FINAL YEAR PROJECT

Evaluating the Performance of EfficientNet for Synthetic Image Generation

---

Author:
Christopher ANTHONY MAYES

Supervisor:
Hooman OROOJENI

A thesis submitted in fulfillment of the requirements
for BSc Computer Science Degree

May 1, 2021

UNIVERSITY OF LONDON

Abstract

Computing Department

BSc Computer Science Degree

Evaluating the Performance of EfficientNet for Synthetic Image Generation

by Christopher ANTHONY MAYES

Synthetic image generation with Generative Adversarial Networks has had a huge impact in the development of generative modelling in deep learning. GANs have shown how deep neural networks can be used to create high dimensional, natural looking image samples with features learned from an input distribution. Despite their potential, there are still many concepts that are unexplored or outdated. This thesis aims to aid in the development of such models by researching and evaluating the application of uniform model scaling from the EfficientNet model to previous generative tasks to create more efficient, faster image generation models. This will be done primarily with the models' integration into a cycle consistent adversarial network. The performance will be evaluated by comparing the results to those from the 2017 CycleGAN [10] implementation, as well as assessing the visual outputs from the model on a unique Coke and Pepsi can dataset.

Acknowledgements

The acknowledgments and people to thank go here i.e. your supervisor…

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Generative Adversarial Networks are an outstanding innovation in deep learning and have the potential to produce very impressive results under many use cases. The applications of these networks have been increasing exponentially since their introduction by Goodfellow et al. [1] in 2014. The unlimited real world applications, including healthcare and image restoration, make them an important field to study and develop. A significant downside to these models is the computational costs associated with training when producing high quality outputs. The high computational costs result in very long training times, and investing in expensive hardware becomes essential to mitigate these drawbacks. The image generators and discriminators within a Generative Adversarial Network consist of several convolutional layers for down-scaling and feature extraction, transpose convolutions for up-scaling, as well as countless others depending on the given task. This results in deep neural networks often having millions of parameters, producing a complicated error landscape due to the considerable number of dimensions. The more parameters there are in a network, the more dimensions in the error surface, which results in longer convergence to a global minima. This task becomes harder every time a new dimension is added, as the distance between any set of given points is increased.

This project focuses on exploring a potential method of reducing the costs when training these networks and generating image outputs without forgoing quality. EfficientNet was chosen as the focus for this project due to its innovative concept of uniformly scaling all the dimensions of a convolutional neural network using a compound coefficient. By doing

so, the number of parameters should be significantly decreased, and with it, the training times. By replacing the ResNet architecture of the original CycleGAN implementation with that of EfficientNet, it will be clear as to the potential of this technology for generative tasks. Up to now, this method of model scaling has only been documented on classification problems, and not yet explored for the use of generation, which is why it has become the driving force for this project.

1.2 Project Specification

The goal of this research is to deliver a new method of performing image generation that is computationally efficient relative to existing models. It must provide accurately translated features from one dataset to another using a cycle consistent adversarial network. This network will be written in Python using Tensorflow, and tested against a baseline model derived from the CycleGAN [10] papers Residual Network [2] implementation. A set of generator models will be presented, each taking a different approach to the application of EfficientNet.

After creating the models, they will be tested on a dataset consisting of Coke and Pepsi can images with the intention of translating one to the other. For example, an image containing a Coke can should become the same image with a Pepsi can in its place. The performance of this translation will be evaluated using multiple metrics that utilise the Mean Absolute Error loss function, comparing both the translation from Pepsi to Coke, as well as the translated Pepsi back to the original Coke to fulfill the cyclic aspect of the CycleGAN. In addition, the performance of the network will also be monitored by visually evaluating the image outputs during and after training.

The project specification can be summarised into 3 main objectives:

- Investigate a range of past models and methods for image generation and translation, and propose an updated, more efficient model.
- Implement a modern solution to efficient image generation in Python with Tensorflow.

- Test the new models on an unseen dataset and evaluate the performance compared to existing solutions.

1.3 Report Structure

Chapter 2 - Presents the research and review of current technologies designed to tackle the problem of image generation. This is done through a brief exploration into the fundamentals of computer vision, followed by a more in depth analysis into some of the most popular and impressive applications of this technology in relation to this project.

Chapter 3 - An overview of important design choices that were made throughout the process of implementing the models. It covers how the data set was obtained and processed, including characteristics required for it to function in a cycle consistent adversarial network. In addition, performance tracking methods are described, with explanations of why they were chosen and how they work to improve training.

Chapter 4 - Presents the full implementation of the baseline model as well as each integration of EfficientNet for image generation. Following each implementation, the corresponding results, including the generated images, are displayed and evaluated.

Chapter 5 - Concludes the report with an analysis of the results obtained from each test, highlighting the successes and failures encountered throughout the project. In addition, the future direction for this project is explained.

Chapter 2

Literature Review

2.1 Computer Vision

Computer vision is a domain in which techniques are developed to give computers the ability to extract a high level understanding from digital images and videos.

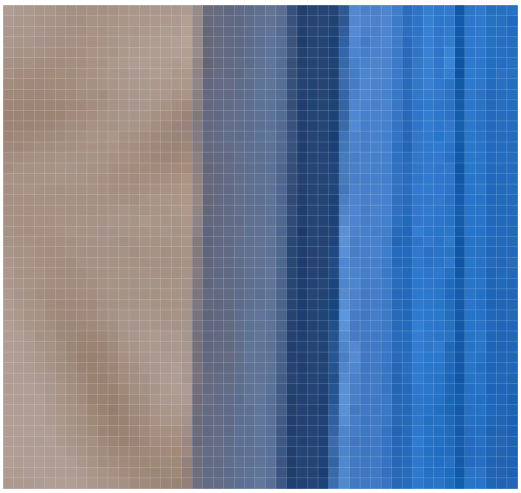


FIGURE 2.1: Pixel Region of Pepsi Can

Computer vision algorithms are used to consider small regions of an image, called patches. These patches contain the edges of objects which are inherently made up of many pixels. Using an example of a pixel region in figure 2.1, it is clear where the edge of an object starts because there is a changing colour that persists across many pixels vertically. This can be defined by a rule that states the likelihood of a pixel being a vertical edge is the magnitude of the difference in colour between some surrounding pixels.

The bigger the difference, the more likely the pixel is on an edge. The notation for this operation is as follows:

$$g(x,y) = w * f(x,y)$$

where $g(x,y)$ is the filtered image, w is the filter kernel and $f(x,y)$ is the original image.

$$kernel = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

This kernel contains the values for a pixel-wise multiplication, the sum of which is saved to the centre pixel. This kernel passes over the image, and from these calculations across all pixels, the network is provided with the location of all edges. There are a range of kernels that can be used to detect different features within a pixel space. The difference is the content of the kernel that is to be multiplied with the pixels of an image.

On the Pepsi can region from Figure 2.1, the can itself tends to be a very distinct blue colour that is rarely seen in any other part of the image. The edge of the can is also identified by a different shade of blue to surrounding regions.

2.1.1 Convolutions

The operation of applying a kernel to a pattern of pixels is called a convolution. These convolutions scan through an image by sliding a window across the pixel space. Each window can look for combinations of features indicative of a Pepsi can. Although each kernel is a weak feature detector on its own, when combined they are likely to detect the drinks can when a group of can like features are clustered together.

Detecting features in an RGB image increases the number of colour dimensions from 1 in a grey-scale image, to 3. To detect features in an RGB image, it must be convolved with a 3D filter, or a kernel with 3 layers that each correspond to the 3 colour channels. The number of channels must match between the image and the filter. To compute the output of such a convolution operation, the 3-dimensional kernel is slid across the 3-dimensional image in such a way that each of the numbers in the kernel are multiplied by the corresponding numbers from the red, green, and blue colour channels in the image. Adding all the numbers from this operation will provide the first position in the output matrix. This process is repeated as the kernel is moved along the image based on the stride parameter.

A stride of 1 will move the kernel one pixel to the right for each convolution. This allows the network to detect features specific to the colour channel as the contents of the kernel can be different for each.

The aforementioned convolutions are the same process that occurs in a convolutional neural network. The neurons are passed a set of pixel data, which is essentially the process of a convolution. The input weights are equivalent to kernel values, but instead of using a predefined kernel, the neural network learns its own values that can recognise a much larger range of features within an image. At each layer of neurons, more and more detailed features are extracted ranging from simple edges to complex shapes. These features are built up and processed by a layer that performs a convolution and extracts the final contents. To recognise complex objects, convolutional neural networks must contain many layers, and therefore many parameters which result long training times.

2.1.2 Max Pooling

Convolutional Neural Networks often use Max Pooling layers to reduce the size of the input sample to speed up computation, as well as make the features more robust. The output of a Max Pooling layer will be the maximum value from a specified filter size that is passed along a data set of values representing the features of an image. If a large number is detected in a filter as it is passing over the data, it is likely to indicate a detected feature. By extracting this number, we are removing redundant information and keeping the prominent features. In summary, so long as a feature is detected in one of the kernels, it remains preserved in the output of max pooling.

To calculate the output size of the data after Max Pooling, the following formula can be applied:

$$\frac{n + 2p - f}{s} + 1$$

where n is the input dimension, p is padding, f is kernel size, and s is stride.

Max Pooling is a valuable addition to convolutional neural networks because it has no learnable parameters for gradient descent, and therefore can only make the network more efficient if used appropriately.

2.1.3 Depthwise Separable Convolutions

Depthwise separable convolutions are an alternative method to standard convolutions which require less computation power and parameters. The concept of the standard convolution is explained in detail in section 2.1.1. The cost of a convolution can be calculated by the number of multiplications required. For a single convolution operation, the number of multiplication operations is equal to the number of elements in a given kernel, or $((kernelwidth * kernelheight) * channels)$. That kernel is then multiplied by the passes over the input, and again by the number of kernels. In standard convolutions, the application of kernels across all input channels and the combination of these values are done in a single step. Depthwise Separable Convolutions break this down into two parts, depthwise convolution, followed by pointwise convolution. In the depthwise stage, convolution occurs on the channels separately by using 3 kernels which are then stacked to recreate a colour image. This output then passes through pointwise convolution to increase the number of channels of each image to match the output of a standard convolution. This is done by iterating over the output from the depthwise stage with a 1 x 1 x *channels* kernel which produces a matrix of size *kernelwidth* x *kernelheight* x 1. This is done multiple times to create an output of *kernelwidth* x *kernelheight* x *desiredchannels*. Performing Depthwise Separable Convolution can reduce the number of multiplications from millions to tens of thousands, allowing the network to process more in less time. It is possible that due to the reduction in parameters, the network might fail to learn effectively during training but when used correctly, should have a positive effect on the efficiency.

