INVESTIGATING THE IMPACT OF INDIVIDUAL DATA AUGMENTATION TECHNIQUES ON SKIN LESION CLASSIFICATION

A REPORT SUBMITTED TO MANCHESTER METROPOLITAN UNIVERSITY
FOR THE DEGREE OF BACHELOR OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2024

By
William Husband
Department of Computing and Mathematics

Contents

List of Tables

List of Figures

Abstract

Skin cancer is a serious disease which is responsible for numerous deaths every year, however early detection of the disease can significantly increase the chances of survival. Detecting the disease proves a challenge even for dermatologists. However, in recent years the interest in deep learning methods like Convolutional Neural Networks (CNNs) has risen due to the potential they have shown to effectively classify skin cancer. CNNs require a large amount of training data to be effective, however, there is a lack of publicly available skin lesion data. Data augmentation methods allow data to be synthetically created out of the existing available data, which has been shown to improve the performance of a CNN. There are several methods of data augmentation, which are usually applied simultaneously. There is a prevalent lack of a general strategy for selecting which data augmentation methods to apply together however, which this study aims to remedy. By analysing the impact that several individual data augmentation methods have on commonly used CNN models for skin lesion classification, this study aims to create a reference point that assists in future studies that wish to apply multiple data augmentation methods simultaneously by showing the potential of different data augmentation methods. Due to resource constraints this study did not fully achieve its goal, however given more of these resources the study could easily be re-run to create the reference point that was intended to be created in this study.

Declaration

No part of this project has been submitted in support of an application for any other degree or qualification at this or any other institute of learning. Apart from those parts of the project containing citations to the work of others, this project is my own unaided work. This work has been carried out in accordance with the Manchester Metropolitan University research ethics procedures, and has received ethical approval number 64156.

Signed:

Date:

Acknowledgements

I would like to thank Ismail Adeniran, John Darby, Adrian Davison, Li Guo and Wenqi Lu for their support throughout this project and making everything possible. I would like to thank Dr Adrian Davison in particular for allowing me to use his GPU to run the final version of my code, which assisted this study significantly.

Abbreviations

PCA	Principal Components Analysis
LTA	Three Letter Acronym

Chapter 1

Introduction

Skin cancer refers to a group of diseases that cause skin cells to grow uncontrolled and form tumours (Mazhar et al. 2023). Types of skin cancer can be split into two categories; Melanoma and Non-Melanoma skin cancer. Non-Melanoma skin cancers include Basal Cell Carcinoma (BCC) and Squamas Cell Carcinoma (SCC), which are responsible for 80% and 16% of all cases respectively and are relatively harmless if detected and treated early (Debelee 2023). Melanoma skin cancer on the other hand only makes up around 4% of cases, but is responsible for more than 20,000 deaths per year in Europe alone (Mazhar et al. 2023). Melanoma leads to death in 80% of cases, however if detected and treated early it can be cured in 95% of the time which significantly improves the chance of survival (Debelee 2023).

Early detection is a large factor in the survival rate of skin cancer (Brinker et al. 2018). However, distinguishing the nature of skin lesions is a challenging task even for experienced dermatologists (Höhn et al. 2021); dermatologists have been shown to have an accuracy rate of 75% to 84% when detecting Melanoma (Kumari and Sharma 2021). In recent years, the use of deep learning methods like Convolutional Neural Networks (CNNs) for medical image classification tasks, including skin lesion classification, have shown comparable, if not better results that dermatologists of 72% to 94% accuracy (Kumari and Sharma 2021). CNNs learn hierarchical features of images, from low level patterns like edges and textures, to higher level patterns like objects and scene compositions (Yamashita et al. 2018). However, CNNs require significant amounts of data to learn these patterns effectively (Fenza et al. 2021), and unfortunately there is limited publicly available skin lesion data. Of the data there is available, there is often significant class imbalance or limited sample size, which makes it hard for CNNs to generalise across diverse nation populations and different manifestations

of skin lesions (Wu et al. 2022).

Data augmentation arises as a potential remedy for the prevalent lack of skin lesion data. Data augmentation is the process of artificially increasing the quantity of samples in a dataset by creating new data samples from the original dataset (Awan, Abid Ali 2022). There are several methods of data augmentation, the methods investigated in this study can be categorised as either basic data augmentation or generative data augmentation. Basic data augmentations are those which apply a simple transformation to the data, creating modified copies of samples to add back to the original dataset. Generative augmentation however creates entirely new samples to add to the dataset by training and generating images with a Generative Adversarial Network (GAN). Data augmentation has shown incredible potential for skin lesion classification, improving the ability of a CNN to differentiate between different types of skin lesion and improve the its performance (Perez et al. 2018, Hamida et al. 2023).

In order to assess the performance of a classifier it is essential to monitor several metrics, not just the overall accuracy. Accuracy is the measure of how many predictions made by the classifier are correct, and is a useful metric to show an overview of model performance. However, there are scenarios where a classifier's accuracy can be high, but is still producing high amounts of false negatives. In the context of skin lesion classification, the ramifications that false negatives have can be much greater than those of false positives. It is therefore essential to utilise several metrics like precision, recall and f1 score to assess the performance of a classifier.

Generally, studies will utilise multiple different data augmentation techniques simultaneously in order to improve a model's performance (Hamida et al. 2023, Bozkurt 2023). However, there is a prevalent lack of a general strategies for selecting augmentation techniques, even though choosing appropriate data augmentation techniques can be more effective for performance than model choice (Shijie et al. 2017).

The aim of this study is to show the impact that individual data augmentation methods have on the effectiveness of a CNN when classifying skin lesions, the purpose of this is to provide the foundation of a strategy for selecting data augmentation methods to apply simultaneously in the context of skin lesion classification.

This study will measure the different ways that data augmentations affect a model's accuracy, precision, recall and f1 score to give a clear understanding of the merits and faults of each method. The result of this study is to provide a reference point of the advantages and disadvantages of different data augmentations when applied to skin lesion classification. The results of this study will act as a guide for future studies

that wish to implement multiple data augmentations to skin lesion datasets, providing guidance for selecting augmentation methods to include in an assembly. The findings of this study will be presented to show the exact impact that each data augmentation method has had on both a ResNet-50 model and a DenseNet-121 model, remedying the lack of a general method for data augmentation selection for future studies.

By using a dataset and CNN models that are already commonplace within the field of skin lesion classification, this information aims to be applicable to future studies in the field. To achieve this, a ResNet-50 and a DenseNet-121 models that are pretrained on the ImageNet dataset are employed. These models are common in the field of skin lesion classification (Rashid, Tanveer, and H. A. Khan 2019, M. A. Khan et al. 2019, Mohamed and El-Behaidy 2019). The dataset being used is the ISIC 2019 datset (Codella, Gutman, et al. 2017, Tschandl, Rosendahl, and Kittler 2018, Combalia et al. 2019), which in itself is commonly used in this field, and it encompasses another commonly used dataset; the HAM10000 dataset (Cassidy et al. 2022). The models will be trained, tested and reset multiple times, the first time they are trained and tested the models will not utilise any data augmentation on the training data, this is to give a baseline of the model performance. After that, each time they are trained a different data augmentation will be applied to the training data. Each time the models are trained and tested with a data augmentation method, the accuracy, precision, recall and F1 score will be calculated. After the models have been trained and tested with each augmentation, the results will be compared to the models baseline performance in order to show the direct impact that each individual method of data augmentation has on a CNN to be able to classify skin lesions.

Chapter 2

Literature Review

2.1 Metrics

Several metrics must be observed to measure the effectiveness of a classifier. The most common metric is accuracy. Accuracy measures the number of correct predictions by the classifier divided by the total predictions made (equation 2.1), providing a straightforward indication of a classifier's performance. However, alone, it is not a reliable metric when testing using an imbalanced dataset. For example, a dataset with 100 data, with 90 in the negative class and 10 in the positive class, would show a 90% accuracy if the classifier predicted all data as negative. This is a high score for accuracy but does not reflect the condition of the classifier as it has not predicted any data of the positive class correctly (Wardhani et al. 2019).

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.1}$$

A confusion matrix is a way of recording results from a classifier. It is a table which plots the classes of the real data against the classes predicted by the classifier. Figure 2.1 shows a confusion matrix for a binary classification problem.

From the confusion matrix, precision and recall can be derived. Precision (equation 2.2) measures the number of true positive elements divided by the number of total positive predictions made by the classifier; true positive (TP) elements being those that are predicted to be positive by the classifier and are actually positive. False positive elements are those which are predicted to be positive by the classifier but are not positive. Recall (equation 2.3) measures the percentage of the positive data which has been correctly predicted by the classifier; the true positives made by the classifier divided

		Predicted		
	classes	Positive (1)	Negative (0)	*total*
Actuals	Positive (1)	TP = 10	FN = 15	25
	Negative (0)	FP = 5	TN = 70	75
	total	15	85	100

Figure 2.1: Confusion matrix for positive and negative classification

by the total number of positives in the testing set. True positives and negatives are important in the context of skin lesion classification as a false negative (FN), where the classifier predicts an image to be negative when it is in fact positive, could lead to delayed diagnosis.

$$Precision = \frac{TP}{TP + FP} \tag{2.2}$$

$$Recall = \frac{TP}{TP + FN} \tag{2.3}$$

F1 score (equation 2.4) is a metric which aggregates precision and recall under the concept of harmonic mean. If either the precision or recall score are close to 0, the F1 score will take a suffer a large drop, especially since the harmonic mean will give more weight to lower values (Grandini, Bagli, and Visani 2020).

$$F1 = \frac{2}{Precision^{-}1 + Recall^{-}1} = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{2.4}$$

These metrics give more insight into the robustness of a classifier, as precision shows how trustworthy a positive prediction from the classifier is, recall shows how well a classifier can find all the positive elements in a dataset, and the F1 score combines these scores into a single metric. The importance of these metrics as well as accuracy can be seen in relation to figure 2.1. The accuracy for this confusion matrix would be 80%. However, the precision and recall scores are 0.67 and 0.4 respectively, resulting in a low F1 score of 0.5. This means that despite having a respectable accuracy of 80 %, the classifier can be shown to struggle to detect the positive class by using other metrics.

2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of deep learning model inspired by the animal visual cortex that is used for analysing grid-based data, like images (Hubel and Wiesel 1968). Traditional methods for analysing images involved manual feature extraction, whereas CNNs remove this need by automating the process of extracting features from images (Li et al. 2021). CNNs generally consist of three types of layers: convolutional layers, pooling layers, and fully connected layers. Convolutional and pooling layers aim to extract features from images, and fully connected layers aim to transform the extracted features into an output; such as predicting an input's class (O'shea and Nash 2015). Figure 2.2 shows an abstraction of a generic CNN's layers. CNNs learn features hierarchically from low level patterns like edges and textures, to high level patterns like objects and scene compositions. Through back-propagation and gradient descent, parameters are automatically updated and optimised so that the difference between the CNN's prediction of the data's category and the ground truth of the data can be minimised (Yamashita et al. 2018).

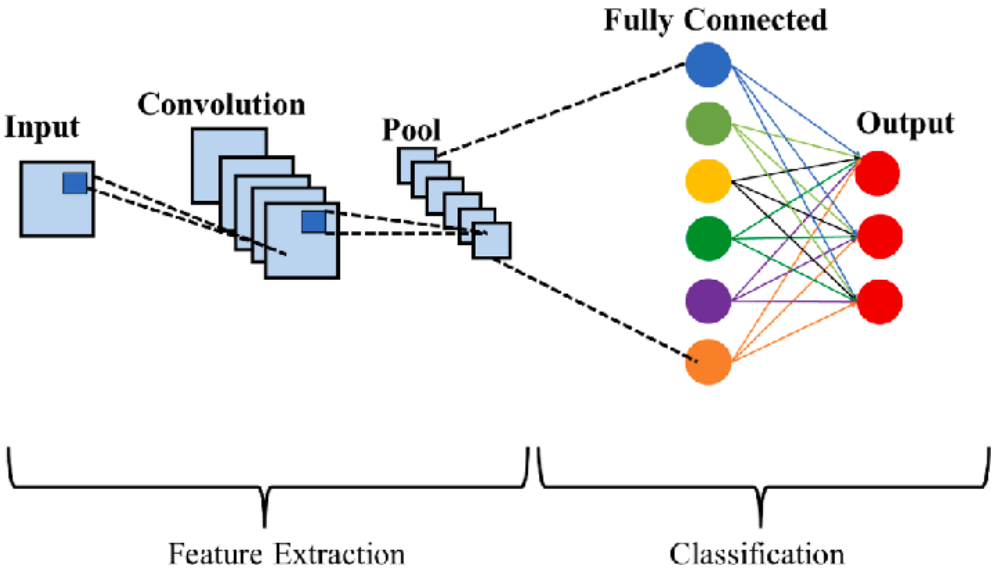

Figure 2.2: Diagram of a generic CNN's layers (Hadi et al. 2023)

CNNs have shown they have the capability to perform better than humans at image recognition tasks. In 2012, Krizhevsky, Sutskever, and Hinton 2012 achieved the best classification result in the ImageNet Large Scale Visual Recognition Competition (ILSVRC) by using a CNN. Breakthroughs like this have led to widespread interest in applying CNNs in the medical field (Chlap et al. 2021). In 2016, Nasr-Esfahani et al.

2016 trained a two-layer CNN using 136 images taken from the Department of Dermatology of the University Medical Center Groningen's public image archive. It was tested using 34 images, producing an 81% accuracy result and an 81% recall result. However, the results of this test are unreliable due to the limited size of the dataset used.

CNNs require a lot of training data to be effective (Sahu et al. 2020), and there is a prevalent lack of available data for skin lesion classification (Chlap et al. 2021). One method that has been used to remedy the lack of available data is transfer learning; pre-training a CNN using the large, publicly available imageNet dataset (*ImageNet* n.d.), then fine tuning the parameters to classify skin lesions (Brinker et al. 2018).

Utilising transfer learning with imageNet, Sun et al. 2016 trained a CaffeNet and a VGGNet CNN with a further 6584 images from dermQuest in 2016. This produced an average of 50.27% accuracy across 198 classes. In 2017, Lopez et al. 2017 achieved an accuracy of 81.33% when testing a VGGNet which utilised transfer learning, and fine tuned for skin lesion classification. They trained the VGGNet with images from ISIC (*ISIC Archive* n.d.), and tested with 379 images from the 2016 challenge dataset from ISBI (IEEE International Symposium on Biomedical Imaging (ISBI) n.d.). The dataset used by Lopez et al. 2017 to train the model was also enriched with transformed copies of itself, like rotating or horizontally flipped copies. This is an example of basic data augmentation.

2.3 Data Augmentation

Data augmentation is another method to mitigate the issue of limited publicly available skin lesion datasets. Data augmentation encompasses a range of strategies which can increase the size and quality of datasets (Shorten and Khoshgoftaar 2019). Using data augmentation has been shown to improve the ability of deep learning models to generalise, reducing the extent to which models over-fit to training data (C. Zhang et al. 2021).

2.3.1 Basic Data Augmentation

Basic data augmentation techniques include methods which apply simple transformations to image data. This involves either mapping the points of an image to a different position, for example flipping or rotating an image, or manipulating image intensity

values, like changing the brightness of an image (Shorten and Khoshgoftaar 2019). Images are then added back to dataset after being modified, along with the original data, increasing the overall size of the dataset (Chlap et al. 2021). Figure 2.3 shows examples of basic data augmentation techniques.

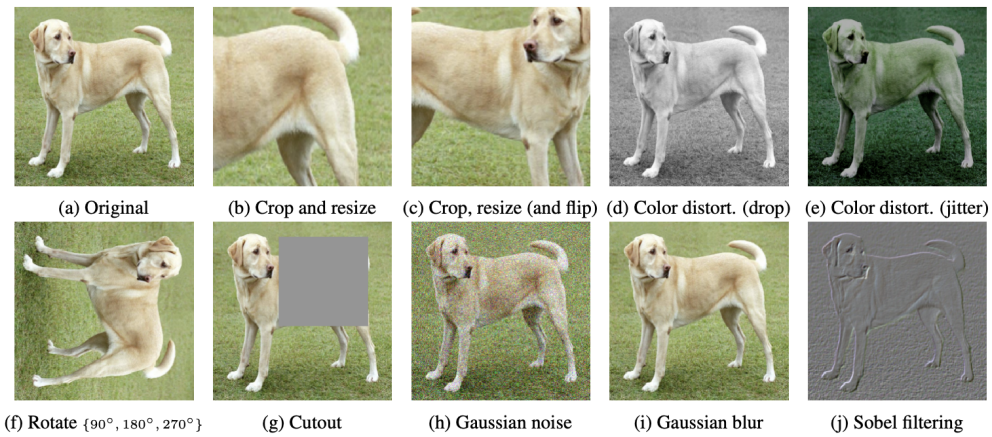

Figure 2.3: Examples of basic data augmentations (Chen et al. 2020)

2.3.2 Generative Data Augmentation

Generative data augmentation refers to techniques which create entirely new artificial images to be added to the original dataset (Saturn Cloud 2024). This is done using Generative Adversarial Networks (GANs). The fundamental parts of a GAN are two neural networks; a generator and a discriminator. The generator aims to create new instances of data that resemble the real data. The discriminator aims to distinguish between real data from the training set and artificial data created by the generator. Both neural networks are trained simultaneously. The discriminator is trained by being shown real data or fake data, predicting if the data is real or fake, then updating its weights depending on it's accuracy. The generator is trained by taking feedback from the discriminator. This process continues until the discriminator is unable to differentiate between real and generated data (Goodfellow et al. 2020). Figure 2.4 visualises the architecture of a GAN.

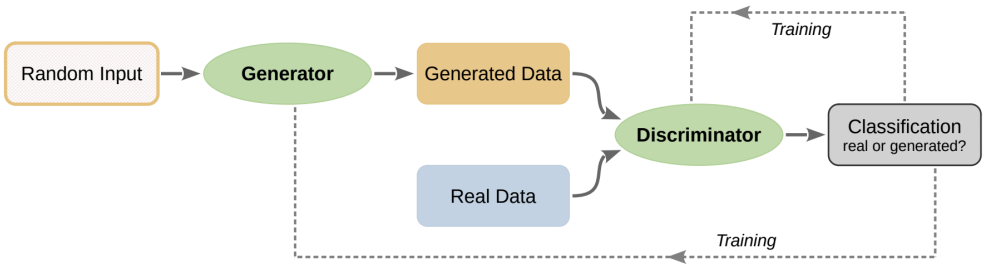

Figure 2.4: Architecture of a GAN (Abedi et al. 2022)

2.3.3 Studies on the Impact of Data Augmentation

Perez et al. 2018 conducted an in-depth analysis of basic data augmentation methods with a focus on skin lesions, employing an Inception-V4, ResNet-152, and DenseNet-161 model to assess their effectiveness. They used the ISIC 2017 Challenge dataset (Codella, Gutman, et al. 2017) and trained the models using an initial learning rate of 0.001, a reduction in learning rate by a factor of 0.1 after 10 epochs and employing Stochastic Gradient Descent with a momentum factor of 0.9 and a batch size of 32. Their study evaluates each basic augmentation method individually across different models, which shows the exact impact that each method has. They visualised their findings on a graph, showing an approximate 0.02 increase in Area Under the Curve (AUC) when basic augmentations were applied. Their study focuses on AUC as the primary metric, which has been shown to be unreliable (Lobo, Jiménez-Valverde, and Real 2008), overlooking other metrics like precision, recall, and F1-score. Also, the study does not consider the potential of generative data augmentation methods. However, this study does show that applying most basic data augmentation methods to a skin lesion dataset improves a classifier's performance.

