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

April 14, 2021

Declaration of Authorship

I, Christopher ANTHONY MAYES, declare that this thesis titled, "Evaluating the Performance of EfficientNet for Synthetic Image Generation" and the work presented in it are my own. I confirm that:

- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Signed:

_____

Date:

_____

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 Despite their potential, there are still many concepts that are unexplored or outdated.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 [8] 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

List of Abbreviations

GAN	**G**enerative **A**dversarial **N**etwork
MAE	**M**ean **A**verage **E**rror
CNN	**C**onvolutional **N**eural **N**etwork

For/Dedicated to/To my...

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. The downside to these models is the computational costs associated with training to produce the desired 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 within a Generative Adversarial Network are made up of many convolutional layers for feature extraction, transpose convolutions for up-scaling the output back to the original input dimensions, as well as many others depending on the given task. This results in deep neural networks often having millions of parameters, producing a hard to navigate error landscape due to the high 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 image generation tasks. Up to now, this method of model scaling has only been documented on classification tasks, 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 based off the CycleGAN [8] papers Residual Network [2] implementation. A selection of generator models will be presented that each take 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 Pepsi can should become the same image with a Coke can in its place. The performance of this translation will be evaluated using the Mean Average Error loss metric, comparing both the translation from Coke to Pepsi, as well as the translated Coke back to the original Pepsi which fulfills the cyclic aspect of the CycleGAN. In addition, the performance of the networks 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

Fill this out at the end

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.

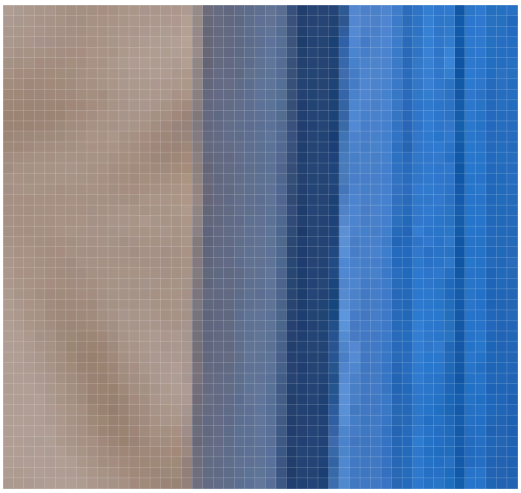

FIGURE 2.1: Pixel Region of Pepsi Can

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 [4] 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.

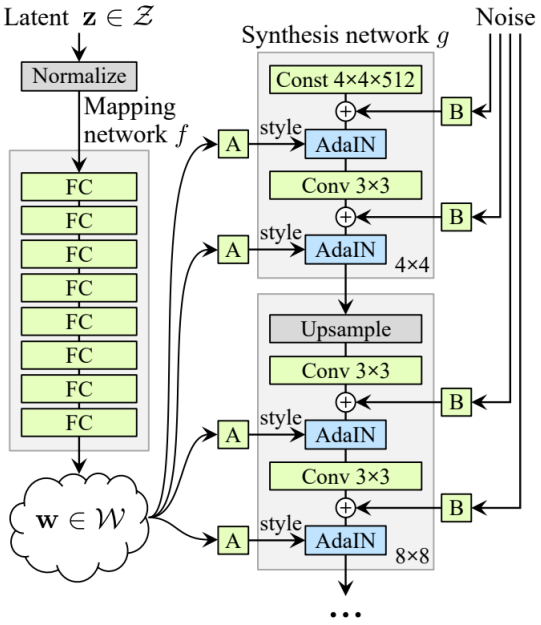

FIGURE 2.2: Architecture of StyleGAN network

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 [8], 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

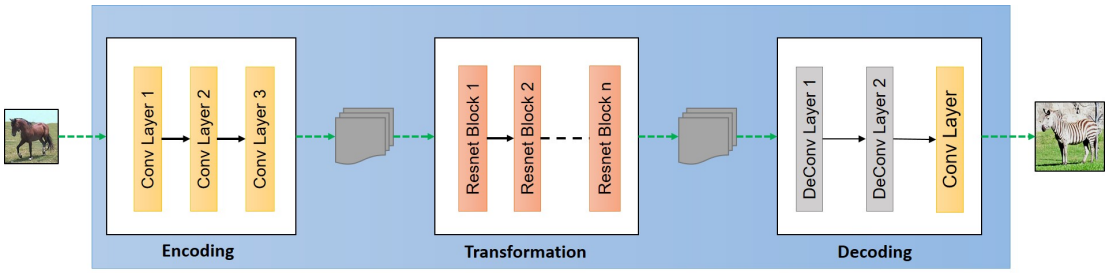

FIGURE 2.3: Architecture of CycleGAN generator

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 [6] 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. All 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

