University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Deep Learning Methods to Solve the Inverse Problem of Artificial Image Colourisation

James Conway

April 6, 2020

Abstract

Colourisation is a difficult and ambiguous problem, and a reliable solution for it needs to account for a significant amount of information. This project investigated the use of Deep Learning to solve the problem, which was chosen for its ability to generate solutions based on the analysis of large quantities of relevant data. In order to understand which Deep Learning methods were most capable of colourisation, we implemented three systems that seemed like the best candidates: A Convolutional Neural Network, a Generative Adversarial Network, and a Variational Autoencoder. We evaluated each one with the same colourisation tasks, and used the information we gathered to determine which was the most viable solution.

Acknowledgements

I would like to thank my supervisor Dr. Roderick Murray-Smith and my co-supervisor Francesco Tonolini for their invaluable guidance, as well as for introducing me to the fascinating field of Deep Learning.

I would also like to thank my friend Amelie Voges for helping me write this dissertation, and my Mum and Dad for supporting me throughout my academic career.

Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: James Conway Date: 6 April 2020

Contents

1 Introduction

1.1 Motivation

Image colourisation is any process where colour is added to a monochrome image, usually performed to make the image seem more realistic or visually appealing to a human viewer. The most reliable current methods for colourising black and white images involve a mainly manual process, whereby an artist analyses the semantic details of a greyscale image, and from them infers the corresponding colours. This process is often assisted by semi-automatic colourisation tools that are still heavily reliant on user input. Even with such tools, colourisation is currently a time-consuming and labour-intensive process, the outcome of which is heavily dependent on the skill of the artist.

The existence of a fully automated tool for colourisation, would allow it to be reliably performed without requiring skilled artists or extensive periods of time. This would mean that projects requiring the colourisation of many numerous images, such as video colourisation, can be carried out far more efficiently than before, and in a fraction of the time. Automatic colourisation could also have other technical uses that are not possible with current manual methods. For example, some monochromatic imaging technologies such as night vision or sonar, could use automatic colourisation to enhance the details of their images, helping the user to distinguish objects.

1.2 Problem and Solution

Colourisation is part of a particular category of problems known as "ill-posed inverse problems". This means that it does not have a single direct solution which depends continuously on the provided data, and that it is impossible to formulate a single unique model which is capable of correctly colouring any image. The best way to solve it is by exploiting as much additional information as possible, in order to produce a solution that best approximates the correct result.

Deep Learning has recently been employed as the solution of choice to this problem, as it provides tools that are capable of accounting for much of this additional information, including many unknown variables which a human may not account for. There are a variety of different Deep Learning methods that are well adapted to tasks involving image analysis, and can also be used for colourisation, however it is still uncertain which method can provide the best solution. In this project we will implement and test a number of these different models, and will compare their performance in order to find the best one.

1.3 Aim

The specific aims of this project are to:

- Analyse the nature of the problem, in order to understand how Deep Learning in particular can best solve it.
- Investigate how to best prepare the training data to be analysed by the Deep Learning models, and see which kinds of data work best.

- Implement and test a variety of Deep Learning solutions that could be viable, including a:
  - Convolutional Neural Network
  - Generative Adversarial Network
  - Variational Autoencoder
- Evaluate the performance of each model, and compare the strengths and shortcomings of each to determine which is the best solution.

2 Background

In this chapter we will analyse the nature of the problem of colourisation and explain the reasoning behind a Machine Learning approach to solving it. We will also highlight seminal projects that have attempted to solve the problem and what they achieved.

2.1 Colourisation as an Inverse Problem

We call two problems inverses of one another if the formulation of each involves all or part of the solution of the other. To solve an inverse problem, we need to look at the direct problem from which it derives which will have a cause-effect sequence to compute the unknown effects of known causes. The inverse problem is then solved by reversing this cause-effect sequence to compute the unknown causes of known effects[17].