The study titled "Data Balancing through Data Augmentation to Improve Transfer Learning Performance for Skin Disease Prediction" by Hamida et al. 2023 also investigates basic augmentation techniques, but unlike Perez et al. 2018, utilises a wide array of metrics. Training and testing the EfficientNet-V2 model with the HAM10000 dataset (Tschandl, Rosendahl, and Kittler 2018), the study compares the effectiveness of the model without data augmentation against its performance when several basic augmentation methods are applied simultaneously. The study found significant improvements in accuracy, precision, and recall when data augmentation is applied. Although, this study lacks granularity in its analysis. This study does not provide insights into how each individual augmentation method contributes to these enhancements in

model performance, missing opportunities for more detailed understanding of the specific effect of certain basic data augmentation methods when applied to skin lesion classification.

Research conducted by Rashid, Tanveer, and H. A. Khan 2019 focuses on the effect of using Generative Adversarial Networks (GANs) for data augmentation. Their study evaluates a GAN-based CNN against a ResNet-50 and a DenseNet model, utilizing the ISIC 2018 Challenge dataset (Codella, Rotemberg, et al. 2018, Tschandl, Rosendahl, and Kittler 2018). They used an initial learning rate (LR) of 0.0001 across a total of 24 epochs. The results show that the accuracy achieved with the GAN model surpasses that of the ResNet-50 and DenseNet models. While this study utilises several metrics, it limits its scope by only comparing the performance of the GAN-based CNN to these traditional networks. The study does not explore the potential of GANs in a ”plug and play” manner, as demonstrated in a study by Shen et al. 2022, where GAN augmented images are tested on an existing network. Showing the difference in effectiveness of a single model trained with and without GAN augmented images. This would show an evaluation of the direct impact of GAN augmented data by applying it to a constant model framework, keeping the model as a control variable, isolating the effect of the augmentation technology on performance outcomes (Bhandari 2021).

A study by Bozkurt 2023 extends its research by employing both basic and GAN augmentation methods on the Inception-ResNet-V2 model, using the HAM10000 dataset (Tschandl, Rosendahl, and Kittler 2018). This study records a range of metrics; accuracy, precision, recall, and F1-score, which all show significant improvements when data augmentation is applied. Specifically, accuracy increased from 83.59% with no augmentation to 95.09% with augmentation, precision from 0.8117 to 0.95049, recall from 0.6714 to 0.9516, and F1-score from 0.7231 to 0.9527. However, in this study the augmentation of data is applied before the test-train split, leading to potential data leakage, which is where information is accidentally shared between testing and training sets. Data augmentation involves making altered copies of the existing data. This means that if the augmentation is applied before the data split, then the training set and testing set could each have an altered copy of the same data, which means that information has been accidentally shared between the sets. A risk of data leakage is overly optimistic results, which may not transfer to a real-world application (Yagis et al. 2021). The study conducted 100 epochs with an initial learning rate of 0.0001, using Stochastic Gradient Descent with a momentum factor of 0.9 and a batch size of 128. Their study, similar to the one conducted by Hamida et al. 2023, also measures

the impact of applying multiple augmentation methods at the same time, rather than evaluating the influence of each specific method, which could provide more granular insights into their individual effectiveness.

2.4 Ethical Considerations

When conducting a study on the impact of data augmentation for skin lesion classification, there are several considerations that must be made about the data being used and the implications of the study. First of all, images of skin lesions can be considered to be personal information. This means that the data must be used in ways which adhere to privacy regulations of data, like the General Data Protection Regulation (GDPR) (European Parliament and Council of the European Union 2016). All data must be anonymised such that patients can not be identified from the images or any metadata. The source of the data should also be considered and if the data was collected with consent, however consent may not be necessary since the data is anonymised. Datasets from ISIC already adhere to these guidelines, however it is important to stay vigilant about the implications of using skin lesion data, as it is still sensitive data. Due to this data sensitivity, data and anything else related to this study will be stored securely on cloud based platforms, OneDrive and Google Drive, in order to prevent any mishandling of data.

The implications of the findings of a study in this field must also be considered. As mentioned in the introduction, the implications of diagnosing skin cancer are serious. Any findings from this study will not be made public instantly, as they must be verified meticulously in order to be sure they are reliable and can be used in other studies. Ramifications of false or misleading information in this field are huge and could even lead to mortality.

Chapter 3

Design

Different methods of data augmentation will have differing impacts on the effectiveness of a CNN to classify skin lesions. Also, different methods of data augmentation methods are often applied simultaneously, like in the studies by Hamida et al. 2023, Bozkurt 2023. Each one of these data augmentation methods will influence the performance of the classifier differently, and understanding the individual influences of different augmentation methods would form the foundation of a strategy when selecting data augmentation methods to apply to a classifier for skin lesion analysis.

The goal of this study is to show these individual impacts of different data augmentation methods to provide this much needed strategy for future studies. In order to account for differing methodologies used for skin lesion classification with CNNs, the dataset and CNN architectures used in this study will reflect those which are commonplace in this field. Different CNN architectures react differently to data augmentations (Bozkurt 2023). By employing CNN architectures that are commonly used for skin lesion classification, and a dataset that represents commonly used datasets, this study's findings will be more applicable to future studies.

3.1 Dataset

The dataset used in this study is the ISIC 2019 challenge dataset (Codella, Gutman, et al. 2017, Tschandl, Rosendahl, and Kittler 2018, Combalia et al. 2019). It is an amalgamation of the HAM10000 dataset and the BCN20000 dataset (Kassem, Hosny, and Fouad 2020). This means that the ISIC 2019 challenge dataset is a suitable choice if the goal is to use commonly used data, as the HAM10000 times is frequently used; such as in studies conducted by Le et al. 2020, Ratul et al. 2019 and Almaraz-Damian

et al. 2020, and in the aforementioned studies by Hamida et al. 2023 and Bozkurt 2023. Plus, the ISIC 2019 challenge dataset as a whole has been used several times in studies such as those conducted by Kassem, Hosny, and Fouad 2020 and Villa-Pulgarin et al. 2022. The ground truth for the ISIC 2019 challenge dataset is only available for the training dataset, so this study only uses this section of the dataset. The dataset used in this study contains 25,281 images, each representing one of eight classes of skin lesion; Actinic Keratosis (AK), Basal Cell Carcinoma (BCC), Benign Keratosis (BKL), Dermatofibroma (DF) Melanoma (MEL), Melanocytic Nevus (NV), Squamous Cell Carcinoma (SCC) and Vascular Lesion (VASC), examples of which are shown in Figure 3.1. The class distribution can be seen in Figure 3.2. Of this, 70% will be used for training the models, with 20% used for testing and the remaining 10% for validation (Figure 3.3), with the class balance being maintained by using a stratified split.

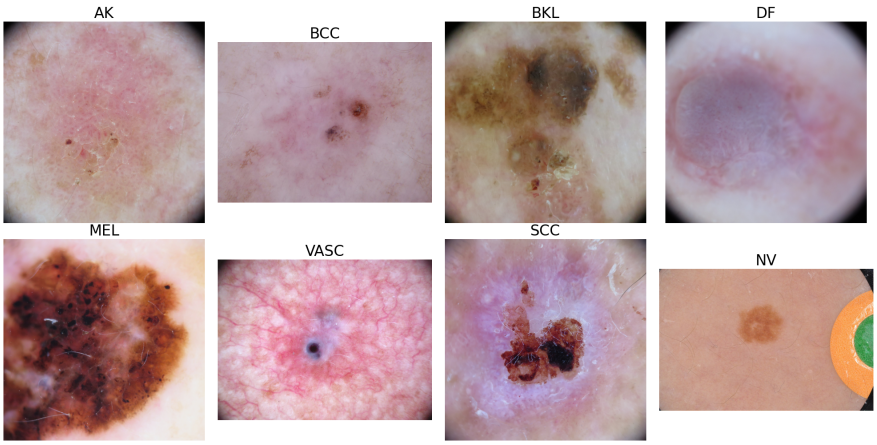

Figure 3.1: Examples of different classes

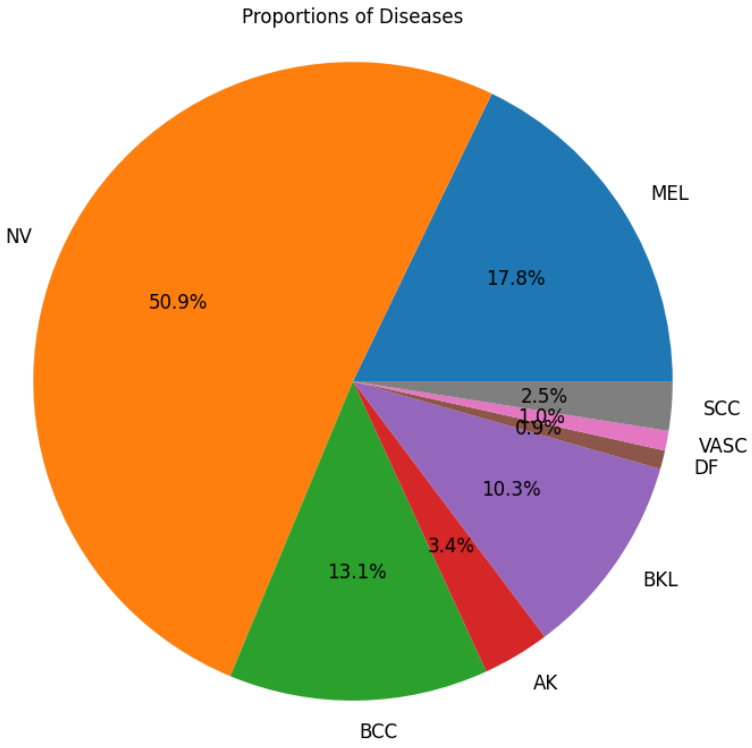

Figure 3.2: Class balance of the ISIC 2019 dataset

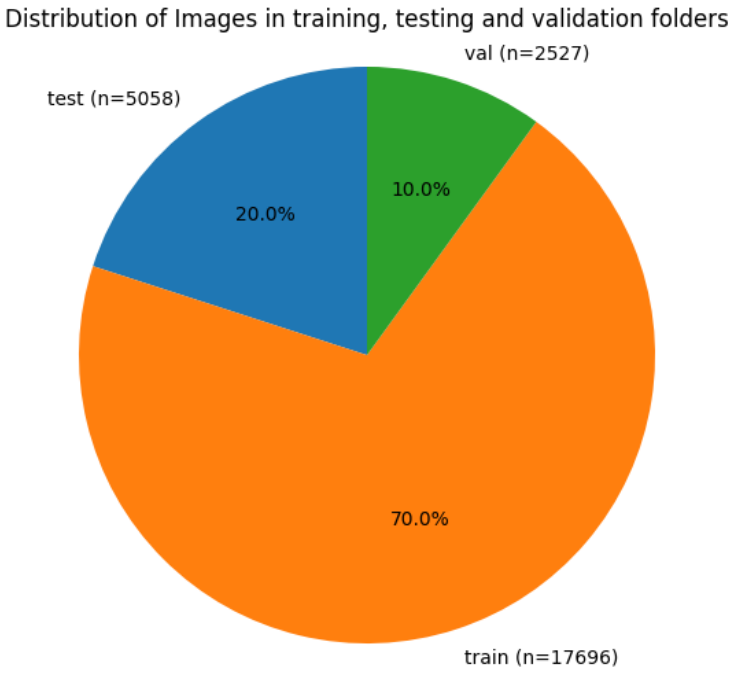

Figure 3.3: Visualisation of training, testing, validation split used in this study

3.2 CNN Models

3.2.1 ResNet-50

The first model used in this study is a ResNet-50 model. ResNet is a CNN architecture that employs a feature called residual blocks; after inputs have been fed through the layers in a block, the following block's inputs are an element wise combination of the inputs to and outputs from the previous block. A residual block can be seen in Figure 3.4. Residual blocks allow the model to skip weight layers if they are not necessary by considering the effect a block has, which helps the model avoid over fitting to the training data (He et al. 2016). ResNet is a popular CNN architecture for skin lesion classification, used in several studies such as those conducted by Perez et al. 2018, Rashid, Tanveer, and H. A. Khan 2019 and Bozkurt 2023. The popularity of ResNet models for skin lesion classification is the reason it is used in this study. By aligning the methodology of this study with other studies in the field, the findings of this study will be more applicable.

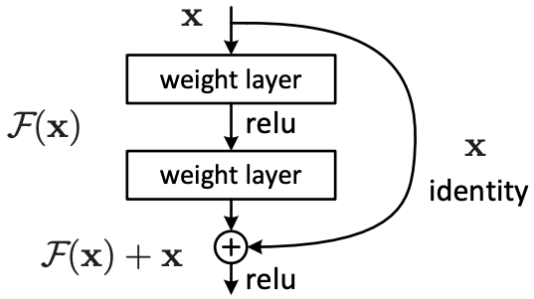

Figure 3.4: Diagram of a residual block (He et al. 2016)

3.2.2 DenseNet-121

The other model used in this study is DenseNet-121. DenseNet is a CNN architecture that utilises a feature called dense blocks. Every layer in a dense block receives additional inputs from all of it's preceding layers, and passes on it's outputs to every layer that follows it. An example of a dense block can be seen in Figure 3.5. Unlike in a residual block, the inputs do not get combined before they are passed through layers. Instead, at each layer the new set of inputs are added to the "collective knowledge", then the final classifier makes a decision based on this entire set of inputs. Layers in

a DenseNet are relatively narrow, meaning there there is not a large amount of information to add at each layer. This arrangement makes DenseNets easier to train due to better flow of information, and fewer parameters needing to be trained (Huang et al. 2017). DenseNet is also used in this study due to it's widespread use for skin lesion classification, like in the studies conducted by Perez et al. 2018, Rashid, Tanveer, and H. A. Khan 2019, Y. Zhang and Wang 2021.

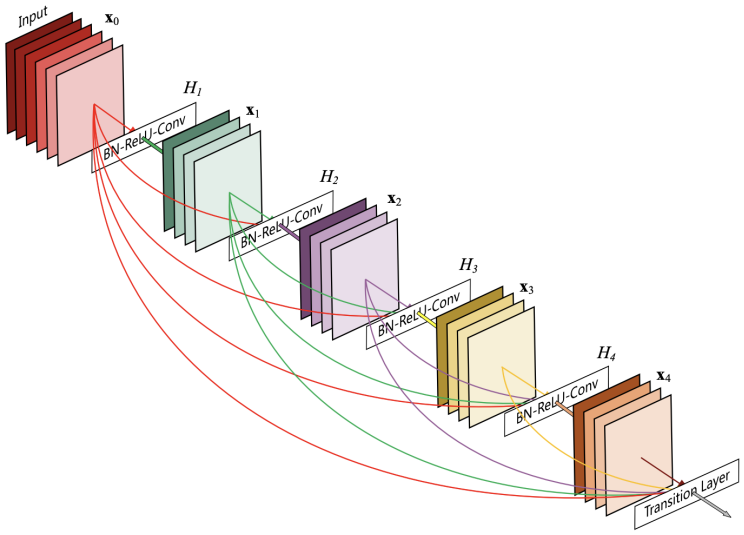

Figure 3.5: Diagram of a dense block (Huang et al. 2017)

3.3 Basic Data Augmentations

The "torchvision.transforms" package (Contributors 2016) is used to apply basic augmentations to the training data. An exhaustive list of basic data augmentations are used in this test to cover as many basic augmentation methods as possible, once again to make the findings of this study more applicable to future studies. The methods of basic data augmentation that are used are:

- Horizontal Flipping
- Rotation
- Cropping
- Translation
- Colour Change

- Gaussian Blurring
- Perspective Change
- Scaling
- Noise Injection
- Random Erasing

3.4 Generative Data Augmentations

The code used to train the GAN is available in the GitHub repository by Aastha Agrawal 2020. The type of GAN used in this study is called a conditional GAN, or cGAN. A cGAN differs slightly from the definition of a GAN in chapter 2, as the generator and discriminator networks are also trained with an additional input to differentiate between classes. This type of GAN is used in this study as the dataset being dealt with has 8 different classes; a regular GAN can only aim to generate images of one mode, whereas a cGAN can generate images with differing classes. It would be possible to utilise 8 different GANs, one for each class, but a single cGAN is used instead for simplicity. The code written by Aastha Agrawal 2020 was originally intended to suit the MNIST Handwriting dataset (Deng 2012), but parameters are changed to suit this study. The architecture of a cGAN compared to a standard GAN can be seen in Figure 3.6.

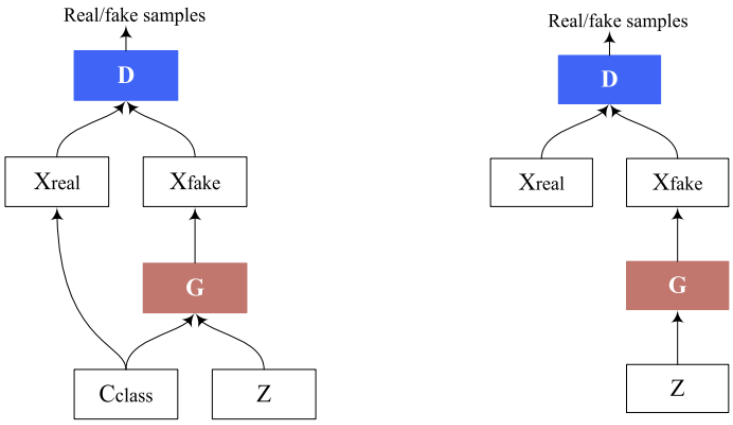

Figure 3.6: Diagram of a cGAN, left and a regular GAN, right (Saxena and Cao 2021)

3.5 Metrics

The description of precision, recall and F1 score in Chapter 2 show how these metrics are calculated for one class, the positive class, in a binary classification problem. The method to calculate any one of these metrics for a class in a multi class classification problem, like the one in this study, is very similar; the class being measured is thought of as positive and the rest as negatives. In this study there are 8 classes, so when metrics are being calculated for each class there is one positive class and 7 negative classes. The precision, recall and F1 scores can be calculated for each of these classes using same equations in Chapter 2, which would give 8 separate scores for each metric. However, in order to make the results from this study more readable, the metrics for all 8 classes are combined into a single value for precision, recall and F1 score. This is done using a weighted average, which gives an average score for each metric that accounts for class imbalance; a weighted average applies more weight to classes which have more instances. For example, Equation 3.1 shows how precision is calculated for class ”k”, and Equation 3.2 shows how the weighted average Precision is calculated for all classes, where ”n” is the number of classes and ”w” is the weight of the class. The weight of a class is determined by the number of instances of that class. The same logic shown in Equations 3.1 and 3.2 can be applied to find the weighted recall and weighted F1 score. Accuracy is calculated in the same way as described in chapter 2, as the number of correct predictions made by the classifier divided by the number of total predictions.

$$\text{Precision}_k = \frac{TP_k}{TP_k + FP_k} \tag{3.1}$$

$$\text{Weighted Average Precision} = \frac{\sum_{k=1}^{n} w_k \cdot \text{Precision}_k}{\sum_{k=1}^{n} w_k} \tag{3.2}$$

3.6 Overall Study Design

Combining all of these parts together results in both the ResNet-50 and DenseNet-121 models being trained and tested 12 separate times, first without any data augmentation and then 11 times for each of the data augmentation methods. The study will be run by utilising a loop, after which the results will be displayed in the format of 8 graphs. 4 graphs per model will be shown, one graph per metric. For each metric, the graph will show the baseline performance of the model without any data augmentation applied,

with a bar for each data augmentation method to show the increase or decrease it has caused for that metric.

