Using Lightweight Neural Networks to Help Detect Cancerous Skin Lesion

Velian Velikov

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Bachelor of Science
of the
University of Aberdeen.

Department of Computing Science

May 30th, 2023

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

Date: May 30th, 2023

Abstract

TODO: An expansion of the title and contraction of the thesis.

Personal Motivation and Acknowledgements

I would like to briefly articulate my personal motivation for pursuing an unfamiliar topic for my dissertation project. This choice allowed me to embark on an exploration into a niche that had not been addressed within the broader confines of my university syllabus.

My previous experiences have spanned across client-server architectures, APIs, databases, and GUIs in both university and non-university team projects. However, I hadn't had the chance to delve extensively into image analysis, study and train image classification models, explore different backbones, fine-tune them, conduct image inferencing, and test various configurations. Therefore, selecting this particular topic was perfectly in line with my personal objectives.

Having immersed myself in this dissertation project, I am now confident in my potential contributions in new areas to a startup that I am co-developing with fellow university students. What is more, this venture has fortified my confidence when discussing the subject matter, and it has ignited my interest to further research the field in the future.

My inclination to choose challenging paths in academia and life has consistently guided my decisions. I maintain the belief that confronting obstacles and embracing the unknown often leads us towards sound decisions. This project was indeed a formidable challenge, but it is also true that the invaluable experience, knowledge and skills acquired through this endeavour will significantly contribute to my future growth and development.

I would like to express my heartfelt gratitude to my mentor, Dr Dewei Yi, for his unweaving support throughout the duration of this project. In light of the circumstance, it was Dr Yi's optimism and attitude that has attributed positively to my entire learning journey and transformed it into a highly enjoyable experience. Every enlightening discussion we had played a vital role in enhancing my understanding of the machine learning domain. Whenever I have found myself in a deadlock situation, his quick wit enabled me to refocus my attention towards other facets of the problem at hand, thus ensuring minimal loss of time and keeping a straight path to the end goal. I am proudly grateful for the dedication, patience and weekly meetings that Dr Dewai Yi has invested in this project. I am also thankful for his meticulous efforts in reviewing, judging and evaluating the contents of this thesis.

Furthermore, I would like to express my sincere thanks to my girlfriend and family for being always there. Their uplifting spirits have been a beacon of positivity and their unconditional support has made the realization of this dissertation project less daunting. Without them, the journey would have been even more challenging.

Contents

1 Introduction **7**

1.1 Overview . . . 7

1.2 What is Melanoma Skin Cancer . . . 7

1.3 ISIC Challenges . . . 7

1.4 Motivation . . . 8

1.5 Goals . . . 8

2 Background and Related Work **9**

2.1 Undestanding Image Classification and Visual Basics . . . 9

2.1.1 Multilayer Perceptron . . . 11

2.1.2 Backpropagation Algorithm . . . 12

2.1.3 Universal Approximation Theorem . . . 13

2.2 Convolutional Neural Networks . . . 14

2.2.1 Convolutional Kernel . . . 14

2.2.2 Convolutional Layer . . . 15

2.2.3 Activation Layer . . . 17

2.2.4 Pooling Layer . . . 17

2.2.5 Normalisation Layer . . . 18

2.2.6 Fully Connected Layer . . . 18

2.2.7 Probabilistic Output Layer . . . 19

2.2.8 Convolution Operations . . . 20

2.3 Evolution of Image Classification Model Design . . . 21

2.3.1 LeNet-5 (1998) . . . 21

2.3.2 AlexNet (2012) . . . 22

2.3.3 VGGNet (2014) . . . 23

2.3.4 GoogLeNet (Inception v1) (2014) . . . 24

2.3.5 ResNet (2016) . . . 25

2.4 Efficient Neural Networks . . . 27

2.4.1 ResNet (2016) . . . 28

2.4.2 MobileNets (2017~2019) . . . 28

2.4.3 ShuffleNets (2018) . . . 32

2.4.4 MobileOne (2022) . . . 34

2.5 Training Image Classification Models . . . 34

3 Methodology **35**
3.1 Medical Image Datasets of Skin Lesions 35
3.1.1 ISIC 2018 35
3.1.2 ISIC 2020 35
3.1.3 Dataset preprocessing into supported folder structure 36
3.2 OpenMMLab: A Unified Benchmark Framework 36
3.2.1 PyTorch 38
3.2.2 MMPreTrain (formerly MMClassification) 38
3.3 Setting up the environment 40
3.4 Applying Transfer Learning Methodology to Efficient Neural Networks 41

4 Experimental Design and Evaluation **43**
4.1 Metrics 43
4.2 Results 43
4.2.1 Result for network 1 43
4.2.2 Result for network 2 43
4.2.3 Result for network 3 43
4.2.4 Result for network 4 43
4.2.5 Result for network 5 43
4.3 Evaluation review 43

5 Practical Deployment of Web-based Application **44**
5.1 Design Stage 44
5.2 Client-Server Architecture 45
5.3 Final Design and Analysing User Flow 45

A Experimental Design and Evaluation **51**
A.1 Pre-requisites for the Ract Application 51
A.2 Installation Steps 51
A.3 Note 51
A.4 Install Flask 51
A.5 Run Flask 51

Chapter 1

Introduction

1.1 Overview

One in five people will receive a skin cancer diagnosis in the course of their lifetime (Skin Cancer Foundation, 2019). With this project, our mission is to create a world where there are no skin cancer deaths - and I believe that we can do it by empowering patients with information about their health through their AI-powered mobile devices. This dissertation project focuses on the creation and analysis of lightweight AI-powered skin cancer risk assessment algorithms.

1.2 What is Melanoma Skin Cancer

Melanoma is the most lethal skin cancer, and its incidence has increased significantly over the past decade. Melanoma is a pigmented lesion occurring on the surface of the skin and can be detected by an expert visually. It has been proven that it can also be detectable by image analysis software (MIT News, 2021) (Soenksen, 2020). This type of cancer can be treated when it is diagnosed in its earliest stages.

1.3 ISIC Challenges

The International Skin Imaging Collaboration (ISIC) is a partnership between academia and industry with the aim of centralising, coordinating and improving the application of digital skin imaging in order to reduce the mortality cases of melanoma skin cancer through the development and promotion of standards for dermatologic imaging and engaging dermatology and computing science communities in the development and improvement of automated diagnosis systems. The archive of digital images is composed of skin lesions that can be used by educated professionals and the public for the development of automated diagnosis systems. ISIC provides public access to an open-source archive of skin images which will be used in the research, development and testing of our intelligent systems. To date, the ISIC challenges have inspired numerous publications in the medical and scientific literature (ISIC, 2023), which is a sign of growing interest and increased development in this field.

ISIC hosts challenges which engage the computer science community with the aim of improving dermatologic diagnosis with the help of AI. This dissertation project uses high-quality human-validated training and testing dataset of thousands of images and respective metadata published under the Creative Commons Zero license. While the initial focus of the challenges was solely on melanoma diagnosis, now it is also critical to pursue the advanced imaging algorithms able to diagnose a broader landscape of skin developments, including non-melanoma skin cancer

and inflammatory dermatoses.

1.4 Motivation

Early detection of skin cancer is essential to treatment success. According to (Anthony and Bifulco, 2000), if melanoma is diagnosed early, while it is still limited to the outer layers of the skin, a simple surgical removal is generally curative with a 5-year survival of approximately 98% of patients. Early-stage melanomas can be difficult to distinguish from benign skin lesions, even by experienced doctors, and despite a growing number of skin biopsies, many melanoma cases are still being missed (Sun, 2023).

I believe that everyone should have access to the best care possible, but many people don't have equal access to healthcare resources and cannot receive adequate dermatologic care. Medical image and data diagnosis are vital for guiding clinical treatment plans. Ideally, dermatologists should routinely assign full-body photography which requires expensive dermoscopy tools to diagnose melanomas, track changes in moles over time and avoid assigning biopsies of stable lesions. In some cases, dermatologists rely only on their dermatoscopes, which are handheld magnifying devices that allow them to see easier different characteristics of melanomas on the skin. The wider adoption of technologies in clinical practice has been slow due to cost and inconvenience (Miller et al., 2021). However, this is quickly beginning to change as nowadays telemedicine is more widely used by GPs (Koelink et al., 2022), quality smartphone digital cameras are getting cheaper and digital dermatoscopes are becoming more affordable and entering the professional market.

If one of my proposed algorithms detects a lesion that is deemed cancerogenic, the doctor can refer its patient to a dermatologist, who would professionally confirm the suspicion and the user will be treated. Doctors or patients should be able to perform skin disease risk assessments with their phones or on a computer with limited computational power.

1.5 Goals

The goal of this project is to use state-of-the-art machine learning algorithms to provide reliable risk assessment for melanoma skin cancer and to develop an app that allows users to save skin assessments and track skin lesions over time. Users would never miss a scan thanks to automatic scan reminders on when to perform the next assessment. At its peak, this application will use the latest technology and be constantly updated to provide its users with the most accurate assessment.

With the rapid of Convolutional Neural Networks (CNN), models obtained expert-level performance on medical images and data diagnosis, but they usually involve high-computational costs and require expensive computer hardware (Sharma et al., 2021). With recent developments, skin cancer risk assessment can be made more accessible and affordable by utilizing lightweight neural networks (Ukwandu, 2022). What is more, with the utilisation of efficient neural networks, scans can be processed and stored locally for maximum protection and privacy. The main objectives of this dissertation project are to acquire substantial knowledge about the related field. Ideally, after the literature review, we will work on finding, optimising and evaluating high-efficient neural network structures with the objective of satisfying the requirement of real-time mobile applications.

Chapter 2

Background and Related Work

2.1 Undestanding Image Classification and Visual Basics

This chapter contains information on how the image classification model class is designed and how it is trained. Later, we will cover why the MMPreTrain (formerly MMClassification) toolkit is the framework of choice for this project, but before that, there are some basic knowledge prerequisites that must be covered first.

Image classification is an image processing method that distinguishes different types of targets based on the different characteristics reflected in the image information. It uses computers to quantitatively analyze images, classifying each pixel or region in the image into one of several categories, substituting human visual judgement, in our case, the expert knowledge of the general partitioning doctor. Simply put, it is the identification of whether an image is of a certain class. This process is suitable for scenarios where the subject in the image is relatively singular, such as identifying whether the lesion spot on an image is nevus or melanoma.

As shown in Figure 2.1, every image is an array of pixels: $X \in \mathbb{R}^{H \times W \times 3}$. The problem of image classification lies in assigning correctly the category of an input image, where the category of the image belongs to one of the predetermined class members. For example, the number of categories could be: nevus = 0, melanoma = 1, scar = 2, etc. Mathematically $y \in \{0, \ldots, K\}$, where K is the total number of categories. Although this task seems simple, it is a fundamental issue in computer vision and has a wide range of practical applications, even beyond the field of medical image analysis.

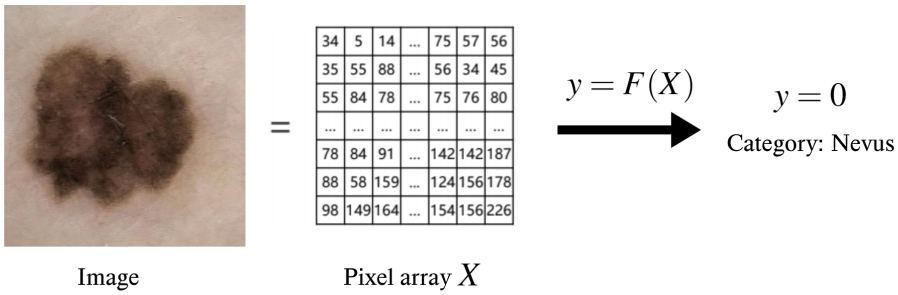


Figure 2.1: A simple overview of an image classification task.

Classification is a form of supervised learning. Our goal is to learn a function that can help us complete the task of classifying data. There are two types of predictions: one is directly outputting the predicted class, telling you which class it is and the other is predicting the probability of the

data belonging to each of the possible classes, without directly telling the corresponding class.

As an example, our aim is the task of classifying skin lesions (See Figure 2.2). If we have a 28x28 image, we can actually treat it as a 784-dimensional vector. Then we feed this data into the function. The function then produces a prediction, which is a number between 0 and 2, representing the class of the input image. Alternatively, the function can output a three-dimensional vector, where each dimension represents the probability of the input image belonging to the corresponding class.

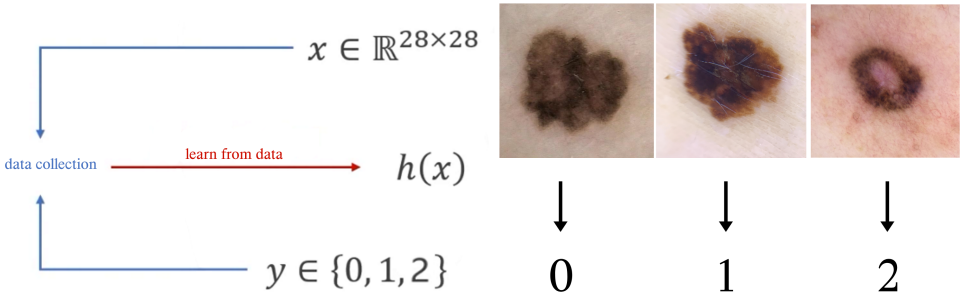


Figure 2.2: $P(y|x) = h(x)$ is the model prediction that the data belongs to category y given x. $h(x)$ is also known as the classifier.

To accomplish this task, we need to collect a dataset with images of different lesions and have them annotated. The concept of learning from data refers to the use of this dataset in the training of the classifier.

But how does the classifier learn? It can be divided into two steps. Firstly, we construct a family of classifier functions, denoted as H, which includes all the classifiers we are interested in. Every possible classifier that can accomplish this classification task is put into this parameterized function family $H = \{h(x;\theta) \mid \theta \in \Theta\}$ where for each θ, $h(x;\theta)$ is a classifier. This function family, H, could be a linear classifier or a convolutional neural network structure. For each parameter, our $h(x;\theta)$ is a different classifier that can classify all the classes obtained by scanning the parameter space. By putting all these classifiers into the function family, we form our function family.

Now, in the second step, we have this function family that contains many classifiers, and we need to find the best classifier from within. There are two problems in this search process. First, how do we define and determine what a "good" classifier is? The criteria for a "good" classifier in classification problems is relatively simple; we measure it by the error rate. The lower the error rate, the better the classifier. More generally, the error rate can be abstracted as a loss function. Thus, for a pile of classification problems, we can approximate the loss function as the error rate.

Second, once we have defined what is considered "good", how do we find "the best" one? We know that within this parameterized classifier family, H, there are many classifiers. By adjusting the parameter θ, we can obtain different classifiers. Different classifiers will have different error rates. We just need to adjust θ to minimize this error rate. In practice, this is finding the minimum value of a function. Because the error rate is essentially a function of θ, in mathematics, this problem is referred to as an optimization problem.

In this dissertation project, the function family is a series of convolutional neural networks.

To measure the quality of a classification model we commonly use a loss function called the cross entropy, while for searching for the best model in H, the algorithm most commonly used is the stochastic gradient descent. Training neural networks is quite a difficult task, so simply using gradient descent is not enough, so we also need to apply our own experience and utilize various strategies.

The gradient descent is a very general algorithm for solving optimization problems and finding the minimum value of the function of our parameters L. We can plot this function in our space, and this results in a three-dimensional surface. This surface has some valleys, and our goal is to find these blue areas of the loss function surface $L(\theta)$.

The first step is to randomly select a starting point $\theta^{(0)}$, which can be any point on our surface. Secondly, we find the direction of the fastest decline, that is in the negative gradient direction $\Delta_\theta L(\theta^{(0)})$. To make the loss function a bit smaller, we move forward in this direction at a rate of $\theta^{(1)} = \theta^{(0)} - \eta\Delta_\theta L(\theta^{(0)})$ where η is the step size (our learning rate). We repeat this until we reach the lowest point in one of these valleys. For a convex function using this approach, it is guaranteed that will find the global minimum value of the function, but for a non-convex function which has multiple valleys, it is not guaranteed that this method will find the minimum value point, as this outcome highly depends on to the initial position, the step size and the direction of each step.

