BSc (Honours) Degree in Mathematics

Project

Identifying Melanoma Cancer using Convolutional Neural Networks and Image Recognition Techniques

By

Mahir Asef Ur Rahman

Project unit: U30219

Supervisor: Michal Gnacik

May 2021

Computing and Mathematics Programme Area

Contents

Chapter 1

Introduction

1.1 Literature Review

In modern days, the use of Artificial Neural Networks has been increasing more and more each day. These network models are exceptional when it comes to solving problems involving price regression, image classification and pattern recognition. These play a huge role in modern technology. Some of the main ones include cybersecurity, Cloud storage, Automated Robotics even AI and engines for games of various kinds [1]. The whole concept of making a model after the brain began in about 1943, by Warren McCulloch and Walter Pitts, by using electric circuits and using that to model the brain and how neurons essentially work [5]

The first ever perceptron model was created by Frank Rosenblatt, who was a psychologist. This was a simple model consisting of some input nodes, one node connected to the input and an output node after it. The output of the model would be either 0 or 1 depending on the threshold, which is now known as activation functions. Rosenblatt realised that the model performs better if more inputs are fed to the model as the weights become more and more optimised. This became the base idea of a neural network and later on would be expanded on.

The first real life application of neural networks would be shortly after, in 1959, which was used to reduce unnecessary noise during phone calls. Since phone calls were very new and rather poor in quality, this application was very useful. Modern iterations of these networks are still used today in noise suppression, noise cancelling and gain [5].

Ever since then, many scientists worked on the project to make better and multi layer neural network models which can be used to make useful prediction, find patterns within the data and analayse without human interaction. By then end of 1990, they started to come up with multi layered perceptrons (MLPs) and began to implement the technique of "backpropagation" and "Gradient Descent" within them. These techniques are the absolute essentials when it

comes to neural networks and is still used to this date.

1.2 Outline of the project

We will be first discussing the whole concept behind neural networks and convolutional neural networks (CNN) such as a visual representation of the structure and the mathematics behind the process. Afterwards, we will use neural networks on a dataset involving types of fashion ware to illustrate the process even further. Then, we will finally use the techniques (mainly CNN) to train our model to melanoma dataset and draw conclusions from the model while noting down any possible improvements to the model using statistical methods and other pre-trained models.

Chapter 2

What Are Neural Networks?

2.1 Neural Networks

Neural network is a system which is made up of many layers of nodes, which represents "neurons", densely connected to each other with some weights and biases. The model is then given many training data and the model tries to reduce the errors as much as possible by changing the weights and biases. It is said that the structure is modelled after the brain, hence why it is called Neural Networks.
A brief description of how they work is as follows: training data is first fed into the input layer nodes. They will contain some numerical values associated with the input data. Then, a weighted sum of each of the node is calculated and transferred into the nodes of the next layer, known as the hidden layer. An activation function is used to apply some transformation to the weighted sum before entering the hidden layer. After that, the output receives the weighted sum applied upon an activation function from the previous hidden layer and outputs the relevant results. The output is then compared with the actual output and using a cost function, the error is calculated. Then the weights are adjusted to have the lowest error using backpropagation and gradient descent.
Neural networks are exceptional when it comes to predicting certain features and finding patters given some datasets. These models can go through a huge dataset in a very short amount of time and identify key features, much faster than manually doing it by hand. Neural Networks are also very good at analysing complex data structure, such as images and speech patterns.
Neural Networks are very versatile when it comes to real life applications and uses. Some examples are predicting sales and prices in a business economics and management by analysing internal data, predicting the weather by analysing temperature, humidity etc, classifying images and objects which can be used for airport security, estimating stock market and more.

2.2 Modern Uses of NN

In the modern days, neural networks are one of the main focus in the machine learning field. It has many uses in our everyday lives. Here are some of the few uses of neural networks:

- Self-driving cars
- Algorithms for many shopping and entertainment websites
- Cybersecurity
- Game Analysis such as engines for Chess, Go etc.
- Automated Processes such as vehicle building etc.

Of course they aren't limited to just these. Nowadays, there is a constant need of AI and machine learning to automate processes that would otherwise be very time consuming. There are different type of neural networks used for different purposes which are discussed below.

2.3 Types of NN

There are various different types of neural networks to this date along with their individual uses. In this project, we will focus on the before-mentioned Convolution Neural Networks as this is mainly used for image recognition. Other useful types are,

- Long Short-Term Memory (LTSM): Mainly used for speech recognition
- Recurrent Neural Networks (RNN): Used in natural language recognition
- Auto Enconders (AE): Used for classification and feature compression
- Deconvolutional Network (DE): Used for image upscaling

There are many more types as more and more types of neural networks are now being modelled for a larger variety of work in the current times [13].

Chapter 3

Mathematics of Neural Networks

3.1 Structure of an NN model

In this project, we will be focusing on feed-forward neural networks.Feed-forward means that information travels in one direction (forward) from layer to layer and not backwards. A typical feed-forward neural network consists of many layers of nodes which are densely connected. Below (Figure 3.1) is a visual representation of a simple neural network .

The layers can be catagorised into three types,

1. Input layer: This layer is the beginning of the model where the data and information is first sent to this layer. The number of nodes in the input layer will depend on the given dataset. For example, it can depend on the number of features/predictors, x_i, in the case of a regression problem,

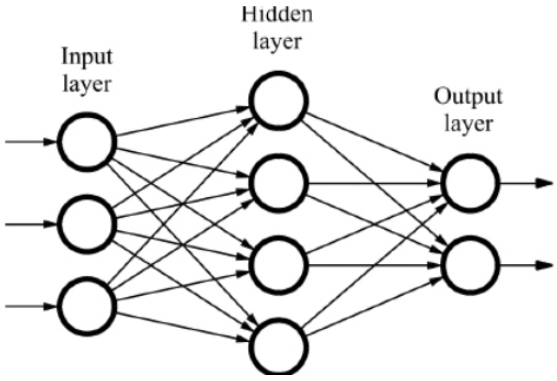


Figure 3.1: Structure of a Neural Network

or the shape of the image if the problem is image recognition and etc. All nodes from the input layer is connected to all the nodes in the next (hidden) layer.

2. Output layer: This is where the model gives out a prediction value. The number of the nodes in this layer depends on the type of problem at hand. In the case of a regression problem, there is only one node in the output layer. In classification, the number of nodes will depend on each of the class within the problem. Each of the nodes are connected to each individual nodes from the previous (hidden) layer.

