Construction of a Convolutional Neural Network

Application to Flower Image Recognition

Sohan Suchdev

Abstract

This paper explores the mathematical architecture and implementation of Convolutional Neural Networks (CNNs). It details the construction of a model designed for multi-class image classification. Key mathematical concepts, including matrix convolution, pooling operations, and backpropagation algorithms, are analysed in the context of optimising recognition accuracy.

Page Count: 20

1 Introduction

Artificial intelligence (AI) has become one of the most significant advancements, driving innovation across various fields, from healthcare to autonomous vehicles. Many AI advancements come from machine learning, a branch of AI that allows computers to learn from data and make predictions or decisions without being explicitly programmed. In machine learning, the development of neural networks, which are computational models inspired by the human brain's structure and function[1] to learn from data and make predictions, has been vital.

Among the different types of neural networks, Convolutional Neural Networks (CNNs) have proven to be exceptionally powerful, particularly in classifying images. CNNs are specifically designed to process and analyse visual data, making them powerful at object recognition, facial recognition, and even medical image analysis[2].

1.1 Importance of CNNs

Traditional image processing techniques relied heavily on manual feature extraction, where experts had to handcraft features like edges, textures, and shapes to feed into machine learning models. Features are specific, quantifiable attributes or patterns in the data that are relevant for solving a particular problem[3]. This time-consuming process led to sub-optimal results because it limited the model's ability to learn more complex patterns. CNNs, however, automate this feature extraction process, learning to identify patterns directly from raw data, such as simple edges in initial layers and complex shapes in deeper layers[4]. This ability to learn directly from data has made CNNs indispensable in fields ranging from self-driving cars, which need to recognize objects and obstacles in real-time, to healthcare, where CNNs assist in diagnosing diseases by analysing medical images with high accuracy comparable to human experts.

1.2 How Neural Networks Work

A neural network is a computational model that processes information by passing it through layers of interconnected nodes. It typically consists of three types of layers:

1. **Input Layer**: Receives the raw input data, such as an image represented as a grid of

pixel values[5].

2. **Hidden Layers**: Perform calculations and transformations on the input data, progressively identifying patterns and features[3].

3. **Output Layer**: Produces the final result, such as classifying the input[6].

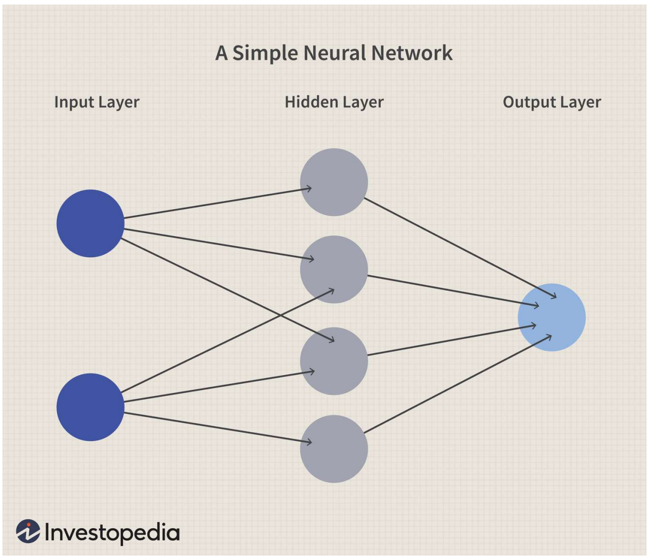


Figure 1: *A Simple Neural Network and its 3 Layers [7]*

The network uses hyperparameters to process the input and detect patterns. These hyperparameters determine how the input is transformed and directly influence the network's predictions. During training, the network learns to improve its predictions through a process called Gradient descent, which adjusts the hyperparameters to minimise the difference between predicted and actual outputs, measured by a Cost Function[8].

This optimisation is achieved using Backpropagation, a method where errors are calculated at the output layer and propagated backwards by adjusting the hyperparameters through the network to improve pattern detection and accuracy[9]. Through this iterative process, the neural network refines its ability to process inputs and classify them effectively.

A CNN functions similarly to standard neural networks but the key difference is the use of convolutional layers. Convolutional layers use filters that slide across the input image, performing convolutions to detect specific features such as edges, corners, and textures, allowing the CNN to learn more complex patterns and thereby increasing its accuracy[10].

1.3 Plan

For my Maths IA, I plan to build a Convolutional Neural Network (CNN) to classify flowers with a target accuracy of at least 95% on the test dataset while maintaining a low loss below 0.1 during training. My interest in image recognition, combined with the availability of a large dataset of flower images, makes this an ideal project since the dataset facilitates effective training. I will start by using a generic neural network template, which I will gradually enhance by incorporating convolutional layers to extract image features, as well as optimising the learning process through Gradient descent and backpropagation. Throughout the project, I will explore the mathematics of these components, including convolution operations, gradient calculations, and optimisation techniques, and investigate how to fine-tune the network for better performance.

2 Convolution Layer

2.1 Convolutions

A convolution is a mathematical operation that takes in 2 functions and outputs a third. It expresses the amount of overlap of one function g as it is shifted over another function f, which can be described as "blending" one function with another.[9] Convolutions allow us to filter images and as we can combine the original image function and a filter function, to result in a filtered image.[11] This is useful in the CNN model as we can detect patterns helping the neural network learn.

To understand a convolution, we will break down it with the analogy of a ball dropping. If we drop the ball from some arbitrary height, it will land a units away from the starting point with probability $f(a)$, where f is the probability distribution.
After this first drop, we drop it from a different height above the point where it first landed. The probability of the ball rolling b units away from the new starting point is $g(b)$, where g may be a different probability distribution.

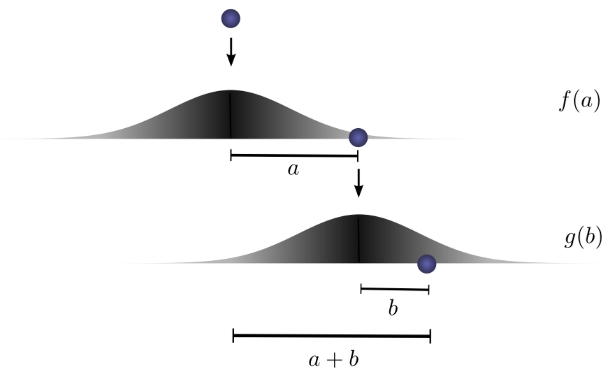


Figure 2: *Probability distributions of $f(a)$ and $g(b)$[12]*

If we say the ball travelling a total distance is c, where $a + b = c$, the probability of the ball rolling a distance c is $f(a) \cdot g(b)$.
For example, if the total distance $c = 2$ and if the first time it rolls, $a = 2$, the second time it must roll $b = 1$ to reach our total distance $a + b = 3$. The probability of this is $f(2) \cdot g(1)$. However, there are multiple ways to reach a total distance of 3. So, to find the total probability of the ball reaching a total distance of c, sum over the probability of each possible way:

$$\sum_{a+b=c} f(a) \cdot g(b)$$

This the convolution of f and g, evaluated at c which is defined as:

$$(f * g)(c) = \sum_{a+b=c} f(a) \cdot g(b)$$

If we substitute $b = c - a$, we get the standard definition of convolution:

$$(f * g)(c) = \sum_{a} f(a) \cdot g(c - a)$$

The convolution can be written as an integral as well, but here it is written as a sum as an image is discrete, rather than continuous.

Convolutions in higher dimensions

Convolutions can also be used in higher number of dimensions. With the example of dropping a ball, as it falls, its position shifts not only in one dimension but in two.

