Deep Learning

Candidate number: 2410492

Submitted for the Degree of Master of Science in

Artificial Intelligence.

Department of Computer Science
Royal Holloway University of London
Egham, Surrey TW20 0EX, UK

September 02, 2024

Word Count: 15520 (Approx)

Abstract

This dissertation presents a comprehensive study on the implementation, optimization, and analysis of deep learning algorithms, focusing on image classification tasks. The project begins with the development of baseline Artificial Neural Network (ANN) and Convolutional Neural Network (CNN) models using the CIFAR-10 dataset, exploring fundamental deep learning concepts. Through a series of experiments, the impact of network architecture complexity, including the number of layers, neurons, and activation functions, is investigated.

Key factors such as optimal training epochs, weight initialization techniques, and various optimization algorithms are systematically analysed to enhance model performance. Regularization techniques, including L1, L2, Dropout, Early Stopping, and Batch Normalization, are implemented to mitigate overfitting and improve generalization. The project also includes a brief report on related deep learning architectures, providing context for the experiments conducted.

In the final phase, the project shifts focus to a real-world dataset involving images of human faces with and without masks. Custom neural networks and transfer learning approaches are employed using several pretrained models, including ResNet50, VGG16, InceptionV3, DenseNet121, and MobileNetV2, to develop a robust face mask detection system. These models were trained and evaluated, with several achieving strong results in predicting new, unseen data.

The project over all demonstrates the effectiveness of leveraging pretrained models to enhance accuracy and efficiency in image classification tasks. This work successfully achieves the project's original aims, highlighting the practical application of deep learning techniques in solving real-world problems like mask detection.

Contents

1 Introduction

1.1 Background and Motivation

Over the past decade, deep learning has advanced rapidly, with many new architectures being introduced and developed. These advancements have significantly improved the performance of machine learning models, making them more accurate and efficient than previous methods. The increase in computational power, along with the availability of large datasets, has further driven progress in deep learning, making it an important area of research in fields like computer vision, natural language processing, and robotics.

The motivation for this project stems from a desire to explore and understand the broad capabilities of deep learning by implementing a range of approaches, including simple architectures like ANN and CNN as well as more advanced techniques like transfer learning with pre-trained networks. By evaluating and comparing these various models, the project aims to uncover their respective strengths and weaknesses across different tasks. Through careful visualization and experimentation, the goal is to identify opportunities to improve these models' performance. This investigation is important as it contributes to the ongoing efforts to refine and enhance deep learning techniques, making them more versatile and effective for a wide array of practical applications.

1.2 Aims and Objectives

The primary aim of this project is to implement several deep learning algorithms on a real-world dataset and improve their performance by exploring and comparing multiple architectures. The goal is to identify the most effective configurations by utilizing various approaches that enhance the accuracy and adaptability of these algorithms.

The initial objectives involve creating a proof-of-concept program that applies deep learning techniques to a widely-used, easily accessible dataset, where different architectural choices will be tested to assess their impact on performance. An early report will provide an overview of the various deep learning methods used and evaluate the initial experiments.

As the project progresses, the final objectives focus on applying these algorithms to more complex, real-world datasets, implementing and comparing multiple architectures. Additionally, the project will include visualizing and analysing the performance of these models, using different strategies to refine and improve the outcomes. A comprehensive report will document the methods, architectural decisions, and overall implementation process.

1.3 Project Significance and Career Impact

This project is crucial for bridging the gap between theoretical knowledge and practical application. It allows me to apply and test concepts learned through academic coursework in real-world scenarios, providing a deeper understanding of deep learning and neural networks. Working with various deep learning models and datasets enhances my grasp of these fundamental concepts, which is essential for building a robust foundation in data science and machine learning.

Additionally, the project has been instrumental in developing essential career skills, such as time management, project planning, and problem-solving. These skills are vital for navigating complex tasks and meeting deadlines in a professional setting. By managing different aspects of the project, from experimentation to analysis and documentation, I have gained valuable experience that will aid in my future career. This hands-on experience not only strengthens my technical expertise but also prepares me for the demands of the industry.

2 Background Research

In this chapter, we look at key ideas and methods in deep learning that are important for this project. We will start with the basics of deep learning and then explore important models used for image classification. We will also discuss techniques to improve and control model performance, as well as methods like transfer learning and active learning that help make models more effective. This review helps us understand the concepts and methods we will use in our project and why they matter. For the entire background research, I have gone through several books and slides including [1], [2] .

2.1 Introduction to Deep Learning

Deep learning is a branch of artificial intelligence that focuses on training computers to recognize patterns and make decisions using large amounts of data. It involves using neural networks with many layers, which can learn complex features from data. Unlike traditional machine learning methods, deep learning models can automatically discover intricate patterns and relationships in the data without needing explicit instructions for every task.

Neural networks, the backbone of deep learning, are inspired by the human brain's structure. These networks consist of interconnected nodes or "neurons" that process information. The more layers a neural network has, the deeper it is, and the more features it can learn. This depth allows deep learning models to excel in tasks like image and speech recognition, where they can achieve remarkable accuracy.

Training these models requires a significant amount of data and computational power. However, once trained, they can make highly accurate predictions or classifications, which makes them powerful tools for various applications. The advancements in deep learning have led to improvements in many fields, including healthcare, finance, and autonomous systems.

2.1.1 A simple Neural network and its training process.

A simple neural network is the building block for understanding more complex models. It consists of layers of neurons connected by weights as shown in fig 2.1. Each neuron receives inputs, applies weights to them, and then passes the result through an activation function to produce an output.

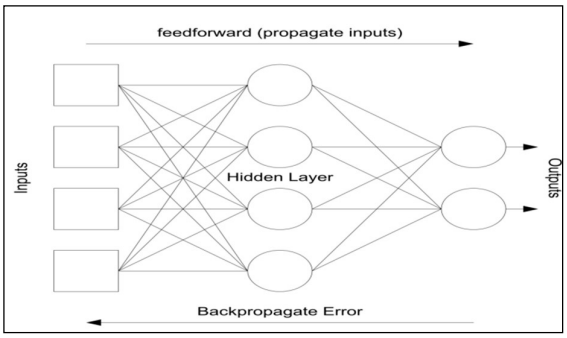


Fig 2.1 A simple Neural Network.

The training of a neural network involves two main processes: forward propagation and backward propagation. In forward propagation, the network takes input data, processes it through its layers using the weights, and generates an output. This output is then compared to the expected result to determine the error or difference.

Backward propagation is used to minimize this error. It works by adjusting the weights based on how much each weight contributed to the error. The error is sent back through the network, and weights are updated incrementally to improve the network's performance. This process is repeated many times, allowing the network to learn from the data and make more accurate predictions.

Building and evaluating a neural network involves setting up the network structure, training it on data, and testing its performance to ensure it is learning correctly and making reliable predictions. This simple neural network serves as a foundation for more advanced models and helps in understanding the core concepts of how neural networks function and improve.

2.1.2 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs)[3] are a specialized type of neural network designed to handle image data effectively. Unlike simple neural networks, CNNs use convolutional layers to automatically and adaptively learn spatial hierarchies of features from input images. This makes them particularly well-suited for tasks like image classification, object detection, and recognition.

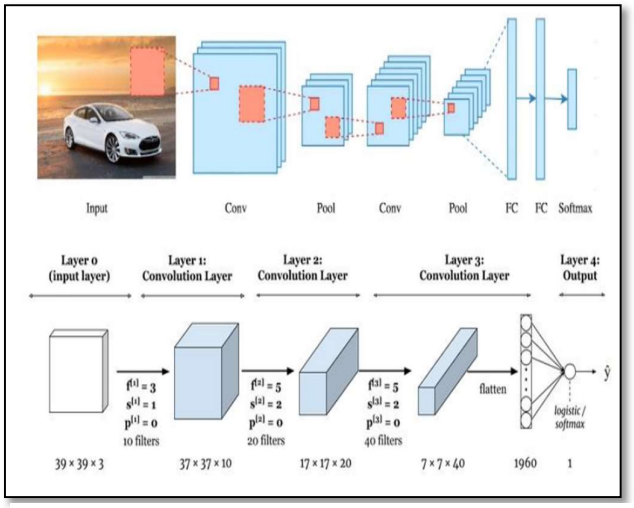


Fig 2.2 CNN

The core idea behind CNNs is the convolution operation, where a small matrix (called a filter or kernel) slides over the input image to produce a feature map. Each filter detects a specific feature, such as edges or textures, by focusing on local regions of the image. As the network goes deeper, these features are combined and refined to capture more complex patterns.

CNNs also include pooling layers, which reduce the spatial dimensions of the feature maps, helping to lower the computational load and making the network more robust to variations in the input. The final layers of a CNN are

typically fully connected layers that aggregate the features learned by the convolutional layers and make the final predictions.

Training a CNN involves the same forward and backward propagation processes as in simple neural networks, but with the added complexity of learning filters that can detect meaningful features in the data. CNNs have proven to be powerful tools in computer vision, achieving state-of-the-art results in various image-related tasks.

2.2 Optimization and Regularization Techniques in Deep Learning

2.2.1 Optimization Techniques

In a nutshell, Optimization techniques are at the core of training deep learning models. The goal of training a model is to minimize the difference between the model's predictions and the actual target values, a process often referred to as minimizing the loss function. This minimization is achieved through optimization techniques that adjust the model's parameters (weights and biases) iteratively to find the optimal set of values that result in the lowest possible loss.

Optimization techniques in deep learning are algorithms that guide the update of model parameters during training, ensuring that the model progressively improves. These techniques vary in their approach to adjusting parameters, with some focusing on speeding up convergence, others on stabilizing the training process, and still others on escaping local minima. Popular optimization methods include Gradient Descent and its variants, each offering different advantages depending on the characteristics of the problem at hand.

Now let us look at some of the algorithms[4] at brief:

- **Gradient Descent**

Gradient Descent is the most basic optimization algorithm used to minimize a loss function $L(\theta)$, where θ represents the model parameters. The update rule for gradient descent is given by:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t)$$

- θ_t: The parameters at the current step t.
- η: The learning rate, a small positive value that controls the size of the step.
- $\nabla_\theta L(\theta_t)$: The gradient of the loss function with respect to the parameters θ_t at step t.

This equation updates the parameters in the direction of the steepest descent (opposite to the gradient) to minimize the loss.

- **Stochastic Gradient Descent (SGD)**

Stochastic Gradient Descent (SGD) is a variation of Gradient Descent where the parameter update is performed for each training example rather than for the entire dataset. The update rule is given by:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t; x_i, y_i)$$

- θ_t: The parameters at the current step t.
- η: The learning rate.
- $\nabla_\theta L(\theta_t; x_i, y_i)$: The gradient of the loss function with respect to the parameters θ_t calculated using the i-th training example (x_i, y_i).

SGD introduces noise to the parameter updates, which can help escape local minima but also makes convergence less stable.

- **Mini-Batch Stochastic Gradient Descent**

Mini-Batch SGD combines the concepts of Batch Gradient Descent and Stochastic Gradient Descent by updating the parameters using a small subset (mini-batch) of the training data. The update rule is:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t; \mathcal{B}_t)$$

- θ_t: The parameters at the current step t.
- η: The learning rate.
- $\nabla_\theta L(\theta_t; \mathcal{B}_t)$: The gradient of the loss function with respect to the parameters θ_t calculated using the mini-batch $\mathcal{B}_t$.

Mini-Batch SGD offers a trade-off between the stability of Batch Gradient Descent and the computational efficiency of SGD.

- **Momentum**

Momentum is an optimization technique that accelerates Gradient Descent by considering the past gradients in the parameter update. The update rules are:

$$v_{t+1} = \gamma v_t + \eta \nabla_\theta L(\theta_t)$$

$$\theta_{t+1} = \theta_t - v_{t+1}$$

- v_t: The velocity vector at step t, which accumulates the gradient.
- γ: The momentum factor, typically between 0 and 1, controlling the influence of the past gradients.
- η: The learning rate.
- $\nabla_\theta L(\theta_t)$: The gradient of the loss function with respect to the parameters θ_t.

Momentum helps in smoothing the updates and accelerates convergence in relevant directions while damping oscillations.

- **RMSprop**

RMSprop is an optimization technique that adjusts the learning rate for each parameter based on the magnitude of its recent gradients. The update rules are:

$$s_{t+1} = \rho s_t + (1 - \rho) \nabla_\theta L(\theta_t)^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{s_{t+1} + \epsilon}} \nabla_\theta L(\theta_t)$$

- s_t: The moving average of the squared gradients.
- ρ: The decay rate, typically set to 0.9.

- η: The learning rate.
- ϵ: A small constant added for numerical stability.
- $\nabla_\theta L(\theta_t)$: The gradient of the loss function with respect to the parameters θ_t.

RMSprop helps in dealing with the problem of vanishing and exploding gradients by normalizing the gradients.

- **Adam (Adaptive Moment Estimation)**

Adam combines the ideas of Momentum and RMSprop by computing adaptive learning rates for each parameter. The update rules are:

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1)\nabla_\theta L(\theta_t)$$

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2)\nabla_\theta L(\theta_t)^2$$

$$\hat{m}_{t+1} = \frac{m_{t+1}}{1 - \beta_1^t}, \quad \hat{v}_{t+1} = \frac{v_{t+1}}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_{t+1}} + \epsilon}\hat{m}_{t+1}$$

- m_t: The first moment (mean) of the gradients.
- v_t: The second moment (uncentered variance) of the gradients.
- β_1, β_2: Exponential decay rates for the moment estimates, typically set to 0.9 and 0.999, respectively.
- η: The learning rate.
- ϵ: A small constant added for numerical stability.
- $\nabla_\theta L(\theta_t)$: The gradient of the loss function with respect to the parameters θ_t.

Adam is widely used due to its ability to adapt learning rates for each parameter, making it robust and efficient in various scenarios.

While the above algorithms form the core of optimization in deep learning, other techniques also play significant roles. These include Adagrad, Nadam, FTRL, and LBFGS, each providing different approaches to enhance training efficiency and model performance.

2.2.2 Regularization Techniques

In deep learning, the primary goal is to create models that not only perform well on training data but also generalize effectively to unseen data. To achieve this, it is crucial to address two common issues: overfitting and underfitting.

- **Overfitting** occurs when a model learns the training data too well, capturing noise and outliers rather than the underlying patterns. This results in high accuracy on training data but poor performance on validation or test data. Essentially, the model becomes too complex and fails to generalize to new data.