3. Hidden layer: This type of layer comes in between the input and the output layers and there can be more than one hidden layers.The number of nodes can vary and can be changed to see if the model's accuracy improves. There can also be no hidden layers and such networks are called perceptions.

3.2 Activation Functions

Each hidden layer is given an activation function. The activation function takes the weighed sum of the input and transforms into a suitable output for the layer. The weighed sum can be written as

$$\sum(x \cdot weight) + bias$$

where x represents the input from the previous layer.
The activation function essentially speeds up the training process of the weights and biases. The activation functions must be differentiable as this is useful when going through backpropagation. We will denote the activation functions as g_i for hidden layer i and g_{out} for the output layer when going through the "Feedforward" process in the next section.
Few examples of different activation functions are listed below:

1. Identity $I(x)$ with range $\mathbb{R}$
2. Sigmoid/Logistic function $\sigma(x)$: $\frac{1}{1+e^{-x}}$ with range $(0, 1]$
3. tanh function $tanh(x)$: $\frac{e^x - e^{-x}}{e^x + e^{-x}}$ with range $(-1, 1)$
4. Softplus function $f(x) = \ln(1 + e^x)$ with range $(0, \infty)$
5. ReLu (Rectified Linear Unit) which can be written as

$$\begin{cases} 0 & if x \leq 0 \\ x & if x \geq 0 \end{cases}$$

with range $(0, \infty)$

Usually, we pick nonlinear activation functions (nonlinearities) as they allow the networks to compute non-trivial problems using only lower number of nodes [12].

3.3 Feeding Forward

Let $n \in \mathbb{N}$ be the number of predictors/features and $m \in \mathbb{N}$ be the number of output classes. Let $\mathbf{X} = (x_1, x_2, \dots, x_n)$ be the predictor values and $\mathbf{Y} = (y_1, y_2, \dots, y_m)$ be the response values.
Let $n_h \geq 0$ be the number of hidden layers present in the network let h_k be the number of nodes in the hidden layer k. For clarity, $h_0 = n$ and $h_{n_h+1} = m$.
Define matrix $W_k = [w_{ij}^{(k)}]$, where the entries represent the individual weights from node i of layer $k-1$ to node j of layer k. Next, define vector $\mathbf{B_k} = (b_1^k, b_2^k, \dots, b_{h_k}^k)$ for $k \in \{1, \dots, n\}$
Let g_i be the activation function present in the i-th layer for $i \in \{1, \dots, n\}$ as stated previously. Let $\mathbf{a^{(0)}} = \mathbf{x}$ so we have $\mathbf{a^{(0)}} = \left(a_1^{(0)}, a_2^{(0)}, \dots, a_n^{(0)}\right)$
Now, at each node k in the first hidden layer, we calculate

$$z_k^{(1)} := \sum_{j=1}^{h_0} a_j^{(0)} w_{jk}^{(1)} + b_k^{(1)}$$

$$a_k^{(1)} = g_1\left(z_k^{(1)}\right)$$

If we let $\mathbf{Z^{(1)}} = \left(z_1^{(1)}, z_2^{(1)} \dots, z_{h_1}^{(1)}\right)$ and $\mathbf{A^{(0)}} = \left(a_1^{(1)}, a_2^{(1)} \dots, a_{h_1}^{(1)}\right)$, we can write the above computation the following way,

$$\mathbf{Z^{(1)}} = \left(\mathbf{A^{(0)}}\right)^T \mathbf{W_1} + \mathbf{B_1}$$

If we iteratively calculate the next layer until the output layer, we can generalise the computations for layer l

$$z_k^{(l)} := \sum_{j=1}^{h_l - 1} a_j^{(l-1)} w_{jk}^{(l)} + b_k^{(l)}$$

$$a_k^{(l)} = g_l\left(z_k^{(l)}\right)$$

If we use similar consistent way of denoting $\mathbf{Z_l}$ and $\mathbf{A^{(l)}}$, we get

$$\mathbf{Z^{(l)}} = \left(\mathbf{A^{(l-1)}}\right)^T \mathbf{W_l} + \mathbf{B_l}$$

We obtain the output

$$\hat{y_k} = a_k^{(n_h+1)}$$

where $k \in \{1, 2, \dots, m\}$. The weights initially are just guessed with random weights and biases during the feedforward process. The weights and biases are then altered in a way so that the "cost/error function" is minimised during backpropagation using gradient descent method in the next section.

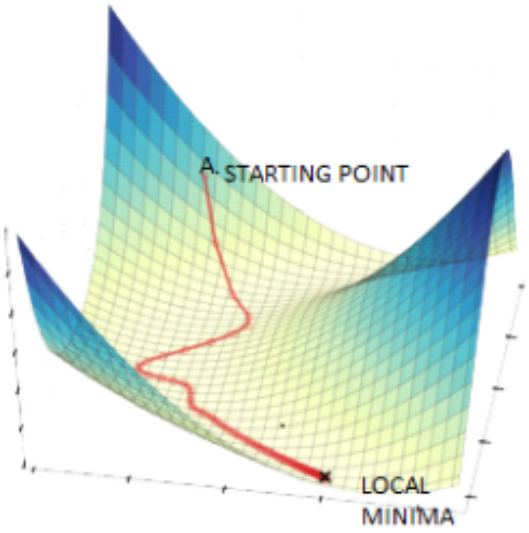


Figure 3.2: Gradient Descent

3.4 Backpropagation

In backpropagation, the goal is to minimise the "cost/loss function" of the given problem by adjusting the weights accordingly. The technique used is gradient descent method. Figure 3.2 is to visualise the process of minimising the cost function above [4].
There are many different cost functions, depending on the type of problem. For simple generalisation, we will consider the following cost function

$$C = \sum_{i=1}^{m} (\hat{y}_i - y_i)^2$$

For gradient descent, the following are computed $\frac{\partial C}{\partial w_{ij}^{(k)}}$ and $\frac{\partial C}{\partial b_i^{(k)}}$ where i represents the nodes from the previous layer $k-1$ and j represents the nodes from the next layer k and $k \in \{1, 2, \ldots, n_h + 1\}$
The process is called backpropagation as we start on the last layer, work out the derivatives and go to the previous layer and iterate until at the first layer.
Since C is a function of multiple variables, the partial derivative with respect to weights will require chain rule. Let's look at the last layer,

$$\frac{\partial C}{\partial w_{ij}^{(n_h+1)}} = \frac{\partial C}{\partial z_j^{(n_h+1)}} \frac{\partial z_j^{(n_h+1)}}{\partial w_{ij}^{(n_h+1)}}$$

