Breast Cancer Detection Using CNN

Submitted in partial fulfillment of the requirements
for the degree of

PGDM Finance

by

A S Krishnapriya
(Roll No. xxxxxx)

Supervisors:
Dr.Rakesh Nigam

PGDM
MADRAS SCHOOL OF ECONOMICS
2020

Dedicated to my beloved parents.

Abstract

Early detection of breast cancer and it's correct identification can go a long way in not only saving a woman's life but also her quality of life. The identification of breast cancer depends on biomedical images such as mammography scans and histopathological slides which are obtained from biopsies. Analysing these images require specialists in the oncology domain, and there can be times of uncertainty and differences of opinion among experts. Computer aided diagnostic techniques step in to resolve such a problem. The use of Deep Learning in the field of radiology has shown great promise in improving early and accurate cancer detection.

Breast Cancer Detection using various Deep Learning algorithms by using mammographic scans and histopathological slides have been carried out by researchers in this domain. This study aims to compare the performance of ResNet of 3 different depths (a Convolutional Neural Network) trained from scratch for the task of classifying benign and malignant histopathological images. Kaiming initialization of weights was implemented and the final layer was replaced with a fully connected network with softmax activation.

Contents

List of Figures

List of Abbreviations

ANN	Artificial Neural Networks
CNN	Convolutional Neural Networks
ReLu	Rectified Linear Units
BreakHis	Breast Cancer Histopathological Database
MLP	Multi Layer Perceptron

List of Symbols

b	Bias
W	Weight
$\sum$	Summation
A	Vector of Activations
x_i	i^{th} input factor
y	actual output
$\hat{y}$	predicted output
α	learning rate
$g(.)$	the function of adding the product of weights and input to bias
Z	output of adding the product of weights and inputs to bias
G	Convolution function
m	number of rows of input matrix
n	number of columns of input matrix
j	number of rows of kernel
k	number of columns of kernel
f	kernel
v	input image
F	Kernel Size
V	Input image size
P	amount of zero padding
S	Stride

n_c	number of channels
n_f	number of filters
$\mathscr{F}$	Residual Function

Chapter 1

Introduction and Motivation

1.1 Background

The breast consists of different tissues such as very fatty tissue to very dense tissue. A network of lobes lie within these tissues, and each lobe is made up of lobules which are small tube-like structures. Blood and lymph vessels are present throughout the structure of the breast. Cancer of the breast is when the cells present begin to grow out of control and form a mass of cells. This mass is called a tumor. There are benign tumors as well as malignant. A benign tumor will not spread to other parts of the body, but a malignant tumor will spread, and this is called metastasis. [Fer+18] BRCA1/2 genes, increased exposure to radiation,and increased exposure to estrogen hormone are some of the risk factors for breast cancer. It is primarily detected by a self examination of the breast by the patient, and upon consulting with their medical professional usually a mammography, and then if deemed necessary a biopsy is done.[Rak+18]

Mammography, MRI (Magnetic Resonance Imaging), Ultrasound of the breast, PET (Positron Emission Tomography, thermography are the imaging techniques used to detect a mass or an

abnormality in the breast. A biopsy, which is taking a sample of suspected cancerous tissue via a needle or surgical incision, is done to confirm the the presence of cancer and to identify it as benign or malignant[Fer+18]. The tissue is studied under the microscope by a professional and these histopathological images are used to identify which type of benign or malignant form of cancer it is. This interpretation is a very time consuming and strenuous process as there are various different types of cancer. However, the correct identification of the histopathological images is crucial in deciding the course of treatment.[NMK18]

1.2 Motivation

During the biopsy, the tissue is collected and stained with hemotoxylin and eosin before analysis by pathologists. The staining gives a clearer picture by enhancing the nuclei and cytoplasm of the cells, making it easier to differentiate. The nucleus appears purple and and cytoplasm appears pinkish in colour after staining.It is shown in Figure 1.1. A pathologist analyzes the organization of the nuclei, density, variability and the overall structure of the tissues. Cancerous tissues are distorted in structure and have a higher density of nuclei. The diagnosis process is extremely crucial and according to a study by Elmore et al. the diagnostic concordance among specialists is approximately 75 percent. This gives rise to the need to automate such a labour-intensive task with Computer Aided Diagnostic techniques and improve the accuracy of diagnosis overall in the medical community. This is where Machine Learning techniques have emerged to provide reliable and intelligent solutions.

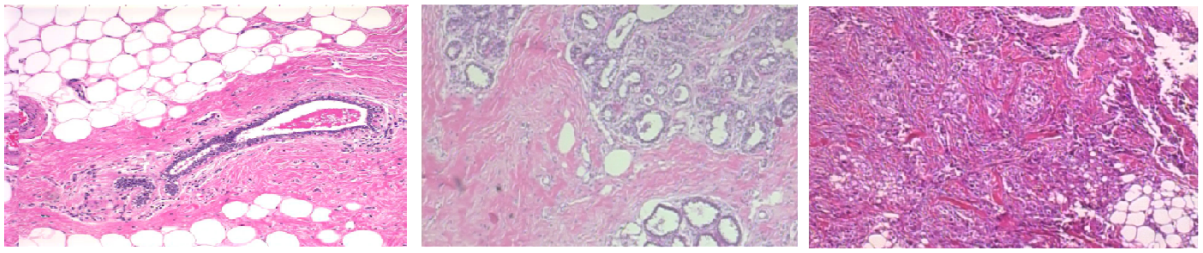


Figure 1.1: The nuclei appear purple and cytoplasm appears pinkish. image A is a normal tissue, B is a benign abnormality and C is a cancerous abnormality. We can see the density of nuclei(purple colour) to be dramatically different in the three images.

1.3 Thesis outline

The subject matter of the thesis is presented in the following five chapters,

- ✓ Chapter-1 Gives an introduction to breast cancer and the various imaging techniques by which it is detected. It also gives the motivation for the study.
- ✓ Chapter-2 gives a review of literature and traverses through the various studies conducted to detect breast cancer using deep learing models.
- ✓ Chapter 3 Describes the dataset in detail and discusses the challenges of handling such a dataset. It also discusses the python code used to handle the data as well as the method by which the data was made accessible to the google colab environment.
- ✓ Chapter-4 Discusses and gives an overview about Deep Neural Networks and their various frameworks. It also gives an introduction to CNN and the various types of CNN and the key differences between them.It gives a detailed framework about Resnet.
- ✓ Chapter-5 Gives the contributions of the thesis, the results and comparison of the various models on classification of the dataset.
- ✓ Chapter-6 Gives the conclusion and directions for future work.
- ✓ References and Appendix : A list of References and in the appendix the Backpropagation algorithms and the pytorch implementation of the study.

This page was intentionally left blank.

Chapter 2

Review of Literature

2.1 Introduction

Ongoing research for screening breast cancer at an early stage, be it with mammography images or histapathological images has been concentrated towards developing better deep learning models. In 2004 a study was conducted to evaluate the performance of Computer Aided diagnostics in screening breast cancer, with a follow up scan of the patients.[Ike+04]. An interesting article published in 2017 spoke about the over reliance of diagnostics of Machine Learning, it spoke about how Deep learning algorithms were not transparent enough, and why they do not work in some cases was still unclear.[LTH17]. It also stressed on the importance of understanding Deep learning algorithms for diagnosticians as the fields are now getting intertwined more than ever. A study that compared the perrformance of radiologists and that of Deep Learning Models [Rod+19a], showed the outperformance of the Deep Learning Models over Radiologists.An article published in 2019 [Rod+19b] showed an improved performance of radiologists in classifying breast cancer with the use of CNN. A simulation study [Yal+19b] to triage mammograms as cancer free showed improvement in sensitivity without affecting specificity of the

Deep Learning model. A hybrid DL model with Risk Factor based logistic regression and Resnet-18 was used to predict the risk of getting cancer in the next 5 years.[Yal+19a] The hybrid DL model showed AUC of 0.70. An ensemble model of by assembling compact CNNs and with a decision function was seen to perform reasonably well in a study conducted in 2019 [Zhu+19]. A study developed a novel BiCNN for the classification of breast cancer data. [Ben+17]. Another study compared the performance of classification of thyroid tumors using Renset and VGG, here VGG showed marginally better performance than Resnet.[Wan+19]. The datset used in this study is a widely accepted standard of histopathalogical data and has been used extensively in many research papers globally. One such paper compares the performance of various classical pre-trained Convolutional Neural Networks on the dataset. Out of these, Resnet and its variants showed higher performance.[Zho+20] Another study compared the performance of inception V2 with Resnet and here as well, the fine tuned ResNet architecture outperformed. [Mot+18]. Many variants of the ResNet module have been successful in classifying the histopathological dataset [Jia+19],[BFR18] [Rak+18] [Fer+18]

Hence, ResNet was used in this study, to compare the performance of various depths of Resnet models traines from scratch on the Breast Cancer histopathological Database. FOr understanding of CNN books by Ian Goodfellow [GBC16], Charu C agarwal [Agg18], and Ian H [WFH11] were referred to.

Chapter 3

The Dataset

3.1 Breast Cancer Histopathological Database

The Breast Cancer Histopathological Image Classification (BreakHis) is composed of 9,109 microscopic images of breast tumor tissue collected from 82 patients using different magnifying factors (40X, 100X, 200X, and 400X). It contains 2,480 benign and 5,429 malignant samples (700X460 pixels, 3-channel RGB, 8-bit depth in each channel, PNG format).

Two main groups form the dataset : Benign tumors and Malignant Tumors. Benign tumors in a histopathological perspective are those lesions that do not match any criteria of malignancy such as : marked cellular atypia,mitosis, disruption of basement membranes etc. Malignant lesions can destroy adjacent structures, metastasize and can be life threatening.

The samples present in the dataset are generated from breast tissue biopsy slides,stained with hematoxylin and eosin (HE). They are prepared for histological study and labelled by pathologists of the P and D Lab.The breast tumor specimens were assessed by Immunohistochemistry (IHC).The samples were collected Core Needle Biopsy (CNB) and Surgical Open Biopsy (SOB).Each section is of 3µm thickness in the sample. The images are acquired using

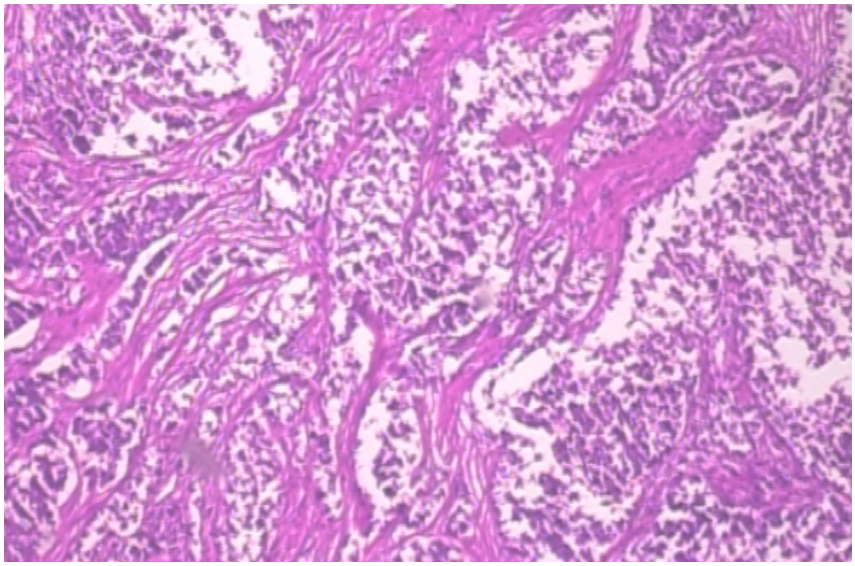


Figure 3.1: A Malignant Sample. Note the higher density of nuclei (purple in colour).

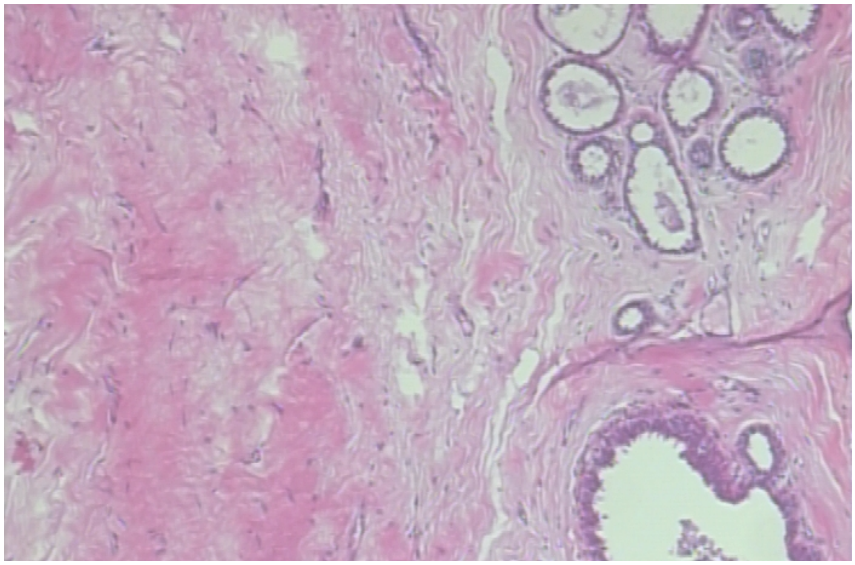


Figure 3.2: A Benign Sample. A lower density of nuclei present than in a malignant sample, but higher than a normal slide.

an Olympus BX-50 system microscope with a relay lens with magnification of 3.3× coupled to a Samsung digital color camera SCC-131AN There are 4 magnifications of the images : 40×, 100×, 200×, and 400× (objective lens 4×, 10×, 20×, and 40× with ocular lens 10×).The camera pixel size 6.5 µm and these are raw images without normalization nor color standardization. The resulting images saved in 3-channel RGB, 8-bit depth in each channel, PNG format. Below is a summary of the dataset in figure 3.3.

The BreaKHis 1.0 is structured as follows:

Magnification	Benign	Malignant	Total
40X	652	1,370	1,995
100X	644	1,437	2,081
200X	623	1,390	2,013
400X	588	1,232	1,820
Total of Images	2,480	5,429	7,909

Figure 3.3: A summary of the BreakHis Dataset. Source :BreakHis Database(https://web.inf.ufpr.br)

3.2 Challenges, Constraints and Data Augmentation

In order to train a Convolutional Neural Network to effectively classify images large amounts of data and computing power is required. In this study, due to constraints a total of thousand images of various magnifications were taken from the database and split into training, testing and validation sets in the ratio 80:10:10 . The data was uploaded into Google Drive which was then mounted onto the Google Colaboratory environment.Google Colab is a cloud platform that provides Graphics Processing Units or GPUs to be used remotely. A lot of researchers in Deep Learning use this platform. The code to mount the data onto the environmetn is given in the figure 3.5. The availability and accessibility of sufficient training data was a major constraint in this study. As this was raw data, the images needed to be normalised and resized to match the requirement of the CNN.

Data Augmentation techniques give a way around the issue of smaller data size as well as reduce over-fitting of the model. Data Augmentation refers to flipping, cropping, resizing, rotation, increasing sharpness of the images other such processes.

The main libraries used were PyTorch, Numpy, Sklearn and Pandas. Pytorch as defined in their official website, is an open source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing. It provides a lot of resources for deep learning and is widely used for deep learning projects. Scikit-learn

is a free software machine learning library for the Python programming language. This library was used for producing the metrics such as ROC curves and accuracy.Numpy and Pandas are standard libraries in python for handling data.

The path of the data is also set here, as it will be required to reference it later. The code to augment the data by applying transformations is given in the figure 3.6

In the paper "Research on data augmentation for image classification based on convolution neural networks," 2017 published in Chinese Automation Congress (CAC) by J. Shijie et al the importance of the usage of a combination of data augmentation techniques was made. They state that a combination of rotation, colour jitter, cropping, and flipping can go a long way in training the model better. The images below show side by side, an image sample after augmentation and before augmentation.

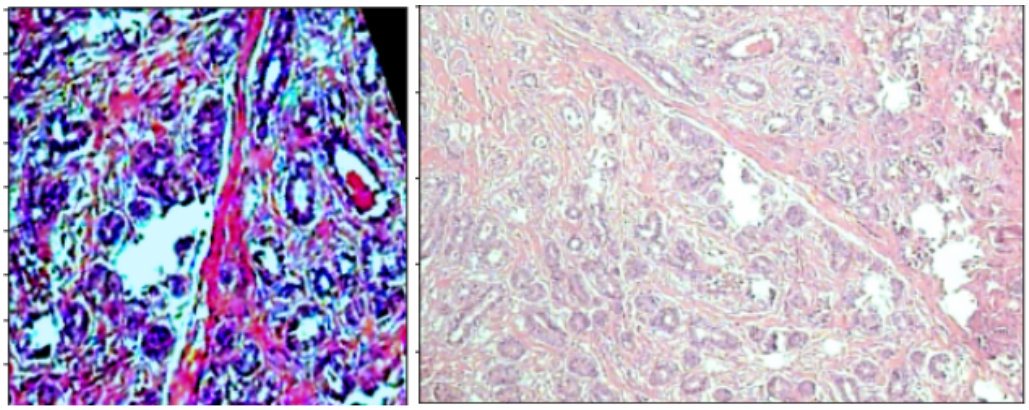


Figure 3.4: An augmented image and an un-augmented image

Figure 3.5: Code used to mount google drive onto google colab

```
data_transforms = {
'train': transforms.Compose([
        transforms.RandomRotation(90),
        transforms.RandomResizedCrop(224),
        transforms.ColorJitter(brightness=0.4, contrast=0.4, saturation=0.4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
'test': transforms.Compose([
        transforms.Resize(224),
        #transforms.ColorJitter(brightness=0.4, contrast=0.4, saturation=0.4),
        #transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
'val': transforms.Compose([
        transforms.Resize(224),
        #transforms.ColorJitter(brightness=0.4, contrast=0.4, saturation=0.4),
        #transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])

}
```

Figure 3.6: Code used for data transformations

Chapter 4

Deep Neural Networks

Deep Neural Networks or Deep Feed Forward Neural Networks or Multi-Layer Perceptrons are essentially Machine Learning Models that come together with a series of mathematical operations. The goal of a network is to approximate some function f^*, as explained in detail and an illuminating manner in the text book Deep Learning by Goodfellow et al[GBC16]

They describe this with an example of a classifier that maps an input x to a category y. Let the input be an animal or an item and the category be Living and Non-Living. A network defines a mapping y = f(x;Θ) and learns the value of the parameter θ that gives the best classification. To illustrate this function mapping, we can the input of a water bottle and a dog. We know that a water bottle falls in the category of a non-living thing and that of a dog falls in the Living category. What helped us arrive at the correct classification of the inputs were the parameters. A water bottle is non-living as it does not move, breathe, grow or reproduce. But a dog does all of these. Hence in Kindergarten we are taught the very basics of a classification. We take parameters into consideration such as growth and movement, and compare it to the input to arrive at our answer. This is exactly how a classification task is carried out by a Deep Neural Network.This is also why Neural Networks were inspired by neuroscience, each unit of an MLP is often called a neuron.