- **Underfitting**, on the other hand, happens when a model is too simple to capture the underlying patterns in the data. This leads to poor performance on both training and test data, as the model does not have enough capacity to learn effectively.

To tackle overfitting, we use regularization techniques. Regularization methods add constraints or penalties to the model's training process, discouraging it from fitting the training data too closely. These techniques help in reducing the model's complexity, ensuring that it captures the essential patterns without overfitting to the noise. By incorporating regularization, we aim to improve the model's ability to generalize and perform well on unseen data.

Some of the important techniques [5]are:

- **L1 Regularization**

  L1 regularization, also known as Lasso regularization, adds a penalty equal to the absolute value of the magnitude of coefficients to the loss function. The updated loss function $L(\theta)$ with L1 regularization is given by:

$$L(\theta) = L_0(\theta) + \lambda \sum_i |\theta_i|$$

  where:

  - $L_0(\theta)$ is the original loss function.
  - λ is the regularization parameter that controls the strength of the penalty.
  - θ_i represents the model parameters.

- **L2 Regularization**

  L2 regularization, also known as Ridge regularization, adds a penalty equal to the square of the magnitude of coefficients to the loss function. The updated loss function $L(\theta)$ with L2 regularization is given by:

$$L(\theta) = L_0(\theta) + \lambda \sum_i \theta_i^2$$

  where:

  - $L_0(\theta)$ is the original loss function.
  - λ is the regularization parameter.
  - θ_i represents the model parameters.

- **Dropout**

  Dropout is a technique used to randomly "drop out" or set to zero a fraction of neurons during training. This helps to prevent the network from becoming overly dependent on any one neuron and promotes better generalization. The dropout layer's output $\mathbf{h}_{drop}$ can be defined as:

$$\mathbf{h}_{drop} = \mathbf{h} \cdot \mathbf{r}$$

  where:

- **h** is the output from the previous layer.
- **r** is a binary mask vector generated by dropout, where each element is 0 with probability p and 1 with probability $1 - p$.
- p is the dropout rate.

During training, p is set to a value like 0.5, and during inference, dropout is turned off.

- **Early Stopping**

Early stopping is a technique where training is halted when the performance on a validation set starts to deteriorate, even if the model is still improving on the training set. This helps in preventing overfitting. Formally, early stopping involves monitoring a validation metric, such as the validation loss L_{val}, and stopping training if L_{val} does not improve for a specified number of epochs.

- **Batch Normalization**

Batch normalization normalizes the input of each layer to have a mean of zero and a standard deviation of one. This technique helps in stabilizing and accelerating training. The normalized output $\mathbf{h}_{norm}$ for a mini-batch is given by:

$$\mathbf{h}_{norm} = \frac{\mathbf{h} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \cdot \gamma + \beta$$

where:

- **h** is the input to the batch normalization layer.
- μ_B and σ_B^2 are the mean and variance of the mini-batch.
- ϵ is a small constant to avoid division by zero.
- γ and β are learnable parameters that scale and shift the normalized output.

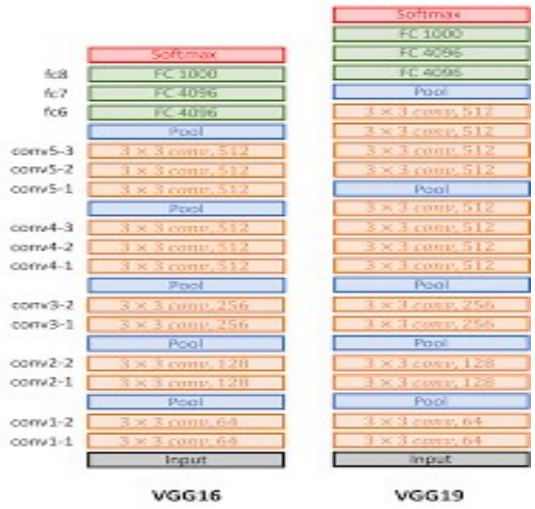
2.3 Important Deep Learning Architectures for image classification tasks

Image classification is a fundamental task in computer vision where the goal is to assign a label or category to an input image based on its content. This process involves training a model to recognize patterns and features within images, allowing it to make predictions about the image's content. Image classification can be categorized into two main types:

- **Binary Classification:** This involves distinguishing between two classes. For example, a model might be trained to classify images as either "cats" or "dogs."
- **Multiclass Classification:** This involves distinguishing among more than two classes. For instance, a model might classify images into multiple categories such as "cat," "dog," "horse," and "bird."

To achieve accurate image classification, various deep learning architectures have been developed. In the following sections, we will explore some of the significant architectures used for image classification tasks, including VGG16, VGG19, ResNet50, ResNet101, InceptionV3, DenseNet121, and MobileNetV2

- **VGG16**



Developed by the Visual Geometry Group at the University of Oxford, VGG16 is a convolutional neural network architecture known for its simplicity and uniform design. It consists of 13 convolutional layers and 3 fully connected layers, making a total of 16 weight layers. The network uses 3×3 convolutional filters and 2×2 max-pooling layers, allowing it to capture hierarchical features from the input images.

Fig 2.3 VGG16 vs VGG19

$$\text{Output} = \text{Softmax}(W_3 \cdot \left(\text{ReLU}\left(W_2 \cdot \left(\text{ReLU}\left(W_1 \cdot \text{Flatten}(\text{Conv}(X))\right)\right)\right)\right)$$

where W_1, W_2, W_3 are the weights of the fully connected layers, and $\text{Conv}(X)$ denotes the convolutional operation applied to the input image X.

- **VGG19**

  Also developed by the Visual Geometry Group at Oxford, VGG19 extends VGG16 by adding additional convolutional layers, resulting in a total of 19 layers. The architecture maintains the use of 3×3 convolutional filters and 2×2 max-pooling layers, allowing it to capture more complex features from images. The network also includes three fully connected layers like VGG16.

$$\text{Output} = \text{Softmax}(W_3 \cdot \left(\text{ReLU}\left(W_2 \cdot \left(\text{ReLU}\left(W_1 \cdot \text{Flatten}(\text{Conv}(X))\right)\right)\right)\right)$$

- **ResNet50**

  ResNet50 was introduced by Kaiming He and his colleagues in the paper "Deep Residual Learning for Image Recognition." It is part of the Residual Networks (ResNet) family and is known for its use of residual learning through skip connections. The architecture comprises 50 layers and utilizes residual blocks to bypass one or more convolutional layers, addressing the vanishing gradient problem and enabling the training of very deep networks. A residual block is formulated as:

$$\text{Output} = \text{ReLU}(F(X) + X)$$

  where $F(X)$ represents the residual function, often consisting of multiple convolutional layers.

- **ResNet101**
  ResNet101 extends ResNet50 by increasing the depth to 101 layers, allowing for more complex feature extraction and improved performance. The architecture uses the same residual blocks as ResNet50 but with additional layers to capture more detailed hierarchical features.

$$\text{Output} = \text{ReLU}(F(X) + X)$$

- **InceptionV3**
  InceptionV3 [6] was developed by Google researchers and introduced in the paper "Rethinking the Inception Architecture for Computer Vision." It employs the Inception module, which uses a combination of convolutions and pooling operations in parallel. The module applies 1×1 convolutions for dimensionality reduction and various 3×3 and 5×5 convolutions to capture multi-scale features. The overall architecture is designed for efficient computation and feature extraction.

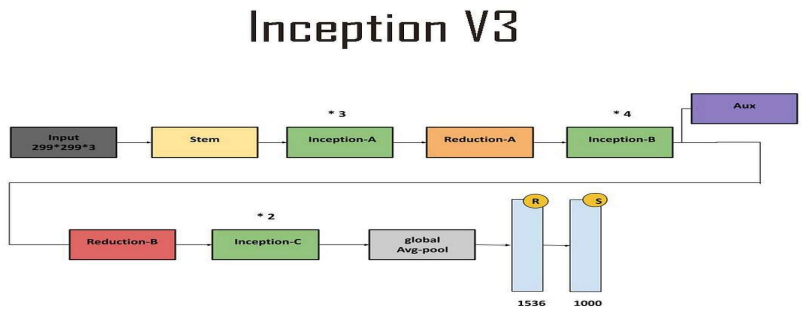


Fig 2.4 Inception V3 main Architecture.

$$\text{Inception Module} = \text{Concat}\left(\begin{matrix}\text{Conv}_1\left(\text{ReLU}\left(\text{BN}(X)\right)\right), \text{Conv}_3\left(\text{ReLU}\left(\text{BN}(X)\right)\right), \text{Conv}_5\left(\text{ReLU}\left(\text{BN}(X)\right)\right) \\ , \text{MaxPool}\left(\text{ReLU}\left(\text{BN}(X)\right)\right)\end{matrix}\right)$$

where BN stands for Batch Normalization and Concat denotes the concatenation of feature maps.

- **DenseNet121**
  DenseNet121, introduced by Gao Huang and colleagues in "Densely Connected Convolutional Networks," features dense connectivity between layers. Each layer receives inputs from all previous layers and passes its output to all subsequent layers. This architecture improves gradient flow and reduces the number of parameters compared to traditional convolutional neural networks. Each dense block is defined as:

$$\text{Output} = \text{DenseBlock}\left(\text{Concatenate}\left(\text{ReLU}\left(\text{BatchNorm}\left(\text{Conv}(X)\right)\right), \text{ReLU}\left(\text{BatchNorm}\left(\text{Conv}(X)\right)\right)\right)\right)$$

- **MobileNetV2**
  MobileNetV2 [7]was introduced by Google researchers in the paper "MobileNetV2: Inverted Residuals and Linear Bottlenecks." It is optimized for mobile and edge

devices, utilizing depth wise separable convolutions. This technique separates a standard convolution into a depth wise convolution followed by a pointwise convolution, significantly reducing the number of parameters and computational cost. The architecture also includes linear bottlenecks and residual connections to balance performance and efficiency.

$$\text{Depth wise Separable Convolution} = \text{DepthwiseConv} \circ \text{PointwiseConv}$$

where $\circ$ denotes the sequential application of depthwise and pointwise convolutions.

2.4 Transfer Learning in Deep Learning

Transfer Learning is a powerful technique in deep learning where a model developed for a particular task is reused as the starting point for a model on a second task. This approach leverages knowledge gained from solving one problem to address a different, but related problem. It is particularly useful when there is limited data available for the target task, allowing models to benefit from pre-trained models on large datasets.

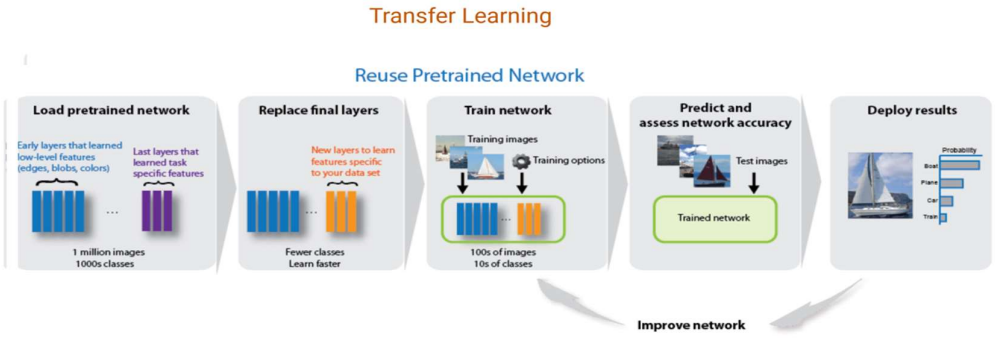


Fig 2.5 Transfer Learning[16]

In transfer learning, the key idea is to transfer knowledge from a source domain to a target domain. This typically involves using a pre-trained model on a large dataset and adapting it to the specific requirements of the target task. The transfer learning process generally involves three main steps:

- **Pre-training**: Train a model on a large dataset (source domain).
- **Feature Extraction**: Use the learned features from the pre-trained model as input for the target task.
- **Fine-tuning**: Adjust the pre-trained model's parameters to better fit the target domain.

Transfer learning can be categorized into several types depending on how the pre-trained model is used and adapted:

- **Feature Extraction**: Utilize the features extracted by the pre-trained model as inputs to a new model. In this case, the pre-trained model's weights are kept frozen, and only the final classifier is trained.

- **Fine-tuning**: Retrain the pre-trained model on the target task with a lower learning rate. This involves updating the weights of the model to adapt to the new data while preserving the knowledge from the source task.
- **Domain Adaptation**: Adapt a model trained on one domain to perform well on a different, but related, domain. This approach adjusts the model to account for differences in data distribution between the source and target domains.

In summary, transfer learning allows models to leverage knowledge from related tasks, improving performance and efficiency, especially in scenarios with limited data for the target task.

2.5 Active Learning

Active learning is a machine learning approach where the model actively selects the most informative data points for labelling to improve its performance. This technique is especially useful when acquiring labelled data is expensive or time-consuming. By focusing on the most uncertain or ambiguous examples, active learning reduces the amount of labelled data required to achieve high performance.

In active learning, the model employs various query strategies to identify which data points should be labelled next. For instance, uncertainty sampling involves selecting examples where the model is least confident, while query-by-committee uses multiple models to find samples with high disagreement. Another strategy, diversity sampling, aims to choose examples that represent diverse parts of the input space, ensuring a comprehensive training set.

Overall, active learning enhances model efficiency and reduces labelling costs by prioritizing the most informative samples. This approach is widely used in fields like medical imaging and natural language processing, where high-quality labelled data can significantly improve model performance.

3 Early Project Deliverables.

In this chapter, I present the early deliverables of my project, focusing on the initial experiments conducted to explore various deep learning models and techniques. The experiments were designed to provide a foundational understanding of neural networks, including the effects of different network architectures, optimization strategies, and regularization techniques. The insights gained from these early experiments guided the development of more complex models in later stages of the project.

For all my experiments, I utilized the CIFAR-10 dataset[8] , a widely recognized benchmark in machine learning and computer vision. The CIFAR-10 dataset comprises 60,000 32x32 colour images, categorized into 10 distinct classes, with each class containing 6,000 images. The dataset is split into 50,000 training images and 10,000 test images, providing a robust basis for evaluating the performance of various models.

The 10 classes represented in the CIFAR-10 dataset include airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. This diversity, coupled with the manageable size of the dataset, makes it ideal for experimenting with different machine learning and deep learning techniques. The dataset was originally created by Alex Krizhevsky and Geoffrey Hinton and has become a standard for testing and comparing

models in supervised learning tasks, where the objective is to accurately classify new images into one of the predefined classes.