Now, using z_k-s we found on the last section, we can see that,

$$\frac{\partial z_j^{(n_h+1)}}{\partial w_{ij}^{(n_h+1)}} = a_i^{(n_h)}$$

Since $\hat{y_k} = a_k^{(n_h+1)} = g_{out}\left(z_k^{(n_h+1)}\right)$, we can find that,

$$\frac{\partial C}{\partial z_j^{(n_h+1)}} = 2\left(\hat{y_i} - y_i\right) g'_{out}\left(z_j^{(n_h+1)}\right)$$

Altogether, we have

$$\frac{\partial C}{\partial w_{ij}^{(n_h+1)}} = 2a_i^{(n_h)}\left(\hat{y_i} - y_i\right) g'_{out}\left(z_j^{(n_h+1)}\right)$$

Now if we consider for $k \in \{1, \ldots, n_h\}$, we get

$$\frac{\partial C}{\partial w_{ij}^{(k)}} = \frac{\partial C}{\partial z_j^{(k)}} \frac{\partial z_j^{(k)}}{\partial w_{ij}^{(k)}}$$

Similar as before, we know that,

$$\frac{\partial z_i^{(k)}}{\partial w_{ij}^{(k)}} = a_i^{(k-1)}$$

Once again, by applying chain rule,

$$\frac{\partial C}{\partial z_j^{(k)}} = \sum_{l=1}^{h_k+1} \frac{\partial C}{\partial z_l^{(k+1)}} \frac{\partial z_l^{(k+1)}}{\partial z_j^{(k)}}$$

Since,

$$\frac{\partial z_l^{(k+1)}}{\partial_j^{(k)}} = g'_k\left(z_j^{(k)}\right) w_{jl}^{(k+1)}$$

we get,

$$\frac{\partial C}{\partial w_{ij}^{(k)}} = g'_k\left(z_j^{(k)}\right) a_i^{(k-1)} \sum_{l=1}^{h_k+1} \frac{\partial C}{\partial z_l^{(k+1)}} w_{jl}^{(k+1)}$$

for $k \in \{1, \ldots, n_h\}$
For the biases, first note that

$$\frac{\partial z_j^{(k)}}{\partial b_i^{(k)}} = 1$$

Then we can get

$$\frac{\partial C}{\partial b_i^{(k)}} = g'_k\left(z_j^{(k)}\right) \sum_{l=1}^{h_k+1} \frac{\partial C}{\partial b_i^{(k)}} w_{jl}^{(k+1)}$$

for $k \in \{1, \dots, n_h\}$. And for the final layer,

$$\frac{\partial C}{\partial b_{ij}^{(n_h+1)}} = 2\left(\hat{y_j} - y_j\right) g'_{out}\left(z_j^{(n_h+1)}\right)$$

3.5 Training the model

Let (X, Y) be a set of training datasets with N entries i.e. $X = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n, Y = (y_1, y_2, \dots y_n) \in \mathbb{R}^m$. In this case, our cost function becomes the following

$$\frac{1}{N}\sum_{i=1}^{N} \parallel \hat{y_i} - y_i \parallel^2 = \frac{1}{N}\sum_{i=1}^{N} C_i$$

To perform Gradient Descent, we are required to minimise the above expression. We update the weights and biases in the following way using the results in the backpropagation section,

$$w_{ij}^{(k)*} = w_{ij}^{(k)} - \frac{\alpha}{N}\sum_{l=1}^{N}\frac{\partial C_l}{\partial w_{ij}^{(k)}}$$

$$b_{i}^{(k)*} = b_{i}^{(k)} - \frac{\alpha}{N}\sum_{l=1}^{N}\frac{\partial C_l}{\partial b_{i}^{(k)}}$$

where $\alpha > 0$ is called the "learning rate". α cannot be too large (may end up jumping between edges of the minimum) or too small (longer computations/iterations). $w_{ij}^{(k)*}$ and $b_i^{(k)*}$ are the new weights and biases.
This process is repeated until we reach as close to the minimum as possible. Each iteration is called an "epoch".
There is an alternative to the Gradient Descent Method, which is known as "Stochastic Gradient Descent (SGD)" where the dataset is trained to the model in mini-batches rather than the whole data. The advantage of using SGD compared to normal GD is that the computation is faster at the cost of the minimum not being as small as GD would otherwise achieve.

3.6 Cross-Validation

This technique will be mentioned multiple times during the report as this is a very important technique in machine learning in general but essential in neural networks when training a certain dataset. Cross-validation means taking parts or batches of the training data and use that as testing the model. There are various reasons of doing this but the most important one is to prevent overfitting the model to the given training data. If there is a large training dataset, often times the model "memorises" the training data and its output rather than learning the patterns and information. As a result, when it is given a brand

new testing data point, the model would end up guessing the output instead of matching patterns.
Using some of the training data to test is a good way to prevent that. One other reason is that sometimes data is not always available and so we can use the training data to test beforehand so we know what to expect when the model is tested against true testing datasets.

Chapter 4

NN in Image Recognition

A typical neural network as described in the previous section can be used to identify and recognise gray-scale images. The next section covers an example of image recognition using dense neural networks which is used to classify different type of fashion ware and clothing. The training and testing datasets are taken from tensorflow keras datasets and is called "fashion_mnist".
The network works by having its input nodes equal to the size of the image (e,g in this case, it is 28×28) and does the process of feedforward and backpropagation. However, if we are to classify coloured images, dense neural networks start to become less and less effective at identifying patterns. This is because a coloured image effectively is made up of three layers (RGB). For these types of problems, we introduce a new set of layers to the network called "convolutional layers", which consists of filters. More on this will be explain on the next sections.

4.1 Example: Fashion mnist dataset

The training and testing is done using python on google colab. We train the dataset using purely dense layers. First, We extract the data from the tensorflow keras website. The number of classes is 10. Here are the list of classes:

1. T-shirt
2. Trouser
3. Pullover
4. Dress
5. Coat
6. Sandal
7. Shirt

8. Sneaker
9. Bag
10. Ankle Boot

Below is an example gray-scale image of a sneaker taken from the training dataset.