2.2 Image Generation Networks

2.2.1 Generative Adversarial Networks

Generative Adversarial Networks were developed by Ian Goodfellow [1] in 2014. They are a type of generative network that trains two models simultaneously. These two models are a generator that captures details from a data distribution, and a discriminator that estimates the probability that an input sample is real or fake. These two models learn through a game theoretic formulation which can otherwise be interpreted as a two player game where the two adverserial players are the generator and discriminator networks.

The training procedure for the generator is to maximise the probability of the discriminator making a mistake, so the generator will try to fool the discriminator by generating as close to realistic looking images as possible. The model takes a unique approach of training from discriminative power rather than from an input distribution. The discriminator tries to discern whether or not the input image is a real image from the training set or a fake image output by the generator model. At the time of the GANs introduction, image classifiers were already a well explored and very powerful concept. The underlying idea behind the creation of the GAN was to harness this already very capable technology, and use it for training an image generator that would provide clearer outputs relative to previously existing networks.

The two models play a minimax game with value function $V(G, D)$ where G is the generator and D is the discriminator.

$$\min_G \max_D V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

This loss function calculates the log probability of input data and the log probability of 1 - input data. There is a prior distribution of noise z which is the input to the generator. As it is a non stochastic function, random noise is used as the input because it needs to produce a new data point for every iteration. This noise is fed into the generator and used to produce an output. This output is passed into the discriminator which is trying to maximise the probability of real data and minimise the probability of fake data, which is a standard classification problem. At the same time, the generator is trying to minimise the value function, or the best possible discriminator. The discriminator is used in both terms, however the generator is used only in the second as it has one objective - to make the discriminator class fake data as real. Splitting the function up into its two terms, the first is the discriminator trying to classify real data as real which remains constant throughout. In the second it is trying to classify fake data as fake, however, at the same time, the generator is minimising this output to trick the discriminator into classifying fake data as real.

2.2.2 StyleGAN

The StyleGAN [5] is generative adversarial network with a style-based generator architecture that allows for precise control over the synthesis of the image that is generated. This

means that it can clearly separate the features of the input distribution and use them as 'styles' that can be weighted to change the output of the network. These outputs are of very high quality and diversity in terms of features. Generating images with a standard generative adversarial network provides no ability to control the output as the input is only a latent noise vector.

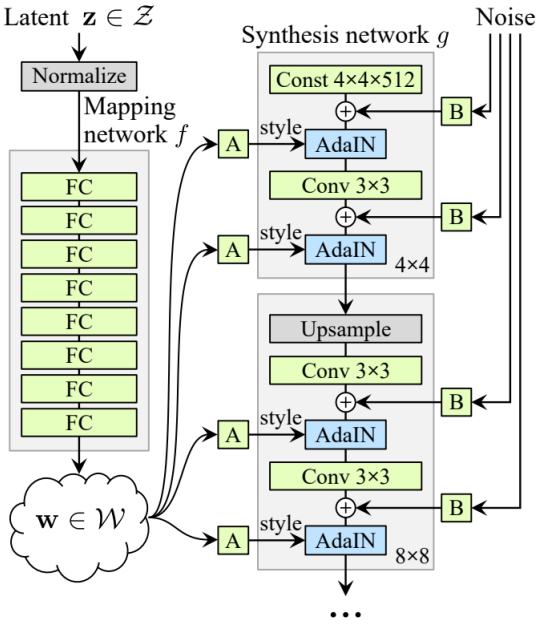


FIGURE 2.2: Architecture of StyleGAN network

The StyleGAN architecture achieves the additional customisation and quality through the following:

- Introduction of a mapping network. This takes the input noise vector and maps it to another latent vector w with a multi-layer perceptron network consisting of 8 layers used to generate the styles
- Using noise vectors throughout the network at convolution stages rather than just one at the input
- Adaptive Instance Normalization

Adaptive Instance Normalization

Adaptive instance normalisation was derived from batch normalisation. Batch normalisation computes the mean and standard deviation of input data x, and has parameters for

translation factor and scale factor. Instance normalisation computes the same calculation but for a specific instance of x within a batch, so each sample can be treated separately.

$$\text{AdaIN}(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)} + \mathbf{y}_{b,i},$$

Adaptive instance normalisation takes this one step further by taking the same input x and an additional y where x are the features of the previous layers convolutions and y is comprised of y_s, the scale factor, and y_b, the translation factor, which are taken from the learned vector w. Because y defines the scale and the transitional features, the network has control over the style of the output images.

Mixing Regularisation

The StyleGAN also uses a mixed regularisation technique for the noise inputs that improves the quality of generated images further by decorrelating neighboring styles and enables more fine-grained control over the generated imagery. Instead of passing one latent noise vector z as input and getting one vector w as output, at least two inputs are given to produce two outputs $w1$ and $w2$. These are then passed randomly for each iteration in pairs to the adaptive instance normalisation layers.

2.2.3 Cycle Consistent Adversarial Networks

A cycle consistent adversarial network, or CycleGAN [10], is used for image to image translation. Previous methods required the used of paired image data and as a result the models had to be trained on both the original image, and the corresponding target image after translation. These methods assume it is possible to create a data set of original and target images which is not always true. This type of network became the focus for this project because unless developed for a given task, it is better to assume that not all real world applications for image translation will come with a data set of targets, and therefore will have a wider range of applications. In addition, creating these data sets is difficult and may not contain enough samples to generate meaningful outputs. The CycleGAN uses unpaired image data sets x and y, where we assume there exists a mapping between the two.

Loss Functions

The goal is to train a model that learns this mapping G while preserving some aspects of the original image. This is done by using an adversarial loss that minimises the difference between the input sample and the generated sample. Due to the lack of paired data, the chance of learning a meaningful mapping with a standard generative adversarial network is very low. In order to reduce the number of possible mappings a second loss is introduced, the cycle consistency loss. This additional mapping F is the inverse of G, which ensures the translated image can be converted back to the original with minimal loss. This additional mapping means the network must train two GANs where each has a generator and discriminator.

The cycle consistency loss is defined by the addition of the l1 norm of the two adversarial loss functions, one for G and another for F.

$$\begin{aligned}\mathcal{L}_{\text{cyc}}(G, F) = &\mathbb{E}_{x \sim p_{\text{data}}(x)}[\|F(G(x)) - x\|_1] \\ &+ \mathbb{E}_{y \sim p_{\text{data}}(y)}[\|G(F(y)) - y\|_1]\end{aligned}$$

There exists both a forward cycle consistency for establishing when the sample input x matches its transformation after applying G and F, and a backward cycle consistency established when a sample image from y matches the output after applying F and then G in succession.

Generator Architecture

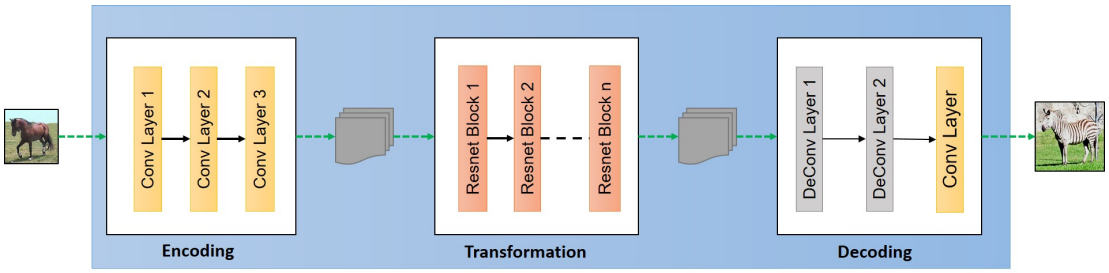


FIGURE 2.3: Architecture of CycleGAN generator

The generators consist of 3 sections; the encoder, transformation, and decoder. The encoder block is a set of convolutional layers that takes an image input and outputs a feature volume. The transformer takes the feature volume and passes it through a set of residual blocks, each of which is a set of two convolution layers with a bypass. The bypass allows

the transformation of previous layers to be retained throughout the network which allows for deeper networks. The decoder performs the opposite operations of the encoder and outputs a generated image. This is done using transpose convolutions to rebuild from the low level extracted features.

2.3 Efficient Networks

2.3.1 Residual Networks

Residual networks [2] are one of the most popular deep learning architectures that has been published. As previously explained, deep learning is the process of extracting and learning features that become more detailed as the network becomes deeper. However, building a better network is not analogous with adding more layers as performance will eventually decline due to vanishing and exploding gradients. Residual Networks use skipped connections, which is the process of taking the activation from one layer and feeding it directly into a deeper layer. Rather than an activation having to pass through a group of linear functions, it can be passed straight into a later non-linear function allowing for much deeper networks while continuously decreasing the training error. With the addition of skipped connections, adding more layers to a network doesn't have such a negative affect on the performance. It becomes easier to learn an identity function that maps two activations because the network is only learning the residual rather than the true output. Without this, a deep plain network will struggle to choose parameters to learn something as simple as an identity function in the deeper layers, which results in reduced performance.

2.3.2 EfficientNet

EfficientNet [8] introduces a modern computer vision technique that achieves very impressive results in a number of tasks including image classification and object detection. It explores a new approach of scaling convolutional neural networks to improve accuracy and efficiency. Previously, a common way to do this would be to arbitrarily add more layers or kernels, however this technique aims to systematically scale up the width, depth and resolution of a convolutional neural network.

Uniform Model Scaling

Width, depth, and resolution are the three dimensions considered when scaling up convolutional neural networks. Width scaling refers to adding more feature maps to each layer, depth refers to the addition of more layers, and resolution is increasing the resolution of the input image. Scaling each of these separately will improve the accuracy up to a point, but eventually begins to saturate. The uniform scaling method is introduced to balance the up-sampling of the three dimensions through the use of a constant ratio consisting of α, β, and γ. These terms are exponenciated by phi, which denotes the increase in computational resources to the network, with a constraint that $\alpha * \beta^2 * \gamma^2$ should equal roughly 2.

$$\begin{aligned} \text{depth: } & d = \alpha^\phi \\ \text{width: } & w = \beta^\phi \\ \text{resolution: } & r = \gamma^\phi \\ \text{s.t. } & \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2 \\ & \alpha \geq 1, \beta \geq 1, \gamma \geq 1 \end{aligned}$$

FIGURE 2.4: Compound Scaling for EfficientNet

The constrained parameters are found using a grid search on a small baseline model. These parameters can then be scaled up by the calculated parameters to create larger, more accurate networks. It is understood that a network with a large input image requires more layers to increase the receptive field, and more channels to extract the finer patterns on the larger image. In order to generate high resolution synthetic images, it is important to implement these modern techniques.