TODO Figure: A figure of the valley. TODO: Reference figure in text.

2.1.1 Multilayer Perceptron

The most fundamental unit of a neural network is called a neuron. As shown in Figure 2.3, firstly, we input a data vector into a neuron. It is a d-dimensional vector with d components from x_1 to x_d. Then, we find a series of parameters that can be learned for each component of the input vector. We apply weights to each component and sum them up to obtain an intermediate variable, z. Then, we pass z through a nonlinear function. Typically, it is a curve-shaped function $\sigma(z)$ that represents the expression shown on the left-hand side of the figure. This function creates a curve, and we obtain the final output y. This constitutes a neural unit. In fact, a neural unit performs the work of a linear classifier because we perform weighted summation. The activation function actually splits z into positive and negative classes and produces the output of two different classes or, strictly speaking, probabilities. The output, denoted as y, is a value in the interval between 0 and 1.

So, a single neuron can only perform linear classification tasks and if we connect multiple neurons together, we can perform the task of non-linear classification. When we combine neurons in layers we create the simplest neural network model, called a multilayer perceptron. First, within a layer, we compute multiple outputs, such as h_1 and h_2. We perform many computations and obtaining the outputs, we merge them and treat them as inputs for the next layer, where we calculate the outputs again. On the right-hand side of the figure we have two layers, but in reality, we can have many layers. Finally, after configuring multiple layers, we obtain the final output, y.

In this process, we know the input and output, but the intermediate variables within the network, which we cannot see, are also called hidden layers. This multi-hidden-layer structure can effectively implement nonlinear classifiers where each layer contains many parameters. Each line on the graph represents a learnable parameter and we are used to representing them in matrix form.

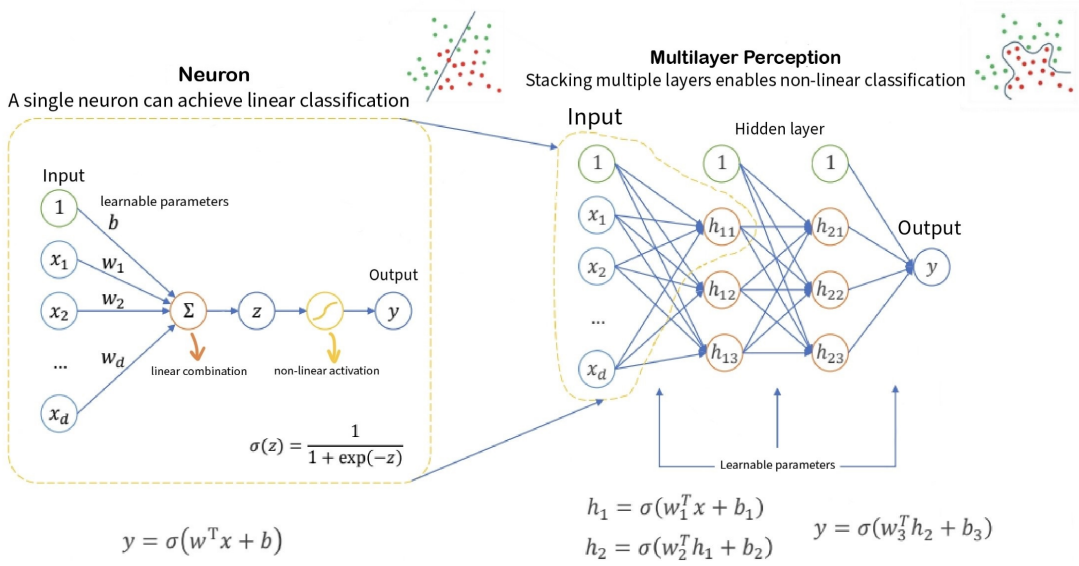


Figure 2.3: Visual representation of a neuron and a multilayer perceptron.

In summary, there are many parameters, and one of our tasks is to learn these parameters.

2.1.2 Backpropagation Algorithm

To calculate the gradient of the loss function with respect to all hidden layer parameters of a multilayer perceptron we use the algorithm called backpropagation.

On the top of Figure 2.4, we have written out the expressions for a multilayer perceptron, which describe how to calculate the output from the input. We can see the structure of certain functions, so h_1 is a function of x, h_2 is a function of h_1, which is a composition, and then y is a function of h_2, which is another composition. We build up this composition of multiple layers of functions, and in the end, we calculate the loss function l. which is also another layer of composition, where $\hat{y}$ is our estimated value, y is the truth value and l is the squared difference between y and $\hat{y}$.

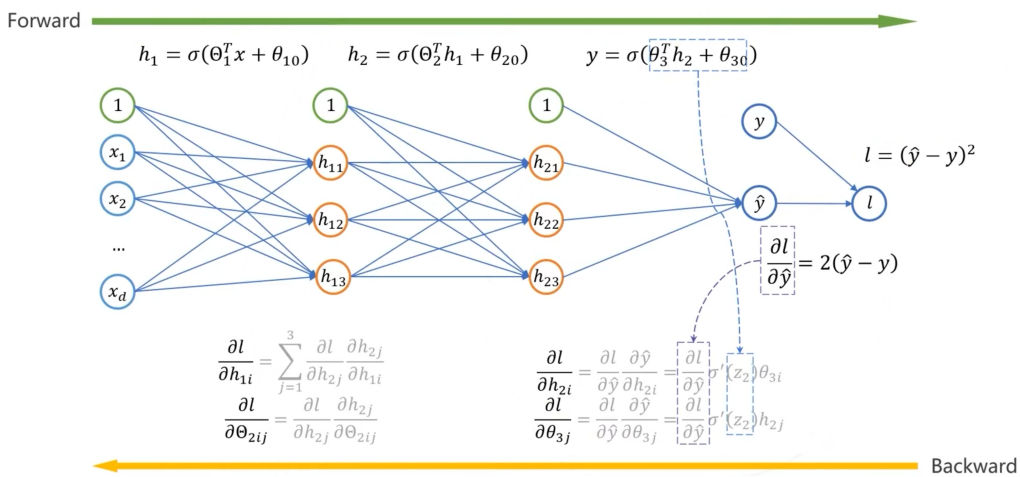


Figure 2.4: Visual representation of the forward and backward steps of the backpropagation algorithm.

For composite functions, calculating the gradient is not difficult in mathematics. We have

something called the chain rule, which can be used to calculate the derivative. We calculate some derivatives and we observe a phenomenon, which is that when we calculate one of the gradients, we also need to know in advance some other gradients, or some other intermediate variables. For example, we want to calculate the partial derivative of the loss function with respect to the parameters of the third layer. We need to know beforehand the partial derivative of the loss function with respect to the output $\hat{y}$ of the neural network, as well as some intermediate variables in the process of computing $\hat{y}$, such as the z_2. Similarly, when we calculate the derivative of the loss function with respect to a parameter in the second layer, we also need to know beforehand the partial derivative of the loss function with respect to the output of the second layer, as well as the partial derivative with respect to h_2. When calculating the hidden output of the second layer, some intermediate variables change, so this means that when we calculate this series of derivatives, we are applying the chain rule and we follow an order of computations. Combining this computation in this order is a key part of the backpropagation algorithm.

The first step is called the forward step, which is when the input is fed into the network. During the forward step, the input is passed through the network, and the activation values of the hidden layers are computed one layer at a time until the output is reached and the loss function is applied. The second step is called the backward step, also known as backpropagation, where we compute the gradients of the loss function with respect to the parameters of the network by propagating the error from the output back to the input. During the backward step, we compute the gradient of the loss function with respect to the parameters of the network by propagating the error from the output back to the input, and we need to calculate the influence of the activation values of the higher layers on the gradient of the lower layers, such as when calculating the derivative of the second layer, we need to consider the effect of the activation values of the third layer in order to compute the derivative of the second layer. Once we have computed the gradients and the intermediate variables for the second layer, we can continue the backward pass and compute the gradients and intermediate variables for the lower layers until we reach the input. The backpropagation algorithm is essentially a systematic and sequentially organised chain rule for derivatives of composite functions.

2.1.3 Universal Approximation Theorem

In the 1990s, the Universal Approximation Theorem was proven and states that even with only one hidden layer, a neural network with a sufficient number of neurons can approximate any continuous function, which is a very strong theorem. This theorem shows that the ability of neural networks can be arbitrarily strong, but it comes with a cost. Although a neural network with one hidden layer can approximate any continuous function, the number of neurons required grows exponentially with the dimensionality of the input, which is a significant cost. For example, if an image has a resolution of 100×100, which is a very low-resolution image, it still has 10,000 pixels. If we apply this theorem to an input with 10,000 dimensions, it tells us that we would need at least 2^{10000} neurons to accurately approximate any continuous function. It may not be feasible to implement this on any computer in the world.

From a practical point of view, networks with an exponential number of neurons are not feasible, therefore, we can see that our aim is to produce the most accurate approximation of reality. We definitely need to design models based on the characteristics of the image to achieve

higher efficiency in our image classification tasks.

2.2 Convolutional Neural Networks

First, it is important to answer the question of why convolutional neural networks are called "convolutional". In mathematics, as discussed by Song Ho Ahn (2018), a convolution is a function derived from two other functions by integration which expresses how one function is modified by the other. The convolution operation is the building block in CNNs and refers to the process of applying a set of learnable parameters (also called kernels) to an input image or feature map.

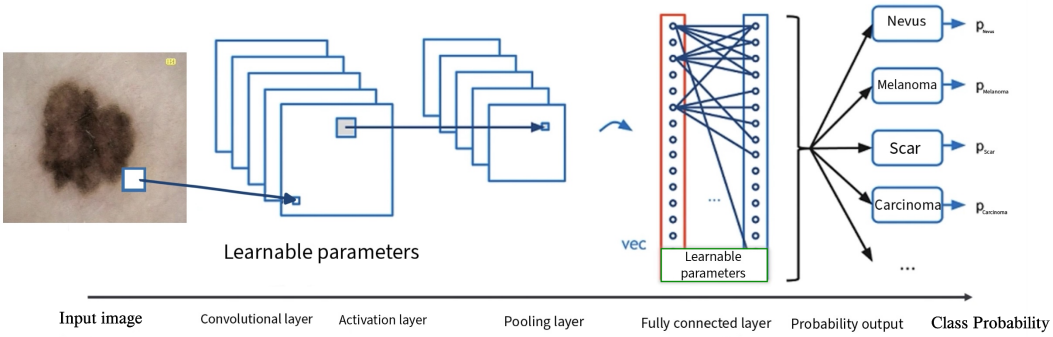


Figure 2.5: A high-level overview of the layers of a convolutional neural network.

Like the multilayer perceptron, the overall structure of the convolutional neural network is also a multi-layer structure, where each layer has some learnable parameters (See Figure 2.5). The difference between convolutional neural networks and multilayer perceptrons is that the former contains various types of layers, such as convolutional layers and activation layers, pooling layers, fully connected layers, and other various layers, while in the multilayer perceptron, there is basically only one type of layer, called the fully connected layer.

This section will serve as an introduction to the various layers and describe how they are composed to form a convolutional neural network.

2.2.1 Convolutional Kernel

We can apply convolution to image processing. For example, in Figure 2.6, we have a grayscale image with a nevus lesion on it then we use a convolutional kernel to convolve it. Here, we use a kernel with 8 in the centre and 1s surrounding it. We apply this convolution kernel to the image by laying it over the pixels of the image and computing the result. Then we slide this convolutional kernel on the image to obtain the output feature map. This feature map, as we can see, is actually different from the input image because we used this convolutional kernel with a value of 8 in the centre, which effectively performs edge detection. Therefore, we can observe that the edges between the lesions and the background in the image have been extracted. Because we are extracting the edge features of the object, we generally refer to this kind of output as a feature map.

Usually, the image that we process is a coloured image as in Figure 2.7, meaning it is a three-dimensional object with RGB channels. In order to extend the concept of convolution in three dimensions we can convert the convolution kernel into a three-channel kernel as well. In this case, when we apply the convolution kernel to the image, it becomes a cubic block. The weights of the two cubic blocks are multiplied element-wise for each corresponding pixel position, then we

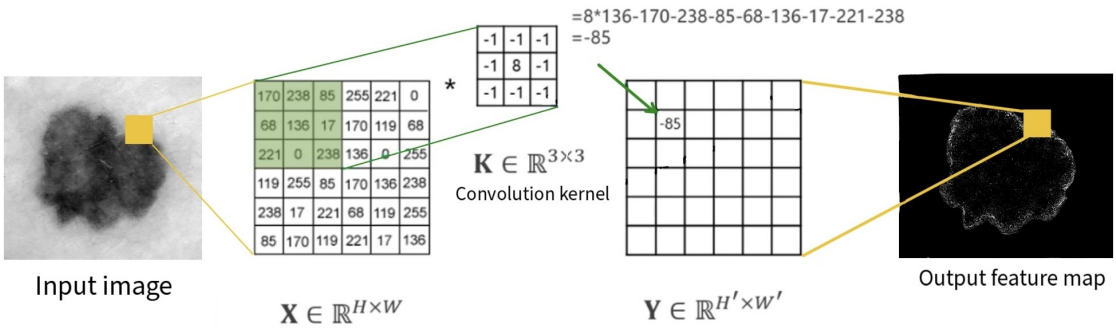


Figure 2.6: Convolution of a grayscale image of a nevus lesion.

sum them up in order to obtain a feature map. If we adjust our convolutional kernel to match the number of channels in this image, the results can be quite interesting, because no matter how many input channels the image has, after scanning with the convolutional kernel, we can always obtain an output feature map.

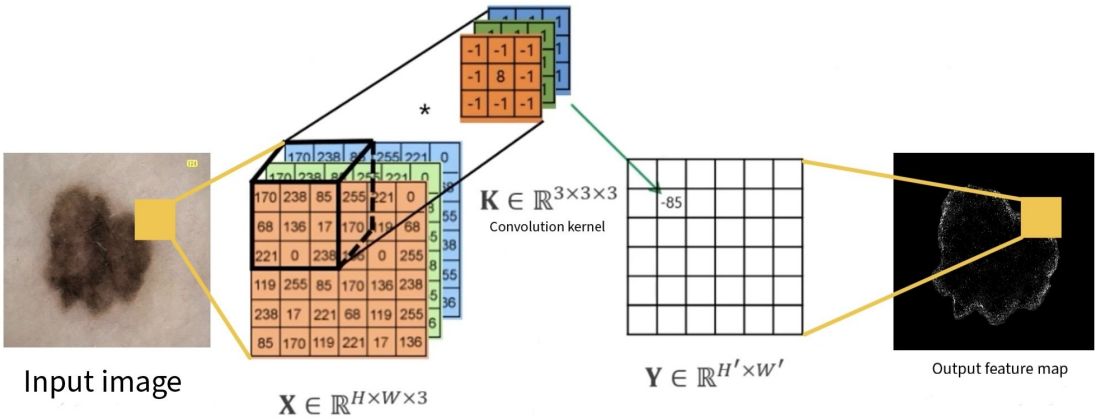


Figure 2.7: Convolution of an RGB image of a nevus lesion.

In the previous examples, we only used one convolutional kernel to obtain a feature map. We can modify the convolution operation by adding another convolutional kernel (Refer to Figure 2.8) and use it to scan the image separately and get another feature map. This means that if we want to create feature maps with multiple channels, we can simply employ multiple convolution kernels on the input image, leading to the output of multiple feature maps. In practice, this approach significantly improves the richness of the data we are able to capture from each image.

2.2.2 Convolutional Layer

The additional kernel when handling three-dimensional data, allows us to capture spatial information across multiple channels. The idea of the convolutional layer is that it can process multiple channels of input and output feature maps. According to Datahacker (2018), in Figure 2.9 we can see that we have the three-channel inputs of the small 6×6 RGB image. Then we use two $3 \times 3 \times 3$ convolution kernels – we have three-dimensional convolution kernels for the three channels. After that, we add a bias to this feature map, which is equivalent to adding a certain value to each element of the map, and as a final output, we get a feature map with two channels.