4.2 Training and testing the model

The training dataset is denoted as "x_{train}" and the test dataset "x_{test}". Before we make the model, we preprocessed our data in the following way

$$X_{train} = x_{train}/255$$

$$X_{test} = x_{test}/255$$

This step is essential in speeding up the training process. Before, the pixels would take values in the range $0 - 255$. After the preprocessing, the range is $0 - 1$ and this makes training much more efficient.
The next step is to work out the shapes of each of the image. It turns out to be 28×28 array gray-scale images, which means each pixels have values between 0 and 255. This means our input layer of the network will take the same shape as the images. The layer is then "flattened" into a vector shape with $28^2 = 784$ input neurons.
Now we build our network. In this particular example, we have input layer, two hidden layers and an output layer.

1. Input layer: Consists of 784 nodes which is flattened
2. Hidden layers: Both layers each consists of 100 nodes and is equipped with activation function "rectified linear unit (ReLu)".
3. Output layer: Consists of 10 nodes, same number as the classes

Here is a model summary printed by the code below

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 28, 28)]          0
_________________________________________________________________
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 100)               78500
_________________________________________________________________
dense_1 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 89,610
Trainable params: 89,610
Non-trainable params: 0
_________________________________________________________________
```

Now, we train the model. For the optimizer function, "Adaptive Moment Estimation (ADAM)" was chosen for this particular example. This is faster and much more efficient compared to typical Gradient Descent due to the input data being rather simple. For the cost function, we chose "mean squared error (MSE)", which will be minimised during backpropagation and lastly, we set number of epochs equal to 10. Too high of epoch may overfit the model to the training data and may not perform as well when predicting on the test data.
After the training, the model is then fed the test data and the accuracy of the prediction is recorded. The model turns out to be 87.83% accurate at classifying the test dataset.
The result can be further improved by using different weight parameters and optimizers etc. Another way is using multiple "convolutional layers" on top of the dense layers. The addition of the convolutional layers resulted in an impressive 99% accuracy on the testing dataset, which is a major improvement.

4.3 Comparing results with KNN

In this section, we compare the model to KNN, which is another common classification model. KNN, known as K-Nearest Neighbours, is a classification model in which the class of a testing point is determined by the nearest (i.e euclidean distance) neighbouring points and taking the majority class . The number of neighbours can be controlled and is labled k.
First, we use "GridSearch" technique, where we give the model some parameter space (i.e k) which gives the best accuracy. We try parameter space $k = [1, 30]$. We find out that the optimal k value is 7. We will use this in our comparison

table. In this comparison, we used 5 different k values and fit our image training data each time. We print out the testing accuracy afterwards. Below is the table of the different accuracy values.

k	**Accuracy %**
3	85.5
7	85.58
11	0.8514
15	0.8464
19	0.8434

When k is 7, we see the highest accuracy of 85.58%. If we compare this to our original dense network, we see that neural networks comes on top with 2.25% increase. Furthermore, we have mentioned that convolutional layers can up the accuracy even more. Also, KNN took longer to train and test compared to our network by utilising GPU. So we can say that for image recognition, neural networks are significantly better than other classification techniques in terms of result and time.

Chapter 5

Convolutional NN

5.1 What is CNN?

A Convolutional Neural Netowrk, or CNN for short, consists of one or multiple convolutional layers. These convolutional layers as especially useful in image recognition as compared to dense layers, convolutional layers are able to detect subtle differences between images very quickly because they detect patterns and information locally, rather than globally. With multiple convolutional layers, it is much faster to anlayse an image and work out patterns between multiple images compared to a dense layer.
For example, suppose there is an image of an animal (i.e cat). A convolutional network will "search" locally, As a result, it will be able to pick up and identify important features of the given animal (i.e. eyes, ears, face, nose etc). Convolutional network is also very useful in coloured images for the same reason: it is faster and even more reliable when it is compared to a dense network.
It is called "convolutional" layer as it performs a convolution operation with the input image and another smaller n×m pixels of image called a kernel or a filter.

5.2 History of CNN

It is stated that in 1958, an experiment took place by David Hubel and Torsten Weiesel where they studied how the eyes and pupils differed when different images are seen and how the cerebral cortex (the neurons) changes. They performed this on a cat and it was shown different images, each having unique features, like shapes, objects and other things. They realised that neurons fire differently depending on what people see. That is what known as the concept of receptive field. This is when they proposed the idea of finding a model that can represent the exact idea into a model. In 1984, a model of receptive field was introduced and was the first convolutional network to be made. This was a basic convolution layer. It consisted of two parts: receptive field to study the given image and extract key information and features of the image[9].

Now, the convolutional layers use very similar ideas but much more sophisticated for better results. There are many parts to a convolutional layer which will be discussed in the later sections.

5.3 Filters, the effects and Feature Maps