2.3.3 Squeeze-and-Excitation Networks

Squeeze and excitation blocks were introduced by Hu et al. (2017) [3] and were designed to improve channel inter-dependencies in convolutional neural networks with minimal additional computational cost. This involves modifying the parameters of each channel separately so the network can adjust the weights of each feature map to aid in the efficiency of feature extraction. The feature maps are squeezed into a single numeric value using global average pooling, and a vector equal to the size $1x1xChannels$ is returned. This squeezed output is fed through a fully connected multi-layer perceptron network with

two layers, one of which is a bottleneck that compresses the input based on a specified ratio and adds non-linearity. This is where the weight representations are learned which are used to modify the dependencies of each channel. The final fully connected layer expands the compressed vector back to its original input dimensions allowing the network to update the weights of the most important channels. This is done by an element-wise multiplication between the weight vector and the input x. This process adds a negligible amount of computational resource and can be used in any convolutional neural network. These blocks were integrated into ResNet-50 which provided almost the same accuracy as ResNet-101. The authors of the EfficientNet model also applied these squeeze and excitation blocks to further increase performance and reduce network cost.

Chapter 3

Methodology

This chapter details specific design choices that were made throughout the implementation process. The first section covers how the data set was gathered and how it is used to produce meaningful outputs during the testing stage. This touches on the key characteristics of the data set and explains how the samples are pre-processed for use in the network. Following this, the methods and metrics used for tracking the performance are covered. Full results are detailed in chapter 4 as they are used to evaluate how effective the network is at generating new images. Finally, some visual methods of tracking the networks' performance are explained as they must be used to reach a meaningful conclusion.

3.1 Data collection and data set

3.1.1 Collection

It was important to compile a data set that would effectively test the performance of the network. Due to this being an image translation task, it was important to find two sets of samples that contain an object of similar shape so the pattern could be learned and translated between the two. The Coke and Pepsi can data sets were chosen for this reason. A drink can remains constant in shape, and the prominent colour makes it easier for the network learn during the feature extraction stage. The majority of image samples were obtained from google image search using a chrome extension to download all images being displayed, or free stock photo sites such as Unsplash.

It proved a challenge to collect an acceptable sample size through this method so a small

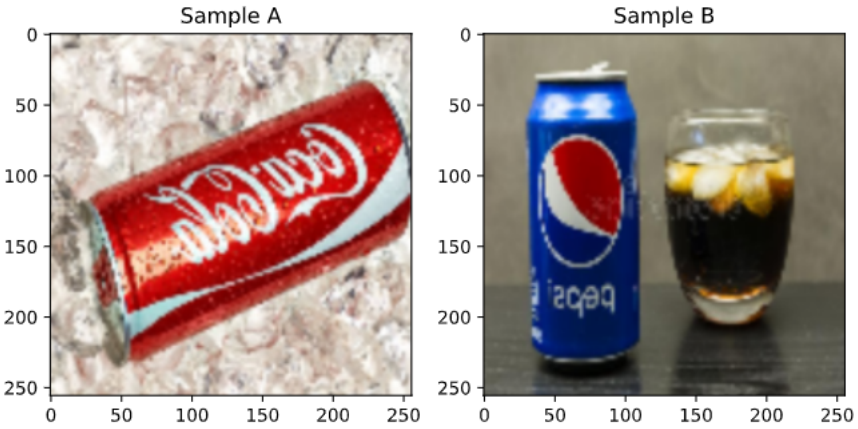


FIGURE 3.1: Data set sample images

additional set of images was obtained by taking pictures of the cans in a range of environments. These techniques resulted in a data set of Coke images consisting of 300 samples, and a data set of almost 200 Pepsi images. They each had to be pre-processed to avoid the network generalising to the dataset which would result in poor performance on unseen data.

3.1.2 Pre-processing

The raw image data had to be converted into a format that would work in Tensorflow which meant it needed to be resized and placed into a Tensorflow data set. This was done in python by first reading the image folder path with the os library, importing, recolouring, and resizing each image with the cv2 library. These were then appended to python lists, shuffled, and finally converted to numpy arrays to be exported using the np.save function. The numpy arrays were loaded into the CycleGAN model file and used to create tensorflow datasets with tf.data.Dataset.from_tensor_slices.

To avoid overfitting, the samples were pre-processed before being fed into the model. The original pixel values of the images range from 0 to 255 which is not ideal, so the values are normalised to between -1 and 1. The network would otherwise have to process larger weight values which leads to slower convergence and poorer performance. Random flips and crops are applied to each sample to increase the variation of images, ensuring each is unique and covers a wider range of circumstances. These techniques are also used to increase the number of samples as it proved difficult to produce a large enough, varied data set from just downloaded images.

3.2 Performance Tracking

The goal of this project is to produce an efficient model for the task of generating synthetic images. To do so, a number of metrics need to be tracked to test if the new implementations are worse than, or surpassing the recorded baseline. The following loss metrics and methods are being used to compare EfficientNet's model scaling generator to that of the ResNet CycleGAN.

3.2.1 Generator and Discriminator Loss

The loss objective for the generator and discriminator networks is calculated using binary cross entropy which is simply a binary classification task. A Tensorflow function is being used which utilises a sigmoid activation to predict the probability of an image being real or fake. It computes the log probability of a given class existing through the following function:

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

This function is minimised by punishing wrong predictions. If the output should be 'real' and the prediction is 'real', the cost will be 0. In contrast, if the output should be 'real' and the prediction is 'fake', the network is punished with a large cost. This process refers to the term entropy which is used to measure the performance of a classification algorithm.

The CycleGAN takes a unique approach to training its image generators by trying to maximise the probability of the discriminator making a mistake. The discriminator then tries to discern whether or not the input image is real or fake with the same loss objective.

```
loss_obj = tf.keras.losses.BinaryCrossentropy(from_logits=True)
def generator loss(generated):
    return loss_obj(tf.ones_like(generated), generated)
```

3.2.2 Mean Absolute Error

Mean absolute error computes the difference between the mean of all absolute values across the dimensions of the real input image and the generated input image. The formula takes the difference between two absolute values and is divided by the total number

of samples.

$$\text{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$$

In this case, it is being used to measure the distance between the pixel values of two inputs to inform the network if updates are having a positive or negative effect after every epoch. This ensures that the generated images maintain a similar structure to the real samples. An output of 0 means there has been no change between the two input tensors. When converting image X→Y, the MAE is expected to be greater than 0 due to the style change, but should be 0 when converting back to input X which is how the cycle consistency loss is calculated.

```
def mae(real_image, cycled_image):
    loss1 = tf.reduce_mean(tf.abs(real_image - cycled_image))
    return LAMBDA * loss1
```

3.2.3 Cycle consistency Loss

The cycle consistency loss exists for both GANs in the network. The first generator maps image X to Y, and the second maps image Y back to the original X. Cycle consistency expects that after passing an image through both generators, the output should look exactly like the original because only the styles should have changed. The loss is calculated by taking the pixel difference between the two images and adding to the total loss function. This can be done starting at either the X or Y image, and the cycle consistency loss should take both into account. That is, the sum of both the X→Y→X translation and the Y→X→Y translation.

3.2.4 Identity Loss

Identity loss is an optional loss term for generative adversarial networks used to preserve the colour outputs in the generated images. It ensures that when passing an image into its opposite translation generator, it will output the same image as input because it is already of that class and set of styles. Using the mean absolute error, it applies an identity mapping that means there should be no change to the input. It calculates the pixel distance between the input and the generated output and is added to the overall generator loss function.

An identity loss of 0 is ideal as it means the pixels of the image haven't changed. If the generator changes the image significantly in some way, like a complete colour change, the loss will be greater than 0 and should discourage the network from continuing this mapping. The identity loss is made up of both the pixel distance between Pepsi to Coke and the pixel distance of Coke to Pepsi combined together.

3.2.5 Optimiser

An optimiser is used during the training process to ensure the loss functions reach their global minima. This implementation will update its parameters using the ADAM Optimiser, developed by Diederik P. Kingma and Jimmy Ba in 2014 [6] specifically for use in deep neural networks. ADAM refers to adaptive moment estimation and makes use of both momentum and RMSProp for balancing the updates, allowing traversal of complicated error terrains. For each parameter update step, the expected value of past gradients is added, so the initial updates are slow but gain momentum over time. It is able to make unique updates for different parameters so they can each be updated separately, leading to faster convergence. This is valuable in an image generation network due to the large number of trainable parameters from convolutions.

3.3 Visual Outputs

3.3.1 Tensorboard

Tensorboard is a tool offered by Tensorflow that can be used for tracking and comparing network performance based on given metrics. This is being used to evaluate the effectiveness of each version of EfficientNet implementation as well to compare them against the ResNet baseline. The graphs update automatically during training and are each saved in a separate file to be loaded again any time. This method of tracking losses proved far more efficient than generating graphs directly in the python notebook due to the volume of testing that was being done. Through the process of building the model, many iterations were tested and compared and could all be displayed on a single graph with Tensorboard.

3.3.2 Generated Images

The performance of each model is also tracked through the images that are generated. For each epoch, the training loop displays both the generated image and the cycled image which are used to monitor how well the network is learning. Simply monitoring the outputs can be one of the most effective methods of evaluating the performance of a generative task as the primary objective is to produce realistic images.