Chapter 4

Implementation

The code for this study was developed in Google Colab, then the final version of the code was run using an NVIDIA RTX A5000 with 24GB of memory.

4.1 Data Preparation

The data used in this study comes in the form of a single folder of all the images and a .csv file for the ground truth of the data, showing which disease each image in the folder represents; each row in the .csv file represents an image. Before the data can be used to train and test a CNN, its integrity must first be verified through cleaning, then it must be split into separate folders for training, validation and testing, and each of these folders must contain further sub-folders containing images according to the disease they show.

4.1.1 Cleaning the Data

Cleaning the data for this study involves ensuring that there are no data that show less than or more than one disease, and making sure that there are no duplicate images in the dataset.

The ground truth file comes in the format shown in Figure 4.1, where an image is shown to portray a disease if the column for that disease contains ”1”, else it contains ”0”. This means that to find any images which contain less than or more than one disease, the total of all columns for each image can be summed. The sum of all columns for every image should be just one if it only contains one disease. The process used to do this is shown in Figure (4.2), the output showed there were no images which

returned a value greater than or less than one, so the dataset did not need cleaning of any missing or incorrect data. The code used for this can be seen in Figure A.1 in Appendix A.

image	MEL	NV	BCC	AK	BK
ISIC_0000000	0	1	0	0	
ISIC_0000001	0	1	0	0	
ISIC_0000002	1	0	0	0	
ISIC_0000003	0	1	0	0	
ISIC_0000004	1	0	0	0	
ISIC_0000006	0	1	0	0	
ISIC_0000007	0	1	0	0	
ISIC_0000008	0	1	0	0	

Figure 4.1: Section of the ground truth file

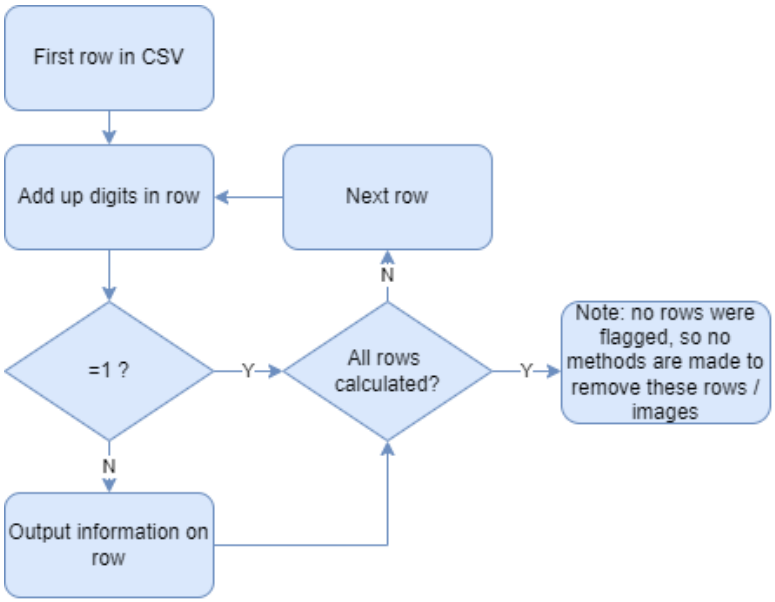

Figure 4.2: Flowchart to show process of removing images showing less than or more than one disease

In order to clean the dataset of duplicate images, hash functions of each image in the image folder are calculated and compared. First a function to find duplicate hash functions is employed; iterating through each image, calculating its hash function, then adding the image name to a dictionary with the hash function as the key. If any hash function key in the dictionary contains more than one image name, this means that

these images are duplicates of each other. Any key in the dictionary with only one entry is deleted from the dictionary, leaving only hash functions of duplicate images and their corresponding image names. For each remaining hash function key in the dictionary, another function is then used to delete all but one of the images under each key from the image folder. Then, another function removes the entry for those images from the ground truth .csv file. This process can be seen in Figure 4.3. Running this code removes around 50 images from the dataset. The code is run again and removes no more images from the image folder, verifying that there are no more duplicates in the image folder. The code used for this can be seen in Figures A.2, A.3, A.4 and A.5 in Appendix A.

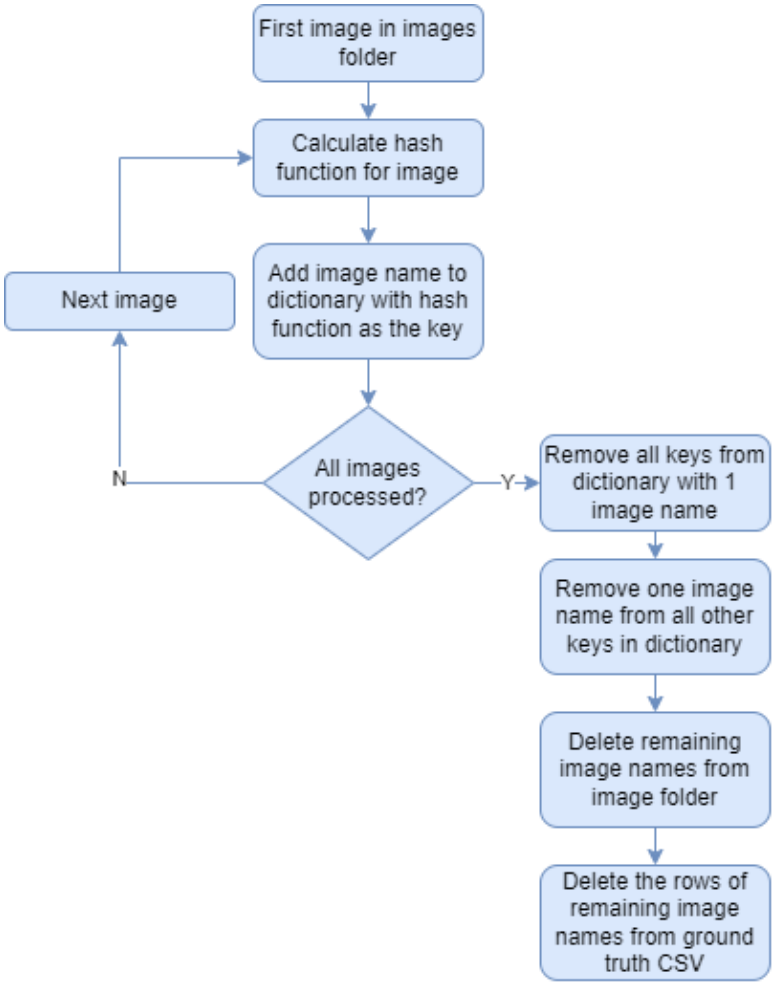

Figure 4.3: Flowchart to show process of removing duplicate images by comparing hash functions

4.1.2 Splitting and Organising the Data

Before the data arranged into folders it must be shuffled to ensure that the data in each folder is from random parts of the dataset, in case there is any bias in the way the data is initially organised. To shuffle the dataset, a new column is added to the ground truth file containing a random float from zero to one. The rows in the ground truth file are then put in order according to this float, randomising each row's position.

Also, the total number of images to put in each folder and sub-folder must be calculated. This is done by iterating through each disease in the ground truth file, counting the total number of entries for that disease, then calculating how many entries of the disease to put in the training, testing and validation folders. Each disease is to have 70% of its instances in the training folder, 20% in the testing folder, and the remaining 10% in the validation folder. Since the result of a percentage may not be an integer, the result is rounded for the training and testing totals, and the validation folder total is whatever remains. The results are added to a dictionary, with the disease name as the key. The dictionary contains the total number of images and the number of images that each sub-folder should contain for that disease; this can be thought of as the disease totals dictionary. Each total in this dictionary for training, testing and validation folders will all have the same ratio of images for each disease. This is an example of a stratified split.

Sub-folders for each disease are made in the training, testing and validation folders. Counters are initialised at zero, one counter for each disease for each of the training, testing and validation folders. Iterating through each disease, going down the ground truth file, if an image name contains the current disease, the corresponding image from the images folder is added to its disease's sub-folder in the training folder. Each time an image is added to the training folder, the counter for that disease is incremented. When that counter reaches the amount that is meant to be in the training folder for that disease, held in the disease totals dictionary, images containing the current disease will start to populate the sub-folder in the testing folder. Then, once the testing folder reaches the correct number, the rest of the instances are placed in the validation sub-folder. After iterating through each disease, the correct number of images should be in each folder and sub-folder. This is verified by code shown in Figure A.6 in Appendix A, which also verifies that there are no duplicate files across the folders and sub-folders.

4.1.3 Final Steps for Data Preparation

After the data has been cleaned and organised, the average and standard deviations for each colour channel of all the images in the training folder must be calculated and recorded in order to normalise the data in later stages. These are calculated using the ”torch” package (Contributors 2016), then saved. This can take several minutes, so is done at this stage and added to the final .zip file containing the prepared data, so it only has to be done once. Once the averages and standard deviations are calculated, a .zip folder is created. This contains the data in the correct format, and the averages and standard deviations needed for normalisation when the data is loaded into the CNNs. The whole of the data preparation stage is shown in a flowchart in Figure 4.4.

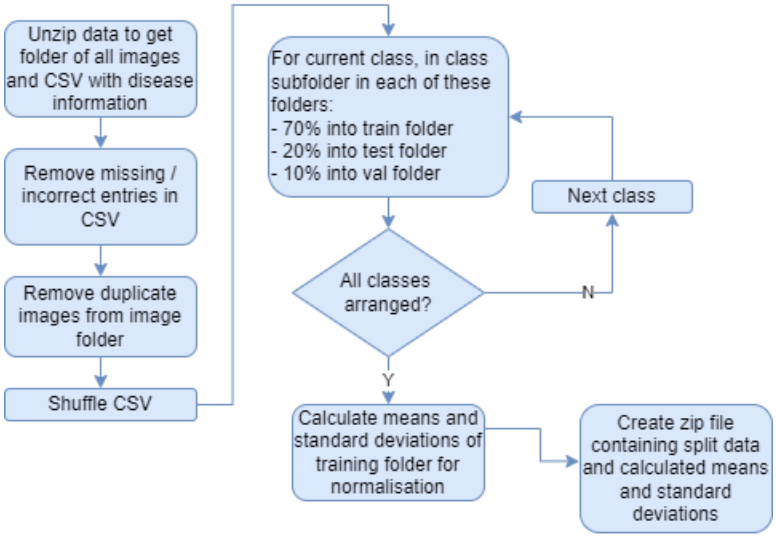

Figure 4.4: Flowchart to show the entire process of preparing data for use

4.2 Initial Data Loading

In order to put the images in a format that the CNN can take, the images must be put into data loaders. The initial transformations applied to training, testing and validation data resize the images to 224 pixels by 224 pixels, transform the images to a tensor, and to normalise the tensors according to the mean and standard deviation values that were calculated in the data preparation stage. The training, testing and validation sets are then loaded with a batch size of 32, with the training data being shuffled again when loaded. This is the data loading method used when training a CNN with no data augmentation methods applied to the training data.

The same transformations are also used to reset the training data loader after a CNN has been trained with a data augmentation technique applied to the training data. The function to reset the data loader takes the original training data from the prepared dataset, applies these transformations to it, and replaces the data that was previously in the training data loader. The transformations used for initial data loading and to reset the training data loader can be seen in Figure 4.5 in green, under ”D”, as this is the default data loader method. The code for the default data loaders can be seen in Figures A.7 and A.8, and the code to reset the training data to its default state can be seen in Figure A.9 in Appendix A.

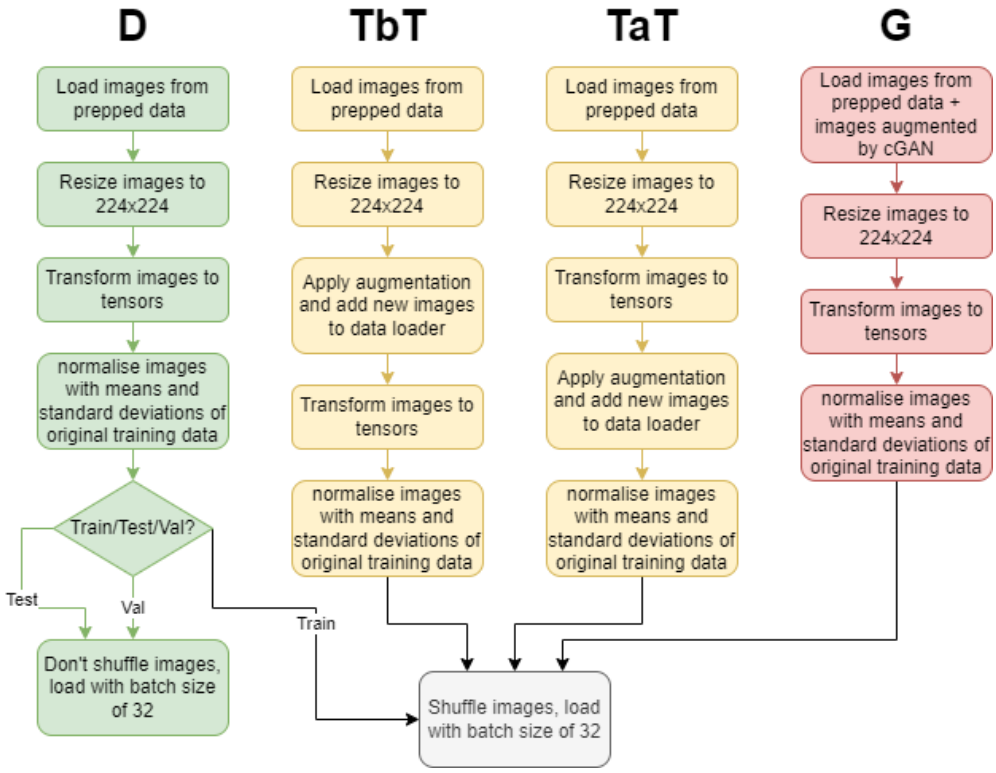

Figure 4.5: Diagram of the logic behind different data loaders

4.3 CNN Initialisation

Both CNNs in this study are initialised very similarly, using packages from ”torch”. The only differences in initialisation between ResNet-50 and DenseNet-121 in this study is that ResNet-50’s fully connected layer has an input size of 2048 and DenseNet-121’s has an input size of 1024, both being reduced to the 8 classes. Both models take images of size 224 by 224 pixels, both models are downloaded from ”torch” and are

set to be initialised with transfer learning, being pre-trained with ImageNet, and both are set to use the same model hyper-parameters. These hyper-parameters are chosen to reflect those used in similar studies like those conducted by Perez et al. 2018 and Rashid, Tanveer, and H. A. Khan 2019:

- 30 epochs
- Loss function: Cross Entropy Loss
- Optimiser: Stochastic Gradient Descent with an initial learning rate of 0.001 and a batch size of 32
- Scheduler: Reduce learning rate by a factor of 0.1 after no improvement in validation loss for 5 epochs
- Early stopping: Stop model training if there is no improvement in validation loss for 10 epochs

This study revolves around repeatedly training the CNNs models from a constant starting point, which requires them to be initialised several times. Figure 4.6 shows how the models are initialised. The initialisation of each model is held in a function so that it can be called to multiple times to set the CNN and reset its weights while the code is running. The code in Figures A.10 and A.11 in Appendix A shows that a separate function is used to initialise each model.

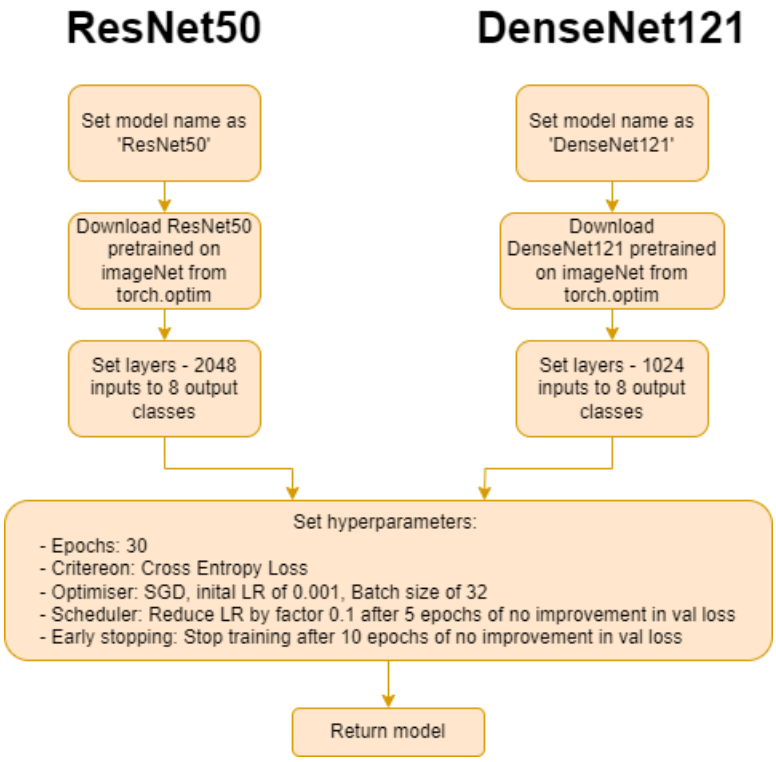

Figure 4.6: Flowchart to show the similarities in CNN model initialisation

4.4 Defining How Models will be Trained

The same training function is used to train both CNN models, which can be seen in figure 4.7. It starts by initialising arrays to hold training and validation accuracies and losses, then starts looping through each epoch; the number of epochs has been set when the CNN was initialised. During every epoch the model is first set to train, where it does a forward pass with the training data and calculates the training losses and accuracies. Then, the model does a backward pass and updates its weights according to the training losses and accuracies. Then the model is set to evaluate and does a forward pass with the validation data and calculates the validation losses and accuracies. All these passes are done in batches to manage memory usage. After all batches have been passed for the current epoch the overall training and validation accuracies and losses are calculated for the epoch and added to the arrays. Then, checks are made for whether the model's learning rate should decrease or if the model should stop training early. This is why a forward pass is also done with validation data. As was set when the CNN was initialised, 5 successive epochs with no improvement in validation accuracy

will cause the learning rate to be reduced by 10 times (multiplied by 0.1), and 10 successive epochs with no improvement in validation accuracy will cause the model to stop training early, as this shows the model is not learning any new features. The code to implement the training method can be seen in Figures A.12 and A.13 in Appendix A.