This chapter outlines the step-by-step progression of experiments starting from the implementation of basic neural networks, moving through analysis of network complexities, to the evaluation of optimization and regularization methods. The results from these experiments form the foundation for the more advanced work detailed in subsequent chapters.

3.1 Implementation of Basic Models

3.1.1 Experimental Setup and Methodology.

In this section, I outline the steps taken to load the CIFAR-10 dataset, followed by the implementation of a simple Artificial Neural Network (ANN) and a Convolutional Neural Network (CNN). The evaluation of these models is primarily based on accuracy as the key performance metric.

Step 1: Data Loading and Preprocessing

The CIFAR-10 dataset, a well-known benchmark in image classification, consists of 60,000 32x32 colour images divided into 10 classes, with 6,000 images per class. The dataset is split into 50,000 training images and 10,000 test images.

- **Data Loading**: I utilized TensorFlow's datasets module to load the CIFAR-10 dataset. This dataset is inherently divided into training and test sets.

- **Normalization**: Given that pixel values range from 0 to 255, I normalized the images to a range of 0 to 1 by dividing by 255.0. This step is crucial for ensuring that the models converge faster during training.

Step 2: Simple Artificial Neural Network (ANN) Implementation

To establish a baseline, a straightforward Artificial Neural Network (ANN) was implemented on the CIFAR-10 dataset. This dataset consists of 32x32 pixel RGB images, flattened into a 3072-element vector (32x32x3 = 3072) for each image before being passed into the network.

- **Input Layer**: The ANN's input layer directly received the 3072-dimensional vectors representing the flattened images.

- **Hidden Layers:**

The first hidden layer comprised 128 neurons, using the ReLU activation function to introduce non-linearity.

The second hidden layer consisted of 64 neurons, also with ReLU activation.

- **Output Layer:** The final layer was configured with 10 neurons, corresponding to the 10 classes in the CIFAR-10 dataset, and used a softmax activation function to output the probability distribution over the classes.

The model was trained using categorical cross-entropy as the loss function, appropriate for multi-class classification tasks. Gradient Descent (SGD) was employed as the optimizer, with a learning rate of 0.01. The training process spanned 30 epochs, allowing the model to adjust its weights sufficiently. To monitor generalization performance, 10% of the training data was set aside for validation during training. The model utilized a batch size equal to the entire dataset (batch gradient descent).

Training Observations: The training and validation accuracy were tracked throughout the epochs to assess the model's learning behaviour and its ability to generalize to unseen data.

Step 3: Transition to Convolutional Neural Network (CNN)

Building on the results from the ANN, a Convolutional Neural Network (CNN) was implemented to better exploit the spatial structure of the image data. Unlike the ANN, the CNN processes the images in their original 32x32x3 format without flattening, allowing it to capture local patterns more effectively.

Convolutional Layers:

- The initial convolutional layer applied 32 filters with a 3x3 kernel size, followed by a ReLU activation function. This layer was designed to capture basic visual patterns such as edges.
- The second convolutional layer increased the number of filters to 64, again using a 3x3 kernel with ReLU activation, enabling the model to detect more complex features.
- Max-pooling layers were inserted after each convolutional layer to reduce the spatial dimensions, thus reducing computational load and mitigating overfitting by providing some translation invariance.

Fully Connected Layers:

- The output from the convolutional layers was flattened into a vector and passed through two fully connected layers.
- The first fully connected layer had 128 neurons, and the second had 64 neurons, both using ReLU activation to maintain the non-linear transformation.
- The output layer remained the same as in the ANN, with 10 neurons and a softmax activation function for class probability output.

For this model, categorical cross-entropy was again used as the loss function. The optimizer was Stochastic Gradient Descent (SGD), but this time, with a batch size of 64, striking a balance between convergence speed and the stochastic nature of updates. The training was conducted over 15 epochs, which was adequate given the complexity of the network and the necessity to avoid overfitting.

Training Observations: The performance on the training and validation datasets was monitored throughout to ensure the model was learning effectively and to detect any signs of overfitting or underfitting.

Evaluation Metrics

For both the ANN and CNN, accuracy was chosen as the primary evaluation metric. Accuracy, defined as the proportion of correct predictions to the total number of predictions, is particularly relevant for this multi-class classification problem. It provides a clear, interpretable measure of the model's performance across all classes.

In conclusion, the approach began with loading and normalizing the CIFAR-10 dataset to prepare the image data for training. By first implementing a simple Artificial Neural Network (ANN), a baseline accuracy was established, serving as a useful benchmark. Transitioning to a more advanced Convolutional Neural Network (CNN) architecture, which is inherently better suited for image data, resulted in a noticeable improvement in accuracy. Throughout the process, accuracy served as the primary metric for evaluating and comparing the models' performances. This comparison highlighted the significance of choosing the right architecture for specific data types, demonstrating the effectiveness of CNNs in handling image-based tasks.

3.1.2 Results and Performance Analysis

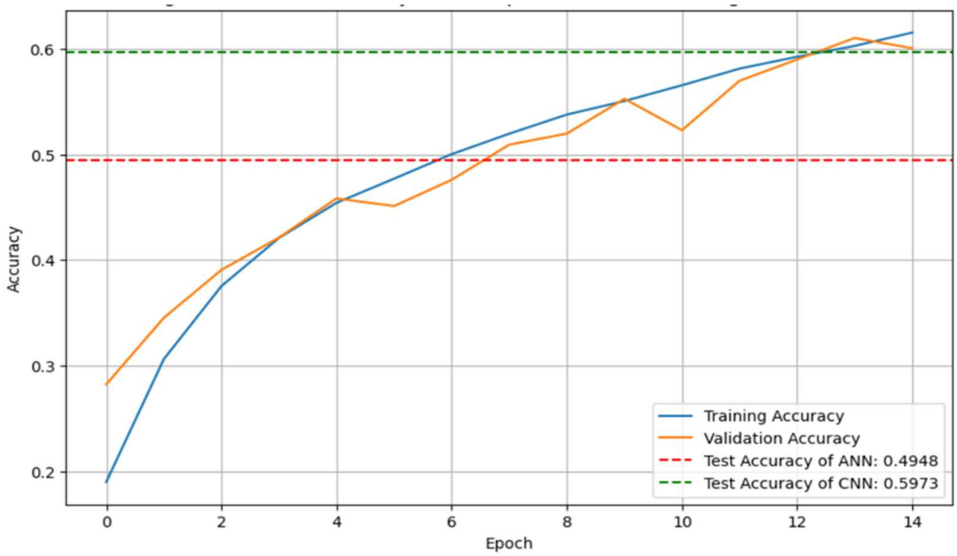


Fig 3.1 Accuracy vs Epoch plot for CNN

The results of the model training, as illustrated by the graph in figure 3.1, demonstrate a clear distinction in performance between the simple ANN and the more sophisticated CNN. The test accuracy of the ANN plateaued at approximately 0.4948, while the CNN achieved a test accuracy of 0.5973, indicating a marked improvement in performance with the more complex architecture. Notably, the graph shows a minimal gap between training and validation accuracy for both models, suggesting that overfitting was effectively managed in these implementations. However, despite the CNN's superior performance, the overall accuracy of both models remains suboptimal.

This is likely due to the basic nature of the models implemented, where advanced techniques such as hyperparameter tuning, optimization of parameters, or regularization methods were not fully utilized. These results underscore the potential for significant performance gains with more refined models and optimized training strategies. Therefore, while the CNN demonstrates the expected advantages over the ANN, the relatively modest accuracy levels achieved highlight the importance of further enhancing the model architecture and training processes to unlock the full potential of these approaches. This

suggests that, in the realm of image classification, even a transition to CNN from a basic ANN is only the beginning, and substantial improvements are achievable with more sophisticated techniques.

Moving forward, to further enhance model performance, I have conducted additional experiments detailed in Section 3.2, focusing on analysing the complexity of network architecture. In these experiments, the density of the layers was increased, exploring whether a greater number of neurons could lead to more efficient learning and better overall accuracy. This next step aims to investigate the impact of deeper and more intricate network designs, setting the stage for potentially unlocking higher performance levels.

3.2 Analysis of Network Architecture Complexity

3.2.1 Impact of Network Density on Model Performance

In this section, I explore how increasing the density of the network architecture both in an ANN and a CNN affects performance on the CIFAR-10 dataset. The focus is on understanding whether a higher number of layers and neurons within these layers can lead to improved model accuracy.

Dense ANN Architecture

The Dense ANN model was constructed using a total of 8 fully connected layers with varying neuron counts. Specifically, the model included the following layers:

- **Input Layer**: Flattened input shape of (32, 32, 3) to prepare the image data for the fully connected layers.
- **Hidden Layers**: The architecture comprised layers with the following neuron configurations: 256, 512, 512, 256, 128, 128, 64, and 64 neurons, each utilizing the ReLU activation function.
- **Output Layer**: A final layer with 10 neurons using the softmax activation function to classify the 10 categories in the CIFAR-10 dataset.

The model was trained using stochastic gradient descent (SGD) as the optimizer, with the loss function set to sparse categorical cross-entropy, a typical choice for multi-class classification problems. The training was conducted over 15 epochs. This number of epochs was chosen as a starting point based on common practice in similar image classification tasks. Fifteen epochs often provide a good balance between allowing the model enough time to learn meaningful patterns without risking excessive overfitting. This selection allows for initial observation of the model's performance trends, with the possibility of adjusting based on the results.

Performance Observations:

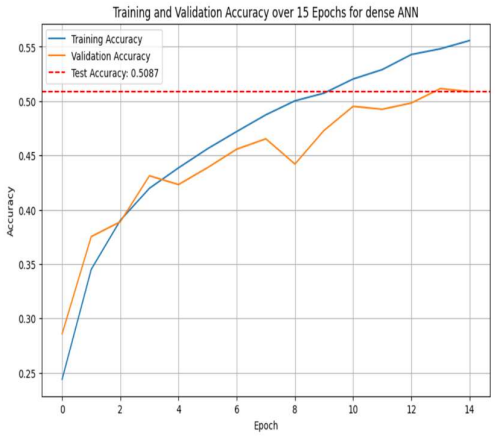


Fig 3.2 Accuracy vs Epoch plot for Dense ANN

- **Training Accuracy**: The accuracy steadily improved across epochs, starting from 18.63% and reaching 55.80% by the end of training.
- **Validation Accuracy**: The validation accuracy also showed consistent improvement, but the final accuracy was slightly lower than training, indicating the model's capacity to generalize.
- **Test Accuracy**: The final test accuracy achieved was 50.87%, which, although improved from the initial epochs, is still indicative of the limitations inherent in this dense architecture.

Analysis:

- **Overfitting Concerns**: The close alignment between training and validation accuracy curves suggests that overfitting was not a significant issue in this dense ANN architecture.
- **Model Complexity vs. Performance**: Despite the increased number of neurons and layers, the final performance of the dense ANN reached a plateau, suggesting that simply adding more layers and neurons might not suffice in significantly enhancing performance. This plateau could indicate the limits of dense architectures for complex image classification tasks like CIFAR-10.

Dense CNN Architecture

For comparison, a dense CNN model was also constructed, emphasizing the use of convolutional layers, which are particularly well-suited for image data due to their ability to capture spatial hierarchies.

Architecture:

- **Convolutional Layers**: The model began with two convolutional layers with 256 filters each, followed by a layer with 128 filters. Each convolutional layer used 3x3 filters with ReLU activation and same padding to preserve the input dimensions.
- **Pooling Layers**: MaxPooling layers followed the first two convolutional layers to reduce the spatial dimensions.
- **Fully Connected Layers**: The network concluded with a series of fully connected layers with 128, 128, 64, and 64 neurons, leading into a final softmax layer for classification.

Performance Observations:

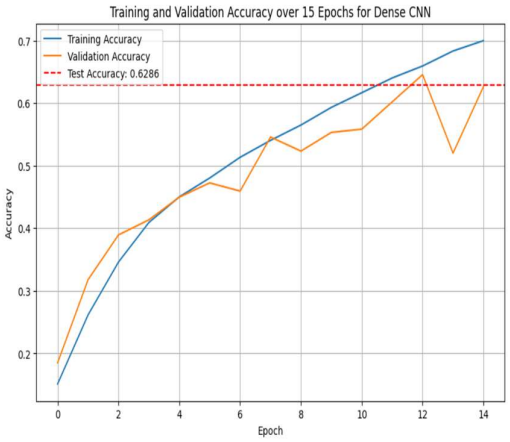


Fig 3.3 Accuracy vs Epoch plot for Dense CNN

- **Training Accuracy**: The training accuracy demonstrated a steady increase, culminating at 69.92% by the end of the 15 epochs.
- **Validation Accuracy**: Validation accuracy, although slightly lagging training accuracy, indicated reasonable generalization with a final score of 62.63%.
- **Test Accuracy**: The final test accuracy was 62.86%, which, compared to the dense ANN, represents a notable improvement.

Analysis:

- **Enhanced Feature Extraction**: The dense CNN significantly outperformed the dense ANN, highlighting the effectiveness of convolutional layers in extracting and utilizing spatial features from the input images.
- **Impact of Layer Density**: Like the dense ANN, the CNN also showed that adding more layers and filters, while beneficial to some extent, eventually led to diminishing returns. The model's performance plateaued, indicating the necessity of further optimization or architectural changes for significant gains.

The results from both the dense ANN and dense CNN experiments indicate that while increasing network density can improve model performance, it alone is insufficient for achieving high accuracy, especially on complex datasets like CIFAR-10. The dense CNN outperformed the dense ANN, suggesting that convolutional layers are more effective at handling image data. However, the observed performance plateau in both models implies that simply increasing the number of layers and neurons is not enough; further refinement in architecture design, such as incorporating dropout layers, batch normalization, or advanced optimization techniques, could be necessary to push the model's accuracy beyond the current limits.

3.2.2 Evaluation of Activation Functions

This experiment was conducted to analyse the model's performance when different activation functions ReLU, ELU, SELU, Tanh, and Sigmoid—are used individually. The CIFAR-10 dataset was loaded and pre-processed, followed by a split into training, validation, and test sets. The neural network architecture consisted of multiple fully connected layers, and the only varying factor across experiments was the activation function used in these layers. The five activation functions tested were:

- **ReLU (Rectified Linear Unit)**:

$$\text{ReLU}(x) = max(0, x)$$

This function outputs x if x is greater than zero; otherwise, it outputs zero.

- **Tanh (Hyperbolic Tangent)**:

$$\text{Tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

This function outputs values between -1 and 1, providing a smooth gradient and zero-centred output.

