

**NANYANG  
TECHNOLOGICAL  
UNIVERSITY**  
**SINGAPORE**

**SCSE23-0504**

# **Exploring Lightweight Deep Learning Techniques for Efficient Deepfake Detection**

Chua Gim Aik (Cai JinYi)

U2022142B

Project Supervisor: Assoc. Prof. Deepu Rajan

Examiner: Dr Tay Kian Boon

**School of Computer Science and Engineering**

Academic Year 2023/2024

**Nanyang Technological University**

**SCSE23-0504**

**Exploring Lightweight Deep Learning Techniques for Efficient  
Deepfake Detection**

Submitted in Partial Fulfilment of the Requirements for the Degree of  
Bachelor of Engineering in Computer Science of the Nanyang  
Technological University

By

CHUA GIM AIK

CAI JIN YI

School of Computer Science and Engineering

Academic Year 2023/2024

# Abstract

Deepfakes refer to a form of synthetic media where artificial intelligence and deep learning techniques are used to create or manipulate audio and video content to depict individuals saying or doing things they never actually said or did. Deepfake detection is the ability to differentiate between deepfakes and real media. In this project, we undertake the task of proposing strategies to achieve high deepfake detection accuracy, while attempting to reduce model complexity, size, and inference time, with the goal of pushing the deployment of complex deep-network based solutions such as EfficientNet closer towards hardware constrained environments such as edge devices. In this project, we propose strategies to tackle the deepfake video detection problem and learning strategies that achieved high model accuracy over the FaceForensics++ dataset using EfficientNet as our primary model. We also discuss several model reduction techniques such as architectural changes, model pruning and knowledge distillation to cut our model footprint by half while preserving much of our model's original performance. Finally, we demonstrate how our proposed model can be turned into a deepfake detection application at the video level for solution completeness.

# Acknowledgements

I would like to express my sincerest gratitude to Associate Professor Deepu Rajan for his encouraging support, guidance, and patience throughout the duration of this project. I remember stepping into his office on the first meeting feeling extremely lost as this was a difficult FYP that I had no prior experience in. Despite my lack of knowledge in this research area and Professor Deepu's busy schedule, he finds time to point me in the right direction. I am extremely grateful to be Prof. Deepu's FYP student.

I would like to also extend my thanks to Prof Deepu's Ph.D. student, Jonathan Weston Burton-Barr for his constant encouragement and feedback. Despite his busy schedule, he enjoys sharing knowledge with me and motivating me to do better for my project. I really could not have completed the project without him either.

# Table of Contents

Abstract .....	3
Acknowledgements .....	4
Table of Contents .....	5
Chapter 1 Introduction .....	9
1.1 Background & Motivations .....	9
1.2 Project Objectives .....	10
1.3 Project Scope .....	10
1.4 Contributions .....	11
Chapter 2 Literature Review .....	12
2.1 Deepfake Generation .....	12
2.2 Deepfake Detection .....	14
2.3 Lightweight Deep Learning .....	16
2.3.1 Model Pruning .....	16
2.3.2 Knowledge Distillation .....	18
2.3.3 Transfer Learning .....	18
2.3.4 Model Quantization .....	18
2.4 EfficientNet .....	19
2.4.1 Model Architecture .....	19
2.4.2 MBConv Blocks .....	19
2.4.3 Squeeze & Excitation blocks .....	20
2.4.4 Stochastic Depth .....	20
2.4.5 Model Scaling .....	20
2.4.6 EfficientNet-Lite .....	21
Chapter 3 Introducing the Dataset .....	23
3.1 FaceForensics++ .....	23

3.1.1	Youtube (“Real”) .....	24
3.1.2	Deepfakes.....	24
3.1.3	Face2Face.....	25
3.1.4	FaceShifter .....	25
3.1.5	FaceSwap .....	26
3.1.6	NeuralTextures.....	26
Chapter 4 Computation Environment .....		27
Chapter 5 Detailed Methodology & Implementation .....		28
5.1	Problem Formulation .....	28
5.1.1	Model Predictions .....	29
5.1.2	Binary Cross-Entropy Loss.....	29
5.2	Data Pre-processing .....	29
5.2.1	Frame Extraction & Sampling.....	29
5.2.2	Face Detection & Cropping .....	30
5.2.3	Data Sampling, Loading and Transformation.....	32
5.3	Model Implementation .....	36
5.3.1	Creating EfficientNet from scratch .....	36
5.3.2	Pytorch’s official implementation.....	37
5.3.3	EfficientNet-Lite .....	37
5.3.4	Modifying the classifier and enabling gradients .....	37
5.4	Model Learning Strategies .....	38
5.4.1	Model Hyperparameters.....	38
5.4.2	Learning Optimizer: Stochastic Gradient Descent .....	39
5.4.3	Learning Scheduler: Cosine Annealing.....	39
5.4.4	Linear Warm-up .....	40
5.4.5	Weight Initialization & Transfer Learning .....	41
5.4.6	Early Stopping .....	41
5.4.7	Fine-tuning .....	42
5.5	Model Scaling.....	42
5.5.1	Model Scaling Experiment .....	43
5.6	Architectural Changes – EfficientNet Lite .....	43
5.6.1	Removing Squeeze & Excitation Layers .....	43

5.6.2 Replacing Swish Activation with Relu6 .....	44
5.7 Model Pruning.....	45
5.7.1 Taylor’s Importance Pruning.....	45
5.7.2 Structured Pruning.....	46
5.7.3 Local Pruning VS Global Pruning .....	47
5.7.4 One-shot Pruning VS Iterative Pruning .....	48
5.8 Knowledge Distillation .....	48
5.8.1 Distillation Loss Formulation.....	49
5.8.2 Augmentation to Training Logic.....	50
5.8.3 Teacher Model .....	51
5.8.4 Knowledge Distilled Training and Pruning .....	51
Chapter 6 Experiments Results and Findings .....	52
6.1 Evaluation Metrics .....	52
6.1.1 Model Accuracy.....	52
6.1.2 Model Size .....	53
6.1.3 Inference Speed .....	54
6.2 Experiments .....	54
6.2.1 Transfer Learning using ImageNet-1K.....	54
6.2.2 EfficientNet Model Scaling Effectiveness.....	57
6.2.3 EfficientNet-Lite .....	58
6.2.4 Model Pruning.....	62
6.2.5 Knowledge Distillation .....	66
6.2.6 Findings .....	72
Chapter 7 Web Demo .....	74
Chapter 8 Conclusion & Future Works .....	76
8.1 Conclusion.....	76
8.2 Future Works .....	77
8.2.1 Quantization.....	77
8.2.2 Expanding Dataset .....	77
8.2.3 EfficientNet V2 Findings.....	78
8.2.4 Deployment on Edge Devices .....	78

Bibliography .....	79
--------------------	----



# Chapter 1

## Introduction

### 1.1 Background & Motivations

Deepfake is a generative AI technology capable of producing synthetic digital media that uses deep learning techniques to replace or superimpose a person's likeness with fabricated features that appears highly realistic that can take the form of images or videos.

In recent years, deepfake has emerged and proliferated across online media platforms and the technology has become increasingly popular and accessible to the public [1]. While deepfake technology can have practical and productive applications, such as enabling users to experiment with different hairstyles or outfits through mobile apps, it is currently more commonly and notoriously being used to disseminate deceptive content. Deepfake technology enables malevolent actors to engage in a spectrum of malicious activities that include, but are not confined to: disseminating disinformation and counterfeit news to subvert public sentiment, orchestrating acts of identity theft for the purpose of perpetrating fraudulent schemes, orchestrating character defamation by generating content that appropriates an individual's likeness without their consent, and tampering with the veracity of facts and evidentiary material.

The recent reveal of SORA [6], OpenAI's new cutting-edge text-to-video model further accentuates the global concern that synthetic content will become mainstream. Hence, there is an urgent need to lessen the harm that can be brought by malicious use of deepfake generation tools. The race to detect deepfakes spawned the research and development of many deepfake detection

technology, which is capable of recognizing and differentiating deepfake generated content from authentic content.

Several existing state-of-the-art deep learning based deepfake detection model such as the ones found in [2] and [5] achieves high detection accuracy against deepfake detection benchmarks but their effectiveness comes at the cost of the model being very large and impractical for deployment in constraint environments like edge devices (e.g. smartphones) where users are exposed to artificially generated content through internet and social media applications. While we can technically deploy these large models on cloud to handle the processing overhead, there is the concern of user privacy since the data would have to leave the users' devices to support such an infrastructure. Hence, to simultaneously protect users from the dangers of deepfakes while preserving their privacy, there is a pressing need to mount deep fake detection models onto constrained computing environments. To this end, we need to achieve lighter deepfake detection models.

## **1.2 Project Objectives**

Our project seeks to accomplish three main things: (1) formulate an effective strategy to tackle the video deepfake detection problem, (2) achieve good detection accuracy using common computational resources available to us and (3) explore and apply the lightweight deep learning techniques to reduce model footprint while preserving model performance.

## **1.3 Project Scope**

In this project, our focus is on applying lightweight deep learning techniques towards the domain of deepfake video detection to find ways to achieve good deepfake detection accuracy using

smaller and lighter models. We also mainly study the EfficientNet architecture and its suitability for deepfake detection. Finally, we execute the project with common computational resources to improve the accessibility and reproducibility of our proposed solutions.

## 1.4 Contributions

Our contributions can be summarized as follows:

- **Video Deepfake detection strategy:** in this project, we offer insights and propose methods to tackle the deepfake detection problem
- **Training strategies:** we propose training strategies to obtain timely and effective deepfake detection results under a low-cost hardware environment.
- **Model reduction strategies:** we provide insights into the lightweight deep learning techniques that help make our models smaller and faster while preserving model performance

# Chapter 2

## Literature Review

### 2.1 Deepfake Generation

Currently, [8] Deepfake Generation favors the use of Generative Adversarial Networks (GANs), where two neural network models, appropriately coined *generator*, and *discriminator*, compete against each other in a zero-sum game. In this framework, the generator aims to create new data (e.g. deepfakes) in attempt to fool the discriminator to think that the generated content is real. The discriminator, on the other hand, attempts to “guess” if the provided input is real or generated. In this two-player game, both models update their weights when they “lose” the game (i.e. fail to fool discriminator or fail to correctly distinguish real from fake). In the ideal scenario, with sufficient and proper training, the generator becomes so proficient at generating realistic content that the discriminator’s best strategy for discerning real and generated content is a random coin-flip.

[8] Another common technique is the use of Autoencoders which consist of an encoder and a decoder, that work together to compress input data into a lower-dimensional representation (encoding) and then reconstruct the original input from this representation (decoding). By compressing facial features to lower dimensions, the model can store the latent space representations of the face which can later aid in reconstructing the original face at a desired target location.

[8, 22] propose that there are 5 common deepfake manipulation types: (1) **Attribute Manipulation**, (2) **Entire Face Synthesis**, (3) **Lip-syncing**, (4) **Face Reenactment** and (5) **Identity Swapping**.

- In **attribute manipulation**, specific characteristics or attributes of an individual's face are altered or modified. This can include changing features such as hairstyle, hair color, age, gender, facial expressions, and even adding or removing accessories like glasses or hats.
- **Face synthesis** involves generating a completely new face that does not belong to any real individual.
- **Lip-syncing deepfakes** involve manipulating the mouth movements and speech of an individual in a video so that it appears as though they are saying something different from what was originally spoken.
- **Face reenactment** deepfakes involve transferring the expressions of a source person to the target person in a video.
- **Identity swapping** deepfakes involve replacing the face of one individual in a video with the face of another person.

In **FaceForensic++** [43], the authors curated a popular and powerful deepfake benchmark dataset that applied various deepfake generation techniques to generate synthetic videos from YouTube videos which we briefly discuss:

- **DeepFakes** – not to be confused with deepfakes (which is the general term for “face replacement based on deep learning”) – is a face swapping technique that attempts to map a face in the target sequence with one that has been observed in the source video. This

technology employs two autoencoders that use a shared encoder architecture to reconstruct both the source and target face.

- **Face2Face** is a real-time face tracking and reenactment system that can manipulate facial expressions of the target actor in a video based on the facial expressions of an actor in the source video.
- **FaceSwap** is a graphics-based approach to transfer the face region from a source video to a target video that makes use of extracted facial landmarks from the source video to fit “blendshapes” on a 3D template model. The model is then back-projected to the target image before finally being blended with the image.
- **NeuralTextures** learns and replaces the original facial textures in a target video with synthetic ones while preserving the overall facial structure and appearance. In FF++, the authors only modified the facial expressions corresponding to the mouth region.
- **FaceShifter** is a deep learning-based face swapping technique that utilizes a combination of adversarial training and identity preserving loss functions to perform high fidelity and occlusion aware face swaps. This was not originally included in the paper [43] but the authors later added it to the FF++ dataset.

## 2.2 Deepfake Detection

As proposed by [8], there currently exists several deepfake detection clues that could give deepfakes away:

- Deepfake manipulation often introduces **spatial artifacts** or inconsistencies in the generated images or videos that can be detected through careful analysis. These artifacts

may manifest as unnatural distortions, blurriness, or inconsistencies in facial features, textures, or lighting.

- **Convolutional traces** are patterns left behind by the convolutional operations of deep learning networks during the generation of deepfake content. These traces may manifest as repeating patterns, grid-like structures, or other artifacts that are characteristic of the neural network architecture used to create the deepfake.
- Deepfake manipulation often struggles to accurately replicate **subtle facial emotions** and expressions, leading to inconsistencies between the emotional content of the manipulated face and the corresponding audio or context.
- Deepfake manipulation may struggle to maintain **temporal coherence** between different sequential elements of a video, leading to noticeable discontinuities.

In [7], the authors discuss the effectiveness of employing deep learning as a counter-strategy to detect deepfakes. A common approach is to use CNNs (Convolutional Neural Networks) for deepfake detection [30]:

- CNNs can analyze visual data efficiently, making them well-suited for tasks like identifying manipulated images and videos
- CNNs excel in recognizing patterns and features within images, which is crucial for detecting inconsistencies or alterations present in deepfake media.
- CNNs are robust against adversarial attacks, which are attempts to deceive the network by introducing subtle perturbations into the data.
- CNNs can be adapted to handle various forms of deepfake media (such as images or videos)

## 2.3 Lightweight Deep Learning

Our work involves applying lightweight deep learning techniques to make our models more compact without sacrificing too much performance. To this end, we peruse surveys conducted on efficient deep learning such as [40, 42] and consolidated some of the most common practices to achieve model reduction using light weight deep learning techniques.

### 2.3.1 Model Pruning

Model pruning is a technique used in deep learning to enhance model efficiency by removing unnecessary parameters, thereby reducing the model's size and computational requirements. This process involves eliminating specific weights or connections within a neural network that are deemed less important or redundant.