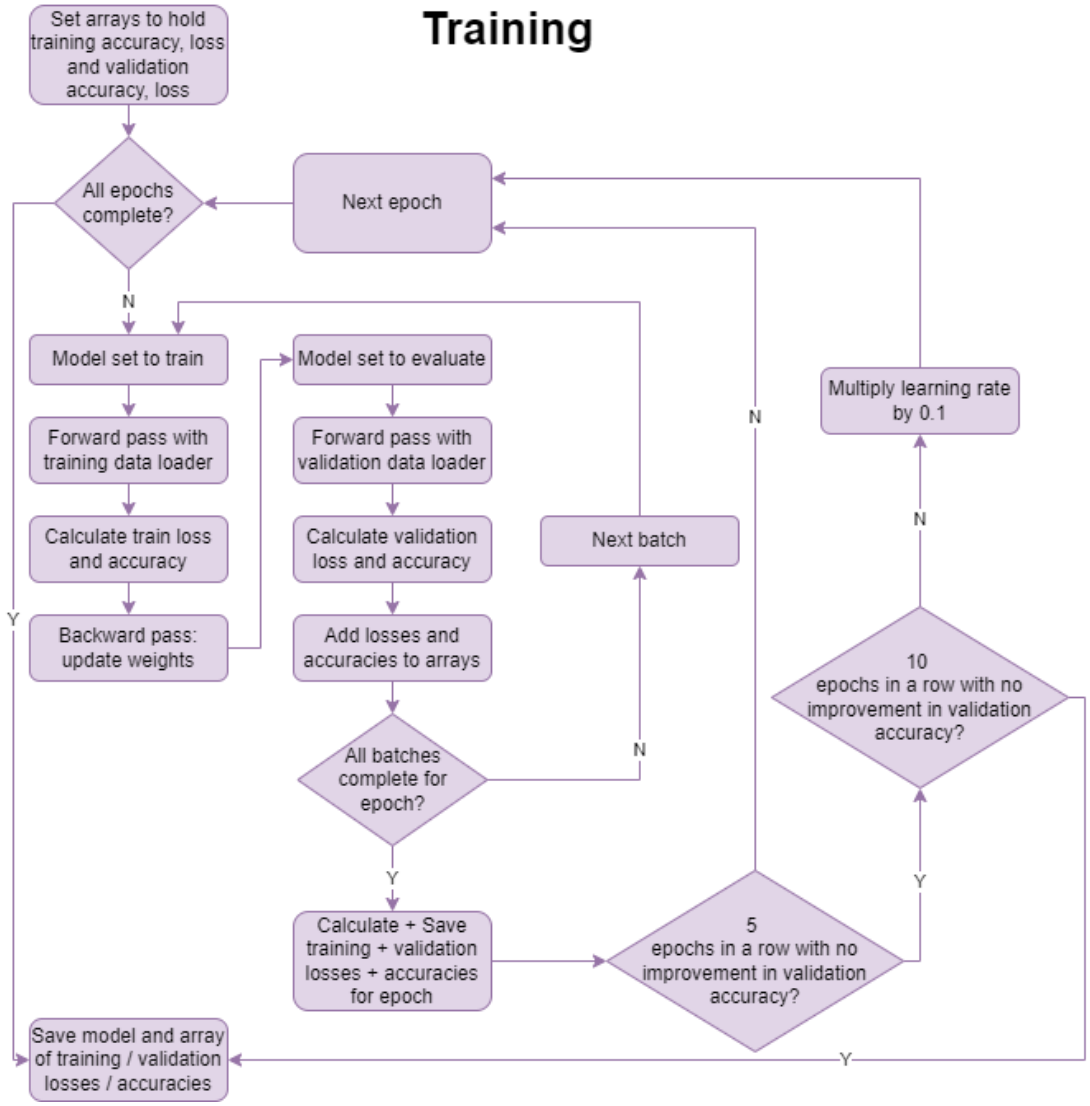

Figure 4.7: Flowchart to show how CNNs are trained

4.5 Defining How Models will be Tested

Testing involves evaluating the model on the various metrics. To test the model, first an array is initialed to hold the true class of each image and the predicted class of each

image. The model is set to evaluate and does a forward pass with the testing data in batches. Once all predictions have been made and added to the predicted labels array, the true labels and the predicted labels are compared. The metrics are then calculated using the ”sklearn.metrics” package (Pedregosa et al. 2011), the confusion matrix is generated, and the results are saved. The process of testing can be seen in Figure 4.8, the code of which can be seen in Figures A.14 and A.15 in Appendix A.

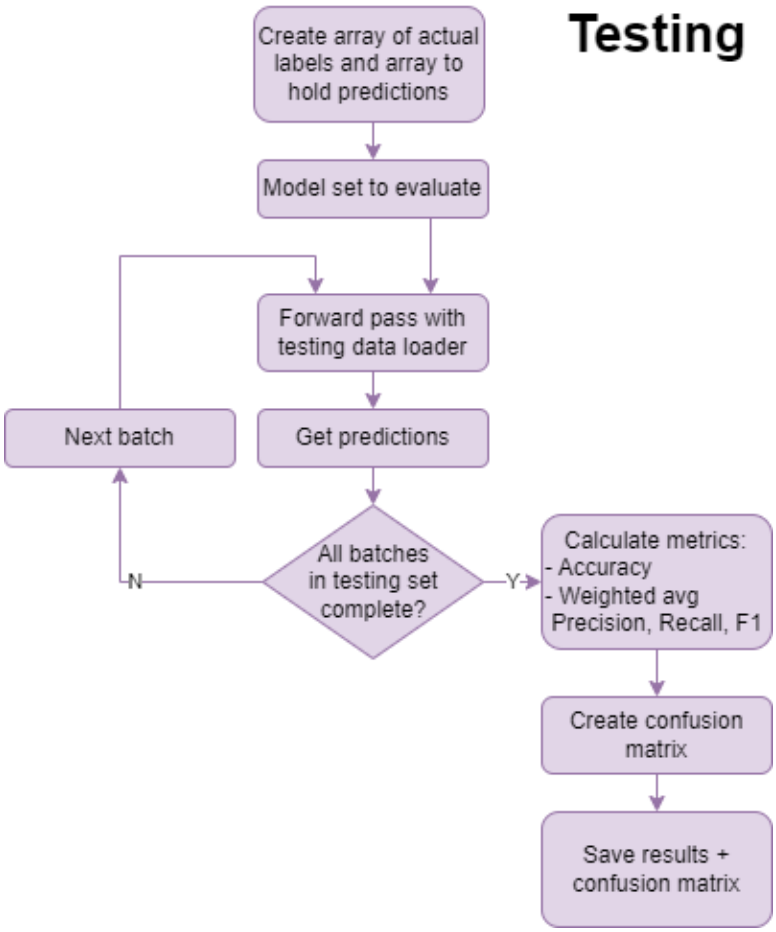

Figure 4.8: Flowchart to show how CNNs are tested

4.6 Basic Data Augmentations

The basic data augmentations are arranged in arrays as shown in code in figure 4.9. There are 3 arrays of basic data augmentations; one array holds those that need to be applied before the image is transformed into a tensor, another holds those that need to be applied to an image after it is transformed into a tensor, and the other array

holds only one augmentation, scaling. Scaling also needs to be applied before the image is transformed into a tensor, but it utilises the same transformation function from ”torchvision.transforms” package (Contributors 2016) as the cropping augmentation does, but with different parameters.

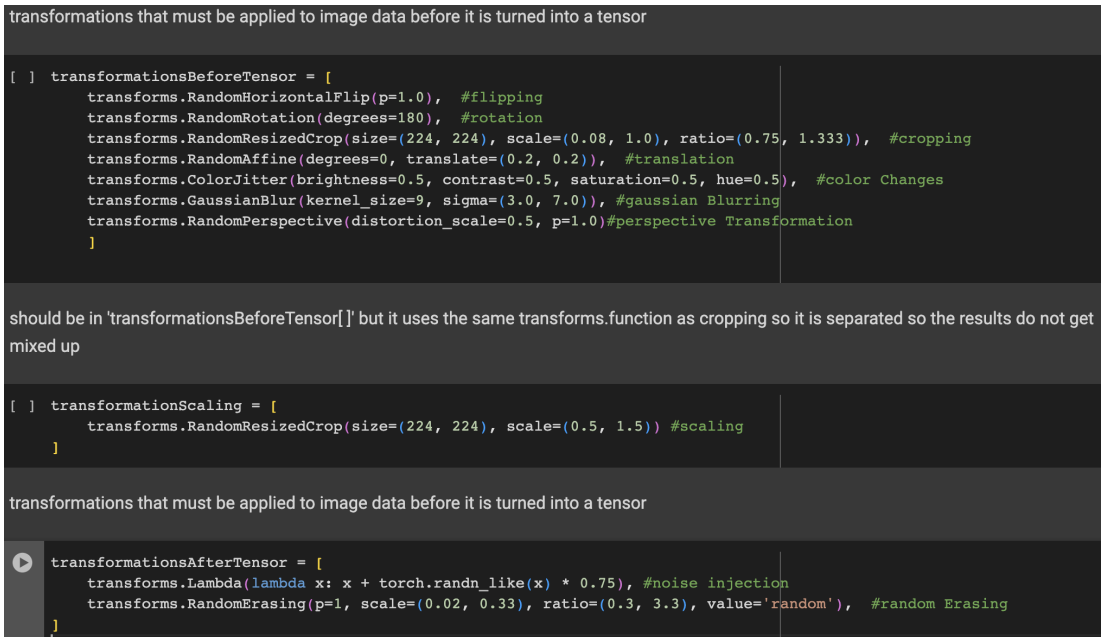

Figure 4.9: The code used to initialise the basic data augmentation methods, showing the parameters of each one.

The reason that it is in a different folder is that when the results are written to a .csv file while the code is running, the results are written according to the name of the transformation from the ”torchvision” package. Placing scaling in a separate folder allows for easier intervention in the code to write the results for the scaling augmentation under a different alias to the one used for cropping. Figure 4.10 shows the different folders of basic augmentations. Each ending of a branch is a different folder; ”Tbt” refers to ”Transform before Tensor”, augmentations that are applied before the image is turned to a tensor, and ”TaT”, ”Transform after Tensor”, refers to augmentations that need to be applied after the image is turned into a tensor.

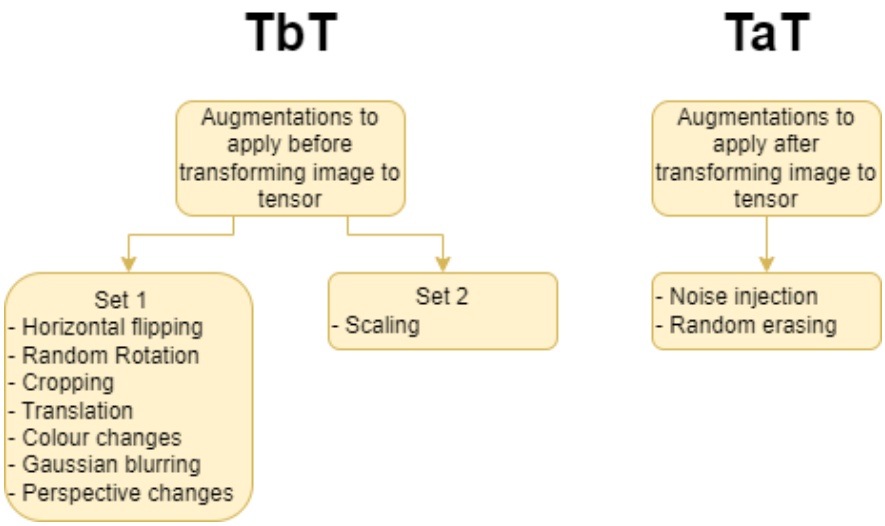

Figure 4.10: Basic augmentation folders

Applying these augmentations involves one of two functions, which are simple modifications of the initial training data loading method. To apply the augmentations that need to be applied before an image is transformed to a tensor, the ”augmentDataBeforeTensor()” is used. This is used to apply methods in the first array of basic augmentations and scaling. For the other basic data augmentations, ”augmentDataAfterTensor()” is used. The only difference between these functions is that two lines, shown with arrows next to them in figure 4.11, are swapped. The procedure to update the training data loaders to hold data with a basic data augmentation method are shown in yellow in figure 4.5. The titles ”TbT” and ”TaT” mean the same thing as they do in figure 4.10. The specific parameters for each basic data augmentation method can be seen in figure 4.9. Examples of an image with each of the basic augmentations applied can be seen in figure 4.12. These are images taken from this study.

```
def augmentDataBeforeTensor(transformation_type):

    #apply data augmentation to the train data loader directly
    data_transforms['train'] = transforms.Compose([
        transforms.Resize((224,224)),
        transformation_type,  #apply the current transformation
        transforms.ToTensor(), #then turn to tensor
        normalize
    ])

    #update train data loader with the augmented transformations
    image_datasets['train'] = datasets.ImageFolder(os.path.join(currentDataset, 'train'), data_transforms['train'])
    dataloaders['train'] = torch.utils.data.DataLoader(image_datasets['train'],
                                                       batch_size=32,
                                                       shuffle=True,
                                                       num_workers=0)
    return image_datasets['train'], dataloaders['train']
```

Figure 4.11: The function to apply basic data augmentations before image data is transformed to a tensor

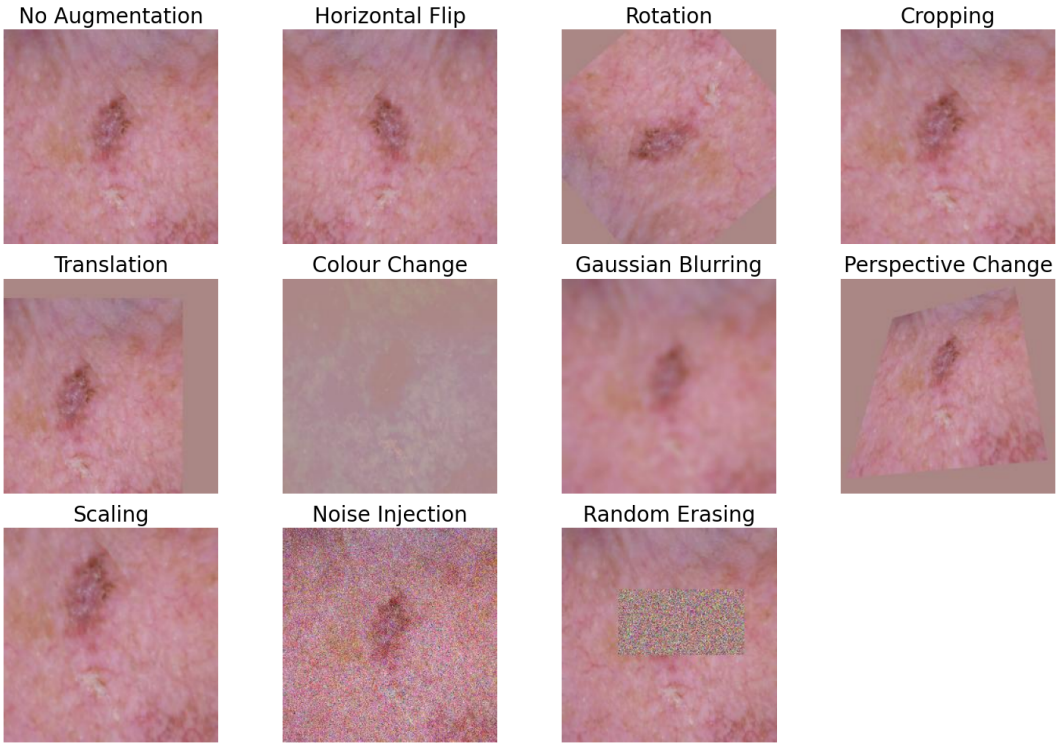

Figure 4.12: Examples of all basic data augmentations on an image, taken from this study

4.7 GANs

The code used to define and train a cGAN for this study is available on GitHub, and was written by Aastha Agrawal 2020. The code was initially written to be trained with

the MNIST handwriting dataset (Deng 2012), to generate 32 by 32 pixel images of 10 different classes; the numbers 0 to 9. The parameters are changed for this study to suit the 8 classes in the ISIC dataset and to take input images of the dimensions 224 by 224.

The cGAN is trained for 10 epochs, where the generator creates images by taking random noise and class labels as an input, and the discriminator is trained to distinguish between the images from the generator and the real data from the training data loader, while also taking note of class labels. The generator will initially output random noise, but after each epoch the output from the generator improves more to resemble a real skin lesion. The cGAN will output images at regular intervals between inputs so the progress can be visualised, this can be seen in the case of the class NV in figure 4.13.

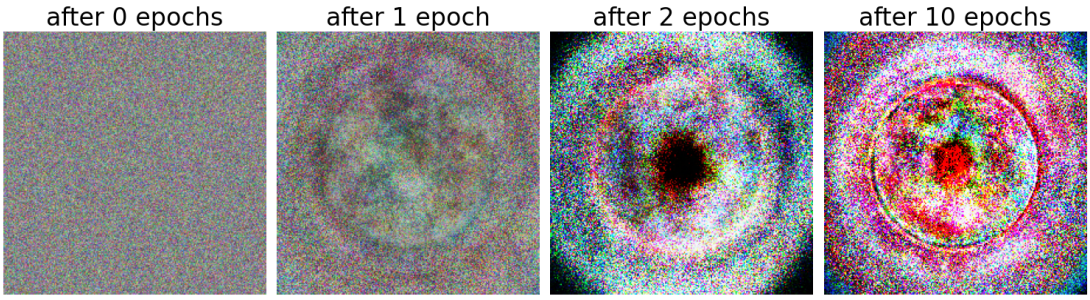

Figure 4.13: Output from cGAN for class NV after intervals of epochs

The cGAN was initially trained with 100 epochs, returning images at intervals during training. After inspecting the results at each stage, it was concluded that the image quality started to degrade after 10 epochs, so the final images were generated after the cGAN had trained for 10 epochs. An example of an image generated after 100 epochs for the NV class can be seen in figure 4.14.

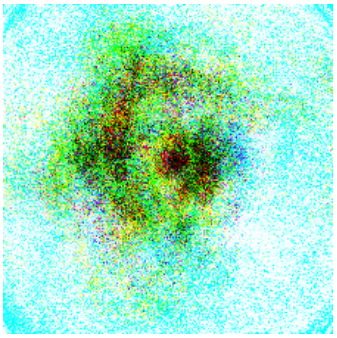

Figure 4.14: Generated image of NV after 100 epochs

After the cGAN has been trained on the training data, the number of images it generates for each class depends on how many images are already present in each class. The same number of images are generated and added to each class as are already present, the code to work out the number of each class to generate can be seen in Figure A.16 in Appendix A. This is also done with the basic augmentations, the reason being to not allow any alterations to class balance impact the results that each data augmentation method has on metrics. Examples of each class generated from the trained cGAN can be seen in figure 4.15.