"Deep" and "Network" part of the name is arrived from how they are represented by many different mathematical functions layered on top of one another. Let's take three functions f^1 , f^2, andf^3 , connected in a chain they form f(x) = $f^3(f^2(f^1(x)))$. This chain structure is used in neural networks. The overall length of the chain gives the depth of the network. The final layer of a network is called the output layer, and we try to match f(x) to match $f^*(x) \approx y$. The training training examples must produce an output that is close to y, so the learning algorithm must decide how to use the layers to reach the best approximation of y.[GBC16]

4.1 A Perceptron

A description of Neural Networks seems incomplete without drawing parallels to a biological neuron. So to explain what a perceptron is let's look at an image of a biological neuron in figure 4.1

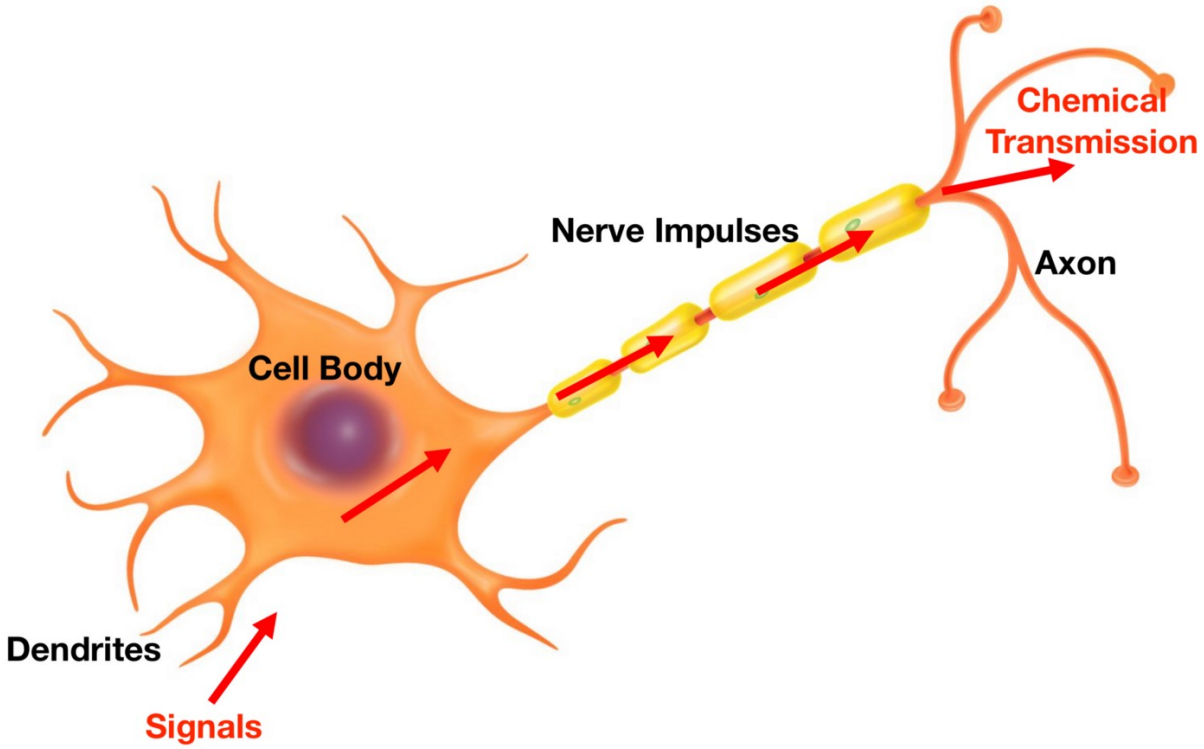


Figure 4.1: Biological Neuron, source : towardsdatascience.com

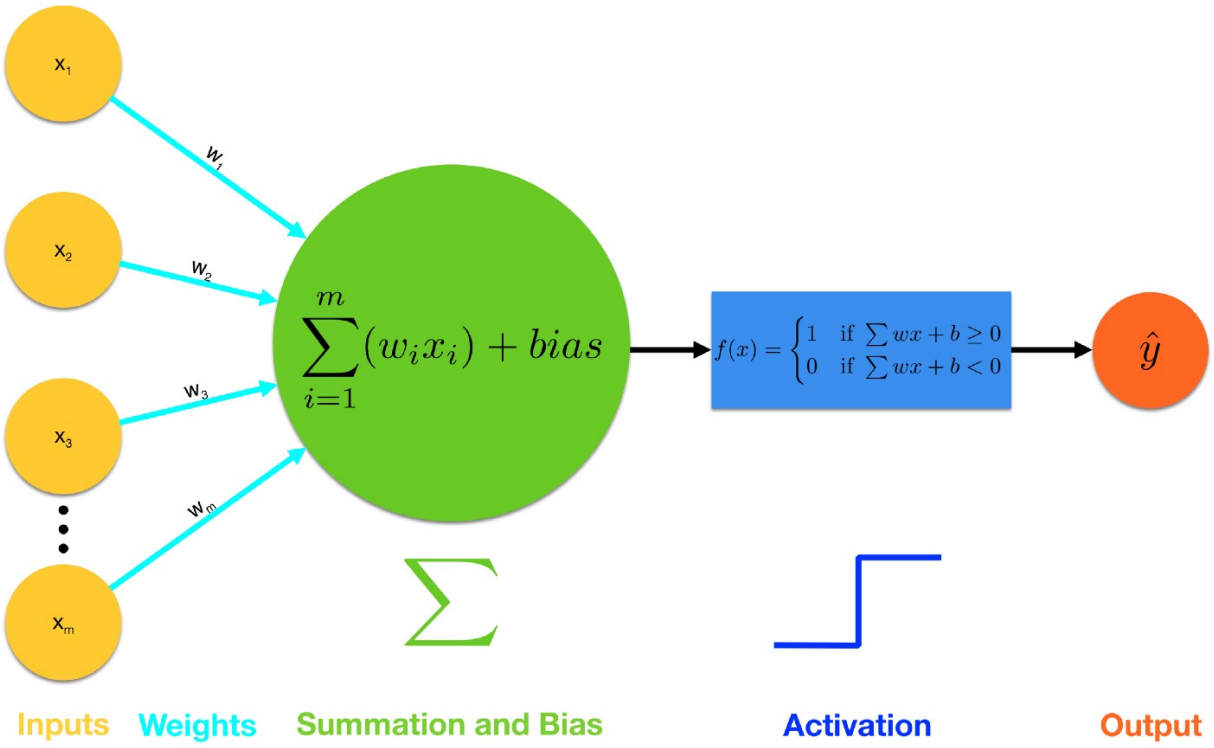


Figure 4.2: Perceptron : a mathematical representation, source : towardsdatascience.com

The dendrites of the neuron receive signals and then transmit signals to the body of the neuron where the stimulus is processed. After processing the stimulus, a decision is made whether to stimulate other neurons or not, i.e 1 or 0 becomes the "output". If the output is 1, the other neurons are stimulated by neurotransmitters released from the axons of the neuron. Similar to a nerve cell is the perceptron. The perceptrons receive inputs, and each input is multiplied by a weight w. This is then summed up say in the body of the perceptron and a predetermined number called a bias is added to this sum. If the number obtained is greater than 0, the ouput is 1, else it is 0. This is achieved by an activation function. In order to achieve an output, the summation is then passed through an activation function (which will be discussed in coming sections), and an output of 1 or 0 is arrived at. Now, in the form of an equation, all this would look like this :

$$output = f(x) = \begin{cases} 1 & \text{if } \sum w_i x_i + b \geq 0 \\ 0 & \text{if } \sum w_i x_i + b < 0 \end{cases}$$

Let's illustrate how a perceptron learns with an example. Say there are two classes Living and Non-Living taking the labels 1 and 0 respectively. Let there be three things that'll influence

the classification of inputs into these classes: Whether it moves, it grows and whether it can reproduce. Let's use x_1,x_2 and x_3 for these input factors respectively and assign a binary 1 or 0 i.e yes or no to them.

Suppose we are presented with a task of classifying a Book as living or Non-living, let's proceed with this classification with a perceptron in the following manner:

The input factor x_1 which corresponds to movement is takes the value 0 as a book cannot move. The input factor x_2 takes the value 0 as it cannot grow and the input factor x_3 also takes the value 0 as it cannot reproduce.

These inputs are then multiplied by weights. The weights can be initialized, the higher the weight the more influential the factor. Let w_1 = 6 , w_2 = 3 and w_3 =2 (These weights are initial weights). The important thing here is that the input factor cannot be changed it is the weights that can be changed to improve the learning process. Let the the threshold value be 5 hence, the bias b becomes -5. [Pyo19]

$$x_1 * w_1 + x_2 * w_2 + x_3 * w_3 - b \tag{4.1}$$

$$= 0 * 6 + 0 * 3 + 0 * 2 - 5 \tag{4.2}$$

$$= 0 - 5 \tag{4.3}$$

$$= -5 \tag{4.4}$$

Since -5 < 0 the output class is 0 which corresponds to Non-Living.

Say, suppose our input was a car and our objective was to classify a car as living and non-living, we would proceed in the following manner : The input factor x_1 which corresponds to movement is takes the value 1 as a car can, technically move. The input factor x_2 takes the value 0 as it cannot grow and the input factor x_3 also takes the value 0 as it cannot reproduce.

These inputs are then multiplied by weights. Let w_1 = 6 , w_2 = 3 and w_3 =2 the same as before. The threshold value is 5 hence, the bias b becomes -5 as in the previous example.

$$\begin{aligned} x_1 * w_1 + x_2 * w_2 + x_3 * w_3 - b \\ = 1 * 6 + 0 * 3 + 0 * 2 - 5 \\ = 6 - 5 \\ = 1 \end{aligned} \tag{4.5}$$

Since 1>0 the classifies labels the input car as living. But we know that this is incorrect. This difference between the ideal solution and the solution given by the network is called the *Loss*. Minimising this loss is how the perceptron learns. The perceptron can update w_1 so that a car is correctly classified as non-living. This can be achieved if w_1 is changed to 5.[Pyo19]

$$\begin{aligned} x_1 * w_1 + x_2 * w_2 + x_3 * w_3 - b \\ = 1 * 5 + 0 * 3 + 0 * 2 - 5 \\ = 5 - 5 \\ = 0 \end{aligned} \tag{4.6}$$

Now, the car gets correctly classified as non-living.

Why have a bias or threshold? A high enough bias or threshold ensures that the dominating factor i.e the factor with the highest weight, must be satisfied for the perceptron to give an output of 1. [Pyo19] Varying the weights will result in different decision making models. This varying of the weights is done by using various learning algorithm such as Gradient Descent. The process of updating weights, i.e learning is done by backpropagation. The main aim here is to find the function minimum, so, yes a bit of calculus is involved. The learning algorithm in the toy example is as follows :

$$f(x) = \begin{cases} 1 & \text{if } \sum w_i x_i + b \geq 0 \\ 0 & \text{if } \sum w_i x_i + b < 0 \end{cases}$$
$$\text{Learning algorithm:} \quad w_i = w_i + (y - \hat{y}) x_i$$

Backpropagation is a mathematical algorithm that is used to calculate gradients, which is

then used to update the weights.[Ska19] The following equation describes an l layer deep neural network, where W is the weight parameter and b the bias, A is the vector of activations,

$$\begin{aligned} W^{[l]} &= W^{[l]} - \alpha dW^{[l]} \\ b^{[l]} &= b^{[l]} - \alpha db^{[l]} \end{aligned} \tag{4.7}$$

$$\begin{aligned} \mathbf{dW}^{[l]} &= \frac{\partial L}{\partial \mathbf{W}^{[l]}} = \frac{1}{m}\mathbf{dZ}^{[l]}\mathbf{A}^{[l-1]T} \\ \mathbf{db}^{[l]} &= \frac{\partial L}{\partial \mathbf{b}^{[l]}} = \frac{1}{m}\sum_{i=1}^{m}\mathbf{dZ}^{[l](i)} \\ \mathbf{dA}^{[l-1]} &= \frac{\partial L}{\partial \mathbf{A}^{[l-1]}} = \mathbf{W}^{[l]T}\mathbf{dZ}^{[l]} \\ \mathbf{dZ}^{[l]} &= \mathbf{dA}^{[l]} * g'(\mathbf{Z}^{[l]}) \end{aligned}$$

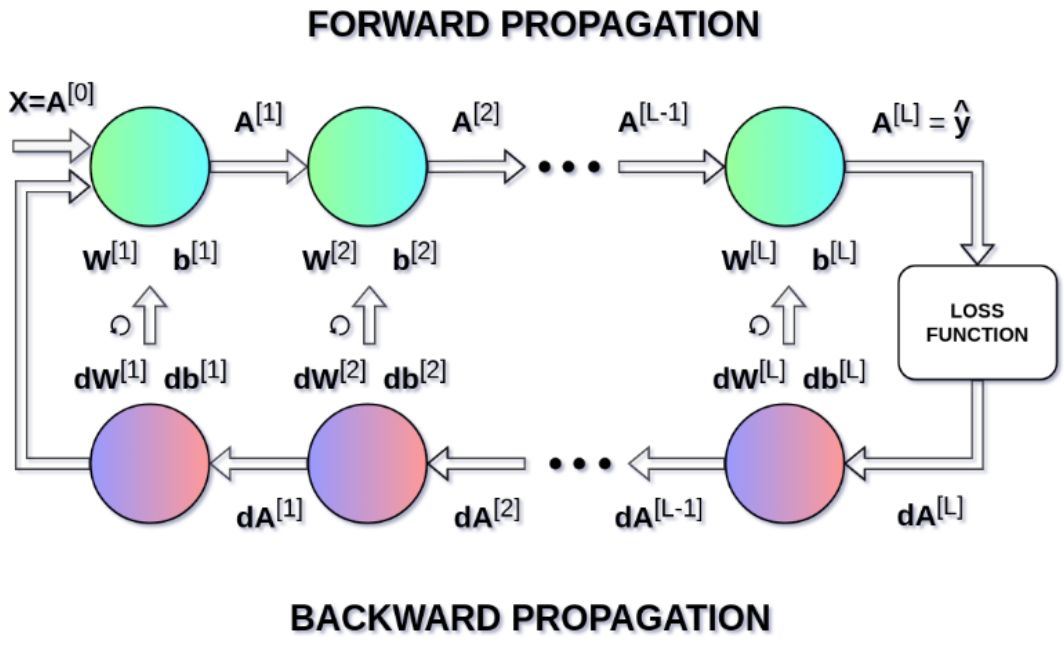


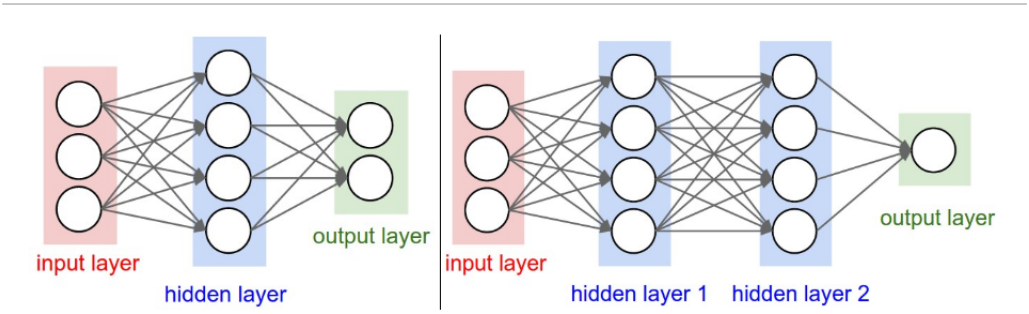
Figure 4.3: Forward and Backward Propagation, source:https://towardsdatascience.com

Where α represents the learning rate. Learning rate is a hyper-parameter. It can be best described in the following manner. If we were to run down a hill to a valley, looking for the deepest hole or puddle in the valley we can rush through it easily. But, as we speed across the

terrain, we may miss the truly deepest hole in the valley. If we were to go too slow, it would take too long and too much energy. Hence, going at the right pace is important. This is what the learning rate essentially is.

In this paper, A detailed explanation of Stochastic Gradient Descent is explained in the appendix.

This is an example of a single perceptron, a row of perceptrons form a layer and multiple layers form a **Multi Layered Perceptron** or a Deep Neural Network.



Left: A 2-layer Neural Network (one hidden layer of 4 neurons (or units) and one output layer with 2 neurons), and three inputs.
Right: A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Notice that in both cases there are connections (synapses) between neurons across layers, but not within a layer.