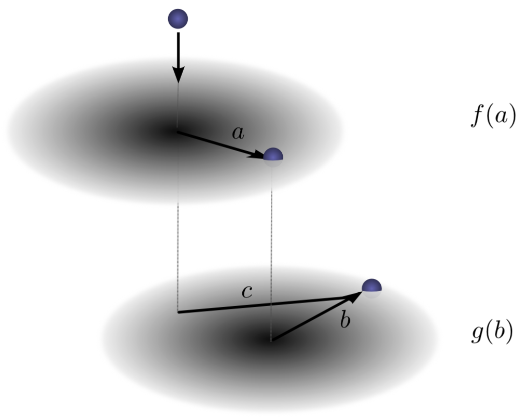


Figure 3: *Ball dropping in 3D [12]*

The convolution is the same as before but now a, b, and c are vectors where $\mathbf{c} = (c_1, c_2)$, $\mathbf{a} = (a_1, a_2)$, and $\mathbf{b} = (b_1, b_2)$. Therefore:

$$(f * g)(c_1, c_2) = \sum_{a_1, a_2} f(a_1, a_2) \cdot g(c_1 - a_1, c_2 - a_2)$$

Just like one-dimensional convolutions, a two-dimensional convolution is the same as "blending" one function with another, by sliding them over each other.

2.2 Convolutional layer

This occurs after the input layer and applies a set of filters, using convolutions, to the input image, transforming it into feature maps highlighting specific patterns in the data. The filters are small matrices that slide over the input image detecting features such as edges, textures, or specific patterns[13].

The conventional formula is:

$$(f * g)(c_1, c_2) = \sum_{a_1, a_2} f(a_1, a_2) \cdot g(c_1 - a_1, c_2 - a_2)$$

Where:

- $f(a_1, a_2)$ is the input image.
- g is the filter.

- (c_1, c_2) is the position in the output feature map where the convolution result is being computed.
- The sum is taken over all positions a_1 and a_2 that the filter covers.

Here, the filter g is reversed before it's applied to the image. This is because $g(c_1 - a_1, c_2 - a_2)$ represents a reversal of the filter's indices relative to the current position (c_1, c_2). Then reversed filter slides over the image $f(a_1, a_2)$, multiply corresponding elements (i.e., the pixel values under the filter and the filter values), and sum these products. The result of the sum gives you the value of the output feature map at position (c_1, c_2).

However, for the CNN model, the convolution formula is different and is as follows:

$$S(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} I(i+m, j+n) \cdot F(m,n)$$

Where:

- I is the image (input).
- F is the filter.
- $S(i,j)$ is the output feature map.

The main difference between the two formulas is in the handling of the filter g. The simplified formula $S(i,j)$ does not reverse the filter and applies it as is. This operation is technically a correlation rather than a true convolution because it doesn't reverse the filter.[10] In CNNs, this is overlooked because the filters are learned during training, and whether they are flipped or not doesn't affect the process. Therefore, the two formulas lead to equivalent operations in practice, so the simplified one is used to save computational resources.[11]

2.3 Filters

Two commonly used filters in CNNs are the normalized filter and the Gaussian filter. Both filters are designed to modify image regions while ensuring the overall intensity of the image remains consistent. To ensure that applying the filter preserves the overall brightness or intensity of the region it covers, the sum of the weights in the filter must equal 1.

Normalized Filter

A simple filter often used to average out the values in a specific region of the image. A 3×3 normalized filter could be represented as:

$$\text{Normalized Filter} = \frac{1}{9} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

To normalise the filter, the sum of all the elements in the matrix is computed:

$$1+1+1+1+1+1+1+1+1=9$$

To ensure that applying the filter preserves the overall brightness or intensity of the region it covers, the sum of the weights in the filter must equal 1. Therefore, each element is divided by the total sum of 9, giving the normalized filter as:

$$\frac{1}{9} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$$

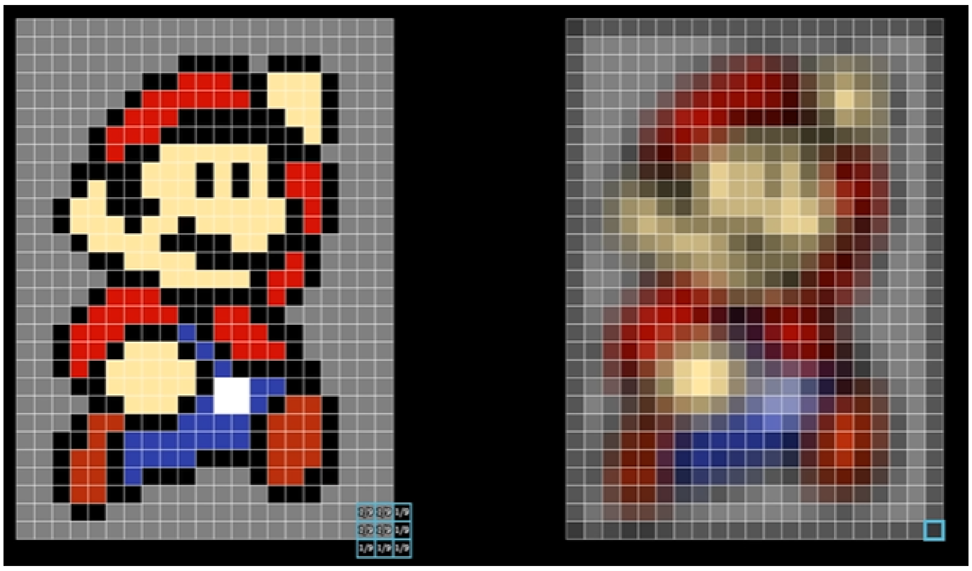


Figure 4: *Normalized Filter [14]*

Gaussian Filter

This filter is used to smooth the image, reducing noise and detail. Unlike the normalised filter, the Gaussian filter assigns more weight to the centre pixel and less to the surrounding ones, creating a blurring effect. A 3×3 Gaussian filter might look like:

$$\text{Gaussian Filter} = \frac{1}{16} \times \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

The sum of all elements in this Gaussian matrix is calculated as:

$$1 + 2 + 1 + 2 + 4 + 2 + 1 + 2 + 1 = 16$$

To ensure that applying the filter preserves the overall brightness or intensity of the region it covers, the sum of the weights in the filter must equal 1. Therefore, each element in the matrix is divided by the sum of 16. This results in the normalized Gaussian filter:

$$\frac{1}{16} \times \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\ \frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\ \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \end{bmatrix}$$

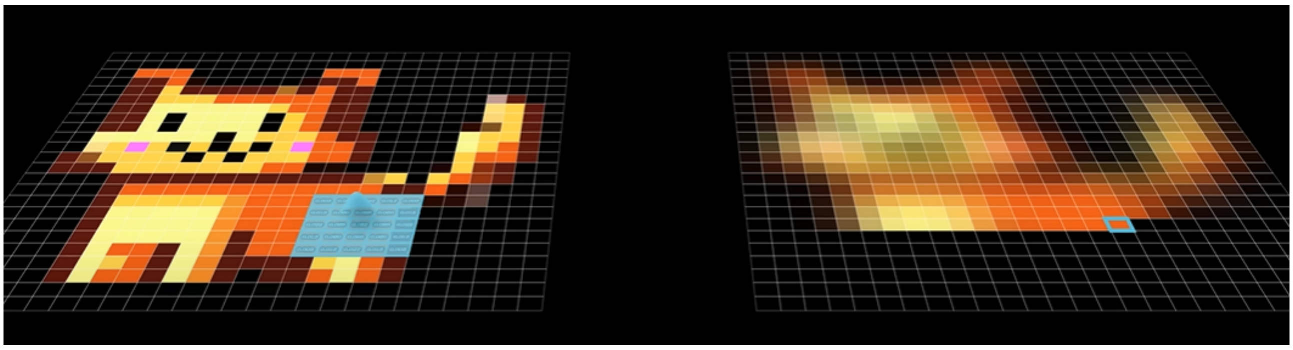


Figure 5: *Gaussian Filter[14]*

Example of Convolution Operation

Let's consider a simple example where a 3x3 normalized filter is applied to a 5x5 input matrix:

$$\text{Input Image} = \begin{bmatrix} 2 & 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 & 8 \\ 6 & 7 & 8 & 9 & 10 \\ 8 & 9 & 10 & 11 & 12 \\ 10 & 11 & 12 & 13 & 14 \end{bmatrix}$$