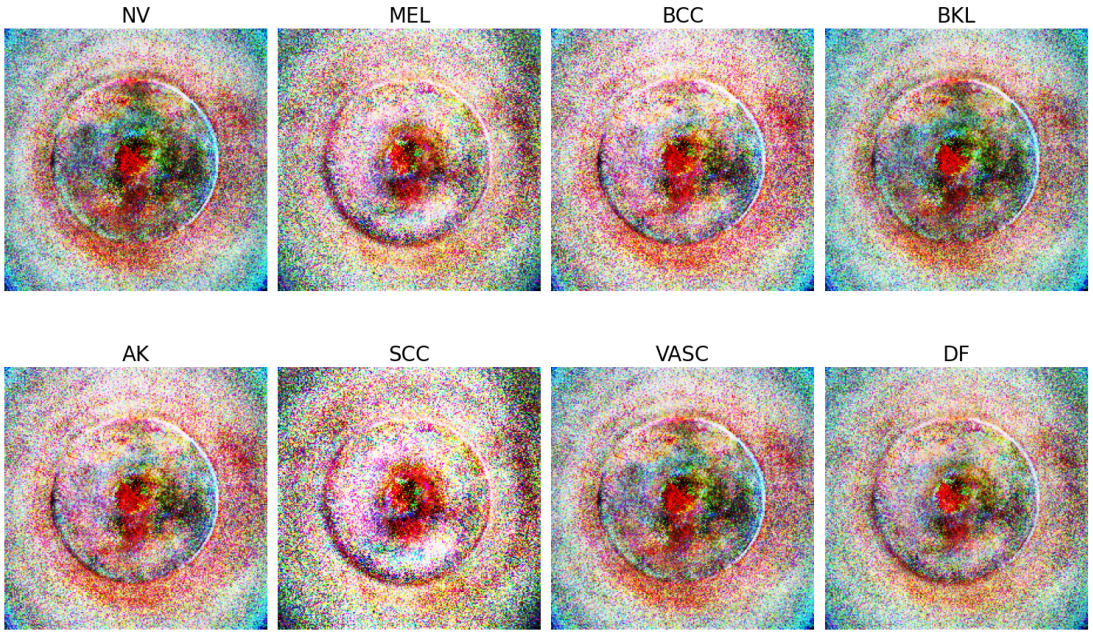

Figure 4.15: Examples of images generated by trained cGAN for each class

The process of generating images with the cGAN can be seen in figure 4.16, and the data loading method when training and testing CNNs with data augmented with the cGAN can be seen in red, under ”G” for generative, in figure 4.5. The code for these two processes can be seen respectively in Figures A.17 and A.19. The code to create the folders that hold the original data and the cGAN data can be seen in Figure A.18 in Appendix A.

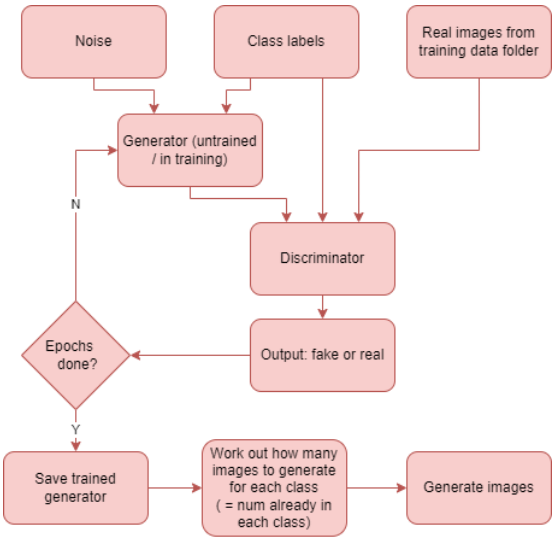

Figure 4.16: The process of training a cGAN and generating correct number of images for each class

4.8 Running the experiment

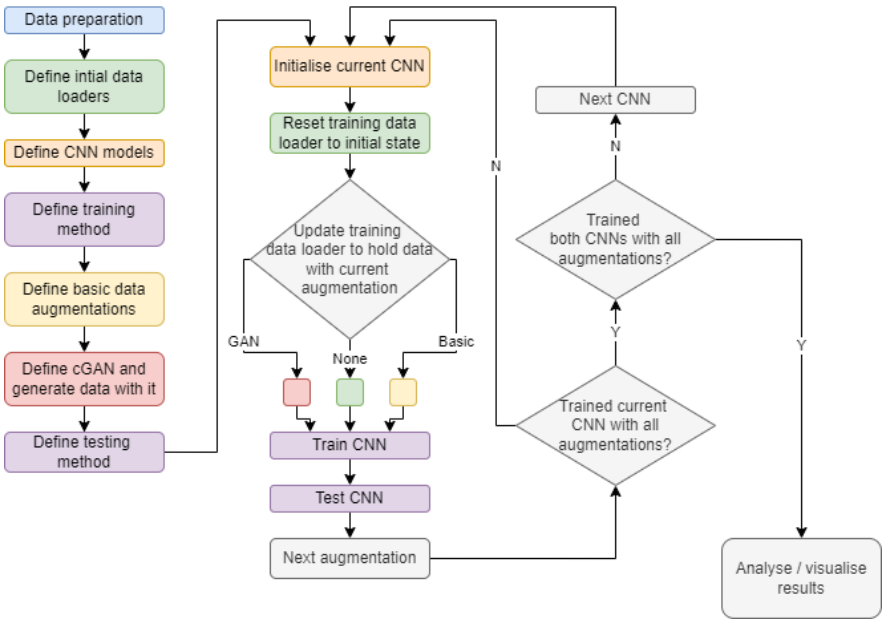

Figure 4.17: Method to run the experiment

A loop is used to run the experiment, bringing together all the parts defined in this chapter. The loop can be seen in figure 4.17, each part of this flowchart is colour coded

to match the flowcharts shown previously in this chapter. The processes that happen before the loop starts are as follows. First, shown in blue is the data preparation stage to clean and organise the data. Then, this data is put in a format the CNNs can take when the initial training, testing and validation data loaders are defined, shown in green. After that, the steps for CNN initialisation are defined, shown in orange. Then the training method is defined, shown in purple, the lists of basic data augmentations are defined, shown in yellow, the cGAN is defined and generates new images, shown in red, and the testing method is defined, shown again in purple.

After these initial steps are complete, the loop commences. At each iteration of the loop, the current CNN is initialised, The data loaders are reset to their initial state, then updated to hold the data with the current augmentation, then the model is trained and tested. After this the loop iterates again, starting by re-initialising the CNN. The loop will iterate through every augmentation until they have all been implemented individually, then the CNN changes to the next model and iterates through every augmentation individually again. The order which the models and augmentations are implemented are as follows: The first model is ResNet-50, then DenseNet-121, and the first augmentation is no augmentation, then the cGAN augmented data, then each individual augmentation in the separate folders of basic augmentations. This loop continues until both models have been tested after being trained with all augmentations, a total of 12 times for each model, at which point the results can be analysed. The code for this loop, which shows the specific order of CNN models and augmentation methods can be seen in Figures A.20 and A.21 in Appendix A.

Chapter 5

Results

5.1 ResNet-50

Augmentation	accuracy	precision	recall	f1_score
Horizontal Flip	0.6822854877	0.659231568	0.6822854883	0.6553892579
Random Erasing	0.6336496472	0.5985646796	0.6336496639	0.6028338798
Translation	0.6818900704	0.6617635344	0.6818900751	0.6559742351
Noise Injection	0.5086991191	0.2587747647	0.5086990905	0.3430435749
Perspective Change	0.5776987076	0.6069890312	0.5776986951	0.5739877208
Rotation	0.6706207991	0.6381088802	0.6706207987	0.6411330513
Scaling	0.669632256	0.6368606137	0.6696322657	0.6379985238
Colour change	0.6500592828	0.6250262008	0.650059312	0.6034364971
Cropping	0.6518386602	0.6237500734	0.6518386714	0.6160271853
No Augmentation	0.6947410107	0.6681267186	0.6947410043	0.6679483225
Gaussian Blurring	0.5636615157	0.5089875342	0.5636615263	0.486805029
GANs	0.6755634546	0.6441782829	0.6755634638	0.6438744326

Figure 5.1: Results for each augmentation on ResNet-50 (results are measured as a proportion of 1)

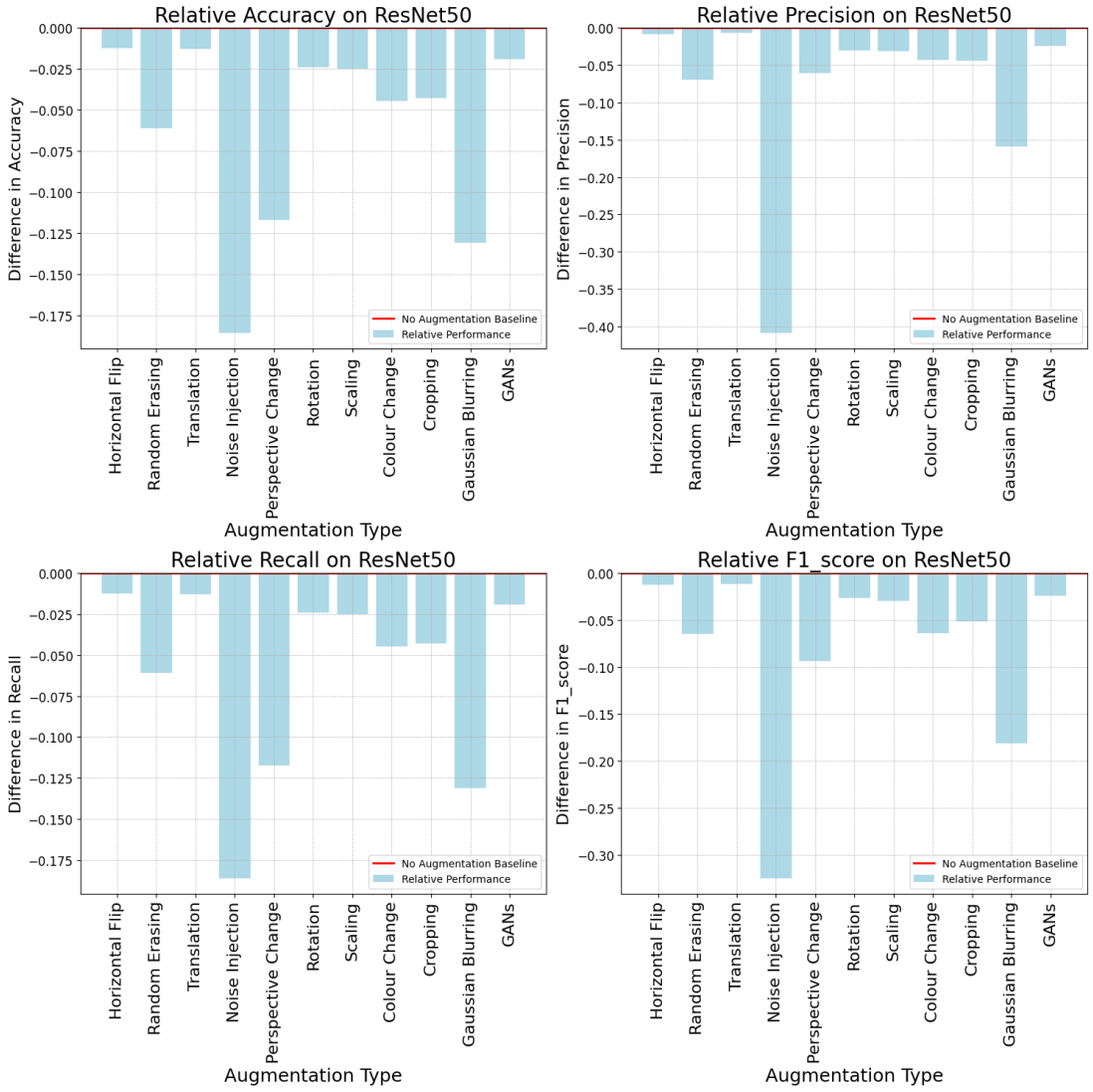

Figure 5.2: Graph to show results from ResNet-50: metrics from each augmentation compared to baseline metrics from no augmentation

The overall accuracy, precision, recall and F1 score achieved by each augmentation on the ResNet-50 model can be seen in figure 5.1. The results show that for ResNet-50, none of the augmentation methods lead to any improvement over the results from training the model without any augmentation. Applying no augmentation to any of the training data lead to the ResNet-50 classifying with an accuracy of 0.6948, a precision of 0.6681, a recall of 0.6947 and an F1 score of 0.6679. Figure 5.2 shows the baseline performance of no augmentation at the top, with bars to show the impact that each augmentation had on each metric. The translation method performed the second best after no augmentation. Augmenting the training data with translation decreased all

metrics. The least affected metric was precision which suffered a decrease of 0.0064 and the most affected metric was accuracy which suffered a decrease of 0.0129. Recall and F1 score also suffered decreases of 0.0129 and 0.0120 respectively. The worst performing augmentation method was noise injection. Augmenting the training data with noise injection lead to significant drop offs in each metric, from a reduction in accuracy by 0.1860 to a reduction in precision by 0.4094. Noise injection also lead to a reduction in recall by 0.1860 and a reduction in F1 score by 0.3249.

5.2 DenseNet-121

Augmentation	accuracy	precision	recall	f1_score
Horizontal Flip	0.6941478848	0.6861359729	0.6941478845	0.6798640989
Random Erasing	0.6231712103	0.5872635674	0.6231712139	0.5796631151
Translation	0.6613286138	0.6488591528	0.6613285884	0.6430547905
Noise Injection	0.5086991191	0.2587747647	0.5086990905	0.3430435749
Perspective Change	0.6103202701	0.6138058964	0.6103202847	0.5813058419
Rotation	0.6597469449	0.6445226855	0.6597469355	0.6329935988
Scaling	0.6767497063	0.6644996805	0.6767497034	0.6484306768
Colour change	0.6532226205	0.5982838962	0.6532226176	0.6034052808
Cropping	0.6684460044	0.6326653928	0.6684460261	0.6366155616
No Augmentation	0.6894029379	0.6654555325	0.6894029261	0.6587053539
Gaussian Blurring	0.5933175087	0.5581264758	0.5933175168	0.526383115
GANs	0.6751680374	0.6396638569	0.6751680506	0.6405914967

Figure 5.3: Results for each augmentation on DenseNet-121 (results are measured as a proportion of 1)

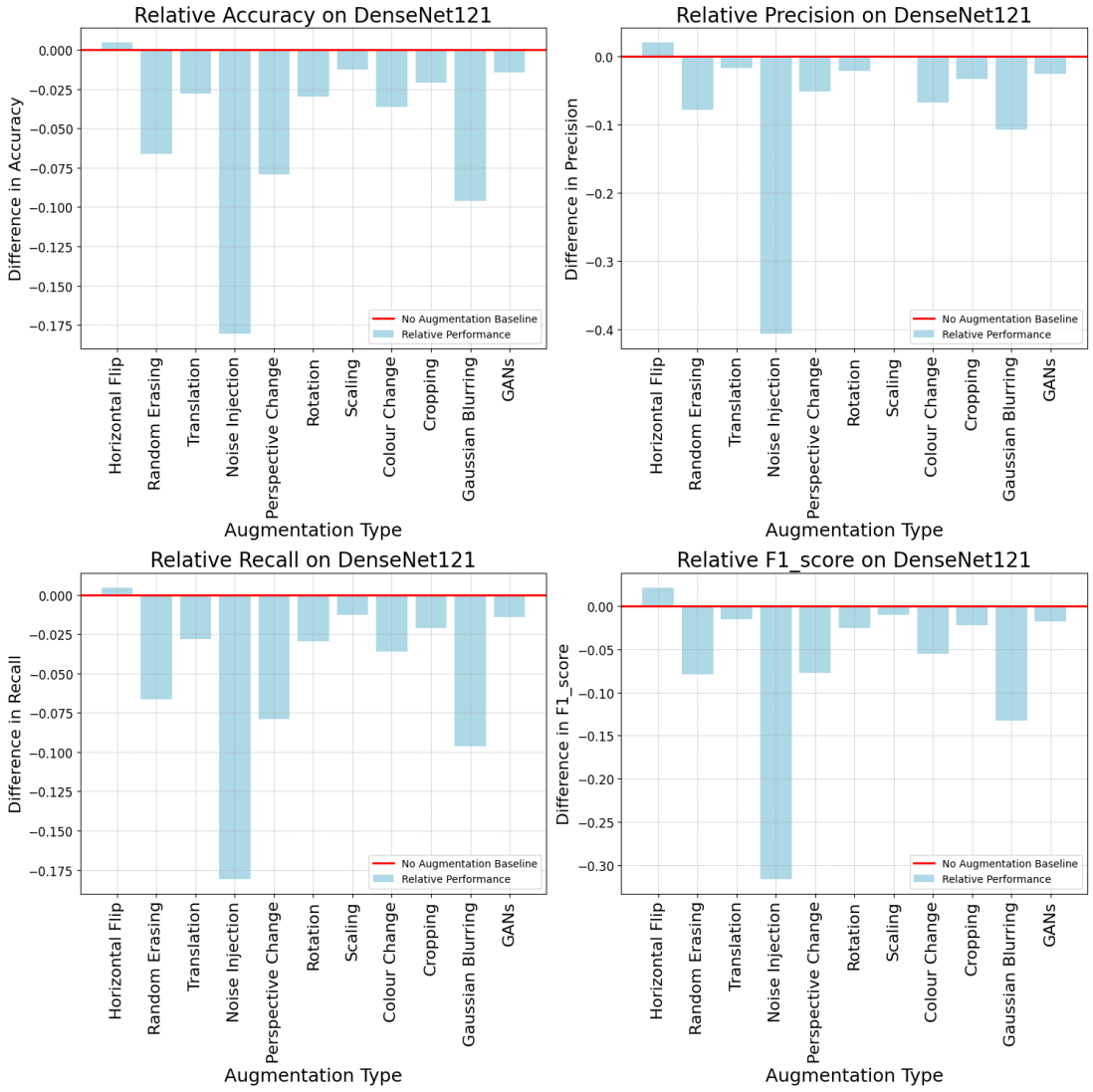

Figure 5.4: Graph to show results from ResNet-50: metrics from each augmentation compared to baseline metrics from no augmentation

The results for DenseNet-121 show that the horizontal flipping method of augmentation lead to the model being improved slightly in each metric compared to applying no augmentation at all. All of the other augmentation methods however fall short of the metrics obtained when training DenseNet-121 with no augmentation. The worst performing augmentation for DenseNet-121 was also noise injection. The metrics from applying no augmentations to the training data were 0.6894 for accuracy, 0.6655 for precision, 0.6894 for recall and 0.6587 for F1 score. Horizontal flipping lead to an increase in all of these metrics, improving accuracy by 0.0047, precision by 0.0207,

recall by 0.0047 and F1 score by 0.02116. Noise injection however lead to large reductions in all metrics, decreasing accuracy by 0.1807, precision by 0.4066, recall by 0.1807 and F1 score by 0.3157.