Chapter 4

Implementation and Results

4.1 Establishing a baseline

4.1.1 Design and Implementation

In order to establish a reliable baseline for comparison, the residual network and cycle consistent adversarial network needed to be constructed first. Residual Networks use skipped connections, which take the activation from one layer and feeds it directly into a later layer allowing for much deeper networks while continuously decreasing the training error. A residual block function was created consisting of two sets of the following layers; Reflection Padding, 2D convolution, Instance Normalization [9] and a ReLU activation. The input image is stored in a variable called *input_tensor* to be used for the skipped connection.

```
def resnet_block(res_layer):
    features = input_layer.shape[-1]
    func_input = input_layer

    res_layer = ReflectionPad2D()(input_layer)
    res_layer = layers.Conv2D(features, kernel_size=(3, 3), strides=(1, 1), padding="
        valid", use_bias=False)(res_layer)
    res_layer = tfa.layers.InstanceNormalization()(res_layer)
    res_layer = layers.Activation("relu")(res_layer)
    res_layer = ReflectionPad2D()(res_layer)
    res_layer = layers.Conv2D(features, kernel_size=(3, 3), strides=(1, 1), padding="
        valid", use_bias=False)(res_layer)
    res_layer = tfa.layers.InstanceNormalization()(res_layer)
    output = layers.add([input_layer, res_layer])

    return output
```

This block can be called multiple times and stacked to create the residual network. Tensorflow and keras do not have an inbuilt function that achieves reflection padding on its own so is implemented through a class that can be called as a layer in any other function. This class takes a layer as input which provides the function with a tensor. The padding is calculated in the call function and is applied and returned in the tf.pad function to the input tensor. This class was inspired by Keras implementation.

```
def __init__(self, padding=(1, 1), **kwargs):
    self.padding = tuple(padding)
    super(ReflectionPad2D, self).__init__(**kwargs)

def call(self, matrix_in):
    w, h = self.padding
    additional_padding = [[0, 0], [h, h], [w, w], [0, 0]]
    return tf.pad(matrix_in, additional_padding, mode="REFLECT")
```

Reflection padding uses the contents of the image matrices for the padding values by reflecting the rows for use as a border. This ensures that the outputs will have smooth transitions into the padding rather than using values that may create a harsher edge. The following convolutional layer is used to extract features from the input image which become more detailed and specific as more blocks are stacked, but is done at the cost of increased parameters. Instance normalisation [9] is used to speed up training. The parameters of the network are normalised by computing the layers mean and standard deviation for each individual channel and sample. By normalising the layers, the inputs to the next layer will be computed faster, resulting in decreased training times. A ReLU activation is used to increase the non-linearity in the input sample after previously applying linear operations because images consist of non-linear features. These layers are applied a second time, however the final layer of the residual block adds the saved input tensor to the output of the previous layer for the integration of the skipped connection.

These residual blocks are called in the ResNet generator as many times as required between a set of down-sampling and up-sampling layers for encoding and decoding the inputs. The down-sampling block consists of two sets of 2D convolutions, Instance Normalisation and ReLU activation. The 2D convolution extracts progressively higher levels of features, compressing features that might be spatially distant, but are related in the image, and transforms it into a tensor of shape $(64, 64, 128)$. The instance normalisation

and ReLU activations have the same use as previously described in the Residual Block explanation, to ultimately increase performance of the network.

```
x = layers.Conv2D(64, kernel_size=(3,3), strides=(2,2), padding="same", use_bias=
    False)(x)
x = tfa.layers.InstanceNormalization()(x)
x = layers.Activation("relu")(x)
x = layers.Conv2D(128, kernel_size=(3,3), strides=(2,2), padding="same", use_bias
    =False)(x)
x = tfa.layers.InstanceNormalization()(x)
x = layers.Activation("relu")(x)
```

After encoding the input sample, the style transformation to the that of the opposite data set takes place in the residual network blocks.

```
for i in range(num_resnet_blocks):
    x = resnet_block(x)
```

These blocks capture the distinct features of the input samples, which is the red and white of the Coke can or blue and red of the Pepsi can. These features are learned by the model as they are a constant throughout both data sets. These blocks are used to train the final network when generating images that minimise the losses to produce an output of matching style to the input.

The image must be scaled back to the original input dimensions in the up-sampling block to produce an output. This has a similar construction to the decoder, but uses transpose convolutions for up-scaling. They perform the same operations as a standard convolution but in reverse. A learned filter is passed over the input and the values are multiplied together to produce an output of increased dimensions. This concludes the primary function of the ResNet generator, and the function is returned to be called in the cycle consistent adversarial network python notebook.

```
x = layers.Conv2DTranspose(64, kernel_size=(3,3), strides=(2,2), padding="same",
    use_bias=False)(x)
x = tfa.layers.InstanceNormalization()(x)
x = layers.Activation("relu")(x)
x = layers.Conv2DTranspose(3, kernel_size=(3,3), strides=(2,2), padding="same",
    use_bias=False)(x)
x = tfa.layers.InstanceNormalization()(x)
x = layers.Activation("relu")(x)
```

The discriminator network is defined in the *get_discriminator* function and is used in the training step to predict whether the image inputs are real or generated, and updates the loss functions accordingly. It uses a PatchGAN architecture introduced in the 'Image-to-Image Translation with Conditional Adversarial Networks' paper [4]. This is a more efficient discriminator as it classifies patches of the input as real or fake by iterating over the image and taking the average of results.

```
model_input = tf.keras.layers.Input(shape=[256,256,3])
x = layers.Conv2D(filters, kernel_size=(4, 4), strides=(2, 2),padding="same")(
    model_input)
x = layers.LeakyReLU(0.2)(x)

for i in range(3):
    filters = filters * 2
    if i < 2:
        x = layers.Conv2D(filters, (4, 4), strides=(2, 2))(x)
        x = tfa.layers.InstanceNormalization()(x)
        x = layers.LeakyReLU(0.2)(x)
    else:
        x = layers.Conv2D(filters, (4, 4), strides=(1, 1))(x)
        x = tfa.layers.InstanceNormalization()(x)
        x = layers.LeakyReLU(0.2)(x)

x = layers.Conv2D(1, (4, 4), strides=(1, 1), padding="same")(x)
```

This model takes an image input of size (256, 256, 3) which is passed through a set down-sampling blocks to decrease the resolution and extract features. Four of these blocks are applied, the first two using a stride of 2 while the last a stride of 1. These values provided the best scaling and output resolution for a 256x256 input. The network outputs a (32,32,1) tensor, or grid, where each element is a value that corresponds to the networks real or fake prediction for every 32x32 patch. This architecture can be applied to much larger images as it does not need to be fed through as many convolutional layers as a standard image classifier. This discriminator is used on both the real and generated image at the same time and the loss is calculated by comparing the two.

After creating the generator and discriminator networks in a separate python file, the cycle consistent adversarial network is constructed in a python notebook. The data set arrays are imported with numpy's 'load' function and used to create Tensorflow data sets. This was done to utilise operations that are only available on Tensorflow data sets.

```
load_train_pepsi = np.load('datasets/train_pepsi.npy')
load_train_coke = np.load('datasets/train_coke.npy')

trainB = tf.data.Dataset.from_tensor_slices((load_train_pepsi))
trainA = tf.data.Dataset.from_tensor_slices((load_train_coke))
```

After pre-processing the samples, as described in subsection 3.1.2, the residual network is imported and assigned to generator G and F variables for use in the CycleGAN. Generator G transforms Pepsi to Coke while F transforms Coke to Pepsi. The number of residual blocks to be used in the generator is specified as a parameter to the function. The discriminator is also imported and assigned to its respective X and Y variables.

```
generator_g = resnet_generator(
    filters =64,
    num_residual_blocks=9)
generator_f = resnet_generator(
    filters =64,
    num_residual_blocks=9)

coke_discriminator = discriminator(name="coke_discriminator")
pepsi_discriminator = discriminator(name="pepsi_discriminator")
```

Sample images from the data sets are then assigned to corresponding variables to be used throughout the model for visual performance tracking. Pythons itertools library was used to extract an element from each data set for simplicity. To ensure the samples have been pre-processed and extracted correctly, a matplotlib graph is used to display the two images. A loop over the images list applies each to a subplot and assigns the corresponding title. The same samples are also passed through the untrained discriminator to gain an understanding of how the network is visualising and handling its inputs.

```
sampleA = next(itertools.cycle(trainA))
sampleB = next(itertools.cycle(trainB))

plt.figure( figsize =(10, 10))

images = [coke_sample, pepsi_sample]
 title = ['Coke Sample', 'Pepsi Sample']
for i in range(len(images)):
    plt.subplot(1, 2, i+1)
    plt. title ( title [i])
    plt.imshow(images[i][0] * 0.5 + 0.5)
plt.show()
```

plt.imshow(pepsi_discriminator(sampleA)[0, ..., −1], cmap='RdBu_r')

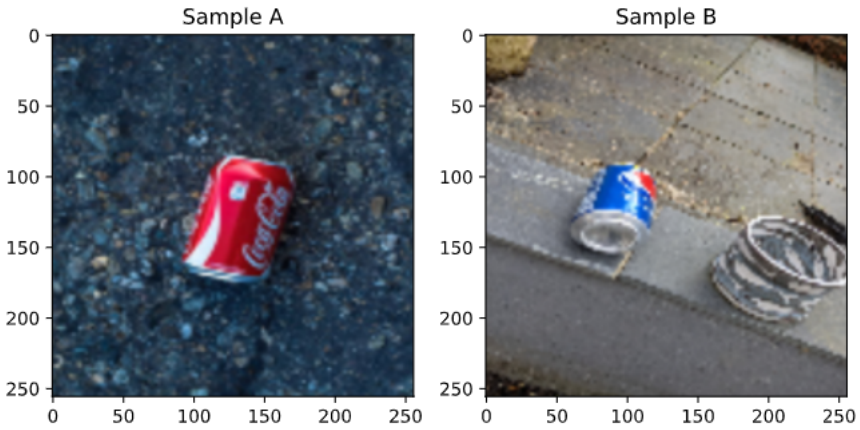


FIGURE 4.1: Pre-processed data samples

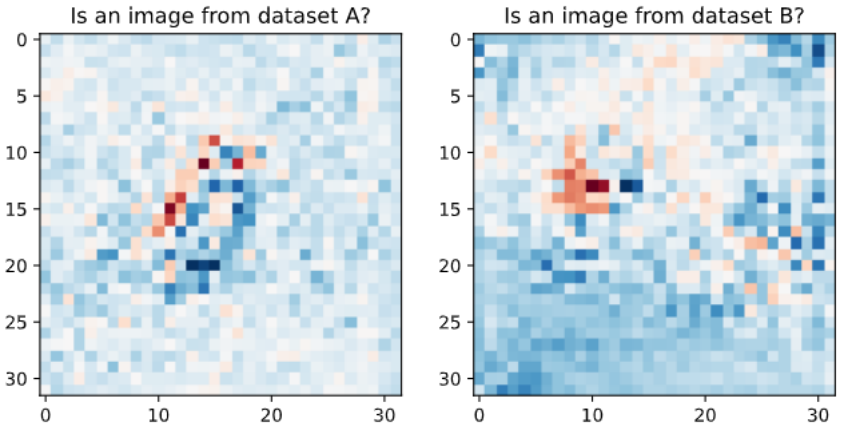


FIGURE 4.2: Output from untrained discriminator

The losses and optimiser are defined in separate functions to be called in the training step. These are all explained in detail in section 3.2.