We will apply the normalised filter:

$$\text{Normalised Filter} = \frac{1}{9} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

As this filter slides over the input image, it multiplies corresponding elements and sums them up. Here's how the filter operates on a 3x3 section starting at the top-left corner of the image:

$$\text{Filtered Section} = \begin{bmatrix} 2 & 3 & 4 \\ 4 & 5 & 6 \\ 6 & 7 & 8 \end{bmatrix}$$

$$\text{Filtered Section} = \frac{1}{9} \times (2 \times 1 + 3 \times 1 + 4 \times 1 + 4 \times 1 + 5 \times 1 + 6 \times 1 + 6 \times 1 + 7 \times 1 + 8 \times 1)$$

$$\text{Result} = \frac{1}{9} \times (2 + 3 + 4 + 4 + 5 + 6 + 6 + 7 + 8) = \frac{45}{9} = 5$$

This value of 5 is now in the top left corner of the feature map. The process is repeated as the filter slides over the entire image, producing a smaller output matrix (feature map) that highlights the features detected by the filter:

$$\text{Output Feature Map} = \begin{bmatrix} 5 & 6 & 7 \\ 7 & 8 & 9 \\ 9 & 10 & 11 \end{bmatrix}$$

Now, through this understanding, the convolutional layer can be coded and added to the neural network as shown in **Appendix 1**.

3 Output Layer and Cost Function

After the input image is passed through the network and processed by all the layers, the network predicts what the image is. During training, its predictions will be incorrect, therefore the network needs to improve and it does so through the Cost function.
The Cost Function measures the accuracy of the network by calculating the difference between the predicted outputs of the network and the actual target values.[8] The goal of training a neural network is to minimise this Cost function, improving the accuracy of the network's predictions.

One of the most common methods of calculating cost functions used in neural networks is the **Mean Squared Error (MSE)**. For a single training example, it is calculated as:

$$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

Where:

- y is the true target value.
- $\hat{y}$ is the predicted value from the network.
- The factor $\frac{1}{2}$ is included to simplify the derivative calculation during backpropagation later on.

For a batch of m training examples, the overall cost function is the average of the individual squared losses:

$$J(W, b) = \frac{1}{m}\sum_{i=1}^{m} L(y^{(i)}, \hat{y}^{(i)}) = \frac{1}{2m}\sum_{i=1}^{m}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Where:

- $J(W, b)$ is the cost function to be minimized.
- W and b represent the weights and biases in the network.

- m is the number of training examples.
- $y^{(i)}$ and $\hat{y}^{(i)}$ are the actual and predicted values for the i-th training example, respectively.

4 Gradient Descent

When training neural networks, the objective is to minimise the cost function. Ideally, this function can be minimised using calculus, as one would for a simple function. However, the cost function in neural networks depends on multiple hyperparameters and has a complex, non-linear shape, often with numerous minima and saddle points. We use an optimisation algorithm known as **Gradient Descent** to address this[15].

4.1 Understanding Gradient Descent in 2D (Single Parameter)

To understand Gradient descent, let's begin with a 2D example where the cost function $J(\theta)$ depends on a single parameter θ.
The idea behind Gradient descent is to iteratively adjust the parameter θ in the direction that reduces the cost function the most rapidly, to eventually reach the minimum. Since the gradient points in the direction of the steepest increase, moving in the opposite direction ensures the fastest decrease in the cost function. So, the adjustment is done using the negative gradient of the cost function [16].

To perform Gradient descent:

1. Start at some initial value of θ, chosen randomly.
2. Compute the slope of the cost function at that point.
3. Move in the direction that decreases the cost function by updating the parameter
4. Repeat until the function has been minimised

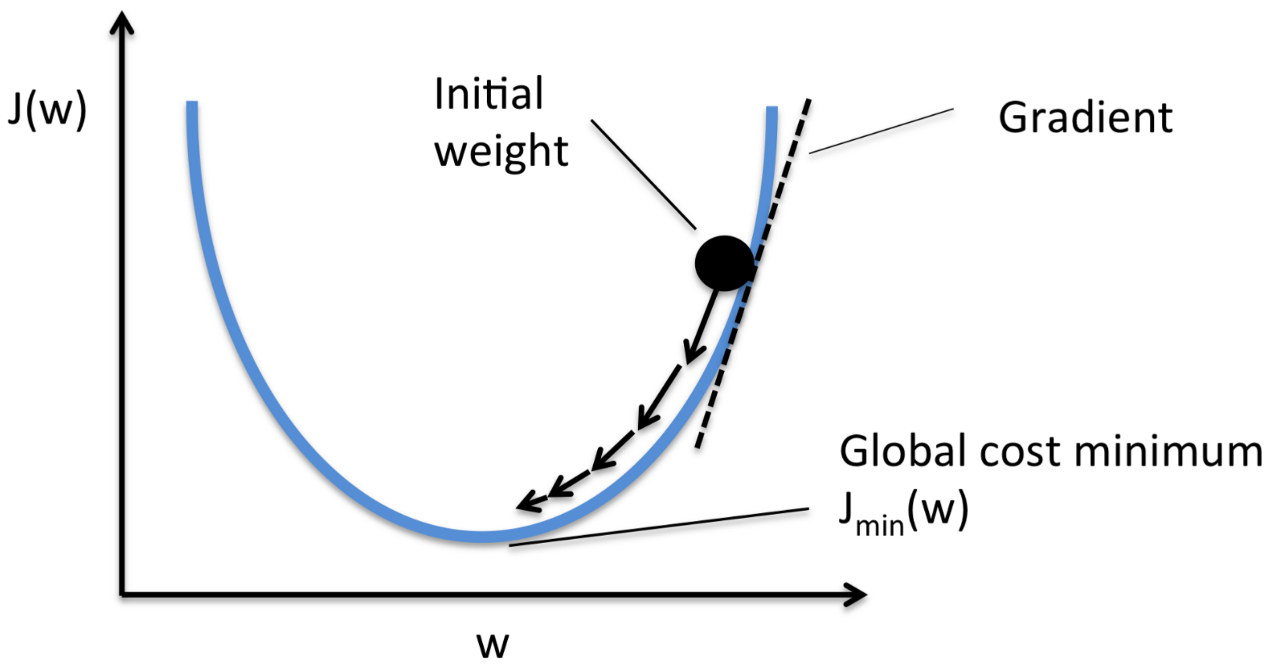


Figure 6: *Gradient Descent in 2D[17]*

In reality, due to it iteratively adjusting the parameter, the cost function will never truly reach the complete minimum, so there are constraints to prevent time wastage, such as: Repeating for only 200 iterations or repeating until the gradient is less than 0.001[18].

Using this understanding of Gradient descent, the update rule for the parameter is:

$$\theta := \theta - \alpha \frac{dJ}{d\theta}$$

Where:

- $:=$ is the assignment operator.
- θ is the hyperparameter we are updating.
- α is the **learning rate**, which controls the step size of each update.
- $\frac{dJ}{d\theta}$ is the gradient of the Cost function with respect to θ.