The convolutional layer transforms the feature map $\mathbf{X} \in \mathbb{R}^{H\times W\times C}$ of the C channel into the

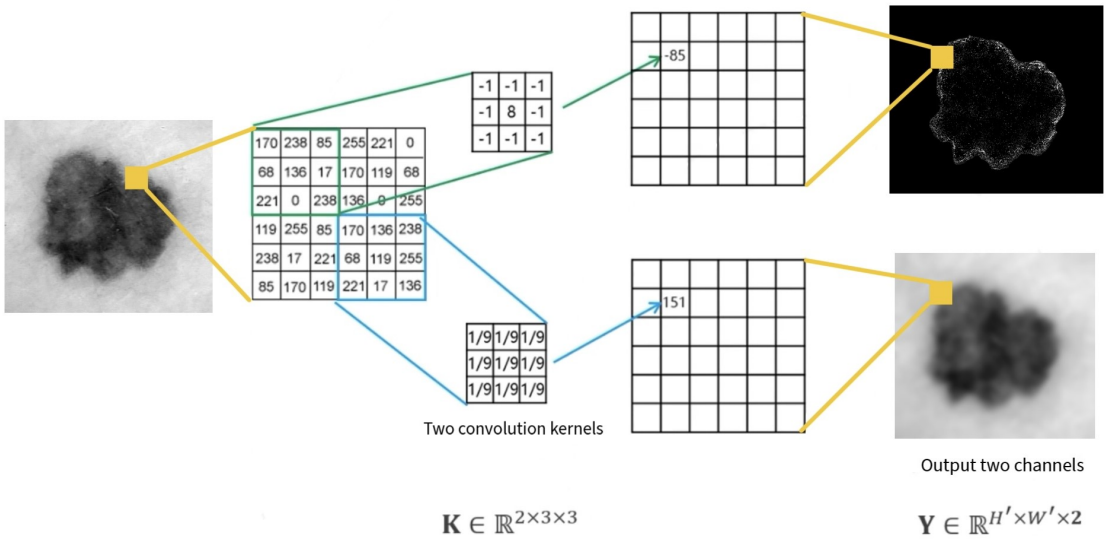


Figure 2.8: Convolution of a grayscale image using multiple convolution kernels.

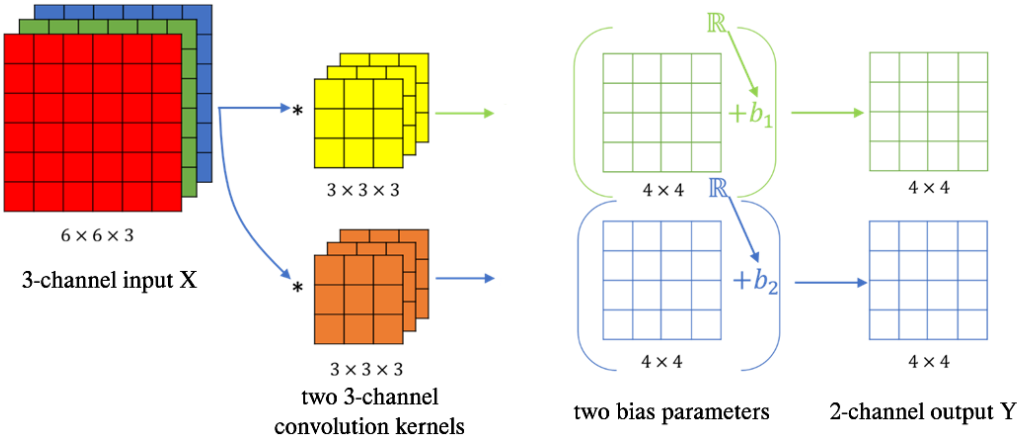


Figure 2.9: Convolution of an RGB image using multiple convolution kernels.
Image source: (Datahacker, 2018)

feature map $\mathbf{Y} \in \mathbb{R}^{H'\times W'\times C'}$ of the C' channel through the convolution operation, as described by the following calculation formula:

$$\mathbf{Y}[h,w,c] = \sum_{i=1}^{K}\sum_{j=1}^{K}\sum_{k=1}^{C} \mathbf{X}[h+i, w+j, k] \cdot \mathbf{K}_C[i,j,k] + \mathbf{b}[c] \tag{2.1}$$

In this formula, $\mathbf{K} \in \mathbb{R}^{C'\times K\times K\times C}$ is the convolution kernel of C' number of individual C channels, and $\mathbf{b} \in \mathbb{R}^{C'}$ is the bias corresponding to each output channel.

The calculation formula 2.1 is a linear weighted summation process. In our convolutional layer, we have some learnable parameters. Specifically, we use C' for the number of convolutional kernels and C for the number of input channels. What is more, the spatial resolution of the $k \times k$ convolution kernel results in so many parameters inside it. Additionally, for each channel, we add a bias, b, so there is a b as a parameter for each channel, which will be C' biases in total. It is important to understand the number of parameters our model for CNN needs to learn continues to grow as we add more complexity. If we add all the multiplication and addition operations, we will

have so $H'W'C'CK^2$ operations in total.

2.2.3 Activation Layer

After the convolutional layer outputs a feature map with multiple channels, we can further input this feature map into the next layer, which is usually the activation layer. The activation layer transforms pointwise the input feature map $\mathbf{X} \in \mathbb{R}^{H \times W \times C}$ using a non-linear transformation function $\sigma : \mathbb{R} \to \mathbb{R}$ and applying it to each element of the input feature map:

$$\mathbf{Y}[h, w, c] = \sigma(\mathbf{X}[h, w, c]) \tag{2.2}$$

The activation layer typically contains no learnable parameters and just applies the non-linear function 2.2 to the input. The most commonly used non-linear activation function in convolutional neural networks is the **R**ectified **L**inear **U**nit (*ReLU*) function (See Figure 2.10). For non-negative inputs, the output is equal to the input. For negative inputs, the output is always zero. The *ReLU* function is very simple compared to the *Sigmoid* function we use in multi-layer perceptrons. The computational cost of *ReLU* is much lower than that of the *Sigmoid* function as it only involves a comparison operation, while the sigmoid function often involves exponential calculations.

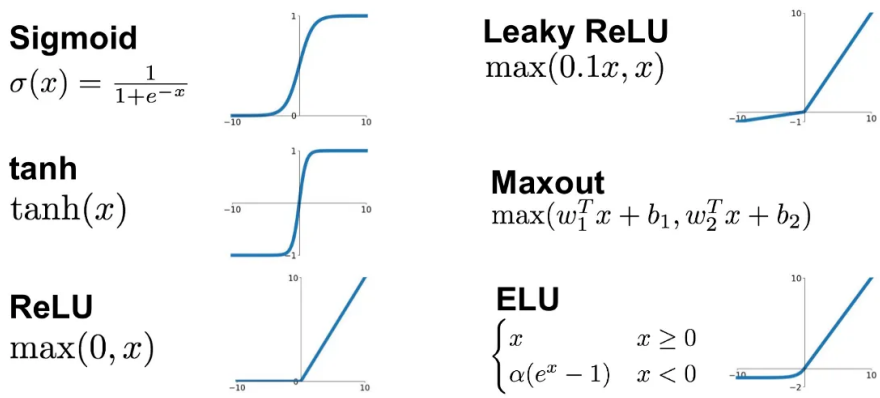


Figure 2.10: A list of commonly used activation functions.
Image source: Núcleo ML (2019)

As listed by Núcleo ML (2019), there are also other activation functions used in multi-layer perceptrons, such as the *Sigmoid* function and the hyperbolic tangent (*tanh*) function. We also have the *LeakyReLU* function. The *LeakyReLU* function allows for a small slope on the negative axis. There is also the Parametric ReLU (*PReLU*) function, which allows the slope to be learned during training. The *PReLU* function contains learnable parameters. There is also the possibility of using a more complex activation function called the Exponential Linear Unit (*ELU*).

2.2.4 Pooling Layer

After going through the convolution and activation layers, we need to input the feature map into something called the pooling layer.

The CNN cheatsheet from the CS230 course of Shervine Amidi, Stanford University (2023) is an excellent academic resource from which we can learn more about the different layers. The pooling layer calculates either the maximum or average value within a small spatial region of the feature map. For example, in Figure 2.11, we calculate either the maximum or average value

of every four numbers in the feature map. After pasting through the pooling layer, every four elements of the input map are reduced to one element in the output map, and this results in a reduction of the spatial resolution of the output feature map which is four times smaller than the spatial resolution of the input.

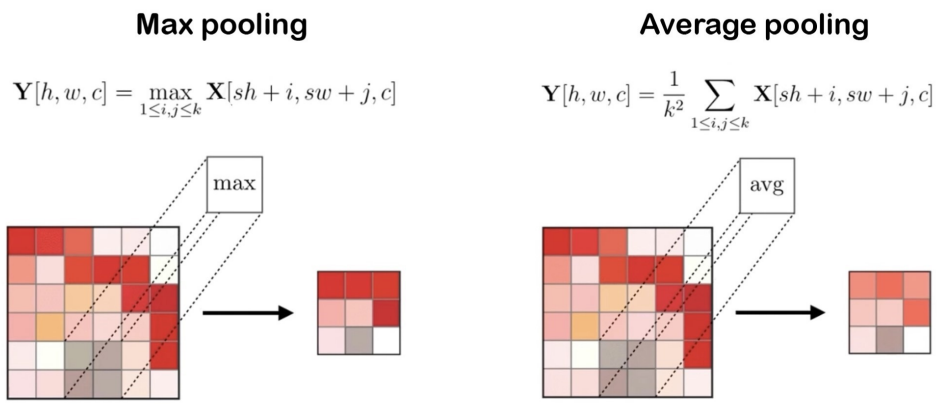


Figure 2.11: Difference between maximum and average pooling.
Image source: Shervine Amidi, Stanford University (2023)

As we have seen so far, the operations involved in the convolutional and activation layers can be computationally expensive due to the large number of calculations involved. By introducing pooling layers in our CNN, we can reduce the spatial resolution of the feature map layer by layer, achieving a significant reduction in computational costs, while still improving and maintaining the spatial robustness of important features.

2.2.5 Normalisation Layer

There is another special layer in convolutional neural networks called the normalization layer. According to Wu and He (2018), during training, a normalization layer is used to normalize the activation values (i.e., the feature map values) to improve the performance of the network. For example, we can normalize the activation values to have zero mean and unit variance. The purpose of this normalization is to reduce the difficulty of training and improve the stability of the training process. Because normalization is related to training, we will discuss this topic in more detail later on in **Section X TODO**.

2.2.6 Fully Connected Layer

After a series of convolutions and pooling operations, the feature map becomes smaller and smaller. When the feature map becomes very small, we can flatten it into a vector. Then we can use a multilayer perceptron (MLP) to further transform the flattened feature vector. In convolutional neural networks, the multilayer perceptron layer is also called a fully connected layer. This idea is further discussed in the Cheatsheet of Shervine Amidi, Stanford University (2023) from earlier.

The operation in this layer, as visualised in Figure 2.12, maps the input feature $X \in \mathbb{R}^N$ to the output feature $Y \in \mathbb{R}^M$ through matrix multiplication. The fully connected layer contains learnable parameters matrix $W \in \mathbb{R}^{M\times N}$, where $b \in \mathbb{R}^M$. Because we have already flattened the feature map into a vector, we can directly multiply this matrix W by the vector X and add a bias term b:

$$\mathbf{Y} = W\mathbf{X} + b \quad \text{OR} \quad \mathbf{Y} = W vec(\mathbf{X}) + b \tag{2.3}$$

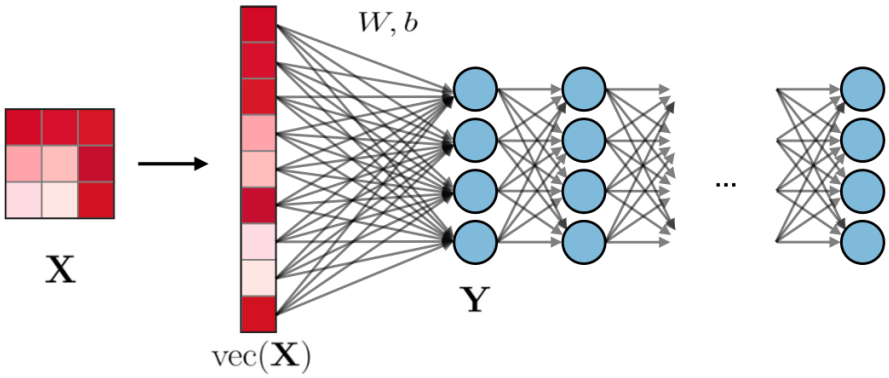


Figure 2.12: Visual representation of the fully connected layer.
Image source: Cheatsheet of Shervine Amidi, Stanford University (2023)

2.2.7 Probabilistic Output Layer

After going through a series of fully connected layers, the output of our network is a vector, but what we essentially want to output is the probability of the image belonging to each of the possible classes. We need to transform a feature vector into a probability vector $S : \mathbb{R}^K \to [0,1]^K$. This is where the concept of a probability output layer comes in, and it could be written down as $P(y|x) = S(h(x;\theta))$.

Typically, for binary classification problems, we use the Sigmoid function, so our goal can be achieved by compressing any real number to the range of $[0,1]$. The probability of the positive class is calculated by:

$$P(y=1|x) = \frac{1}{1+\exp(-z)} \tag{2.4}$$

For multi-dimensional classification problems, the Softmax function can be used. Let's assume we have a k-dimensional feature vector. First, we apply an exponential function to each element of the feature vector to transform it from a real number to a positive real number. Then, we normalize the values to compress each feature value within the range of $[0,1]$:

$$P(y=k|x) = \frac{exp(z_k)}{\sum_{i=1}^{K} \exp(z_k)} \tag{2.5}$$

Now, suppose we input an image of a lesion into a network (See Figure 2.13), and this image undergoes a series of convolution, pooling, activation, and linear layers. We obtain a three-dimensional feature vector with both positive and negative values. After applying the Softmax operation, we obtain a probability vector whose elements sum up to 1. This vector represents the probabilities of the image belonging to each of the three categories, which in our case are nevus, melanoma, or scar.

We can refer back to Figure 2.5. We input an image and pass it through a string of multiple layers, such as convolutional, activation, and pooling layers. Then, we obtain a small feature

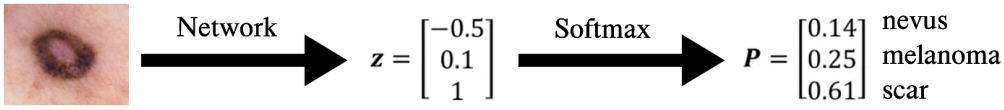


Figure 2.13: Softmax function as part of the Probabilistic Output Layer.

map, which we vectorize and pass through a series of fully connected layers. We perform a transformation between feature vectors. Finally, we take this feature vector, apply a softmax function to it, and then we obtain the probabilities of the image belonging to different classes. This is the end-to-end learning concept, where the image is on one end, and the probabilities are on the other.

What we would like is to consider this neural network from another perspective as well. We can divide it into two parts. The first part consists of a series of convolutional layers, activation layers, and pooling layers. We call this structure the **backbone** of the network, which takes an image as input and outputs a feature map. We hope that the backbone network can learn a "good" feature representation of the image. The fully connected layers, probability output layers, and so on, form a **simple classifier**.

In fact, the original intention of our computer vision tasks is not to hope to fit some arbitrarily complex classifier functions. What we hope to learn are good visual features of the target objects. These "good" visual features should be very robust to various pixels and deformations, no matter how the pixels change, as long as the target inside the images is a nevus or any other type of lesion. It should not make a difference where on the skin the lesion is located, no matter the lighting, skin colour or the angle from which the image was obtained, we should still be able to obtain robust features that can express our target lesion.

2.2.8 Convolution Operations

Convolutional operations can have some variations. Specifically, we can use techniques such as padding, stride, and dilation. After using these special techniques, the convolutional formula will also change. This will not be a focus of this report, so our small goal is to understand the idea behind it.