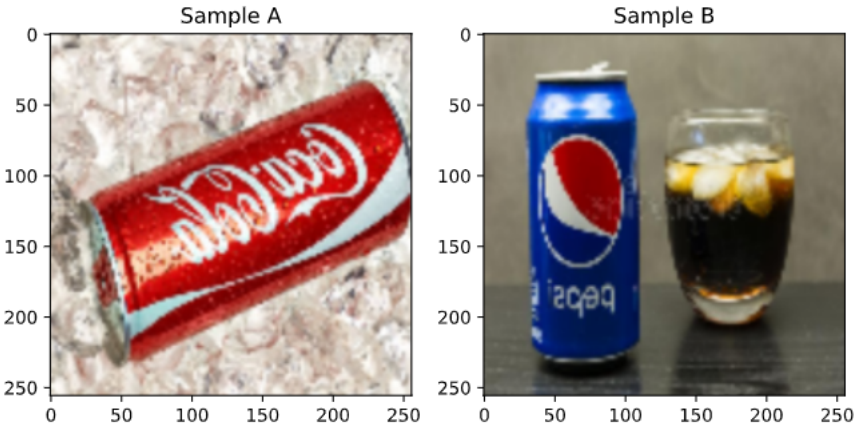

FIGURE 3.1: Data set sample images

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 results 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 data 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 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 both 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 gets the pixel distance between the input and the generated output and is added to the overall cycle 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

The 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 [5] specifically for the 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 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

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 takes 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 [7] and a ReLU activation. The input image is stored in a variable called *input_tensor* to be used for the skipped connection.

```
def residual_block(x, activation, kernel_size=(3, 3), strides=(1, 1), padding="valid",
    use_bias=False):
    dim = x.shape[-1]
    input_tensor = x

    x = ReflectionPadding2D()(input_tensor)
    x = layers.Conv2D(dim,kernel_size,strides=strides,padding=padding,use_bias=
        use_bias,)(x)
    x = tfa.layers.InstanceNormalization()(x)
    x = activation(x)

    x = ReflectionPadding2D()(x)
    x = layers.Conv2D(dim,kernel_size,strides=strides,padding=padding,use_bias=
        use_bias,)(x)
    x = tfa.layers.InstanceNormalization()(x)
    x = layers.add([input_tensor, x])
    return x
```

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 returned in the tf.pad function to be applied to the input tensor. This class was inspired by Tensorflows implementation.

```
class ReflectionPadding2D(layers.Layer):

def __init__(self, padding=(1, 1), **kwargs):
    self.padding = tuple(padding)
    super(ReflectionPadding2D, self).__init__(**kwargs)

def call(self, input_tensor, mask=None):
    width, height = self.padding
    pad_output = [[0, 0], [height, height], [width, width], [0, 0]]
    return tf.pad(input_tensor, pad_output, mode="REFLECT")
```

Reflection padding uses the contents of the image matrices for the padding values and reflects the row into the padding ensuring 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 and become more detailed and specific the more blocks are stacked at the cost of increased parameters. Instance normalisation [7] 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 specified 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 a 2D convolution, Instance Normalisation and ReLU activation. The 2D convolution extracts progressively higher levels

of features, compressing features that might be distant spatially, but are related in the image, and transforming 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 blocks.

```
for _ in range(num_residual_blocks):
    x = residual_block(x, activation=layers.Activation("relu"))
```

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 image of matching style to the input.

The image must be scaled back to the original input dimensions in the encoding 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 and will be used in the training loop to predict whether the image inputs are real or generated.

4.1.1 Design and Implementation

4.1.2 Tests and Results

4.2 Method I: Use of the complete EfficientNet model

4.2.1 Design and Implementation

4.2.2 Tests and Results

4.3 Method II

4.3.1 Design and Implementation

4.3.2 Tests and Results

4.4 Method III

4.4.1 Design and Implementation

4.4.2 Tests and Results

4.5 Method IV

4.5.1 Design and Implementation

4.5.2 Tests and Results

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] 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.

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

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

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

[8] 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.