- **ELU (Exponential Linear Unit)**:

$$\text{ELU}(x) = \begin{cases} x & \text{if} x > 0 \\ \alpha(e^x - 1) & \text{if} x \leq 0 \end{cases}$$

where α is a hyperparameter that controls the value to which the ELU saturates for negative inputs.

- **SELU (Scaled Exponential Linear Unit)**:

$$\text{SELU}(x) = \lambda \begin{cases} x & \text{if} x > 0 \\ \alpha(e^x - 1) & \text{if} x \leq 0 \end{cases}$$

where $\alpha \approx 1.6733$ and $\lambda \approx 1.0507$. The SELU function is designed to self-normalize the outputs.

- **Sigmoid**:

$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

This function outputs values between 0 and 1, providing a smooth gradient but is susceptible to vanishing gradients for large positive or negative inputs.

The results from these experiments are plotted in the figure, showing the maximum validation accuracy and test accuracy for each activation function.

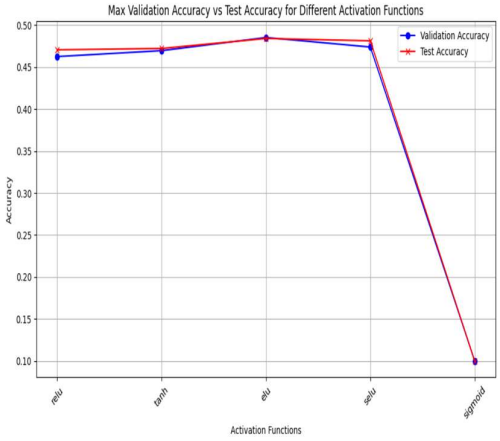


Fig 3.4 Accuracy vs Activation functions plot.

The plot in figure 3.4 shows the performance comparison:

•ReLU: Achieved a validation accuracy of 0.4625 and a test accuracy of 0.4705. ReLU is known for its simplicity and efficiency in many tasks, which likely contributed to its strong performance here.

•Tanh: Performed similarly to ReLU, with slightly higher validation (0.4695) and test accuracy (0.4721). Tanh can capture more non-linearity in data than ReLU, which might explain its competitive performance.

- ELU: Outperformed ReLU and Tanh slightly with a validation accuracy of 0.4851 and a test accuracy of 0.4840. ELU's capability to produce negative values could have contributed to better gradient flow during training, resulting in improved accuracy.

- SELU: Provided a competitive validation accuracy of 0.4738 and a test accuracy of 0.4812. SELU is designed to maintain the self-normalizing properties of neural networks, which might explain its strong performance.

- Sigmoid: Performed the worst, with both validation and test accuracies around 0.1000. This poor performance is expected because the sigmoid function often suffers from the vanishing gradient problem, especially in deeper networks, making it less effective for CIFAR-10, a dataset with complex patterns.

From the experiments, ELU and SELU emerged as the top-performing activation functions, with ReLU and Tanh also showing solid results. Sigmoid was notably the worst performer due to its limitations in handling complex data patterns, as seen in CIFAR-10.

While this experiment provided valuable insights, it could have been further extended by exploring combinations of different activation functions within the network layers. However, due to time constraints and the need to proceed with other hyperparameter experiments, these additional experiments were not conducted.

3.3 Experimenting with Epochs, weight initialization techniques and Data Splitting ratios.

3.3.1 Determining optimal number of Epochs

In deep learning, determining the optimal number of epochs is crucial for achieving a balance between underfitting and overfitting. The goal is to train the model sufficiently without allowing it to overfit to the training data, which would reduce its generalization capability on unseen data.

The CIFAR-10 dataset was used to train two distinct models: a densely connected ANN and a dense CNN. For the ANN, training was conducted over 250 epochs, while the CNN was trained for 150 epochs. Both models were trained using stochastic gradient descent (SGD) as the optimizer with a learning rate of 0.01.

As shown in the fig 3.1, the training accuracy for the ANN increases rapidly during the early epochs, reaching approximately 70-80% accuracy by epoch 70. This indicates that the model has effectively learned the training data within the first 80 epochs. However, as training continues beyond this point, the validation accuracy stabilizes around 54%, showing no further improvement, despite the increase in training accuracy. This suggests that extending training beyond 80 epochs does not contribute to better generalization but rather leads to overfitting. The gap between the training accuracy and validation accuracy highlights this overfitting, where the model becomes too specialized in the training data and fails to generalize well to new data.

In the case of the CNN, as depicted in the fig 3.2, the training accuracy reaches its peak much faster, around 40 epochs. This rapid convergence indicates that the CNN, being a more complex model better suited for image data, can learn effectively in fewer epochs. However, like the ANN, continuing to train the CNN beyond this point does not yield any significant improvement in validation accuracy, which remains around 72%. The persistent gap between training and validation accuracy towards the latter part of the training indicates that the model is overfitting as well.

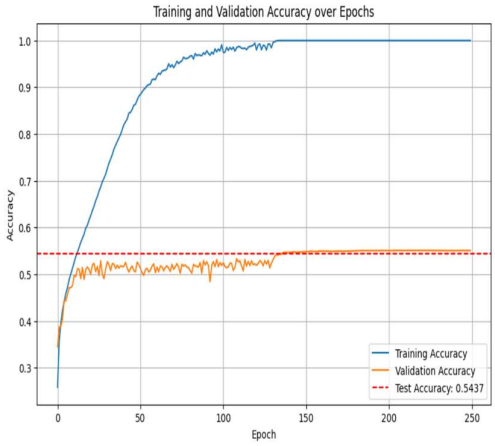


Fig 3.5 ANN accuracy plot for 250 epochs.

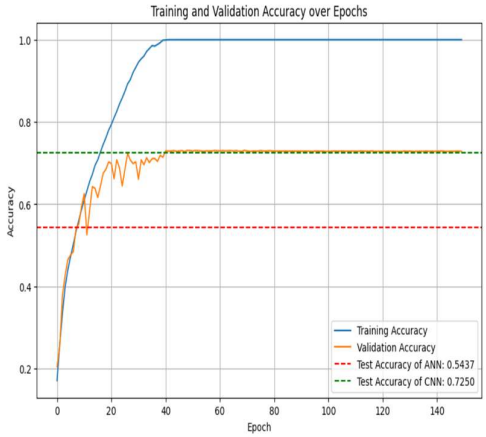


Fig 3.6 CNN accuracy plot for 150 epochs.

The analysis of epochs in this experiment as shown in figures 3.5 and 3.6 reveals that the maximum training accuracy for the ANN is reached within 70-80 epochs, while the CNN achieves its peak much earlier, around 40 epochs. This suggests that more complex models like CNNs can learn effectively over fewer epochs compared to simpler models like ANNs. However, extending training beyond these points leads to overfitting, where there is no improvement in validation accuracy despite continued improvements in training accuracy.

To avoid overfitting and achieve a more generalized model, it is recommended to limit the number of epochs to a range of 30-40, depending on the complexity of the model. Additionally, employing regularization techniques such as dropout, early stopping, or batch normalization can help improve validation accuracy and prevent the model from becoming too specialized to the training data.

3.3.2 Effect of weight Initializing techniques on models' performance.

Weight initialization is a crucial factor in the training process of deep learning models. It significantly affects the convergence rate, stability, and ultimately the accuracy of the model. In this section, I explore the impact of different weight initialization techniques on the performance of a simple Artificial Neural Network (ANN) trained on the CIFAR-10 dataset. The goal is to determine which initialization strategy leads to the best generalization, as indicated by validation and test accuracy. I experimented with several weight initialization methods using a basic ANN model. The following weight initialization techniques [9] were tested:

- **Zero Initialization**:

$$W_{ij} = 0$$

All weights W_{ij} initialized to zero. Leads to no learning due to symmetry.

- **Random Normal Initialization**:

$$W_{ij} \sim N(\mu, \sigma^2)$$

Weights initialized from a normal distribution with mean $\mu = 0$ and variance σ^2.

- **Random Uniform Initialization**:

$$W_{ij} \sim U(a, b)$$

Weights initialized uniformly in range $[a, b]$.

- **Xavier/Glorot Uniform Initialization**:

$$W_{ij} \sim U\left(-\sqrt{\frac{6}{n_{in} + n_{out}}}, \sqrt{\frac{6}{n_{in} + n_{out}}}\right)$$

Uniform initialization scaled by n_{in} (input units) and n_{out} (output units).

- **He Uniform Initialization**:

$$W_{ij} \sim U\left(-\sqrt{\frac{6}{n_{in}}}, \sqrt{\frac{6}{n_{in}}}\right)$$

Uniform initialization scaled by n_{in} (input units).

- **Xavier/Glorot Normal Initialization**:

$$W_{ij} \sim N\left(0, \frac{2}{n_{in} + n_{out}}\right)$$

Normal initialization with variance scaled by n_{in} and n_{out}.

- **He Normal Initialization**:

$$W_{ij} \sim N\left(0, \frac{2}{n_{in}}\right)$$

Normal initialization with variance scaled by n_{in}.

The performance of each initialization method was evaluated based on the maximum validation accuracy achieved during training and the final test accuracy on unseen data.

The results in figure 3.7 highlights the critical role of weight initialization in neural network training. Zero initialization led to the model being unable to learn, while simple random

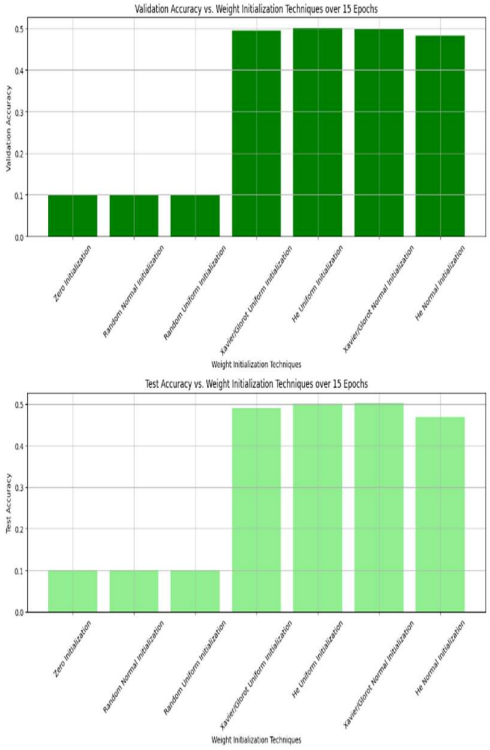


Fig 3.7 performance comparison of different weight initialization techniques

initialization methods were insufficient for this setup. In contrast, the Xavier/Glorot and He initialization techniques, which are designed to maintain the scale of gradients throughout the layers of the network, enabled the model to achieve significantly better accuracy. The gap between validation and test accuracy remained relatively small for the best-performing initializations, suggesting that these methods also contribute to better generalization, reducing the risk of overfitting.

The choice of weight initialization has a profound impact on the training dynamics and final performance of a neural network. For the ANN model trained on CIFAR-10, Xavier/Glorot and He initialization methods outperformed others, achieving validation and test accuracies around 50%. These techniques should be preferred in similar contexts to enhance model training efficiency and generalization

3.3.3 Effect of Data split ratios on models' performance.

In machine learning, generally splitting the dataset into training, validation, and test sets is crucial for evaluating a model's performance. The training set is used to fit the model, the validation set helps in tuning the model and preventing overfitting, and the test set provides an unbiased evaluation of the model's final performance. The selection of these split ratios can significantly impact the effectiveness of the model, so I experimented with various common split ratios to determine their influence on performance.

I tested several split ratios [70:15:15], [80:10:10], [90:05:05], [60:20:20], [50:25:25], [40:30:30], [75:15:10], [85:10:05], [65:20:15]. These ratios were selected to cover a wide range of possible configurations, from training-heavy to validation/test-heavy splits.

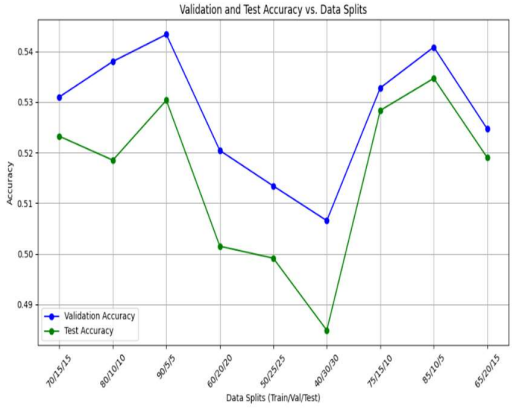


Fig 3.8 Test Accuracy vs data split ratios.

The plot in figure 3.8 illustrates the validation and test accuracy across different data split ratios. The results show considerable variability, with no clear optimal split ratio emerging as consistently superior. Among the above ratios, the 80:10:10 and 70:15:15 splits performed relatively well, with validation and test accuracies generally higher compared to other configurations. These ratios seem to provide a balanced approach, allowing sufficient data for training while keeping enough data for

validation and testing. On the other hand, the 90:05:05 split achieved the highest validation accuracy, but the test accuracy was not as impressive, suggesting possible overfitting due to the minimal validation set. The 40:30:30 split showed the lowest test accuracy, highlighting that too much data in the validation and test sets might not leave enough data for effective training.

The experiment did not reveal a universally optimal data split ratio, as performance varied across different configurations. However, the 80:10:10 and 70:15:15 splits were generally the most effective, providing a good balance between training, validation, and test data. These findings suggest that while the data split ratio is important, it might need to be adjusted based on the specific dataset and model. Due to time constraints, I concluded that these ratios were the most reliable for this experiment, though further testing could potentially refine these results.

3.4 Optimization and Regularization Techniques

3.4.1 Optimization Techniques.

As discussed in Section 2, optimization algorithms play a crucial role in training neural networks by minimizing the loss function and improving the model's performance. Several popular optimizers, such as Full Batch Gradient Descent (GD), Stochastic Gradient Descent (SGD), Mini-Batch SGD, Momentum, RMSprop, and Adam, each have unique characteristics that influence their effectiveness in different scenarios. The choice of optimizer, combined with the right learning rate, significantly affects the model's convergence speed and final accuracy.

Initially, I conducted experiments with each optimizer using a single learning rate to identify overall performance trends. After identifying the top-performing optimizers, I ran a series of experiments using various learning rates [0.001,0.003,0.006,0.01,0.03,0.06,0.1,0.3,0.6] to observe how the combination of learning rate and optimizer affects model performance.