Figure 4.4: ANN source :CS231 [FRD19]

4.2 Activation Functions

An activation function or a transfer function is used to get the output from a perceptron. It transforms the output to within a range of 0 to 1 or -1 to 1. There are broadly two types Linear and Non-Linear Activation functions.

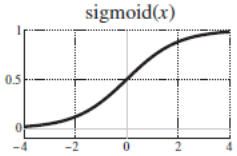
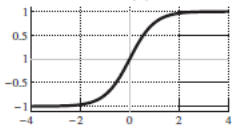
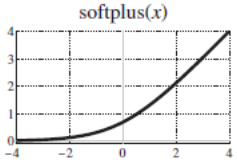
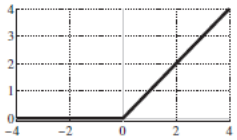
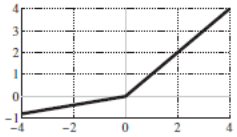
Name and Graph	Function	Derivative
sigmoid(x) 	$h(x) = \frac{1}{1 + \exp(-x)}$	$h'(x) = h(x)[1 - h(x)]$
tanh(x) 	$h(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$	$h'(x) = 1 - h(x)^2$
softplus(x) 	$h(x) = \log(1 + \exp(x))$	$h'(x) = \frac{1}{1 + \exp(-x)}$
rectify(x) 	$h(x) = \max(0, x)$	$h'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$
pw_linear(x) 	$h(x) = \begin{cases} x & \text{if } x \geq 0 \\ ax & \text{if } x < 0 \end{cases}$	$h'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ a & \text{if } x < 0 \end{cases}$

Figure 4.5: Activation functions and their derivatives source :Data Mining [WFH11]

4.2.1 Linear

Linear functions are as they sound, linear. When plotted they give a striaght line. They lie between $-\infty$ and ∞ and the equation is f(x) = x.

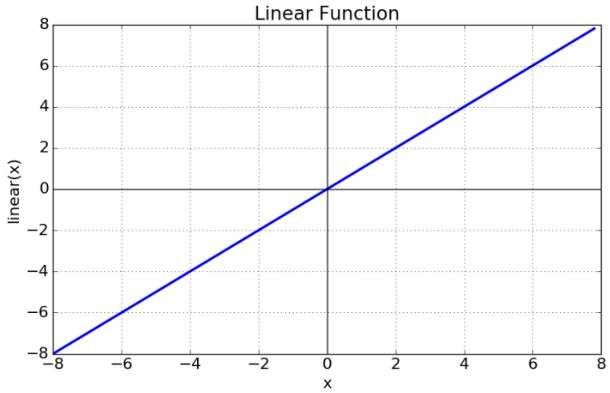


Figure 4.6: Linear Function

4.2.2 Non-Linear functions

Examples of Non-Linear functions include the sigmoid function, TanH, ReLu, and Leaky ReLu.

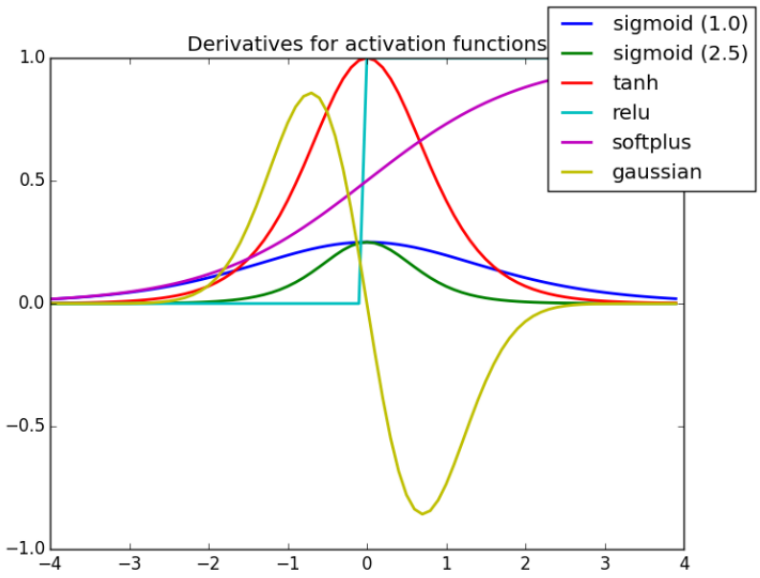


Figure 4.7: Non Linear Activation Functions, source:https://towardsdatascience.com

4.2.2.1 Sigmoid Function

This is a differentiable monotonic function. When plotted it takes the form of an S and is used for getting probabilities as the output lies between 0 and 1. The softmax function which is used in this study is a generalised form of this function.

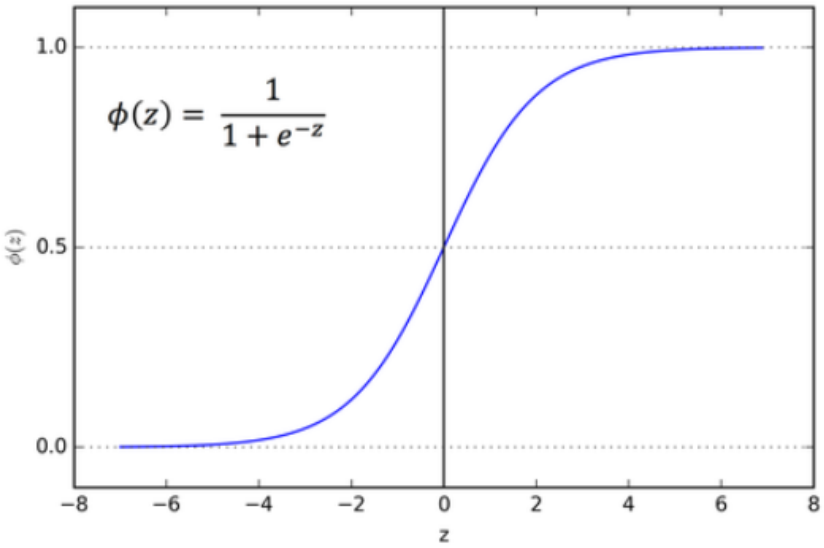


Figure 4.8: Sigmoid function and equation source : towardsdatascience.com

The softmax function is special in the sense that given an input of vectors of real numbers it gives us an output of real number,s such that the values in the output vector sum up to 1. In the softmax function instead of selecting one maximal element,the input vector is broken up into parts that sum up 1, the largest input gets a greater proportion. It is a function that maps $\mathbb{R}^N \rightarrow \mathbb{R}^N$

$$S(\mathbf{a}) : \begin{bmatrix} a_1 \\ a_2 \\ \dots \\ a_N \end{bmatrix} \rightarrow \begin{bmatrix} S_1 \\ S_2 \\ \dots \\ S_N \end{bmatrix} \tag{4.8}$$

As the output of the softmax function ranges from 0 to 1 it is often used to estimate the probability.

Computing the derivative of the softmax function is slightly different, as the input and output are vectors. It is important to understand of which output element the derivative is found for and with respect to which input element. Hence, to denote the partial derivative and the elements $D_j S_i$ is used in the following equations.

$$D_j S_i = \frac{\partial S_i}{\partial a_j} = \frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} \tag{4.9}$$

The derivative of the softmax function is obtained in the following manner with the help of the quotient rule:

$$f(x) = \frac{g(x)}{h(x)} \tag{4.10}$$

$$f(x) = \frac{g'(x)h(x) - h'(x)g(x)}{[h(x)]^2} \tag{4.11}$$

$$Here: \tag{4.12}$$

$$g_i = e^{a_i} \tag{4.13}$$

$$h_i = \sum_{k=1}^{N} e^{a_k} \tag{4.14}$$

For a_j the derivative will always be e^{a_j}. The derivative of g_i with respect to a_j is e^{a_j} only if i= j else it is 0.

$$\frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} = \frac{e^{a_i}\Sigma - e^{a_j}e^{a_i}}{\Sigma^2} \tag{4.15}$$

$$\tag{4.16}$$

$$\frac{\partial \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a_k}}}{\partial a_j} = \frac{e^{a_i}\Sigma - e^{a_j}e^{a_i}}{\Sigma^2} \tag{4.17}$$

$$= \frac{e^{a_i}}{\Sigma}\frac{\Sigma - e^{a_j}}{\Sigma} \tag{4.18}$$

$$= S_i(1 - S_j) \tag{4.19}$$

For i $\neq$ j :

$$\frac{\partial \frac{e^{a_i}}{\Sigma_{k=1}^{N} e^{a_k}}}{\partial a_j} = \frac{0 - e^{a_j} e^{a_i}}{\Sigma^2} \tag{4.20}$$

$$= -\frac{e^{a_j}}{\Sigma} \frac{e^{a_i}}{\Sigma} \tag{4.21}$$

$$= -S_j S_i \tag{4.22}$$

And finally it can be written as :

$$D_j S_i = \begin{cases} S_i(1 - S_j) i = j \\ -S_j S_i i \neq j \end{cases} \tag{4.23}$$

The softmax function is often used with the negative log likelihood function as the Loss function in ML models. In this study the Loss function is the Negative Log Likelihood Function which is described as in the following section.

4.2.2.2 Negative Log-Likelihood Function

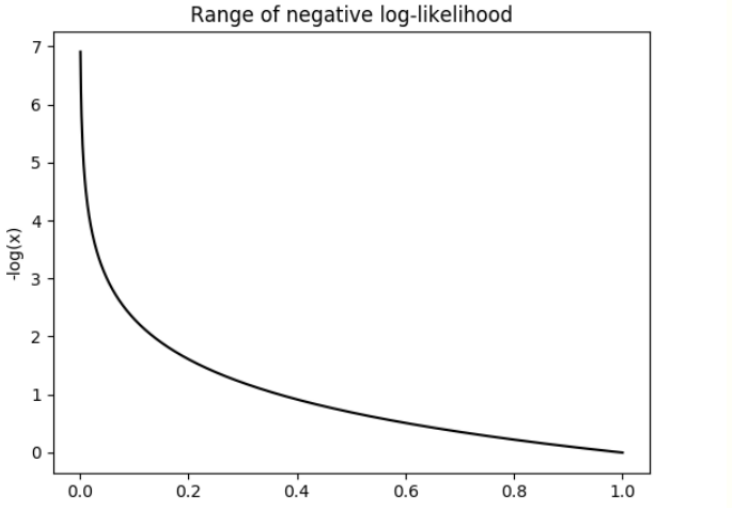


Figure 4.9: Graph for Negative Log Likelihood Function, source:https://towardsdatascience.com

The function is represented as

$$L(y) = -log(y) \tag{4.24}$$

Now, let's differentiate the softmax function with respect to the negative log likelihood function. We shall denote f as a vector containing the class scores for a single example, that is, the output of the network. Thus f_k is an element for a certain class k in all j classes. The softmax output denoted by p_k is :

$$p_k = \frac{e^{f_k}}{\sum_j e^{f_j}} \tag{4.25}$$

And the negative log likelihood function is :

$$L_i = -log(p_{y_i}) \tag{4.26}$$

During the backpropagation phase, we need to see how the loss changes with respect to the output, i.e we need to find the gradient or the partial derivative:

$$\frac{\partial L_i}{\partial f_k} \tag{4.27}$$

The Loss function L depends on p_k and this in turn requires f_k we can write the partial derivative in the following manner using the chain rule :

$$\frac{\partial L_i}{\partial f_k} = \frac{\partial L_i}{\partial p_k}\frac{\partial p_k}{\partial f_k} \tag{4.28}$$

So first we solve this $\frac{\partial L_i}{\partial p_k}$

$$\frac{\partial L_i}{\partial p_k} = -\frac{1}{p_k} \tag{4.29}$$

then we solve this :

$$\frac{p_{y_i}}{\partial f_k} \tag{4.30}$$

To solve the second part we need to recall the quotient rule, let the derivative operator be represented by **D**

$$\frac{f(x)}{g(x)} = \frac{g(x)\mathbf{D}f(x) - f(x)\mathbf{D}g(x)}{g(x)^2} \tag{4.31}$$

$$\tag{4.32}$$

Let $\sum_j e^{f_j} = \Sigma$, this is because in the softmax function we always seek the derivative of the k^{th} element. In this case it will be 0 in non k elements and e^{f_k} at k. By substituting, we get :

$$\frac{\partial p_k}{\partial f_k} = \frac{\partial}{\partial f_k}\left(\frac{e^{f_k}}{\sum_j e^{f_j}}\right) \tag{4.33}$$

$$= \frac{\Sigma \mathbf{D}e^{f_k} - e^{f_k}\mathbf{D}\Sigma}{\Sigma^2} \tag{4.34}$$

$$= \frac{e^{f_k}(\Sigma - e^{f_k})}{\Sigma^2} \tag{4.35}$$

Expanding on the expression we get :

$$\frac{\partial p_k}{\partial f_k} = \frac{e^{f_k}(\Sigma - e^{f_k})}{\Sigma^2} \tag{4.36}$$

$$= \frac{e^{f_k}}{\Sigma}\frac{\Sigma - e^{f_k}}{\Sigma} \tag{4.37}$$

$$= p_k * (1 - p_k) \tag{4.38}$$

Now, we combine the two derivatives :

$$\frac{\partial L_i}{\partial f_k} = \frac{\partial L_i}{\partial p_k}\frac{\partial p_k}{\partial f_k} \tag{4.39}$$

$$= -\frac{1}{p_k}(p_k * (1 - p_k)) \tag{4.40}$$

$$= (p_k - 1) \tag{4.41}$$

Hence, the derivative of the Negative Log Likelihood function with respect to the softmax layer is obtained. These are the functions that are used in this study.

4.2.2.3 ReLu

The ReLu function or the Rectified Linear Units function has is a very important function in the field of Deep Learning. This function is the most used function in Deep Learning, often acting as it's own layer. It is defined as max(x,0) ranging from 0 to infinity.

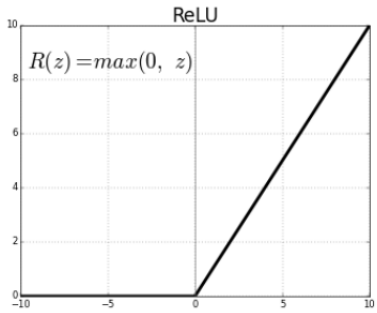


Figure 4.10: Graph for ReLu source :towardsdatascience.com

4.3 Convolutional Neural Networks

Before the discussion of CNN is begun, a discussion on how digital images are stored becomes important. Digital images are actually huge matrices of numbers and each number corresponds to the brightness of a pixel. In color images, the image is composed of three such matrices corresponding to the three color channels Red, Green and Blue (RGB). For black and white images, only one matrix is required to store the image. Each value stored in these matrices ranges from 0 to 255 (256 values fit in 1 byte). [Pyo19]

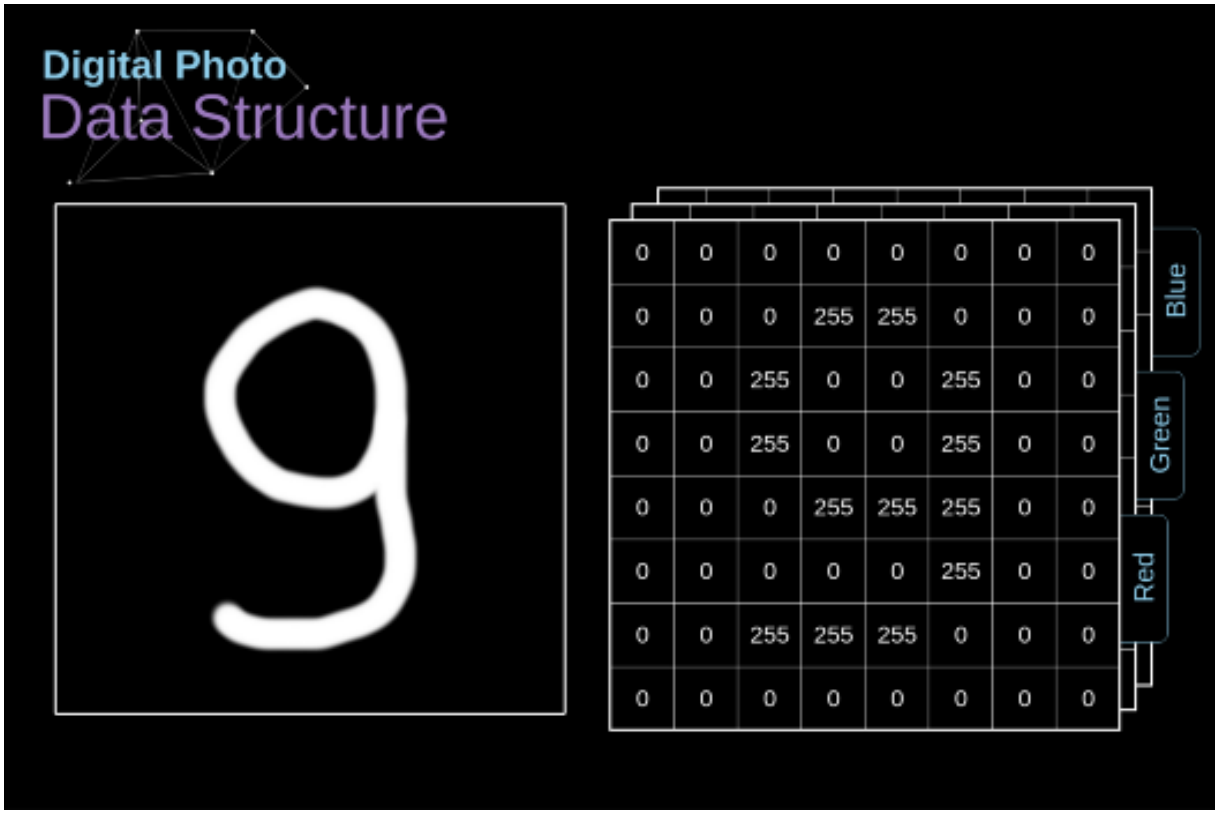
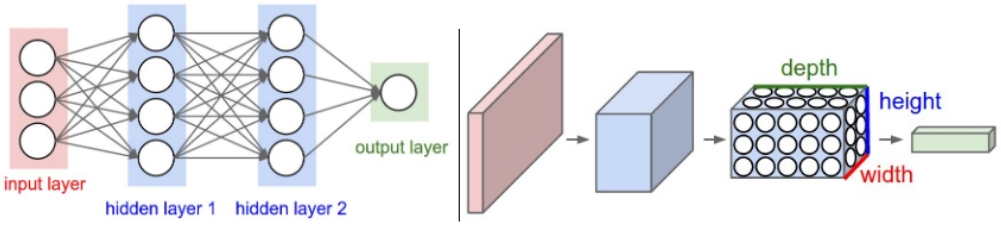


Figure 4.11: Digital Photo Data Structure source:https://towardsdatascience.com

A convolutional neural network can be considered as a special type of a Neural Netowork where there is a convolutional layer present. It is the process or mathematical operation of convolution that enables the CNN to extract features from images. The mathematical operations and various layers of a CNN are discussed in the following sections.