#### *2.3.1.1 Unstructured Pruning*

In unstructured pruning, individual weight connections are removed, typically by setting them to zero. This method introduces sparsity into the network, making it more efficient in terms of execution speed and storage space. Unstructured pruning allows for faster inference on resource-constrained devices.

#### *2.3.1.2 Structured Pruning*

Structured pruning involves removing groups of weight connections together, such as entire channels or filters. Unlike unstructured pruning, structured pruning maintains the overall structure of the network but reduces the size of specific components.



### ***2.3.1.3 Local Pruning***

Local pruning involves removing a fixed percentage of units or connections from each layer by comparing within the layer itself. This technique targets specific substructures within the model, allowing for more granular control over the pruning process.

### ***2.3.1.4 Global Pruning***

Global pruning involves removing connections or units across the entire neural network, allowing for the removal of less important or redundant parameters on a global scale. Global pruning can lead to significant model compression and efficiency improvements by eliminating unnecessary weights or connections throughout the entire network.

### ***2.3.1.5 Iterative Pruning***

Iterative pruning is a technique used in deep learning to enhance model efficiency by iteratively removing less important parameters or connections from a neural network. This process typically involves multiple rounds of pruning and retraining, where the network is pruned to remove redundant or less significant weights, followed by fine-tuning to recover any lost accuracy. This contrasts with *one-shot pruning*, where model weights are pruned and fine-tuned exactly once.

### ***2.3.1.4 Importance Pruning***

[15]: Importance pruning identifies and removes less important parameters or connections based on their significance to the model's performance. This method assigns importance scores to parameters, allowing for the removal of those deemed less critical without significantly impacting the model's accuracy. By focusing on retaining the most influential weights and connections while discarding the less relevant ones, importance pruning helps streamline neural networks, making them more efficient and compact. This approach plays a crucial role in optimizing model efficiency

and resource utilization by eliminating redundant parameters that do not contribute significantly to the network's predictive capabilities

### **2.3.2 Knowledge Distillation**

Knowledge distillation is a technique used to compress a complex and large neural network into a smaller and simpler one while maintaining its performance and accuracy. This process involves training a smaller neural network, known as the student network, to mimic the behavior of a larger and more complex "teacher" network.

### **2.3.3 Transfer Learning**

Transfer learning is a machine learning technique where knowledge gained from training on one task is repurposed to improve performance on a related task. This method allows for leveraging pre-existing knowledge from one domain to enhance learning efficiency in another.

### **2.3.4 Model Quantization**

Quantization is the process of converting parameters from higher precision representations like 32-bit floats to smaller representations like 8-bit integers. This significantly reduces the model size which leads to improved memory efficiency. This technique also enhances network speed by allowing operations to be performed using integer data types, which require fewer computations on processor cores. However, the benefits of quantization in terms of memory usage and power efficiency often comes with an accuracy trade-off due to the reduced precision representation of the model parameters.

## 2.4 EfficientNet

Our project will focus on the use of EfficientNet as described in [3]. EfficientNet is family of networks designed to strike an optimal balance between computational efficiency and model performance, making it an ideal choice of study for our lightweight deep fake detection problem.

### 2.4.1 Model Architecture

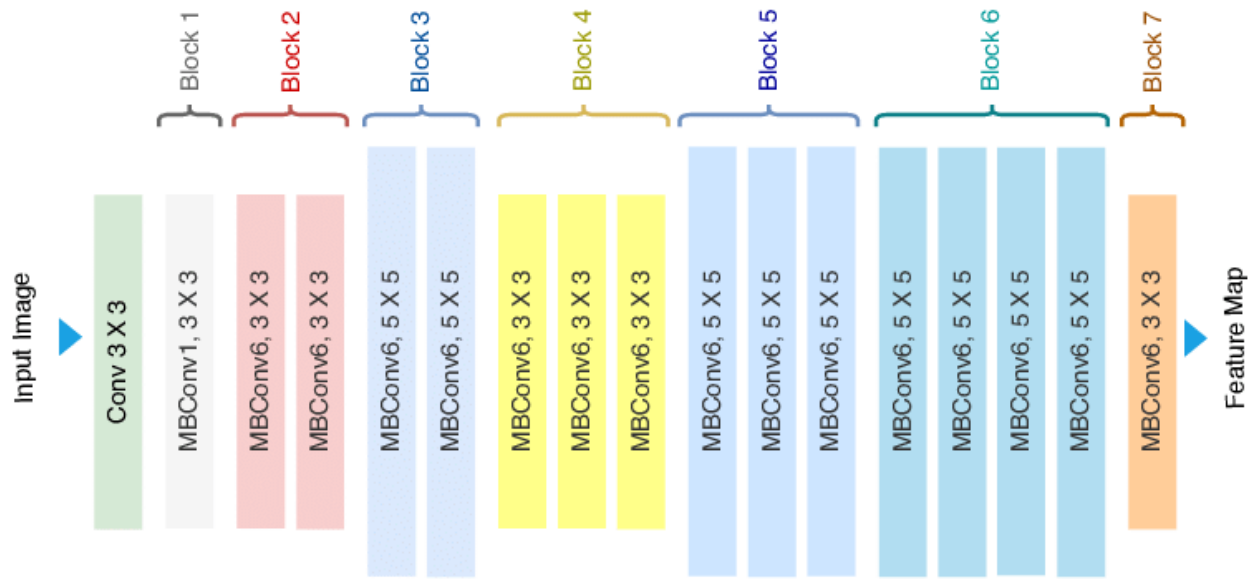


Figure 2.3.1.1 EfficientNet-B0's baseline network architecture

### 2.4.2 MBConv Blocks

EfficientNet is primarily made up of Mobile Inverted Bottleneck Convolutions blocks. MBConv Blocks or Inverted Residual Blocks follow a narrow-wide-narrow structure by first widening the input channels with pointwise convolutions, reducing the number of parameters with depthwise convolution (3x3) before reducing the channels again using pointwise convolutions. The design of MBConv blocks optimizes memory usage and computational efficiency as it effectively shrinks the number of passing parameters.

### **2.4.3 Squeeze & Excitation blocks**

In [3], the MBConv blocks are implemented with a Squeeze and Excitation (SE) block. In SE blocks, there are two phases: (1) the squeeze phase and (2) the excitation phase. During the squeeze phase, global average pooling is applied to the output tensor of the MBConv block along the channel dimension. This computes statistics to create logit representation of each channel, these statistics are fed into a fully-connected layer using the Swish Activation function during the excitation phase to learn the importance of each channel by assigning them adaptive weights to adjust the influence of each channel based on the learned importance during the excitation phase.

### **2.4.4 Stochastic Depth**

In [3], the authors also implemented stochastic depth as proposed by [20], which is a layer regularization technique, this causes layers to drop randomly during training with each layer having a survival probability of 0.8. This allows for the layer skipping to occur which can help with gradient flow due to shorter flows through the architecture, and can help the network achieve better generalization by encouraging the network to learn robust representations instead of depending on specific layers in the network.

### **2.4.5 Model Scaling**

The biggest contribution of [3], is the demonstration of model scaling. In the paper, the authors proposed a novel method to scale a network's size (width, depth and image resolution) uniformly using a compound coefficient ( $\Phi$ ). They found that scaling the model's size methodically in this manner achieves better result than traditional scaling which often arbitrarily chooses 1 factor to scale. The authors in [3] obtained the base version of EfficientNet (B0) by performing Neural

Architecture Search before scaling it up using the proposed model scaling method to larger EfficientNet versions.

Version	Phi value	Resolution	Drop rate
B0	0	224	0.2
B1	0.5	240	0.2
B2	1	260	0.3
B3	2	300	0.3
B4	3	380	0.4
B5	4	456	0.4
B6	5	528	0.5
B7	6	600	0.5

Table 2.3.5.1 – Model scaling table

The table shows the settings used for each EfficientNet version. Note that this is not the official version as [3] never made clear what the actual values are. Nevertheless, we follow [19]’s implementation for the scaling table. The compound coefficient phi value here is taken as the power to the depths (alpha) factors and width (beta) factor which are constants that [3] found to be optimized at 1.2 and 1.1 respectively. These factors are then multiplied with the layer repeats and number of channels directly to achieve the scaling effect. Our EfficientNet implementations will be using the values from this table and scaling logic.

## 2.4.6 EfficientNet-Lite

In a recent study [10], Google proposed architectural changes to the EfficientNet architecture to make EfficientNet more mobile friendly. They accomplish this by firstly, removing squeeze-and-

excitation layers from network, reasoning that they are not well supported for heterogeneous hardware, and secondly, replacing all Swish activation functions with RELU6 which they reported to improve post-training quantization efforts. Finally, they also fixed the stem and head while scaling the new Lite models up to reduce the size and computations of scaled models.

The above changes spawned the EfficientNet-Lite architectures which come in 4 scaling variants (EfficientNet-Lite0 to EfficientNet-Lite4). The lite variant of EfficientNet is much lighter compared to the original model and is better optimized for edge devices like mobile phones.

# Chapter 3

## Introducing the Dataset

### 3.1 FaceForensics++

For our project, we use the FaceForensics++ dataset as introduced in chapter 2 as it is a robust dataset that included the use of several deepfake techniques which allows us to perform reliable deepfake classification benchmarking. In this chapter, we briefly describe our dataset.

For reference, we acquired this dataset by submitting a request form to the FaceForensics++ team (special thanks to the FF++ team for curating this dataset and allowing us to use it). We were then provided with a download script which allowed us to access the videos. We downloaded the C23 compressed videos rather than the raw or C40 compressed videos to strike a balance between video quality and download speed.

The FF++ dataset comes with several deepfake videos and source videos (real). We chose to perform deep fake detection of “*Youtube*” videos as it is the only class that received augmentation from several deepfake techniques. In total, we have 1 real class (the source Youtube videos), and 5 deepfake classes (using various deepfake techniques on the real Youtube class), namely – these fake classes are Deepfakes, Face2Face, FaceShifter, FaceSwap, and NeuralTextures. We clarify our data selection using illustrations in the sections to come.

### 3.1.1 Youtube (“Real”)



Figure 3.1.1.1 – A single frame of an example source (real) video

\* “Youtube” is a video collection of Youtube videos typically depicting content such as newscasting, video blogs, etc.

### 3.1.2 Deepfakes



Figure 3.1.2.1 – Demonstrating the effects of “Deepfakes”



### 3.1.3 Face2Face



Figure 3.1.3.1 – Demonstrating the effects of “Face2Face”

### 3.1.4 FaceShifter



Figure 3.1.4.1 – Demonstrating the effects of “FaceShifter”

### 3.1.5 FaceSwap



Figure 3.1.5.1 – Demonstrating the effects of “FaceSwap”

### 3.1.6 NeuralTextures



Figure 3.1.6.1 – Demonstrating the effects of “NeuralTextures”

# Chapter 4

## Computation Environment

For our training set-up, we use:

- CPU – Intel i7-9700 with a RAM of 32 GB
- GPU – NVIDIA GeForce RTX 2070 with 8 GB memory
- PyTorch as the primary deep learning framework
- Python – Version 3.10.8

With our current hardware set-up, we are unable to perform large batch size training processing or any training involving large amounts of data as commonly seen in research outputs like [3]. Instead, we make do with our computational limitations and our workarounds for any hardware limitation will be detailed in our implementation chapter.

# Chapter 5

## Detailed Methodology & Implementation

In this section, we discuss the thought process and methodology that went into project execution.

### 5.1 Problem Formulation

The first step is to properly define and formulate our deepfake detection problem. As introduced in chapter 3, our FaceForensics (FF++) dataset is in video format. Our primary studied architecture, EfficientNet, is mainly designed to handle image inputs and hence, there is a need to reduce the video processing problem to an image classification one. Since videos are fundamentally “sequences of images,” the most obvious approach is to break each video into individual image frames and use each frame as model input. We go into detail about our frame extraction method in the next section.

The next consideration is defining the classes in our classification problem. In the FF++ dataset, we are presented with several categories. While a detection model trained to identify deepfake techniques (multi-class classification) would be helpful for general deepfake research, our goal is to create a model that is able to tell real and deepfake images apart and hence our problem will need to be a **binary image classification problem**. To achieve this, we group the various deepfake categories into a single “fake” class while preserving the “Youtube” class (original source data) as the “real” class.

### **5.1.1 Model Predictions**

Since we are dealing with a binary image classification problem, we design our neural network to output a single class probability (i.e. 1 output neuron) whereby values closer to ‘1’ signify the model’s confidence that the image is ‘fake’ and values closer to ‘0’ signify the model’s confidence that the image is ‘real’. To obtain the model’s prediction, we apply the sigmoid function to the output logits to obtain a prediction probability value from 0 to 1. We also set 0.5 as the largest threshold probability such that the model predicts that an image is real (i.e. if the probability tips over 0.5, the model predicts the image as fake).

### **5.1.2 Binary Cross-Entropy Loss**

Our loss function needs to be the Binary Cross Entropy (BCE) criterion which quantifies the similarity between our model’s prediction probabilities and the ground truth labels of the image which is a hard label of 1 or 0 for ‘fake’ and ‘real’ classes respectively. Our objective is to reduce BCE loss during model training.

## **5.2 Data Pre-processing**

### **5.2.1 Frame Extraction & Sampling**

We established earlier that we need to operate on the video frame level, one problem that arises is that each video in our dataset is made up of hundreds and thousands of frames – which is going to cost us a lot to store and process. Therefore, we draw inspiration from [2] where the authors extracted a frame image every 30 frames to create a sufficiently large dataset while still maintaining image diversity. For our project, we did the same for the “Youtube” source videos (real), yielding about 17.5 thousand image frames. For our deepfake classes, we decided to sample

10 frames per video instead which yielded 10 thousand frames per deepfake category. The 10 frames were sampled randomly across all possible frames to enhance image diversity. We decided to handle the frame extraction of deepfake classes differently to tackle the class imbalance of deepfakes to real ratio of 5:1 and to reduce the burden on storage and processing.

### 5.2.2 Face Detection & Cropping

In FF++, the main difference between deepfakes and the original videos are that the facial region of the person subject were altered. Knowing this, it makes sense to attempt face cropping to allow our models to focus on only the relevant features and hence improve learning through the removal of background noise. Face cropping was also done in [2] where our idea originated from but our methods for face cropping differs. In [2], the authors used Dlib’s feature point detection script to identify faces. In our project, we chose to use OpenCV’s Haar cascade classifiers to detect faces which we found to be effective at detecting face-like objects and faster than popular options like MTCNN (Multi-Task Cascaded CNN). One issue with using the Haar cascade classifier is the high rate of false positives. While the classifier correctly identifies faces most of the time, it also misidentifies certain objects such as wall patterns, posters, clothing, and blurry backgrounds as faces. Additionally, the classifier tends to also return extremely blurred faces by capturing the faces of people in the background and this leads to tiny face crops that is too pixelated for use. The authors of [2] also faced the same issues – which they addressed by manually screening each extracted picture. In our case, we made two important modifications to the face cropping logic to obtain higher quality face crops: (1) we dial up the **scale factor** and **minimum number of neighbours** parameters which reduces the detection of smaller objects and mandates the classifier to return results with higher confidence respectively, and (2) we only **extract the largest face crop** identified by the classifier for any given frame. This improved the quality of our face crops greatly

and while it did oust a few true positive (real faces) cases as well, we were still left with an overall large dataset to work with (less than 10% data reduction).