The learning rate α controls the step size at each iteration while moving toward the minimum of the cost function[19]. A small learning rate ensures gradual updates, leading to a more careful descent toward the minimum, but may make the training process inefficient if too small.

Conversely, a large learning rate enables faster convergence by making larger updates, but if α is too large, it risks overshooting the minimum.

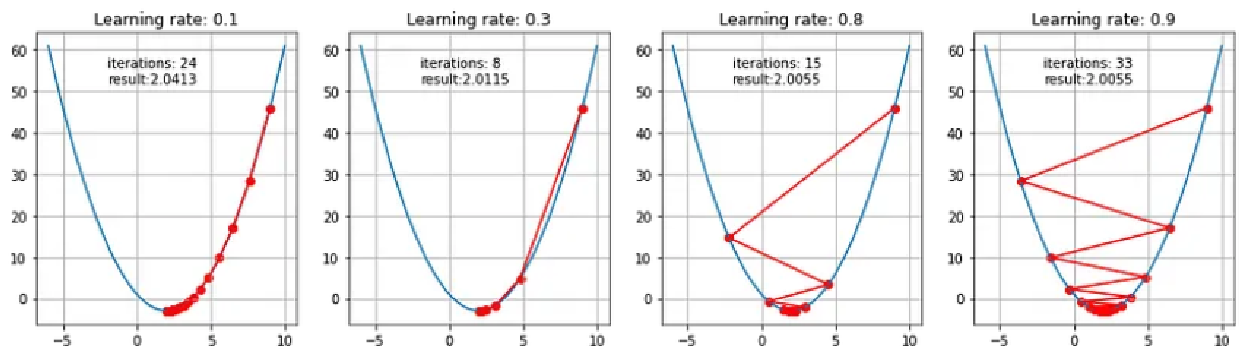


Figure 7: *Different learning rates and their convergence to the minimum [16]*

4.2 Gradient Descent in 3D (Two Parameters)

This can be extended in 3D where the cost function $J(\theta_0, \theta_1)$ depends on two hyperparameters, θ_0 and θ_1.
In this case, the Gradient descent algorithm computes the partial derivatives of the cost function with respect to each parameter and updates them simultaneously. The update rules for both parameters are:

$$\theta_i := \theta_i - \alpha \frac{\partial J}{\partial \theta_i}$$

Where:

- θ_i represents either θ_0 or θ_1.
- α is the learning rate.
- $\frac{\partial J}{\partial \theta_i}$ is the partial derivative of the cost function with respect to θ_i.

The gradient $\nabla J(\theta_0, \theta_1)$ is a vector pointing in the direction of where the cost function increases most rapidly. By moving in the opposite direction (the negative gradient), the algorithm ensures that the cost function decreases most rapidly.[20]

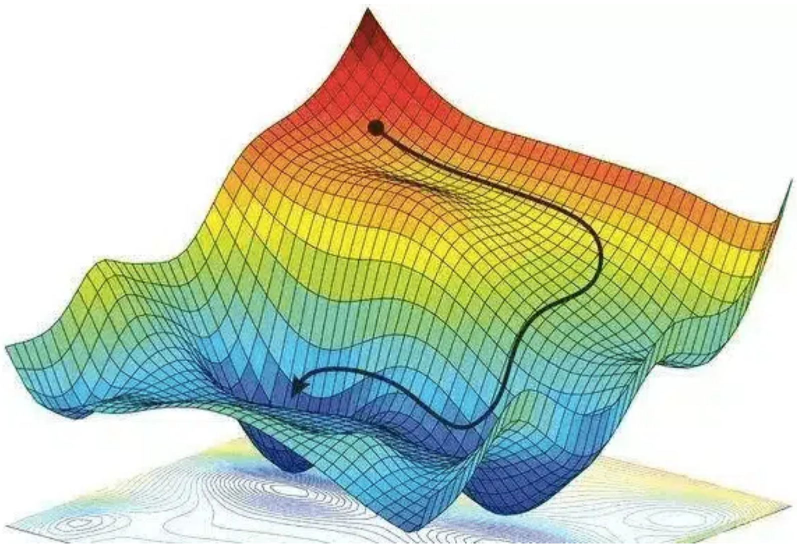


Figure 8: *Gradient Descent in 3D [20]*

4.3 Velocity and Momentum in Gradient Descent

One of the key challenges is ensuring that the algorithm finds the global minimum of the cost function rather than a local minimum. If Gradient descent gets trapped in a local minimum, the optimisation process will fail to find the best possible solution, which results in incorrect prediction from the network.

To address this, the concept of **velocity** and **momentum** is introduced. Momentum helps the algorithm continue to move in a consistent direction, even if it encounters a flat region or a shallow local minimum. [21]

The idea of momentum in Gradient descent can be understood through the analogy of a ball rolling down a hill:
Imagine you have a ball rolling from point A. As the ball rolls down the slope from point A to point B, it starts slowly but gradually picks up speed, accumulating momentum as it moves. When the ball reaches point B, it might encounter a plateau where it would normally slow down or stop. However, because the ball has gathered enough momentum while moving down the slope AB, it can push itself across the plateau region B. This momentum allows the ball to continue moving forward until it reaches the downward slope BC, eventually leading it to the global minimum at point C.

The update rule incorporating momentum is:

$$v_i := \beta v_i + (1-\beta)\frac{\partial J}{\partial \theta_i}$$

$$\theta_i := \theta_i - \alpha v_i$$

Where:

- v_i is the velocity term for each parameter, accumulating gradients over time.
- β is the momentum coefficient (usually 0.9), which influences the contribution of past gradients.
- α is the learning rate, controlling the step size.
- θ_i represents the parameters being updated (weights and biases).

By remembering previous gradients, momentum reduces chaotic movement and allows the algorithm to take larger, more direct steps toward the minimum of the cost function as shown in **Figure 9**, where the purple lines show normal Gradient descent, the blue lines show momentum-based Gradient descent and the red dot representing the global minimum.

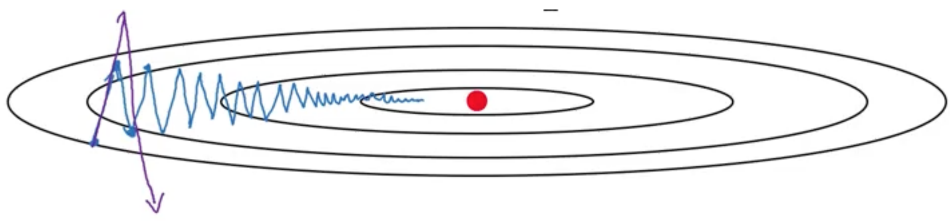


Figure 9: *Convergence of Gradient Descent with momentum [22]*

Now, through this understanding, the Momentum-based Gradient Descent can be coded and added to the neural network as shown in **Appendix 2**.

5 Backpropagation

5.1 Partial Differentiation and Chain Rule

In neural networks, partial differentiation is essential for optimising the network's performance. Specifically, partial derivatives are used in the backpropagation algorithm to compute gradients, which are then used to update the hyperparameters during training, which minimises the error of the neural network.

Partial differentiation deals with functions with multiple variables, and it involves differentiating the function with respect to one variable while keeping the other variables constant.[23]

For example, consider a simple function $f(x, y) = 2x^2 + 3y$. This function depends on two variables, x and y. The partial derivative of f with respect to x and y is denoted as $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ respectively.