Convolutional Neural Networks have the right architecture for dealing with inputs that are

images. The layers of a Convolutional Neural Network have 3 dimensions : height, Width and depth. A coloured image of size 224 X 224 will have the dimensions 224x224x3. The depth of 3 refers to the Red,Green and Blue color channels of the image. These dimensions at the end of a convolutional neural network get reduced to a single vector of class scores, arranged along the dimension of depth. [FRD19]



Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

Figure 4.12: Source : Stanford CS231

A CNN is a sequence of layers where each layer transforms one set of activations into another through activation functions.

The architecture of a typical CNN is as follows : A simple CNN would consist of the following layers :

- Input Layer : This layer will hold the raw pixel values of the image. Before an image is passed through the input layer, it may be resized, or cropped randomly as a part of the normalisation process.
- Conv layer : This is the feature extraction layer. How this layer works will be discussed in detail in the following section.
- ReLu Layer: This is an example of a layer with an activation function. The ReLu layer applies an elemnt wise non-linear activation such as max(0,x) thresholding at zero. This does not change the size of the input received from the previous Conv layer. This layer increases the non linearity of the images, i.e the features that have been extracted.
- Pooling Layer : A pooling layer is another building block of a CNN. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network. Pooling layer operates on each feature map independently.
- Fully Connected Layer : The Fully Connected Layer will compute the class scores and classify the images.

Some layers have parameters while others don't. The ReLu and Pooling layers perform fixed operations. The Conv and Fully Connected layers have parameters that are trained with gradient descent. The transformations of the Conv layer and the Fully Connected layer are a function of not only the activations from the input layer but also a function of the weights and biases of the neurons. After repeatedly training the network, the network parameters get adjusted to arrive at an output ever closer to the correct one.[FRD19] The architecture of a CNN can run multiple layers deep. There are many well established CNNs such as VGG, ResNet,Dense Net, Inception Net and others which are often the starting point of exploratory analysis. In this project ResNet was used to attempt to classify the Histopathological slides as Malignant and Benign.

So, in this manner the input image is passed through layer by layer and the pixel values of the image are transformed into class scores for classification.

4.3.1 Convolutional Layer

This layer is the most important layer of the network. The parameters of this layer are made up of kernles or filters. A filter essentially views a small part say 3X3 pixels but views the entire depth of an image i.e all three colors. Each filter slides or convolves across the whole image and computes a dot product between the values in the filter and pixel values.We must remember that an image is essentially a matrix of pixels. As we slide over the image with the filter the dot products get computed and are stored on a 2-dimensional activation map. An entire set of filters are present in convolutional layers and each will produce activation maps which will be stacked to be presented to the next layer. [FRD19]

Essentially, during convolution we take a small matrix of numbers (filter) and pass it our the image which is a matrix of pixel values, a dot product is computed between them. This transforms the image into a feature map based on the values of the filter, the mathematical expression for convolution is as follows :

$$G[m,n] = (v * f)[m,n] = \sum_j \sum_k f[j,k]v[m-j,n-k] \tag{4.42}$$

where the input image is denoted by v and our kernel by f. The indexes of rows and columns of the result matrix are marked with m and n respectively.

An image illustrating the process of convolution is presented below :

A receptive field or filter size is a hyper-parameter of a CNN. It essentially controls the field view or how many pixels are viewed by the filter at a time. As the filter slides across the image, whether it should move one pixel at a time or 5 can also be controlled and this parameter is called *stride*. Though the receptive filed captures only a small part of the image i.e a few pixels at a time, it is important to remember that it captures the depth, i.e all 3 color channels of

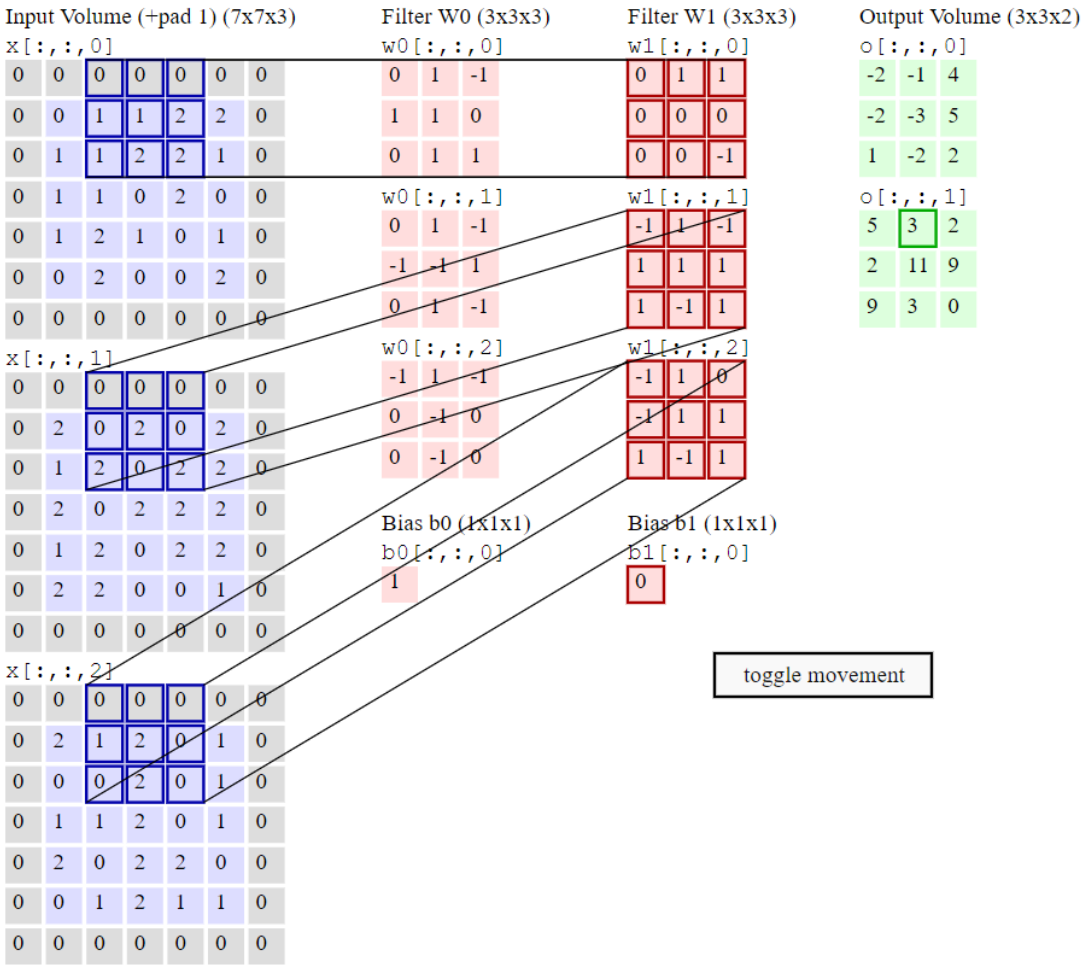


Figure 4.13: source : Stanford CS231

the image.[WFH11] Another hyper-parameter is depth, it describes how many filters are used.

Zero- Padding is the third hyper-parameter, here, we pad the input with zeroes. A vector of zeroes is put around the pixels of the image basically. This allows us to control the size of the output matrix. It ensures that the input volume and the output volume will have the same size spatially.

The size of the output of the convolutional layer can be computed in the following manner:

$$\frac{(V-F+2P)}{S}+1 \tag{4.43}$$

Where V corresponds to the size or volume of the input image, F corresponds to the receptive field or size of the filter, P refers to the amount of 0 padding used and S corresponds to the stride in equation 4.43.

When working with color images, we must understand that the process is in a 3 dimensional space. So, while handling color images,it is important the number of channels in the filter must be the same as the number of channels in the image. The process is similar to convolution as explained in the fig 4.13, but in 3 dimensional space, this is illustrated in figure 4.14. We create feature maps as before with multiple filters, and stack the feature maps on top of each other, this creates a 3 dimensional matrix called a tensor[Pyo19]. The dimensions of the tensor are as follows :

$$[V,V,n_c] * [F,F,n_f] = [[\frac{(V-F+2P)}{S}+1],[\frac{(V-F+2P)}{S}+1],n_f] \tag{4.44}$$

Where V — image size, F — filter size, n_c — number of channels in the image, P —used padding, S — used stride, n_f — number of filters.

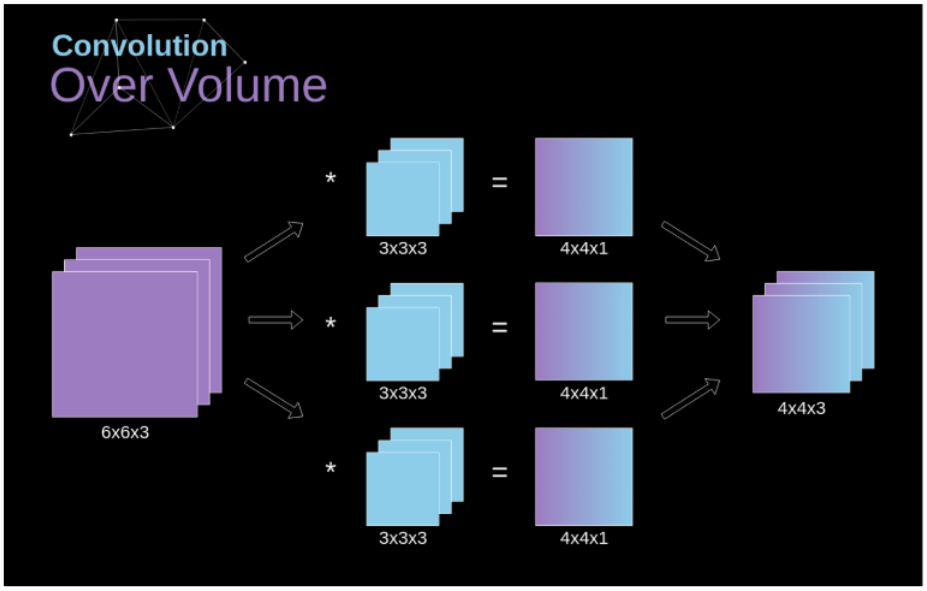
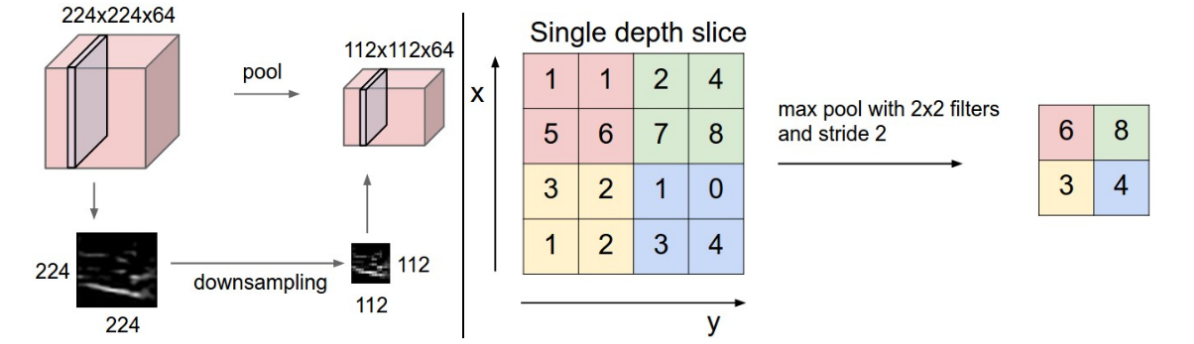


Figure 4.14: Volume aspect of convolution, source:https://towardsdatascience.com

This product of convolution is then added to a bias, and the their sum is activated by a non-linear activation function.This process is the forward pass in a CNN Just as in Deep Neural Networks our goal here is to again minimize a loss function. For this we calculate derivatives of the parameters and use them to update the weights. Here, the chain rule is used in the mathematical calculations. A detailed explanation of this process and the learning algorithm used is explained in the appendix.

4.3.2 Pooling Layers

Pooling layers are primarily used to reduce the size of the tensor (3 Dimensional matrix) and to speed up calculations. It reduces the size and also reduces the chance of over-fitting. We take the example of the commonly used Max Pool Layer, where we select the maximum value from each region and place it in the output correspondingly. Again, here we have two hyper-parameters available for this layer, filter size and stride. Pooling for multi-channel images are done separately for each channel. This is illustrated in figure 4.15



Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

Figure 4.15: Max-Pooling illustration and explanation, source:Stanford CS231

During backpropagation, the gradient does not affect the elements that were not included in the forward pass in this layer. The position of the elements used are marked (a mask) and used to calculate gradients in the backpropagation. This is illustrated in the figure 4.16 [Pyo19]

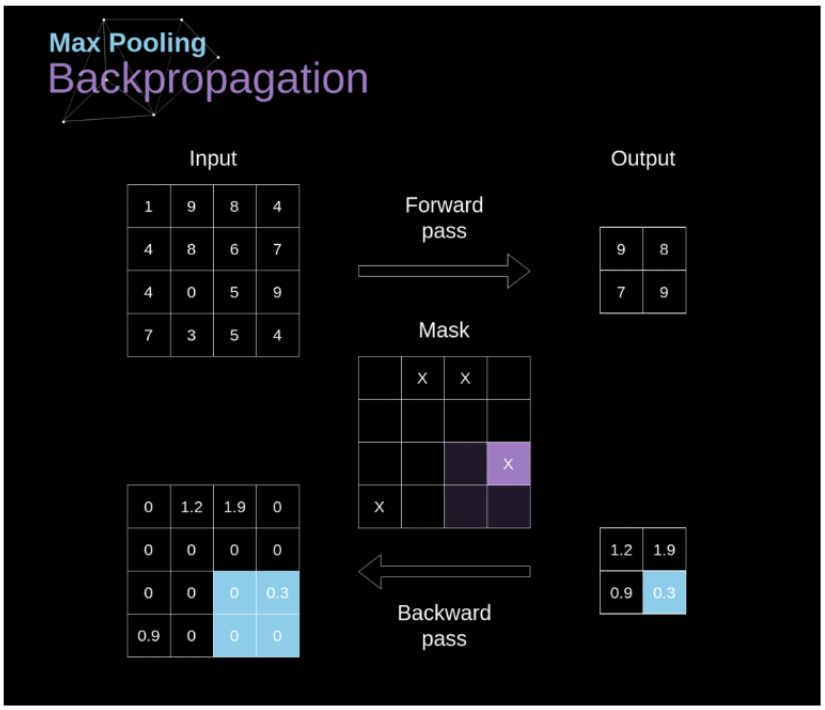


Figure 4.16: Max-Pooling Forward and Backward Pass illustration, source : towardsdatascience.com

Parameter sharing is done in convolutional layers to control the number of parameters. Here the neurons in a particular depth slice, i.e the 2 dimensional activation map, will have the same weights and bias.This reduces drastically the number of unique weights that need to be updated.

4.3.3 Fully-connected layer

The perceptrons of this layer are fully connected to all others as seen previously in Deep Neural Networks. Their activations and backpropagations are the same discussed in the previous section.

4.3.4 Layer Patterns

CNNs stack the Convolutional layer and the ReLu layer followed by a Pooling Layer and this pattern is repeated until the image is reduced to a much smaller size. Then this is fed into a Fully Connected Layer which calculates class scores. The way to describe these layers follows the following notation :

INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC

Here, convolutional layers are present before a pooling layer and the process is repeated N times. The FC stands for Fully Connected Layer. The POOL? indicates an optional pooling layer. N >= 0 , M >= 0, K >= 0.

4.4 Widely Used Convolutional Neural Networks

In this section we describe briefly some of the widely used CNNs. The legend for the illustraions of their architecture is below :

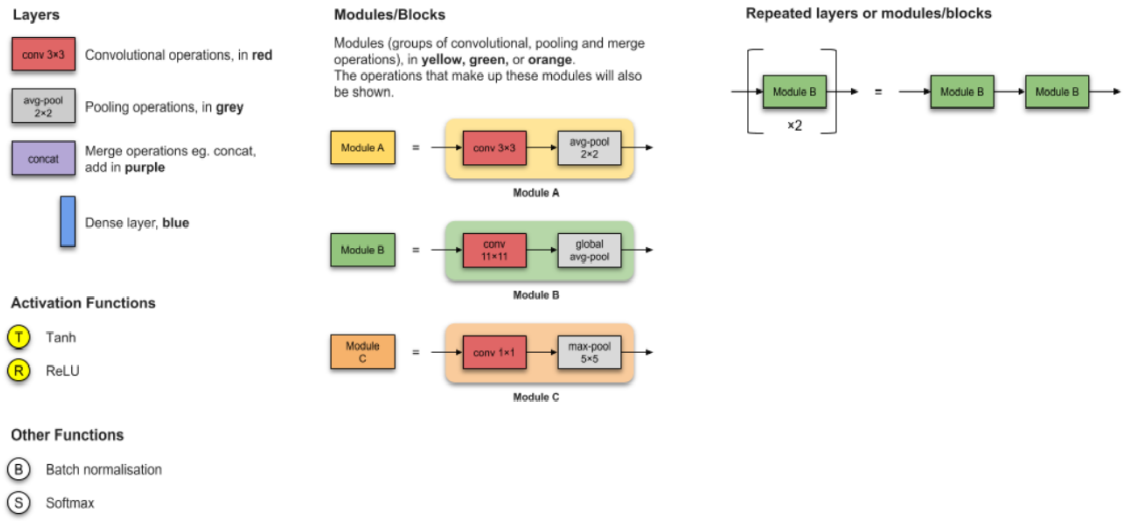


Figure 4.17: Legend for the illustrations, source : towardsdatascience.com

4.4.1 LeNet

This was one of the first succesful CNNs and follows a very simple architecture. It has 2 Convolutional Layers and 3 Fully COnnected Layers. It has about 60,000 parameters. This gave the template of stacking layers on top of each other and ending the architecture with a fully connected layer.