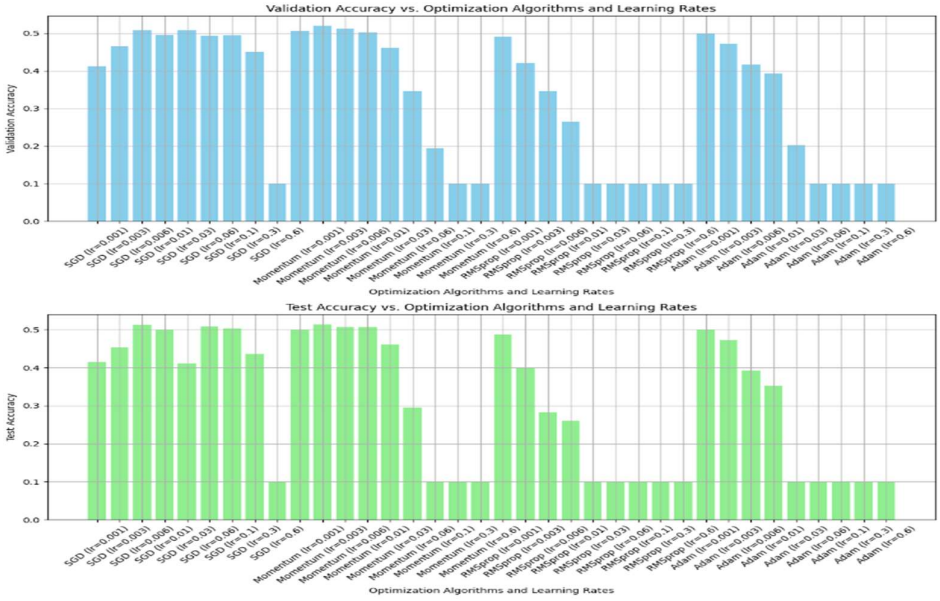


Fig 3.9 Accuracy plots for determining the effect of optimizers and learning rate

Figure 3.9 display the results, showing the validation and test accuracies for each combination of optimizer and learning rate.

Analysis and Conclusion of the plot:

- **SGD (Stochastic Gradient Descent)**: The results indicate that SGD performs best within the learning rate range of 0.01-0.06. These moderate learning rates allow the model to converge efficiently without overshooting the minimum or converging too slowly.
- **Momentum**: Like SGD, Momentum performs well with learning rates between 0.001-0.03. The addition of momentum helps to accelerate convergence, particularly in the presence of high variance in the gradients, making this range optimal for balancing speed and stability.
- **RMSprop**: For RMSprop, the most effective learning rates are slightly lower, in the range of 0.001-0.03. RMSprop's adaptive learning rate mechanism works best when it starts with a lower rate, allowing it to adjust dynamically based on the gradient magnitudes, ensuring stable progress during training.
- **Adam**: Adam, known for its robustness and adaptability, performs best within the learning rate range of 0.001-0.01. These lower to moderate learning rates help Adam leverage its adaptive moment estimation effectively, leading to efficient and stable convergence.

The results clearly demonstrate that the effectiveness of an optimizer is closely tied to the chosen learning rate. Moderate learning rates generally worked best across different optimizers, with Momentum, RMSprop, and Adam consistently achieving higher validation and test accuracies. Conversely, extreme learning rates, whether too high or too low, led to suboptimal performance across most optimizers. The experiment underscores the importance of tuning both the optimizer and learning rate together to achieve optimal performance. For SGD moderate learning rates in the range of were most effective and for momentum little lesser rates were effective. RMSprop and Adam performed best with slightly lower learning rates. This suggests that careful tuning of both parameters is necessary for achieving the best model performance, rather than relying on one to compensate for the other.

3.4.2 Regularization Techniques

Regularization techniques are essential in deep learning to prevent overfitting and to improve the generalization ability of models. By applying different regularization strategies, one can control the complexity of the model, thus ensuring that it performs well not only on the training data but also on unseen data.

For this experiment, I implemented and compared the effects of various regularization techniques: L1 regularization, L2 regularization, Dropout, Batch Normalization, and Early Stopping. The goal was to study how each of these techniques impacts the training, validation, and test accuracies of a convolutional neural network (CNN) model. The plot

depicted in figure 3.10 shows the training, validation, and test accuracies for each regularization technique.

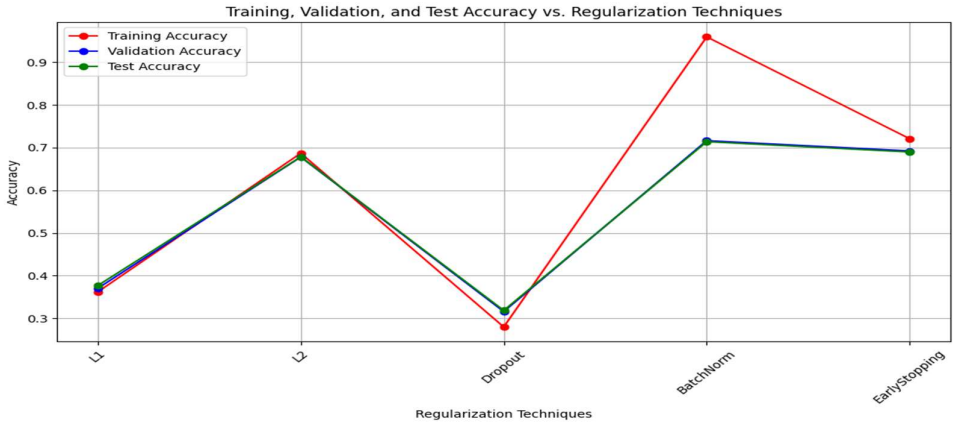


Fig 3.10 Accuracy vs Regularization techniques.

- **L1 Regularization**: The training accuracy was relatively low, around 36%, with similar validation and test accuracies. This suggests that L1 regularization significantly constrained the model's capacity, leading to underfitting.
- **L2 Regularization**: This technique resulted in better performance, with a training accuracy of approximately 69% and similar validation and test accuracies. L2 regularization seems to have struck a balance between model complexity and generalization.
- **Dropout**: Dropout led to the lowest accuracies across the board (around 28-32%), indicating that the model might have struggled to learn effectively due to the high dropout rate of 50%.
- **Batch Normalization**: This technique achieved the highest training accuracy (around 96%) and significantly improved validation and test accuracies (around 71%), making it the most effective regularization technique in this experiment. The high training accuracy suggests the model was able to learn effectively while still generalizing well.
- **Early Stopping**: Early Stopping provided a balanced approach with training, validation, and test accuracies around 72%, 69%, and 69%, respectively. This suggests that stopping the training at the right time helps prevent overfitting while maintaining good generalization.

Among the regularization techniques tested, Batch Normalization stood out as the most effective, significantly improving both the training and generalization performance of the model. L2 regularization also performed well, offering a balanced approach to regularization. On the other hand, Dropout, especially at a high rate, and L1 regularization led to underfitting, with lower overall accuracies. Early Stopping provided a good trade-off, preventing overfitting by halting training at the optimal point. These results underscore the importance of choosing the right regularization strategy to ensure model robustness and generalization.

4 Face Mask Detection Using Deep Learning

4.1 Introduction

4.1.1 Aim

The aim of this chapter is to develop an effective and robust deep learning-based solution for face mask detection, a task that has become increasingly important in the context of public health and safety. The COVID-19 pandemic has underscored the critical need for technologies that can help enforce health protocols, such as the wearing of face masks, to mitigate the spread of the virus. Automated face mask detection systems can play a pivotal role in public spaces, workplaces, and healthcare settings by providing real-time monitoring and ensuring compliance with mask-wearing guidelines.

This project seeks to leverage advanced deep learning techniques to build a model capable of accurately distinguishing between masked and unmasked faces in images. By utilizing real-world datasets that reflect the diversity and complexity of actual scenarios, the project aims to create a solution that is not only accurate but also adaptable to different environments and conditions. Multiple neural network architectures, including both custom-built models and those utilizing transfer learning from pre-trained models, will be explored to determine the most effective approach. The goal is to contribute a reliable tool for face mask detection that can support ongoing efforts to protect public health.

4.1.2 Objectives

- **Work with Real Datasets**: Utilize and process real-world datasets specifically curated for face mask detection to ensure the robustness and applicability of the developed models in practical scenarios.
- **Implement and Compare Multiple Architectures**: Develop, implement, and compare the performance of multiple deep learning architectures, including custom models and pre-trained models (using transfer learning techniques).
- **Visualize and Assess Performance**: Visualize the results and assess the performance of the different models using various metrics such as accuracy, precision, recall, and F1-score. This objective ensures that the strengths and weaknesses of each approach are thoroughly understood.

4.2 Data Preparation

The quality and preparation of data are critical steps in building an effective deep learning model, particularly for a task as nuanced as face mask detection. For this project, I utilized the "Face Mask 12k Images Dataset" from Kaggle, which provided a diverse set of images depicting individuals with and without face masks. To ensure that the model was trained on high-quality data, I carefully curated and pre-processed the dataset, leading to a more refined and informative collection of images.

4.2.1 Data Loading and Preprocessing

Initially, the dataset comprised many images that varied widely in quality and relevance. To enhance the dataset's effectiveness, I conducted a thorough selection process, removing uninformative or low-quality images. This refinement step ensured that the final dataset contained only high-quality images that would contribute meaningfully to the model's learning process.

Initially, the dataset [10] comprised many images that varied widely in quality and relevance. To enhance the dataset's effectiveness, I conducted a thorough selection process, removing uninformative or low-quality images. This refinement step ensured that the final dataset contained only high-quality images that would contribute meaningfully to the model's learning process. Throughout this project, I employed two different datasets based on the experimental requirements:

- **Unrefined Large Dataset**:
  - **Training Set**: 8022 images belonging to 2 classes.
  - **Validation Set**: 1746 images belonging to 2 classes.
  - **Test Set**: 2024 images belonging to 2 classes.

This larger dataset was primarily used for initial model development and basic model implementations, where having a larger volume of data was crucial for training the models and obtaining a broad understanding of the problem.

- **Refined Smaller Dataset**:
  - **Training Set**: 1600 images with masks and 1600 images without masks.
  - **Validation Set**: 200 images with masks and 200 images without masks.
  - **Test Set**: 200 images with masks and 200 images without masks.

This refined dataset, following a **70:15:15** split ratio, was specifically used for experiments involving robust pre-trained models and advanced deep learning techniques. The smaller, high-quality dataset allowed for more precise training and fine-tuning of models, ensuring that the trained models were highly performant on critical cases.

Data Preprocessing:

- **Rescaling**: All images were rescaled to normalize the pixel values, ensuring consistency in the data fed into the model.
- **Image Resizing**: Each image was resized to 128x128 pixels, a standard size for input into convolutional neural networks (CNNs).
- **Batch Processing**: The data was processed in batches to optimize the training process.

These preprocessing steps were crucial for preparing the data for effective training, ensuring that the model could learn from clear, high-quality images that accurately

Fig 4.1 Few sample data points.

represented the target classes. To gain a better understanding of the dataset, I visualized a sample of images from the training set. This step was important for verifying that the

images were correctly labelled and diverse enough to train a model capable of generalizing well to unseen data.

4.2.2 Class Distribution Analysis

A balanced class distribution is critical for ensuring that the model does not develop a bias toward one class over the other. To analyse the distribution of classes across the training, validation, and test datasets, I conducted a class distribution analysis. The results, as shown in the plot, indicate a balanced distribution across all datasets, with an equal number of images for both classes (with mask, without mask).

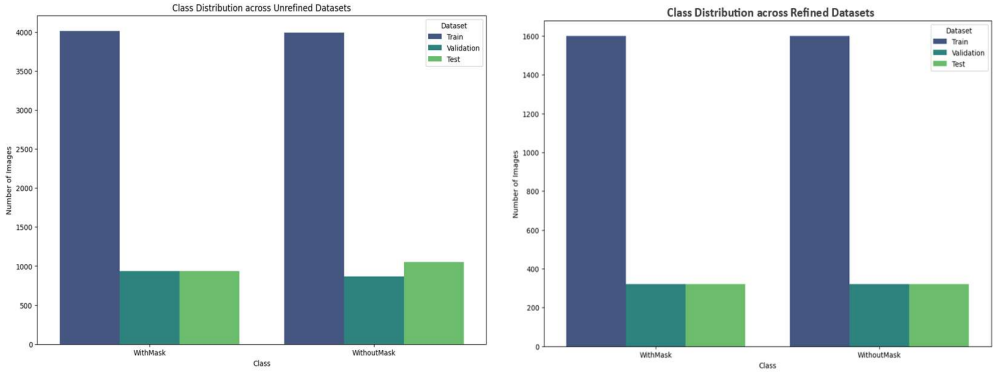


Fig 4.2 Class Distribution plots of unrefined and refined data.

This balance was maintained throughout the different datasets to prevent any bias during training, thus ensuring that the model could learn to detect both classes effectively, even in the presence of challenging scenarios such as facial hair or patterned masks.

4.2.3 Real-Time Data Collection

In addition to the curated dataset, I also collected a set of real-time images. These images were not included in the training, validation and regular testing process and were reserved for a final, completely unseen test of the model's performance. This real-time data represents current conditions and includes additional challenging scenarios, ensuring a rigorous evaluation of how well the model generalizes to new, unseen environments. I made a deliberate effort to include special cases in the dataset to challenge the model's ability to distinguish between masked and unmasked faces accurately. These special cases included:

- **Facial Hair**: Images where facial hair might be confused as a mask.
- **Tanned Faces**: Images with tanning that might obscure facial features, potentially confusing the model.
- **Masks with Patterns**: Masks that have patterns resembling facial features, such as mouths or teeth, which could be misinterpreted as unmasked faces.

4.3 Building and Training the Models

4.3.1 Training with a Custom Model

In this section, I built and trained a custom neural network from scratch without relying on any high-level machine learning libraries for the model architecture. This approach allowed for a deeper understanding of the underlying mechanics of neural networks, including how weights are updated during training and how hyperparameters influence the learning process.