Another noteworthy deviation from [2] is that while the authors of the paper extracted multiple faces from each frame, we only extracted the largest face crop. We did this for two reasons: (1) most of the videos have a “lead” subject which means there is only 1 main face to extract, and (2) during model training, each face crop will be handled independently and hence whether multiple faces were extracted from a single frame or not should not affect the model’s ability to generalize. We acknowledge that a limitation to our approach is that we reduce the variability in face crop sizes we train our model with but we believe that the increase in data quality is well worth this limitation. We also theorize that a model trained on just the largest face crop of each frame will be able to generalize on other smaller face crops found within those frames post-training by expanding the scope of the face extraction (but this goes beyond our project objectives).

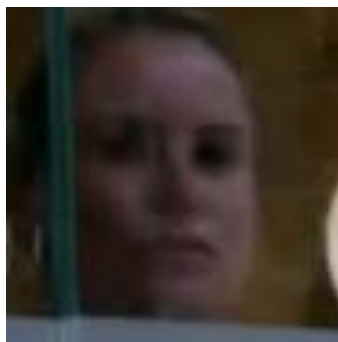


Figure 5.2.2.1 – example of a false positive: “reflections”



Figure 5.2.2.2 – example of an extracted video frame



Figure 5.2.2.3 – example of a face crop

## 5.2.3 Data Sampling, Loading and Transformation

### 5.1.3.1 Tensor Loading

For CNNs to process images, they first must be transformed into tensors which are numerical representations of the image. There are two ways to go about this, we could read images and transform them into tensors as we need them OR we can perform the loading and transformation all at once and store the tensors in memory ahead of time. The first method requires us to keep track of the file paths of every image and perform a file read every time we need an image, which



induces a lot of I/O cost and makes data access impractically slow. The second method is the fastest and most preferred as the data loading and transforming overhead is incurred at the start, making data access extremely smooth during training and inference.

However, the disadvantage of the second method is that memory is often small meaning the number of image tensors we can load into memory is constrained as well. Often, we want to keep our tensors in the GPU memory as the GPU can parallelize tensor computations. Unfortunately, our GPU only has 8 GB of memory which is too little for our training requirement. To address this, we decided to load our image tensors into our CPU memory which has 32 GB RAM and transfer tensors to GPU memory for processing in batches during runtime. This allows us to maximize our computational resources without overburdening any component.

#### ***5.1.3.2 Data Sampling & Shuffling***

Despite having roughly 9 thousand face crops per deepfake category and 16 thousand face crops for the real category, we need to perform further data sampling to respect our hardware constraints as described above.

We randomly sampled 3000 “fake” face crops from each deepfake category and 10000 real face crops. We then shuffled the data to ensure that the dataset do not retain any knowledge regarding the original sequence of the images as we are performing each image classification independently. This results in 15 thousand “fake” face images (60%) and 10 thousand “real” ones (40%). This slight class imbalance was intentionally introduced to reflect the original nature of the data distribution (there are more deepfakes than original) without over doing it.

Our chosen sampling size would occupy ~80% of the operating environment’s memory and we found this to strike the ideal balance between computing performance (as exceeding memory usage will severely slow down the computer) and having a large enough dataset for effective training.

#### ***5.1.3.3 Train-Test-Validation split***

Of the 15000 fake images and 10000 real ones, we performed a 70-15-15 train-test-validation split to train, validate, and test our models. This means that 70% of data is used to train the models’ weights, 15% of it is used to check if the model is overfitting during training and the remaining 15% is used to test the model’s performance post-training. Note that this split was handled at the category level, this means that there are an equal proportion of images from each deepfake and real category across training, validation, and testing datasets. The table below illustrates our splitting methodology.

Dataset	Sampled	Train	Test	Validation
“Youtube” (Real)	10000	7000	1500	1500
Deepfakes	3000	2100	450	450
Face2Face	3000	2100	450	450
FaceShifter	3000	2100	450	450
FaceSwap	3000	2100	450	450
NeuralTextures	3000	2100	450	450

Table 5.1.3.3.1 Illustration of data sampling and training-testing-validation split

#### ***5.1.3.4 Data Augmentation & Regularization***

When training CNNs, it is common to perform data regularization to prevent overfitting by controlling the model’s complexity and enhancing generalization. In our project, we accomplish

image regularization by: performing random horizontal image flips, introducing random image rotations up to 10 degrees, and introducing colour jitter which affects brightness, contrast, saturation, and hue. Note that these augmentations are only performed on training data, validation and testing data are unaffected. Additionally, since EfficientNet, our primary studied model accepts image resolution sizes depending on its version (e.g. EfficientNet B0 takes 224x224 images), we resize all image tensors accordingly during our data loading phase. The following diagram shows the effect of our image augmentations.



Figure 5.1.3.4.1 – Dataset images without regularization



Figure 5.1.3.4.2 – Augmented/Regularized images

#### ***5.1.3.5 Frame Categorical Tagging***

Even though we are carrying out a binary classification task, it is helpful to know how our models fare at identifying fake images produced by the various deepfake categories. To accomplish this, we tagged each image input with their originating category (this does not affect the binary prediction label classes). This means that while the model only trains on the “real” and “fake” labels, during testing, we can acquire statistics on how the model fares for the various deepfake categories by tracking the originating categories of each image evaluated by the model.

## **5.3 Model Implementation**

In this section, we briefly go through some of the implementations of EfficientNet that helped progress this project.

### **5.3.1 Creating EfficientNet from scratch**

Our very first starting point was to try creating EfficientNet from scratch. Recreating [3]’s neural network architecture is an extremely daunting task but we were able to do so with Aladdin Persson’s follow along video tutorial [18] where he walked us through how to create EfficientNet using Pytorch modules from scratch. We found that Persson’s implementation of EfficientNet stayed faithful to the paper’s implementation except for a small error where the convolution layers had the wrong kernel size which resulted in models 2-3x larger (parameter-wise) than those described in [3]. Fixing this error aligned our implementation with the paper’s. The process of making EfficientNet was enlightening and helped us in understanding the components that made up the architecture which will prove extremely valuable in the sections to come where we need to modify the architecture.

### **5.3.2 Pytorch’s official implementation**

Despite having our own implementation of EfficientNet, we still turned to Pytorch’s official implementation. The main reason for this is because Pytorch’s implementation comes with pretrained ImageNet-1K weights (which we discuss more on later) and this allows us to seamlessly perform transfer learning without having to train our EfficientNet on ImageNet which is considerably more expensive than our current tasks in terms of the training data size and the number of classes (1 thousand) involved.

### **5.3.3 EfficientNet-Lite**

To implement EfficientNet Lite, we simply modified the architecture of Pytorch’s implementation of EfficientNet to match [10]. However, we found that it is much more effective to implement [9]’s EfficientNet-Lite because the author provided pretrained imagenet-1k weights specific to EfficientNet-Lite. The choice to leverage [13]’s for their pretrained weights improved training significantly as we will show in our EfficientNet-Lite Experiments in the next chapter.

### **5.3.4 Modifying the classifier and enabling gradients**

To run any of the proposed implementations above, we must make modifications to the final fully connected layer (the classifier) as the EfficientNet implementations are still fitted with the classifier for ImageNet-1K (1000 classes). We can achieve this by adding a Sequential component that incorporates (1) the correct dropout value of the current EfficientNet version (different version use different dropout rates), (2) a linear layer that takes in the same number of inputs as the number of outputs in the preceding layer, and (3) a classifier that handles the appropriate number of output classes (1 in our case). We also turn on gradient computation for all model parameters as we want to train the whole model to fit our deepfake classification problem.

```
def pytorch_effnet(num_classes, pretrained = False):
    if pretrained:
        model = efficientnet(weights = 'IMAGENET1K_V1')
    else:
        model = efficientnet()
    in_feats = model.classifier[1].in_features
    dropout = model.classifier[0].p
    model.classifier = nn.Sequential(nn.Dropout(dropout, inplace = True),
                                     nn.Linear(in_features=in_feats,
                                               out_features=num_classes, bias=True))

    for param in model.children():
        param.requires_grad_(True)

    model = model.to(device)
    return model
```

5.3.4.1 – Python logic to modify EfficientNet architecture

## 5.4 Model Learning Strategies

Our model training strategies were progressively tuned throughout the course of the project, the following sections highlights the overall strategies that we found worked best for us consistently.

### 5.4.1 Model Hyperparameters

- Learning Rate = 0.0005
- Weight Decay = 0.00001
- Number of training epochs = 30
- Early Stopping Patience = 5
- Training Batch Size = 32
- Warm-up Period = 10

Note here that our batch size is limited by our GPU memory, [3] proposes batch size to be 4096 but without the right hardware, this feat is tricky to accomplish. We found the learning batch size

of 32 to be our upper limit that our computing environment can run on and it complements the rest of our training settings well.

### **5.4.2 Learning Optimizer: Stochastic Gradient Descent**

Optimizers are algorithms or learning strategies that adjust a model's parameters to improve accuracy and reduce loss. We first tried [3]'s RMSProp optimizer settings but was unable to replicate its effectiveness demonstrated in the paper. This could be due to our differing batch size and set up.

We also tried ADAM optimizer, but we found that it interfered with our linear warm-up routine (see section 5.4.4) and hence opted for the SGD optimizer which worked great with the warm-up strategy. This configuration helped us obtain training stability.

### **5.4.3 Learning Scheduler: Cosine Annealing**

Learning rate schedulers modifies learning rate as training progresses methodically to help model achieve better training stability and convergence. For instance, the Plateau Scheduler drops the learning rate (e.g. reduce learning rate by a factor of 10) when it detects that the monitored metric (e.g. validation loss) has not improved in the past few training epochs.

We tried various learning schedulers such as StepLR, Plateau, Cosine Annealing and Cosine Annealing with Warm restarts. Without going into the details of each scheduler, we found that Cosine Annealing worked the best for us.

Cosine Annealing learning rate scheduling operates by reducing the learning rate in a manner that mimics the Cosine function. The smoothening of the learning rate helped many of our models achieve training stability and better convergence.

### 5.4.4 Linear Warm-up

We also found that incorporating a learning rate warm-up scheduler [14] was effective at smoothening the training process. Our project uses a warm-up period of 10 which means that the model starts off learning at a tenth of our proposed learning rate and slowly increases its learning rate every epoch uniformly (linearly) until reaching our intended learning rate by the tenth epoch. We note here that our Cosine Annealing learning scheduler only begin adjusting learning rates after the warm-up period.



Figure 5.4.4.1 – Example training plots of model training without any gradient warm-up or learning rate scheduling (unstable learning observed)



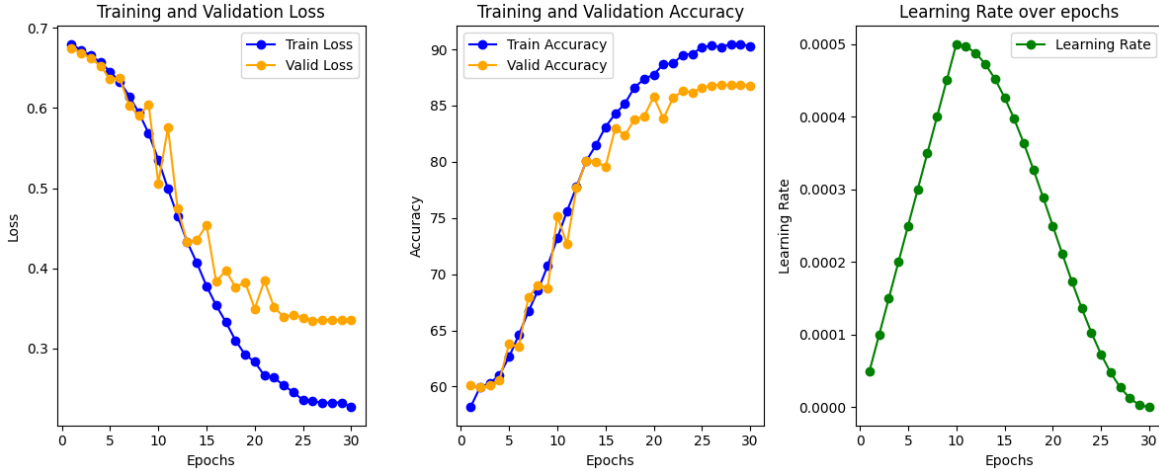


Figure 5.4.4.2 – Training plots demonstrating that introducing linear warm-ups and cosine learning rate annealing stabilizes training and improves model convergence.

## 5.4.5 Weight Initialization & Transfer Learning

Another important aspect of our learning strategy is weight initialization. The initial weights of the model influence the starting point of the optimization process and can affect the time taken for the model training to converge and the quality of the found solution.

The authors of [11] found that transfer learning using weights obtained from training against ImageNet classifications improved model performance. In our experiments, we found that using pre-trained ImageNet weights as initialization was a great starting point and help us achieve better training stability and model performance consistently.

## 5.4.6 Early Stopping

To avoid the problem of over-fitting, we test our model performance against the validation set after each epoch training and terminate the training early if the validation loss does not improve within a certain number of epochs. We call this our patience variable and our models are trained with

patience = 5, this means that the model will stop training if it has been exactly 5 epochs since its last seen best validation score. At the end of training, we recover the model state that produced the best validation score during training.

### **5.4.7 Fine-tuning**

Lastly, we employed a different set of training algorithm for fine-tuning. We use fine-tuning to improve the performance of post-trained models and for adjusting models during model pruning without overtraining them. The key difference between our fine-tuning algorithm and our normal training algorithm is that we train for fewer epochs, at lower learning rate, we use the ADAM optimizer with no learning scheduler and have a shorter early stopping patience of 3.

- number of epochs = 20
- patience = 3
- learning rate = 0.0002
- weight decay = 0.00001
- optimizer: ADAM
- learning rate scheduler: None
- warm-up period: 0 (no linear warm-up)

## **5.5 Model Scaling**

Our experiment here revolves around scaling EfficientNet and finding out the effects of model scaling. Thankfully, Pytorch's EfficientNet provides support for all EfficientNet versions and included the option to load pretrained weights for ImageNet-1K for each version. However, increasing the EfficientNet version also increases the training time and memory requirement due

to the increase in parameter size in larger architectures. Moreover, due to our GPU memory limitation, we found that the biggest model version we could implement was EfficientNet-B3.