In the case of colourisation we can say that greyscale to colour image conversion is the inverse of colour to greyscale conversion, the solution of which can be found by reversing the colour to greyscale transformation. We'll demonstrate this by first solving the problem of colour to greyscale image conversion. Typically, colour images are represented in data using the Red-Green-Blue (RGB) colour model (see figure 2.1).

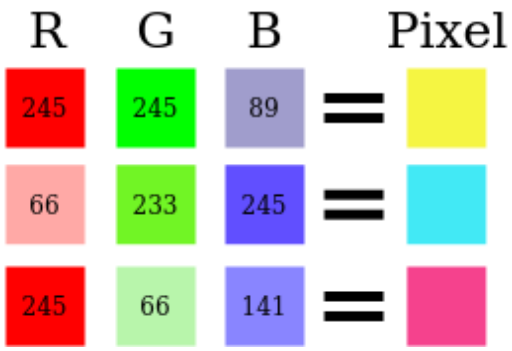


Figure 2.1: *In the RGB encoding, each pixel's colour value is produced by superimposing Red, Green, and Blue colours of varying intensities.*

There is no unique method to perform a RGB to greyscale conversion, however, a common solution is the "Weighted" or "Luminosity" method. This involves multiplying the Red, Green, and Blue values of each pixel by coefficients which represent how much each colour contributes to the overall luminosity of the image, and then summing them together to produce corresponding greyscale pixel values. For instance, if we decided that Red contributes 30% to the luminosity of an image, Green contributes 59%, and Blue contributes 11%, the per-pixel conversion would be as follows:

$$greyscale = ((0.3 * R) + (0.59 * G) + (0.11 * B)) \tag{2.1}$$

However, these channel weightings are not constant, meaning a loss of information is intrinsic to the conversion from RGB to greyscale. With this loss of information, the inverse problem of then converting from greyscale back to RGB has no unique solution which it can follow to recover the original RGB values. This lack of a unique solution is where the main difficulty of the problem lies, and it puts image colourisation into a particular category known as "ill-posed" inverse problems.

2.2 Ill-posedness

An ill-posed problem is a problem which violates at least one of Hadamard's three properties of well-posedness[15]:

1. The problem has a solution.
2. The solution is unique.
3. The solution depends continuously on the parameters or input data.

Colourisation is ill-posed as it lacks a unique solution and does not depend continuously on the input data. A high intensity greyscale pixel can correspond to a high intensity red, green, or blue value, with no discrete information to determine which. No exact algorithm can be developed which is capable of producing correct solutions for any arbitrary input, the only option is to develop an approximate solution.

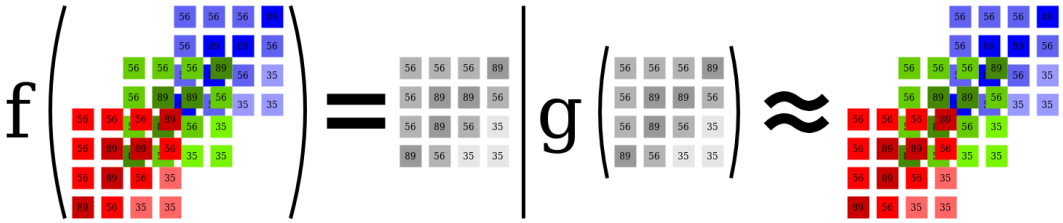


Figure 2.2: *With the information lost in the greyscale to colour computation of f, the only solution is to develop a separate function g, capable of producing an approximate computation of f's input.*

The general rule for finding the best approximate solution is to construct it around additional constraints derived from the nature of the problem, also known as "priors" or "prior information"[17]. There are a multitude of potential priors which can be considered when dealing with colourisation.

In the context of colourisation, the priors can come in a variety of forms. Low level priors can be very specific details about the image, such as edges and contours, which can indicate where an object and its colours end. High level priors are more general details, like object categories, which can inform us of what colour an object is through recognition. For example, a stone is likely to be grey.