The training step utilises all the previous code implementations to generate translated images. Within a step the models are trained, losses are calculated, gradient updates are calculated and then applied to update the model for the following iterations. GradientTape is used to calculate and apply the updates to the model as it provides more control over where gradient information is being used, and is a more efficient method of differentiating the large number of layers in the network.

fake_coke = pepsi_to_coke_gen(true_pepsi, training=True)
fake_pepsi = coke_to_pepsi_gen(true_coke, training=True)

```
coke_cycle = pepsi_to_coke_gen(fake_pepsi, training=True)
pepsi_cycle = coke_to_pepsi_gen(fake_coke, training=True)

coke_identity = pepsi_to_coke_gen(true_coke, training=True)
pepsi_identity = coke_to_pepsi_gen(true_pepsi, training=True)

real_coke_disc = coke_discriminator(true_coke, training=True)
real_pepsi_disc = pepsi_discriminator(true_pepsi, training=True)

fake_coke_disc = coke_discriminator(fake_coke, training=True)
fake_pepsi_disc = pepsi_discriminator(fake_pepsi, training=True)
```

The function takes two image inputs, one from each of the Coke and Pepsi data sets, as well as the current epoch the training loop is on for Tensorboard. Due to the size of the model and limited computing power, the model is trained for 40 epochs. Passing the Coke and Pepsi images through their corresponding generator produces the 'fake' outputs which are then used to generate the cycled image by running them through the opposite generator function. The real images are also passed through their matching generator to calculate the identity loss which states that the pixel values of the output should be the same as the input. Finally, the discriminators are trained on both the real input images, and the generated fake images to compare for the discriminator loss.

```
# calculate generator loss with binary crossentropy
coke_to_pepsi_gen_loss = generator_loss(fake_pepsi_disc)
pepsi_to_coke_gen_loss = generator_loss(fake_coke_disc)

# calculating separate mean absolute errors for x and y images
coke_cycle_loss = mae(true_coke, coke_cycle)
pepsi_cycle_loss = mae(true_pepsi, pepsi_cycle)

# calculating the total mean absolute error (both x and y images)
complete_cycle = mae(true_coke, coke_cycle) + mae(true_pepsi, pepsi_cycle)

# total generator loss = adversarial loss + cycle loss
total_coke_to_pepsi_gen_loss = coke_to_pepsi_gen_loss + complete_cycle +
    identity_loss(true_pepsi, pepsi_identity)
total_pepsi_to_coke_gen_loss = pepsi_to_coke_gen_loss + complete_cycle +
    identity_loss(true_coke, coke_identity)

coke_disc_loss = discriminator_loss(real_coke_disc, fake_coke_disc)
pepsi_disc_loss = discriminator_loss(real_pepsi_disc, fake_pepsi_disc)
```

The losses are calculated using the outputs from the training section. The generator loss

functions are called with the generated outputs from the discriminators as a parameter. The generator loss creates a matrix of ones with the same dimensions as the input, and uses binary cross entropy to calculate the loss between true and predicted values from the discriminator.

```
[[[[1 1 1]   [[[[2.4385941]
   [1 1 1]      [2.81485271]
   [1 1 1]      [3.25494194]
   ...          ...
   [1 1 1]      [2.83196425]
   [1 1 1]      [2.29057264]
   [1 1 1]]     [1.86237478]]
```

FIGURE 4.3: Matrix of ones and discriminator output

The individual cycle losses and total cycle loss is calculated by passing the original image input and the previously computed cycled image into the mean absolute error function. The total cycle loss is obtained from the product of the individual Coke and Pepsi cycle losses. Using the generator, cycle, and identity loss, the total generator loss can be calculated by combining these for the corresponding Coke or Pepsi image. Finally, the discriminator losses are calculated by passing the outputs from the discriminator models from training to the discriminator loss function.

```
coke_to_pepsi_gen_gradients = tape.gradient(total_coke_to_pepsi_gen_loss,
    coke_to_pepsi_gen.trainable_variables)
pepsi_to_coke_gen_gradients = tape.gradient(total_pepsi_to_coke_gen_loss,
    pepsi_to_coke_gen.trainable_variables)
coke_discriminator_gradients = tape.gradient(coke_disc_loss, coke_discriminator.
    trainable_variables)
pepsi_discriminator_gradients = tape.gradient(pepsi_disc_loss, pepsi_discriminator.
    trainable_variables)
```

The new parameters for the models are generated using Tensorflow's tape.gradient API. Tape.gradient saves the operations that are carried out during the forward pass using the total loss and trainable parameters from the corresponding generators and discriminators. These operations are traversed in reverse order during the backward pass and the derivatives are calculated and applied to the layers for each epoch.

```
coke_to_pepsi_gen_optimizer.apply_gradients(zip(coke_to_pepsi_gen_gradients,
    coke_to_pepsi_gen.trainable_variables))
```

```
pepsi_to_coke_gen_optimizer.apply_gradients(zip(pepsi_to_coke_gen_gradients,
    pepsi_to_coke_gen.trainable_variables))
coke_discriminator_optimizer.apply_gradients(zip(coke_discriminator_gradients,
    coke_discriminator.trainable_variables))
pepsi_discriminator_optimizer.apply_gradients(zip(pepsi_discriminator_gradients,
    pepsi_discriminator.trainable_variables))
```

Using the ADAM optimiser, these gradient updates are applied to the models using keras optimiser's 'apply_gradients' function. This is passed a zip of the gradient values and trainable variables to ensure they are correctly combined. This concludes the primary operations in a single training step.

The training loop is defined last, taking the specified number of epochs as a parameter. The primary goal of this project is to investigate the performance of these networks so within this loop, the time for each epoch is calculated. The training step is called in an inner loop that iterates over the images in the data sets. Both the generated and cycled images are displayed to track model performance over training, and clear_output is used to update the outputs for each epoch.

```
for i in range(epochs):
  start_time = time.time()
  count = 0
  for coke, pepsi in tf.data.Dataset.zip((trainA, trainB)):
    training_iteration(coke, pepsi, i)
    if count % 10 == 0:
      print('.')
    count += 1

  clear_output(wait=True)
  generate_images(coke_to_pepsi_gen, coke_sample)
  img = img_gen(coke_to_pepsi_gen, coke_sample)
  generate_images(pepsi_to_coke_gen, img)
```

4.1.2 Results

Running the cycle consistent adversarial network with the residual based generator for 40 epochs, a clear mapping between the input and generated images was displayed. The full training procedure was completed in 45 minutes and 4 seconds using an Nvidia GTX 1080ti, and produced clearly transformed images from Coke to Pepsi. The quality of these

outputs increased over the 40 iterations. Table 4.3 shows that the average time to complete an epoch is 67.5 seconds with 2.8 million trainable parameters.

Generator Model	**Average Epoch Time (seconds)**	**Total Training Time (40 epochs)**	**Trainable Parameters**
Residual Net	*67.5*	*45 min 4 sec*	*2,800,000*

TABLE 4.1: ResNet generator efficiency

FIGURE 4.4: Training output after 1 epoch - ResNet Baseline

FIGURE 4.5: Training output after 20 epoch - ResNet Baseline

FIGURE 4.6: Training output after 40 epoch - ResNet Baseline

Saving the training output after every 20 epochs highlights the models development during training. It is not only extracting and mapping a colour transformation from a Coke can to a Pepsi can, but also beginning to learn the more detailed features like the Pepsi logo which can be seen on the top of the can in the output from epoch 40. Table 4.2 exhibits the decrease in loss across the primary tracking metrics. The total cycle loss had highs of 7.3 near the beginning of training, and reduced to lows of 4.0 seen a third of the way in.

Epoch	Total Cycle Loss	Generator G Loss (coke - pepsi)	Generator F Loss (pepsi - coke)
1	*7.320*	*8.493*	*12.48*
10	*5.762*	*8.278*	*8.899*
20	*4.559*	*7.175*	*7.051*
30	*4.003*	*7.190*	*6.287*
40	*4.069*	*6.864*	*6.803*

TABLE 4.2: Losses throughout training from ResNet baseline

Using Tensorboard, the losses are displayed over training. A smoothed line is used for a clearer representation of the updates over the 40 epochs. The total generator loss best illustrates the performance as it is the product of each models generator loss, total cycle loss, and identity loss.

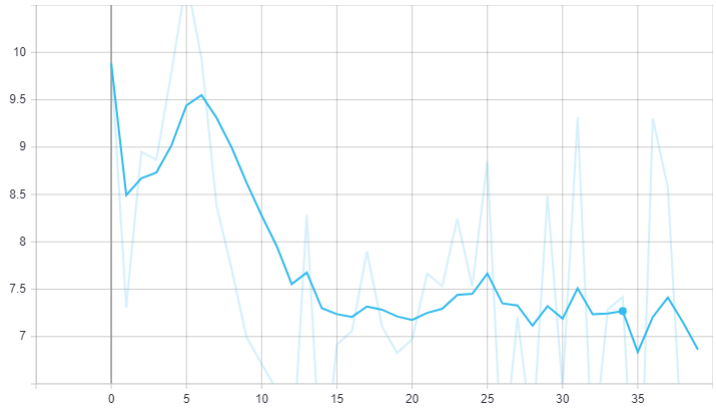


FIGURE 4.7: Total generator G loss over training - ResNet

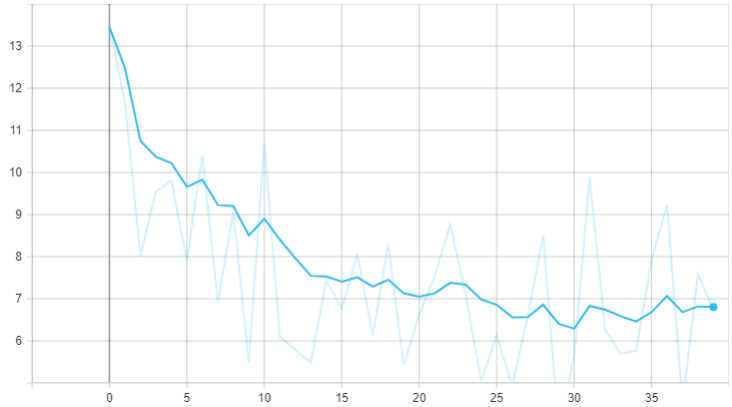


FIGURE 4.8: Total generator F loss over training - ResNet

4.2 Method I: Use of the complete EfficientNet model

4.2.1 Design and Implementation

The entire EfficientNet B3 model was loaded from Tensorflow and used in place of the residual generator to evaluate its performance for feature extraction and image generation. This method did not use a consistent down-sampling and up-sampling process like the ResNet implementation, but instead relied on its own down-sampling layers. This network takes a (256, 256, 3) image input, extracts features, and outputs an (8, 8, 1536) tensor.

```
model = efn.EfficientNetB3(
        include_top=False,
        weights="imagenet",
        input_shape=(256, 256, 3))
```