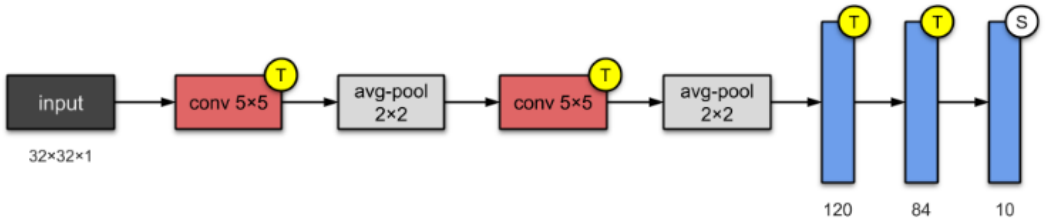


Figure 4.18: LeNet Architecture, source : towardsdatascience.com

4.4.2 AlexNet

AlexNet was the first to attempt Rectifies Linear Units as a layer. It has 60 million parameters, with 8 layers. 5 convolutional and 3 Fully connected layers.

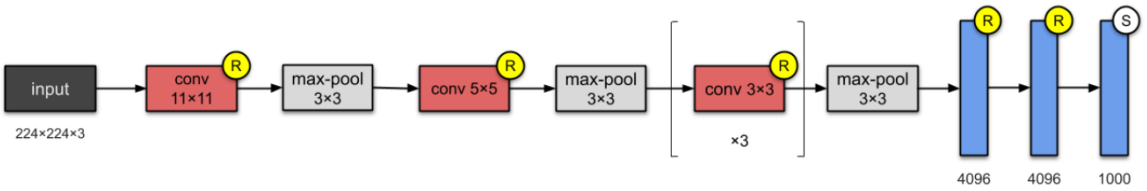


Figure 4.19: AlexNet Architecture, source : towardsdatascience.com

4.4.3 ZFNet

7x7 sized filters were used in this network, essentially this is similar to AlexNet, but is an improved version.

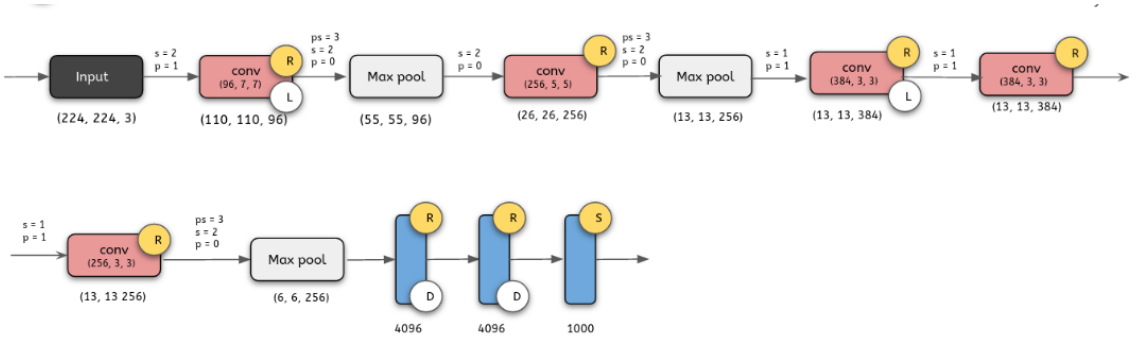


Figure 4.20: ZFNet Architecture, source : towardsdatascience.com

4.4.4 DeConv Neural Networks

A DeconvNet or a Deconvolutional Neural Network can be thought of as the opposite of a CNN. It uses the process of unpooling and deconvolution to put the input back to a larger size.

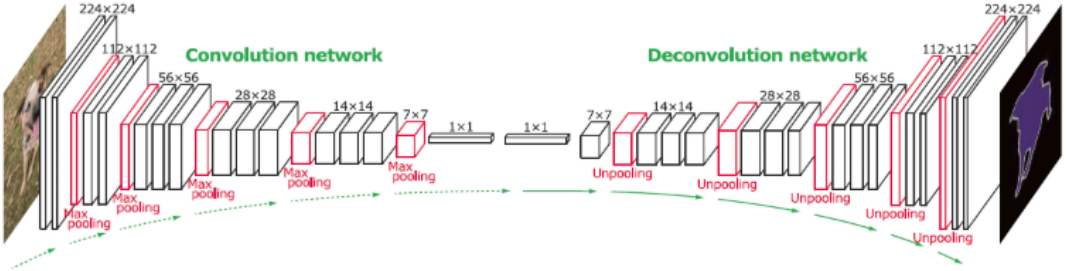


Figure 4.21: DeConvNet Architecture, source : towardsdatascience.com

4.4.5 VGG

Developed by the Visual Geometry Group, there are many variants of VGG such as VGG-16, VGG19 etc. VGG-16 has 13 COnvolutional layers with 3 fully connected networks and 138 million parameters.

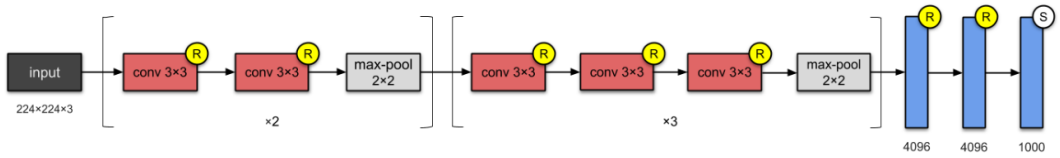


Figure 4.22: VGG-16 Architecture, source : towardsdatascience.com

VGG16 - Structural Details													
#	Input Image			output			Layer	Stride	Kernel		in	out	Param
1	224	224	3	224	224	64	conv3-64	1	3	3	3	64	1792
2	224	224	64	224	224	64	conv3064	1	3	3	64	64	36928
	224	224	64	112	112	64	maxpool	2	2	2	64	64	0
3	112	112	64	112	112	128	conv3-128	1	3	3	64	128	73856
4	112	112	128	112	112	128	conv3-128	1	3	3	128	128	147584
	112	112	128	56	56	128	maxpool	2	2	2	128	128	65664
5	56	56	128	56	56	256	conv3-256	1	3	3	128	256	295168
6	56	56	256	56	56	256	conv3-256	1	3	3	256	256	590080
7	56	56	256	56	56	256	conv3-256	1	3	3	256	256	590080
	56	56	256	28	28	256	maxpool	2	2	2	256	256	0
8	28	28	256	28	28	512	conv3-512	1	3	3	256	512	1180160
9	28	28	512	28	28	512	conv3-512	1	3	3	512	512	2359808
10	28	28	512	28	28	512	conv3-512	1	3	3	512	512	2359808
	28	28	512	14	14	512	maxpool	2	2	2	512	512	0
11	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
12	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
13	14	14	512	14	14	512	conv3-512	1	3	3	512	512	2359808
	14	14	512	7	7	512	maxpool	2	2	2	512	512	0
14	1	1	25088	1	1	4096	fc		1	1	25088	4096	102764544
15	1	1	4096	1	1	4096	fc		1	1	4096	4096	16781312
16	1	1	4096	1	1	1000	fc		1	1	4096	1000	4097000
Total													138,423,208

Figure 4.23: VGG-16 Architecture Details, source : towardsdatascience.com

The convolutional kernels are of size 3x3 and maxpooling kernels are of size 2x2. And the stride here is 2, all these are fixed. This was the novel idea presented in VGG which made

it better than AlexNet's variable size kernels. This reduces the number of trainable variables by 44.9 percent. This makes for faster learning and more robust to over-fitting. The structural details of VGG-16 are given in the figure 2.17.

4.4.6 GoogLeNet / Inception V4

The main contribution of GoogLeNet developed by Google was the development of the inception module. Also, they used average pooling instead of a fully connected layer at the end of the network. The Inception V4 is the improved version (one of many) of the GoogLeNet. It has 43M parameters, and the architecture for Inception V4 is given in figure 4.24

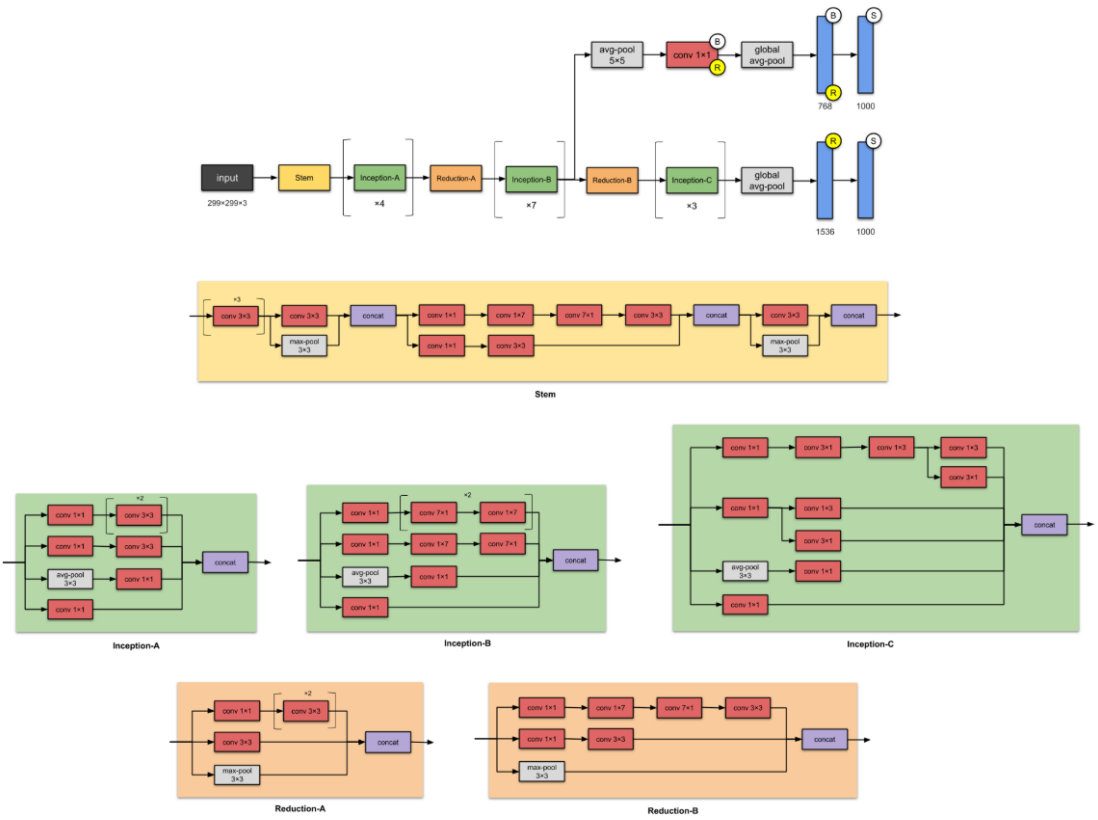


Figure 4.24: Inception V4, source : towardsdatascience.com

4.4.7 ResNet

ResNet or Residual Networks solve an important problem that arises while training networks. As the depth of the network increases, there is a degradation in accuracy. This is because as the layers get deeper and many activation functions are used, the output from each layer gets smaller and smaller due to the activation functions. For example the sigmoid function gives an output between 0 and 1. The input gets significantly reduced by the sigmoid function. This becomes a problem as during backpropagation. During backpropagation, the derivatives of the network parameters (weights and bias) are found from the final layer to the initial layer. By chain rule, these derivatives get multiplied through the network to compute the derivatives of the initial layers. So, if we were to use an activation such as a sigmoid function for l hidden layers, l small derivatives get multiplied and gradient diminishes drastically. Because of a very small gradient, the weights and biases do not get updated properly. This is termed as the vanishing gradient problem.[He+15]

Residual Networks solve this problem in a novel manner, A residual function is defined as : F(x) = H(x) -x. This can be reframed as F(x) + x = H(x). Here the F(x) represents the stacked non-linear layers and x represents the identity function.

It is easier to make F(x) = 0 than F(x) = x using stacked non-linear layers of a cnn. This F(x) is called the residual function This is the intuition behind the residual Block as illustrated by the figure 4.25

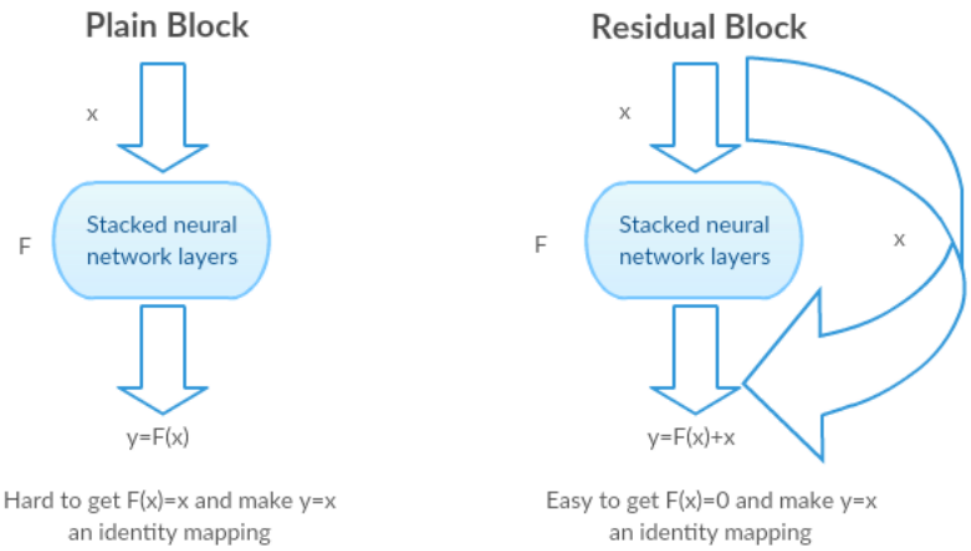


Figure 4.25: Residual Block,[He+15]

There are two kinds of residual connections:

- Identity shortcuts can be used when the input and output are of the same dimensions.

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + \mathbf{x}.$$

- When the dimensions are different, identity mapping is still performed by the shortcut with extra zero padding for the increase in size, this adds no extra parameters. Or else the projection shortcut is used to match the dimension with the following formula given below, this adds parameters.

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + W_s\mathbf{x}.$$

Each ResNet block is either 2 layer deep (Used in small networks like ResNet 18, 34) or 3 layer deep(ResNet 50, 101, 152). These are known as bottlenecks. The bottleneck in a neural network is just a layer with less neurons then the layer below or above it. Having such a layer encourages the network to compress feature representations to best fit in the available space, in order to get the best loss during training. A building block of Resnet is illustrated in figure 4.26

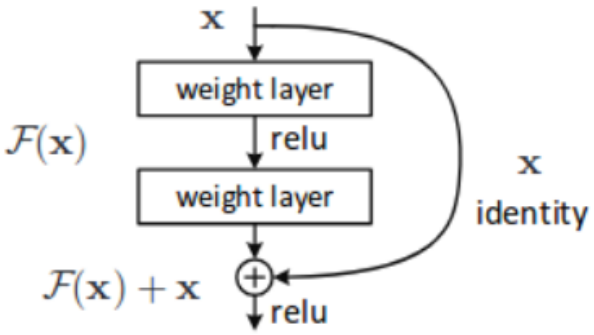


Figure 4.26: Building Blocks of Resnet source [He+15]

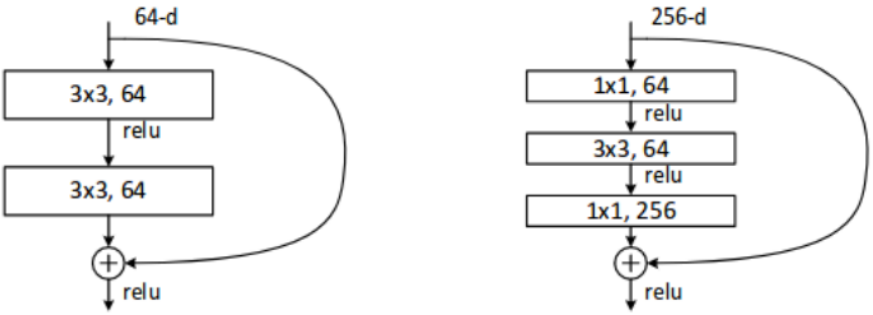


Figure 4.27: Building Blocks of Resnet, 2 layered and 3 layered, [He+15]

Various Resnet Architectures and their structural details are given below in the figure 4.28

If we note in the figure4.28 the 3 layer bottleneck is repeated many times and a 34 layer

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 4.28: ResNet Structural Details,[He+15]

residual unfolds into a 152 layered CNN. A detailed view of ResNet-152 which was used in this study is presented in figure 4.29

A pytorch application with the complete python code used in this study can be found in the appendix.