Chapter 6

Evaluation

The results from this study do not align with other similar studies. In studies that use the HAM10000 dataset (Tschandl, Rosendahl, and Kittler 2018), which is incorporated in the ISIC 2019 dataset (Codella, Gutman, et al. 2017, Tschandl, Rosendahl, and Kittler 2018, Combalia et al. 2019), a ResNet-50 model has been shown to achieve an accuracy of 0.9 with no data augmentation (M. A. Khan et al. 2019), and a DenseNet-121 model which utilises data augmentation has been shown to achieve an accuracy of 0.91 (Mohamed and El-Behaidy 2019). Also, using the ISIC 2019 dataset specifically, the study by Kassem, Hosny, and Fouad 2020 achieves an accuracy of 93.31% without utilising any data augmentation. The results from other studies compared to the highest accuracy result from this study, 69.47%, shows that results from other studies are significantly higher than those achieved in this study.

6.1 Dataset

Poor results could arise from issues with the dataset used. Figure 3.1 shows that there are inconsistencies between types of image in the dataset; some images have a prevalent black ring around the outside and some images include foreign objects. Another issue with the dataset is the unbalanced nature of the ISIC 2019 dataset. A study by Villa-Pulgarin et al. 2022 which incorporates variations of ResNet and DenseNet shows that using data augmentation to balance the dataset can improve the results for every metric. No process for balancing the classes was used in this study. However, the results for all metrics in the study by Villa-Pulgarin et al. 2022, even with no data augmentation, were much higher than any of the results achieved in this study. Since that study uses the same dataset as this study, the issues with the results of this study

likely do not solely arise solely from issues with the dataset.

6.2 Issues With Model Initialisation

The performances of ResNet-50 and DenseNet-121 shown in other studies, like those conducted by Rashid, Tanveer, and H. A. Khan 2019 and Mohamed and El-Behaidy 2019, are far better than the performance shown without any augmentation in this study. Also, data augmentation has been widely shown improve the performance of a CNN when classifying skin lesions (Chlap et al. 2021), whereas only one out of 22 data augmentation methods lead to any improvement in model performance in this entire study. This means that the issues in results potentially originate from the CNN models not being initialised and trained adequately. This would lead to CNNs producing poor results without any data augmentation methods applied, and data augmentation having little effect on a CNN's performance as the CNN is not initialised correctly in the first place.

The reason that the CNN models were not initialised and trained adequately is due to how the code for the study was developed. The code for this study was written using an online IDE which allows the user to rent GPU space. However, if there is no user input for a set amount of time, approximately one hour, then the IDE would disconnect and the code would stop running. Training and testing a model with or without any data augmentation with the entire dataset for 30 epochs took an estimated 3 to 4 hours, and since each model needed to be trained and tested 12 times each, this meant the time taken to train and test all augmentations on both models was an estimated 72-96 hours. This meant that neither CNN model was never successfully trained and tested with the entire dataset and full epochs even once in this IDE, due to disconnections from the server.

To get around this issue, during development a small subset of the entire dataset was used with only 2 epochs each time a model was trained. This was so that the syntax and logic of the code could be tested without being disconnected from the IDE. The parameters for each model were chosen at this stage to simply reflect the parameters used in similar studies.

The results that were being shown in the development period were also poor, but this was assumed to be due to the fact that a fraction of the dataset and a small number of epochs were being used. Once the code was verified to be fully working, it was exported and set to run on a physical GPU, which was run using the entire dataset

and the correct number of epochs each time a model was trained. The models were trained and tested with no augmentation, then with each augmentation, which took approximately a week to run. This was done because the physical GPU would not disconnect when there was a lack of user input, so this was the only way to train and test the models successfully multiple times. After both CNN models had been trained and tested with each augmentation independently, the results were collected and found to be problematic. Unfortunately, due to time constraints the models could not be re-run with different parameters.

It was due to the constraints in resources that the models could not be verified to be initialised correctly. The oversight was assuming that the models would improve their performance when trained with an entire dataset and an appropriate number of epochs. Unfortunately, this assumption was likely one of the causes of skewed results in this study.

6.3 Issues With Data Augmentation Initialisation

Despite the fact that the CNN models were likely not initialised correctly, it is surprising that just one of the 22 scenarios where data augmentation was applied lead to improved results. The inconsistent results of this study could also arise from incorrect initialisation of the data augmentation methods themselves. Any issues with the initialisation of the cGAN and the basic data augmentation methods used in this study also stem from the resource constraints during development; only the logic of the code could be verified, rather than the performance.

6.3.1 cGAN Augmentation

Firstly, the cGAN parameters were changed very little from the original code (Aastha Agrawal 2020). The main parameters changed for this study were the number of classes being dealt with, the number of input channels, and the image size. The cGAN used in this study (Deng 2012) was written to generate the much smaller and simpler images from the MNIST handwriting dataset, so it was not optimised to generate the more complex types of image used in this study. Parameters were not changed in this stage again since it was assumed that once the cGAN was run with the entire dataset and the appropriate number of epochs, the output images would be of a much higher quality. The lack of consideration for the parameters of the cGAN was likely one of

the reasons that when the CNNs were trained with datasets augmented with images generated from the cGAN the results suffered a drop-off. Much more consideration should have been put into optimiser selection, initial learning rate and other parameters of the cGAN, as many of these were left unchanged from the the cGAN's initial purpose to generate much smaller and less complex images. More consideration of these parameters would have allowed the cGAN to produce higher quality images that would improve a CNNs ability to classify skin lesions, rather than hinder it.

One of the main parameters that should be highlighted when considering why the cGAN produced poor results is the input image size. The images in the ISIC 2019 dataset are of high resolution (Cassidy et al. 2022). When training the cGAN and generating images, a possibility would have been to train and generate images with a resolution closer to that of the original ISIC 2019 dataset, using a higher resolution input size rather than 224 by 224 pixels. This would have required a different data loader, as the default data loader was for 224 by 224, and this would cause the runtime of the code to be much greater. However this would generate images of much higher quality which would likely capture more features than training with and generating lower resolution images. After this, the higher quality images could be loaded with the default data loader to be 224 by 224 pixels, to fit the CNNs input size. The resulting images would be of a higher general quality, but still the correct size for being inputted to the CNN models. These higher quality images would likely improve the ability of a CNN to classify skin lesions.

Another consideration for running this study again with a cGAN is the inconsistencies in the dataset; some images portraying rings, or foreign objects. Figure 4.16 shows that when the cGAN was run it created a ring in almost every image. By separating images into groups depending on whether or not they contain different foreign objects or black rings could improve performance of the cGAN.

6.3.2 Basic Augmentation

The logic behind the parameters used for basic data augmentations was similar to the logic used when choosing the parameters for the CNN and cGAN models in this study; they were chosen once at the start and were assumed to lack performance due to the constraints of the dataset size and epochs used in the development stage. Where the choice in parameters between the cGAN and basic data augmentations differ is in the fact that with generative data augmentation, the aim is to create images that resemble the real images, while with the basic data augmentation, there are orders of magnitude

of the extent to which images can be manipulated. For example, with noise injection, which was one of the worst performing data augmentations in this study, there are many degrees to which the data can be manipulated. An image can be injected with a relatively low amount of noise, such that the image only looks slightly different, or an image can be injected with vast amounts of noise, such that the image almost looks like random noise. The same goes for almost all of the basic data augmentations, where it is not a case of either applying the augmentation or not, but it is a case of to what extent the augmentation is applied.

This would explain why horizontal flipping was the only data augmentation to improve the performance of any one of the CNNs. This is because this data augmentation in particular is only a matter of on or off; this augmentation simply involves adding the exact same image to the dataset but flipped horizontally. There is only one way to flip an image horizontally, so there is no way to apply it incorrectly. All of the other data augmentations have ambiguity in how much the augmentation can be applied, and since each one of these lead to a decrease in performance, this leads to the likelihood that they were all initialised incorrectly for this task.

6.4 Analysing Graphs from This Study

Figures 6.1 and 6.2 show the confusion matrices and training and validation loss curves of no augmentation and translation applied to ResNet-50, which were the best performing augmentations for this model. These confusion matrices shows some amount of structure, but with a large number of errors. Also, the loss curves show signs of plateauing towards the end, which could imply that the models were stopping learning any new features, or it could mean that not enough epochs were run on the models with these augmentations. These loss curves show that no early stopping was triggered, meaning that there were not 10 successive epochs were the validation loss was not improved. It could be the case that allowing the models to run with more epochs could induce more learning rate decay, which would allow the models to learn finer details of the images. If the study were to be run again, models should be run with a higher number of total epochs, such that they allow for more features to be potentially learned. However, this could lead to over-fitting and reduce the overall performance of the model. The confusion matrices of these models also imply that class imbalance has had a large effect on the performance of models, as neither of these models predicted any of the testing data to be in class 3.

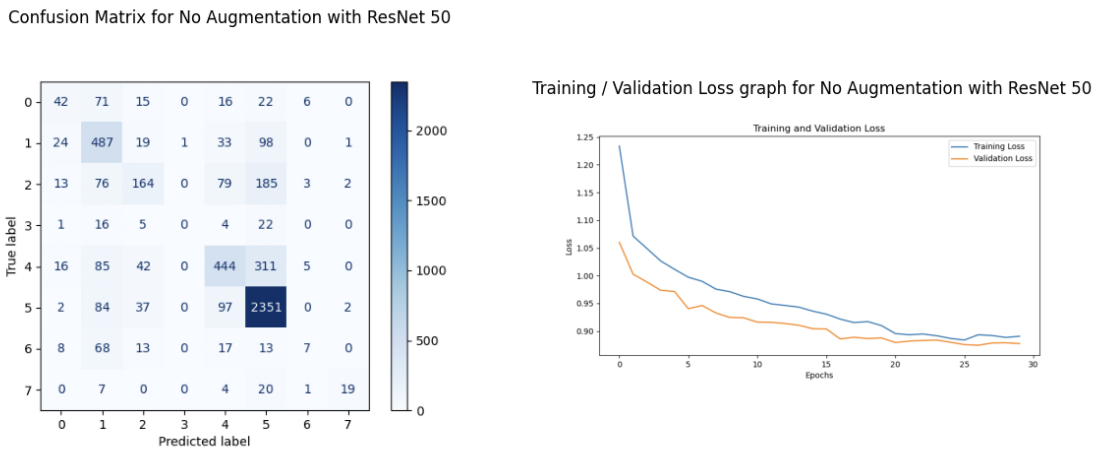

Figure 6.1: Confusion matrix and training and validation loss curves for no augmentation applied to ResNet-50

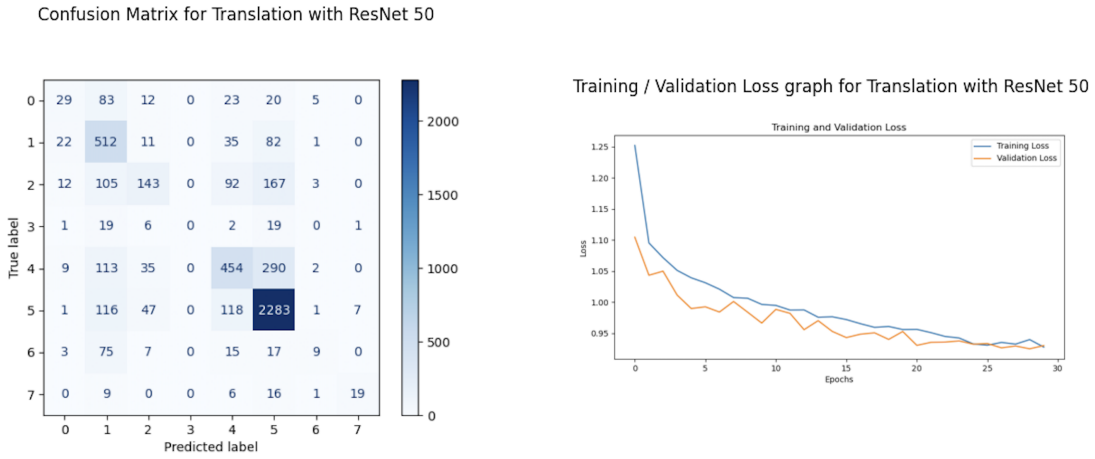

Figure 6.2: Confusion matrix and training and validation loss curves for translation applied to ResNet-50

Figures 6.4 and 6.3 show these graphs for horizontal flipping and no augmentation on DenseNet-121. These again were the two highest performing runs of this CNN architecture. These graphs show a similar issue to those seen in figures 6.1 and 6.2, where the models predicted almost zero images to be in class 3. The training and validation loss curves for these runs however lead further into the suspicion that models may not have been run for a long enough period of time, as the graphs show less sign of plateauing.

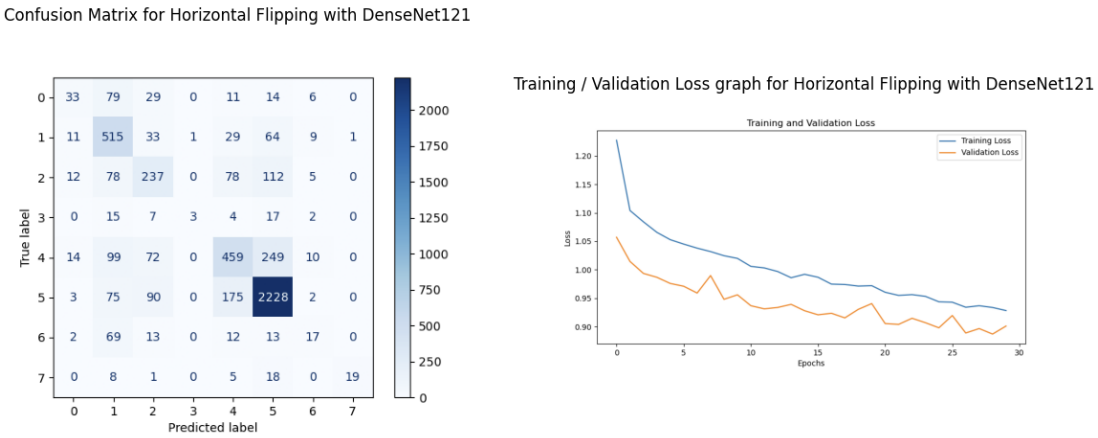

Figure 6.3: Confusion matrix and training and validation loss curves for horizontal flipping applied to DensNet-121

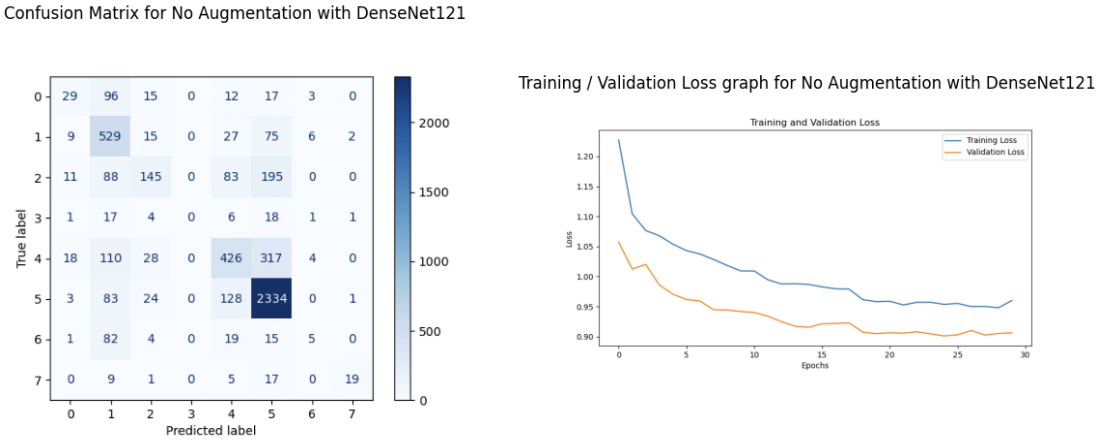

Figure 6.4: Confusion matrix and training and validation loss curves for no augmentation applied to DensNet-121

Figures 6.1 and 6.4 show the graphs for noise injection on both ResNet-50 and DenseNet-121. These graphs show that this augmentation in particular was not generalising at all to the training data and did not predict any images to be of any class other than the most dominant. This implies that this augmentation was applied incorrectly.

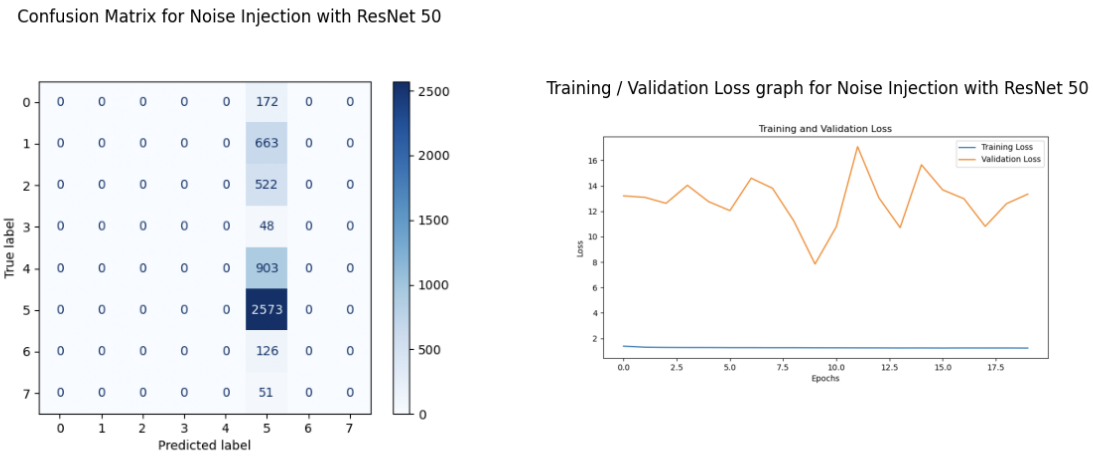

Figure 6.5: Confusion matrix and training and validation loss curves for noise injection applied to ResNet-50

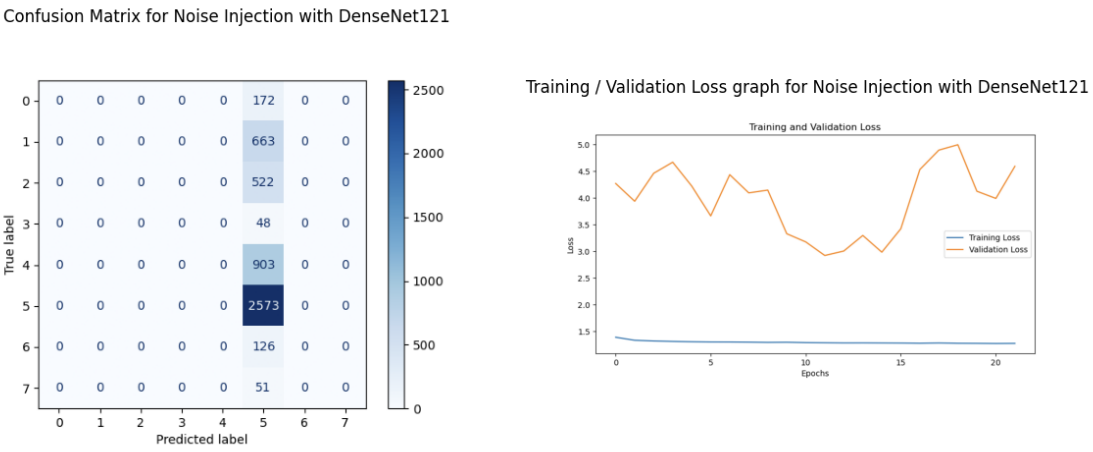

Figure 6.6: Confusion matrix and training and validation loss curves for noise injection applied to DensNet-121

6.5 Amended Execution of This Study

For this study to be run correctly, the issues with CNN model initialisation and all data augmentations must be addressed. First, by running multiple tests to find the optimal parameters for initialising the CNN models to classify skin lesions without any data augmentation applied to the training dataset. These CNN model parameters should remain constant throughout the rest of the study, this is so that the individual data augmentation methods can be measured with a consistent foundation. Also, running

any and all of these tests with more epochs, such that learning rate decay is triggered more often, or early stopping is triggered.