- Edge Padding: What is edge padding? Edge padding refers to a technique used before performing convolution operations on an input feature map. It involves adding zero pixels around the edges of the feature map. This padding expands the size of the feature map slightly. Typically, the convolution kernel has an odd number of pixels. By subtracting 1 from the kernel size and dividing it by 2, we can achieve an equal padding on all sides of the feature map. This results in a feature map with the same size as the input. For example, in Figure X, if the original input image is 5x5 and a 3x3 convolution kernel is used with one pixel of padding, the output feature map will also be 5x5.

- Convolution Stride: Convolution stride refers to the step size at which the convolution kernel slides over the image during the convolution operation. By increasing the stride from one pixel to two pixels, we can reduce the number of computations required and achieve spatial downsampling. With a stride of 2, the spatial resolution of the feature map is reduced by half. This configuration, often used in convolutional neural networks, enables efficient

computation and downsampling of feature maps while maintaining a consistent size through the use of padding when the kernel size is 3.

- Dilated Convolution: The third type of convolution operation is called dilation or atrous convolution, and it involves inserting some zeros in the convolution kernel to make increase its effective size. This technique is mainly used in image semantic segmentation tasks.

For deeper dive, the Deep Dive into Deep Learning course (Zhang et al., 2020) is definitely of use.

2.3 Evolution of Image Classification Model Design

So, the technology of image classification has undergone significant development in the past decade (See Figure 2.14), from methods based on traditional visual features to methods based on deep learning. From convolutional neural networks to updated model structure searches, as well as algorithms using transformers, all have brought the accuracy of image classification from a very low level to almost perfect.

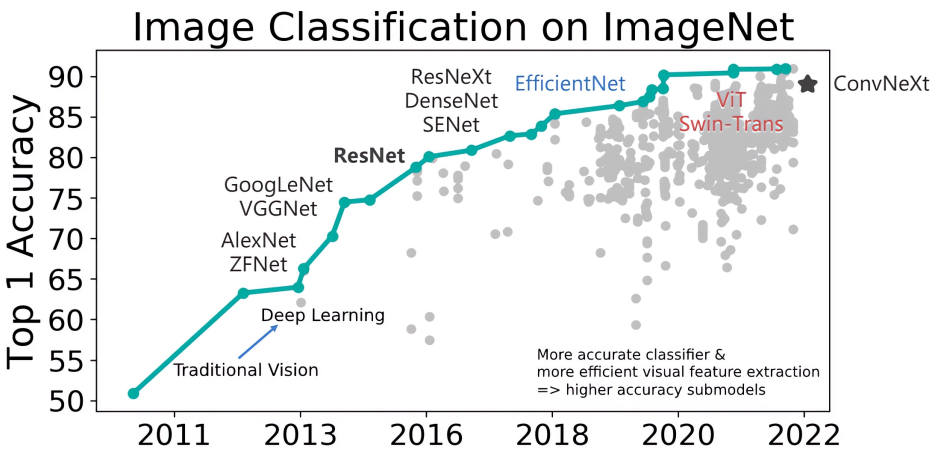


Figure 2.14: Image classification on ImageNet over the years.
Data source: Papers with Code (2023)

After learning about the various layers in a convolutional neural network in the previous section, the research focus in this section will shift to understanding how different combinations of these layers power neural networks that can be used for image classification. As discussed earlier, image classification networks can be divided into two parts. The main part of the network is called the backbone. The network takes an image as input, and after going through several layers of convolution, pooling, and ReLU non-linear activation, it produces a feature map. Then, the second part is a classifier, which usually consists of multiple fully connected layers. This part takes the feature map as input, performs a series of transformations, and outputs the final probability.

2.3.1 LeNet-5 (1998)

Entering the era of convolutional neural networks, it was the ground-breaking LeNet-5 model that was one of the first to introduce the idea of layered architecture. This model is the starting point of our research on the evolution of model designs. In 1998, this model emerged in Yann LeCun and

his team's paper (LeCun et al., 1998) and was intended primarily for the task of digit recognition (See Figure 2.15). With LeNet-5 came revolutionary steps forward in model design - setting a precedent for many future advancements in this field. This early model introduced fundamental concepts of layered structures and shared weights, principles that remain at the core of CNNs. LeNet-5 was relatively small due to limited computing hardware back then. LeNet-5 consists of two convolutional layers followed by pooling layers and three fully connected layers. It's a very small network that contains 60,000 learnable parameters.

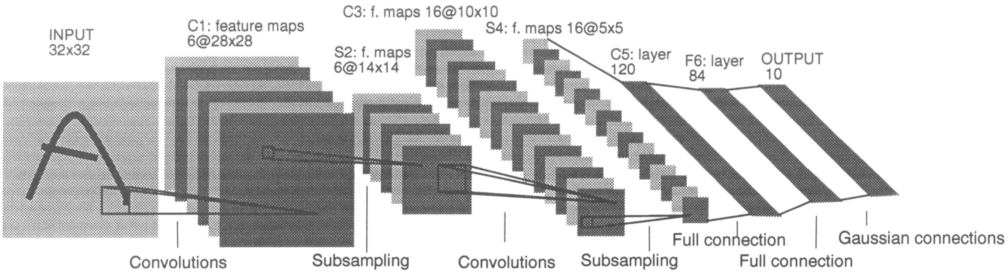


Figure 2.15: The architecture of LeNet-5 Convolutional Neural Network, in its original context of digits recognition, where "each plane is a feature map, i.e. a set of units whose weights are constrained to be identical."
Figure source: (LeCun et al., 1998)

2.3.2 AlexNet (2012)

There were no particularly famous image classification models after LeNet-5 until AlexNet emerged in 2012 improving the classification accuracy on ImageNet (Deng et al., 2009) by about ten percentage points. Compared to LeNet-5, AlexNet (Krizhevsky et al., 2012) is much larger, with eight layers, 8 learnable layers, 5 of which are convolutional and then 3 fully connected layers. The increase in model size also leads to a significant increase in computation cost. At that time, researchers also started using GPUs for parallel computing to accelerate the calculations. Despite the use of fast GPU hardware, the network still took about a week to train on two Nvidia GTX 580 GPUs.

In the second year, another work started to research AlexNet (Zeiler and Fergus, 2014). Since this neural network has so many layers, what is each layer doing? It is necessary to understand the role of each layer and not treat the entire network as a black box. We can see that the work in the second year proposed a method for visualizing neural networks. This method allows us to see what kind of features each layer has learned. This is one of its results (See Figure 2.16). We have extracted some images from their paper, and we can see that the layers close to the image input are usually referred to as the first layer. The first layer usually learns local features such as small edges and colours. These features in the second layer can be combined to form more complex features, such as circles. These features are further combined. Through this progressive combination at each layer, gradually more complex shapes can be obtained, and even some object features can be learned in the end.

After the validation of AlexNet, networks derived from it began to develop towards deeper and more complex architectures.

- VGG-19 (19 layers): ImageNet Top-5 Accuracy: 92.7%

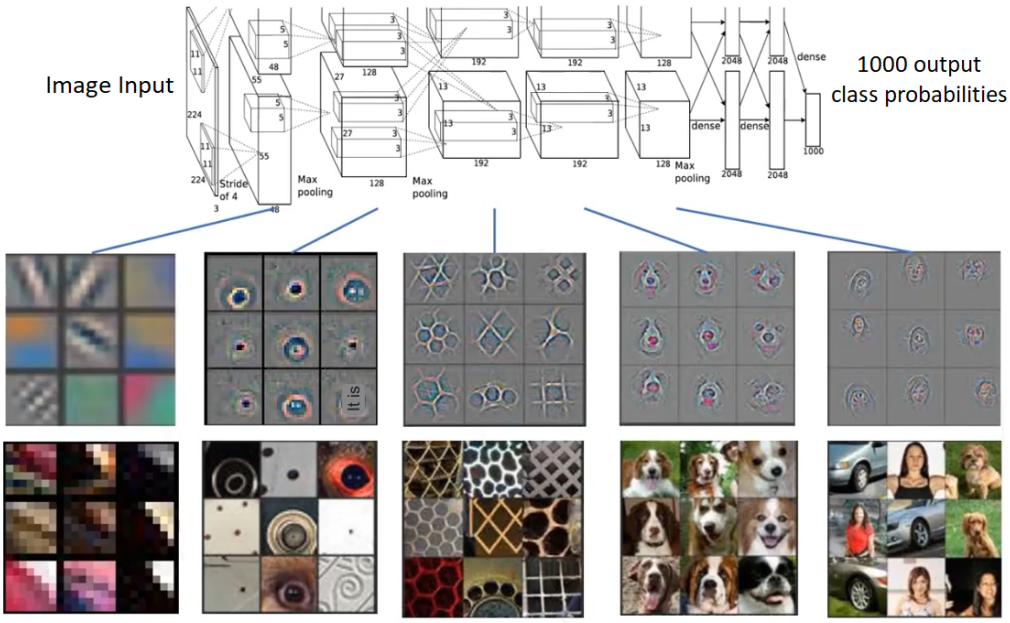


Figure 2.16: An illustration of the architecture of AlexNet CNN with a visual representation of the role for each layer.
Figure source: (Krizhevsky et al., 2012)

- GoogleLeNet (22 layers): ImageNet Top-5 Accuracy: 93.4%

2.3.3 VGGNet (2014)

The first work is called VGG (Simonyan and Zisserman, 2014), the model that secured the second place in the 2014 ImageNet competition (Russakovsky et al., 2014), significantly increased the depth of the network compared to AlexNet, featuring a structure with 16-19 layers (See Figure 2.17). A key innovation of VGG is the exclusive use of 3x3 convolutional kernels across all its layers, contrasting with AlexNet's use of varying kernel sizes including 5x5 and 11x11.

The use of 3x3 convolutional kernels reduces computational complexity and enhances image feature extraction. This reduction in kernel size directly impacts the receptive field, which refers to the region of the original image each neuron can 'see' during multiplication operations. A 5x5 kernel covers a 5x5 region of the image, while a 3x3 kernel covers a smaller 3x3 region. However, stacking two 3x3 kernels effectively covers the same 5x5 region with fewer parameters (18 vs. 25), thus simplifying the model while preserving the receptive field.

This simplification, coupled with an additional layer of nonlinearity, tends to yield superior network performance. The VGG architecture is founded on the principle of adding many such layers, extending up to 16-19, far exceeding the depth of AlexNet. During forward propagation, VGG employs padding to maintain the spatial resolution of the feature map. After a series of convolutional layers, pooling is applied to the feature map, which reduces its spatial resolution while doubling the number of channels.

From the article "Difference between AlexNet, VGGNet, ResNet, and Inception" (Anwar, 2019) we can see that AlexNet has 60 million parameters, while VGGNet has 138 million parameters, with over 100 million parameters coming from the final fully connected layer. Because there are so many model parameters, it's easy for overfitting to occur. Therefore, VGGNet-like

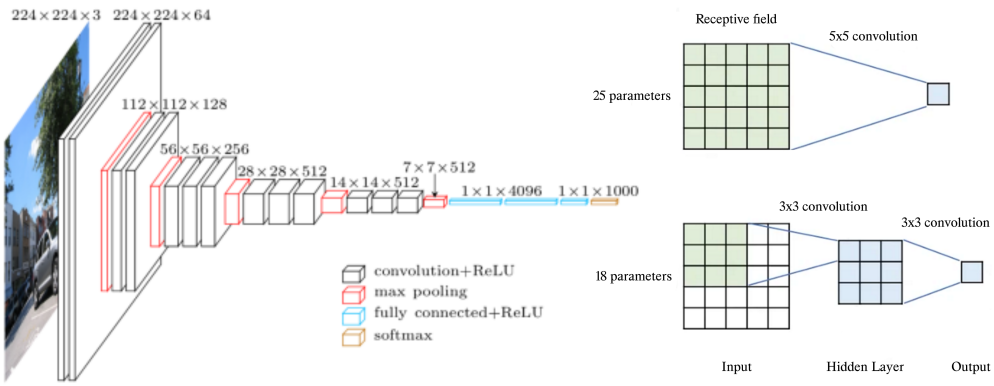


Figure 2.17: (LHS) Diagram of the VGG16 architecture.
(RHS) Receptive field of single 5x5 convolution vs. two 3x3 convolutions.
(LHS) Figure source: Deep Learning in Medicine: Classifying Melanoma (Cloudera, 2019)

networks must be combined with other strategies, which we will explain in detail later.

2.3.4 GoogLeNet (Inception v1) (2014)

In the same 2014 ImageNet image recognition challenge (Russakovsky et al., 2014), a network architecture called GoogLeNet (Szegedy et al., 2014) showcased remarkable performance. While its name pays tribute to LeNet, it is difficult to see any resemblance to LeNet in the network structure (See Figure 2.18).

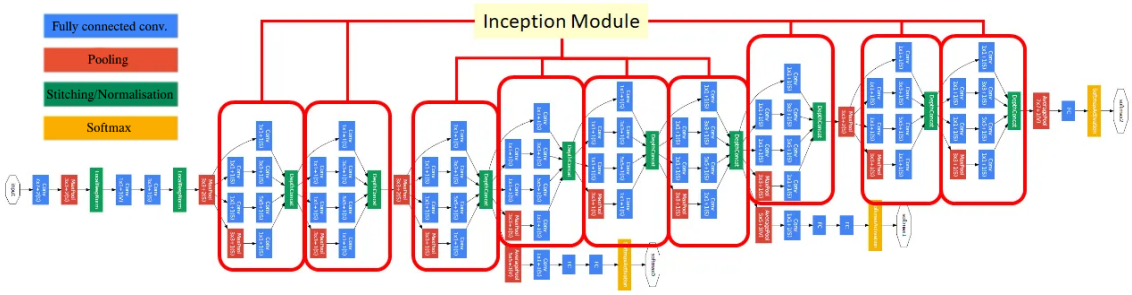


Figure 2.18: "GoogLeNet network with all the bells and whistles."
An edited graphic of the original figure: (Szegedy et al., 2014)

The basic convolutional block in GoogLeNet is called the Inception block, named after the movie "Inception". This module allows the use of convolution kernels with different sizes to perform convolution operations. AlexNet and VGGNet have a similar depth, but with its 22 layers, GoogLeNet is able to utilize the specifically designed Inception module. Due to the Inception module, GoogLeNet reduced the number of parameters significantly and has only 7M weight parameters.

The Inception module consists of four parallel lines (See Figure 2.19). The first three paths use convolutional layers with window sizes of 1×1, 3×3, and 5×5 to extract information from different spatial scales. The middle two paths perform 1×1 convolutions on the input to reduce the number of input channels, thus reducing model complexity. The fourth path utilizes a 3×3 max-pooling layer followed by a 1×1 convolutional layer to modify the channel dimension. All four paths use appropriate padding to ensure consistency in the height and width of the input and

output. Finally, the outputs from each path are concatenated along the channel dimension and fed into the subsequent layers.

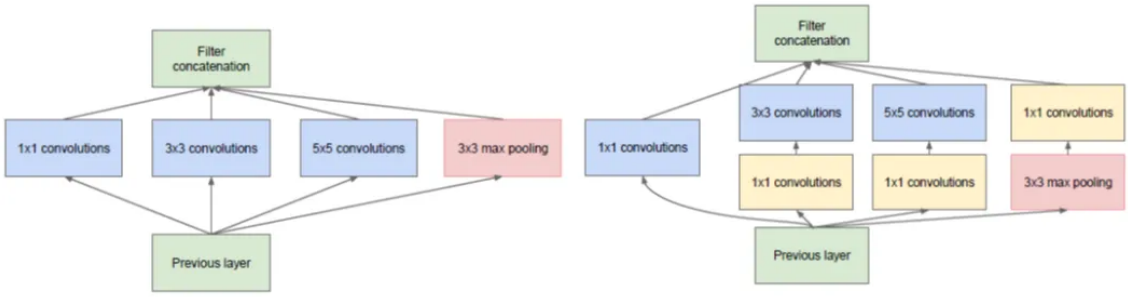


Figure 2.19: (LHS) Inception Module (Left)
(RHS) Inception Module with Dimensionality Reduction.
Image source: (Tsang, 2018)

The customizable hyperparameter in the Inception block is the output channel size for each path, which allows us to control the model's complexity.