### **5.5.1 Model Scaling Experiment**

To test the effectiveness of model scaling on our deepfake problem, we trained EfficientNet B0, B1, B2 and B3 using their respective pre-trained ImageNet weights and compared their test accuracies and model sizes. After which, we will select the best model to be used as our baseline model for further tuning.

## **5.6 Architectural Changes – EfficientNet Lite**

As discussed in our literature review, EfficientNet Lite is a lighter variant of EfficientNet that is more edge device friendly. Our experiment here revolves around finding out whether EfficientNet-Lite yields any performance gains or even accuracy gains.

### **5.6.1 Removing Squeeze & Excitation Layers**

In EfficientNet-Lite, the Squeeze & Excitation layers were removed as they were not well-supported on edge devices.

This can be easily accomplished by recursively searching a model for Squeeze & Excitation Layers and replacing them with the Torch’s Identity layer which acts as a placeholder by forwarding its inputs as outputs (i.e. no computations performed).

```
# removing SqueezeExcitation layers
def remove_se_layers(model):
    for name, child in model.named_children():
        if isinstance(child, SqueezeExcitation):
            setattr(model, name, nn.Identity())
        else:
            remove_se_layers(child)

remove_se_layers(model)

return model
```

#### 5.6.1.1 – Removing Squeeze and Excitation Layers

### 5.6.2 Replacing Swish Activation with Relu6

Additionally, the Swish activation (SiLU) functions are replaced with Relu6 which is computationally simpler and more compatible with post-training quantization.

Likewise, this change is brought about by recursively searching the model for use of SiLU functions and replacing them with ReLU6:

```
### Model Changes: EfficientNet-Lite
def make_lite(model):

    # recursively changes SiLUs into ReLUs
    def replace_silu(model):
        for name, child in model.named_children():
            if isinstance(child, nn.SiLU):
                setattr(model, name, nn.ReLU6())
            else:
                replace_silu(child)

    replace_silu(model)
```

#### 5.6.2.1 – Replacing SiLUs with ReLU6s

Executing the above two changes will convert EfficientNet to an EfficientNet-Lite architecture. Specifically, introducing said changes to EfficientNet-B0, turns it into “EfficientNet-Lite0” the smallest variant in the EfficientNet-Lite family.

## 5.7 Model Pruning

Recent works like [34, 35] have showed that model pruning can reduce the number of parameters and FLOPs without the model suffering large accuracy loss. In this section, we investigate if pruning can reduce our model footprint while preserving accuracy.

### 5.7.1 Taylor’s Importance Pruning

Rather than removing network components randomly, we can strategically prune neuron weights by evaluating weights and removing them based on some understanding of their level of importance to the network. One common and simple strategy is magnitude pruning which involves identifying and pruning weights with small magnitudes – this strategy assumes that smaller weights contribute less to the network and are hence far less important. However, in [17], the authors challenged the approximation assumption by questioning whether “magnitude equals saliency”.

In the papers [15] and [16], the authors employed Taylor’s Importance pruning, a technique aimed at identifying the weights within a neural network that have minimal impact on the model’s performance when removed. Taylor’s Importance pruning operates by evaluating the importance of each neuron based on its Taylor expansion. Mathematically, the Taylor expansion approximates a function using its first-order (gradient) and second-order (change in gradient) derivatives. In the context of neural networks, Taylor’s importance helps the pruner approximate the change in the model’s output concerning each neuron’s activation.

Specifically, to prune using this metric, we feed sample input images into the model during pruning, and have the pruner evaluate the Taylor’s Importance of each neuron. This metric helps

determine which weights contribute the least to the model's overall performance, allowing for targeted pruning and model compression without significant loss in accuracy or functionality. All our pruning experiments will incorporate the use of Taylor's Importance as the magnitude pruning metric.

### **5.7.2 Structured Pruning**

Next, we must consider if we want to perform structured pruning or unstructured pruning. In unstructured pruning, pruned weights are simply set to 0 but not actually removed from the model and hence model size remains the same despite the pruning. This would be beneficial computation wise if the deep learning framework supports sparse tensor computation where the processing of tensors that have majority of its elements set to zero is greatly accelerated. Currently, Pytorch's implementation of sparse tensor representation and computation is still in the beta development stage and so we turn to structured pruning, which removes pruned weights entirely from the model, reducing model size.

In general, pruning our model structurally requires us to respect the inter-dependencies between the layers of the model. For instance, when we remove certain components, similar changes must be made to the layers that come before and after it (e.g. pruning away output neurons in a layer means that the subsequent layer must adjust their inputs to matched the reduced output dimension of the pruned layer). This is relatively simple to accomplish in sequential networks where the pruning interdependencies can be resolved by propagating the required changes sequentially across the networks. However, in complicated neural networks like EfficientNet where there exist non-sequential components like skip connections, the dependencies are much more complex and makes structured pruning much more challenging.

To accomplish structured model pruning, we will leverage [13]’s implementation which cleverly prunes neural networks by identifying and grouping “minimal removable units” together. By pruning in groups rather than each component at a time, their mechanism ensures that changes to any component of the neural network is propagated to all dependent components.

Our experiments in the following chapter will show that structured pruning will reduce model size and our aim is to perform pruning in a manner that preserves accuracy.

### **5.7.3 Local Pruning VS Global Pruning**

In local pruning, the model is pruned by targeting specific substructures such as layers within the model. For instance, a 20% local pruning ratio will prune 20% of weights from specified layers. This contrasts with global pruning where the whole network is considered for pruning. To perform local pruning effectively, it is important to have a good understanding of the inner-workings of the layers that are selected for pruning, which is not a trivial assessment to be made. To this end, we propose the use of global pruning instead which allows the pruner to prune away weaker contributors in the network to reduce model size. Nevertheless, we will also include a local pruning experiment in the following chapter to compare with the global pruning method.

Our interest and confidence in global pruning stems from our use of Taylor’s importance as a pruning metric. Our theory is that, by pruning at the network level, the pruner can determine the weakest contributors across the entire network and eliminate them to reduce model footprint which is not the same as pruning away the weakest contributors at every layer.

While global pruning can theoretically lead to the best model performance, overdoing the pruning ratio can lead to entire substructures of the network being pruned away and this can cause

significant performance degradations. In our experiments, we will investigate the limits of how much we can reliably prune our model.

### **5.7.4 One-shot Pruning VS Iterative Pruning**

As the name suggest, one-shot pruning means that we prune away the desired percentage of our model and fine-tune it exactly once. In contrast, iterative pruning performs the pruning process by progressively and repeatedly pruning away the network and fine-tuning it over a set number of iterations until the desired pruning ratio is fulfilled.

The trade-off between the two pruning methods is that one-shot pruning can be completed much more quickly while iterative pruning may yield a pruned model with better performance due to the fine-tuning that happens in-between each pruning iteration.

To accomplish one-shot pruning, we set a desired pruning ratio and have our pruner prune the model in a single step, we then perform fine-tuning to recover the model's accuracy after the prune.

For our iterative pruning experiments, we first set a desired pruning ratio and have our model prune it for 5 iterations, at the end of each iteration, the model goes through fine-tuning to recover performance.

In our experiments, we investigate the effects of one-shot vs iterative pruning.

## **5.8 Knowledge Distillation**

Our last and perhaps largest contribution is the application of Knowledge Distillation. So far, we have discussed methods that shrink a model's size at the cost of accuracy. In this section, we



discuss the ways that we can recover accuracy by using a reference teacher model that is much bigger but superior in performance to teach our shrunken models (students).

### 5.8.1 Distillation Loss Formulation

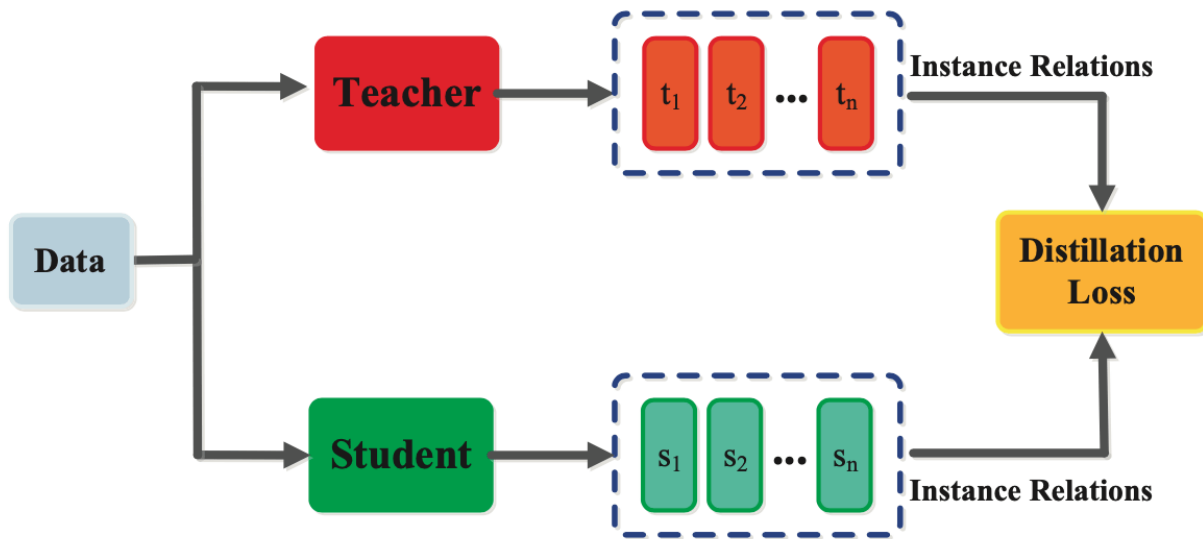


Figure 5.8.1.1 – Illustration of Knowledge Distillation

Our implementation of knowledge distillation involves the teaching of prediction loss values that belong to the teacher model to the target student model. This is done by getting the student to model the teacher's output predictions closely. The underlying rationale behind this method is the belief that the teacher model holds important internal understanding about the classification problem and by getting the student to mimic the teacher's losses, the student can learn some of those internal representations. For instance, the teacher model might have reasons for believing that a face image is 75% likely to be fake and 25% real – this can be much more useful than the ground truth labels of 'fake' and 'real' as it tells us that most of the signals in the image point to it being fake but a quarter of it has the characteristic of a real image. It falls onto the student model to try and create an internal representation to match this knowledge obtained from the teacher.

To achieve knowledge distillation, we must reframe our loss objective to also consider the similarity between our student's and teacher's prediction. We refer to the difference between the teacher and student prediction values as **soft label loss** (because the teacher's predictions are a floating-point prediction value between 0 and 1) and the difference between the student's prediction and the ground truth label as **hard label loss** (since the actual label is an integer value that is strictly 0 or 1). We have established earlier that the hard label loss can be computed by taking the binary cross entropy between the student model's prediction probabilities and the ground truth to measure the difference between the student's prediction confidence and the actual binary labels. We do the same for soft label loss by applying binary cross entropy between the student model's prediction and the teacher's prediction. To combine the two losses, we use an alpha parameter to adjust the weight contribution of the soft label and hard label losses as follows:

$$L_{KD} = \alpha * \text{soft label loss} + (1 - \alpha) * \text{hard label loss}$$

- Where  $L_{KD}$  is our revised loss term to include knowledge distillation loss.

In our experiments, we will be tuning this alpha parameter in magnitudes of 0%, 25%, 50%, 75% and 100% to discover the influence of our proposed knowledge distillation approach on training.

### 5.8.2 Augmentation to Training Logic

To incorporate knowledge distillation, we must revise our training approach. During knowledge distillation training, the teacher model's weights will be frozen to prevent any further learning and changes to the model. Furthermore, the batch of input images fed to the student model during each training step is also fed to the teacher model. We then compute training loss using our distillation loss formulation above and have the model train to reduce this loss.

It is important to note that the proposed loss function is only used to evaluate model fit during training. During the model validation phase, we revert to our original loss function to evaluate if our model improves. Our theory is that reducing a mixture of both soft and hard label loss during training has a positive impact on hard label loss during validation and hence resulting in model performance improvements (i.e. increased accuracy).

### **5.8.3 Teacher Model**

To conduct our proposed knowledge distillation, we would need to first designate a teacher model. This model should be chosen for its accuracy without paying regards to its size. For our knowledge distillation experiments, we will choose the best performing model (accuracy-wise) from previous experiments to become the teacher model.

### **5.8.4 Knowledge Distilled Training and Pruning**

There are two phenomenon that we are interested in studying: (1) the effects of knowledge distillation on model training and (2) the effects of knowledge distillation as a fine-tuning mechanism during the pruning phase. In the next chapter, we conduct experiments that answers these two study objectives by incorporating knowledge distillation during training and pruning.

# Chapter 6

## Experiments Results and Findings

In this section we discuss our model evaluation metrics, experimentation results and findings.

### 6.1 Evaluation Metrics

To properly quantify whether our project has met its objectives, we must first define what constitutes success. In the following sections we discuss three important metrics: model accuracy, model size and model inference speed.

#### 6.1.1 Model Accuracy

Model accuracy measures the model’s ability to correctly differentiate deepfakes from real images by evaluating the percentage of predictions that correctly match the label during model testing.

It is the most common metric used to benchmark classification problems and a metric we prioritize since we want to produce a reliable deepfake detection model. Determining a good model accuracy on the other hand is tricky. In [2], the authors managed to achieve an impressive detection result of **97.90%** on the FF++ dataset, however they used the *EfficientNet-V2 (S)* model from [4] which uses significantly more model parameters than *EfficientNet-B0* from [3] (24 million params vs 5.3 million params). In [5], the authors achieved a remarkable **98.9%**, **98%** and **96.2%** detection accuracy for the FaceShifter, Face2Face, and NeuralTextures deepfake categories respectively. However, the proposed model used in [5] had 18.2 million parameters which is still relatively computationally expensive without the proper hardware.

The official FF++ website [12] also offers benchmark results for the binary classification of almost every deepfake category in the dataset. We compiled the best scores obtained for each category obtained by various submitted models as reference but one caveat is that we are unable to ascertain the size of the models used to obtain the results which impairs our ability to make a robust assessment.

Face2Face	FaceSwap	NeuralTextures
98.9	98%	96.2%

Table 6.1.1 – [5]’s model accuracy on three FF++ categories

Deepfakes	Face2Face	FaceSwap	NeuralTextures	Pristine (“Real”)
100%	95.6%	98.1%	96.7%	99.2%

Table 6.1.2 – Consolidated top accuracy for each category found on the FF++ official benchmark website

With the above statistics in mind, we believe that a good goal is to produce a lightweight deepfake detection model that is capable of reaching ~90% accuracy across the various categories to remain competitive with existing solutions.