EfficientNet B3 was chosen for its balance between performance and model size. Network B4 onward produced diminishing returns while the number trainable parameters still increased. The parameters to construct EfficientNet B3 are chosen through a grid search on a small baseline model. Through transfer learning, it utilises the pre-trained weights obtained from the Imagenet data set in an attempt to further reduce the training times. The architecture for this model is comprised of many inverted residual blocks introduced by Mark Sandler in the MobileNetV2 paper [7]. They perform the skipped connections on the narrow layers rather than the wide, as seen in a standard residual network. This process results in a loss in performance, so a linear bottleneck is used before the final convolution in each block.

```
block1a_dwconv (DepthwiseConv2D (None, 128, 128, 40)
_____
block1a_bn (BatchNormalization) (None, 128, 128, 40)
_____
block1a_activation (Activation) (None, 128, 128, 40)
_____
block1a_se_squeeze (GlobalAvera (None, 40)
_____
block1a_se_reshape (Reshape)    (None, 1, 1, 40)
_____
block1a_se_reduce (Conv2D)      (None, 1, 1, 10)
_____
```

block1a_se_expand (Conv2D)	(None, 1, 1, 40)
block1a_se_excite (Multiply)	(None, 128, 128, 40)
block1a_project_conv (Conv2D)	(None, 128, 128, 24)
block1a_project_bn (BatchNormal	(None, 128, 128, 24)

This was then scaled back to original input dimensions using transpose convolutions with instance normalisation and max pooling. The output of each transpose convolution was calculated with the following formula:

$$(n-1)s - 2p + (f-1) + 1$$

and the size after max pooling was calculated with:

$$\frac{n + 2p - f}{s} + 1$$

where n is the input dimension, p is padding, f is kernel size, and s is stride. This gradually reduced the number of features back to 3, while increasing the resolution to 256x256.

```
x = layers.Conv2DTranspose(filters=640, kernel_size=(2, 2), strides=(2, 2),
    use_bias=False)(model.output)
x = layers.MaxPooling2D(pool_size=(2, 2), strides=(1, 1))(x)
x = tfa.layers.InstanceNormalization()(x)

x = layers.Conv2DTranspose(filters=160, kernel_size=(8, 8), strides=(4, 4),
    use_bias=False)(x)
x = layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2))(x)
x = tfa.layers.InstanceNormalization()(x)

x = layers.Conv2DTranspose(filters=40, kernel_size=(4, 4), strides=(4, 4), use_bias
    =False)(x)
x = layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2))(x)
x = tfa.layers.InstanceNormalization()(x)

x = layers.Conv2DTranspose(filters=3, kernel_size=(4, 4), strides=(4, 4), use_bias=
    False)(x)
x = tfa.layers.InstanceNormalization()(x)
```

4.2.2 Results

Using the full EfficientNet B3 model for image generation in the CycleGAN provided poor results. Due to the increase in layers over the residual implementation, the number of trainable parameters is vastly increased. Despite this, the training time per epoch does not scale up at the same rate. The additional 19 million parameters only result in a 45 second increase per epoch, highlighting how the model scaling technique is functioning as expected, however, the generated outputs are unclear and noisy.

Generator Model	Average Epoch Time (seconds)	Total Training Time (40 epochs)	Trainable Parameters
Residual Net	*67.5*	*45 min 4 sec*	*2,800,000*
EfficientNet B3	*130.5*	*1hr 43 min 27 sec*	*21,287,998*

TABLE 4.3: EfficientNet B3 generator efficiency

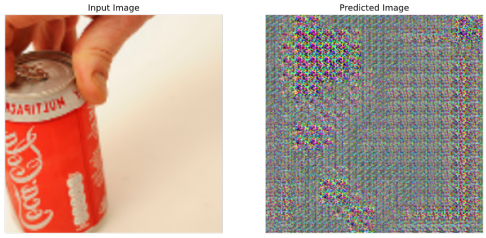


FIGURE 4.9: Training output after 1 epoch - Entire EfficientNet B3

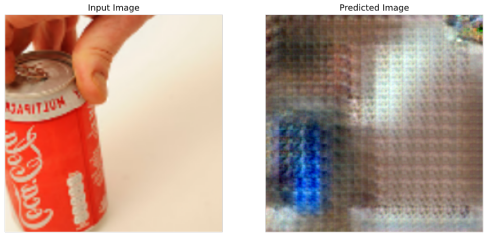


FIGURE 4.10: Training output after 20 epoch - Entire EfficientNet B3

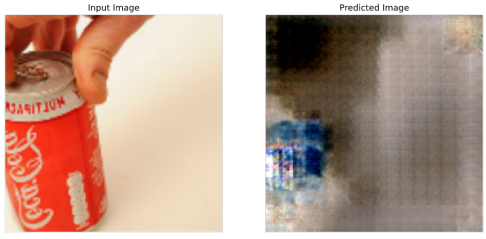


FIGURE 4.11: Training output after 40 epoch - Entire EfficientNet B3

The grid like pattern may be occurring as a result of the amount of down-sampling taking place in the network, in addition to the lesser number of up-sampling layers to bring

the image back to the original dimensions. Over the training time, the outputs become smoother after the addition of max pooling to the up-sampling layers, but fails to effectively transform the Coke can to Pepsi. The model is extracting some low level features, exhibited in figure 4.21 as the blue region in place of the can.

Epoch	Total Cycle Loss	Generator G Loss (coke - pepsi)	Generator F Loss (pepsi - coke)
1	*12.29*	*22.15*	*20.00*
10	*11.89*	*18.51*	*19.71*
20	*9.674*	*15.70*	*14.41*
30	*9.305*	*13.80*	*14.42*
40	*8.169*	*13.57*	*15.96*

TABLE 4.4: Losses throughout training from complete EfficientNet B3

The metrics that were logged during training the EfficientNet B3 generator are far higher than those of the ResNet baseline. The mean absolute error between the input and generated images is much larger, which is visibly apparent in figures 4.9 to 4.11. The losses did decrease over the 40 iterations but are unlikely to have reached a similar point to the baseline if training had been extended. It is expected that the generator and discriminator reach an equilibrium, but the discriminator loss diminished to near 0, suggesting it was dominating the generator resulting in a lack of improvement.

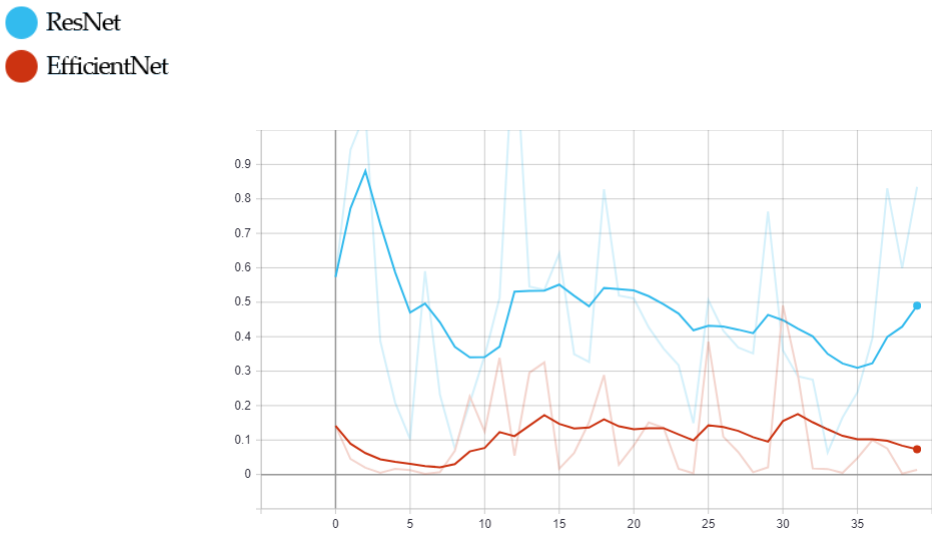


FIGURE 4.12: Discriminator loss over training - Complete EfficientNetB3

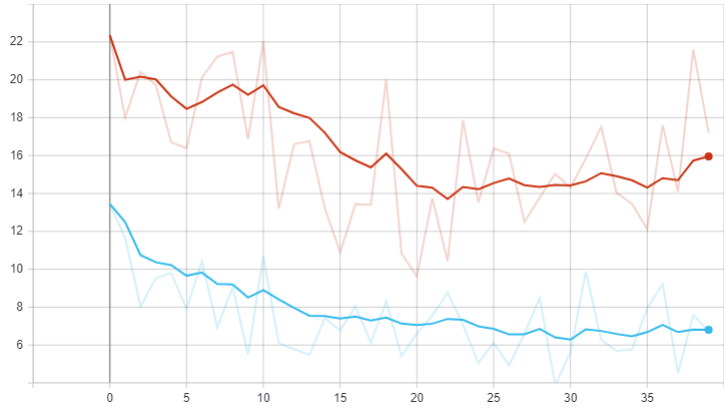


FIGURE 4.13: Total generator G loss over training - Complete EfficientNetB3

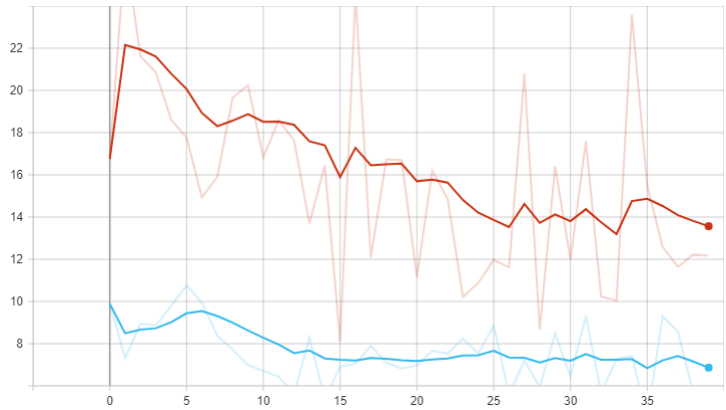


FIGURE 4.14: Total generator F loss over training - Complete EfficientNetB3

4.3 Method II: EfficientNet block - layer by layer

4.3.1 Design and Implementation

The results after testing the entire EfficientNet model for the task of image generation were not ideal. After revisiting the baseline ResNet generator architecture, a deeper understanding of its structure and function was gained. It became apparent that the up-sampling and down-sampling stages play a large part in the clarity of the generated outputs. Following this, a new approach to the integration of the EfficientNet model was explored. Merging aspects of the two models was more likely to produce meaningful results. That is, the up-sampling and down-sampling stages from the baseline, and extracting and stacking blocks of the EfficientNet B3 model for feature extraction. The middle portion of the EfficientNet model was selected in order to utilise a smooth down and up-sampling process. This block consists of a 29 layers, each outputting a 64x64 tensor.

```
for layer in efficientnet_model.layers:
    layer._name = layer.name + str(name)

x = efficientnet_model.layers[30](input)

x1 = efficientnet_model.layers[31](x)
x = efficientnet_model.layers[32](x1)
x = efficientnet_model.layers[33](x)
x = efficientnet_model.layers[34](x)
x2 = efficientnet_model.layers[35](x)
x = tf.keras.layers.Multiply()([x1, x2])
 ...
x = layers.Dropout(0.5)(x) #try  0.5
 ...
x = efficientnet_model.layers[69](x)
x = efficientnet_model.layers[70](x)
x = efficientnet_model.layers[71](x)
x = layers.Conv2DTranspose(144, kernel_size=(1,1), strides=(1,1),  padding="same",
    use_bias=False)(x)
```