Before delving into the detailed explanation of the process, I first outlined the entire approach in the form of pseudocode or an algorithm. This structured outline served as the foundation, guiding the subsequent development of the code. By mapping out each step-in advance, I was able to ensure that the implementation was both logical and efficient, leading to a more robust final model.

```
Algorithm: Train_and_Evaluate_Neural_Network

Input:
    - IMG_HEIGHT = 32
    - IMG_WIDTH = 32
    - BATCH_SIZE = 32
    - input_size = 3072 (32 * 32 * 3)
    - hidden1_size = 128
    - hidden2_size = 64
    - output_size = 1
    - epochs = 30
    - learning_rate = 0.01

Data_Preparation:
    train_data ← Load and preprocess data from 'Train'
directory
    val_data ← Load and preprocess data from 'Validation'
directory
    test_data ← Load and preprocess data from 'Test' directory

Model_Initialization:
    Initialize w1, w2, w3 using He initialization
    Initialize b1, b2, b3 as zero vectors

for epoch = 1 to epochs do
    Forward_Pass:
        z1 ← train_data * w1 + b1
        a1 ← ReLU(z1)
        z2 ← a1 * w2 + b2
        a2 ← ReLU(z2)
        z3 ← a2 * w3 + b3
        a3 ← Sigmoid(z3)

    Compute_Loss:
        loss ← Binary_Cross_Entropy(train_labels, a3)

    Backward_Pass:
        dz3 ← Binary_Cross_Entropy_Derivative(train_labels,
a3)
        dw3 ← a2^T * dz3
        db3 ← sum(dz3)
        dz2 ← dz3 * w3^T * ReLU_Derivative(z2)
        dw2 ← a1^T * dz2
        db2 ← sum(dz2)
```

```
        dz1 ← dz2 * w2^T * ReLU_Derivative(z1)
        dw1 ← train_data^T * dz1
        db1 ← sum(dz1)

    Update_Weights:
        w1 ← w1 − learning_rate * dw1
        b1 ← b1 − learning_rate * db1
        w2 ← w2 − learning_rate * dw2
        b2 ← b2 − learning_rate * db2
        w3 ← w3 − learning_rate * dw3
        b3 ← b3 − learning_rate * db3

    Compute_Train_Accuracy:
        train_preds ← (a3 > 0.5)
        train_acc ← mean(train_preds == train_labels)

    Compute_Validation_Accuracy:
        z1_val ← val_data * w1 + b1
        a1_val ← ReLU(z1_val)
        z2_val ← a1_val * w2 + b2
        a2_val ← ReLU(z2_val)
        z3_val ← a2_val * w3 + b3
        a3_val ← Sigmoid(z3_val)
        val_preds ← (a3_val > 0.5)
        val_acc ← mean(val_preds == val_labels)

End for

Model_Evaluation:
    z1_test ← test_data * w1 + b1
    a1_test ← ReLU(z1_test)
    z2_test ← a1_test * w2 + b2
    a2_test ← ReLU(z2_test)
    z3_test ← a2_test * w3 + b3
    a3_test ← Sigmoid(z3_test)
    test_preds ← (a3_test > 0.5)
    test_acc ← mean(test_preds == test_labels)

Output:
    Print test_acc
```

Data Preparation:

The first step in training the custom neural network was preparing the dataset. The data was loaded from three directories: Train, Validation, and Test. These directories contained images of faces categorized into two classes: "With Mask" and "Without Mask."

- **Rescaling:** All images were rescaled by a factor of 1/255 to normalize the pixel values, converting them from a range of 0-255 to 0-1. This normalization is

crucial for ensuring that the model trains efficiently without being influenced by large input values.

- **Image Sizing:** The images were resized to 32x32 pixels. This lower resolution was chosen to simplify computations while still retaining essential features for classification.
- **Batch Processing:** The images were processed in batches of 32, which is standard practice in deep learning to make training more efficient.

Model Architecture:

- **Input Layer:** The input layer accepts a flattened version of the image. Each 32x32 image with 3 color channels (RGB) was flattened into a vector of size 3072 (32 * 32 * 3).
- **Hidden Layers:**
    - The first hidden layer had 128 neurons, using the ReLU (Rectified Linear Unit) activation function. ReLU was chosen because it helps in mitigating the vanishing gradient problem and accelerates convergence.
    - The second hidden layer consisted of 64 neurons, also using the ReLU activation function.
- **Output Layer:** The output layer had a single neuron with a sigmoid activation function, which is appropriate for binary classification problems. The output of the sigmoid function is a probability score between 0 and 1, which is then thresholded to classify the input as either "With Mask" or "Without Mask."

Weight Initialization:

Weights for the layers were initialized using a variant of the He initialization method, specifically designed for layers using the ReLU activation function. The initialization involved setting the weights to small random values scaled by the square root of 2 divided by the number of input units to the layer, ensuring that the variance of the activations remains consistent across layers.

Forward Pass:

- **Layer 1:** The input data was multiplied by the first set of weights, and a bias term was added. The result was then passed through the ReLU activation function.
- **Layer 2:** The output from Layer 1 was multiplied by the second set of weights, with an added bias term, and again passed through the ReLU activation function.
- **Output Layer:** The output from Layer 2 was multiplied by the third set of weights and passed through the sigmoid activation function, producing a final output value.

Loss Function:

The binary cross-entropy loss function measures the difference between predicted probabilities and actual labels in binary classification tasks. Mathematically, for a single example with label y (either 0 or 1) and predicted probability $\hat{y}$, the loss is defined as:

$$\text{Binary Cross-Entropy} = -[y \cdot \log(\hat{y}) + (1 - y) \cdot \log(1 - \hat{y})]$$

This loss function penalizes the model more heavily when the predictions are incorrect, thus encouraging the model to improve its accuracy. For a dataset with n examples, the average binary cross-entropy loss is used:

$$Loss = -\frac{1}{n}\sum_{i=1}^{n}[y_i \cdot log(\hat{y}_i) + (1 - y_i) \cdot log(1 - \hat{y}_i)]$$

Backward Pass (Backpropagation):

The backward pass involved calculating the gradients of the loss function with respect to each weight in the network using backpropagation:

- **Gradient Calculation:** Gradients for each layer were computed using the chain rule, involving the derivatives of the loss function and the activation functions (sigmoid for the output layer and ReLU for the hidden layers).
- **Weight Update:** After computing the gradients, the weights were updated using gradient descent with a learning rate of 0.01. The learning rate controls how much to change the weights based on the calculated gradients.

Training Process:

The model was trained for 30 epochs. During each epoch:

- The forward pass was performed on the training data to compute the predictions.
- The loss was calculated using the binary cross-entropy function.
- The backward pass was used to update the weights based on the computed gradients.
- After each epoch, the model's accuracy on both the training and validation datasets was recorded.

Validation: To evaluate the model's performance during training, a validation set was used. The validation accuracy was calculated after each epoch to monitor how well the model was generalizing to unseen data. The results showed that the model's accuracy improved steadily over the training period.

Test Evaluation: After training, the model was tested on the test dataset to assess its final performance. The test accuracy was calculated by comparing the model's predictions to the actual labels. This provided a measure of how well the model could generalize to new, unseen data.

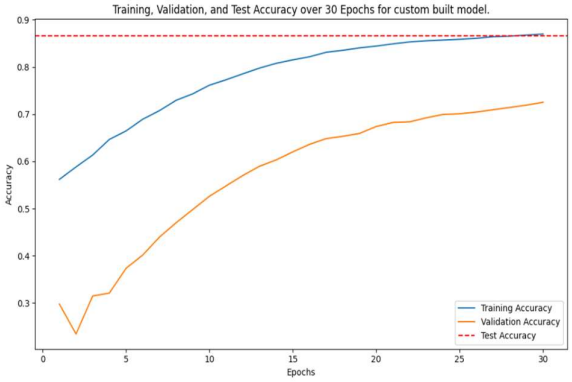


Fig 4.3 Accuracy plot for Custom built face mask detector.

The results, as depicted in the figure 4.3, show that the custom neural network was able to achieve reasonable accuracy on both the validation and test datasets. The model demonstrated a solid understanding of the basic patterns in the data, correctly classifying most images as either "With Mask" or "Without Mask." However, it is important to note that this exercise was primarily focused on gaining a deep

understanding of the training process rather than optimizing the model for maximum efficiency or performance. The insights gained from building and training this model from scratch will be invaluable in the subsequent steps of developing more sophisticated and efficient deep learning models.

4.3.2 Using High-Level Libraries of PyTorch

In this section, I explored PyTorch [11] by building, training, and evaluating ANN and a CNN. This was a brief, exploratory exercise to familiarize myself with PyTorch's capabilities and workflow.

The ANN was constructed with multiple fully connected layers, each followed by ReLU activation functions. The CNN incorporated convolutional layers followed by max-pooling layers, designed to capture spatial hierarchies in the input images. This model was expected to perform better on image data due to its ability to recognize patterns and features such as edges and textures.

Both models were trained on the prepared dataset using the Adam optimizer and binary cross-entropy loss function. The training process was straightforward, with each model undergoing 10 epochs of training. The CNN, with its more sophisticated architecture, quickly outperformed the ANN, achieving higher accuracy on both the training and test datasets.

The ANN achieved a test accuracy of approximately 87%, reflecting its ability to learn from the data but also highlighting its limitations in handling more complex image patterns.

The CNN, on the other hand, reached a test accuracy of about 96%, demonstrating its superior capability in image classification tasks due to its convolutional architecture.

In conclusion, this experiment provided valuable insights into PyTorch's functionality and reinforced the effectiveness of CNNs over ANNs for image-based tasks. While this was a preliminary exploration, the experience gained here lays the foundation for more advanced implementations using PyTorch in future work.

4.3.3 Building and Comparing TensorFlow Models with Custom-Built Models

In this exercise, I leveraged TensorFlow's [12] high-level APIs to construct and evaluate ANN and CNN. The aim was to compare the performance of these TensorFlow models with the custom-built models developed earlier, and to gain insights into how TensorFlow’s functionalities streamline the model-building process.

TensorFlow provides a range of high-level methods that significantly simplify the development of deep learning models:

- **Sequential Model:** This is a straightforward way to build a model layer by layer. The Sequential model allows for easy stacking of layers, making it ideal for both ANN and CNN architectures.
- **model.compile:** This method configures the model for training. It allows for specifying the optimizer, loss function, and metrics. For this experiment, I used the Adam optimizer and binary cross-entropy loss, which are well-suited for binary classification tasks like face mask detection.

- **model.fit:** This method is used for training the model. It efficiently handles the training loop, including the forward pass, loss calculation, backward pass, and weight updates. I used this method to train both the ANN and CNN models over 10 epochs, with real-time feedback on training and validation accuracy.
- **model.evaluate:** This method evaluates the model on a test dataset, providing metrics such as loss and accuracy. It allowed for a direct comparison of the models' performances on unseen data.

By utilizing these methods, I was able to develop and train both the ANN and CNN models efficiently. TensorFlow's abstraction layers enabled a smoother implementation process compared to manually coding each step, as done in the custom-

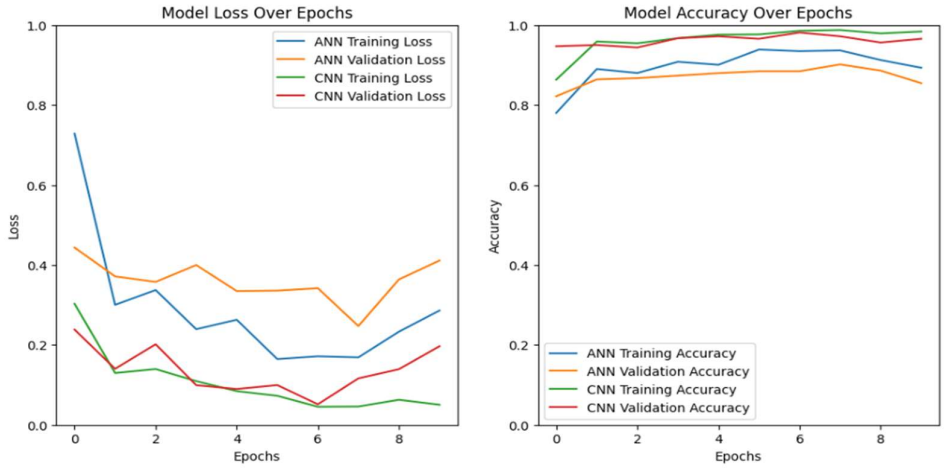


Fig 4.4 Loss and Accuracy plots for Face mask detectors built using CNN

built models. When comparing the performance of the TensorFlow-based models in figure 4.4 with the custom-built model in figure 4.3, several observations were made:

- The custom-built model exhibits a steady increase in both training and validation accuracy over the course of 30 epochs. However, there is a noticeable gap between training and validation accuracy, indicating that the model is learning but is not generalizing as well to unseen data. The final test accuracy remains constant, suggesting that the model may be overfitting to some extent as the training progresses.
- In contrast, the TensorFlow ANN and CNN models achieve a higher level of accuracy earlier in the training process (within the first 10 epochs). The CNN model reaches and maintains an accuracy close to 97%, which is significantly higher than the custom-built model's test accuracy of around 87%. The ANN, while less effective than the CNN, still performs comparably to the custom-built model, indicating the advantages of using more complex architectures like CNNs for image classification tasks.
- The loss plots for the TensorFlow models demonstrate a clear downward trend for both training and validation loss in the CNN model, indicating effective learning. The ANN model shows a less stable loss reduction, particularly in the validation set, which may indicate some difficulty in generalizing as effectively as the CNN.

The trained CNN model was further validated by predicting new, unseen images to test its generalization capability. In one instance, the model accurately identified an individual wearing a mask with a perfect confidence level of 1.0000, while in another, it correctly classified an image of a person without a mask with a confidence level of 0.9996.

Fig 4.5 Unseen data predictions

These results underscore the model's robustness and effectiveness in making confident and accurate predictions on new data, demonstrating its potential for real-world application in face mask detection.

The final evaluation of the CNN model was performed using the test dataset, and the results are summarized in the confusion matrix and the classification report generated by scikit-learn. The confusion matrix visually represents the performance of the classification model, showing the correct and incorrect predictions made by the model.