Differentiating f with respect to x while treating y as a constant, results in:

$$\frac{\partial f}{\partial x} = \frac{\partial}{\partial x}\left(2x^2 + 3y\right) = 4x$$

Similarly, the partial derivative of f with respect to y is:

$$\frac{\partial f}{\partial y} = \frac{\partial}{\partial y}\left(2x^2 + 3y\right) = 3$$

These derivatives tell us how the function $f(x, y)$ changes as x or y changes, independently of each other.

Chain rule

The chain rule is a fundamental concept in calculus that allows us to differentiate composite functions[23]. When dealing with multi-variable functions in the context of partial differentiation, the chain rule becomes essential for understanding how a change in one variable affects another through intermediate variables. This is significant in neural networks, where the output of it is a combination of several functions and hyperparameters, which each need to be adjusted through Backpropagation.

Suppose there is:

- $z = g(x_1, x_2, \ldots, x_p)$, where z is a differentiable function of p independent variables $x_1, x_2, \ldots, x_p$.
- Each x_i is itself a function of q independent variables $u_1, u_2, \ldots, u_q$, i.e., for each $i \in \{1, 2, \ldots, p\}$, $x_i = x_i(u_1, u_2, \ldots, u_q)$.

The goal is to compute $\frac{\partial z}{\partial u_k}$, the rate of change of z with respect to u_k.

Since z depends on the variables $x_1, x_2, \ldots, x_p$, and each x_i depends on u_k, the way z changes with respect to u_k is determined by the indirect effect of u_k on z through each of the x_i's. Because z relies on each x_i, and each x_i depends on u_k, it follows that the total change in z with respect to u_k is influenced by both how z changes with respect to each x_i and how each x_i changes with respect to u_k.

To express this relationship mathematically, the total rate of change of z with respect to u_k is the sum of the contributions from all the intermediate variables $x_1, x_2, \ldots, x_p$. For each x_i, the contribution is the product of two factors: the sensitivity of z to changes in x_i (i.e., $\frac{\partial z}{\partial x_i}$) and the sensitivity of x_i to changes in u_k (i.e., $\frac{\partial x_i}{\partial u_k}$).
Thus, the total derivative of z with respect to u_k is given by the sum:

$$\frac{\partial z}{\partial u_k} = \sum_{i=1}^{p} \frac{\partial z}{\partial x_i} \cdot \frac{\partial x_i}{\partial u_k}$$

5.2 Backpropogation

Backpropagation is the process neural networks use to adjust the hyperparameters to minimise the Cost function J[24]. It works by computing the gradient of the cost function with respect to each parameter in the network, using the chain rule.

These adjustments propagate backwards through the network, layer by layer, refining the hyperparameters. Each change at one layer affects the next, allowing the network to gradually minimise the cost function and improve its predictions.

Consider a simple neural network with 3 layers, each containing a single neuron:

- The input layer, with a single input a_0.
- A hidden layer with a single neuron and hyperparameters (weight w_1, and bias b_1)
- The output layer with a single neuron and hyperparameters (weight w_2, and bias b_2)

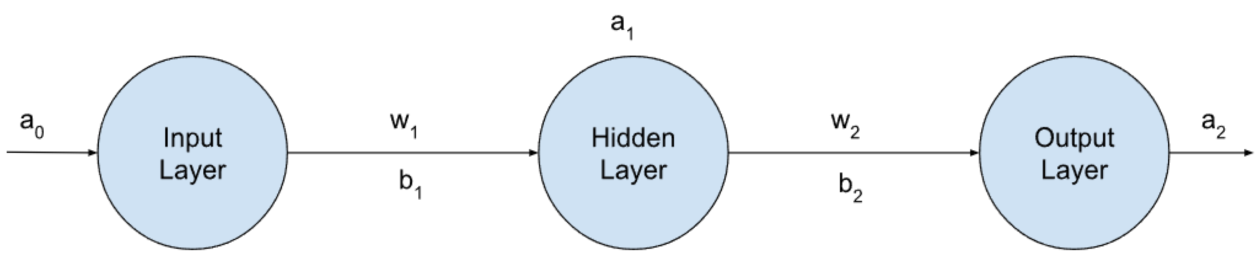


Figure 10: *Simple Neural Network with hyperparameters*

During forward propagation, the input a_0 is combined with the hyperparameters as it moves through the network. These combinations determine the intermediate values at each layer, leading to a final output a_2. While the specifics of activation functions and intermediate calculations aren't critical here, the key point is that the final output depends on all the hyperparameters in the network.

Once the output a_2 is calculated, the Cost Function is calculated as:

$$J = \frac{1}{2}(y - a_2)^2,$$

where y is the true value.

Therefore, Backpropogation can be performed for a specific hyperparameter. Here, the gradient for the Cost function with respect to w_2 is:

$$\frac{\partial J}{\partial w_2}$$

This represents how much the cost function changed with a change in w_2, and it is known that w_2 causes some change in a_1, which causes a change in a_2, ultimately affecting J, the cost function. Using this idea, the chain rule can be applied to arrive at the above expression.

Therefore, the derivative of the cost function J with respect to w_2, using the chain rule, is:

$$\frac{\partial J}{\partial w_2} = \frac{\partial J}{\partial a_2} \cdot \frac{\partial a_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_2}$$

Generalisation to Multiple Layers

When the network has multiple layers, the chain rule is applied but the gradients are summed across all neurons in each layer. For a weight w_{jk}^l connecting neuron k in layer $l-1$ to neuron j in layer l:

$$\frac{\partial J}{\partial w_{jk}^l} = \sum_m \frac{\partial J}{\partial a_m^L} \cdot \frac{\partial a_m^L}{\partial z_m^L} \cdot \frac{\partial z_m^L}{\partial w_{jk}^l}$$

Here, the summation occurs over all neurons m in the final layer L. This formula generalises the calculation of gradients across any number of layers. This approach ensures that backpropagation effectively adjusts all hyperparameters in the network, minimising the cost function J and enabling the network to learn from the data.

Now, through this understanding, the Backpropogation can be coded and added to the neural network as shown in **Appendix 3**

6 Building the CNN

Using the generic neural network template, and adding to it at each stage through the mathematical understanding, the CNN has been successfully constructed, combining everything discussed in the IA, as shown in **Appendix 4**. This CNN has been trained to accurately classify five types of flowers: Daisy, Rose, Sunflower, Tulip, and Dandelion.