The layers were extracted by referencing each one manually and stacking in a function to form a new functional model. Dropout was included in the middle to reduce any over-fitting, although it didn't have any noticeable effect in the outputs. Layers that performed operations between previous outputs could not be referenced in the same way as merge layers could only be called on a list of inputs. The simplest solution was to manually perform these operations with keras' Multiply and Add layers.

An EfficientNet generator function was then created that contains convolution layers for down-sampling the 256,256,3 image input to a 64x64x144 tensor to be fed into the EfficientNet blocks, followed by transpose convolutions for up-sampling to the original dimensions, similar to that of the residual generator.

```
x = layers.Conv2D(filters,  kernel_size=(3,3), strides=(2,2),  padding='same')(x)
x = tfa.layers.InstanceNormalization()(x)
x = layers.Activation("relu")(x)
x = layers.Conv2D(144, kernel_size=(3,3), strides=(2,2),  padding='same')(x)
x = tfa.layers.InstanceNormalization()(x)
x = layers.Activation("relu")(x)

eff_1 = efficient_block_v3(x, 'block1')
eff_2 = efficient_block_v3(eff_1, 'block2')
eff_3 = efficient_block_v3(eff_2, 'block3')
eff_4 = efficient_block_v3(eff_3, 'block4')
eff_5 = efficient_block_v3(eff_4, 'block5')
```

```
x = layers.Conv2DTranspose(64,kernel_size=(3,3),strides=(2,2),padding='same')(
    eff_5)
x = tfa.layers.InstanceNormalization()(x)
x = layers.Activation("relu")(x)
x = layers.Conv2DTranspose(32,kernel_size=(3,3),strides=(2,2),padding='same')(x)
x = tfa.layers.InstanceNormalization()(x)
x = layers.Activation("relu")(x)
```

4.3.2 Results

Using a combination of down and up-sampling and stacked blocks of EfficientNet produced the best results for both the tracked metrics and image outputs. The quality of the generated images was similar to those from the ResNet baseline but was achieved with 2,178,000 less trainable parameters. In addition, training times were decreased from 67.5 to 54 seconds per epoch, which is equivalent to 9 minutes overall.

Generator Model	Average Epoch Time (seconds)	Total Training Time (40 epochs)	Trainable Parameters
Residual Net	*67.5*	*45 min 4 sec*	*2,800,000*
EfficientNet B3	*130.5*	*1hr 43 min 27 sec*	*21,287,998*
EfficientNet Split V1	*54*	*36 min 4 sec*	*622,305*

TABLE 4.5: EfficientNet split model generator efficiency

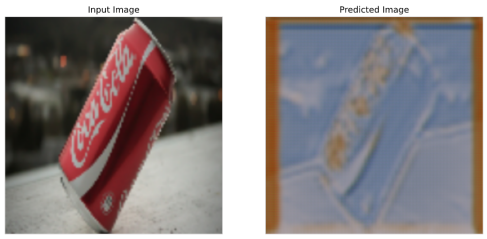


FIGURE 4.15: Training output after 1 epoch - EfficientNet Split V1

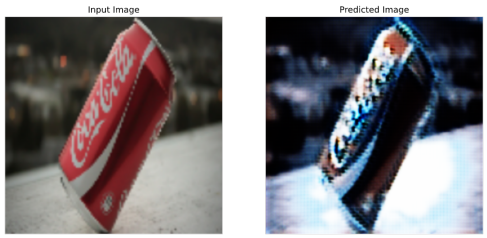


FIGURE 4.16: Training output after 20 epoch - EfficientNet Split V1

The generated images are far clearer than those from the integration of the entire EfficientNet model. They are comparable to the outputs of the ResNet baseline, both displaying

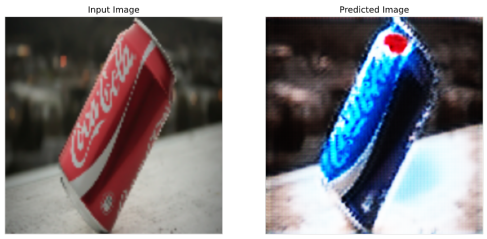


FIGURE 4.17: Training output after 40 epoch - EfficientNet Split V1

a clear mapping from the inputs to generated images. The outputs from the trained EfficientNet split are brighter, contain less noise, and have a more accurate identity mapping, however the baseline did a slightly better job at replacing the 'Coca Cola' text on the can. Table 4.6 illustrates how the initial losses are higher, but all decrease to a similar level as the baseline over the 40 iterations.

Epoch	Total Cycle Loss	Generator G Loss (coke - pepsi)	Generator F Loss (pepsi - coke)
1	*10.14*	*13.08*	*12.77*
10	*5.575*	*8.826*	*9.170*
20	*4.785*	*7.913*	*7.520*
30	*4.030*	*6.724*	*5.915*
40	*3.904*	*6.075*	*6.827*

TABLE 4.6: Losses throughout training from EfficientNet Split V1

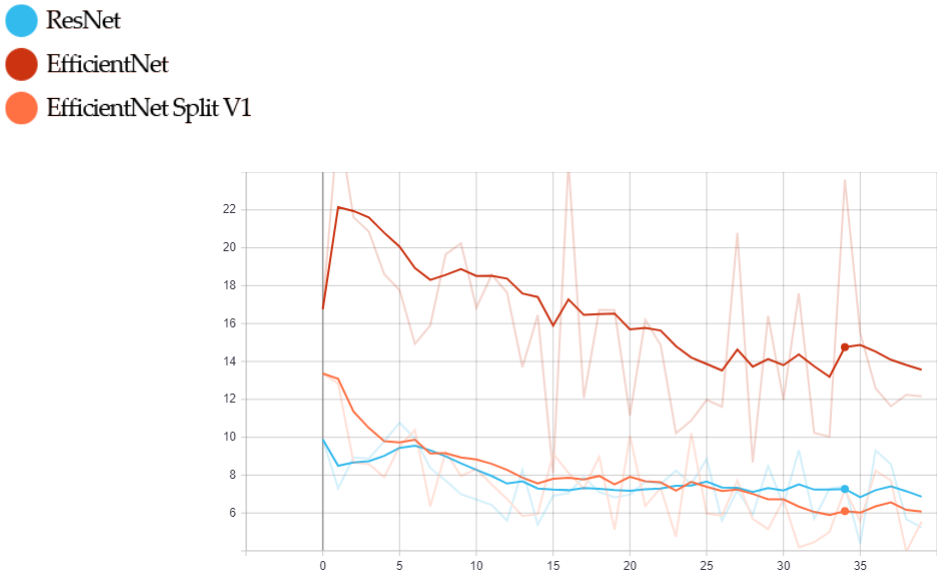


FIGURE 4.18: Total generator G loss over training - EfficientNet Split V1

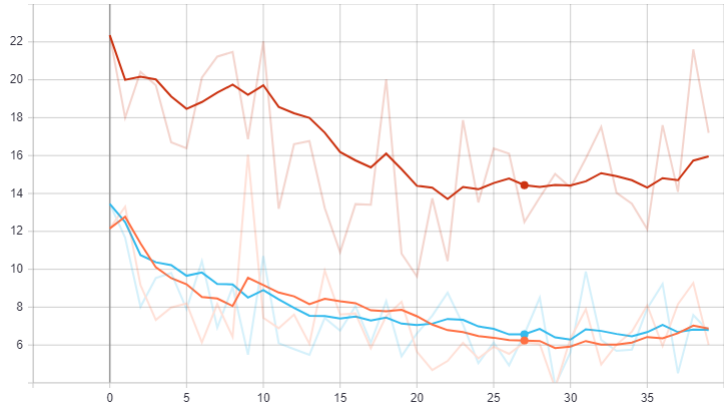


FIGURE 4.19: Total generator F loss over training - EfficientNet Split V1

4.4 Method III: EfficientNet block - model split

4.4.1 Design and Implementation

After the successful implementation of an EfficientNet generator, a final alteration was made to the method in which layer extraction was performed. It was possible that the method used for the EfficientNet model could have had a negative affect on the generators performance. It was unknown if performing the multiplication and addition operations between layers manually would interfere with the connections in the pre-trained network, so a function to split the model while retaining these layers was created to ensure performance was not being lost.

```
l_used = set()
structure = net_in.get_config()
for i, layer in enumerate(structure['layers']):
    if i == 0:
        structure['layers'][0]['config']['batch_input_shape'] = net_in.layers[
            first_layer].input_shape
        if i != first_layer:
            structure['layers'][0]['name']
            structure['layers'][0]['config']['name'] = structure['layers'][0]['
                name']
    elif i < first_layer:
        continue
    elif i > last_layer:
        continue
    l_used.add(layer['name'])
```

A python dictionary containing the configuration of the complete EfficientNet B3 model is assigned to a variable using the get_config function. An empty set is then defined which

is used to store the layer names that are to be extracted from the model. A loop is then performed over the layers stored in the configuration dictionary where the names are extracted and added to the set.

```
l_extracted=[]
for l in structure['layers']:
    if l['name'] in l_used:
        l_extracted.append(l)
l_extracted[1]['inbound_nodes'][0][0][0] = l_extracted[0]['name']

structure['layers'] = l_extracted
structure['input_layers'][0][0]  = l_extracted[0]['name']
structure['output_layers'][0][0]  = l_extracted[-1]['name']
```

The information in the dictionary related to the layer names is assigned to a new list which is used to recompile the start, middle and end layers.

```
modelsplit = Model.from_config(structure)

for layer in modelsplit.layers:
    layer._name = layer.name + str(name)

modelsplit._name = 'EfficientNet_' + str(name)
return modelsplit
```

The new model is then reconstructed by assigning the layers that have been extracted from the input model and assigned to the structure dictionary using model.from_config.