Confusion Matrix Insights:

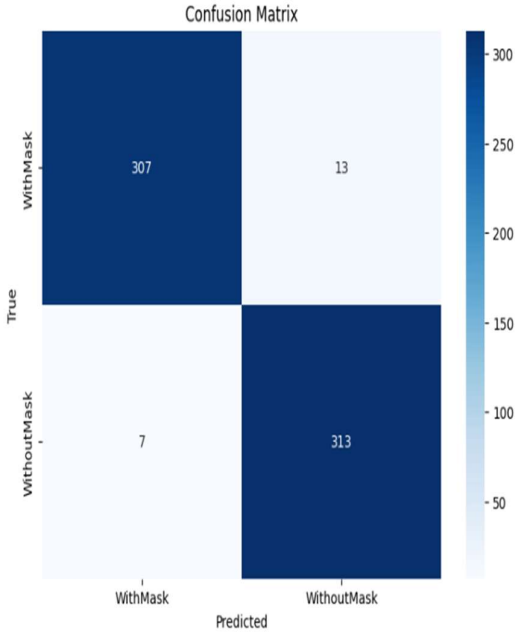


Fig 4.6 Confusion matrix for CNN Face mask Detector

- **True Positives (WithMask predicted as WithMask):** The model correctly identified 307 out of 320 images where individuals were wearing masks.
- **True Negatives (WithoutMask predicted as WithoutMask):** The model correctly identified 313 out of 320 images where individuals were not wearing masks.
- **False Positives (WithoutMask predicted as WithMask):** The model incorrectly classified 7 images where individuals were not wearing masks as wearing masks.
- **False Negatives (WithMask predicted as WithoutMask):** The model incorrectly classified 13 images where individuals were wearing masks as not wearing masks.

This confusion matrix in figure 4.6 indicates that the model has a high degree of accuracy in distinguishing between masked and unmasked individuals, with very few misclassifications.

4.4 Transfer Learning Approach

In this section, I explored the use of various pretrained models for the task of face mask detection using transfer learning [13] . The pretrained models chosen for this experiment were VGG19, ResNet101, InceptionV3, DenseNet121, and MobileNetV2. These models were selected based on their proven effectiveness in image classification tasks, making them well-suited for feature extraction and transfer learning applications. Additionally, these models are well-documented and highly regarded in the deep learning community, which made them ideal candidates for this experiment.

For each model, I initialized the base with the pretrained weights from the ImageNet dataset without updating these weights during training. This approach ensures that the models leverage the rich feature representations learned from large-scale image data. The models were then adapted to our specific task by adding custom fully connected layers on top, which were trained to classify images into two categories: "With Mask" and "Without Mask.". Below algorithm gives the clear picture of the entire approach.

```
Algorithm: Transfer_Learning_for_Face_Mask_Detection

- Import Required Libraries and Modules
- Import TensorFlow, Keras, Matplotlib, Seaborn, Sklearn, and
other necessary libraries.
- Define the pretrained models and hyperparameters such as
IMG_HEIGHT, IMG_WIDTH, BATCH_SIZE, EPOCHS, and MODELS.

- Data Preparation
   - Initialize an `ImageDataGenerator` with rescaling to
   preprocess images.
   - Load training, validation, and test datasets using
   `flow_from_directory` with target size (IMG_HEIGHT,
   IMG_WIDTH) and batch size (BATCH_SIZE).

- Model Creation
   For each model_name in MODELS.keys():
   - Load the base model with pretrained weights from ImageNet:
       - Set `include_top=False` to exclude the top
classification layer.
       - Set base model layers as non-trainable to freeze the
       pretrained weights.
   - Add custom layers on top of the base model:
       - Global Average Pooling layer.
       - Dense layers with ReLU activation.
       - Output Dense layer with sigmoid activation for binary
       classification.
   - Compile the model using:
       - Optimizer: Adam
       - Loss Function: Binary Cross-Entropy
       - Evaluation Metric: Accuracy

- Training
   For each model_name in MODELS.keys():
```

```
        - Train the model using `fit` on the training dataset:
            - Validate the model using the validation dataset.
            - Save the trained model and its training history.

    - Evaluation
       For each model_name in MODELS.keys():
       - Load the trained model.
       - Evaluate the model's performance on the test dataset:
           - Compute predictions and convert them into binary
labels.
       - Generate and save evaluation metrics:
           - Confusion Matrix
           - Classification Report
           - Save the test accuracy, loss, and other evaluation
          metrics for comparison.

    - Performance Metrics Calculation
       For each model_name in MODELS.keys():
       - Compute the following metrics:
           - Precision
           - Recall
           - F1-Score
           - ROC-AUC
       - Plot and compare metrics across different models using
bar charts and ROC curves.

    End Algorithm
```

This structured approach to transfer learning enabled me to efficiently experiment with different pretrained models and assess their effectiveness in face mask detection. The chosen hyperparameters and methods ensured that the models leveraged the strength of transfer learning while being optimized for the specific task at hand.

In implementing the transfer learning approach for face mask detection, several key hyperparameters were carefully selected to optimize model performance while maintaining computational efficiency. The image dimensions were set to 128x128 pixels, balancing the need for detailed feature extraction with the memory and processing limitations typical of deep learning tasks on standard hardware. The batch size of 32 was chosen to ensure that each update to the model's weights was based on a reasonably sized subset of data, enabling more stable convergence during training.

The choice of 5 epochs for training reflects a compromise between sufficient learning and preventing overfitting, particularly given that the models were initialized with pretrained weights. Importantly, the decision to freeze the base layers of these models ensured that the pre-learned features were preserved, while the newly added fully connected layers were trained to adapt specifically to the face mask detection task.

After implementing this algorithm, I plotted the test accuracies and confusion matrices of all the models and analysed which models performed better and why, as well as which models did not perform as well and the possible reasons behind their lower performance.

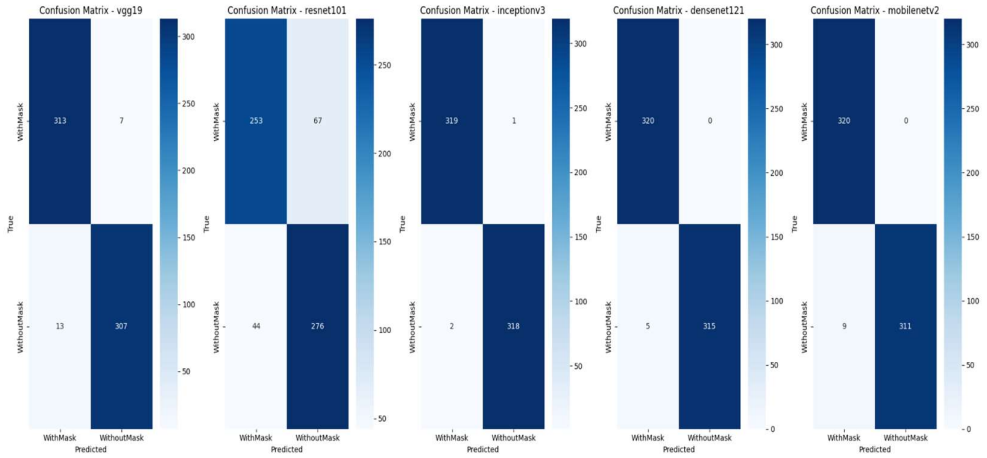


Fig 4.7 Confusion matrices for transfer learning approach applied to 5 models iteratively

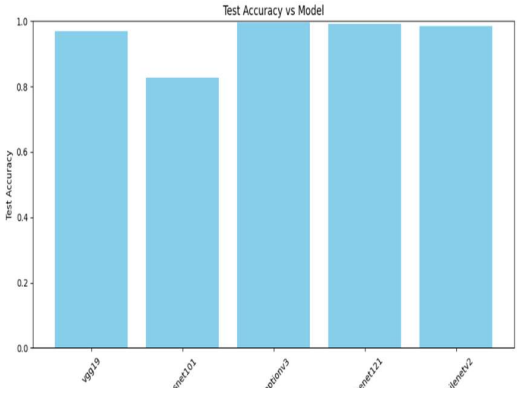


Fig 4.8 Test Accuracies of all the 5 models.

- InceptionV3, DenseNet121, and MobileNetV2 emerged as the top performers, with DenseNet121 achieving near-perfect classification and no errors for "With Mask." These models' robust architectures and efficient feature extraction capabilities, especially in capturing subtle details, contributed to their superior performance.
- VGG19 performed well but with slightly more misclassifications compared to the top models, likely due to its simpler architecture. ResNet101, the lowest performer, struggled with distinguishing between classes, which could be attributed to its deep architecture not being fully optimized during the training process.

In conclusion, InceptionV3, DenseNet121, and MobileNetV2 proved to be the most effective models for this task, likely due to their ability to extract complex features from images. ResNet101, while generally a strong model, did not perform as well in this context, potentially due to the complexity of its architecture not being fully exploited. These findings underscore the importance of selecting the appropriate model architecture for the specific nuances of the task at hand.

4.4.1 Predicting some unseen data using the trained and saved models.

In this section, I tested the robustness of the trained models by using them to predict unseen data from a new set of images. I loaded each of the saved models and applied them to a set of images not previously seen during training or testing. Each image was pre-processed to match the input requirements of the models, and predictions were made. The models were tasked with classifying the images as either "With Mask" or "Without Mask." The results were then visualized, highlighting how each model performed on these completely new images, providing a practical demonstration of their generalization capabilities beyond the initial datasets.

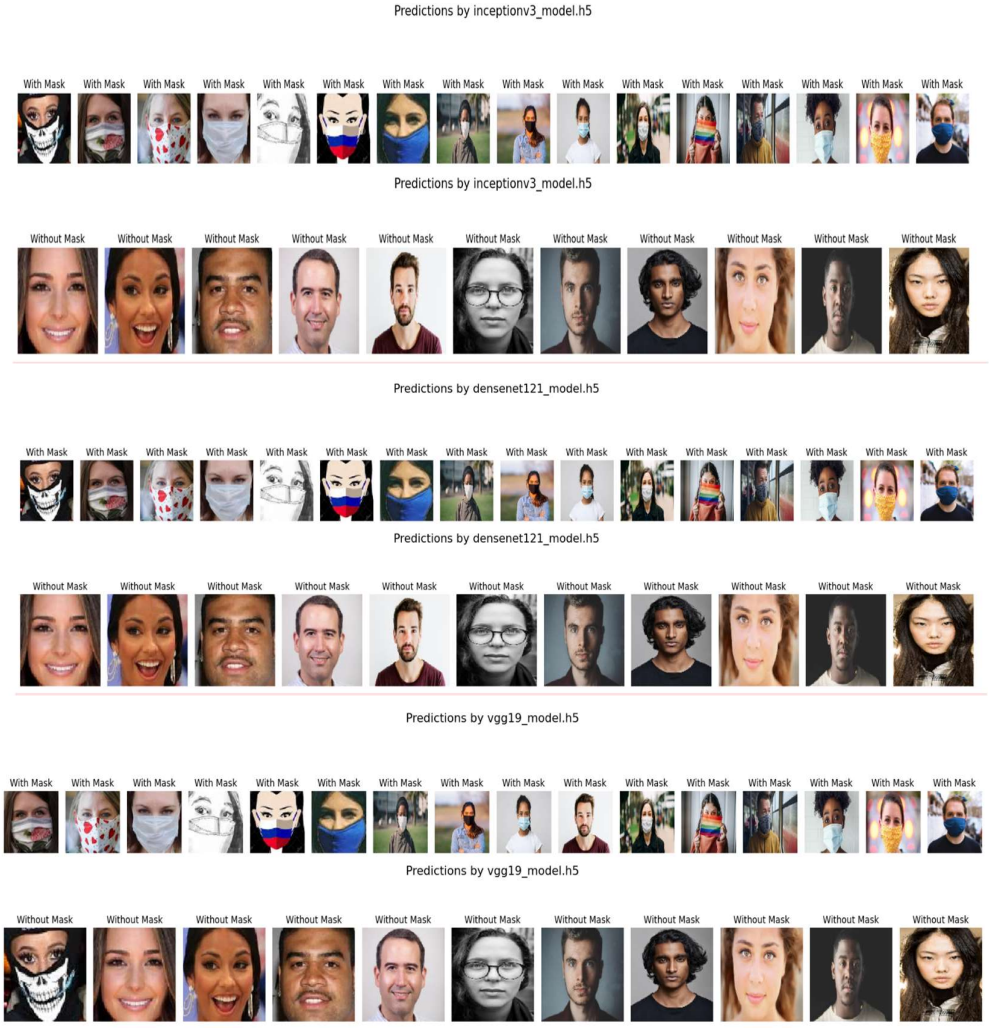


Fig 4.9 Predictions made by few models in Transfer learning.

Notably, InceptionV3 and DenseNet121 performed exceptionally well, accurately capturing all the minute details, and delivering precise classifications for these unseen images.

4.4.2 Evaluation Using Additional Performance Metrics

- **Precision score: It** is a metric used in binary classification to measure the accuracy of the positive predictions made by a model. Specifically, precision is defined as

the ratio of true positive predictions to the sum of true positive and false positive predictions. Mathematically, it is expressed as:

$$Precision = \frac{True\ Positives}{[True\ Positives + False\ Positives]}$$

Precision answers the question: *Of all the examples that were classified as positive, how many were positive?* This metric is particularly important in scenarios where the cost of false positives is high. For example, in face mask detection, a model with high precision would correctly identify most individuals wearing masks, with few instances of mistakenly identifying someone without a mask as wearing one.

The precision score plot reveals that InceptionV3, DenseNet121, and MobileNetV2 achieved perfect precision, meaning they made no false positive predictions and were highly effective in distinguishing between masked and unmasked individuals. VGG19 also performed well with a slightly lower precision, indicating minimal false positives. In contrast, ResNet101 had a lower precision score, suggesting it struggled more with accurately distinguishing between the classes, leading to more misclassifications. Models with higher precision are preferable in scenarios where minimizing false positives is crucial.

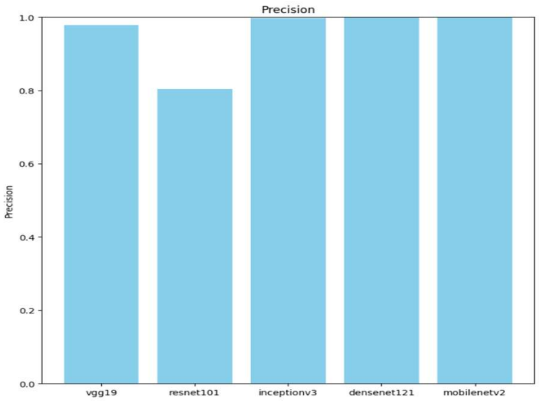


Fig 4.10 Precision Plot for transfer learning.