As for all of the basic data augmentations utilised in this study, except horizontal flipping, there should be multiple sub-tests to find their optimal parameters. Such as finding the optimal amount of noise injection to apply, the optimal amount of scaling to apply, or the optimal amount of Gaussian blurring to apply in order to produce the best results for each augmentation. In a similar manner, many sub-tests should be run with the cGAN in order to find the optimal parameters for producing images that improve the performance of each CNN model the most. Running similar tests to those used to find optimal CNN parameters to find the optimal model parameters for the cGAN. A larger input image size should also be used so that the images are generated with a higher quality, then made small enough to be fed into the CNN models. And the cGAN should be trained more carefully by separating image types from the training set, depending on if an image contains black rings or foreign objects.

It could be that these sub-tests will find that different parameters for data augmentations will produce optimal results for different models and different metrics. For example, one set of parameters for noise injection may cause a CNN to predict with optimal precision, at a sacrifice of recall, which would mean there are multiple sets of optimal parameters for noise injection, one set for optimising precision and another for recall. Or another set of parameters for a data augmentation may produce more favourable overall results on a DenseNet-121 model rather than a ResNet-50 model, which would mean there are optimal parameters for the given data augmentation depending on the CNN it is being applied to. This change means that there rather than showing the impact of implementing a data augmentation method, this test would show the potential that each data augmentation has to improve any one of the metrics.

So, there are a few changes that must be made to run this study again and gain results that would achieve the goal that was intended for this study. The main principle being optimal parameter selection for every element of this study. Optimal parameters for CNN model initialisation, for the cGAN and for the basic data augmentations. The optimal parameters for the ResNet-50 and DenseNet-121 models may differ from each other, but this does not matter as the results from the models are not being compared to each other. The data augmentation methods may have multiple sets of optimal parameters depending on the CNN they are applied to and the metric they aim to improve.

Overall, with these changes made to the methodology of the study, the quality of results should reflect that of other studies and provide the reference point for future

studies that was intended to be created from this study. There will also likely be more information provided as well, as there is a possibility that data augmentations will have multiple sets of optimal parameters, therefore multiple sets of optimal results.

This amended methodology would require more time and hardware resources. More time would be required due to the number of times that models must be run and run again to alter parameters in order to find the optimal settings. More hardware resources are also needed in order to facilitate the amount of time models are being run, as the IDE previously used for development would disconnect, so more access to a physical GPU would be required.

As for alterations to the code, this would not require much alteration as the code was developed to be reusable. Utilising relative file paths and placing almost all parameter options either at the top of the page or into clearly labelled areas, the code can be run on any machine and parameters can be changed easily. Other than this, the code trains, tests and resets models and generates results automatically. Changes would need to be made during development to find the optimal parameters, but running the study itself and recording the results would not require significant changes to the software. The re-usability of this code also leads to the possibility that the code can be used for examining the effect of augmentation in more contexts than just skin lesions. As long as data is organised in files in the correct way, and optimal parameters are known for certain models, then this code could be used for examining the effect of augmentation on any set of data.

Chapter 7

Conclusion

The overall aim of this study was to create a reference point for future studies which would show the individual impacts of data augmentation methods, which could be used when applying data augmentation methods in an assembly. Due to the significant difference in the performance of models compared with other studies, and the fact that the CNN models and data augmentation methods were not initialised properly, the reference point that this study shows is not deemed to be an appropriate reference point for future studies. Therefore, even though the impacts of each data augmentation method were shown, the results are not reliable and likely should not be taken into consideration by future studies in their current format.

The reason for the lacking results is mainly down to the resource constraints; the available time and hardware, but I do take responsibility myself for the oversights mentioned in Chapter 6, where not enough attention was given to parameter selection. However, given more of these resources, it would be possible to run the study with minor alterations to the software. Running the study with the changes outlined in this chapter would fulfil this study's goal of providing a reference point of the individual impacts of data augmentation methods. Also, the code produced for this study was of good quality and has potential to be easily utilised for a successful run of this study in future.

Bibliography

Aastha Agrawal, Karthik Nama Anil (Feb. 11, 2020). *Image-Augmentation-using-GANs*. Version 1. URL: https://github.com/KarthikNA/Image-Augmentation-using-GANs/blob/master/GANs%20-%20Complex%20Techniques/implementations/cgan/cgan.py.

Abedi, Masoud et al. (2022). "GAN-based approaches for generating structured data in the medical domain". In: *Applied Sciences* 12.14, p. 7075.

Almaraz-Damian, Jose-Agustin et al. (2020). "Melanoma and nevus skin lesion classification using handcraft and deep learning feature fusion via mutual information measures". In: *Entropy* 22.4, p. 484.

Awan, Abid Ali (2022). *A complete guide to data augmentation*. https://www.datacamp.com/tutorial/complete-guide-data-augmentation. Accessed: May 10, 2024.

Bhandari, Pritha (Mar. 2021). *Control Variable: Definition and Examples*. Published on March 1, 2021. Revised on June 22, 2023. URL: https://www.scribbr.com/methodology/control-variable/.

Bozkurt, Ferhat (2023). "Skin lesion classification on dermatoscopic images using effective data augmentation and pre-trained deep learning approach". In: *Multimedia Tools and Applications* 82.12, pp. 18985–19003.

Brinker, Titus Josef et al. (2018). "Skin cancer classification using convolutional neural networks: systematic review". In: *Journal of medical Internet research* 20.10, e11936.

Cassidy, Bill et al. (2022). "Analysis of the ISIC image datasets: Usage, benchmarks and recommendations". In: *Medical image analysis* 75, p. 102305.

Chen, Ting et al. (2020). "A simple framework for contrastive learning of visual representations". In: *International conference on machine learning*. PMLR, pp. 1597–1607.

Chlap, Phillip et al. (2021). "A review of medical image data augmentation techniques for deep learning applications". In: *Journal of Medical Imaging and Radiation Oncology* 65.5, pp. 545–563.

Codella, Noel, David Gutman, et al. (2017). "Skin Lesion Analysis Toward Melanoma Detection: A Challenge at the 2017 International Symposium on Biomedical Imaging (ISBI), Hosted by the International Skin Imaging Collaboration (ISIC)". In: *arXiv preprint arXiv:1710.05006*. arXiv: 1710.05006 [cs.CV].

Codella, Noel, Veronica Rotemberg, et al. (2018). "Skin Lesion Analysis Toward Melanoma Detection 2018: A Challenge Hosted by the International Skin Imaging Collaboration (ISIC)". In: *arXiv preprint arXiv:1902.03368*. arXiv: 1902.03368 [cs.CV]. URL: https://arxiv.org/abs/1902.03368.

Combalia, Marc et al. (2019). "BCN20000: Dermoscopic Lesions in the Wild". In: *arXiv preprint arXiv:1908.02288*. arXiv: 1908.02288 [cs.CV].

Contributors, Torchvision (2016). *Torchvision: Data Augmentation Library*. https://pytorch.org/vision/stable/transforms.html. Accessed: 2024-05-16. URL: https://pytorch.org/vision/stable/transforms.html.

Debelee, Taye Girma (2023). "Skin Lesion Classification and Detection Using Machine Learning Techniques: A Systematic Review". In: *Diagnostics* 13.19, p. 3147.

Deng, Li (2012). "The mnist database of handwritten digit images for machine learning research". In: *IEEE Signal Processing Magazine* 29.6, pp. 141–142.

European Parliament and Council of the European Union (2016). *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)*. Accessed: 2024-05-17. URL: https://eur-lex.europa.eu/eli/reg/2016/679/oj.

Fenza, Giuseppe et al. (2021). "Data set quality in machine learning: consistency measure based on group decision making". In: *Applied Soft Computing* 106, p. 107366.

Goodfellow, Ian et al. (2020). "Generative adversarial networks". In: *Communications of the ACM* 63.11, pp. 139–144.

Grandini, Margherita, Enrico Bagli, and Giorgio Visani (2020). "Metrics for multi-class classification: an overview". In: *arXiv preprint arXiv:2008.05756*.

Hadi, Muhammad Usman et al. (2023). "A lightweight CORONA-NET for COVID-19 detection in X-ray images". In: *Expert Systems with Applications* 225, p. 120023.

Hamida, Soufiane et al. (2023). "Data Balancing through Data Augmentation to Improve Transfer Learning Performance for Skin Disease Prediction". In: *2023 3rd International Conference on Innovative Research in Applied Science, Engineering and Technology (IRASET)*. IEEE, pp. 1–7.

He, Kaiming et al. (2016). "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778.

Höhn, Julia et al. (2021). "Integrating patient data into skin cancer classification using convolutional neural networks: systematic review". In: *Journal of medical Internet research* 23.7, e20708.

Huang, Gao et al. (2017). "Densely connected convolutional networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708.

Hubel, David H and Torsten N Wiesel (1968). "Receptive fields and functional architecture of monkey striate cortex". In: *The Journal of physiology* 195.1, pp. 215–243.

IEEE International Symposium on Biomedical Imaging (ISBI) (n.d.). *ISBI Challenges*. `https://biomedicalimaging.org/2017/challenges/`. Accessed: April 22, 2024.

ImageNet (n.d.). `https://www.image-net.org/`. Accessed: April 22, 2024.

ISIC Archive (n.d.). `https://isic-archive.com`. Accessed: April 22, 2024.

Kassem, Mohamed A, Khalid M Hosny, and Mohamed M Fouad (2020). "Skin lesions classification into eight classes for ISIC 2019 using deep convolutional neural network and transfer learning". In: *IEEE Access* 8, pp. 114822–114832.

Khan, Muhammad Attique et al. (2019). "Multi-model deep neural network based features extraction and optimal selection approach for skin lesion classification". In: *2019 international conference on computer and information sciences (ICCIS)*. IEEE, pp. 1–7.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25.

Kumari, Anita and Nonita Sharma (2021). "A review on convolutional neural networks for skin lesion classification". In: *2021 2nd International Conference on Secure Cyber Computing and Communications (ICSCCC)*. IEEE, pp. 186–191.

Le, Duyen NT et al. (2020). "Transfer learning with class-weighted and focal loss function for automatic skin cancer classification". In: *arXiv preprint arXiv:2009.05977*.

Li, Zewen et al. (2021). "A survey of convolutional neural networks: analysis, applications, and prospects". In: *IEEE transactions on neural networks and learning systems* 33.12, pp. 6999–7019.

Lobo, Jorge M, Alberto Jiménez-Valverde, and Raimundo Real (2008). "AUC: a misleading measure of the performance of predictive distribution models". In: *Global ecology and Biogeography* 17.2, pp. 145–151.

Lopez, Adria Romero et al. (2017). "Skin lesion classification from dermoscopic images using deep learning techniques". In: *2017 13th IASTED international conference on biomedical engineering (BioMed)*. IEEE, pp. 49–54.

Mazhar, Tehseen et al. (2023). "The role of machine learning and deep learning approaches for the detection of skin cancer". In: *Healthcare*. Vol. 11. 3. MDPI, p. 415.

Mohamed, Ensaf Hussein and Wessam H El-Behaidy (2019). "Enhanced skin lesions classification using deep convolutional networks". In: *2019 ninth international conference on intelligent computing and information systems (ICICIS)*. IEEE, pp. 180–188.

Nasr-Esfahani, Ebrahim et al. (2016). "Melanoma detection by analysis of clinical images using convolutional neural network". In: *2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. IEEE, pp. 1373–1376.

O'shea, Keiron and Ryan Nash (2015). "An introduction to convolutional neural networks". In: *arXiv preprint arXiv:1511.08458*.

Pedregosa, Fabian et al. (2011). "Scikit-learn: Machine learning in Python". In: *Journal of machine learning research* 12.Oct, pp. 2825–2830.

Perez, Fábio et al. (2018). "Data augmentation for skin lesion analysis". In: *OR 2.0 Context-Aware Operating Theaters, Computer Assisted Robotic Endoscopy, Clinical Image-Based Procedures, and Skin Image Analysis: First International Workshop, OR 2.0 2018, 5th International Workshop, CARE 2018, 7th International Workshop, CLIP 2018, Third International Workshop, ISIC 2018, Held in Conjunction with MICCAI 2018, Granada, Spain, September 16 and 20, 2018, Proceedings 5*. Springer, pp. 303–311.

Rashid, Haroon, M Asjid Tanveer, and Hassan Aqeel Khan (2019). "Skin lesion classification using GAN based data augmentation". In: *2019 41St annual international*

conference of the IEEE engineering in medicine and biology society (EMBC). IEEE, pp. 916–919.

Ratul, Md Aminur Rab et al. (2019). "Skin lesions classification using deep learning based on dilated convolution". In: *BioRxiv*, p. 860700.

Sahu, Priyanka et al. (2020). "Implementation of CNNs for Crop diseases classification: A comparison of pre-trained model and training from scratch". In: *IJCSNS* 20.10, p. 206.

Saturn Cloud (2024). *Data Augmentation with Generative AI*. `https://saturncloud.io/glossary/data-augmentation-with-generative-ai/`. Accessed: April 22, 2024.

Saxena, Divya and Jiannong Cao (2021). "Generative adversarial networks (GANs) challenges, solutions, and future directions". In: *ACM Computing Surveys (CSUR)* 54.3, pp. 1–42.

Shen, Shuwei et al. (2022). "A low-cost high-performance data augmentation for deep learning-based skin lesion classification". In: *BME frontiers*.

Shijie, Jia et al. (2017). "Research on data augmentation for image classification based on convolution neural networks". In: *2017 Chinese automation congress (CAC)*. IEEE, pp. 4165–4170.

Shorten, Connor and Taghi M Khoshgoftaar (2019). "A survey on image data augmentation for deep learning". In: *Journal of big data* 6.1, pp. 1–48.

Sun, Xiaoxiao et al. (2016). "A benchmark for automatic visual classification of clinical skin disease images". In: *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part VI 14*. Springer, pp. 206–222.

Tschandl, Philipp, Cliff Rosendahl, and Harald Kittler (2018). "The HAM10000 dataset, a large collection of multi-source dermatoscopic images of common pigmented skin lesions". In: *Scientific data* 5.1, pp. 1–9.

Villa-Pulgarin, Juan Pablo et al. (2022). "Optimized Convolutional Neural Network Models for Skin Lesion Classification." In: *Computers, Materials & Continua* 70.2.

Wardhani, Ni Wayan Surya et al. (2019). "Cross-validation metrics for evaluating classification performance on imbalanced data". In: *2019 international conference on computer, control, informatics and its applications (IC3INA)*. IEEE, pp. 14–18.

Wu, Yinhao et al. (2022). "Skin cancer classification with deep learning: a systematic review". In: *Frontiers in Oncology* 12, p. 893972.

Yagis, Ekin et al. (2021). “Effect of data leakage in brain MRI classification using 2D convolutional neural networks”. In: *Scientific reports* 11.1, p. 22544.

Yamashita, Rikiya et al. (2018). “Convolutional neural networks: an overview and application in radiology”. In: *Insights into imaging* 9, pp. 611–629.

Zhang, Chiyuan et al. (2021). “Understanding deep learning (still) requires rethinking generalization”. In: *Communications of the ACM* 64.3, pp. 107–115.

Zhang, Yiming and Chong Wang (2021). “SIIM-ISIC melanoma classification with DenseNet”. In: *2021 IEEE 2nd International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)*. IEEE, pp. 14–17.