With the iterative development of deep learning and neural networks, numerous network model structures have emerged. However, when following paper (He et al., 2016) demonstrates the problem that arises when the number of layers in a model increases to a certain extent, the accuracy of model classification decreases instead of improving. The problem of accuracy degradation may seem counterintuitive, but the experiments have shown that when convolution degenerates into identity mapping, deep networks perform the same as shallow networks. Therefore, deep networks should have classification accuracy that is no worse than shallow networks, so that the stacking of deep layers can be effective.

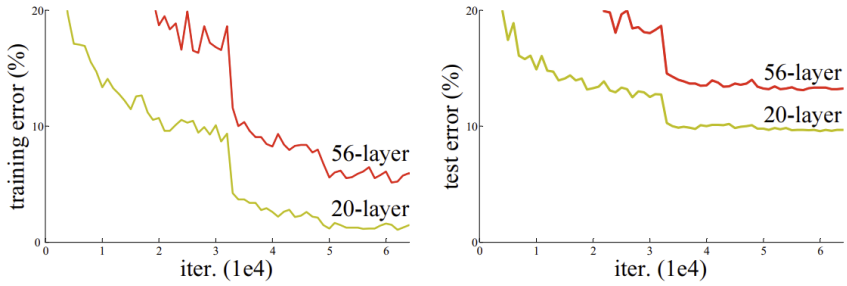


Figure 2.20: "Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error."
Figure source: (He et al., 2016)

2.3.5 ResNet (2016)

Next follows, one of the most influential and widely used model structures so far, which won the CVPR 2016 Best Paper Award.

The paper (He et al., 2016) also presents the innovative concept of residual learning which introduces cross-layer connections to increase the network depth to hundreds of layers. We can see from the network structure diagram in Figure 2.21 that it actually involves some skip connections between layers. Traditionally, each layer in a network would output its result as input to the next layer. But now with these skip connections, we can directly skip over several layers in the network.

his allows us to give information to layers that are much further along in the network.

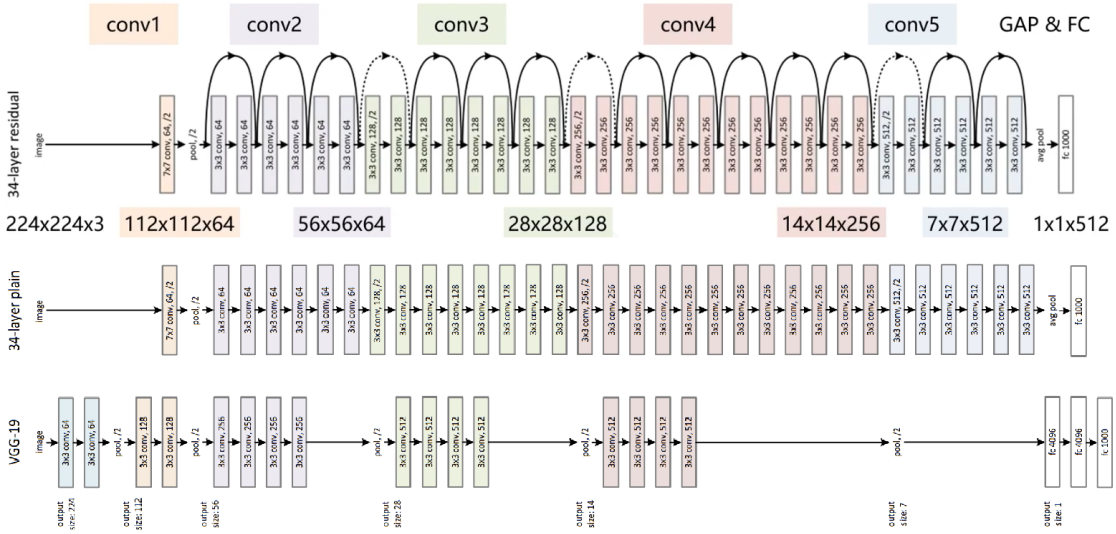


Figure 2.21: Example network architectures for ImageNet.
Bottom: The VGG-19 model (19.6 billion FLOPs) as a reference.
Middle: A plain network with 34 parameter layers (3.6 billion FLOPs).
Top: A residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase the dimensions.
Figure source: (He et al., 2016)

The cross-layer connections used in ResNet are called residual modules, which have two structures (See Figure 2.22): Basic blocks and Bottleneck blocks. The Basic block is essentially a cross-layer connection, and then two 3x3 convolutional layers are used to learn the residual term. The Bottleneck block is mainly used for larger modules. We can use a 1x1 convolution to compress the number of channels and reduce its computational complexity. Then, we can perform a 3x3 convolution on the feature map with a lower channel number and restore its dimension using another 1x1 convolution. This is mainly used for a larger ResNet networks to compress computational complexity.

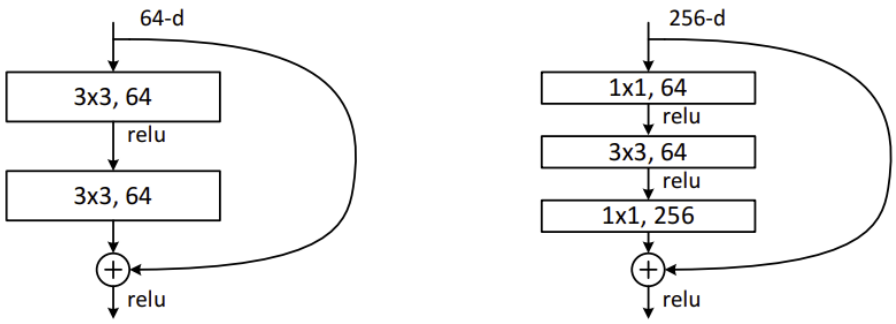


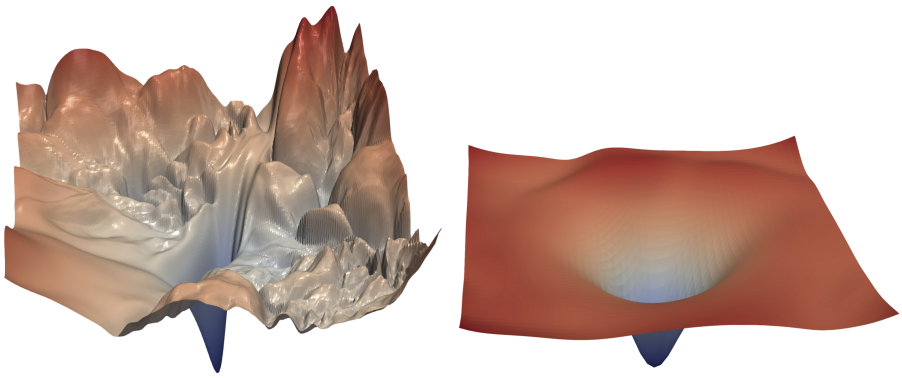
Figure 2.22: (LHS) Basic Building Block for ResNet-18/34
(RHS) Bottleneck Building Block for ResNet-50/101/152
Figure source: (He et al., 2016)

With the understanding of the residual module, we can look at the overall structure of the ResNet model which is the top part of Figure 2.21. ResNet typically consists of five stages, and

the first stage is very simple, consisting of only conv1, which is a 7x7 convolutional layer. Conv2 to Conv5 are actually the accumulation of residual modules in the five stages. By using different residual blocks in a residual module, we can obtain different ResNet structures. ResNet can range from 18 layers to a maximum of 152 layers.

These five stages have a specific spatial resolution, and the spatial resolution will be halved after each stage. After a series of residual modules, we obtain the final feature map. If we input an image of size 224x224, after five rounds of sampling and downsampling, the image becomes 1/32 of its original size, which is a 7x7 feature map. Then we perform global average pooling on this feature map and obtain a 1x1x512 feature vector. Then we take this feature vector, pass it through a fully connected layer, and output the probability of belonging to each class for the image. What is more, if we replace the basic modules with bottleneck modules we can further increase the number of residual modules at each level.

For the structure of neural networks, ResNet can be considered as an ensemble of deep models and shallow models, equivalent to a multi-model ensemble. The residual network has $O(2n)$ implicit paths connecting input and output, and the number of paths doubles with each added block. The so-called residual connection refers to summing the output of the shallow layer and the deep layer as the input for the next stage. The result of this is that the weights originally needed to learn the mapping from $x \rightarrow f(x)$. Actually, residual connections can be seen as a special kind of skip connection. After using the residual link, the mapping that the weight needs to learn becomes $x \rightarrow f(x) - x$. In this way, as described in (Li et al., 2018), during the process of backpropagation, the gradient of small losses is more likely to reach the neurons in the shallow layer.



(a) without skip connections (b) with skip connections

Figure 2.23: The loss surfaces of ResNet-56 with/without skip connections.
Figure source: (Li et al., 2018)

2.4 Efficient Neural Networks

Next, let's look at the model design from another perspective. Traditional convolutional neural networks have achieved excellent results in the field of computer vision. When we design models, we strive for higher accuracy. However, if a model is to be practical, we must also consider its computational cost. The calculation cost cannot be too high because devices like smartphones or Nvidia Jetson Nano have very small chips. It's difficult to use large models on such small devices. The most notable advantage of smaller models (See Figure 2.24) is that they can be efficiently trained in a distributed manner, reducing model update costs, overcoming limitations in platform

size, power consumption, storage, and computational capabilities, and facilitating deployment on mobile devices.

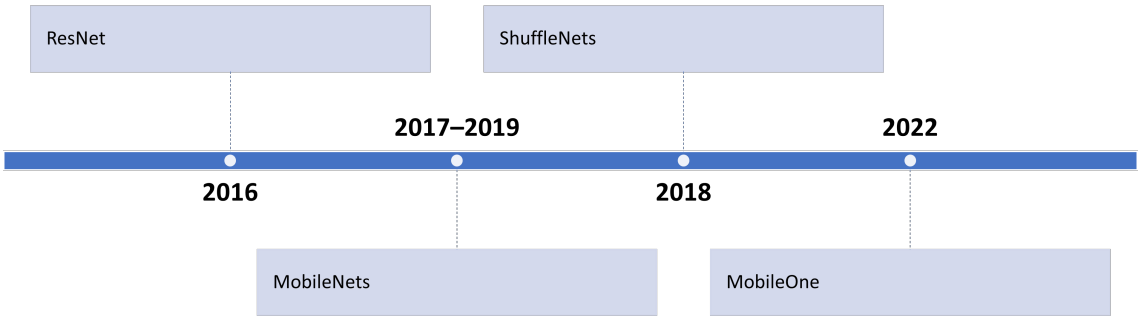


Figure 2.24: Candidate List of Efficient Neural Networks

2.4.1 ResNet (2016)

Indeed, we list again ResNet (He et al., 2016) but this time it enters our candidate list of efficient neural networks. This specific "resnet34_8xb32_in1k" variant of ResNet has 34 layers (depth) and utilises a batch size of 32 during training. The reason this model has sparked our interest is because of its performance metrics (See Table 2.1).

Table 2.1: Performance metrics of ResNet.
Model results are from Image Classification on ImageNet-1k.
Data source: (OpenMMLab's Model Zoo, 2016)

Model	Params (M)	Flops (G)	Top-1 (%)	Top-5 (%)
mobilenet-v2_8xb32_in1k	2.18	3.68	73.62	91.59

For a detailed description of each of the columns in the table above, please, refer to the description below Table 2.2.

2.4.2 MobileNets (2017~2019)

The MobileNet series of models is a pioneering work in the domain of efficient neural networks. These models were designed specifically to reduce the computational cost, making them suitable and notable candidates for deployment on smaller devices with limited computational resources. This efficiency is achieved mainly through the use of depthwise separable convolutions, a technique that significantly reduces the number of computations needed compared to regular convolutions.

MobileNet v1 (2017)

The first work on MobileNet (Howard et al., 2017) proposed a straight tube network structure that reduced computational costs. It strikes a balance between latency, size, and accuracy while demonstrating favourable comparisons with well-known models from existing literature (Google, 2017) (See Figure 2.25).

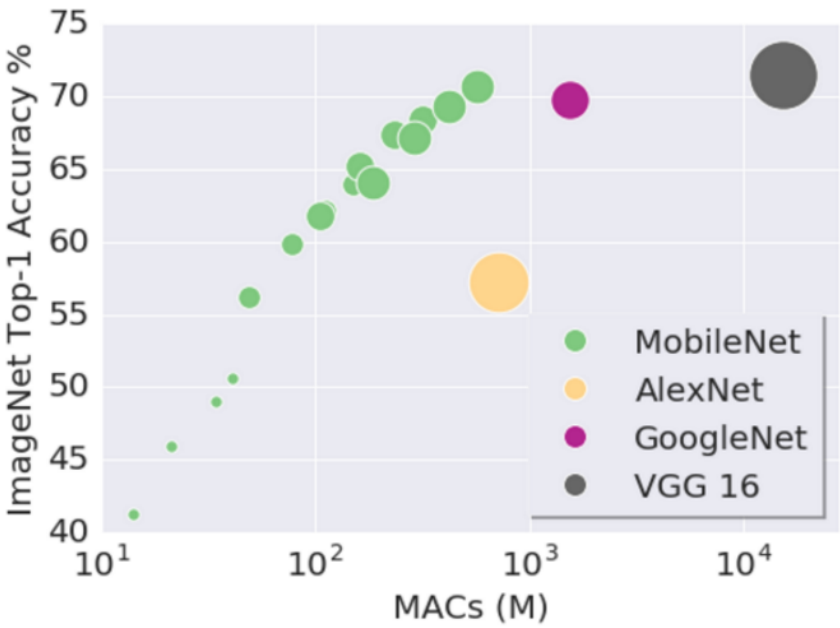


Figure 2.25: A comparison between MobileNet v1, AlexNet, GoogLeNet and VGG 16. The horizontal scale is in MACs or Multiplication and Addition operations needed. Figure source: (Google, 2017)

MobileNet decomposes the regular convolution into depthwise convolution and pointwise convolution which reduces the number of computations. But how does this deconstruction process work? In Figure 2.26, we will first explore the concept of a standard convolution. Suppose we have a three-channel image and we are using a three-channel convolution kernel. The computational cost for each point on the output feature map involves multiplication with a parameter of size $D_k^2 N$, where D_k is the size of the kernel, and N is the number of parameters.

However, we can decompose this into a sparse connection by introducing sparsity in the channel dimension. First, we use a single-channel convolution kernel to perform convolution with each input channel, resulting in a single-channel feature map for each input channel. Then, we perform a one-channel convolution for each of these feature maps, resulting in a series of feature maps. We then combine these feature maps, and we are able to maintain the same number of channels as the original image. Finally, we use a 1x1 convolution to add channels, which, for example, could convert the input from three channels to 32 channels. The number of parameters for this operation is shown in Figure 2.26 and we can see that the computational cost is much lower than that of a regular convolution.

The MobileNet model is trained using RMSprop (Hinton, 2012) with asynchronous gradient descent, similar to Inception V3. Another advantage, due to their smaller size, is that these models are less prone to overfitting and hence do not require as much regularization or data augmentation techniques as their larger counterparts. The MobileNet models are also designed to be very adaptable, with global hyperparameters that allow the model builder to efficiently trade-off between latency and accuracy based on the constraints of the problem.

