a (.bin) file that can be used to download the network into the hardware. This (.bin) file is then downloaded into the FPGA using software called 'Diamond Programmer'.

3.4.3 Downloading the bit-stream into the FPGA

The bit-stream created is then supposed to be downloaded into the FPGA. The data is to be downloaded into the internal DRAM Memory blocks of the FPGA. This is where the biggest complication of this implementation shows up. When trying to download the bit-stream, the following error shows up:

```

Output
INFO - Check configuration setup: Start.

ERROR - Fuse size exceeds expected device fuse estimates.
The data file may not be valid for the device selected.
Device: iCE40UP5K
Device Fuse Estimate: 839680
Data file: C:/Users/amall/Desktop/impl2.bin
Data Fuse Number: 3653248

ERROR - Check configuration setup: Unsuccessful.
ERROR - Programming failed.

```

FIGURE 3.14: Error while downloading the bit-stream

This problem arises as the FPGA is not big enough to program all the functions of the Neural Network. The projected fuses used is 3653248, which almost 4 times the actual fuses present(839680). One possible solution was to decrease the complexity of the model. Therefore, a model with just one convolutional layer was implemented. That did bring down the number of fuses required to 2511032, but it was still above the acceptable limit, as can be seen in the following snippet:

```

Output
INFO - Check configuration setup: Start.

ERROR - Fuse size exceeds expected device fuse estimates.
The data file may not be valid for the device selected.
Device: iCE40UP5K
Device Fuse Estimate: 839680
Data file: C:/Users/amall/Desktop/imp3.bin
Data Fuse Number: 2511032

ERROR - Check configuration setup: Unsuccessful.
ERROR - Programming failed.

```

FIGURE 3.15: Persisting Error while downloading the bit-stream

3.4.4 Possible Solutions

The problem discussed in the previous section could arise due to various reasons like the size of the neural network is too big or the weights are bulky numbers. The problem couldn't be solved by making changes to the Convolutional Neural Network model. The

problem can be solved by developing a binary neural network for classification, as they have found to give a similar performance[18]. This algorithm converts all weights into a binary weight, which can take the values 1 or 0. Since the circuitry for 1-bit operations is a lot easier than floating-point operations, this would make the circuit less complex. This algorithm wasn't implemented in this project due to a constraint on time. Along with this, some rigorous Regularisation techniques like L1 regularisation should be used to drive the weights to small values. A more feasible solution could be using a board with a bigger FPGA, with a higher LUT count, capable of realizing more complex functions. This report doesn't indulge in these solutions due to a restriction on time but these solutions can be in the purview of a project that would be an extension to this one.

3.5 Conclusion

The project was divided into three major parts. The first part of the project involved collecting data from the sensor using a Raspberry Pi 3 board. This enabled me to have hands-on experience with using a Raspberry Pi board and taught me how to program the board. This involved programming the GPIO pins to control the multiplexer that selected the sensor from the array of sensors and using the I2C bus to read data from the selected sensor. Another part of the program was to save the data from the sensor into an excel sheet in the Raspberry device itself with a time stamp. The second part of the project was modeling and training a neural network that would help in classification. This was the most important part of the project and the part where I learned the most. Starting with no knowledge about the realm of Deep Learning, it was most interesting to learn about different types of neural networks, understand their working and learning about ways to improve them. Implementing all the things I learned, enabled me to develop an image classifier with very high classification accuracy. This process took a lot of iterations, changing and improving the model continuously, from initially having an accuracy of 80% to eventually having a 98% accuracy. The final model was a CNN, with ADAM optimizer and employed batch-normalization for each layer. The model was developed in two frameworks simultaneously (TensorFlow and Keras), to test out the ease of implementation using both. Once this was done, the next and the last stage involved implementing the said classifier on the FPGA. SensAI was the software used to convert the model into a bitstream to be downloaded into an FPGA. This stage helped discover the complications and restrictions while trying to embed a Neural network, leading to possible solutions to the problems faced. Despite not being able to realize this part of the project due to a constraint on time, this served as a good learning experience and can be used as a learning curve in realizing this goal.

Bibliography

- [1] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.
- [2] Raymond E Wright. Logistic regression. 1995.
- [3] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima, 2016.
- [4] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016.
- [5] Moritz Hardt, Benjamin Recht, and Yoram Singer. Train faster, generalize better: Stability of stochastic gradient descent. *arXiv preprint arXiv:1509.01240*, 2015.
- [6] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14:8, 2012.
- [7] P Liang and NK Bose. Neural network fundamentals with graphs, algorithms and applications. *Mac Graw-Hill*, 1996.
- [8] Forest Agostinelli, Matthew Hoffman, Peter Sadowski, and Pierre Baldi. Learning activation functions to improve deep neural networks. *arXiv preprint arXiv:1412.6830*, 2014.
- [9] Mercedes Fernández-Redondo and Carlos Hernández-Espinosa. Weight initialization methods for multilayer feedforward. In *ESANN*, pages 119–124, 2001.
- [10] Jing Li, Ji-hang Cheng, Jing-yuan Shi, and Fei Huang. Brief introduction of back propagation (bp) neural network algorithm and its improvement. In *Advances in computer science and information engineering*, pages 553–558. Springer, 2012.
- [11] Stuart Geman, Elie Bienenstock, and René Doursat. Neural networks and the bias/variance dilemma. *Neural computation*, 4(1):1–58, 1992.
- [12] Katarzyna Janocha and Wojciech Marian Czarnecki. On loss functions for deep neural networks in classification. *arXiv preprint arXiv:1702.05659*, 2017.

- [13] Mee Young Park and Trevor Hastie. L1-regularization path algorithm for generalized linear models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 69(4):659–677, 2007.
- [14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.
- [15] T Tieleman and G Hinton. Rmsprop: Divide the gradient by a running average of its recent magnitude. coursera: Neural networks for machine learning. *Tech. Rep., Technical report*, page 31, 2012.
- [16] Zijun Zhang. Improved adam optimizer for deep neural networks. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pages 1–2. IEEE, 2018.
- [17] Steve Lawrence, C Lee Giles, Ah Chung Tsoi, and Andrew D Back. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1):98–113, 1997.
- [18] Wei Tang, Gang Hua, and Liang Wang. How to train a compact binary neural network with high accuracy? In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.