Appendix A

Appendix A

```
[ ] df = pd.read_csv(copyof_csv_path) #load csv file

    #check if any image portrays more than one disease

    disease_df = df.iloc[:, 1:]

    multiple_diseases = disease_df[disease_df.sum(axis=1) > 1] #create directory for diseases which columns add up to more than one

    if not multiple_diseases.empty:
        print("Images portraying more than one disease:")
        print(multiple_diseases)
    else:
        print("No images portray more than one disease.")


    #check if any image portrays more than one disease

    no_diseases = disease_df[disease_df.sum(axis=1) > 1] #create directory for diseases which columns add up to less than one

    if not multiple_diseases.empty:
        print("Images portraying no diseases:")
        print(no_diseases)
    else:
        print("No images portray no diseases.")

    #NOTE: the folder does not contain any images portraying >1 or <1 diseases, so no function is written to remove these images

    No images portray more than one disease.
    No images portray no diseases.
```

Figure A.1: Data pre-processing: cleaning dataset of missing or incorrect data

removing duplicate images

function to find duplicate images through hash function

```
def find_duplicates(folder_path):
    duplicateImages = defaultdict(list) #create a list to store hash functions of duplicate images

    #iterate through all files in the folder
    for filename in os.listdir(folder_path):
        file_path = os.path.join(folder_path, filename)
        if os.path.isfile(file_path):
            try:
                with Image.open(file_path) as img: #open the image file
                    img_hash = hash(img.tobytes()) #calculate a hash for the image
                    duplicateImages[img_hash].append(file_path) #add the file name to the list of files with the same hash
            except Exception as e:
                print(f"Error processing {filename}: {e}") #exception is needed as there are some license files etc contained in the data

    duplicateImages = {img_hash: filenames for img_hash, filenames in duplicateImages.items() if len(filenames) > 1} #filter out unique images by only

    return duplicateImages
```

Figure A.2: Data pre-processing: removing duplicates by hash function (1/4)

function for showing file names of duplicate images

```
def show_duplicate_files(duplicate_files):
  for img_hash, filenames in duplicate_files.items():
        print(f"Hash: {img_hash}")
        print(f"Duplicate Files: {filenames}") #prints the contents of duplicate_files list
  print("no more duplicates") #until there are no more duplicates to show
```

function to delete the duplicate files from images folder

```
[ ] def delete_duplicates(duplicate_files):
      for img_hash, filenames in duplicate_files.items():
          for file_to_delete in filenames[1:]: #keep the first file, delete the rest
              os.remove(file_to_delete)
              print(f"Deleted: {file_to_delete}")
```

Figure A.3: Data pre-processing: removing duplicates by hash function (2/4)

function to remove the entries of duplicate images in the csv file

```
def delete_duplicates_from_csv(duplicate_files):
  duplicate_filenames = []
  for img_hash, filenames in duplicate_files.items():  #finding list of filenames to delete
        for file_to_delete in filenames[1:]: #keeping the first file name from duplicate_files, delete the rest
            filename = os.path.basename(file_to_delete).replace('.jpg', '') #removing .jpg from file name as this is not in the csv
            duplicate_filenames.append(filename)

  input_csv_path = copyof_csv_path #original csv

  output_csv_path = os.path.join(main_directory_for_all, "no_duplicates_ISIC_2019_Training_GroundTruth.csv")#new, clean csv

  open(output_csv_path, 'w').close() #deletes contents of output csv in case it has already been run and is populated

  with open(input_csv_path, mode='r', newline='', encoding='utf-8') as infile, \
     open(output_csv_path, mode='w', newline='', encoding='utf-8') as outfile: #open original csv to be read as infile and new clean

    reader = csv.reader(infile)
    writer = csv.writer(outfile)

    for row in reader:
        image_name = row[0] #image_name is first column of csv
        if image_name not in duplicate_filenames: #check that image_name isnt in list of duplicates
            writer.writerow(row) #add it to new csv
```

Figure A.4: Data pre-processing: removing duplicates by hash function (3/4)

finding the duplicates and deleting them from image folder + csv

```
folder_path = os.path.join(datasets_path, "ISIC_2019_Training_Input")
duplicate_files = find_duplicates(folder_path)
show_duplicate_files(duplicate_files)
delete_duplicates_from_csv(duplicate_files)
delete_duplicates(duplicate_files)
```

Figure A.5: Data pre-processing: removing duplicates by hash function (4/4)

```python
import os

def list_files(directory): #creates a list of all files in a folder
    files_list = []
    for root, _, files in os.walk(directory):
        for file in files:
            files_list.append(os.path.join(root, file))
    return files_list

def find_duplicates(folders):
    all_files = {} #function to hold all files
    duplicates = {} #function to hold file names that appear more than once

    for folder in folders: #every folder
        files = list_files(folder)
        for file in files: #every file in every folder
            file_name = os.path.basename(file)
            if file_name in all_files: #check if file name already exists in the dictionary
                if file_name in duplicates: #if it does, append the new path
                    duplicates[file_name].append(file)
                else:
                    duplicates[file_name] = [all_files[file_name], file] #if not, create new entry in duplicates
            else:
                all_files[file_name] = file #adds the file to the all files list if it isn't already in it

    return duplicates


folders = [os.path.join(split_data_path, 'test'), os.path.join(split_data_path, 'train'), os.path.join(split_data_path, 'val')]

duplicates = find_duplicates(folders)

if duplicates:
    print("duplicate files found:")
    for file_name, paths in duplicates.items():
        print(f"File name: {file_name}")
        for path in paths:
            print(f" --Located at: {path}") #shows any duplicate file names
else:
    print("No duplicate files found")
```

```
No duplicate files found
```

Figure A.6: Data pre-processing: verifying that there are no duplicate file names in any of the folders and sub-folders

```
means_path = os.path.join(unzipped_path, "normalisation_means_torch.pt")
stds_path = os.path.join(unzipped_path, "normalisation_stds_torch.pt")

means_content = torch.load(means_path)
stds_content = torch.load(stds_path) #getting means and STDs from zip file


normalize = transforms.Normalize(mean=means_content,
                                 std=stds_content) #normalises the data to the means and stds of the training data


data_transforms = { #basic data loding methods to resize the image etc so that it is readable by the model
    'train':
    transforms.Compose([
        transforms.Resize((224,224)),
        transforms.ToTensor(),
        normalize
    ]),
    'val':
    transforms.Compose([
        transforms.Resize((224,224)),
        transforms.ToTensor(),
        normalize
    ]),
    'test':
    transforms.Compose([
        transforms.Resize((224,224)),
        transforms.ToTensor(),
        normalize
    ]),
}
```

Figure A.7: Default data loading (1/2)

```
image_datasets = {
    'train':
    datasets.ImageFolder(os.path.join(currentDataset, 'train'), data_transforms['train']),
    'val':
    datasets.ImageFolder(os.path.join(currentDataset, 'val'), data_transforms['val']),
    'test':
    datasets.ImageFolder(os.path.join(currentDataset, 'test'), data_transforms['test']) #creating the image data

}

dataloaders = {
    'train':
    torch.utils.data.DataLoader(image_datasets['train'],
                                batch_size=32,
                                shuffle=True,
                                num_workers=0),
    'val':
    torch.utils.data.DataLoader(image_datasets['val'],
                                batch_size=32,
                                shuffle=False,
                                num_workers=0),
    'test':
    torch.utils.data.DataLoader(image_datasets['test'],
                                batch_size=32,
                                shuffle=False,
                                num_workers=0) #organising how the model will be fed the data (defining the data
}
```

Figure A.8: Default data loading (2/2)

```
[ ] def resetData():

        data_transforms['train'] = transforms.Compose([ #defining the transformations (no augmentation)
            transforms.Resize((224,224)),
            transforms.ToTensor(),
            normalize
        ])

        #update train data loader with no augmentations
        image_datasets['train'] = datasets.ImageFolder(os.path.join(currentDataset, 'train'), data_transforms['train'])
        dataloaders['train'] = torch.utils.data.DataLoader(image_datasets['train'],
                                                           batch_size=32,
                                                           shuffle=True,
                                                           num_workers=0)
        return image_datasets['train'], dataloaders['train']
```

Figure A.9: Reset training data to default

```
def initialize_model_ResNet():

        model_name = "ResNet50"

        device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        device #setting device as cuda

        model = models.resnet50(pretrained=True).to(device) #download model

        for param in model.parameters():
          param.requires_grad = True

        model.fc = nn.Sequential( #fully connected layer
            nn.Dropout(0.5),
            nn.Linear(2048, 256), #2048 input features reduced to 256
            nn.ReLU(inplace=True),
            nn.Linear(256, 8)).to(device) #setting layers, there are 8 classes in the dataset

        criterion = nn.CrossEntropyLoss() #setting criterion

        #stochastic gradient descent |/- with momentum set to 0.9
        optimizer = optim.SGD(model.fc.parameters(), lr=initLR, momentum=0.9)

        #ReduceLROnPlateau for learning rate scheduling
        scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=LRDecayFactor, patience=LRDecayPatience)

        return model, device, model.fc, criterion, optimizer, scheduler, model_name
```

Figure A.10: Initialise the CNN as ResNet-50

```python
def initialize_model_DenseNet(): #setup is the same as ResNet50, except layer configuration

    model_name = "DenseNet121"

    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    model = models.densenet121(pretrained=True).to(device)

    for param in model.parameters():
        param.requires_grad = True

    model.classifier = nn.Sequential( #classifier
        nn.Dropout(0.5),
        nn.Linear(1024, 256), #1024 input features reduced to 256
        nn.ReLU(inplace=True),
        nn.Linear(256, 8)
    ).to(device)

    criterion = nn.CrossEntropyLoss()

    optimizer = optim.SGD(model.classifier.parameters(), lr=initLR, momentum=0.9)

    scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=LRDecayFactor, patience=LRDecayPatience)

    return model, device, model.classifier, criterion, optimizer, scheduler, model_name
```

Figure A.11: Initialise the model as DenseNet-121

```python
def train_model(model, criterion, optimizer, num_epochs, early_stopping_patience = earlyStopPatience): #standard training loop
    train_losses = []
    train_accs = []
    val_losses = []
    val_accs = []

    best_val_loss = float('inf')
    epochs_no_improve = 0

    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch + 1, num_epochs))
        print('-' * 10)

        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()
            else:
                model.eval()

            running_loss = 0.0
            running_corrects = 0

            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)

                outputs = model(inputs)
                loss = criterion(outputs, labels)

                if phase == 'train':
                    optimizer.zero_grad()
                    loss.backward()
                    optimizer.step()

                _, preds = torch.max(outputs, 1)
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
```

Figure A.12: Training method (1/2)

```
            epoch_loss = running_loss / len(image_datasets[phase])
            epoch_acc = running_corrects.double() / len(image_datasets[phase])

            if phase == 'train':
                train_losses.append(epoch_loss)
                train_accs.append(epoch_acc)
            else:
                val_losses.append(epoch_loss)
                val_accs.append(epoch_acc)

                if epoch_loss < best_val_loss:
                    best_val_loss = epoch_loss
                    epochs_no_improve = 0
                else:
                    epochs_no_improve += 1

            scheduler.step(epoch_loss)

            print('{} loss: {:.4f}, acc: {:.4f}'.format(phase, epoch_loss, epoch_acc))


        if epochs_no_improve == early_stopping_patience: #stops training if validation loss doesnt improve for x num epochs
          print("Early stopping triggered")
          break

    return model, train_losses, train_accs, val_losses, val_accs
```

Figure A.13: Training method (2/2)

```
[ ] def evaluate_model_and_save_results(model_trained, dataloaders, device, base_directory, transformation_name):

        evaluation_results = [] #array to store results

        predlist = torch.zeros(0, dtype=torch.long, device='cpu') #predcted labels
        lbllist = torch.zeros(0, dtype=torch.long, device='cpu') #true labels

es      probs = torch.zeros(0, dtype=torch.float, device='cpu') #predicted probabilites
        actuals = torch.zeros(0, dtype=torch.long, device='cpu') #also stores true labels but is kept seperate to lbllist for debug

        with torch.no_grad():
            for inputs, labels in dataloaders['test']:
                inputs = inputs.to(device)
                labels = labels.to(device)

                outputs = model_trained(inputs)

                _, preds = torch.max(outputs, 1)

                #append batch prediction results
                predlist = torch.cat([predlist, preds.view(-1).cpu()])
                lbllist = torch.cat([lbllist, labels.view(-1).cpu()])

                #apply softmax to get probabilities
                probs_batch = outputs.softmax(dim=1)  #probabilities for all classes
                probs = torch.cat([probs, probs_batch.cpu()])
                actuals = torch.cat([actuals, labels.cpu()])

        #calculate accuracy
        accuracy = (predlist == lbllist).float().mean().item()

        #calculate precision, recall, specificity, and F1 score
        precision = precision_score(lbllist.numpy(), predlist.numpy(), average='weighted')
        recall = recall_score(lbllist.numpy(), predlist.numpy(), average='weighted')

        #calculate F1 score
        f1 = f1_score(lbllist.numpy(), predlist.numpy(), average='weighted')
```

Figure A.14: Testing method (1/2)

```python
#plotting and saving ROC and Confusion matrix
plot_roc_curve_and_save(actuals, probs, base_directory, transformation_name, 'ROC_AUC.png')

plot_confusion_matrix_and_save(actuals, predlist, base_directory, transformation_name, 'Conf_mat.png')

#store evaluation results
evaluation_results.append({
    'transformation': transformation_name,
    'accuracy': accuracy,
    'precision': precision,
    'recall': recall,
    'f1_score': f1,
})

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
print("="*30) #this just separtes lines in the console for readability

individual_field_names = ['transformation', 'accuracy', 'precision', 'recall', 'f1_score']

individual_csv_folder_path = os.path.join(base_directory, transformation_name)
individual_csv_file_path = os.path.join(individual_csv_folder_path, 'results.csv')

with open(individual_csv_file_path, mode='w', newline='') as file:
  writer = csv.DictWriter(file, fieldnames=individual_field_names)
  writer.writeheader()
  for result in evaluation_results:
      writer.writerow(result) #saving the results to csv
```

Figure A.15: Testing method (2/2)

Working out how many images each class to generate

```
[ ] train_directory = os.path.join(currentDataset, 'train')
    class_index_to_file_count = {}

    for class_name, class_idx in image_datasets['train'].class_to_idx.items():
        #counts the num of images in each class
        class_dir_path = os.path.join(train_directory, class_name)
        num_files = len([entry for entry in os.listdir(class_dir_path) if os.path.isfile(os.path.join(class_dir_path, entry))])
        #joins the count for each class to the class index
        class_index_to_file_count[class_idx] = num_files

    print(class_index_to_file_count) #debud

{0: 30, 1: 116, 2: 91, 3: 8, 4: 157, 5: 450, 6: 22, 7: 8}
```

Figure A.16: Working out number of images to generate for each class (cGAN)

```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
generator.eval()

GeneratedImgsFolder = os.path.join(GANsFolder, "Generated_imgs")
os.makedirs(GeneratedImgsFolder, exist_ok=True)

batch_size = GAN_GeneratingIMGs_BatchSize

#loop through each class
for class_id in range(opt.n_classes):
    num_images_per_class = class_index_to_file_count[class_id]
    class_name = idx_to_class[class_id]
    dir_path = os.path.join(GeneratedImgsFolder, class_name)
    os.makedirs(dir_path, exist_ok=True)

    i = 0
    while i < num_images_per_class:
        #determine num of images to process in this batch
        batch_end = min(i + batch_size, num_images_per_class)
        actual_batch_size = batch_end - i

        if actual_batch_size <= 0:
            break

        labels = torch.full((actual_batch_size,), class_id, dtype=torch.long, device=device) #generate labels for this batch

        noise = torch.randn(actual_batch_size, opt.latent_dim, device=device) #generate noise for the generator

        with torch.no_grad():
            generated_images = generator(noise, labels).detach().cpu()  #generate images with no gradient calculation

        #save each generated image
        for image_number, image in enumerate(generated_images, start=i):
            image_name = f"{class_name}_{image_number}.png"
            save_path = os.path.join(dir_path, image_name)
            save_image(image, save_path, normalize=True)

        print(f"Generated images for class {class_name} from {i} to {batch_end - 1}.") #print progress and update i for next batch start
        i = batch_end  # update i for next batch
```

Figure A.17: Generating images in batches

Creating a dataset of the original images plus the generated images

```python
GANsDataset = os.path.join(temporary_storage_path, "/Datasets/GANsDataset") #create a new dataset while running (technically temporary)
os.makedirs(GANsDataset, exist_ok=True)

originalDataset = os.path.join(currentDataset, 'train') #getting original training set

def copy_to_temp(source_folder, temp_folder):
    subdirs = os.listdir(source_folder)
    for subdir in subdirs: #for each subdirectory (class)
        source_dir = os.path.join(source_folder, subdir)
        temp_dir = os.path.join(temp_folder, subdir) #get path to the class from the source and path to class for the temp

        if not os.path.exists(temp_dir):
            os.makedirs(temp_dir) #create the temp directory for class if it doesnt exist

        files = os.listdir(source_dir) #get all the files from the original source

        for file in files:
            source_file = os.path.join(source_dir, file)# get file name of original image
            temp_file = os.path.join(temp_dir, file) #get the file name to be saved in new temp dataset

            shutil.copy(source_file, temp_file) #copy original for temp

copy_to_temp(originalDataset, GANsDataset) #copys all original train data to new dataset
copy_to_temp(GeneratedImgsFolder, GANsDataset) #copys all generated data to new dataset

print("folders combined")
```

Figure A.18: Creating new datasets with original training data plus cGAN generated data

Data loading for GANs dataset

```
#basically the same as the regular data load but with the merged dataset
def GANsDataLoad():
    data_transforms['train'] = transforms.Compose([
        transforms.Resize((224,224)),
        transforms.ToTensor(),
        normalize
    ])

    #update train data loader with the augmented transformations
    image_datasets['train'] = datasets.ImageFolder(GANsDataset, data_transforms['train'])
    dataloaders['train'] = torch.utils.data.DataLoader(image_datasets['train'],
                                                       batch_size=32,
                                                       shuffle=True,
                                                       num_workers=0)
    return image_datasets['train'], dataloaders['train']
```

Figure A.19: Data loader for cGAN generated data plus original training data

```
content_directory = os.path.join(data_and_graphs_path, 'Results')
os.makedirs(content_directory, exist_ok=True) ###setting up results folder

#names of models I'm using
modelNames = ['ResNet50', 'DenseNet121']

#different folders of transformation
transformationFolders = {
    'No_augmentation': [None],  #use None to represent no augmentation
    'GANs_augmentation': [None], #also none, because the generated data is 'manually' added to the dataset, not using a library function
    'Basic_augmentation_set_1': transformationsBeforeTensor, #transformations that need to be done before image is turned to tensor
    'Scaling_augmentation': transformationScaling, #also neeeds to be done before turned to tensor but needs to be in seperate folder (explained above)
    'Basic_augmentation_set_2': transformationsAfterTensor, #transformations that need to be done after the image is turned into a tensor

}

for modelName in modelNames: #for each model
  for transformationFolder, transformations in transformationFolders.items(): #for each folder of transformations
    for transformation in transformations: #for each transformation in that folder

      ##initialising model
      if modelName == 'ResNet50':
        model, device, model.fc, criterion, optimizer, scheduler, model_name = initialize_model_ResNet()
        print("ResNet reset")
      else:
        model, device, model.fc, criterion, optimizer, scheduler, model_name = initialize_model_DenseNet()
        print("DenseNet reset")

      ##creating directory to store results for specific model
      base_directory = os.path.join(content_directory, model_name)
      os.makedirs(base_directory, exist_ok=True)
```

Figure A.20: Code for the loop to run all models with all augmentations (1/2)

```
##outlining the name of the transformations
if transformationFolder == 'No_augmentation':
  transformation_name = 'No_augmentation'
elif transformationFolder == 'GANs_augmentation':
  transformation_name = 'GANs'
elif transformationFolder == 'Scaling_augmentation':
  transformation_name = 'Scaling'
else:
  transformation_name = transformation.__class__.__name__

##tells you whats happening
print("Training ", modelName, " with transformation:", transformation_name)

##reset the data to prevent data leakage
image_datasets['train'], dataloaders['train'] = resetData()

##only use this function to visualise the data augmented with basic techniques
if transformationFolder != 'No_augmentation' and transformationFolder != 'GANs_augmentation':
  visualize_augmented_data(image_datasets['train'], transformation, base_directory, transformation_name, means_content, stds_content)

#selecting which data load method to use, no_augmentation doesnt satisfy any of these so just uses the initial data from resetData()
if transformationFolder == 'GANs_augmentation':
  image_datasets['train'], dataloaders['train'] = GANsDataLoad()
elif transformationFolder == 'Basic_augmentation_set_1' or transformationFolder == 'Scaling_augmentation':
  image_datasets['train'], dataloaders['train'] = augmentDataBeforeTensor(transformation)
elif transformationFolder == 'Basic_augmentation_set_2':
  image_datasets['train'], dataloaders['train'] = augmentDataAfterTensor(transformation)



##training
model_trained, train_losses, train_accs, val_losses, val_accs = train_model(model, criterion, optimizer, num_epochs=CNN_Epochs)

##loss plot
plot_and_save_losses(train_losses, val_losses, base_directory, transformation_name, 'loss.png') #create loss plot for train val loss

##model evaluation
model_trained.eval() #switch to model testing

#calculates metrics and saves graphs
evaluate_model_and_save_results(model_trained, dataloaders, device, base_directory, transformation_name)
```

Figure A.21: Code for the loop to run all models with all augmentations (2/2)