One of the most notable advantages of MobileNet's architecture is its compact network design. A full-sized MobileNet has only 4.2 million parameters (Google, 2017), in stark contrast to

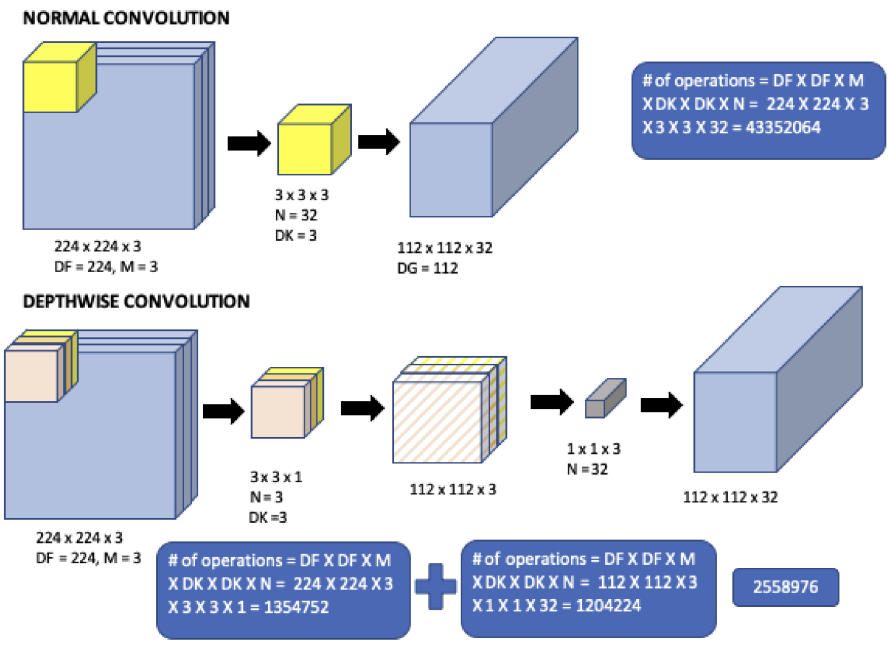


Figure 2.26: (Top) Regular convolution
(Bottom) Layer-by-layer convolution, Point-by-point convolution
Figure source: (Kar, 2020, p. 308)

the tens of millions of parameters seen in earlier network models we reviewed earlier. On top of that, MobileNet can be compressed even further, with the smallest MobileNet having only about one million parameters.

MobileNet v2 (2018)

One year later, MobileNet v2 (Sandler et al., 2018) introduced the concept of an inverted residual structure. This is a modification of the traditional residual structure we discussed (See Figure 2.22). In a traditional residual block, the number of channels is first reduced, then the reduced feature map is convolved, and finally, the number of channels is restored. In contrast, in the inverted residual block, the number of channels is first increased, then the increased feature map is convolved, and finally, the number of channels is decreased. This results in a lighter model with similar or even superior performance compared to its predecessor.

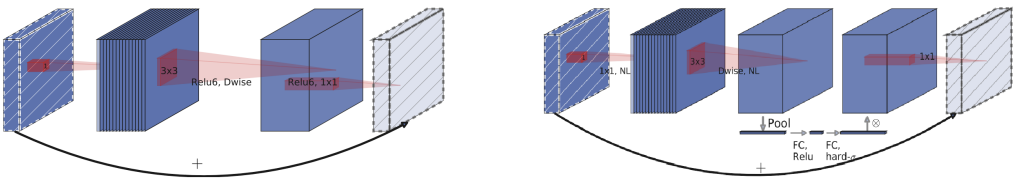


Figure 2.27: (Left) Mobilenet v2: bottleneck with residual
(Right) Mobilenet v3 block: MobileNetV2 + Squeeze-and-Excite
Figure source: (Howard et al., 2019)

MobileNet v3 (2019)

MobileNet has undergone a series of improvements reaching to MobileNet v3 (Howard et al., 2019). This version incorporated an attention mechanism via the SE (Squeeze-and-Excitation)

module (See Figure 2.27). The SE module is a plug-and-play module which introduces an attention mechanism, which means that a convolutional layer can output multiple channel features and each channel may not be the same. As a result, this module allows the model to learn to use global information to selectively emphasise informative features and suppress less useful ones, improving the representational power of the network. MobileNet v3 also made modifications to the network structure and the number of channels and introduced non-linear transformations.

Let's dive into the key differentiating factors between MobileNet v3 and its predecessor, MobileNet v2:

First, the MobileNet v3 architecture is available in two sizes: Large and Small. These sizes are specifically designed to accommodate different resource and latency constraints.

Secondly, MobileNet v3 harnesses the power of Neural Architecture Search (NAS), which is a mix between manual design and automated design. This updated model architecture can to be optimized in a way that it takes into account the specific hardware constraints of the platform where the model will be deployed. This means that the model has been designed to take full advantage of the specific computational resources available on mobile and embedded devices.

In terms of performance metrics such as Top-1 Accuracy, MobileNet v3 outperforms MobileNet v2. This merged table 2.2 provides a summary of various pre-trained models and their performance metrics.

Table 2.2: Performance metrics of different MobileNets.
Model results are from Image Classification on ImageNet-1k.
Data source: (OpenMMLab's Model Zoo, 2018a) and (OpenMMLab's Model Zoo, 2019)

Model	Params (M)	Flops (G)	Top-1 (%)	Top-5 (%)
mobilenet-v2_8xb32_in1k	3.50	0.32	71.86	90.42
mobilenet-v3-small_8xb128_in1k	2.54	0.06	66.68	86.74
mobilenet-v3-large_8xb128_in1k	5.48	0.23	73.49	91.31

A description of how to interpret each column:

- **Model**: This column lists different variants of the MobileNet backbones. The names of the models usually contain information about their structure or training strategy. For example, let's decode *"mobilenet-v3-small_8xb128_in1k"*, where:

  •*"v3-small"* specifically refers to a variant of the MobileNet architecture that it is the smaller version of the MobileNet v3 model.

  •*"8xb128"* denotes the width multiplier and input channel configuration of the model. The width multiplier determines the overall model size and computational complexity. If we have a higher width multiplier this would generally result in a larger and more accurate model but requires more computational resources. We can further decode:

    •*"8xb"* suggests that the width multiplier is set to 8 times the base width.

    •*"128"* indicates that the input channels (also known as input depth) of the model are set to 128.

  •*"in1k"* signifies the number of output classes or categories that the model is trained to classify. In this case, *"in1k"* stands for "input 1000" and implies that the model is trained on

a dataset with 1000 different classes or categories.

- **Params (M)**: This column represents the number of parameters in the model, in millions (M). Models with more parameters can often capture more complex patterns, but they may also be more prone to overfitting and require more computational resources.

- **Flops (G)**: This stands for "floating point operations per second" in billions (G). It's a measure of the computational complexity of the model, where more FLOPS means that the model is more computationally demanding.

- **Top-1 (%)** and **Top-5 (%)**: These are accuracy metrics on the ImageNet-1k dataset, where:

  •*Top-1 accuracy* is the percentage of times the model correctly predicts the most likely category out of 1,000.

  •*Top-5 accuracy* is the percentage of times the true category is within the model's top five predictions out of 1,000.

Later in this report, we are going to evaluate the models from Table 2.2, but before that we would need to consider some of the following situations given our set of priorities:

If our priority is high accuracy and computational power is not a big concern, then *"mobilenet-v3-large_8xb128_in1k"* would be a suitable choice for our project. It has the highest Top-1 accuracy (73.49%) and Top-5 accuracy (91.31%). However, it also requires more computational resources, with 5.48 million parameters and 0.23 billion FLOPs.

If our primary concern is to use a lightweight model that requires fewer computational resources and provides "good enough" predictions, then *"mobilenet-v3-small_8xb128_in1k"* would be the most appropriate choice. Despite having the lowest Top-1 accuracy (66.68%) and Top-5 accuracy (86.74%) among the given models, it only has 2.54 million parameters and 0.06 billion FLOPs, making it the least computationally demanding.

Alternatively, the *"mobilenet-v2_8xb32_in1k"* model provides a middle ground between the other two, with relatively high accuracy (Top-1: 71.86%, Top-5: 90.42%) and moderate computational requirements (3.5 million parameters and 0.32 billion FLOPs).

Given that our use case is for medical image analysis, and considering the importance of accurate diagnosis, a model with higher accuracy would generally be more beneficial. However, this should be balanced with the computational constraints of the mobile device. For best results, we are going to fine-tune our models of choice on the target medical image dataset, which in our case would be either ISIC 2018 (Codella et al., 2018)(Tschandl et al., 2018) or ISIC 2020 (Rotemberg et al., 2021).

2.4.3 ShuffleNets (2018)

ShuffleNet is a computationally efficient convolutional neural network (CNN) architecture designed primarily for mobile devices with limited computing power, like drones, robots, and smartphones, making it a perfect candidate for the use case of this dissertation project. The goal of ShuffleNet is to achieve high accuracy in visual recognition tasks under tight computational budgets, typically in the tens or hundreds of MFLOPs (See Table 2.3).

Table 2.3: Performance metrics of different ShuffleNets.
Model results are from Image Classification on ImageNet-1k.
Data source: (OpenMMLab's Model Zoo, 2018b) and (OpenMMLab's Model Zoo, 2018c)

Model	Params (M)	Flops (G)	Top-1 (%)	Top-5 (%)
shufflenet-v1-1x_16xb64_in1k	1.87	0.15	68.13	87.81
shufflenet-v2-1x_16xb64_in1k	2.28	0.15	69.55	88.92

ShuffleNet v1

The architecture of v1 (Zhang et al., 2018) employs two new operations: pointwise group convolution and channel shuffle (See Figure 2.28). These operations significantly reduce computation cost while maintaining accuracy. In fact, ShuffleNet has shown comparable performance to other efficient structures like MobileNet, achieving Top-1 error on ImageNet classification tasks of 68.13% within a computation budget of 150 MFLOPs.

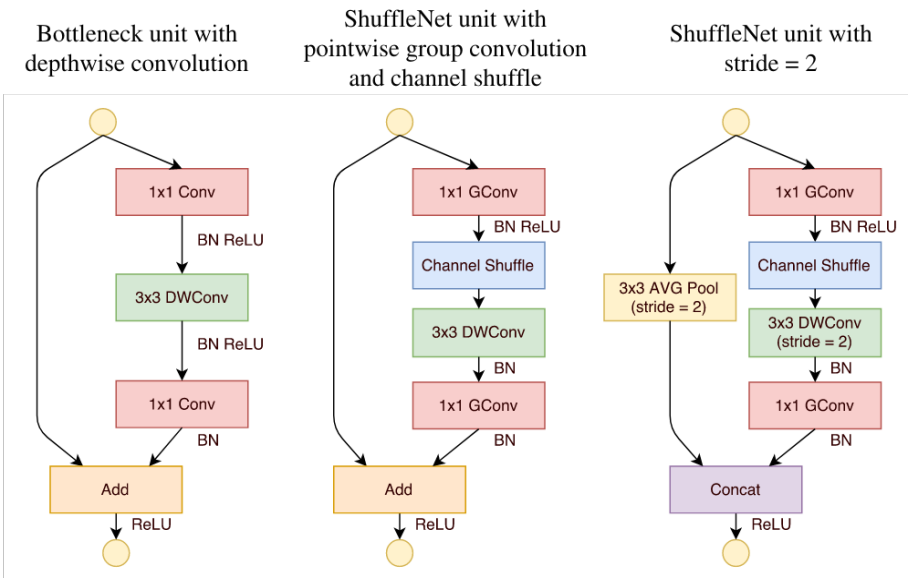


Figure 2.28: ShuffleNet Units.
Figure source: (Zhang et al., 2018)

The channel shuffle operation is a key innovation in ShuffleNet, which helps to ensure information flow across feature channels. This operation is particularly beneficial for smaller networks, where encoding more information becomes critical to their performance.

ShuffleNet v2

The main reason for the release of ShuffleNet v2 is to optimize the architecture under the same complexity as v1, but with an emphasis on being more "hardware friendly." The original paper (Ma et al., 2018) proposes to evaluate the direct metric on the target platform, beyond only considering FLOPs, and derives several practical guidelines for efficient network design:

1. Use "balanced" convolutions (equal channel width)

2. Be aware of the cost of using group convolution

3. Reduce the degree of fragmentation

4. Reduce element-wise operations

Based on these guidelines, V2 uses only one element-wise operation, reducing the complexity compared to V1. Moreover, V2 removes the pointwise group convolutions used in V1 to further simplify the architecture. Ultimately, ShuffleNet V2 is presented as a new architecture that is more efficient in terms of speed and accuracy under similar computational budgets.

2.4.4 MobileOne (2022)

TODO Describe MobileONe

TODO: Conclusive paragraph about Efficient Neural Networks.

2.5 Training Image Classification Models

TODO Describe the training process

Chapter 3

Methodology

This section will discuss any particular approaches, environmental requirements and specific technologies used during the development of this project. It will explore the process of configuring a model and offers a comprehensive explanation of the reasoning behind some specific aspects of the neuron network designs, setting up the process of training, validation and testing pipelines and generation of Kaggle submission files.

3.1 Medical Image Datasets of Skin Lesions

Our main goal is to select and fine-tune accurate and efficient algorithms for classifying skin lesions as benign or malignant. The importance of using a carefully curated dataset for training lies in their potential to improve our algorithm, ultimately leading to better patient outcomes. So in this project, we are going to utilise two datasets maintained by research groups from global universities. Both data sets contain dermoscopic images of skin lesions and the following sections provide a detailed description of them.

3.1.1 ISIC 2018

The ISIC 2018 Challenge Task 3 on Lesion Diagnosis (Codella et al., 2018)(Tschandl et al., 2018) involves classifying dermoscopic images into one of seven disease categories. The data set includes training, validation, and test data, along with their corresponding CSV metadata ground truth values. The training data consists of 10,015 images, while the validation and test data contain 193 and 1512 images, respectively.

The ISIC 2018 data set is used in this project for classifying skin lesions as benign or malignant and it was chosen because of its relatively small number of images. The images are provided in JPEG format and we have to use a Python script to arrange the dataset into ImageNet's folder structure format, which is necessary for compatibility with our framework of choice - OpenMMLab's MMPreTrain.

3.1.2 ISIC 2020

The ISIC 2020 data set (Rotemberg et al., 2021) is much more complex in its nature. It contains 33,126 dermoscopic training images of unique benign and malignant skin lesions from over 2,000 patients. Metadata is also available in CSV files, including patient information and lesion characteristics. In contrast to ISIC 2018, the ISIC 2020 data set only includes training and test data, meaning that the training data must be split into two parts for training and validation purposes. Additionally, ground truth values for the test data are not provided, which means that for testing the accuracy of our candidate efficient models on ISIS 2020, we are required to make a submission

to Kaggle (ISIC, 2020) for evaluation and comparison in order to receive the inference accuracy. Similarly, to ISIC 2018, in this project, the JPEG files are rearranged in the ImageNet format, using a Python script, and are used for training. When using the data source for training, we utilise a binary probability on whether the image is benign or malignant

The main differences between the ISIC 2018 and ISIC 2020 data sets are the sample size, image dimensionality, and availability of ground truth values for test data. The ISIC 2020 data set has a larger sample size and higher image dimensionality than the ISIC 2018 data set. For example, the resolution of the images in ISIC 2018 is fixed at 600x450, while the resolution of the images in ISIC 2020 varies from 600x450 to 6000x4000.

3.1.3 Dataset preprocessing into supported folder structure

The code submission contains two Jupiter notebooks: *isic_2018_data_preprocessing.ipynb* and *isic_2020_data_preprocessing_for_submission.ipynb*.

The first Jupyter notebook *isic_2018_data_preprocessing.ipynb* is focused on preparing the ISIC 2018 Task 3 dataset for training, validation, and testing using MMPreTrain (OpenMMLab, 2023). The full dataset can be downloaded from ISIC 2018 Task 3 skin lesion analysis challenge webpage (Codella et al., 2018)(Tschandl et al., 2018).

In all three sections of the notebook, the script effectively filters and organizes the data to focus on two types of skin lesions (benign and malignant), and it prepares three separate datasets for training, validation, and testing. This organization is crucial for training a machine learning model, as it allows effective learning from the training data, tuning with the validation data, and final performance evaluation with the test data.

This second notebook *isic_2020_data_preprocessing_for_submission.ipynb* also prepares a dataset for MMPreTrain, but it uses the ISIC 2020 dataset, whereas the first notebook uses the ISIC 2018 dataset. The objective remains the same: to format and structure the data in a way that meets the requirements for training a machine learning model with MMPreTrain.

The first key difference here is that the ISIC 2020 contains some duplicates, by mistake of the group that has packaged the images, therefore, the second notebook includes a step to remove duplicate images.