### 6.1.2 Model Size

We also want to evaluate our models by their sizes. To this end, we will consider both **model size in terms of megabytes (MB)** and **the number of parameters in our proposed model**.

To quantify success in this section, we must show that our proposed model size reduction techniques is capable of significantly reducing the baseline model size while preserving our accuracy requirements as expressed in 6.1.1.

### **6.1.3 Inference Speed**

The primary motivation for pursuing this project was to explore lightweight deep learning techniques that can push deepfake detection deep learning models towards deployment on constrained computing environments like edge devices like mobile devices. In such environments, inference latency is extremely important and therefore, we also measure the inference speed of our proposed models. To quantify the success in this section, we must show that our proposed lightweight deep learning techniques improve inference speed under computational constraints. We accomplish this by turning off GPU support and relying on purely our operating environment’s CPU for inference during testing to measure the latency cost incurred by our reduced models compared to their baseline counterparts. We acknowledge that a limitation of this evaluation lies in the fact that a computer’s CPU is generally faster and much more powerful than a mobile phone’s CPU since mobile CPU chips generally prioritize power efficiency over high performance. To compute CPU inference speed, we took the average batch image inference timing obtained using 32 frames per batch.

## **6.2 Experiments**

### **6.2.1 Transfer Learning using ImageNet-1K**

Our first assessment is the applicability of transfer learning using ImageNet-1K training weights for our deepfake detection task.

We employed our proposed training strategy as describe in chapter 5.4 on EfficientNet-B0 without initializing the pre-trained weights. We then execute the same training strategy on EfficientNet-B0, this time using pre-trained ImageNet weights as initialization.

```

Accuracy of the network on tested frames: 64.72 %
Accuracy for class: Deepfakes is 91.11%
Accuracy for class: Face2Face is 76.89%
Accuracy for class: FaceShifter is 91.11%
Accuracy for class: FaceSwap is 67.33%
Accuracy for class: NeuralTextures is 76.67%
Accuracy for class: youtube is 40.87%

```

Figure 6.2.1.1 – Model performance of EfficientNet B0 without transfer learning

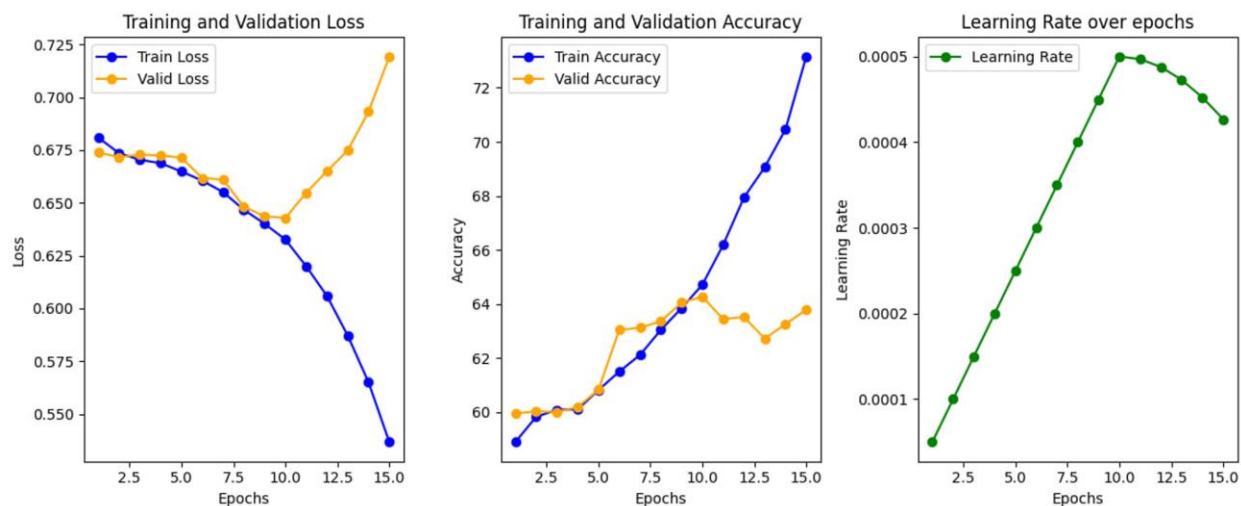


Figure 6.2.1.2 – Training plots for EfficientNet B0 (no transfer learning)

```

Accuracy of the network on tested frames: 90.03 %
Accuracy for class: Deepfakes is 95.56%
Accuracy for class: Face2Face is 94.44%
Accuracy for class: FaceShifter is 90.44%
Accuracy for class: FaceSwap is 90.00%
Accuracy for class: NeuralTextures is 84.67%
Accuracy for class: youtube is 88.53%

```

Figure 6.2.1.3 – Model performance of EfficientNet B0 with transfer learning

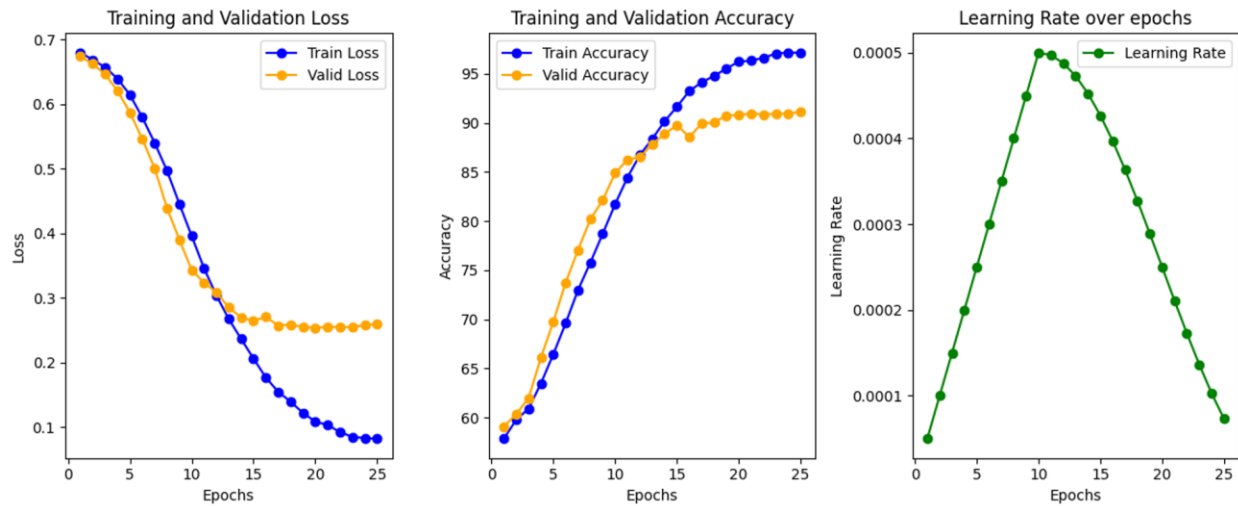


Figure 6.2.1.4 – Training plots for EfficientNet B0 (with transfer learning)

As observed in the above figures, not only does EfficientNet B0 perform considerably better with transfer learning, we find that transfer learning improves training stability and help us secure better model convergence.

**Moving forward:** as recommended by [11], ImageNet transfer learning is a good starting point for our binary deepfake classification task and we will hence, be using ImageNet pretrained weights for the initial training of all our subsequent models in the experiments to come.



## 6.2.2 EfficientNet Model Scaling Effectiveness

Our next task is to evaluate the effects of model scaling on EfficientNet B0, B1, B2 and B3. It is well-established that scaling up the EfficientNet version increases model size and model parameter count and hence our primary concern here is the model accuracy. We want to investigate if scaling the model yields us significant accuracy gains as promised by a larger and more complex architecture.

In this experiment, each model was pre-trained using their respective Pytorch’s ImageNet weights. Unsurprisingly, we found that the larger the EfficientNet version, the longer it took to train it. We consolidate the model statistics for brevity and conciseness:

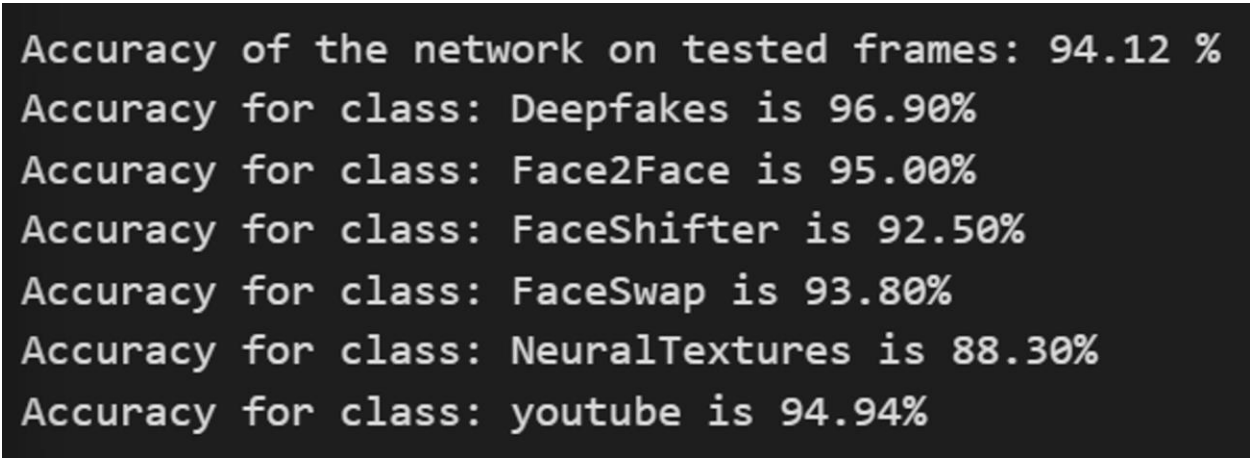
Performance (Acc%) / Model	EfficientNet-B0	EfficientNet-B1	EfficientNet-B2	EfficientNet-B3
Overall Acc	90.03%	86%	89.52%	86%
Deepfakes	95.56%	94%	95.33%	96.89%
Face2Face	94.44%	89.56%	93.11%	92.89%
FaceShifter	90.44%	80.89%	89.78%	90.22%
FaceSwap	90%	83.11%	88.22%	87.56%
NeuralTextures	84.67%	82.89%	87.78%	85.78%
“Youtube” (Real)	88.53%	87.4%	87.53%	80.33%

Table 6.2.2.1 – The consolidated performance for each EfficientNet version

Interestingly, EfficientNet-B0 performed the best across all categories, only losing to EfficientNet-B3 in the detection of *Deepfakes*. One reasonable explanation to our observation is that our training data sample is simply too small – larger EfficientNet architectures require much more training data

than what we have allowed ourselves to achieve better generalization. However, as discussed earlier, our training size is constrained by the size of our operating environment’s memory.

**Moving forward:** Obtaining an accuracy of 90% on our base EfficientNet-B0 model is encouraging as it clears our initial accuracy goals. The challenge is now to maintain this accuracy while reducing the model footprint. Since, we found EfficientNet-B0 to be our lightest model and best performer, **we will set EfficientNet-B0 as our baseline model**. Before moving to the next experiment, we fine-tuned our EfficientNet-B0 model by applying our fine-tuning algorithm and obtained **even better model performance**:



```
Accuracy of the network on tested frames: 94.12 %  
Accuracy for class: Deepfakes is 96.90%  
Accuracy for class: Face2Face is 95.00%  
Accuracy for class: FaceShifter is 92.50%  
Accuracy for class: FaceSwap is 93.80%  
Accuracy for class: NeuralTextures is 88.30%  
Accuracy for class: youtube is 94.94%
```

Figure 6.2.2.2 – Model performance of EfficientNet-B0 after fine-tuning

## 6.2.3 EfficientNet-Lite

### 6.2.3.1 *EfficientNet-Lite0*

In this section, we explore the potential of [10]’s proposed architectural changes to make EfficientNet-B0 (our current base model) even lighter by (1) removing the Squeeze and Excitation Layers and (2) replacing SiLU activation functions with ReLU6s.

Having made the appropriate changes and hence obtaining the EfficientNet-Lite0 architecture, we first perform training using B0's ImageNet Pretrained weights as we did for EfficientNet-B0. However, we found that the model converged to a poor solution with accuracy of only 71.28%, which is a steep decrease from our baseline B0 model. To improve accuracy, we tried to train the Lite0 model using our baseline model B0's model weights instead, doing so yielded a better Lite0 model accuracy of 77.84% but its performance is still below acceptable levels given that our B0 model is currently at ~94% accuracy. Finally, we leveraged [9]'s ImageNet pre-trained weights which was obtained by training Lite0 against ImageNet-1K. Shockingly, this variant of transfer learning yielded the best Lite0 results which achieved 87.89% accuracy which alludes to the idea that the removal of squeeze and excite layers and Swish activation changes the manner the network handles its internal representation drastically.

```
Accuracy of the network on tested frames: 71.28 %  
Accuracy for class: Deepfakes is 92.67%  
Accuracy for class: Face2Face is 84.67%  
Accuracy for class: FaceShifter is 84.00%  
Accuracy for class: FaceSwap is 74.67%  
Accuracy for class: NeuralTextures is 85.56%  
Accuracy for class: youtube is 51.73%
```

Figure 6.2.3.1.1 – Lite0 Performance (transfer learning using B0's ImageNet weights)

```
Accuracy of the network on tested frames: 77.84 %  
Accuracy for class: Deepfakes is 92.67%  
Accuracy for class: Face2Face is 82.89%  
Accuracy for class: FaceShifter is 84.89%  
Accuracy for class: FaceSwap is 71.33%  
Accuracy for class: NeuralTextures is 78.00%  
Accuracy for class: youtube is 71.67%
```

Figure 6.2.3.1.2 – Lite0 performance (transfer learning using B0’s weights)

```
Accuracy of the network on tested frames: 87.89 %  
Accuracy for class: Deepfakes is 96.00%  
Accuracy for class: Face2Face is 93.33%  
Accuracy for class: FaceShifter is 89.33%  
Accuracy for class: FaceSwap is 86.67%  
Accuracy for class: NeuralTextures is 87.56%  
Accuracy for class: youtube is 83.87%
```

Figure 6.2.3.1.3 – Lite0 performance (transfer learning using Lite0’s ImageNet weights)

**Moving forward:** Lite0’s accuracy performance experiences a small degradation from our B0 baseline. Fine-tuning it further using our fine-tuning algorithm pushes Lite0 towards ~90% accuracy. We also take this opportunity to compare Lite0’s model size, parameter size and inference speed with regards to B0. We found that Lite0 had ~600 thousand fewer parameters, was ~2.5MB smaller and had slightly faster CPU inference speeds. Despite the sizeable reduction in

model parameters and size, Lite0 is faring only slightly better in inference speed, this highlights the importance of measuring inference speed as part of our tests. Nevertheless, our experiment with Lite0 can still be considered successful as we achieved significant model reduction but still managed to maintain our 90% accuracy goal. Moving forward, future experiments will be looking at further reducing our Lite0 model size or improving its accuracy.

```

Accuracy of the network on tested frames: 90.78 %
Accuracy for class: Deepfakes is 97.20%
Accuracy for class: Face2Face is 96.60%
Accuracy for class: FaceShifter is 92.50%
Accuracy for class: FaceSwap is 92.80%
Accuracy for class: NeuralTextures is 89.10%
Accuracy for class: youtube is 87.92%