- **Recall** is a metric used in binary classification to measure the model's ability to correctly identify all relevant instances within a dataset. It is calculated as the ratio of true positives (correctly predicted positive instances) to the sum of true positives and false negatives (instances that were not identified as positive by the model but should have been). Mathematically, it is expressed as:

$$Recall = \frac{True\ Positives}{[True\ Positives + False\ Negatives]}$$

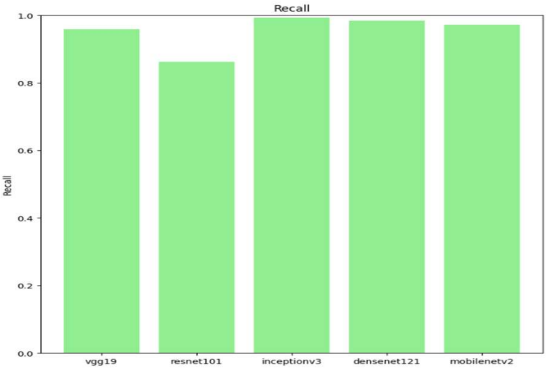


Fig 4.11 Recall Plot for transfer learning.

A high recall indicates that the model is effective at capturing the actual positive cases, making it particularly important in applications where missing a positive case is costly or dangerous. In the context of face mask detection, a high recall would mean that the model correctly identifies most, if not all, instances of people wearing or not wearing masks.

The recall score plot shows that InceptionV3, DenseNet121, and

MobileNetV2 achieved nearly perfect recall, while VGG19 also performed well but slightly lower. ResNet101 had the lowest recall, indicating it missed more positive cases, which suggests it was less effective at recognizing the target class compared to the other models.

- **The ROC (Receiver Operating Characteristic) curve** is a graphical representation used to evaluate the performance of a binary classifier. It plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at various threshold settings, giving insight into the trade-offs between sensitivity (recall) and specificity. The Area Under the Curve (AUC) quantifies the overall ability of the model to discriminate between positive and negative classes. An AUC of 1.0 indicates perfect classification, while an AUC of 0.5 suggests no discriminative power.

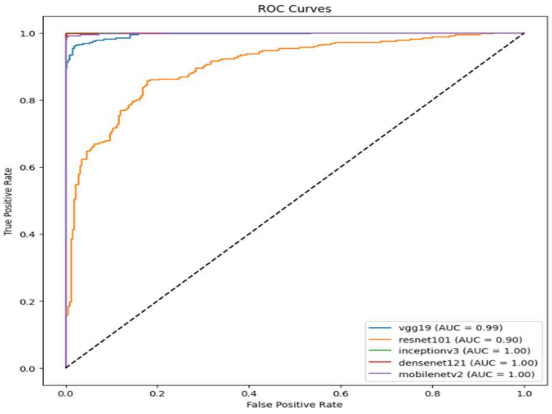


Fig 4.12 ROC curve Plot for transfer learning.

In the ROC curve plot provided, InceptionV3, DenseNet121, and MobileNetV2 achieved an AUC of 1.0, indicating near-perfect classification performance. VGG19 also performed well with an AUC of 0.99, suggesting it is highly effective, though slightly less so than the top performers. ResNet101 had a significantly lower AUC of 0.90, indicating that it was less capable of distinguishing between the classes compared to the other models. This suggests that ResNet101 struggled more with classification, potentially due to overfitting or its architectural characteristics not being as well-suited to this specific task.

- **The F1 score** is a metric that combines both precision and recall, providing a single measure of a model's accuracy in terms of its performance on a specific class. It is particularly useful when the class distribution is imbalanced. The F1 score is calculated as the harmonic mean of precision and recall, giving more insight into the balance between them. A higher F1 score indicates that the model has a good balance between precision and recall.

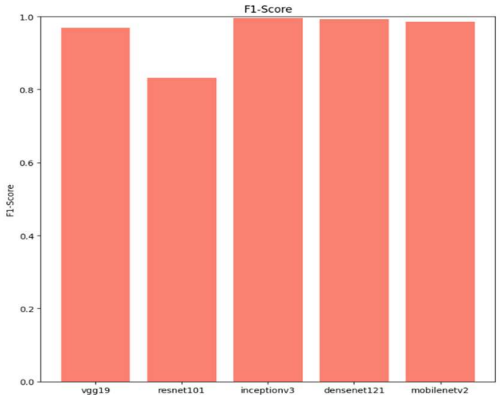


Fig 4.13 F1 score Plot for transfer learning.

In the provided plot, InceptionV3, DenseNet121, and MobileNetV2 achieved the highest F1 scores, indicating their exceptional performance in both precision and recall, leading to highly accurate predictions. VGG19 also performed well, but slightly below the top three models. ResNet101, on the other hand, showed a noticeably lower F1 score, reflecting its struggle to maintain a balance between precision and recall, thus leading to less reliable predictions compared to the other models.

4.5 Active Learning Approach

I explored the concept of Active Learning[14], [15], an extension suggested by my supervisor, which I decided to try out of curiosity. Active Learning is a machine learning approach where the model iteratively selects the most uncertain samples from a pool of unlabelled data, which are then labelled and added to the training set. This process aims to improve the model's performance by focusing on the most challenging cases, potentially reducing the amount of data needed for effective training.

```
Algorithm: Active_Learning_for_Face_Mask_Detection

   -Data Preparation: Load and preprocess training, validation, test,
   and unlabelled datasets.

   -Active Learning Iteration: Use the pretrained model to predict
   labels on the unlabelled data. Identify the most uncertain
   predictions (closest to 0.5).

   -Labelling and Augmentation: Move selected uncertain images to the
   appropriate class directories based on predicted labels.

   -Model Retraining: Retrain the model with the newly labelled data.
   Save the updated model and repeat the process.

   -Final Evaluation: Evaluate and save the final model and its
   performance metrics.

End Algorithm
```

The implementation involved using the ResNet101 model, initially trained in previous sections. The process began by predicting the labels for a set of unlabelled images. The model identified the most uncertain predictions—those closest to the decision boundary (0.5 probability)—and these uncertain images were selected for labelling. The labelled images were then moved to the appropriate training directories based on their predicted class.

After relabelling, the model was retrained with the updated training set, further refining its performance. This iterative process of selecting uncertain samples, labelling them, and retraining the model was repeated multiple times, each time saving the improved model and its training history. By focusing on the most challenging images, the active learning process helped to enhance the model's ability to generalize and perform well on unseen data.

4.6 Conclusion and Future Work

In this project, I aimed to develop a face mask detection system using various deep learning techniques and methodologies. The work encompassed exploring different models, including custom-built neural networks, pretrained models through transfer

learning, and active learning strategies. The primary objective was to build a functional face mask detection model while gaining hands-on experience with different deep learning approaches. Although there are still many areas where the system could be improved, the project served as a valuable learning experience within the constraints of the time and prerequisites available.

For future work, this project could be expanded into a more complex classification problem. Instead of just detecting the presence or absence of a mask, the model could be trained to recognize whether a mask is being worn correctly, such as ensuring that both the nose and mouth are properly covered. Additionally, incorporating more diverse datasets and refining the model to handle more nuanced cases could significantly enhance its accuracy and practical utility. This extension would not only improve the robustness of the system but also align it more closely with real-world requirements.

5 Self-Assessment and Reflection

5.1 Project Overview

The overall experience of this project has been both highly informative and challenging. Throughout the journey, I delved deeper into the concepts of deep learning, which not only broadened my understanding but also allowed me to explore new ideas and techniques that I had previously only encountered in a theoretical context. This project offered the opportunity to resolve many of the doubts and curiosities that were not fully addressed during the academic coursework. By applying these concepts practically, I was able to gain a more comprehensive understanding of machine learning and deep learning, particularly in the context of solving real-world problems. This hands-on approach provided invaluable insights into how these technologies work in practice, beyond the confines of textbook examples.

5.2 Appraisal of Efforts

Before reflecting on my own efforts, I want to acknowledge the significant guidance and support from my supervisor, which was incredibly helpful and crucial for the success of the project. As for my efforts, one of the most rewarding parts was gaining a full understanding of the entire process involved in building and solving a problem using machine learning. This includes recognizing and addressing the challenges that come up during implementation, as well as learning how to properly evaluate the performance of models. Effective time management and research were also key outcomes of this project, as I had to juggle various tasks and explore new information efficiently.

However, there were some challenges. One major difficulty was the gap between the theoretical knowledge from classes and the practical results obtained during experiments. A lot of time was spent diagnosing issues and figuring out how to correct unexpected results. Additionally, the topic was very broad and general, making it hard to explore everything in depth due to time and resource constraints. Despite these challenges, the overall experience was very rewarding and contributed significantly to my growth as a researcher and practitioner.

5.3 Lessons Learned

Looking back on the challenges faced and the successes achieved, several important lessons stand out. First, I learned how important it is to combine practical experience with theoretical knowledge. Real-world applications often present challenges that are not covered in academic settings, and dealing with these challenges has deepened my understanding of machine learning. Second, this experience highlighted the need to be flexible and adaptable when working on complex projects. When things did not go as planned, it was important to approach problems with an open mind and a willingness to explore different solutions. Finally, effective time management and planning were crucial for balancing the many tasks involved in a project of this size and ensuring steady progress. These lessons will be valuable in future projects, where I plan to apply them to achieve even better results.

5.4 Future Directions

This project has provided a solid foundation in machine learning and deep learning, and it will be incredibly valuable for my future career. If I decide to pursue a path in these fields, the knowledge and skills gained from this project will be directly applicable to real-world challenges. The hands-on experience I have gained from building and training models to troubleshooting and optimizing them will be crucial in navigating the complexities of machine learning projects in a professional setting.

Furthermore, the ability to think critically about data, algorithms, and model performance will help me contribute effectively to any team working on machine learning or deep learning projects. The insights and practical skills I have developed will not only enhance my technical capabilities but also give me the confidence to tackle new and challenging problems. This project has shown me the potential of deep learning in solving complex issues, and I am excited about the possibilities it opens for my career. Whether I continue to specialize in machine learning or explore related fields, the experience from this project will be a cornerstone of my professional growth and success.

6 How to run my project.

To successfully run my project, you will need to follow a few key steps involving setting up the environment, downloading the necessary datasets, and ensuring the correct paths are set within the provided Jupiter notebooks (IPYNB files). Below is a detailed guide on how to get everything up and running.

All the files have been uploaded in OneDrive and give public access to everyone inside Royal Holloway network:

OneDrive link (using Royal Holloway Credentials): https://rhul-my.sharepoint.com/:f:/g/personal/mmac259_live_rhul_ac_uk/Eryj1lj0pD1GjicDk7ltBJcB5YMlWwyNQ9nrlsOqkuOplQ?e=yPd5qb

6.1 Required Files and Setup

- **IPYNB Notebooks:**
  There are two IPYNB notebook files that need to be run:

- *Early Deliverables Work*: This notebook with filename "Early_deliverables.ipynb" covers the initial experiments and implementations.
- *Final Deliverables*: This notebook with filename "FaceMaskDetector_FinalDeliverables.ipynb" includes the comprehensive experiments, model training, and evaluations, including the transfer learning and active learning approaches on the facemask dataset.

- **Datasets:**
  - **Unrefined Dataset:** This is the larger, unrefined dataset used in early experiments.
  - **Refined Dataset:** This is a smaller, more curated dataset used for final model training and evaluation.

6.2 Environment Setup

- **Install Required Libraries:**
  - You will need to install several Python libraries to run the notebooks. Use the following sample commands to install the necessary packages:

```
pip install tensorflow keras torch torchvision matplotlib seaborn scikit-learn
```

  - After installing these libraries, it is recommended to restart the Jupyter kernel to ensure all dependencies are loaded correctly.
- **Set Dataset Paths:**
  - Once the datasets are downloaded, you must set the appropriate file paths in the notebooks. Ensure that the train, test, and validation folders are correctly linked to their respective locations in the dataset. This is critical for the successful execution of the data loading and training processes.

6.3 Running the Notebooks

- **Early Deliverables Work:**
  - Open the *Early Deliverables* IPYNB notebook in Jupyter and ensure that all dataset paths are correctly set.
  - Run the cells sequentially, ensuring that all libraries are correctly installed and the data is loaded properly.
- **Final Deliverables:**
  - Open the *Final Deliverables* IPYNB notebook.
  - Like the early deliverables, ensure all paths are correctly set.
  - Run the notebook from start to finish, which will include model training, evaluations, and generating various plots.

6.4 Active Learning Special Instructions

For the active learning portion, special modifications are required:

- **Initial Setup:**
  - You need to have a folder named Unlabelled where all the unlabelled data resides. This folder should contain images that the model has not yet seen.

    - The Train folder should contain subfolders withMask and withoutMask, which are the directories where the actively learned images will be moved.
- **Active Learning Iterations:**
    - As you run the active learning code, the model will select the most uncertain samples from the Unlabelled folder. These images will then be moved to the appropriate subfolders in the Train directory.
    - You must manually update the paths in the notebook to reflect the current state after each iteration if you choose to re-run or modify the steps.

References

[1] Zhang, Aston and Lipton, Zachary C. and Li, Mu and Smola, Alexander J., *Dive into Deep Learning*. (v5 ed.) Cambridge University Press, 2024.

[2] Jeremy Howard,   and Sylvain Gugger, *Deep Learning for Coders with Fastai and PyTorch*. "O'Reilly Media, Inc.", 2020.

[3] G. Huang *et al*, "Densely connected convolutional networks," in 2018-01-28, .

[4] Li Zhang, "Optimization Algorithms Slides," 02/02/, 2024.

[5] &. Zhang and Li, "Overfitting and regularization techniques. ," 02/02/, 2024.

[6] &. Wikipedia contributors, T. F. E. Wikipedia and ., "Inceptionv3," 2024. Available: https://en.wikipedia.org/w/index.php?title=Inceptionv3&oldid=1242305785.

[7] Keras 3 API documentation, "MobileNet, MobileNetV2, and MobileNetV3," Available: https://keras.io/api/applications/mobilenet/.

[8] (). *CIFAR-10 Dataset*. Available: https://www.cs.toronto.edu/~kriz/cifar.html.

[9] (04/07/). *Weight Initialization Techniques for Deep Neural Networks*. Available: https://www.geeksforgeeks.org/weight-initialization-techniques-for-deep-neural-networks/.

[10] Ashish Jangra, "Face Mask Detection ~12K Images Dataset," .

[11] &. Paszke *et al*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[12] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, *et al*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015.

[13] (12-02-). *Transfer Learning*. Available: https://www.ibm.com/topics/transfer-learning#:~:text=Transfer%20learning%20is%20a%20machine,improve%20generalization%20in%20another%20setting.

[14] C. J. Brame, "Active-Learning-article," .

[15] R. M. Felder and R. Brent, "ACTIVE LEARNING: AN INTRODUCTION *," .

[16] &. Zhang and Li, "Tranfer Learning. ," 02/02/, 2024.