A filter, or kernels, is a typical n×m pixels of "patterns", much smaller than the actual image. In a convolutional layer, a typical size of a filter is 3×3 or 5×5 depending on the input image size. The filters contain array of numbers which represent the pixels. The numbers tell information about each pixel (i.e. RGB values).
Let's assume the input images are 28×28 pixels and our filters are 3×3. The filters are then "slid" across the 28×28 image, covering a 3×3 section of the image each time. While that happens, a convolution operation is performed between the input and the filter, which is equivalent of doing a dot product between vectors. The output is stored into what is called a "Feature Map", which is n×m pixels or information. Each time the filter slides over the input image, a number takes place in the feature map. So the size of it depends on the size of the image, filter, and "strides", which will be covered later [16].
Below (figure 5.1) is a diagram which illustrates the above process visually.
Different filters will have different effect on the input image. There can be major/minor transformation depending on the values of the filter and their size. One of the most useful filters is for edge-detection, which is used to find patterns within the outline of objects, which can be essential for classifying certain objects. A typical matrix for such a kernel is

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & -4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Performing a convolution of input image with the filter will make it so we can focus more on the edges of the given object which can help identify it easily. Another useful kernel to use is sharpen. This will make the input image more pronounced. The colours and pixels will be easier to differentiate from its surroundings. Below is a typical filter matrix for sharpening

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

There are many other complex filters that are used in much more detailed images. Each of them have their own uses depending on the properties of the images [15].

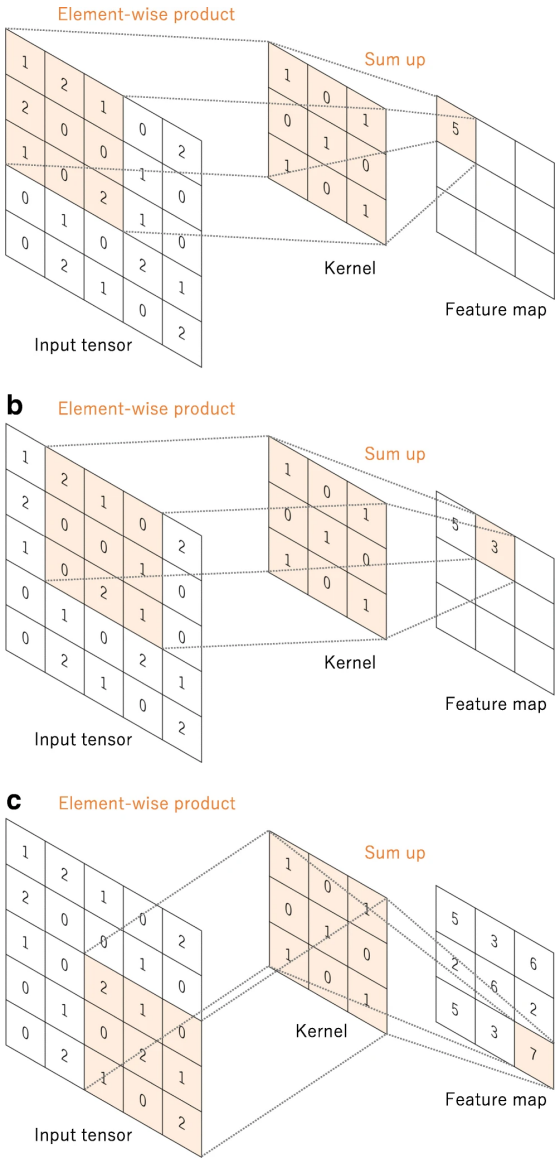


Figure 5.1: Filters and Feature Map Architecture

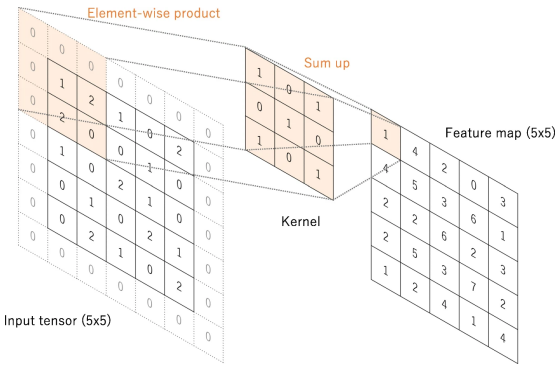


Figure 5.2: Padding Process

5.4 Padding

Padding is a technique that is used when we require the feature map size to be the same (or as close to) as the input image. We would want to do that as we want to capture as many important features as possible. In the above convolution process, the center of the filter does not cover the outermost pixels of the input image. That can cause important detail to be missed.
The most common padding is called "zero padding". The way it works is we add rows and columns of "0"-s to the borders of the input. This way, each and every pixel of the "true" input would be covered by the center of the filter and also will result in a larger feature map than we discussed previously [16]. Figure 5.2 shows this visually.

5.5 Strides

A stride is the number of rows/columns the filter can "skip" through while sliding across the input. This can help "downsample" the feature map. Pooling does a similar job but used more frequently.

5.6 Pooling Layer

In this layer, the feature map is "downsampled" to a smaller size, typically 2×2, while maintaining the most of features and information they captured prior to the process. There are many different type of pooling. The most common ones are max pooling, where the maximum values from the feature map is returned in the pooling layer, average pooling takes the average and so on.
The pooling layer is also used in combination with strides of 2. This greatly reduces the feature map and speeds up computation while not sacrificing any information contained within the feature maps.

5.7 The Mathematics and Process of CNN

A convolution is a mathematical operation which has notation "$*$". For a simple example of a convolution, let's use a simple case of two 3×3 matrices being convoluted. We will work out the $[2, 2]$ element of the convolution. We have,

$$\left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \right)_{[2,2]} = (i \cdot 1) + (h \cdot 2) + (g \cdot 3) + (f \cdot 4) + (e \cdot 5) + (d \cdot 6) + (c \cdot 7) + (b \cdot 8) + (a \cdot 9)$$

Since we are working with images (pixels), let I be our input image and F be our filter with dimensions $d_1 \times d_2$ [6]. Then,

$$(I * F)_{ij} = \sum_{m=0}^{d_1-1} \sum_{n=0}^{d_2-1} I(i-m, j-n) F(m, n)$$

Since this is the center value of the whole convolution, the other boundary values will be similarly weighted, just some values being shifted [15], In convolutional layers, we can have so that $I \in \mathbb{R}^{H \times W \times C}$ meaning input images with height H, width W and channel C(3 for RGB, 1 for grayscale). If we have D filters, then $F \in \mathbb{R}^{d_1 \times d_2 \times C \times D}$ and bias $b \in \mathbb{R}^D$ for each filter. Our convolution would then be

$$(I * F)_{ij} = \sum_{m=0}^{d_1-1} \sum_{n=0}^{d_2-1} \sum_{c=1}^{C} K_{m,n,c} \cdot I_{i+m, j+n, c} + b$$

This process takes place during forward propagation, when the training data is fed. The training images are convoluted with given filters and the output is extracted to a filter map, which is then passed to other dense layers after [6].

5.8 Backpropagation of CNN

An error function is then used to calculate how far we are from the original output pixel values given a prediction. For N predictions,

$$E = \frac{1}{2} \sum_{n=1}^{N} (\hat{y_n} - y_n)$$

After calculating the error, the weights (filter pixel values) and biases need to be altered so that the error function is at its minimum. After the weight adjustments, the convolution is performed between the filters and images again

to get the output. This is very similar process to a dense neural network.
In order to adjust the weights correctly, we need to calculate the following for layer l,

$$\frac{\partial E}{\partial w^l_{m,n}}$$

as well as

$$\frac{\partial E}{\partial x^l_{i,j}}$$

for each filter and input of the images, where $x_{i,j}$ represents the input of the images from node i to j. The chain rule applies here as well. We must perform derivatives while taking into consideration that these functions have convolutions within them [6].