```

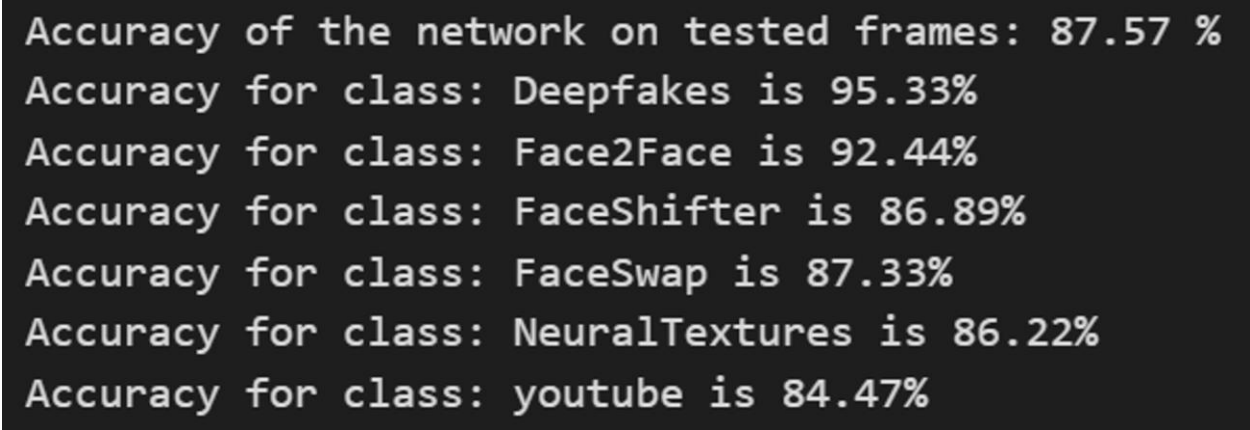
Figure 6.2.3.1.4 – Lite0 after further fine-tuning

Metric	EfficientNet-B0	EfficientNet-Lite0
Number of parameters	4,008,829	3,372,289
Size of Model	16.31 MB	13.74 MB
(Avg.) CPU Inference Speed	0.7654 seconds	0.7236 seconds

Figure 6.2.3.1.5 – Model size and inference speed comparison between B0 and Lite0.

### 6.2.3.2 *EfficientNet Lite Model Scaling*

EfficientNet Lite also comes with its own scaled versions. For instance, Lite0 is a scaled version of B0, Lite2 a scaled version of B2 and so on and so forth. We conduct a mini-experiment to satisfy our curiosity of the impact of model scaling in relation to the Lite's architecture. We accomplish this by training a Lite2 model using Lite2's pretrained ImageNet weight. We found that Lite2 performed closely to that of Lite0, with Lite0 still having overall higher accuracy prior to fine-tuning. The model scaling for the Lite architecture corresponded with our finding regarding the original EfficientNet scaling. With this in mind, we look towards improving Lite0 model for future experiments.



```
Accuracy of the network on tested frames: 87.57 %  
Accuracy for class: Deepfakes is 95.33%  
Accuracy for class: Face2Face is 92.44%  
Accuracy for class: FaceShifter is 86.89%  
Accuracy for class: FaceSwap is 87.33%  
Accuracy for class: NeuralTextures is 86.22%  
Accuracy for class: youtube is 84.47%
```

Figure 6.2.3.2.1 – Lite2 model performance

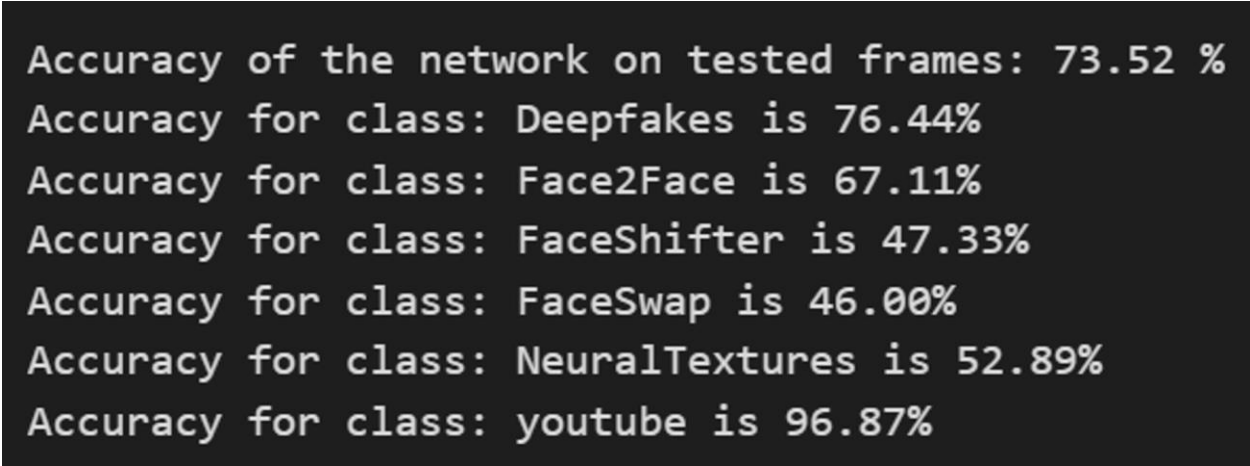
## 6.2.4 Model Pruning

In this section, we attempt to prune our Lite0 model to further reduce model size while preserving its high prediction accuracy using pruning methods as discussed in the previous chapter where we discussed our pruning strategies.

#### 6.2.4.1 Local vs Global

Earlier, we discussed that we are more motivated to perform global pruning as opposed to locally pruning due to our pruning methodology. Our experiment in this subsection supports our interest in global pruning by pitting it against the local pruning method. Here, to test our theory, we perform local-pruning using a 5% pruning ratio. Using our Taylor importance pruning metric, the pruner prunes away 5% of the least contributing weights for every layer. As shown in our experiment, even though both models are now 5% smaller, uniformly pruning every layer resulted in a more significant dip in accuracy. Contrasting this is our proposed global pruning strategy which also prunes 5% of the network, the resulting accuracy is still extremely close to the original LITE0 model, this is owing the fact the pruner selectively removed 5% of the least important weights in the network without regards to which layer they belonged to.

**Moving forward:** with this in mind, our remaining pruning experiments will be conducted using the global pruning method.



```
Accuracy of the network on tested frames: 73.52 %  
Accuracy for class: Deepfakes is 76.44%  
Accuracy for class: Face2Face is 67.11%  
Accuracy for class: FaceShifter is 47.33%  
Accuracy for class: FaceSwap is 46.00%  
Accuracy for class: NeuralTextures is 52.89%  
Accuracy for class: youtube is 96.87%
```

Figure 6.2.4.1.1 – Lite0 performance after locally pruned for 5%

```
Accuracy of the network on tested frames: 90.35 %  
Accuracy for class: Deepfakes is 97.78%  
Accuracy for class: Face2Face is 96.67%  
Accuracy for class: FaceShifter is 91.56%  
Accuracy for class: FaceSwap is 92.44%  
Accuracy for class: NeuralTextures is 91.11%  
Accuracy for class: youtube is 85.00%
```

Figure 6.2.4.1.2 – Lite0 performance after globally pruned for 5%

#### ***6.2.4.2 One-shot VS Iterative***

In this next experiment, using global pruning, we compare the effects of one-shot and iterative pruning. We arbitrarily selected a pruning ratio of 20% and performed both one-shot pruning and “five-shot” pruning (iterative), ensuring to perform model fine-tuning to recover accuracy loss after pruning. We found that even though iterative (5-shot) pruning took 5 times as long to perform, it yielded a ~2% accuracy improvement over the 1-shot pruning version. The parameter counts closely match each other at 2.54 million params which is roughly 80% of the original Lite0 model before pruning.

**Moving forward:** We will employ iterative pruning as it results in the same amount of model reduction as one-shot pruning and yet the model enjoys greater preservation of its original prediction accuracies from being iteratively pruned. We attribute iterative pruning’s better performance to the fine-tuning of weights that goes in-between each pruning step. We believe that progressively pruning the model and finetuning it rather than pruning it in a single sitting gave the pruner a better assessment of the model weights since some of the weights would have been



affected by previous pruning steps (this element would be completely missed by a one-shot pruner).

```
Accuracy of the network on tested frames: 88.29 %  
Accuracy for class: Deepfakes is 97.11%  
Accuracy for class: Face2Face is 92.89%  
Accuracy for class: FaceShifter is 90.00%  
Accuracy for class: FaceSwap is 85.11%  
Accuracy for class: NeuralTextures is 88.44%  
Accuracy for class: youtube is 84.67%
```

Figure 6.2.4.2.1 – Lite0 Performance after 1-shot pruning (20%)

```
Accuracy of the network on tested frames: 90.00 %  
Accuracy for class: Deepfakes is 97.56%  
Accuracy for class: Face2Face is 95.78%  
Accuracy for class: FaceShifter is 92.22%  
Accuracy for class: FaceSwap is 90.22%  
Accuracy for class: NeuralTextures is 89.33%  
Accuracy for class: youtube is 85.47%
```

Figure 6.2.4.2.2 – Lite0 Performance after Iterative (5) pruning (20%)

#### **6.2.4.3 Finding the upper pruning ratio limit**

In general, increasing the pruning ratio will inevitably result in model performance degradation. In this experiment, we attempt to find the upper limits of our pruning ratio using our established pruning strategies (i.e. global & iterative pruning) from earlier experiment. We started from a

pruning ratio of 20% and incremented the pruning ratio by 5% at each step for as long as the pruned model maintains a prediction accuracy that hovers around 90%. We found that **30% was the furthest we can prune our Lite0 model before severe model performance degradation occurs.**

Prune ratio	Model Accuracy	Model Size	Model Parameters	Inference Speed (CPU)
20%	90%	10.33 MB	2,544,127	0.4509s
25%	89.68%	9.57 MB	2,339,132	0.4248s
30%	88.77%	8.81 MB	2,151,361	0.3935s
35%	78.61%	8.07 MB	1,968,970	0.3305s

Figure 6.2.4.3.1 – Lite0 performance after being pruned 20%, 25%, 30% and 35%

## 6.2.5 Knowledge Distillation

At this stage, we have already witnessed a significant model reduction. From B0 to 30% pruned Lite0, we see a reduction of inference time, model parameters and size by **half** while experiencing a model performance degradation of ~5% accuracy.

In our final section, we attempt to recover the lost accuracy by leveraging the use of Knowledge Distillation. Since our **B0** model is our most powerful model with a model accuracy sitting at ~94%, it will be our **teacher model** for the experiments in this section.

### 6.2.5.1 Model Training using Knowledge Distillation

This experiment involves training a Lite0 model using B0 as the teacher model. We also experiment with our alpha parameter in our knowledge distillation loss formulation as introduced earlier by controlling the weight of the soft label loss (the teacher's influence) in orders of 0%, 25%, 50%, 75% and 100%.

Alpha value	0	0.25	0.5	0.75	1
Accuracy (%)	87.68	91.44	90.16	92.24	90.03

Table 6.2.5.1.1 – Impact of Knowledge Distillation on Lite0 training (no fine-tuning)

As shown in the above results table, we observe model improvements when we incorporate knowledge distillation of a powerful teacher model during training, this is strongly suggested by the fact that any amount of knowledge distillation (alpha) increased the model performance after training. Moreover, we note that there are alpha ratios that seem to perform than others. For instance, a 25% to 75% soft-to-hard label loss and a 75% to 25% soft-to-hard label loss seem to achieve better performance than evenly mixed ratios (alpha = 50%) and dominating ratios (alpha = 0% or alpha = 100%). We also found that model training is typically more stable in these optimal ratios.