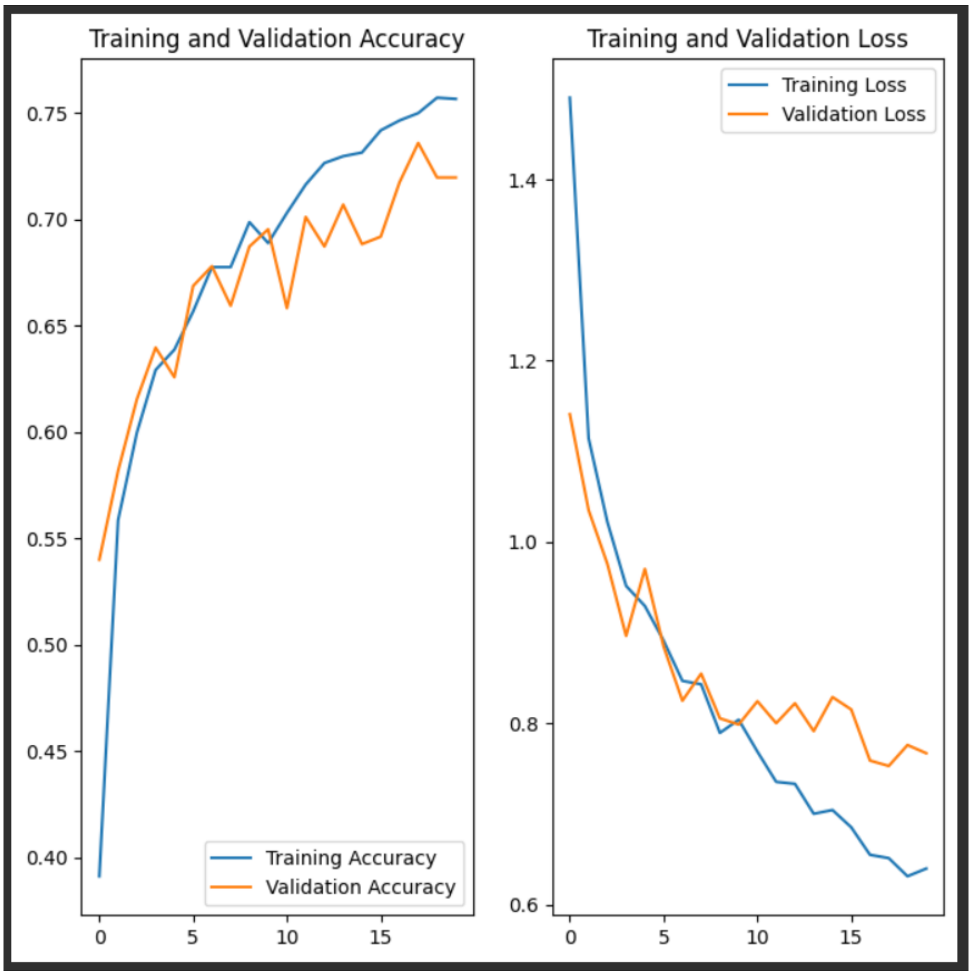
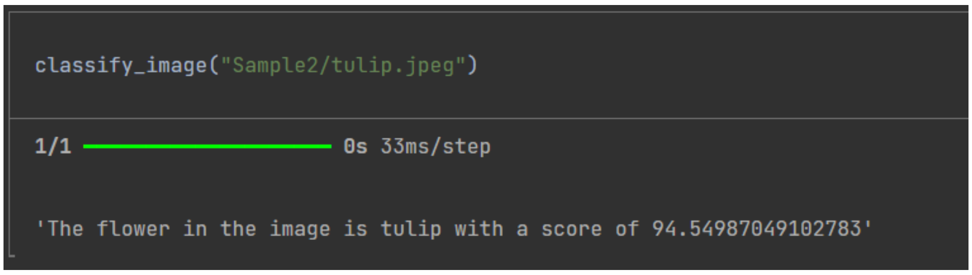


Figure 11: *Training and Validation set Accuracy and Loss*

During training, CNN achieved a 80% accuracy and 10% loss as shown in Figure 11, indicating success and suggesting the optimisation techniques and mathematical foundations are working and result in effective learning.



```
classify_image("Sample2/tulip.jpeg")

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step

'The flower in the image is tulip with a score of 94.54987049102783'
```

Figure 12: *CNN prediction*

After training, the CNN was tested with new data, and as shown in **Figure 12**, the CNN is successfully classifying images of flowers with high accuracy rates, averaging around 95%. Therefore, we have successfully built an accurate CNN.

7 Conclusion

In conclusion, this project involved the construction of a CNN that successfully achieved high accuracy and low loss in recognizing five flower types. Specifically, the model reached a test accuracy of 95%, as shown in Figure 12, and maintained a loss of 0.1, as illustrated in Figure 11, demonstrating that the initial goal was met. The network's performance highlights the importance of understanding the mathematical principles behind its layers, including the convolutional, pooling, and fully connected layers, as well as activation functions, cost functions, and optimization algorithms like gradient descent and momentum.

One of the primary strengths of this project was the simplicity and effectiveness of the CNN architecture. By carefully designing the layers and optimising the learning rate, the network achieved high accuracy with relatively low computational cost. Additionally, the clear relationship between mathematical theory and implementation provided a robust foundation for understanding how each component contributes to overall performance.

However, the CNN faced some limitations. The model's ability to generalise to unseen data was limited by its relatively small dataset, which increased the risk of overfitting despite the low loss achieved during testing. Furthermore, while the use of traditional gradient descent and momentum proved effective, more advanced optimisation techniques could potentially improve convergence and stability, allowing for higher training efficiency.

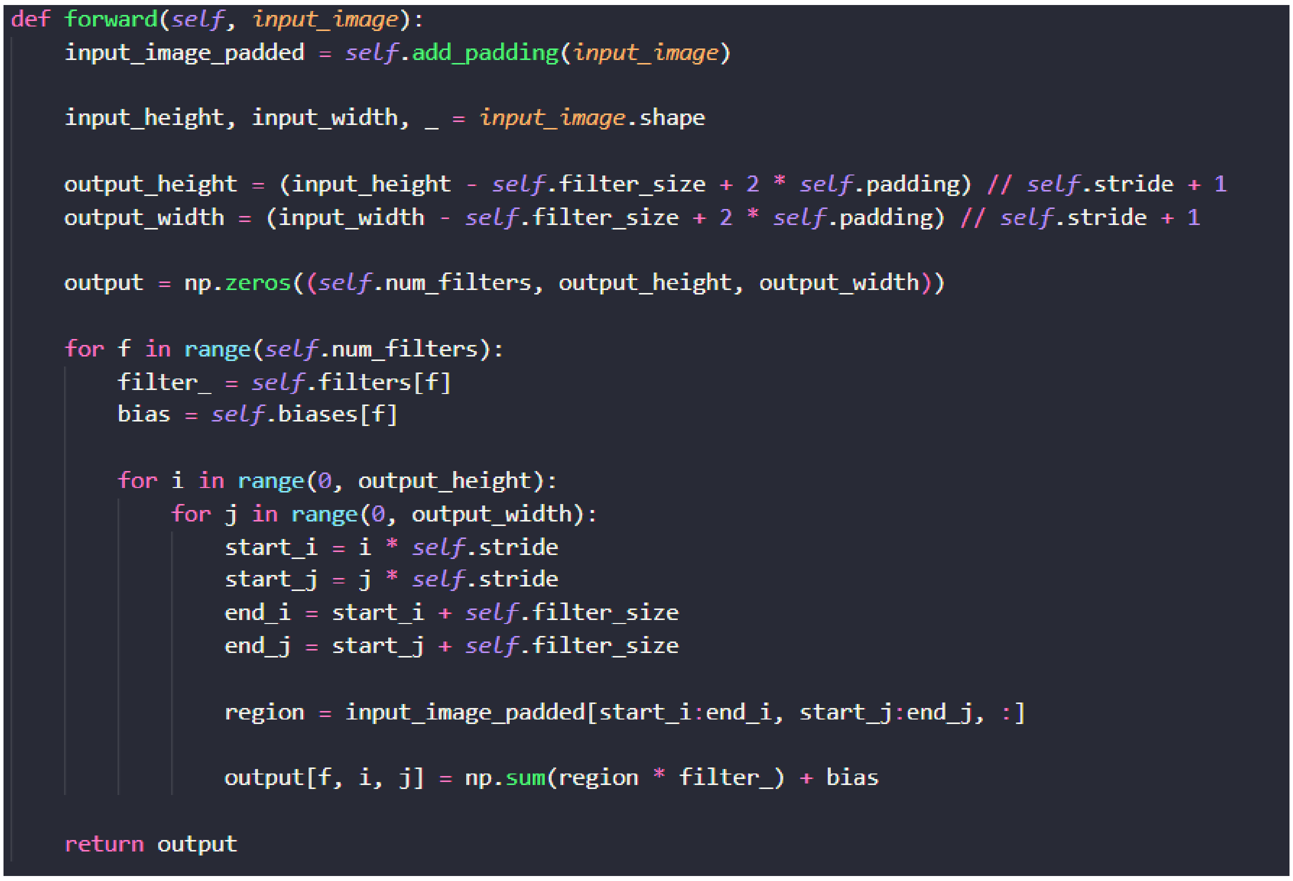
To further improve performance, advanced optimisation algorithms such as Adam could be implemented. Adam combines momentum with adaptive learning rates to optimise the step size during training. It uses both the first moment (mean of the gradients) and the second moment (variance of the gradients) to adjust the learning rate for each parameter[25]. This allows the algorithm to make more stable and efficient updates by considering the magnitude and variance of the gradients. By dynamically adjusting the learning rates for each parameter, Adam can lead to faster convergence and improved accuracy compared to traditional gradient descent[15].