Another key difference is that the dataset is then split into a training set and a validation set, with 80% of the data going to the training set and 20% to the validation set.

3.2 OpenMMLab: A Unified Benchmark Framework

OpenMMLab (OpenMMLab, 2023), is an open-source algorithm platform for computer vision, which depends on PyTorch. This platform is designed to accelerate research in computer vision by providing a unified architecture for various research areas.

OpenMMLab differentiates itself by offering a unified benchmark framework, with a modular design, and high-quality implementation. Their GitHub repositories enjoy constant growth in the number of Stars History, which makes them a reputable project.

The need for a unified benchmark framework is underscored by the complexity and diversity of methods in computer vision research (See Figure 3.1). The unified benchmark framework allows us to compare different methods under the same settings, which is crucial for fair comparison and evaluation. This is particularly important in the field of computer vision, where different

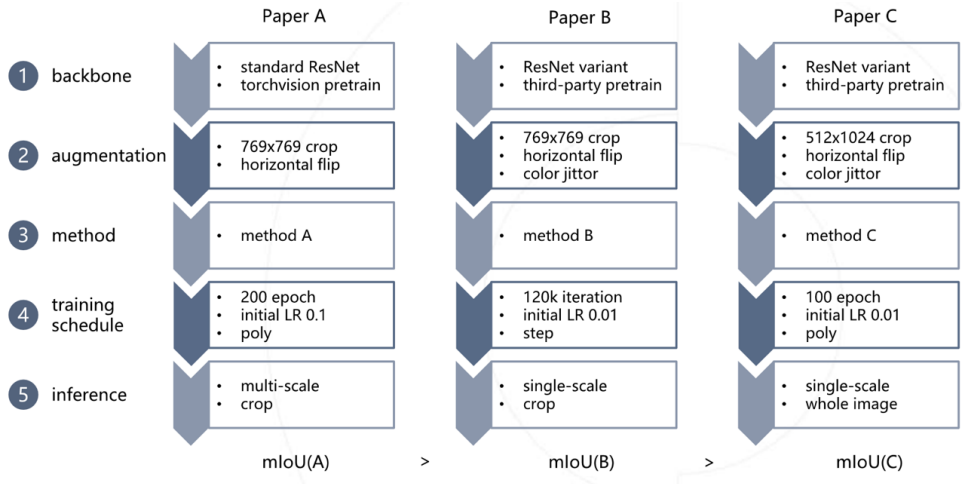


Figure 3.1: Need for Unified Benchmark Framework
Figure source: (OpenMMLab, 2023)

methods often use different backbones, augmentations, training schedules, and inference strategies, making it difficult to determine which method is truly the best.

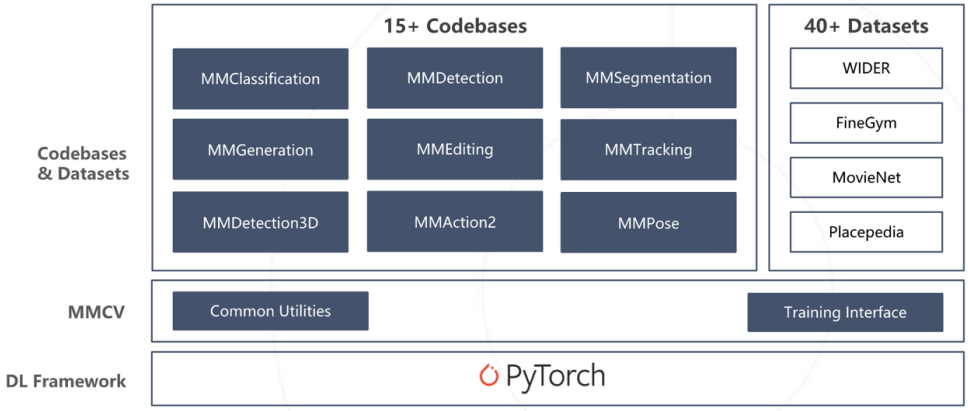


Figure 3.2: OpenMMLab's Framework Modular Design
Figure source: (OpenMMLab, 2023)

OpenMMLab is the framework of choice for our project because its modular design (See Figure 3.2) allows us to quickly research, develop and try new components while keeping the focus on efficiency, accuracy, and good coding style. Embracing this unified benchmark framework would help us standardize the evaluation procedures, making it easier for us to compare our candidate list of efficient neural network models and let us experiment with different potential model improvements. It has implemented over 150 state-of-the-art algorithms and released more than a thousand pre-trained models, making it a rich benchmark environment for computer vision research.

OpenMMLab supports more than 15 code bases, including MMPreTrain for image classification, MMDetection for object detection, and MMSegmentation for semantic segmentation. This makes it an excellent choice for our project. In the future, this framework could help us tackle more complex computer vision tasks. These tasks could enhance our application by providing outputs like segmentation masks of lesions or performing melanoma skin cancer detection on multiple lesions in a single image.

Last, but not least, the Model Zoo is a key feature of OpenMMLab, offering a comprehensive collection of pre-trained models that we can use out-of-the-box. The Model Zoo is not just a repository of models, but also a benchmarking platform. Each model in the zoo comes with detailed performance metrics, allowing us to compare different models and methods under the same conditions. This feature is particularly useful if we want to quickly prototype solutions or benchmark our own models against state-of-the-art methods. By providing access to a wide variety of pre-trained models, the Model Zoo accelerates our research and development in computer vision, making it easier for us to build upon existing work and contribute to the advancement of the field.

OpenMMLab's framework platform is meticulously documented and annually hosts conferences that I used as guiding users on how to effectively utilize the toolkit. Unfortunately the information provided in them was often insufficient. A significant challenge I encountered was that many of the detailed learning resources were only available in Chinese. This language barrier required me to invest a substantial amount of time in translating the materials into English, which was a demanding but ultimately rewarding process.

Recognizing the potential benefit to the broader community, I am considering contributing to the project by translating and releasing some of my notes in English. This initiative could make the platform more accessible to non-Chinese speakers and potentially foster a more diverse user community.

3.2.1 PyTorch

As the OpenMMLab framework is built upon PyTorch it was also important to learn and understand the underlying framework as well.

PyTorch (PyTorch, 2023), an open-source platform developed by Facebook, provides tools to construct, train, and deploy deep learning models. It also offers modules for model training, including "torch" for numerical computation, "autograd" for automatic differentiation, "nn" for building neural networks, and "optim" for optimization algorithms. PyTorch's tensor data structure can represent multi-dimensional arrays, enabling various computations on different hardware backends. Its automatic differentiation feature simplifies the computation of gradients, making it more efficient.

PyTorch also provides a functional library, "torch.nn.functional," for building neural networks and implementing computational functions like linear operations, convolution, pooling, non-linear activation functions, and normalization techniques. The "nn.Module" class defines a generic interface for neural networks, allowing for the construction of layers and the definition of forward functions. PyTorch's torch.optim module provides optimization algorithms, and torch.utils.data offers a unified dataset model and a data loader for multi-threaded preloading.

These features make PyTorch a powerful tool for deep learning and would help us facilitate the task of optimising our efficient neural network models.

3.2.2 MMPreTrain (formerly MMClassification)

MMPreTrain (OpenMMLab, 2023) is an open-source toolkit focused on image classification tasks. It supports various model architectures, including VGG, ResNet, ResNet, MobileNet, ShuffleNet, MobileOne and the latest visual transformers. It also provides support for academic datasets like

CIFAR and ImageNet, and offers a wide range of training algorithms and techniques. A very important aspect of the toolkit is that it is organized around configuration files, which define model structures, dataset usage, data augmentation, optimizer strategies, learning rate policies, and runtime environment configurations. It also supports the Model Zoo with many pre-trained models and a set of tools for training, fine-tuning, and inference.

The toolkit's structure is divided into three components: the backbone network, the optional neck, and the classification head. The backbone network processes input images and produces a feature map, the neck further compresses the spatial dimensions of the feature map, and the classification head transforms the feature vector into class probabilities. The configuration file allows us to test various customization options, providing flexibility in defining and experimenting with our different models. The data configuration is specified under the "data" field, where we set the setting for out ISIC 2018 or ISIC 2020 datasets, and the data loading pipeline defines data augmentation techniques. The learning rate configuration initially is based on the stochastic gradient descent (SGD) with the support of additional learning rate strategies.

MMPreTrain also provides a set of tools that leverage the configuration files and pretrained models for training or fine-tuning. In our case, these tools are of use for performing inference or for fine-tuning the model on one of our lesion datasets. For example, once a model is fine-tuned on one of our datasets, we can use it to make predictions on new data (images of lesions never before seen during training). The training tools internally utilize the configuration files we mentioned earlier. Based on these configuration files, the model is dynamically constructed as a PyTorch nn.Module. The data loaders are created using PyTorch's DataLoader, and the optimizer is initialized. Once these components are prepared, they are placed into a runner, which is responsible for executing the gradient descent process and managing the overall workflow.

One of the powerful features of MMPreTrain and other OpenMMLab toolkits is the ability to modify configuration files using inheritance. This allows us to create new configurations based on existing ones, making it easier to experiment with different settings without having to rewrite entire configuration files. For example, if a user wants to experiment with a different backbone architecture while keeping the rest of the settings the same, we can create a new configuration file that inherits from the original one and only override the backbone setting. This might look something like this:

```
    # Config File Inheritance
_base_ = ['./mobilenet-v3-large_8xb128_in1k.py']

model = dict(
    backbone=dict(
        type='MobileNetV2', #This is an example
        # other parameters for the new backbone
    ),
    head=dict(
        num_classes=2,
        loss=dict(
            class_weight=[25689/26161, 472/26161],
```

```
        ),
        topk=(1, )
    )
# other parameters of the config
```

In this code example `Config File Inheritance`, the new configuration file inherits all settings from 'original_config.py' and only changes the type of the backbone. This feature greatly enhances the flexibility and efficiency of the toolkit, making it easier for us to perform customisations.

For our use case, we can also set the number of classes to 2 as we want to classify between benign and malignant, and we update the 'class_wright' parameter as our malignant is less represented in the ISIC 2018 dataset.

Finally, MMPreTrain provides a Python API for inference tasks, allowing users to build a model by specifying the path to the configuration file and the path to the pre-trained model (checkpoint). The API also supports performing inference on a large dataset, such as millions of images, by distributing the workload across multiple machines and GPUs. Furthermore, it provides a set of training tools that support distributed training using multiple machines and GPUs. Unfortunately, for the scope of this research project, we only have the ability to utilise one laptop-based GPU.

On the 7th of April, a significant shift occurred in the OpenMMLab ecosystem with the release of MMPreTrain, which superseded the existing MMClassification. This transition, while challenging, is a testament to the dynamic nature of software development, where constant evolution and adaptation are integral to progress.

The migration from MMClassification to MMPreTrain was not a trivial task to complete. As I promptly found out that it required a comprehensive understanding of the underlying architecture and functionalities of other underlying toolkits. The process involved adapting to new interfaces, reconfiguring existing workflows, and retraining models. However, these challenges were mitigated by the detailed migration guide (MMPreTrain, 2023) provided by OpenMMLab, which served as a valuable resource throughout the transition process.

3.3 Setting up the environment

Setting up the machine presented several challenges. These challenges primarily stem from the complexities of configuring a Linux environment within a Windows system and ensuring seamless interoperability between the two.

One of the primary challenges is ensuring that the NVIDIA drivers are correctly installed and configured to work with WSL. Another challenge is the installation and configuration of WSL itself. Thirdly, connecting WSL with VS Code also presents its own set of challenges.

Initially, it was imperative to confirm that the most recent Game Ready Nvidia driver is installed and that the Nvidia GPU in use is compatible with CUDA. We use Ubuntu 20.04 on the Windows Subsystem for Linux (WSL).

We set up Anaconda for managing our Python environment and we install PyTorch as it is required in this environment. Then, create directories our desired project folders and we install MMEngine and MMCV using MIM, which are underlying for MMPreTrain. Finally, we install MMPreTrain from the GitHub source, and we verify the installation by downloading demo config

and checkpoint files and we run the inference script to test the structure works.

Downloading a Linux distribution and the ISIC dataset using unlimited mobile data presented significant challenges. The large size of these files, coupled with the often inconsistent and slower speeds of mobile data compared to wired connections, resulted in prolonged download times and frequent interruptions. This required constant monitoring and repeated attempts to successfully complete the downloads, making the process tedious and time-consuming.

Despite all these challenges, the benefits of using WSL with CUDA and VS Code are significant. This setup allows developers to write and test Linux-compatible code on their Windows machines, leverage the power of CUDA for GPU-accelerated computing, and utilize the robust features of VS Code for code editing and debugging. With careful configuration and troubleshooting, these challenges can be successfully overcome.

For more details instructions, please, check the 'mmpretrain_prepare_environment.ipynb' from the source code of this project.

3.4 Applying Transfer Learning Methodology to Efficient Neural Networks

Transfer learning (Serokell, 2022) is a machine learning method that leverages a pre-trained model as a foundational basis for training a new model. It is particularly beneficial when for our new task the pre-trained model has already captured sufficient feature data. In this way, we employ the knowledge gathered from the pre-trained model, reducing the need for a large number of epochs in fine-tuning, and thus speeding up the training process (Spiceworks, 2023).

The prime advantages of transfer learning include conserving resources, enhancing efficiency, and potentially achieving superior performance when training new models. This technique is commonly applied in computer vision and natural language processing tasks, which typically demand large datasets and significant computational power. For instance, in our project, we initially fine-tune our efficient models of choice to recognise skin lesions by using a "starting point" pre-trained general ImageNet model.

In fine-tuning (TensorFlow, 2021), we use a pre-trained model as a starting point and then tweak its parameters to suit the new task. This process typically involves freezing the initial layers of the model, preserving their learned information, and training only the final layers, which are specific to the new task.

Checkpoints (Sato et al., 2020) play an essential role in transfer learning. They are snapshots of a model's state during the training process, stored in non-volatile memory. Checkpoints serve to save the current state of the model, enabling training to be resumed from a specific point. Additionally, checkpoints can be used in transfer learning, where an intermediate model state is used as a base, and then trained towards a different goal.

This project is a practical example of transfer learning in action in the context of classifying skin lesions, such as distinguishing between benign or malignant categories. In this application, transfer learning techniques are applied to our convolutional neural networks (CNNs) of choice, as this significantly speeds up our research, and we can more quickly reach a conclusion.

An integral part of this project involved fine-tuning a model first on the ISIC 2018 dataset, which serves as a stepping stone to handling the larger and more complex ISIC 2020 dataset.

Upon achieving satisfactory results, a checkpoint of the best-performing model on the test data set is chosen and being saved. This checkpoint preserved the model's state, which had already learned significant features from the ISIC 2018 dataset, and we use it as a strong foundation for further training.

The model was then fine-tuned on the ISIC 2020 dataset, which has a more extensive and diverse collection of skin lesion images. The rationale behind this sequential approach is based on the concept of curriculum learning (Bengio et al., 2009). In this context, the model first learns to recognize patterns and features on a simpler, more manageable task before being exposed to a more complex task. Using the best checkpoint from the previous fine-tuning step as a starting point, the model is endowed with a prior understanding of salient features, thus potentially enhancing its ability to learn from the more extensive dataset.

But how do we decide which model checkpoint to preserve? As our application is in the medical domain, it is of crucial importance to select the one that yields more true positive malignant predictions, as our aim is detecting the cancer lesion after all. Although, we should be aware that this decision is a balancing act and there has to be a trade-off. If we capture more true positive malignant, this might also mean that we would also have to deal with more false positives and this is a situation we want to avoid as we do not want to assign biopsy to false positive benign lesions.

In summary, transfer learning is a potent technique that enables us to develop efficient and accurate models more quickly, especially when dealing with enormous amounts of data. It has streamlined and enhanced machine learning capabilities, making model processing faster and more effective. This approach underscores the importance of curriculum learning in deep learning tasks, particularly when dealing with datasets of varying complexity and size. The use of checkpoints further fortifies this process, ensuring that the valuable learning from each phase is retained and utilized in subsequent stages.