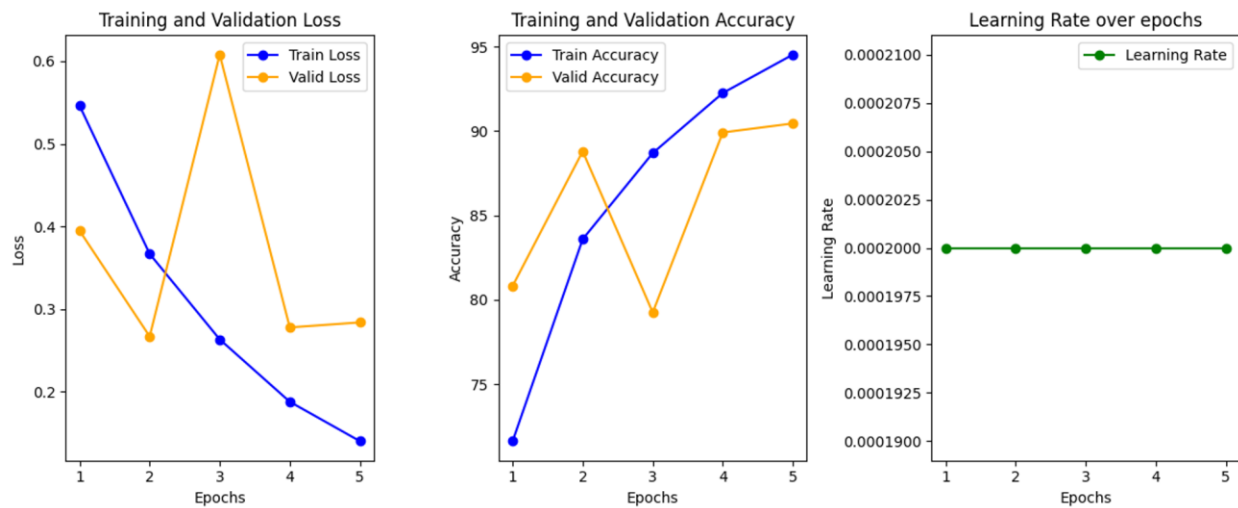


Figure 6.2.5.1.2 – Knowledge Distillation training plot at 0% alpha

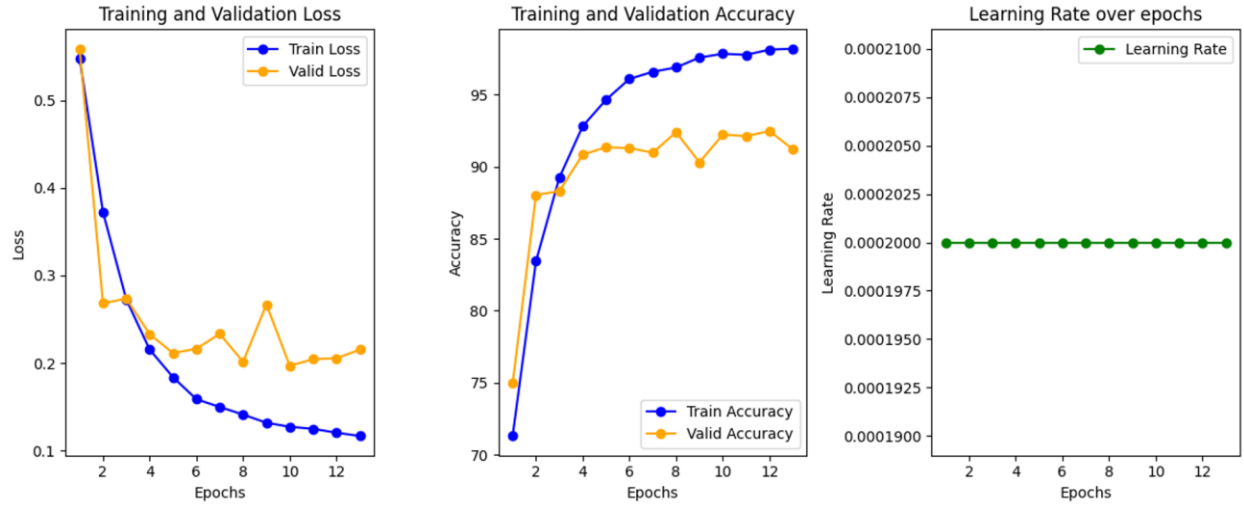


Figure 6.2.5.1.3 – Knowledge Distillation training plot at 25% alpha

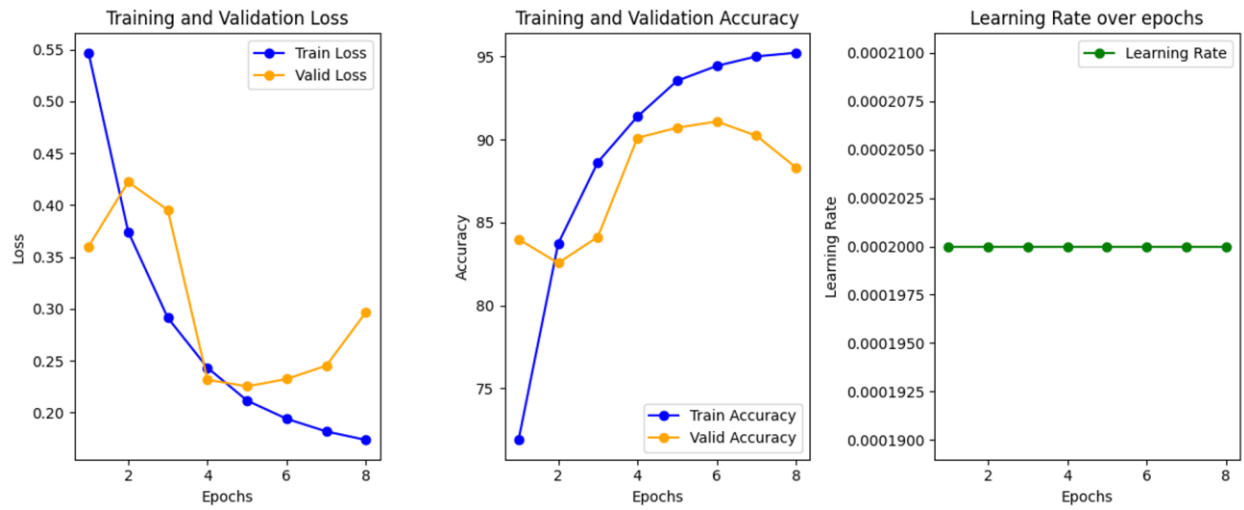


Figure 6.2.5.1.4 – Knowledge Distillation training plot at 50% alpha

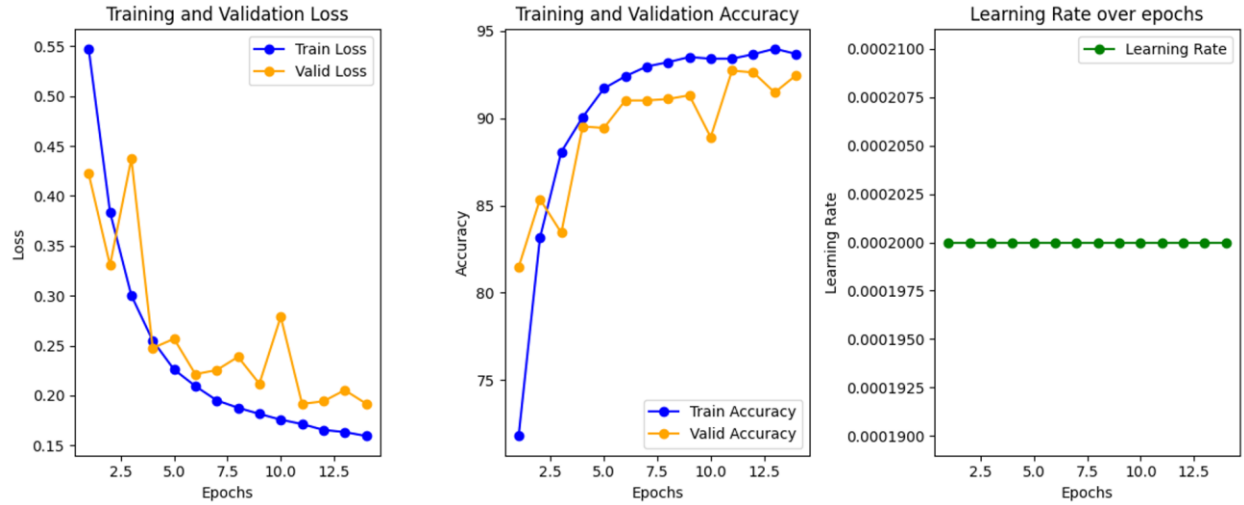


Figure 6.2.5.1.5 – Knowledge Distillation training plot at 75% alpha

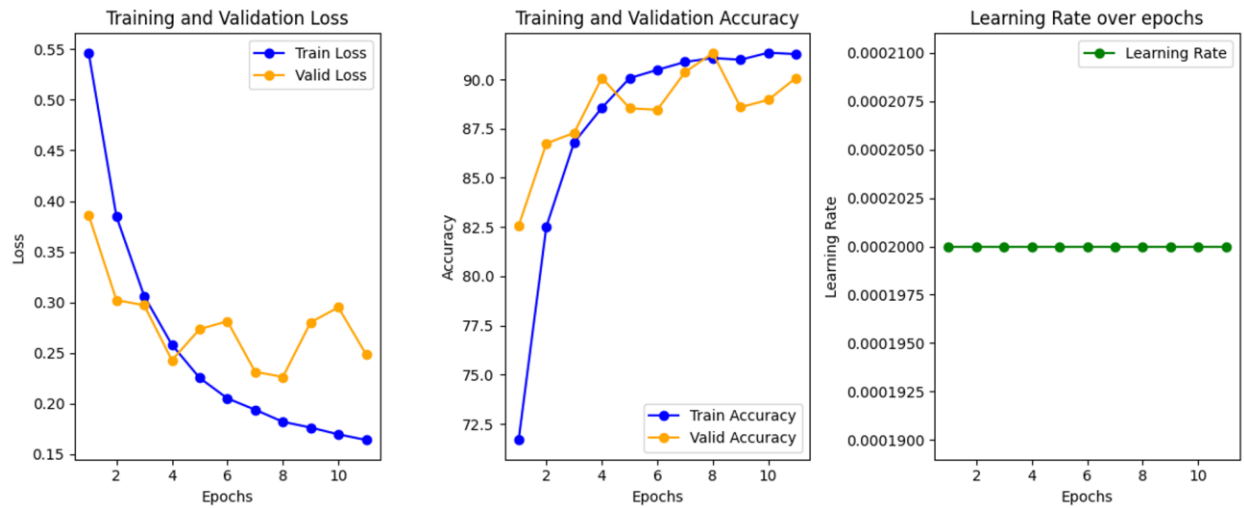


Figure 6.2.5.1.6 – Knowledge Distillation training plot at 100% alpha

We narrowed our alpha selection to a choice between 0.25 and 0.75. To better investigate which alpha parameter to select, we further finetuned the Lite0 models obtained using 25% and 75% distillation to obtain the following results:

```

Accuracy of the network on tested frames: 93.25 %
Accuracy for class: Deepfakes is 98.22%
Accuracy for class: Face2Face is 97.33%
Accuracy for class: FaceShifter is 97.11%
Accuracy for class: FaceSwap is 93.33%
Accuracy for class: NeuralTextures is 89.33%
Accuracy for class: youtube is 90.53%

```

Figure 6.2.5.1.7 – Lite0 performance (25% distillation + fine-tuned)

```

Accuracy of the network on tested frames: 92.91 %
Accuracy for class: Deepfakes is 98.00%
Accuracy for class: Face2Face is 97.78%
Accuracy for class: FaceShifter is 93.78%
Accuracy for class: FaceSwap is 95.11%
Accuracy for class: NeuralTextures is 89.33%
Accuracy for class: youtube is 90.07%

```

Figure 6.2.5.1.8 – Lite0 performance (75% distillation + fine-tuned)

EfficientNet-B0	EfficientNet-Lite0	Lite0 + 25% KD
94.12%	90.78%	<b>93.25%</b>

Table 6.2.5.1.9 – Accuracy comparison between B0, Lite0 and Lite0 w/ knowledge distillation

- (Note:) It is worth noting that the fine-tuning phase also includes the knowledge distillation at the respective alpha levels for both models.

After fine-tuning, knowledge distillation at alpha value of 25% achieves the best learning outcome for our given problem and teacher model. Notably, Lite0 performs considerably closer to B0 than when we trained it earlier without any knowledge distillation. Our findings suggest that the student models should ideally obtain a quarter of their “knowledge” from the B0 teacher model and the remaining 75% from the hard-labels or ground truths.

Amazingly, we also found that performing knowledge distillation removes the need to conduct our elaborate training plan as described earlier (i.e. linear warm-ups + cosine annealing) as we found that our models experience stable training and converged better simply using only our fine-tuning algorithm. This contrasts our previous experience where not using our training strategy led to early and bad model convergence, yielding sub-optimal results. In this way, knowledge distillation also improves training efficiency as an unintended but welcomed consequence.

#### ***6.2.5.2 Pruning with Knowledge Distillation***

So far, we have successfully improved upon the Lite0 model using Knowledge Distillation. In this experiment, we take that one-step further by performing the fine-tuning steps involved during iterative pruning using our 25% knowledge distillation learning method.

Prune ratio	Model Accuracy	Model Size	Model Parameters	Inference Speed (CPU)
25%	93.71%	9.5 MB	2,322,717	0.4175s
30%	93.07%	8.78 MB	2,144,299	0.3866s
<b>35%</b>	<b>91.89%</b>	<b>7.97 MB</b>	<b>1,942,167</b>	<b>0.3699s</b>

40%	78.53%	7.21 MB	1,754,377	0.2645s
-----	--------	---------	-----------	---------

Figure 6.2.5.2.1 – Lite0 performance after being pruned 25%, 30%, 35% and 40% w/ Knowledge Distillation Fine-tuning

From our knowledge distillation and pruning experimentation, not only has our knowledge distillation approach improved model accuracy, we also found that we could prune our model even more than before, reducing model parameters and size even further. 35% was the furthest pruning ratio we can go that still preserves our 90% accuracy requirement which exceeds the amount we could prune (30%) before without knowledge distilled pruning. This seems to suggest that our knowledge distillation approach has benefited the pruner and provided it with a better assessment of the weights in the network which enabled the more effective and aggressive pruning we observed in knowledge distilled pruning.

Moreover, compared to the B0 baseline model, our 35% knowledge distilled pruned Lite0 model achieves almost 92% accuracy, falling short of B0's accuracy only by ~2% but has less than half of B0's original model parameters, model size and CPU inference time.

## 6.2.6 Findings

In this section we consolidate our findings from the experiments earlier:

- First, we showed that transfer learning using ImageNet pre-trained weights stabilized training and improved model performance for the deepfake classification task
- Secondly, we showed that model scaling had little effect on model performance (for our current set-up)
- Thirdly, we demonstrated that reducing EfficientNet to EfficientNet-Lite's architecture netted a positive impact on model footprint without sacrificing too much performance.



- Fourthly, we demonstrated proper application of model pruning to drastically cut down model parameters and size (and inference time)
- Finally, we obtained incredible results with our application of Knowledge Distillation which allowed to us to prune our network further, and obtain better model performance at the same time

Model	Prune ratio	Accuracy	Size	Parameters	Inference time
(Baseline) B0	0%	94.12%	16.31 MB	4,008,829	0.7654s
Lite0	0%	90.78%	13.74 MB	3,372,289	0.7236s
Lite0	30%	88.77%	8.81 MB	2,151,361	0.3935s
Lite0 + KD	30%	93.07%	8.78 MB	2,144,299	0.3866s
<b>Lite0 + KD</b>	<b>35%</b>	<b>91.89%</b>	<b>7.97 MB</b>	<b>1,942,167</b>	<b>0.3699s</b>

Table 6.2.6.1 – consolidation of the best models obtained from each experiment step

```

Accuracy of the network on tested frames: 91.89 %
Accuracy for class: Deepfakes is 97.78%
Accuracy for class: Face2Face is 97.78%
Accuracy for class: FaceShifter is 92.22%
Accuracy for class: FaceSwap is 96.00%
Accuracy for class: NeuralTextures is 88.22%
Accuracy for class: youtube is 88.13%

```

Figure 6.2.6.2 – Model performance of the (best) 35% Lite0 model trained using Knowledge Distillation

# Chapter 7

## Web Demo

Up to this point, our model only performs deepfake detection at the image classification level. For solution completeness, we want to turn our trained model into a working application. To this end, we created a simple web demo to address our starting problem – deepfake detection at the video level.

Due to latency incurred from model inference and frame extraction, it is impractical to perform real-time deepfake detection with our current models and frame extraction methodology. Instead, we propose to handle the video detection problem following [2]’s approach, which is by preprocessing the video inputs.

First, we split a video input into its respective frames and extract faces using our face detection algorithm like Open CV’s Haar Cascade Classifier. We then feed the extracted faces as input into our deepfake classifier model and have it return its prediction results. Next, we apply a bounding box to surround the facial regions in any given frame and apply the respective predicted label to the bounding box. A red box would denote that a face found in the frame is a production of deepfake while a green box would denote that the face found in the frame is authentic. Finally, we stitch together all the augmented frames that now contain the prediction labels and boxes into a video. When played, the bounding boxes should follow the faces detected on screen and the correct label should be displayed. The screenshots below illustrate the workings of our proposed application.