ResNets of various depths was used in this study to explore their performance in classifying the histopathological data. Their variations have been used by many researchers in the field of Computer Aided Diagnostics, and have been known to perform considerably better than others. ResNet was used especially because of the residual function involved and the advantages it presents to overcome the vanishing gradient problem as discussed in the previous section. Many studies also showed ResNets outperforming others [Rak+18]

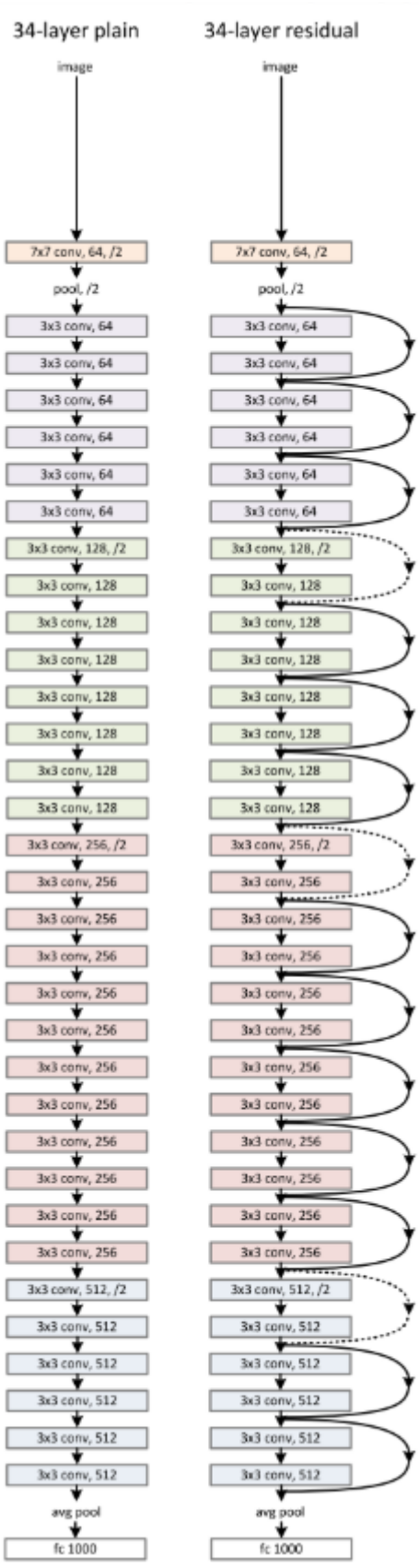


Figure 4.29: ResNet-152s, [He+15]

4.5 Regularization and Parameter Initialization

Standard techniques such as L1 and L2 regularization is applicable to CNN.An L2 regularization was applied in this study. This is found as "weight decay" in literature and in the pytorch implementation. This regularization is applied just to the weights in a network and not the biases.

Although the loss functions used in deep learning may not generally be convex, such regularizers can nevertheless be implemented.

Dropout : Dropout is a form of regularization where units and connections are randomly deleted. We can set this probability and is used to control overfitting. If a unit is retained with probability p during training, the outgoing weights are rescaled by a factor of p during test phase. This in effect creates an ensemble of 2n smaller networks.Pytorch has an inbuilt function for implementing dropout, where we specify the probability.

Batch Normalization : This is a method of accelerating training. Each element of a layer in a layer is normalised to zero and unit variance based on the statistics within its batch. In mini-batch Stochastic Gradient Descent the algorithm is modified by calculating the mean μ_j and variance σ_j^2 for each hidden unit h_j in each layer. and then normalizing the units. These are then scaled using the scaling parameter and shifting them by the learned shifting parameter. The scaling and shifting parameters are also updated by backpropagation.

Parameter Initialization : This is an important aspect of Deep learning, because if done wrong it can lead to exploding or vanishing gradients. Bias terms are initialised to 0 but weight matrices should not be initialised to 0, as all the neurons will get updated the exactly the same way and activation functions will give 0 gradients(TanH activation function), rendering the whole process useless. A common practice is to initialize weights as small random numbers. A distribution is also used to initialize the weights such as the uniform distribution over the interval [2b,b]. The inverse of the square root of the fan in size $h^{l-1}(x)$ is used to scale b.

The Kaiming initialization of weights is expressed as the following mathematical function:

$$\sigma = \sqrt{\frac{2}{n_l + \hat{n}_l}} \tag{4.45}$$

This page was intentionally left blank.

Chapter 5

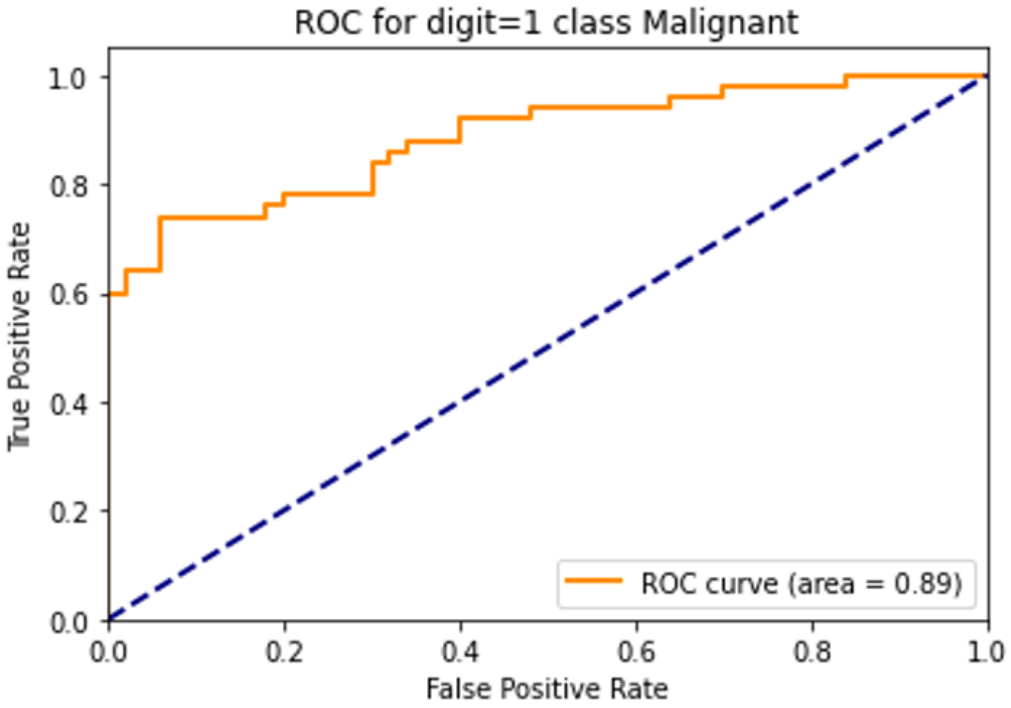
Results

In this study 1000 breast cancer histopathological images obtained fom the BreakHis database was used. This study aims to compare the performance of ResNet18, ResNet-50 and ResNet152 on a magnification independent dataset. Kaiming initialization of weights was implemented and an early stopping for 30 epochs was set in place. The Backpropagation algorithm used is Stochastic Gradient Descent and the Loss function used is Negative Log Likelihood. The results showed that ResNet-152 performed better than the three when trained from scratch. The results are enumerated below:

For ResNet-152 the best accuracy in the validation phase was 0.78 and in the test was the accuracy and F1 sore was 0.75. An AUC of 0.89 was recorded by the model.

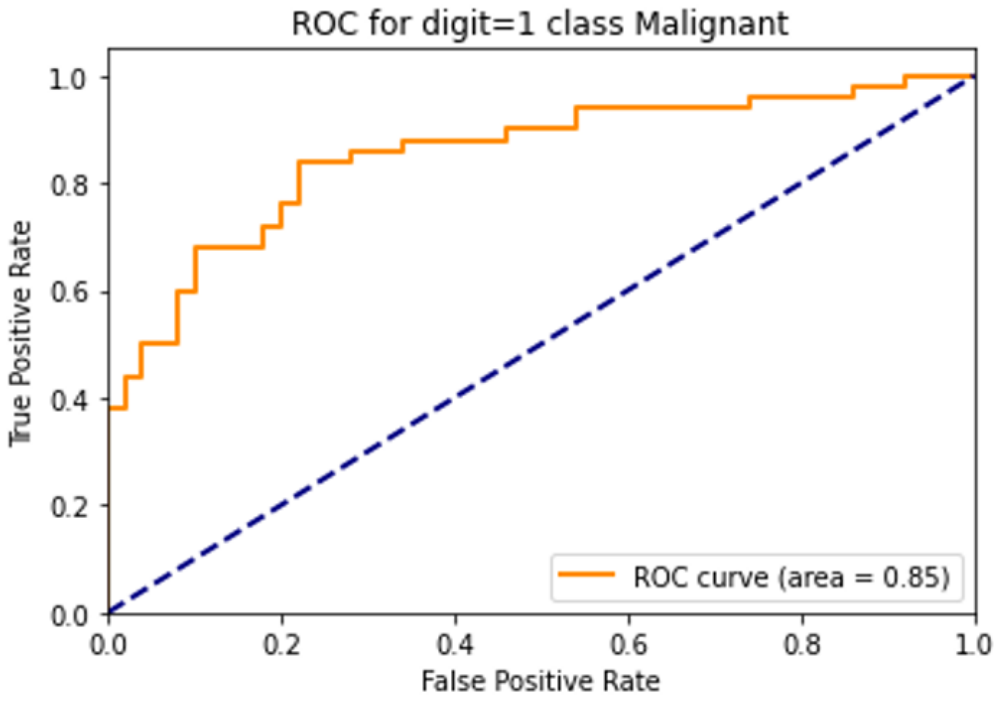
For ResNet-50 the best accuracy in validation phase was 0.72 and in test the accuracy was 0.70. An AUC of 0.85 was recorded. ResNet 18 was the worst performer, with a validation phase accuracy of 0.52 adn an accuracy of 0.5 on the test phase. An AUC score of 0.32.

The results are in conjecture with literature, as they have shown ResNet-152 to outperform the others. The poor performance of ResNet-18 could be attributed to the different bottleneck structure (1 layer) whereas ResNet-50 and ResNet-152 posses the 2 layered bottleneck and are also much deeper compared to ResNet-18.



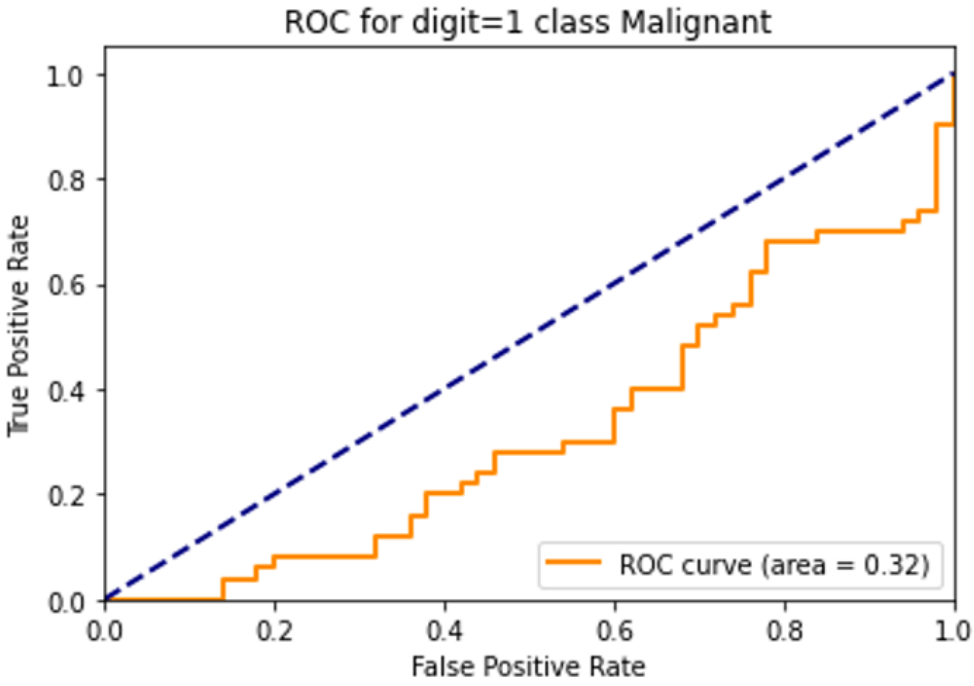

```
Confusion matrix:
[[36 14]
 [11 39]]
F1 score: 0.750000
Accuracy score: 0.750000
```

Figure 5.1: ROC plot for ResNet-152 and the confurion matrix




```
Confusion matrix:
[[25 25]
 [ 5 45]]
F1 score: 0.700000
Accuracy score: 0.700000
```

Figure 5.2: ROC plot for ResNet-50 and the confurion matrix




```
Confusion matrix:
[[50  0]
 [50  0]]
F1 score: 0.500000
Accuracy score: 0.500000
```

Figure 5.3: ROC plot for ResNet-18 and the confurion matrix

Chapter 6

Conclusions

While training from scratch ResNet-152 outperformed among the three. This could be because of the depth of the network and the number of residual bottlenecks being much larger than the other two. This is consistant with many other studies that used pretrained weights where ResNet-152. The way forward would be:

- To create an ensemble classifier of ResNets, and perhaps even a hybrid ensemble and study the performance of such a classifier.
- Using pretrained weights and then replacing the final layers with custom networks is a popular approach in attempts to further research in this field.
- In this study, only a broad classification i.e binary classification was attempted. The next step would be for a multi-class classification, to the type of cancer.A network should be unbiased and equally sensitive to all classes.Advanced Data Augmentation techniques such as the usage of conditional General Adversarial Networks and deep photo style transfer is another avenue that can be explored.
- Different weight initialization techniques such as Xavier, Gaussian distribution, MSRA are some other options. In this study Kaiming initialization of weights was attempted.

This page was intentionally left blank.

Appendix A

Backpropagation in CNN

Before we dive into backpropagation, we need to revisit the chain rule of calculus, as it is vital (some would even say "integral"!) to understanding gradient descent. Let's take the function f(x,y,z) = (x+y)z.

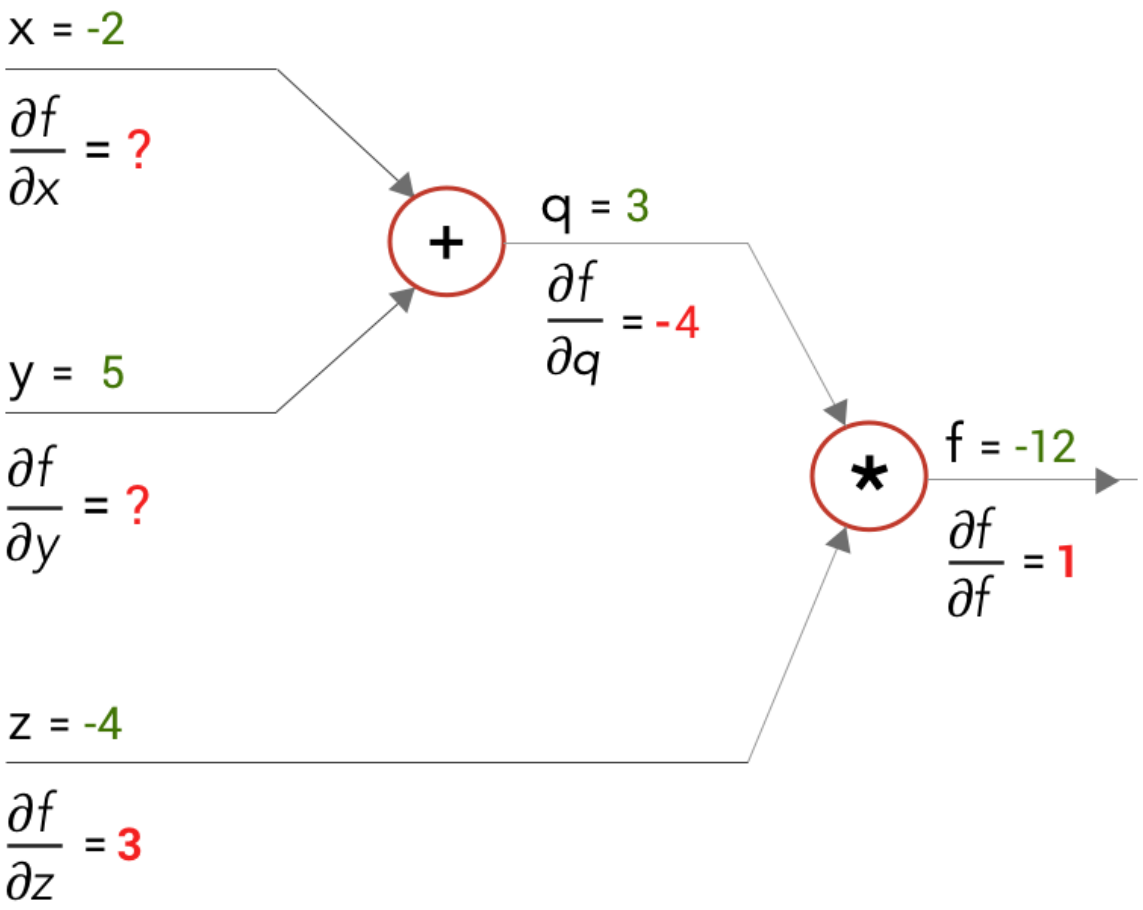
$$f(x,y,z) = (x+y)z \tag{A.1}$$

$$Let, q = (x+y) \tag{A.2}$$

$$then, f = q*z \tag{A.3}$$

Let's take a dummy example of x =-2 , y=5 and z =4, by the equation by we would get f = -12

This here is the forward pass. Now, let's get to the backward pass. We first compute the gradient of the output at every step. We need to find $\frac{\partial f}{\partial y}$ and $\frac{\partial f}{\partial x}$ This can be done using the chain rule as follows :