Once these are calculated, we can find the minima of the error function using similar method such as GD or SGD by using the similar weight adjustment formula as stated in the neural network section.
After that, we can take the intended output and pass the feature map to the pooling layer.

5.9 Structure of CNN

The above procedure can be visually seen in the figure 5.3. In this example, there are 2 different convolutional layers. There are 5×5 filters with padding enabled with feature map of size represented under channels. The feature map is then taken into a Max Pooling layer of size 2×2 (taking the maximum weighted sums on the pixels from the feature map) and that makes up the first convolutional layer. The output is passed into another convolutional layer with similar structure but with different input layout to match the shape of the output from the previous convolutional layer. After the getting the output from the second layer, it is flattened into a vector of appropriate size, which is then passed into a typical densely connected neural network. There can be dropout of neuron connections to minimise overfitting of the model [10].

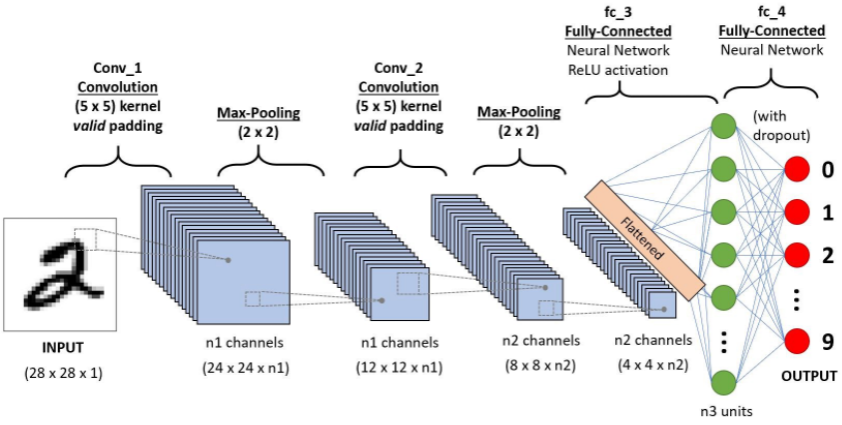


Figure 5.3: Structure of CNN

Chapter 6

CNN to Identify Melanoma Cancer

6.1 About Melanoma Cancer

Melanoma is the most dangerous skin related disease. It produces melanin on the skin cells which gives unusual colour to it. These can look like "moles" with unusual shape and/or colour and evolves into other shapes. This usually is the first main symptom of early melanoma cancer. The main cause of melanoma is exposure to UV light. A lot of people can get melanoma, especially elderly people with very sensitive skin. Melanoma has caused many deaths due to it being tricky to identify and the risk increases if the skin moles are left alone for too long [2]. Identifying if a person has melanoma or not using a image recognising model can greatly help prevent all the risks involved.

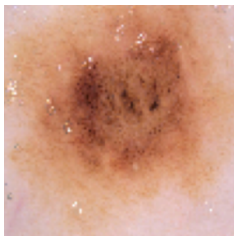
6.2 Implementing CNN to Identify Melanoma

Now, we can implement all the techniques mentioned so far to identify coloured images. We will be looking to create a model that will detect if a person has melanoma cancer or not. In summary, we will be feeding the model a big set of images of skin lesions of different patients. Afterwards, we will create a model containing convolutional layers and dense layers and test it against its testing dataset and see how the accuracy goes. Afterwards, we will discuss how the model can be improved further by changing parameters or techniques in general.

6.3 About the Dataset

This dataset (https://www.kaggle.com/drscarlat/melanoma) is a partition of a larger dataset taken from Kaggle, which was included during a challenge on-

line called "SIIM-ISIC Melanoma Classification" [7]. The images come from the databases of Society for Imaging Informatics in Medicine (SIIM) and International Skin Imaging Collaboration (ISIC), The competition took place in May 27, 2020 and ended on August 18,2020 with more than 4,000 competitors and 102,000 entries of code. The original dataset had a filesize of over 100 GB, around 44,000 images. In this project, only a fraction of the dataset was used (about 6 GB) as 100 GB of images can take up to many days to train and test without a proper machine .
The dataset used in this project contains about 17,805 images in total, including train,test and cross validation. Each image has dimmensions $100 \times 100 \times 3$, since they are coloured images, which is where convolutional layers come in handy. Below is a sample image of a skin lesion taken from the training dataset which was shown in google colab.



6.4 Training and Testing the model

The dataset file consists of three folders, containing training, testing and cross validating data in batches of 64 images. Similar to before, we denote x_{train}, x_{test} and $x_{validation}$ representing each datasets. Each image is prepossessed the following way,

$$X_{train} = x_{train}/255$$

$$X_{test} = x_{test}/255$$

$$X_{validation} = x_{validation}/255$$

Again, this speeds up the training process and is especially useful when working with a large dataset as this.
Below is a quick summary of the structure of the model. Since this is a coloured image, the input layer is a convolutional layer with 32 filters of size 3×3 with ReLu activation function.
There are two more convolutional layers are added after that as hidden layers with 64 and 32 filters respectively with the same size as the input layer. Each of the convolutional layers have set the pooling type is "MaxPooling" and is of size 2×2. We also transform each layers using "BatchNormalization" command, which normalises the output (i.e. mean = 0 and standard deviation = 1). Finally, each of the convolutional layers have a "Dropout" of rate 0.3. Dropout causes some of the connected neurons to be dropped if they don't contribute to the model as much as other neurons. This helps prevent overfitting the model,

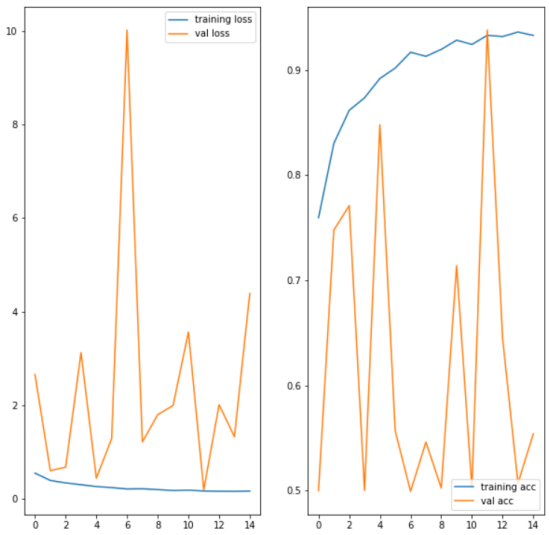


Figure 6.1: CNN Accuracy and Error for Training and Cross Validation

which can be dangerous with a big dataset such as this. All the layers are then flattened
After these layers, two hidden dense layers are added to connect with the previous convolutional layers. They have 64 and 32 nodes respectively with ReLu activation and dropout rate of 0.3. The outputs are also normalised.
Finally, the output layer is another dense layer with output shape of 2 since there are two classes in this problem with activation function SoftMax. We use ADAM optimizer and "categorical_crossentropy" error function.
This particular model, we give epoch of 15. While the model is training each epoch, we create a checkpoint which will keep track of model accuracy and validation accuracy. Since we want to prevent overfitting, cross validation accuracy is of interest. The checkpoint command always takes the epoch with the highest cross validation accuracy score. The model is then saved to be used in the testing dataset for the best result possible.

6.5 Report and Result

After running 15 epochs, we check the training score as well as the validation score and error/loss. We make a plot of score and error against the number of epochs (figure 6.1).