Regularisation techniques, such as L2 regularization, could also be applied to prevent overfitting and improve generalisation. L2 regularisation works by adding a penalty term to the cost function based on the sum of the squared weights[5]. This discourages the network from relying too heavily on any single weight, promoting a simpler model that better generalises to unseen data.

In summary, incorporating advanced optimisation methods such as Adam and regularisation techniques such as L2 regularisation presents key mathematical opportunities to improve CNN's performance. These improvements would lead to a more efficient model with better convergence, reduced overfitting, and increased accuracy.

8 Appendix

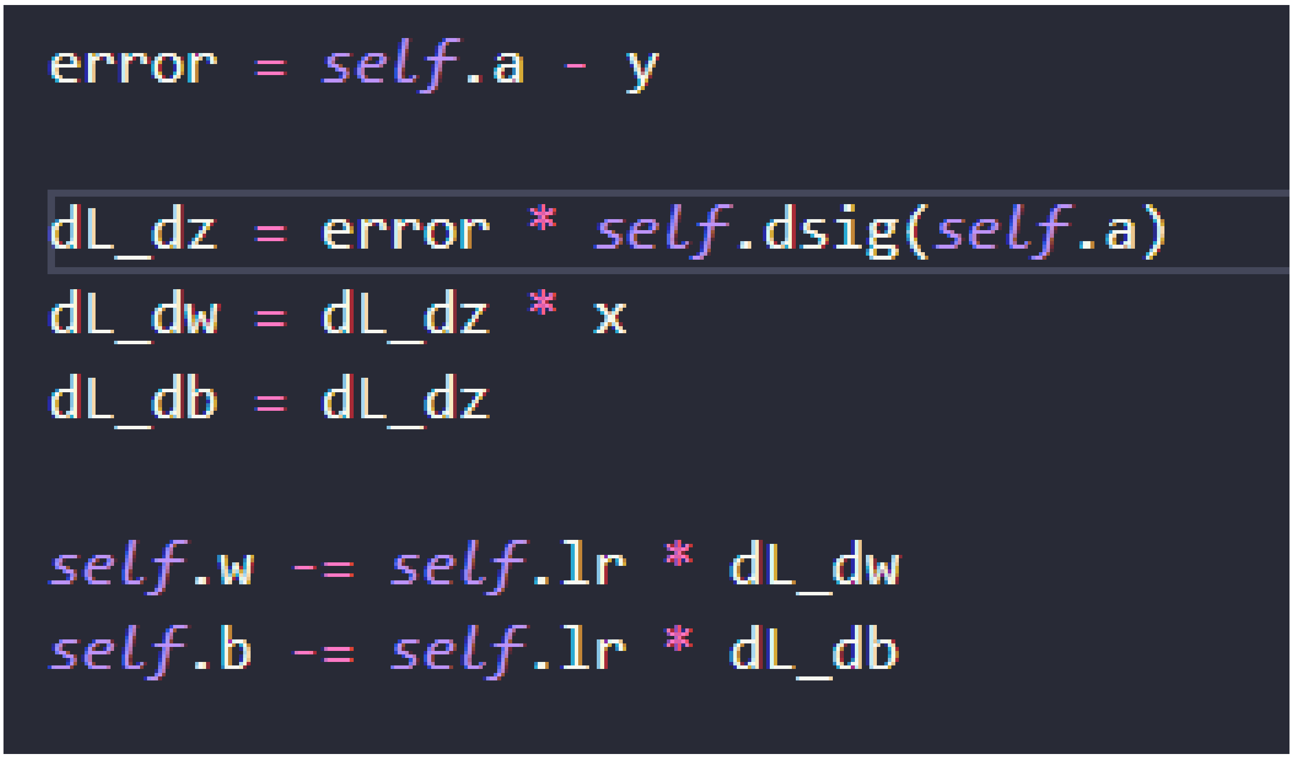
8.1 Appendix 1



```
def forward(self, input_image):
    input_image_padded = self.add_padding(input_image)

    input_height, input_width, _ = input_image.shape

    output_height = (input_height - self.filter_size + 2 * self.padding) // self.stride + 1
    output_width = (input_width - self.filter_size + 2 * self.padding) // self.stride + 1

    output = np.zeros((self.num_filters, output_height, output_width))

    for f in range(self.num_filters):
        filter_ = self.filters[f]
        bias = self.biases[f]

        for i in range(0, output_height):
            for j in range(0, output_width):
                start_i = i * self.stride
                start_j = j * self.stride
                end_i = start_i + self.filter_size
                end_j = start_j + self.filter_size

                region = input_image_padded[start_i:end_i, start_j:end_j, :]

                output[f, i, j] = np.sum(region * filter_) + bias

    return output
```

Figure 13: *Convolutional Layer Code*

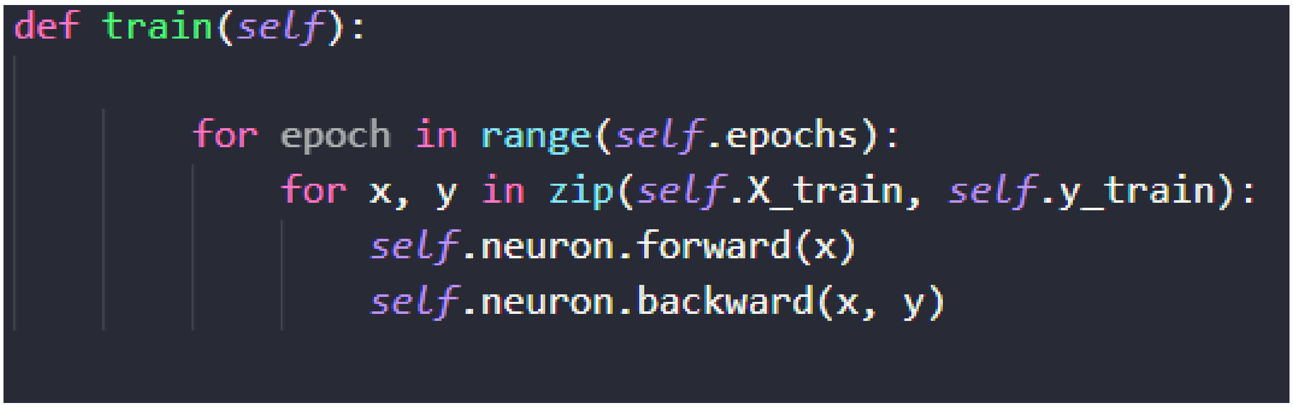
8.2 Appendix 2



```python
error = self.a - y

dL_dz = error * self.dsig(self.a)
dL_dw = dL_dz * x
dL_db = dL_dz

self.w -= self.lr * dL_dw
self.b -= self.lr * dL_db
```