The generator function remained the same as described in section 4.3.1, with the down and up-sampling layers surrounding the stacked blocks from the above split function.

4.4.2 Results

Extracting the EfficientNet blocks through this method provided no additional performance. Table 4.7 highlights how the total training time is increased an amount within a margin of error that suggest it performs near identical to the layer by layer extraction method. Both have 622,305 trainable parameters, which indicates that the implementation from section 4.3.1 functioned as initially planned. Although this implementation was not required, it is a far more versatile method of layer extraction that can serve many other projects.

Generator Model	Average Epoch Time (seconds)	Total Training Time (40 epochs)	Trainable Parameters
Residual Net	*67.5*	*45 min 4 sec*	*2,800,000*
EfficientNet B3	*130.5*	*1hr 43 min 27 sec*	*21,287,998*
EfficientNet Split V1	*54*	*36 min 4 sec*	*622,305*
EfficientNet Split V2	*55.5*	*37 min 8 sec*	*622,305*

TABLE 4.7: Losses throughout training from EfficientNet Split V2

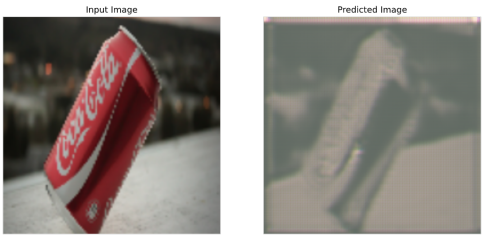


FIGURE 4.20: Training output after 1 epoch - EfficientNet Split V2

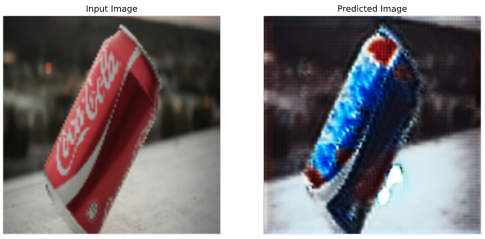


FIGURE 4.21: Training output after 20 epoch - EfficientNet Split V2

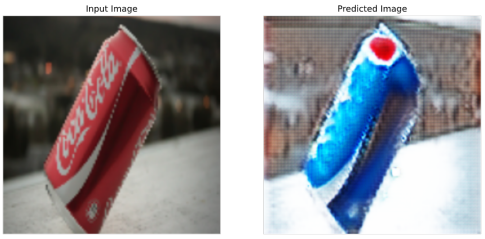
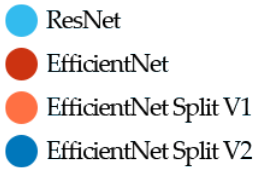


FIGURE 4.22: Training output after 40 epoch - EfficientNet Split V2

The outputs at the end of training show a clear translation from Coke to Pepsi, similar to those from the baseline and first EfficientNet split. The generated images are very similar in quality to those from the layer by layer extraction method, but appear brighter in the later epochs. The generator G (Coke - Pepsi) loss ended 0.767 higher in this implementation compared to Efficient split V1, and only 0.307 for Generator F (Pepsi - Coke). It is safe to assume that the two implementations perform the same for image generation as the figures in tables 4.6 and 4.8 change slightly from run to run and are very similar.

Epoch	Total Cycle Loss	Generator G Loss (coke - pepsi)	Generator F Loss (pepsi - coke)
1	*11.81*	*12.09*	*13.93*
10	*6.279*	*8.824*	*8.041*
20	*4.501*	*7.337*	*7.295*
30	*3.832*	*6.577*	*6.207*
40	*3.583*	*6.842*	*6.520*

TABLE 4.8: Losses throughout training from EfficientNet Split V2



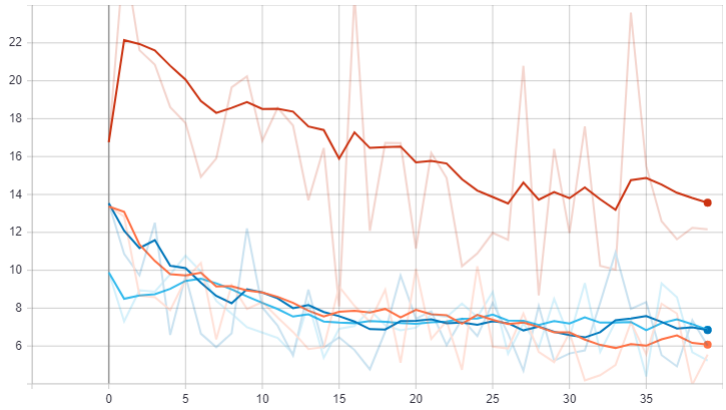


FIGURE 4.23: Total generator G loss over training - EfficientNet Split V2

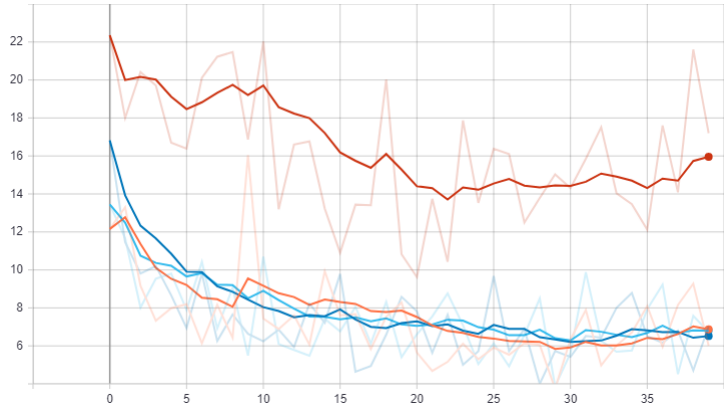


FIGURE 4.24: Total generator F loss over training - EfficientNet Split V2

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The aim of this project was to improve the efficiency of image generation with a focus on the EfficientNet model. The initial plan was to use the entire pre-trained EfficientNet B3 model in place of the residual generator to utilise its uniform model scaling technology. After running and testing the model for 40 epochs, it was clear that this was not as effective as hoped and there was minimal mapping from the image inputs to the generated outputs.

After this initial failure, I was still motivated to discover a functional integration of this technology. Following a review of prior solutions, it became apparent that the encoding and decoding portions are an important element of the generative process. If these are not performed correctly, the network is unable to effectively reconstruct an accurate image output, which can be seen in the results from section 4.2.2. Knowing this, a successful application was developed by separating a portion of the network for efficient feature extraction, coupled with consistent down and up-sampling layers. This reduced the number of trainable parameters by 77.8%, bringing the training time for 40 epochs down by 9 minutes using an Nvidia GTX 1080ti.

From the results obtained after testing each method, it can be concluded that EfficientNet provides improved results over the prior ResNet architecture when generating synthetic images with generative adversarial networks, but the extent to which it can be applied is limited. When using the entire model, the inputs are processed too much and it becomes difficult to recreate a high quality output. In order to utilise this model for image generation, the best performance was observed after separating and stacking small blocks to

ensure a constant image resolution before up-scaling.

5.2 Future Work

Although the final implementation surpassed the baseline performance, and efficiency was improved, the generated images are still clearly fake. Generative adversarial networks aim to create images that are indistinguishable from a real input. Therefore, the primary future direction for this project is to improve the quality of its outputs while utilising the uniform model scaling technology explored in this project. I plan to develop my understanding of how model scaling in EfficientNet works in order to apply it to custom networks. In doing so, I will be able to explore other models that may be better suited for generative tasks, while retaining the efficiency and performance of EfficientNet.

The Pepsi and Coke data sets used for this project consisted of 300 and 200 images respectively. This is considered very small, but due to time limitations could not be avoided. Increasing both of these to a minimum of 1000 each is likely to improve the quality of the outputs so is definitely worth the time investment.

It was discovered after the testing stage that the use of transpose convolutions can lead to checkerboard artifacts when used for up-scaling images. There are other techniques that can be applied in place of transpose convolutions, such as Tensorflow's upscaling layers, which may produce higher quality outputs.

Appendix A

Code Repository and User Guide

The code and documentation for this project can be found at the following GitHub repository:

`https://github.com/Chris-Mayes/Dissertation/tree/devChris`

All the code required to run the model can be cloned in order to replicate the results, however a dedicated GPU is advised to reduce training times. The custom data set used in this project is not included due to file size, but can be replaced with one from `https://www.tensorflow.org/datasets/catalog/cycle_gan`. The documentation and code is split into separate folders - 'documentation' and 'model'. The generator version to be used can be changed in the EfficientNet.py file by modifying the function call within the efficientnet_generator.

Bibliography

[1] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML].

[2] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[3] Jie Hu, Li Shen, and Gang Sun. "Squeeze-and-Excitation Networks". In: *CoRR* abs/1709.01507 (2017). arXiv: 1709.01507. URL: http://arxiv.org/abs/1709.01507.

[4] Phillip Isola et al. *Image-to-Image Translation with Conditional Adversarial Networks*. 2018. arXiv: 1611.07004 [cs.CV].

[5] Tero Karras, Samuli Laine, and Timo Aila. "A Style-Based Generator Architecture for Generative Adversarial Networks". In: *CoRR* abs/1812.04948 (2018). arXiv: 1812.04948. URL: http://arxiv.org/abs/1812.04948.

[6] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[7] Mark Sandler et al. *MobileNetV2: Inverted Residuals and Linear Bottlenecks*. 2019. arXiv: 1801.04381 [cs.CV].

[8] Mingxing Tan and Quoc V. Le. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*. 2020. arXiv: 1905.11946 [cs.LG].

[9] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. *Instance Normalization: The Missing Ingredient for Fast Stylization*. 2017. arXiv: 1607.08022 [cs.CV].

[10] Jun-Yan Zhu et al. "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks". In: *CoRR* abs/1703.10593 (2017). arXiv: 1703.10593. URL: http://arxiv.org/abs/1703.10593.