We are more interested in cross validation results to prevent overfitting to training model. From the graphs, we can see that the highest cross validation accuracy achieved is 93.81% and the corresponding error is 0.1846. This occurs on the 11^{th} epoch.
We then extract that model and apply it on the testing images. We get a testing

accuracy of **93.12**%.

6.6 Comments on Results

The model ended up being about 93% effective at identifying melanoma from skin lesions of different patients. While the number is fairly high, this is not a reliable percentage. Melanoma is a dangerous disease and if identified correctly and early, it is very possible to cure it. Finding a model with accuracy as close to 100% is necessary to prevent any patient deaths as much as possible. Any wrong identification can lead to more deaths as well as loss of time and money. There are many things we can do to the current model to drastically improve the model.

6.7 Hyperparameter Tuning

Hyperparameter Tuning or Hyperparameter Search is a technique in machine learning where a model is optimised by changing different model parameters within a given hyperparameter space and working out the model accuracy (or CV accuracy) for each combination of parameters. This can be used to find the best possible model in the given problem.
We can use "GridSearchCV" again from Scikit-Learn to control the parameters and compare different results. Below are some possible parameters in the given models:

- Learning Rate, Momentum and Decay on Gradient Descent
- Optimzer functions and loss functions
- Batch size and epochs
- Starting with different weights
- Adding/Reducing different layers

For our model, we use the hyperparameter search technique called "Keras Tuner" or "Hyperband Searching", instead of using "GridSearchCV" or "Random Search" that is mentioned above. The main difference between the two is that Random Search takes in every possible combination of a given parameter space. This is very inconvinient for our model since our parameter space is going to be extremely large. We will be looking for optimal number of convolutional layers and dense layers as well as their parameter settings such as their nodes, filter sizes and etc. Already, the parameter space is huge and performing a random search on it could take up to many days and we may not even find the optimal model even then as we should try other parameter spaces as well. Hyperband searching tries a parameter combination and fits the model with a number of given epochs. Afterwards, if the tuner sees that the model is not improving at all after one or two epochs (depending on the patience level), the

tuner scraps the combination and immediately moves onto the next parameter combination. If it finds a promising hyperparameters, then the tuner will focus most of its resources.
The algorithm that Hyperband search uses is called "Successive Halving". In this algorithm, the keras tuner uniformly puts all its resources on the given parameter space on all of the combinations. Afterwards, it evaluates the model with the given configuration with a fraction of the given total epoch. This gives the tuner an idea of "good" and "bad" configurations. The tuner then discards half of the parameter combinations that are the worst performing and keeps the other half. This process is iterated for good half of the configurations. This is much for appropriate and efficient since our dataset is big and the images are rather complex as well as our parameter space being large [8].
For our model, the parameter space is as follows: Up to 3 convolutional layers with filters number range of 32 to 256 with steps 32 and with filters size 3×3. ReLu activation function and max or avg pooling, up to 3 dense layers of nodes range 30 to 100 with steps of 10, dropout rate of 0 to 0.5 with 0.1 steps and finally the output layer is equipped with softmax activation function. The model optimizer is ADAM with learning rate of 0.0001 and loss function of "categorical_crossentropy". We then find the optimal model which has the best validation accuracy with given epoch 20. After running for about 60 trials and runtime of 4 hours, we find a model which is slightly worse than our original CNN model. The model consisted of two conv layers with 96 and 256 filters respectively, 3 dense layers of 100 nodes and max pooling for all the layers and 0.2 dropout rate. We get an accuracy of **92.45%** on the validation and **91.3%** on the testing dataset.
It might seem that we have performed lower, however we have not tested on many other possible parameter spaces. Using larger spaces at a time can make computations exponentially longer and without correct specifications, such as dedicated, well-built GPU, the whole process becomes extremely long and tedious. But, with more parameter space, there can be even more potential improvements to the model.

6.8 Fine Tuning Pre-trained Models

One other thing we can do is implement a pre-trained model to our network, which is also known as "Transfer Learning". A pre-trained model is essentially a model that has been trained beforehand on millions of images on the web. Naturally, they are very good and efficient at detecting small patterns within images that would be otherwise missed by a normal network.
The whole network wouldn't be used but only the convolutional base that is built within the model will be used plus our additional dense layers we have created previously. We tune the convolutional base within the pre-trained model to suit our given problem. Here are few examples of pre-trained models that are used in image recognition:

- VGG-16

- EfficientNetB7
- ResNet-50
- MobileNet V2

The issue with these networks is that they have millions of parameters within them. Hence, the training process can take a huge amount of time without freezing the parameters and without the right computer and equipment.
We now implement a pre-trained model to our problem to see what improvements we can make. We take the "MobileNetV2" model for this.
MobileNetV2 is a model made by google. The model is trained on millions of different images from the internet as well as other databases with a lot of class labels. The figures 6.2 and 6.3 gives the detailed structure of the model [11].