$f = q * z$

$\frac{\partial f}{\partial q} = z \mid z = -4$

$\frac{\partial f}{\partial z} = q \mid q = 3$

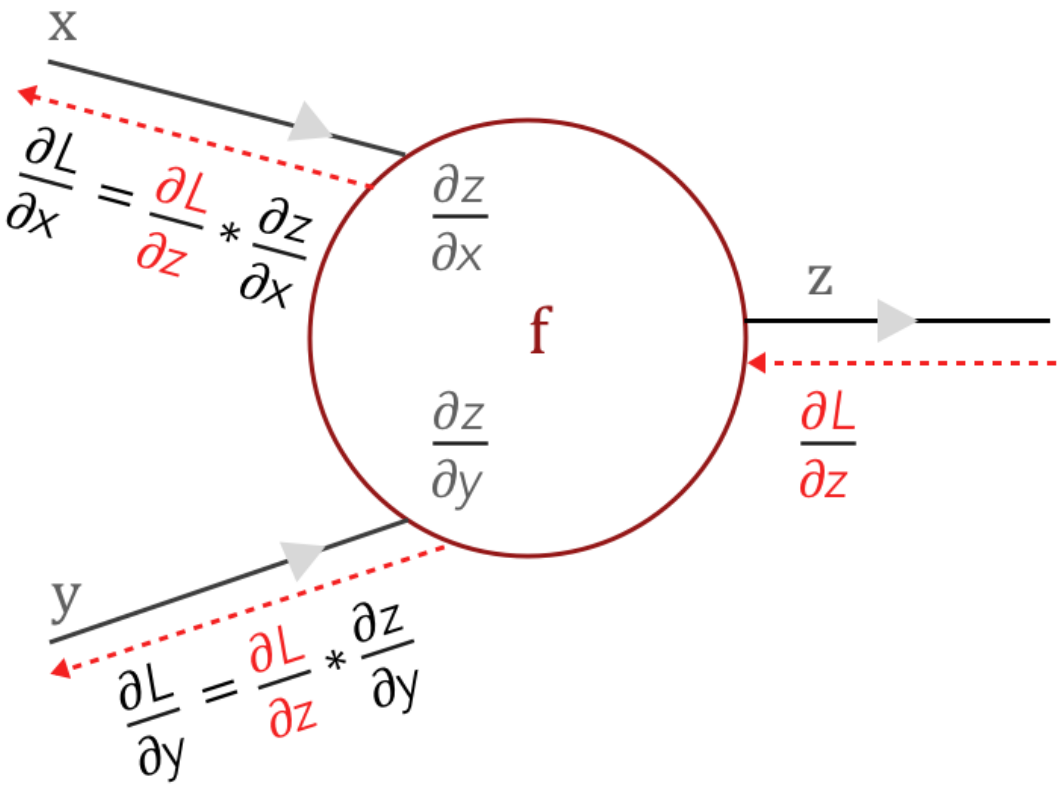
$q = x + y$

$\frac{\partial q}{\partial x} = 1 \qquad \frac{\partial q}{\partial y} = 1$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} * \frac{\partial q}{\partial x} = -4 * 1 = -4 \tag{A.4}$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} * \frac{\partial q}{\partial y} = -4 * 1 = -4 \tag{A.5}$$

A CNN is one such massive computational graph. Imagine a node with a gate (activation function) f that receives inputs x and y. Let's assume this gives an output z. The gradients of the output z with respect to x and y are : $\frac{\partial z}{\partial x}$ and $\frac{\partial f}{\partial y}$. These are called local gradients. During the forward pass, the input is passed through the CNN and the loss is calculated using the loss function. During the backward pass, we work this loss backwards by calculating the gradient of the loss from the previous layer. For this we find $\frac{\partial L}{\partial z}$. To propagate this loss across the layer, we need to find the partial derivative with respect to the input : $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial y}$. This is where the chain rule comes in. This process can be diagramatically represented as follows :

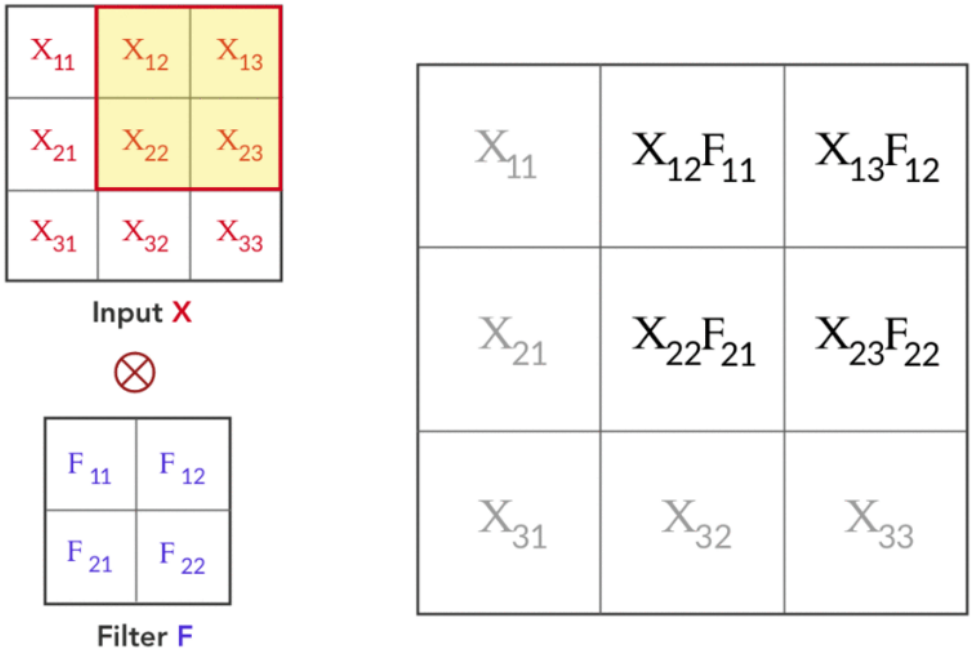


$\frac{\partial z}{\partial x}$ & $\frac{\partial z}{\partial y}$ *are local gradients*

$\frac{\partial L}{\partial z}$ *is the loss from the previous layer which has to be backpropagated to other layers*

Let's see how this happens for convolutions. Let's assume a convolution is carried out by

a filter **F** of size 2x2 and an input **X** of size 3x3. The output **O** the output will be of size 2x2. This is represented below :



$$O_{11} = X_{11}F_{11} + X_{12}F_{12} + X_{21}F_{21} + X_{22}F_{22}$$

$$O_{12} = X_{12}F_{11} + X_{13}F_{12} + X_{22}F_{21} + X_{23}F_{22}$$

So, during the backward pass $\frac{\partial L}{\partial O}$ is computed. Similarly as mentioned before $\frac{\partial L}{\partial X}$, this becomes the loss gradient of the previous layer, and $\frac{\partial L}{\partial F}$ are also computed.

$$F_{updated} = F - \alpha * \frac{\partial L}{\partial F} \tag{A.6}$$

(A.7)

This is how the filter gets As we are dealing with a matrix of inputs, the gradient for each element is to be computed. $\frac{\partial L}{\partial F}$ can be computed by using the chain rule ement wise, in the following manner :

$$\frac{\partial L}{\partial F_i} = \sum_{k=1}^{M} \frac{\partial L}{\partial O_k} * \frac{\partial O_k}{\partial F_i} \tag{A.8}$$

We can subsitute the value for the local gradients and obtain $\frac{\partial L}{\partial F}$ for each element. The local gradients can be obtained by

$$O_{11} = X_{11}F_{11} + X_{12}F_{12} + X_{21}F_{21} + X_{22}F_{22} \tag{A.9}$$

$$\frac{\partial O_1 1}{\partial X_1 1} = F_1 1 ...andsoon. \tag{A.10}$$

$$\tag{A.11}$$

Then we use this to find :

$$\frac{\partial L}{\partial X_i} = \sum_{k=1}^{M} \frac{\partial L}{\partial O_k} * \frac{\partial O_k}{\partial X_i} \tag{A.12}$$

A.1 Stochastic Gradient Descent

Let us begin the derivation of the learning algorithm by first looking at a loss function **J** for parameters θ

$$J(\theta) = \theta^2 \tag{A.13}$$

The objective of the learning algorithm is to find the value of the parameter that minimises the cost We can compute the different values of the function by changing theta and updtating it

based on the updation rule.α is the learning rate. To start,

$$\theta = 3 \tag{1.13}$$

$$\frac{\partial J(\theta)}{\theta} = 2\theta = 6 \tag{1.14}$$

$$\alpha\frac{\partial J(\theta)}{\theta} = 0.6 \tag{1.15}$$

$$\theta := \theta - \alpha\frac{\partial J(\theta)}{\theta} \tag{1.16}$$

$$\theta = 3 - 0.6 = 2.4 \tag{1.17}$$

The value of theta is updated as shown above until the minimum is reached i.e if on the next iteration, if the value of the function increases, we know we have reached the minimum. This gives the direction in which to move in. α is the learning rate, this tells us how fast to move. Gradient Descent for multiple variables uses the chain rule, quotient rule, sum rule and scalar multiple rule of calculus.

Function:

$$J(\theta_1, \theta_2) = {\theta_1}^2 + {\theta_2}^2$$

Objective:

$$\min_{\theta_1, \theta_2} J(\theta_1, \theta_2)$$

Update rules:

$$\theta_1 := \theta_1 - \alpha\frac{\partial}{\partial\theta_1}J(\theta_1, \theta_2)$$

$$\theta_2 := \theta_2 - \alpha\frac{\partial}{\partial\theta_2}J(\theta_1, \theta_2)$$

Derivatives:

$$\frac{\partial}{\partial\theta_1}J(\theta_1, \theta_2) = \frac{\partial}{\partial\theta_1}{\theta_1}^2 + \frac{\partial}{\partial\theta_1}{\theta_2}^2 = 2\theta_1$$

$$\frac{\partial}{\partial\theta_2}J(\theta_1, \theta_2) = \frac{\partial}{\partial\theta_2}{\theta_1}^2 + \frac{\partial}{\partial\theta_2}{\theta_2}^2 = 2\theta_2$$

For a function $h(x) = \theta_0 + \theta_1 x$ let us calculate the derivatives. The derivation is as follows.

We multiply the cost function $J(\theta)$ by half so that the 2 cancels out when we take the

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

$$\frac{\partial}{\partial\theta_0}J(\theta_0, \theta_1) = \frac{\partial}{\partial\theta_0}\left(\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2\right)$$

$$= \frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial\theta_0}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

$$= \frac{1}{m}\sum_{i=1}^{m}2\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)\frac{\partial}{\partial\theta_0}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)$$

$$= \frac{2}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)$$

derivative. This can be summed up below :

Cost Function – "One Half Mean Squared Error":

$$J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

Objective:

$$\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$$

Update rules:

$$\theta_0 := \theta_0 - \alpha\frac{\partial}{\partial\theta_0}J(\theta_0, \theta_1)$$
$$\theta_1 := \theta_1 - \alpha\frac{\partial}{\partial\theta_1}J(\theta_0, \theta_1)$$

Derivatives:

$$\frac{\partial}{\partial\theta_0}J(\theta_0, \theta_1) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)$$

$$\frac{\partial}{\partial\theta_1}J(\theta_0, \theta_1) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)\cdot x^{(i)}$$

Here, we can leave out the 1/m term, but we want it show that the same learning rate

is applied to all batches used in training of size m. Hence α = 1/m in the derivative. if we were to slightly modify this, we would have the algorithm for stochastic gradient descent.If we were to repeatedly run through the training set and update the parameters every time an example was encountered according to the gradient error, we have the Stochastic Gradient Descent Algorithm:

Loop {

for i=1 to m, {

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)} \qquad \text{(for every } j\text{)}.$$

}

}

Chapter 2

Pytorch Implementation

We pick up the pytorch implementation after describing the data transformations in chapter 3. For making this implementation the github repositories of the studies mentioned in chapter 2 [Sha19] as well the pytorch documentation [Doc17] was referred to. Pytorch offers a tutorial on how to implement CNNs and train them from scratch. There is extensive documentation in the website about how to use the functions used in this study. We create a dataloader which facilitates loading of the images into our model in an iterative manner, then we create the loaders for training, test and validation set.

```
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),data_transforms[x])
                                          for x in ['train','test','val']}

#here, we make the dataloaders, these load the images into the model
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_size,
                                              shuffle=True, num_workers=4)
               for x in ['train', 'test','val']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'test','val']}
```

Figure 2.1

After that we specify the architecture of the model we are about to use, and specify the

```
#making the training, test and val sets and their dataloaders
train_set = datasets.ImageFolder(root="/content/drive/My Drive/40mag/train",transform=train_transforms)

train_loader = DataLoader(train_set,batch_size=batch_size,shuffle=True,num_workers=4)



test_set = datasets.ImageFolder(root="/content/drive/My Drive/40mag/test",transform=test_transforms)

test_loader = DataLoader(test_set,batch_size=batch_size,shuffle=False,num_workers=4)

val_set = datasets.ImageFolder(root="/content/drive/My Drive/40mag/val",transform=val_transforms)

val_loader = DataLoader(val_set,batch_size=batch_size,shuffle=True,num_workers=4)
```

Figure 2.2

kind of weights initialization. Here Kaiming initialization of weights was implemented. Then a function to train the model is written. An example of the implementation of the train function is also shown below, note that the parametrs can be changed to get better results. After training a function to test the model is written and the results such as the ROC curve is obtained. We can save the trained model to load it later for sanity checks ,deployment.

```
import torch
import torch.nn as nn
from torch.hub import load_state_dict_from_url


__all__ = ['ResNet', 'resnet18', 'resnet34', 'resnet50', 'resnet101',
           'resnet152', 'resnext50_32x4d', 'resnext101_32x8d',
           'wide_resnet50_2', 'wide_resnet101_2']


model_urls = {
    'resnet18': 'https://download.pytorch.org/models/resnet18-5c106cde.pth',
    'resnet34': 'https://download.pytorch.org/models/resnet34-333f7ec4.pth',
    'resnet50': 'https://download.pytorch.org/models/resnet50-19c8e357.pth',
    'resnet101': 'https://download.pytorch.org/models/resnet101-5d3b4d8f.pth',
    'resnet152': 'https://download.pytorch.org/models/resnet152-b121ed2d.pth',
    'resnext50_32x4d': 'https://download.pytorch.org/models/resnext50_32x4d-7cdf4587.pth',
    'resnext101_32x8d': 'https://download.pytorch.org/models/resnext101_32x8d-8ba56ff5.pth',
    'wide_resnet50_2': 'https://download.pytorch.org/models/wide_resnet50_2-95faca4d.pth',
    'wide_resnet101_2': 'https://download.pytorch.org/models/wide_resnet101_2-32ee1156.pth',
}
```

Figure 2.3

```python
def conv3x3(in_planes, out_planes, stride=1, groups=1, dilation=1):
    """3x3 convolution with padding"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=stride,
                     padding=dilation, groups=groups, bias=False, dilation=dilation)


def conv1x1(in_planes, out_planes, stride=1):
    """1x1 convolution"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=1, stride=stride, bias=False)


class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, inplanes, planes, stride=1, downsample=None, groups=1,
                 base_width=64, dilation=1, norm_layer=None):
        super(BasicBlock, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        if groups != 1 or base_width != 64:
            raise ValueError('BasicBlock only supports groups=1 and base_width=64')
        if dilation > 1:
            raise NotImplementedError("Dilation > 1 not supported in BasicBlock")
        # Both self.conv1 and self.downsample layers downsample the input when stride != 1
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride
```

Figure 2.4

```
    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)

        return out


class Bottleneck(nn.Module):
    # Bottleneck in torchvision places the stride for downsampling at 3x3 convolution(self.conv2)
    # while original implementation places the stride at the first 1x1 convolution(self.conv1)
    # according to "Deep residual learning for image recognition"https://arxiv.org/abs/1512.03385.
    # This variant is also known as ResNet V1.5 and improves accuracy according to
    # https://ngc.nvidia.com/catalog/model-scripts/nvidia:resnet_50_v1_5_for_pytorch.

    expansion = 4
```

Figure 2.5

```
def __init__(self, inplanes, planes, stride=1, downsample=None, groups=1,
             base_width=64, dilation=1, norm_layer=None):
    super(Bottleneck, self).__init__()
    if norm_layer is None:
        norm_layer = nn.BatchNorm2d
    width = int(planes * (base_width / 64.)) * groups
    # Both self.conv2 and self.downsample layers downsample the input when stride != 1
    self.conv1 = conv1x1(inplanes, width)
    self.bn1 = norm_layer(width)
    self.conv2 = conv3x3(width, width, stride, groups, dilation)
    self.bn2 = norm_layer(width)
    self.conv3 = conv1x1(width, planes * self.expansion)
    self.bn3 = norm_layer(planes * self.expansion)
    self.relu = nn.ReLU(inplace=True)
    self.downsample = downsample
    self.stride = stride

def forward(self, x):
    identity = x

    out = self.conv1(x)
    out = self.bn1(out)
    out = self.relu(out)

    out = self.conv2(out)
    out = self.bn2(out)
    out = self.relu(out)

    out = self.conv3(out)
    out = self.bn3(out)

    if self.downsample is not None:
        identity = self.downsample(x)

    out += identity
    out = self.relu(out)

    return out
```

Figure 2.6

```
class ResNet(nn.Module):

    def __init__(self, block, layers, num_classes=1000, zero_init_residual=False,
                 groups=1, width_per_group=64, replace_stride_with_dilation=None,
                 norm_layer=None):
        super(ResNet, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        self._norm_layer = norm_layer

        self.inplanes = 64
        self.dilation = 1
        if replace_stride_with_dilation is None:
            # each element in the tuple indicates if we should replace
            # the 2x2 stride with a dilated convolution instead
            replace_stride_with_dilation = [False, False, False]
        if len(replace_stride_with_dilation) != 3:
            raise ValueError("replace_stride_with_dilation should be None "
                             "or a 3-element tuple, got {}".format(replace_stride_with_dilation))
        self.groups = groups
        self.base_width = width_per_group
        self.conv1 = nn.Conv2d(3, self.inplanes, kernel_size=7, stride=2, padding=3,
                               bias=False)
        self.bn1 = norm_layer(self.inplanes)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2,
                                       dilate=replace_stride_with_dilation[0])
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2,
                                       dilate=replace_stride_with_dilation[1])
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2,
                                       dilate=replace_stride_with_dilation[2])
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * block.expansion, num_classes)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            elif isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
```