Figure 7.1 – example of a “real” prediction



Figure 7.2 – example of a “deepfake” prediction

# Chapter 8

## Conclusion & Future Works

### 8.1 Conclusion

In this project, we broke down our approach to handling deepfake detection as a binary classification problem by detailing our implementation strategies from problem formulation and data processing all the way down to our employed model learning strategies. We also demonstrated the effectiveness of EfficientNet at the deepfake detection problem by achieving a high classification accuracy of 94% with the EfficientNet-B0 model on the FF++ dataset. We also recommended and found success in demonstrating the effectiveness of various proposed training, model reduction and performance enhancing techniques to improve the model's performance while reducing overall model footprint. We showed that implementing EfficientNet-Lite reduced model footprint and computation at a small sacrifice of accuracy. We also demonstrated that proper model pruning can significantly reduce model footprint at very little model performance costs. In particular, we demonstrated the effectiveness of Knowledge Distillation using our EfficientNet-B0 as our base model as the teacher to distill knowledge to our reduced models to recover accuracy and also discovered that knowledge distilled pruning allows us to prune even further by providing the pruner with better weight assessments for a more effective and aggressive prune. All in all, our project not only obtained a high deep fake detection accuracy, we also reduced the model size, number of parameters and inference time required by half using our proposed training, model reduction and performance enhancement strategies. Finally, for solution completeness, we also answered our

original video deepfake detection problem by creating a web demo that identifies deepfakes at the video level.

## **8.2 Future Works**

In this section we discuss potential future works that could progress the project even further.

### **8.2.1 Quantization**

We actually performed quantization as one of our project experiments and found that it resulted in model reduction of 3-4 times but at a significant cost of accuracy. Our challenge in finding success with implementing post-training static quantization and Quantization Aware Training (QAT) might be due to Pytorch's quantization support still being in the beta stage. We also note that Tensorflow's implementation of EfficientNet-Lite (our proposed model) is quantizable and we recommend this to be a good next step.

Nevertheless, quantization reduces model size considerably and is an important step in making models edge device friendly. This is a good direction for the project to continue going forward, we believe that quantization is the currently the major missing ingredient to deploying our proposed models on mobile/edge devices.

### **8.2.2 Expanding Dataset**

One limitation in our project is that we only used the FF++ dataset in our project, while we obtained good results within the dataset, it is worthwhile to also consider other deepfake benchmarks such as DFDC, WildDeepfakes and Celeb-DF, etc. Deepfake generation technology will only progressively get better, and hence it is paramount for deepfake detection research to keep up by

constantly expanding deepfake benchmarks and exploring techniques to help detection model generalize well on never-before-seen deepfake techniques.

### **8.2.3 EfficientNet V2 Findings**

Our project focuses on the implementation and use of EfficientNet but in 2021, [4] proposed EfficientNet Version 2. Even though EfficientNet V2 models are bigger in size than its EfficientNet cousin, the paper proposed certain improvements such as learning techniques like adaptive regularization (progressive learning) and new scaling methodology that should theoretically be applicable to the EfficientNet V1. These changes should be explored to further improve our EfficientNet implementation.

### **8.2.4 Deployment on Edge Devices**

In our project, we took steps to bring us closer to deploying deepfake detection on edge devices. A promising next step would be to quantize the model as recommended in 8.2.1 and deploying the models on mobile devices. This will serve to protect the users from the malpractice of deepfake technology while preserving their privacy as data would not have to leave their phone for the detection models to operate.

END

# Bibliography

- [1] A. Collings, “Accessibility of Deepfakes Accessibility of Deepfakes,” 2020. Available: <https://digitalcommons.odu.edu/cgi/viewcontent.cgi?article=1048&context=covacci-undergraduateresearch>
- [2] L. Deng, H. Suo, and D. Li, “Deepfake Video Detection Based on EfficientNet-V2 Network,” *Computational Intelligence and Neuroscience*, vol. 2022, pp. 1–13, Apr. 2022, doi: <https://doi.org/10.1155/2022/3441549>.
- [3] M. Tan and Q. V. Le, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks,” *arXiv.org*, 2019. <https://arxiv.org/abs/1905.11946>
- [4] M. Tan and Q. V. Le, “EfficientNetV2: Smaller Models and Faster Training,” *arXiv:2104.00298 [cs]*, Jun. 2021, Available: <https://arxiv.org/abs/2104.00298>
- [5] A. Qadir, R. Mahum, M. A. El-Meligy, A. E. Ragab, A. AlSalman, and H. Hassan, “An efficient deepfake video detection using robust deep learning,” *Heliyon*, p. e25757, Feb. 2024, doi: <https://doi.org/10.1016/j.heliyon.2024.e25757>.
- [6] Y. Liu *et al.*, “Sora: A Review on Background, Technology, Limitations, and Opportunities of Large Vision Models,” 2024. Available: <https://arxiv.org/pdf/2402.17177.pdf>
- [7] Y. Mirsky and W. Lee, “2020. e Creation and Detection of Deepfakes: A Survey,” *ACM Comput. Surv. 1, 1, Article*, vol. 1, 2020, Available: <https://arxiv.org/pdf/2004.11138.pdf>

- [8] A. Naitali, M. Ridouani, F. Salahdine, and N. Kaabouch, “Deepfake Attacks: Generation, Detection, Datasets, Challenges, and Research Directions,” *Computers*, vol. 12, no. 10, p. 216, Oct. 2023, doi: <https://doi.org/10.3390/computers12100216>.
- [9] Rangilyu, “Rangilyu/EfficientNet-Lite,” *GitHub*, Mar. 25, 2024. <https://github.com/Rangilyu/EfficientNet-Lite?tab=readme-ov-file> (accessed Mar. 26, 2024).
- [10] R. Liu, “Higher accuracy on vision models with EfficientNet-Lite,” 2020. <https://blog.tensorflow.org/2020/03/higher-accuracy-on-vision-models-with-efficientnet-lite.html>
- [11] M. Huh, P. Agrawal, and A. Efros, “What makes ImageNet good for transfer learning?,” Dec. 2016. Available: <https://arxiv.org/pdf/1608.08614.pdf>
- [12] S. Graphics, “Benchmark Results - FaceForensics Benchmark,” *kaldir.vc.in.tum.de*. [https://kaldir.vc.in.tum.de/faceforensics\\_benchmark/](https://kaldir.vc.in.tum.de/faceforensics_benchmark/) (accessed Mar. 26, 2024).
- [13] G. Fang, “VainF/Torch-Pruning,” *GitHub*, Feb. 03, 2024. <https://github.com/VainF/Torch-Pruning>
- [14] Tony-Y, “Tony-Y/pytorch\_warmup,” *GitHub*, Mar. 25, 2024. [https://github.com/Tony-Y/pytorch\\_warmup](https://github.com/Tony-Y/pytorch_warmup) (accessed Mar. 26, 2024).
- [15] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Nvidia, “Importance Estimation for Neural Network Pruning,” 2019. Available: [https://openaccess.thecvf.com/content\\_CVPR\\_2019/papers/Molchanov\\_Importance\\_Estimation\\_for\\_Neural\\_Network\\_Pruning\\_CVPR\\_2019\\_paper.pdf](https://openaccess.thecvf.com/content_CVPR_2019/papers/Molchanov_Importance_Estimation_for_Neural_Network_Pruning_CVPR_2019_paper.pdf)
- [16] Y. Aflalo *et al.*, “Knapsack Pruning with Inner Distillation,” 2020. Accessed: Mar. 26, 2024. [Online]. Available: <https://arxiv.org/pdf/2002.08258.pdf>



- [17] L. Cun, Y. Le Cun, J. Denker, and S. Solla, “Optimal Brain Damage,” 1989. Available: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=17c0a7de3c17d31f79589d245852b57d083d386e>
- [18] A. Persson, “EfficientNet from scratch in Pytorch,” *www.youtube.com*, Feb. 18, 2021. [https://youtu.be/fR\\_0o25kigM?si=fixlSEZQRzOilWkX](https://youtu.be/fR_0o25kigM?si=fixlSEZQRzOilWkX) (accessed Mar. 26, 2024).
- [19] “Compound scaling differs from paper · Issue #40 · rwightman/gen-efficientnet-pytorch,” *GitHub*. <https://github.com/rwightman/gen-efficientnet-pytorch/issues/40> (accessed Mar. 26, 2024).
- [20] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Weinberger, “Deep Networks with Stochastic Depth,” *arXiv:1603.09382 [cs]*, Jul. 2016, Available: <https://arxiv.org/abs/1603.09382>
- [21] Thing, “Deepfake Detection with Deep Learning: Convolutional Neural Networks versus Transformers,” Apr. 2023, doi: <https://doi.org/10.48550/arxiv.2304.03698>.
- [22] Z. Akhtar, “Deepfakes Generation and Detection: A Short Survey,” *Journal of Imaging*, vol. 9, no. 1, p. 18, Jan. 2023, doi: <https://doi.org/10.3390/jimaging9010018>.
- [23] R. Tolosana, R. Vera-Rodriguez, J. Fierrez, A. Morales, and J. Ortega-Garcia, “Deepfakes and beyond: A Survey of face manipulation and fake detection,” *Information Fusion*, vol. 64, pp. 131–148, Dec. 2020, doi: <https://doi.org/10.1016/j.inffus.2020.06.014>.
- [24] J. Frankle and M. Carbin, “The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks,” *arXiv:1803.03635 [cs]*, Mar. 2019, Available: <https://arxiv.org/abs/1803.03635>

- [25] “Pruning Tutorial — PyTorch Tutorials 2.2.1+cu121 documentation,” *pytorch.org*. [https://pytorch.org/tutorials/intermediate/pruning\\_tutorial.html#global-pruning](https://pytorch.org/tutorials/intermediate/pruning_tutorial.html#global-pruning) (accessed Mar. 26, 2024).
- [26] A. Chariton, “Knowledge Distillation Tutorial — PyTorch Tutorials 2.2.1+cu121 documentation,” *pytorch.org*. [https://pytorch.org/tutorials/beginner/knowledge\\_distillation\\_tutorial.html](https://pytorch.org/tutorials/beginner/knowledge_distillation_tutorial.html)
- [27] R. Krishnamoorthi, “(beta) Static Quantization with Eager Mode in PyTorch — PyTorch Tutorials 2.2.1+cu121 documentation,” *pytorch.org*. [https://pytorch.org/tutorials/advanced/static\\_quantization\\_tutorial.html](https://pytorch.org/tutorials/advanced/static_quantization_tutorial.html) (accessed Mar. 26, 2024).
- [28] Daniel Mas Montserrat *et al.*, “Deepfakes Detection with Automatic Face Weighting,” Apr. 2020, doi: <https://doi.org/10.48550/arxiv.2004.12027>.
- [29] Z. Sun, Y. Han, Z. Hua, N. Ruan, and W. Jia, “Improving the Efficiency and Robustness of Deepfakes Detection through Precise Geometric Features,” *arXiv.org*, Apr. 09, 2021. <https://arxiv.org/abs/2104.04480> (accessed Mar. 26, 2024).
- [30] N. Bonettini, E. D. Cannas, S. Mandelli, L. Bondi, P. Bestagini, and S. Tubaro, “Video Face Manipulation Detection Through Ensemble of CNNs,” *arXiv:2004.07676 [cs, eess]*, Apr. 2020, Available: <https://arxiv.org/abs/2004.07676>
- [31] D. Masters, A. Labatie, Z. Eaton-Rosen, and C. Luschi, “Making EfficientNet More Efficient: Exploring Batch-Independent Normalization, Group Convolutions and Reduced Resolution Training,” *arXiv.org*, Aug. 26, 2021. <https://arxiv.org/abs/2106.03640>

- [32] P. Kumar *et al.*, “MobileOne: An Improved One millisecond Mobile Backbone,” Mar. 2023. Accessed: Mar. 26, 2024. [Online]. Available: <https://arxiv.org/pdf/2206.04040.pdf>
- [33] A. Polino, R. Pascanu, and D. Alistarh, “Model compression via distillation and quantization,” *arXiv.org*, Feb. 15, 2018. <https://arxiv.org/abs/1802.05668>
- [34] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, “Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures,” *arXiv:1607.03250 [cs]*, Jul. 2016, Available: <https://arxiv.org/abs/1607.03250>
- [35] A. Kuzmin, M. Nagel, Mart van Baalen, Arash Behboodi, and Tijmen Blankevoort, “Pruning vs Quantization: Which is Better?,” *arXiv (Cornell University)*, Jul. 2023, doi: <https://doi.org/10.48550/arxiv.2307.02973>.
- [36] J. Kim, Yash Bhalgat, J. Lee, C. J. Patel, and N. Kwak, “QKD: Quantization-aware Knowledge Distillation,” *arXiv (Cornell University)*, Nov. 2019, doi: <https://doi.org/10.48550/arxiv.1911.12491>.
- [37] X. Zhang, X. Zhou, M. Lin, and J. Sun, “ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices,” *arXiv:1707.01083 [cs]*, Dec. 2017, Available: <https://arxiv.org/abs/1707.01083>
- [38] C. Shorten and T. M. Khoshgoftaar, “A survey on Image Data Augmentation for Deep Learning,” *Journal of Big Data*, vol. 6, no. 1, Jul. 2019, doi: <https://doi.org/10.1186/s40537-019-0197-0>.

- [39] K. Han, Y. Wang, Q. Zhang, W. Zhang, C. Xu, and T. Zhang, “Model Rubik’s Cube: Twisting Resolution, Depth and Width for TinyNets,” *arXiv.org*, Dec. 24, 2020. <https://arxiv.org/abs/2010.14819> (accessed Mar. 26, 2024).
- [40] G. Menghani, “Efficient Deep Learning: A Survey on Making Deep Learning Models Smaller, Faster, and Better,” *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–37, Mar. 2023, doi: <https://doi.org/10.1145/3578938>.
- [41] B. R. Bartoldson, B. Kailkhura, and D. Blalock, “Compute-Efficient Deep Learning: Algorithmic Trends and Opportunities,” *arXiv.org*, Mar. 21, 2023. <https://arxiv.org/abs/2210.06640> (accessed Mar. 26, 2024).
- [42] Y. Wang, Y. Han, C. Wang, S. Song, Q. Tian, and G. Huang, “Computation-efficient Deep Learning for Computer Vision: A Survey,” Aug. 2023.
- [43] A. Rössler, D. Cozzolino, L. Verdoliva, C. Riess, J. Thies, and M. Nießner, “FaceForensics++: Learning to Detect Manipulated Facial Images,” Aug. 2019.
- [44] B. Dolhansky *et al.*, “The Deepfake Detection Challenge (DFDC) Preview Dataset,” Oct. 2019.