Chapter 4

Experimental Design and Evaluation

4.1 Metrics

4.2 Results

4.2.1 Result for network 1

4.2.2 Result for network 2

4.2.3 Result for network 3

4.2.4 Result for network 4

4.2.5 Result for network 5

4.3 Evaluation review

Chapter 5

Practical Deployment of Web-based Application

Having successfully fine-tuned and evaluated a range of efficient machine learning models for skin lesion analysis, the task now shifts to an equally critical phase of the project - the development of a user-friendly interface. The completion of the AI models represents a significant milestone; however, the value of these models can only be fully realized when they are accessible and usable to the target audience. Thus, our next essential step is to build an intuitive and responsive web-based application that seamlessly integrates with our backend AI system.

In the subsequent sections, we will delve into the design process of the web-based application, key decisions, architecture design, API calls, and the main flow. We will elucidate how our user interface aims to harness the power of AI for skin lesion analysis, making it available at the fingertips of anyone who needs it.

5.1 Design Stage

The design and deployment of our web-based application, Skin Lesion Analysis, revolve around a user-friendly interface and efficient backend processing. The primary aim of the application is to provide a platform where users can upload images of skin lesions for AI-based analysis. The key decisions in the design process were influenced by the need to maintain simplicity for users while ensuring robustness in the backend where the image analysis is performed.

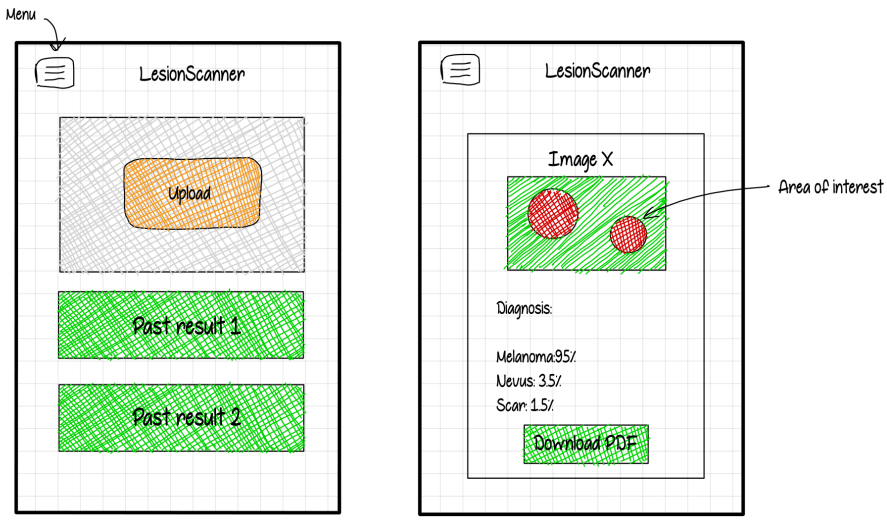


Figure 5.1: Wireframe of initial website design

The user interface (UI) design process (See Figure 5.1 began with wireframe sketches to visually guide the application's architecture. A well-structured navigation system was prioritized to ensure ease of use. The application has three main sections: Home, History, and About, accessible via a navigation bar. The Home page provides the primary functionality of the application – the ability to upload images for analysis. The History and About pages provide access to the user's analysis history and information about the application, respectively. The wireframe sketches proved invaluable in designing an interface that ensures a seamless user experience.

5.2 Client-Server Architecture

The architecture of the application was structured based on the React library, with React Router facilitating efficient routing between the various pages of the application. React's component-based architecture was beneficial in organizing the application's structure. For instance, the Navigation component handles the application's routing, while separate Home, History and About components hold the contents for their respective pages.

This interface will be served by a Flask backend, acting as a bridge between the user interactions on the frontend and the AI models on the backend.

Critical in the application is the API call to the backend server that performs the skin lesion analysis. The user selects an image, which gets stored in a local state variable using React's useState hook. When the user submits the image, an asynchronous function is called that uses JavaScript's fetch function to make a POST request to the server. The selected image is appended to a FormData object and sent in the body of the request.

Upon successful image analysis, the server responds with a JSON object containing the analysis result. The response is then processed, and the result is stored in another state variable. This result is conditionally rendered on the page, allowing the user to view the analysis result as soon as it becomes available.

5.3 Final Design and Analysing User Flow

The primary flow of the application starts at the Home page, where users are greeted with a warm welcome and are prompted to upload an image for analysis. Once the image is uploaded and submitted, the API call is made, and the result is presented to the user in a simple and understandable format. The user can then navigate to other sections of the application as desired.

This graph in Figure 5.2 shows the main interaction flow within the application. The user uploads an image from the Home page, which is sent to the backend API. The API returns an analysis, which is then displayed on the Home page. The Navigation component allows the user to switch between the Home, History and About pages.

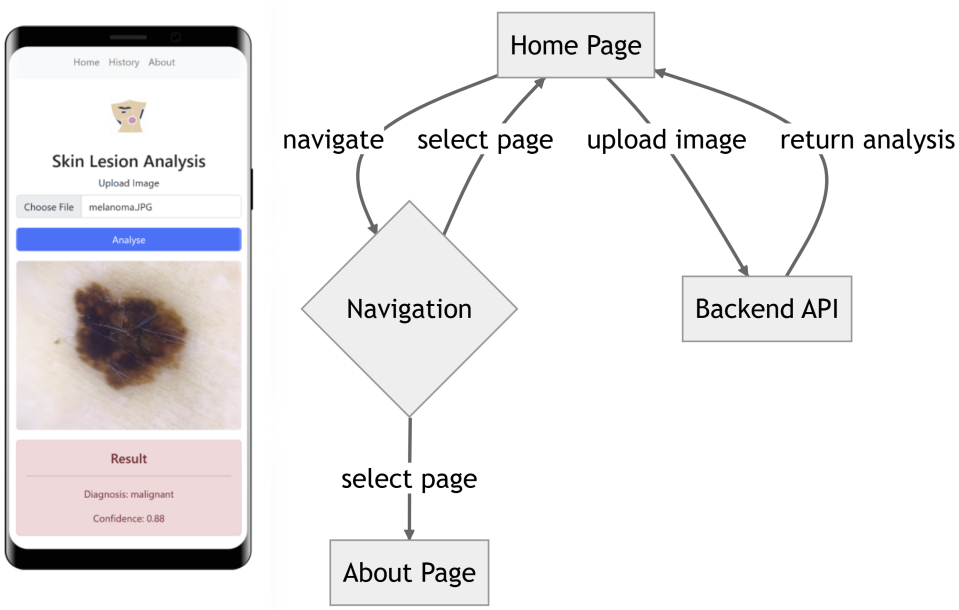


Figure 5.2: End Design and Primary flow of the application.

Bibliography

Anthony, E. and Bifulco, C. (2000). Early detection and treatment of skin cancer. *American family physician*, 62(2):357–368.

Anwar, A. (2019). Difference between alexnet, vggnet, resnet, and inception. https://towardsdatascience.com/the-w3h-of-alexnet-vggnet-resnet-and-inception-7baaaecccc96.

Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM.

Cloudera (2019). Deep learning in medicine: Classifying melanoma. https://community.cloudera.com/t5/Community-Articles/Deep-Learning-in-Medicine-Classifying-Melanoma/ta-p/249119.

Codella, N., Rotemberg, V., Tschandl, P., Celebi, M. E., Dusza, S., Gutman, D., Helba, B., Kalloo, A., Liopyris, K., Marchetti, M., Kittler, H., and Halpern, A. (2018). Skin lesion analysis toward melanoma detection 2018: A challenge hosted by the international skin imaging collaboration (isic). *arXiv preprint arXiv:1902.03368*.

Datahacker (2018). One layer convolutional neural network. https://datahacker.rs/one-layer-covolutional-neural-network/. Accessed May 28, 2023.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. *IEEE Computer Vision and Pattern Recognition*, pages 248–255.

Google (2017). Mobilenetv1 documentation. https://github.com/tensorflow/models/blob/master/research/slim/nets/mobilenet_v1.md.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Hinton, G. (2012). Rmsprop. https://golden.com/wiki/RMSprop. Adaptive learning rate optimization algorithm for mini-batch learning of neural networks.

Howard, A. G., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., et al. (2019). Searching for mobilenetv3. *arXiv preprint arXiv:1905.02244*.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

ISIC (2020). Siim-isic melanoma classification. https://www.kaggle.com/competitions/siim-isic-melanoma-classification/overview. Accessed: May 30, 2023.

ISIC (2023). Isic publications. https://www.isic-archive.com/#!/topWithHeader/tightContentTop/about/isicPublications. Accessed on May 30, 2023.

Kar, K. (2020). *Mastering Computer Vision with TensorFlow 2.x*. Packt.

Koelink, C. J., Jonkman, M. F., van der Meer, K., and van der Heide, W. K. (2022). Dermoscopy practice guidelines for use in telemedicine. *npj Digital Medicine*, 5(1):1–8.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Li, H., Xu, Z., Taylor, G., Studer, C., and Goldstein, T. (2018). Visualizing the loss landscape of neural nets. In *Advances in Neural Information Processing Systems 31*, pages 6389–6399.

Ma, N., Zhang, X., Zheng, H.-T., and Sun, J. (2018). Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV)*, pages 116–131.

Miller, R., Grewal, R., Bristow, I., and Langan, S. M. (2021). Dermoscopy use in primary care: a qualitative study with general practitioners. *BMC primary care*, 21(1):1–9.

MIT News (2021). An artificial intelligence tool that can help detect melanoma. `https://news.mit.edu/2021/artificial-intelligence-tool-can-help-detect-melanoma-0402`. Accessed on May 29, 2023.

MMPreTrain (Access on May 30, 2023). Mmpretrain migration guide. `https://mmpretrain.readthedocs.io/en/latest/migration.html`.

Núcleo ML (2019). Why sigmoid? `https://medium.com/n%C3%BAcleoml/why-sigmoid-ee95299e11fd`. Accessed May 28, 2023.

OpenMMLab (2023). Mmpretrain documentation: Dataset preparation. `https://mmpretrain.readthedocs.io/en/latest/user_guides/dataset_prepare.html`. Accessed on May 30, 2023.

OpenMMLab (Accessed on May 30, 2023). Openmmlab. `https://openmmlab.com/`.

OpenMMLab's Model Zoo (2016). Mmpretrain's model zoo listing of resnet. `https://mmpretrain.readthedocs.io/en/latest/papers/resnet.html`. Accessed on May 29, 2023.

OpenMMLab's Model Zoo (2018a). Mmpretrain's model zoo listing of mobilenet v2. `https://mmpretrain.readthedocs.io/en/latest/papers/mobilenet_v2.html`. Accessed on May 29, 2023.

OpenMMLab's Model Zoo (2018b). Mmpretrain's model zoo listing of shufflenet v1. `https://mmpretrain.readthedocs.io/en/latest/papers/shufflenet_v1.html`. Accessed on May 29, 2023.

OpenMMLab's Model Zoo (2018c). Mmpretrain's model zoo listing of shufflenet v2. `https://mmpretrain.readthedocs.io/en/latest/papers/shufflenet_v2.html`. Accessed on May 29, 2023.

OpenMMLab's Model Zoo (2019). Mmpretrain's model zoo listing of mobilenet v3. `https://mmpretrain.readthedocs.io/en/latest/papers/mobilenet_v3.html`. Accessed on May 29, 2023.

Papers with Code (2023). ImageNet Benchmark (Image Classification). `https://`

paperswithcode.com/sota/image-classification-on-imagenet. Accessed: 27th May.

PyTorch (Accessed on May 30, 2023). Pytorch documentation. https://pytorch.org/docs/stable/index.html.

Rotemberg, V., Kurtansky, N., Betz-Stablein, B., Caffery, L., Chousakos, E., Codella, N., Combalia, M., Dusza, S., Guitera, P., Gutman, D., Halpern, A., Helba, B., Kittler, H., Kose, K., Langer, S., Lioprys, K., Malvehy, J., Musthaq, S., Nanda, J., Reiter, O., Shih, G., Stratigos, A., Tschandl, P., Weber, J., and Soyer, H. P. (2021). A patient-centric dataset of images and metadata for identifying melanomas using clinical context. *Scientific Data*, 8.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2014). Ilsvrc 2014 results. https://image-net.org/challenges/LSVRC/2014/results. Accessed May 30, 2023.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. *arXiv preprint arXiv:1801.04381*.

Sato, R., Kato, H., Inoue, K., Inoue, K., and Matsuo, Y. (2020). A study of checkpointing in large scale training of deep neural networks. *arXiv preprint arXiv:2012.00152*.

Serokell (2022). Guide to transfer learning. *Serokell*.

Sharma, R., Kumar, S., Singh, H., and Kumar, R. (2021). An automated lightweight deep neural network for diagnosis of covid-19 from chest x-ray images. *Arabian Journal for Science and Engineering*, 46(11):12205–12216.

Shervine Amidi, Stanford University (2023). Cheatsheet: Convolutional neural networks. https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks. Accessed May 28, 2023.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Skin Cancer Foundation (2019). Skin cancer facts & statistics. https://www.skincancer.org/skin-cancer-information/skin-cancer-facts/. Accessed May 30, 2023.

Soenksen, L. R. (2020). Using deep learning for dermatologist-level detection of suspicious pigmented skin lesions from wide-field images. *Science Translational Medicine*, 12(529):eaax9113.

Song Ho Ahn (2018). Convolution. https://www.songho.ca/dsp/convolution/convolution.html. Accessed May 28, 2023.

Spiceworks (2023). What is transfer learning? Accessed on May 30, 2023.

Sun, L. H. (2023). Doctors found the world's smallest skin cancer spot: A woman's tiny mole. https://www.washingtonpost.com/nation/2023/05/09/skin-cancer-smallest-mole-oregon/. Accessed on May 30, 2023.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*.

TensorFlow (2021). Transfer learning and fine-tuning. https://www.tensorflow.org/tutorials/images/transfer_learning. Accessed on May 30, 2023.

Tsang, S.-H. (2018). Review: Inception-v4, evolved from googlenet merged with resnet idea, image classification. *Towards Data Science*.

Tschandl, P., Rosendahl, C., and Kittler, H. (2018). The ham10000 dataset, a large collection

of multi-source dermatoscopic images of common pigmented skin lesions. *Scientific Data*, 5:180161.

Ukwandu, O. (2022). An evaluation of lightweight deep learning techniques in medical imaging for high precision covid-19 diagnostics. *Computers in Biology and Medicine*, 140:105042.

Wu, Y. and He, K. (2018). Group normalization. In *European Conference on Computer Vision*, pages 3–19. Springer.

Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer.

Zhang, A., Lipton, Z. C., Li, M., and Smola, A. J. (2020). Dive into deep learning. `https://d2l.ai/chapter_convolutional-neural-networks/padding-and-strides.html#stride`. Accessed May 28, 2023.

Zhang, X., Zhou, X., Lin, M., and Sun, J. (2018). Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856.

Appendix A

Experimental Design and Evaluation

Steps to install and run this React application:

A.1 Pre-requisites for the Ract Application

1. **Node.js**: Install Node.js from the official site: https://nodejs.org/en/download/. Node.js is required to manage your application's dependencies and to run JavaScript tools and your React application.

2. **npm (Node Package Manager)**: npm is distributed with Node.js - which means that when you download Node.js, you automatically get npm installed on your computer. npm is a package manager for Node.js packages, or modules.

A.2 Installation Steps

1. **Navigate to the project directory**:

```

2. **Install the project dependencies**:

Use npm to install all the necessary packages the application needs:

```
npm install
```

3. **Start the project**:

Finally, start the application:

```
npm start
```

The application should now be running on your local machine, typically at 'http://localhost:3000'.

A.3 Note

This application is using an API at 'http://localhost:5000/analyze' to analyze the images. Ensure that this service is also running on your local machine, or update this URL to point to the correct location of this service.

A.4 Install Flask

Set up a Python server using Flask. Install Flask using pip:

Copy code: pip install flask

A.5 Run Flask

To run the server navigate to the folder where server.py is located and execute:

python server.py
```