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Figure 6.2: Structure of MobileNetV2

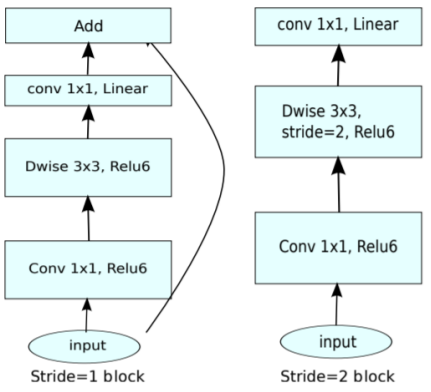


Figure 6.3: Structure of MobileNetV2

The whole idea behind this model, compared to its predecessor MobileNetV1, is that the use of inverted residual layers with bottleneck and it has much less trainable parameters. The idea of using such a layer is that the feature maps can be extracted with a much lower dimension, allowing faster processing [3].
Now, we make the model and train it to our melanoma dataset. We get the model from tensorflow keras [14]. We just want to use the base of the model hence we do not extract the "top" of the model. We give this the input shape of 96 (all images were scaled to 96×96 due to the nature of the model). Set the classification activation as softmax.
Then we freeze the base model as we do not want to retrain the already pre-trained model. So freezing it is essential. We also make it so that the model does a maximum pooling on the pre-trained model after the whole network.
After adding the base layer, we then add some standard densely connected networks. In our case, we simply added the previous dense layers as before from our CNN model.
Now, we set the learning rate to be a very small number (0.0001) so that we don't converge too quickly or model may end up overfitting. Choose ADAM as optimizer and CategoricalCrossEntropy as our loss function. We finally train the model. Below is a similar graph showing error and accuracy of training and cross validation datasets after running for 15 epochs on MobileNetV2 (Figure 6.4).

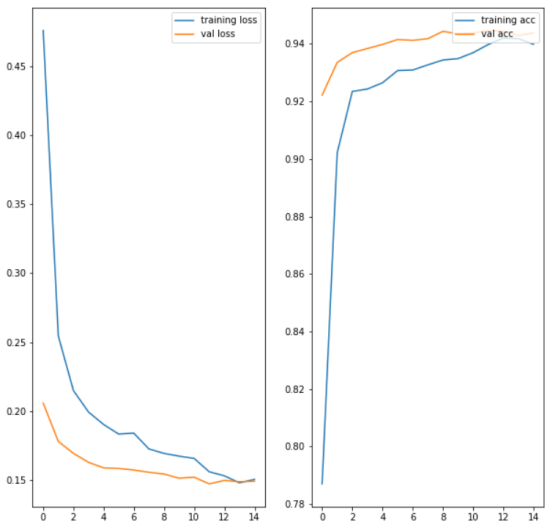


Figure 6.4: MobileNetV2 Accuracy and Error for Training and Cross Validation

We can see major differences in this model compared to our previous one. Firstly, the lines follow a smoother curve. This indicates that the model tends to not easily overfit to the data. Secondly, and most importantly, we see an improvement on the accuracy/error. The peak cross-validation accuracy is 94.46%,

which is a slight improvement from our previous model (93.81%). Hence, we performed better on the testing dataset as well with accuracy of 93.57, which is again a slight improvement from the previous model (93.12%). We ended up improving the model with less trainable parameters as before.
So, we can conclude that pre-trained models can in fact improve the model even with lower trainable parameters. If we tweak the original base, change the dense layers or even use a much more complex pre-trained model, then we can improve the model even further. In theory, the accuracy of 97% is very possible.

Chapter 7

Conclusion

Our model ended up improving to about 95% accuracy. This is a good starting point but we can do much better and this can impact our current society very differently. We could have tried to change the parameter space and use a pre-trained model and use the hyperband technique on that as well, which could possibly increase the accuracy by a lot.
Melanoma is currently a very dangerous disease. Luckily it can be cured during its early stages. According to Cancer Research UK, there are 16,200 melanoma patients every year in the UK alone and 2,300 of them die. This is especially dangerous for the elderly people.
Nowadays, thanks to our technology, melanoma is becoming less and less scary. However, with the help of AI, detecting melanoma cancer without human interaction is possible and hence the chances of having the disease cured increases greatly.
However, for this to be possible, there is a need for more data and images. Without them, we cannot reliably train models. This is only possible thanks to the people and the organisations that work towards the better future.
Networks and machine learning techniques are getting better and faster these days and this is exactly what they should be used for and not just melanoma, but for other real life, medical and business purposes. Neural nets and machine learning in general is going to be more dominant in the future where automation and AI shapes the world.

References

[1] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, Nachaat AbdElatif Mohamed, and Humaira Arshad. State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11):e00938, 2018.

[2] Mayo Clinic. Melanoma symptoms. `https://www.mayoclinic.org/diseases-conditions/melanoma/symptoms-causes/syc-20374884`.

[3] Luis Gonzales. A look at mobilenetv2: Inverted residuals and linear bottlenecks. `https://medium.com/@luis_gonzales/a-look-at-mobilenetv2-inverted-residuals-and-linear-bottlenecks-d49f85c12423)`.

[4] Capri Granville. Alternatives to the gradient descent algorithm. `https://www.datasciencecentral.com/profiles/blogs/alternatives-to-the-gradient-descent-algorithm`, 2019.

[5] Jaspreet. Concise history of neural networks. `https://towardsdatascience.com/a-concise-history-of-neural-networks-2070655d3fec`, 2016.

[6] Jefkine Kafunah. Backpropagation in convolutional neural networks. *DeepGrid—Organic Deep Learning, Nov*, 29:1–10, 2016.

[7] Kaggle. Siim-isic melanoma classification. https://www.kaggle.com/c/siim-isic-melanoma-classification.

[8] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1):6765–6816, 2017.

[9] YD Li, ZB Hao, and Hang Lei. Survey of convolutional neural network. *Journal of Computer Applications*, 36(9):2508–2515, 2016.

[10] Sumit Saha. A comprehensive guide to convolutional neural networks. `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`.

[11] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[12] Sagar Sharma. Activation functions in neural networks. `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6`.

[13] Andrew Tch. The mostly complete chart of neural networks explained. `https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464`.

[14] Tensorflow. tf.keras.applications.mobilenetv2. `https://www.tensorflow.org/api_docs/python/tf/keras/applications/MobileNetV2)`.

[15] Wikipedia. Kernel (image processsing). `https://en.wikipedia.org/wiki/Kernel_(image_processing)`.

[16] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do, and Kaori Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into imaging*, 9(4):611–629, 2018.