Figure 2.7

```
    if zero_init_residual:
        for m in self.modules():
            if isinstance(m, Bottleneck):
                nn.init.constant_(m.bn3.weight, 0)
            elif isinstance(m, BasicBlock):
                nn.init.constant_(m.bn2.weight, 0)

def _make_layer(self, block, planes, blocks, stride=1, dilate=False):
    norm_layer = self._norm_layer
    downsample = None
    previous_dilation = self.dilation
    if dilate:
        self.dilation *= stride
        stride = 1
    if stride != 1 or self.inplanes != planes * block.expansion:
        downsample = nn.Sequential(
            conv1x1(self.inplanes, planes * block.expansion, stride),
            norm_layer(planes * block.expansion),
        )

    layers = []
    layers.append(block(self.inplanes, planes, stride, downsample, self.groups,
                        self.base_width, previous_dilation, norm_layer))
    self.inplanes = planes * block.expansion
    for _ in range(1, blocks):
        layers.append(block(self.inplanes, planes, groups=self.groups,
                            base_width=self.base_width, dilation=self.dilation,
                            norm_layer=norm_layer))

    return nn.Sequential(*layers)
```

Figure 2.8

```
    def _forward_impl(self, x):
        # See note [TorchScript super()]
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x

    def forward(self, x):
        return self._forward_impl(x)


def _resnet(arch, block, layers, pretrained, progress, **kwargs):
    model = ResNet(block, layers, **kwargs)
    if pretrained:
        state_dict = load_state_dict_from_url(model_urls[arch],
                                              progress=progress)
        model.load_state_dict(state_dict)
    return model


def resnet18(pretrained=False, progress=True, **kwargs):
    r"""ResNet-18 model from
    `"Deep Residual Learning for Image Recognition" <https://arxiv.org/pdf/1512.03385.pdf>`_
    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to stderr
    """
    return _resnet('resnet18', BasicBlock, [2, 2, 2, 2], pretrained, progress,
                   **kwargs)
```

Figure 2.9

```
#here we load the model we wrote the code for previously
model = resnet50(pretrained=False)

for param in model.parameters():
    param.requires_grad = False  #this is for whether we want to freeze our parameters or not

#this prints the model architecture
    print(model)




from collections import OrderedDict


#here we describe the structure of our classifier

classifier = nn.Sequential(OrderedDict([
                          ('fc1', nn.Linear(2048, 512)),
                           ('relu', nn.ReLU()),
                           ('dropout1', nn.Dropout(p=0.05)),
                           ('fc2', nn.Linear(512, 2)),
                            # ('relu', nn.ReLU()),
                           ('output', nn.LogSoftmax(dim=1))
                           ]))

#loading our classifier
model.fc = classifier
num_classes =2
valloss = []
trainloss =[]
valacc= []
trainacc = []
patience = 30 #threshold for early stopping

early_stopping = EarlyStopping(patience=patience, verbose=True) # if 30 times conescutively the loss does not decrease, the model stops training
```

Figure 2.10

```python
#Function to train the model
def train_model(model, criterion, optimizer, scheduler, num_epochs,patience):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    for epoch in range(1, num_epochs+1):
        print('Epoch {}/{}'.format(epoch, num_epochs))
        print('-' * 10)


        for phase in ['train', 'val']:
            if phase == 'train':
                scheduler.step()
                model.train()  #training mode
            else:
                model.eval()   # eval mode

            running_loss = 0.0
            running_corrects = 0

            # loading data with dataloaders
            for inputs, labels in dataloaders[phase]:
                inputs, labels = inputs.to(device), labels.to(device)

                # initialize the parameter gradients
                optimizer.zero_grad()

                # forward prop

                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    loss = criterion(outputs, labels)
                    _, preds = torch.max(outputs, 1)

                    # backward prop
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # calculating accuracy, loss
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
```

Figure 2.11

```
        epoch_loss = running_loss / dataset_sizes[phase]
        epoch_acc = running_corrects.double() / dataset_sizes[phase]
        if phase == "train" :
            trainloss.append (epoch_loss) #storing each loss and accuracy of epoch in a list
            trainacc.append(epoch_acc)
        else :
            valloss.append (epoch_loss)
            valacc.append(epoch_acc)


        print('{} Loss: {:.4f} Acc: {:.4f}'.format(
            phase, epoch_loss, epoch_acc)) #printing each loss and accuracy for both phases

        # this stores the model weights for the best model
        if phase == 'val' and epoch_acc > best_acc:
            best_acc = epoch_acc
            en = epoch
            best_model_wts = copy.deepcopy(model.state_dict())

    early_stopping(valloss[epoch-1], model) #early stopping

    if early_stopping.early_stop:
          print("Early stopping")
          break
    model.load_state_dict(torch.load('checkpoint.pt')) #stores the checkpoint if early stopping is implemented


    print()

time_elapsed = time.time() - since
print('Training complete in {:.0f}m {:.0f}s'.format(
    time_elapsed // 60, time_elapsed % 60))
print('Best valid accuracy: {:4f}'.format(best_acc))
#print ("Best Model weights : ",best_model_wts)  #Optional code to print best model weights
print (trainloss)
print (valloss)
```

Figure 2.12

```
#as accuracy gets stored as a tensor, we need to convert it to numpy for plotting a graph
def tensortonumpy (acc) :
      x = acc
      nparray = []

      for i in range (en):
           a = x[i]
           a = a.cpu()
           b = a.numpy()
           c = float(b)
           nparray.append(c)
      return nparray
val_acc = tensortonumpy(valacc)
train_acc =tensortonumpy(trainacc)
print(train_acc)
print(val_acc)

model.load_state_dict(best_model_wts) #loads weights of the best model

#in this section we plot the graphs
def mean(li): return sum(li)/len(li)
plt.figure(figsize=(14, 4))
plt.xlabel('training batch')
plt.ylabel('loss')
plt.plot([mean(trainloss[i:i+num_epochs]) for i in range(len(trainloss))])

def mean(li): return sum(li)/len(li)
plt.figure(figsize=(14, 4))
plt.xlabel('validation batch')
plt.ylabel('loss')
plt.plot([mean(valloss[i:i+num_epochs]) for i in range(len(valloss))])

def mean(li): return sum(li)/len(li)
plt.figure(figsize=(14, 4))
plt.xlabel('training batch')
plt.ylabel('epoch')
plt.plot(train_acc)

def mean(li): return sum(li)/len(li)
plt.figure(figsize=(14, 4))
plt.xlabel('validation batch')
plt.ylabel('epoch')
plt.plot(val_acc)
```

Figure 2.13

```
num_epochs =300
if use_gpu:
    print ("Using GPU: "+ str(use_gpu))
    model = model.cuda()

# NLLLoss because our output is LogSoftmax
criterion = nn.NLLLoss()

# Adam optimizer with a learning rate
#optimizer = optim.Adam(model.fc.parameters(), lr=0.0001,weight_decay = 0.001)
optimizer = optim.SGD(model.fc.parameters(), lr = .0001, momentum=0.09, weight_decay = 0.01)
# Decay LR by a factor--  every -- epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=50, gamma=0.1)


model_ft = train_model(model, criterion, optimizer, exp_lr_scheduler, num_epochs,patience)
torch.save(model_ft.state_dict(), "resnet50trial.pth")
```

Figure 2.14

```
[20] # Do validation on the test set
     def test(model, dataloaders, device):
       model.eval()
       actuals = []
       probabilities = []
       predictions = []
       accuracy = 0

       model.to(device)

       for images, labels in dataloaders['test']:
         images = Variable(images)
         labels = Variable(labels)
         images, labels = images.to(device), labels.to(device)

         output = model.forward(images)
         ps = torch.exp(output)
         equality = (labels.data == ps.max(1)[1])
         accuracy += equality.type_as(torch.FloatTensor()).mean()
         ##




         with torch.no_grad():
             for data, target in test_loader:
                 data, target = data.to(device), target.to(device)
                 output = model(data)
                 out    = output.cpu()
                 prediction = out.argmax(dim=1, keepdim=True)
                 actuals.extend(target.view_as(prediction) == 1)
                 probabilities.extend(np.exp((out[:, 1])))

         return [i.item() for i in actuals], [i.item() for i in probabilities]

         print("Testing Accuracy: {:.3f}".format(accuracy/len(dataloaders['test'])))

     actuals, class_probabilities = test(model, dataloaders,device)
```

Figure 2.15

```
actuals, class_probabilities = test(model, dataloaders,device)

fpr, tpr, _ = sklearn.metrics.roc_curve(actuals, class_probabilities)
roc_auc = auc(fpr, tpr)
plt.figure()
lw = 2
plt.plot(fpr, tpr, color='darkorange',
         lw=lw, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC for digit=1 class Malignant')
plt.legend(loc="lower right")
plt.show()

def test_label_predictions(model, device, test_loader):
    model.eval()
    actuals = []
    predictions = []
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            prediction = output.argmax(dim=1, keepdim=True)
            actuals.extend(target.view_as(prediction))
            predictions.extend(prediction)
    return [i.item() for i in actuals], [i.item() for i in predictions]

actuals, predictions = test_label_predictions(model, device, test_loader)
print('Confusion matrix:')
print(confusion_matrix(actuals, predictions))
print('F1 score: %f' % f1_score(actuals, predictions, average='micro'))
print('Accuracy score: %f' % accuracy_score(actuals, predictions))

#test(model, dataloaders, device)
```

Figure 2.16

```
checkpoint = {'input_size': [3, 224, 224],
                  'batch_size': dataloaders['train'].batch_size,
                   'output_size': 2,
                   'state_dict': model.state_dict(),
                   'data_transforms': data_transforms,
                   'optimizer_dict':optimizer.state_dict(),
                   'class_to_idx': model.class_to_idx,
                   'epoch': model.epochs}
torch.save(checkpoint, 'checkpoint.pth')
```

Figure 2.17

This page was intentionally left blank.

Bibliography

[Ike+04] Debra M. Ikeda et al. “Computer-aided Detection Output on 172 Subtle Findings on Normal Mammograms Previously Obtained in Women with Breast Cancer Detected at Follow-Up Screening Mammography”. In: *Radiology* 230.3 (2004). PMID: 14764891, pp. 811–819. DOI: 10.1148/radiol.2303030254. eprint: https://doi.org/10.1148/radiol.2303030254. URL: https://doi.org/10.1148/radiol.2303030254.

[WFH11] I.H. Witten, E. Frank, and M.A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2011. ISBN: 9780080890364. URL: https://books.google.co.in/books?id=bDtLM8CODsQC.

[He+15] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[Ben+17] Benzheng Wei et al. “Deep learning model based breast cancer histopathological image classification”. In: *2017 IEEE 2nd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*. 2017, pp. 348–353.

[Doc17] Pytorch Documentation. *Pytorch Documentation*. Web resource. The code was built used on the documentation available from here. 2017. URL: Pytorch.org.

[LTH17] Lisa Licitra, Annalisa Trama, and Hykel Hosni. “Benefits and Risks of Machine Learning Decision Support Systems”. In: *JAMA* 318.23 (Dec. 2017), pp. 2354–2354. ISSN: 0098-7484. DOI: 10.1001/jama.2017.16627. eprint: https://jamanetwork.com/journals/jama/articlepdf/2666496/jama_licitra_2017_le_170143.pdf. URL: https://doi.org/10.1001/jama.2017.16627.

[Agg18] C.C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Springer International Publishing, 2018. ISBN: 9783319944630. URL: https://books.google.co.in/books?id=achqDwAAQBAJ.

[BFR18] Nadia Brancati, Maria Frucci, and Daniel Riccio. "Multi-classification of Breast Cancer Histology Images by Using a Fine-Tuning Strategy". In: *Image Analysis and Recognition*. Ed. by Aurélio Campilho, Fakhri Karray, and Bart ter Haar Romeny. Cham: Springer International Publishing, 2018, pp. 771–778. ISBN: 978-3-319-93000-8.

[Fer+18] Carlos A. Ferreira et al. "Classification of Breast Cancer Histology Images Through Transfer Learning Using a Pre-trained Inception Resnet V2". In: *Image Analysis and Recognition*. Ed. by Aurélio Campilho, Fakhri Karray, and Bart ter Haar Romeny. Cham: Springer International Publishing, 2018, pp. 763–770. ISBN: 978-3-319-93000-8.

[Mot+18] Mehdi Habibzadeh Motlagh et al. "Breast Cancer Histopathological Image Classification: A Deep Learning Approach". In: *bioRxiv* (2018). DOI: 10.1101/242818. eprint: https://www.biorxiv.org/content/early/2018/01/04/242818.full.pdf. URL: https://www.biorxiv.org/content/early/2018/01/04/242818.

[NMK18] Abdullah Nahid, Ali Mehrabi, and Yinan Kong. "Histopathological Breast Cancer Image Classification by Deep Neural Network Techniques Guided by Local Clustering". In: *BioMed Research International* 2018 (Mar. 2018), pp. 1–20. DOI: 10.1155/2018/2362108.

[Rak+18] Alexander Rakhlin et al. "Deep Convolutional Neural Networks for Breast Cancer Histology Image Analysis". In: *Image Analysis and Recognition*. Ed. by Aurélio Campilho, Fakhri Karray, and Bart ter Haar Romeny. Cham: Springer International Publishing, 2018, pp. 737–744. ISBN: 978-3-319-93000-8.

[FRD19] Li Fei-Fei, Krishna Ranjay, and Xu Danfei. *Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition.* Web resource. 2019. URL: http://cs231n.stanford.edu/.

[Jia+19] Yun Jiang et al. “Breast cancer histopathological image classification using convolutional neural networks with small SE-ResNet module”. In: *PLOS ONE* 14.3 (Mar. 2019), pp. 1–21. DOI: 10.1371/journal.pone.0214587. URL: https://doi.org/10.1371/journal.pone.0214587.

[Pyo19] Skalski Pyotr. *Gentle Dive into Math Behind Convolutional Neural Networks*. Web Artcicle. 2019. URL: https://towardsdatascience.com/gentle-dive-into-math-behind-convolutional-neural-networks-79a07dd44cf9.

[Rod+19a] Alejandro Rodriguez-Ruiz et al. “Stand-Alone Artificial Intelligence for Breast Cancer Detection in Mammography: Comparison With 101 Radiologists”. In: *JNCI: Journal of the National Cancer Institute* 111.9 (Mar. 2019), pp. 916–922. ISSN: 0027-8874. DOI: 10.1093/jnci/djy222. eprint: https://academic.oup.com/jnci/article-pdf/111/9/916/31963401/djy222.pdf. URL: https://doi.org/10.1093/jnci/djy222.

[Rod+19b] Alejandro Rodríguez-Ruiz et al. “Detection of Breast Cancer with Mammography: Effect of an Artificial Intelligence Support System”. In: *Radiology* 290.2 (2019). PMID: 30457482, pp. 305–314. DOI: 10.1148/radiol.2018181371. eprint: https://doi.org/10.1148/radiol.2018181371. URL: https://doi.org/10.1148/radiol.2018181371.

[Sha19] Pulkit Sharma. *Build an Image Classification Model using Convolutional Neural Networks in PyTorch*. Web Artcicle. 2019. URL: https://www.analyticsvidhya.com/blog/2019/10/building-image-classification-models-cnn-pytorch/.

[Ska19] Pyotr Skalski. *Deep Dive into Math Behind Deep Networks*. Web Artcicle. 2019. URL: https://towardsdatascience.com/https-medium-com-piotr-skalski92-deep-dive-into-deep-networks-math-17660bc376ba.

[Wan+19] Yunjun Wang et al. “Using deep convolutional neural networks for multi-classification of thyroid tumor by histopathology: a large-scale pilot study”. In: *Annals of Translational Medicine* 7.18 (2019). ISSN: 2305-5847. URL: http://atm.amegroups.com/article/view/28771.

[Yal+19a] Adam Yala et al. “A Deep Learning Mammography-based Model for Improved Breast Cancer Risk Prediction”. In: *Radiology* 292.1 (2019). PMID: 31063083, pp. 60–66. DOI: 10.1148/radiol.2019182716. eprint: https://doi.org/10.1148/radiol.2019182716. URL: https://doi.org/10.1148/radiol.2019182716.

[Yal+19b] Adam Yala et al. “A Deep Learning Model to Triage Screening Mammograms: A Simulation Study”. In: *Radiology* 293.1 (2019). PMID: 31385754, pp. 38–46. DOI: 10.1148/radiol.2019182908. eprint: https://doi.org/10.1148/radiol.2019182908. URL: https://doi.org/10.1148/radiol.2019182908.

[Zhu+19] Chuang Zhu et al. “ Breast cancer histopathology image classification through assembling multiple compact CNNs”. In: *BMC Medical Informatics and Decision Making* 19.198 (Oct. 2019), pp. 916–922. ISSN: 1472-6947. DOI: 10.1186/s12911-019-0913-x. URL: https://doi.org/10.1186/s12911-019-0913-x.

[Zho+20] Xiaomin Zhou et al. *A Comprehensive Review for Breast Histopathology Image Analysis Using Classical and Deep Neural Networks*. 2020. arXiv: 2003.12255 [eess.IV].

Acknowledgments

I wish to record a deep sense of gratitude to **Prof. Rakesh Nigam**, my supervisor for his valuable guidance and constant support at all stages of my study. I wish to thank my parents for their constant support, and my friends and classmates who though miles apart helped a lot. None of this would be possible without the almighty, I thank god for giving me faith and strength to move forward in the times I nedded it most.