Figure 14: *Gradient Descent Code*

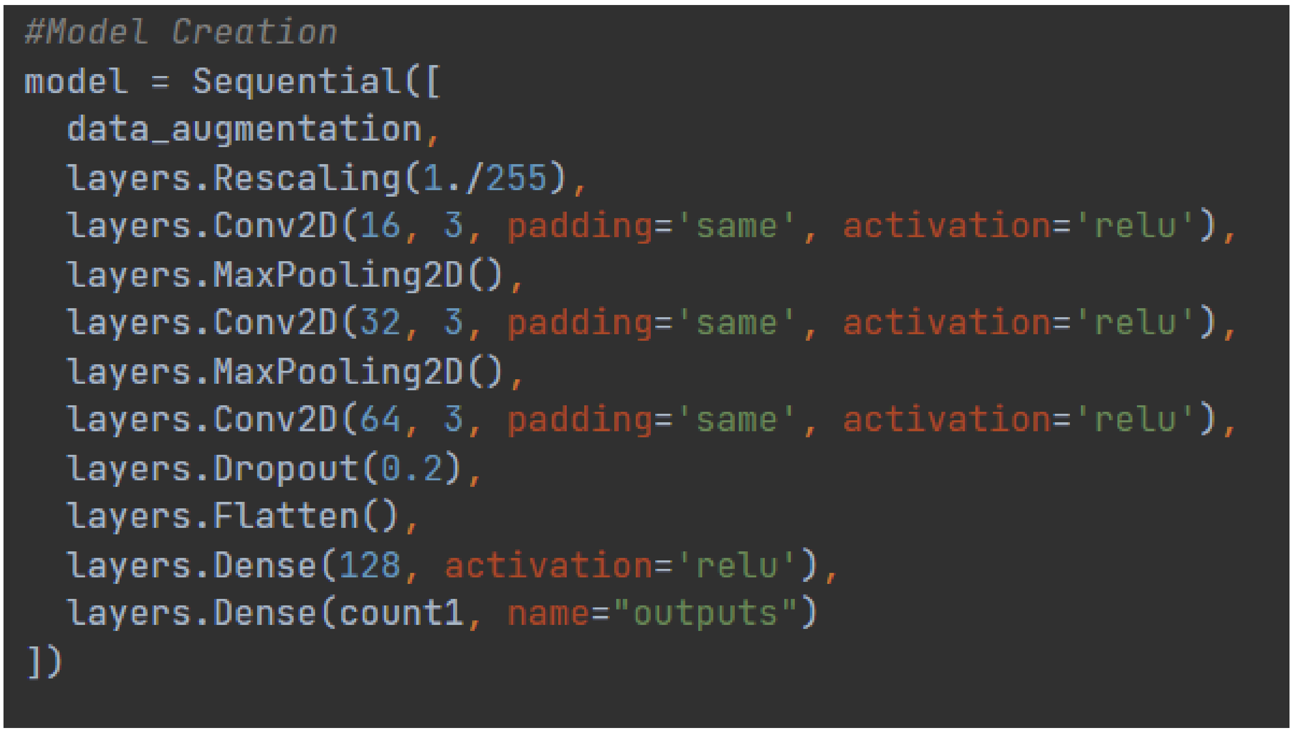
8.3 Appendix 3



```python
def train(self):

        for epoch in range(self.epochs):
            for x, y in zip(self.X_train, self.y_train):
                self.neuron.forward(x)
                self.neuron.backward(x, y)
```

Figure 15: *Backpropogation Code*

8.4 Appendix 4



```
#Model Creation
model = Sequential([
  data_augmentation,
  layers.Rescaling(1./255),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.Dropout(0.2),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(count1, name="outputs")
])
```

Figure 16: *CNN model creation*

References

1. *Neural Networks and Deep Learning* Accessed: 2024-06-25. `http://neuralnetworksanddeeplearning` `com/chap1.html`.

2. *Convolutional Neural Networks* Accessed: 2024-08-12. `https://andrew.gibiansky.com/blog/machine-learning/convolutional-neural-networks/`.

3. *Neural Network Concepts* Accessed: 2024-08-03. `https://ml-cheatsheet.readthedocs.io/en/latest/nn_concepts.html#neuron`.

4. Weisstein, E. W. *Books about Convolution* Accessed: 2024-06-18. `http://www.ericweisstein.com/encyclopedias/books/Convolution.html`.

5. Hardesty, L. *Explained: Neural Networks* MIT News Office. Archived from the original on 18 March 2024. Retrieved 2 June 2022. `https://news.mit.edu/2017/explained-neural-networks-0414`.

6. *The Machine Learning Dictionary* Archived from the original on 26 August 2018. Retrieved 4 November 2009. `https://www.cse.unsw.edu.au`.

7. Investopedia. *What Is a Neural Network?* Accessed: 2024-12-02. https://www.investopedia.com/terms/n/neuralnetwork.asp.

8. *Cost Function* Accessed: 2024-09-15. https://builtin.com/machine-learning/cost-function#:~:text=Is%20Cost%20Function%3F-,Cost%20function%20measures%20the%20performance%20of%20a%20machine%20learning%20model,of%20a%20single%20real%20number.

9. Hirschman, I. I. & Widder, D. V. *The Convolution Transform* (Princeton University Press, Princeton, NJ, 1955).

10. Bracewell, R. *The Fourier Transform and Its Applications* 25–50, 243–244 (McGraw-Hill, 1965).

11. *Intuitive Convolution* Accessed: 2024-08-28. https://betterexplained.com/articles/intuitive-convolution/.

12. *Convolution* Accessed: 2024-06-30. https://colah.github.io.

13. *An Intuitive Explanation of Convnets* Accessed: 2024-07-01. https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/.

14. Deeplizard. *Deep Learning: A Visual Introduction* Accessed: 2024-09-15. https://www.youtube.com/watch?v=KuXjwB4LzSA.

15. Polyak, B. *Introduction to Optimization* (Unknown, 1987).

16. *Gradient Descent Algorithm: A Deep Dive* Accessed: 2024-06-10. https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21.

17. *Gradient Descent: A Deep Dive* Accessed: 2024-07-15. https://ekamperi.github.io/machine%20learning/2019/07/28/gradient-descent.html.

18. *Gradient Descent* Accessed: 2024-08-31. https://en.wikipedia.org/wiki/Gradient_descent.

19. *Convolutional Neural Networks (LeNet) – DeepLearning 0.1 documentation* Accessed: 2024-07-20. https://deeplearning0.1/doc/lenet.html.

20. *Gradient Descent* Accessed: 2024-08-01. https://easyai.tech/en/ai-definition/gradient-descent/#wiki.

21. *Momentum and Learning Rate Adaptation* Willamette University. Retrieved 17 October 2014. `https://willamette.edu`.

22. *Coursera Deep Learning Course: Week 2* Accessed: 2024-06-20. `https://nhannguyen95.github.io/coursera-deep-learning-course-2-week-2/`.

23. *Partial Derivatives* Accessed: 2024-09-15. `https://www.ncl.ac.uk/webtemplate/ask-assets/external/maths-resources/core-mathematics/calculus/partial-derivatives.html`.

24. Press, W. H., Flannery, B. P., Teukolsky, S. A. & Vetterling, W. T. *Numerical Recipes in FORTRAN: The Art of Scientific Computing* 2nd, 531–537 (Cambridge University Press, 1992).

25. Krizhevsky, A. *ImageNet Classification with Deep Convolutional Neural Networks* in *Proceedings of the 25th International Conference on Neural Information Processing Systems* Archived from the original on 25 April 2021. Retrieved 17 November 2013 (2012).