The wide choice of priors presents another level of difficulty inherent to the problem. While some of these priors can provide vital information to assist in developing a solution, others can guide the colourisation process towards incorrect results. Take the previous example of categorising an object in the image as a tree, which could inform the system that the object will have a green colour bias. If the picture of the tree was taken in Autumn however, its colours would have a red/yellow bias which might not be evident from the input (see figure 2.3). An optimal approximate solution should consider as many priors as possible, but must take into account that some priors can have a misleading impact on the outcome of the colourisation.

Figure 2.3: *If the system categorises all trees as having a green colour bias, trees in autumn with red/yellow biases will be coloured green.*

2.3 Machine Learning Solutions

If we were to write a traditional algorithm that could colour images, it would ideally work by detecting and segmenting each object in the image based on its curvature and texture, and then would apply a corresponding colourisation to individual objects. Trees, for example, would be coloured brown around the trunk and green around the leaves. This could work fine until we encountered an image like the autumn tree (see figure 2.3), in which case we would require another sub-procedure to collect more information in order to determine the season. With this we would already have a fairly complex algorithm that is only capable of colouring trees. In order to be considered a viable solution for colourisation, it would require even more specific code for the myriad of other potential objects it could encounter. Since the problem is so unconstrained, an algorithm to solve it would have to be very complex, meaning it would scale poorly and likely be very inefficient. We also need to consider that this is only considering priors that we know about particular objects, there may also be information needed to assist the colourisation process that a programmer might never notice.

Machine Learning (ML) provides a range of techniques that can solve problems like colourisation, which are too complex for traditional algorithms. ML models can analyse millions of reference images and discover patterns vital to the solution that might not be apparent. In doing so, these ML models can colourise images while accounting for many of the known and unknown priors. ML thus provides the most feasible means with which to find an approximate solution to the problem, and are the main focus of this project.

2.4 Related work

Many of the first solutions to colourisation gathered their main priors from user input, and these inputs were typically either reference images or drawn labels. [20] was the first reference image reliant approach, wherein the user would provide the system with a colour image which was similar to the greyscale image to be colourised. This relieved the user of having to specify any of the semantic details of the image, but it was heavily reliant on the user's ability to find a colour image which closely approximated the greyscale one. [8] was a later scribble based labelling method where the user could scribble colours onto parts of the greyscale image where they expected to see them. However, this required a greater effort from the user in having to specify each of the colour's locations and in certain cases could be difficult to determine. Later projects often combined input methods such as [7], which took a greyscale image containing a labelled and segmented foreground object determined by the user. The system would then generate

a set of colourisation results using reference images automatically searched and retrieved from the internet. The provided image however, could often contain objects which were hard to segment without having a rigid shape, and the image may contain multiple complex objects which the system couldn't account for. The performance of these user input methods were all heavily dependent on the user's ability to provide a clear and useful input to the system, which could often be time consuming and difficult when some parts of the image have no clear colour correspondence.

Fully automatic colourisation methods were later developed to solve these limitations. In order to achieve this, a system is needed that is capable of accounting for the multitude of potential priors and reference images needed to produce satisfactory results. Deep learning techniques proved successful at modelling large-scale data using millions of reference images and accounting for many of the image's semantic details. One of the first state of the art examples being [21] which proposes a fully automatic approach using Convolutional Neural Networks, a technology which we also used in this project (see 4.1). The CNN produced promising results with the help of a very large reference database, as well as low and mid level feature extraction and joint bilateral filtering to smooth the result. This CNN was still limited however, in that it relied on image segmentation provided by the feature extractors, meaning that its results were only satisfactory for images that conform to the pretrained segmentation classes it was provided.

The later development of Generative Adversarial Networks provided another method to solve the problem of image colourisation which was first explored by [18]. GANs were used to solve a number of image-to-image translation problems, including colourisation with promising results. GANs were capable of generalising well to a wide range of images and were employed in a number of other projects for colourisation, such as [19], [6], and [16]. The widespread use of GANs to perform image colourisation meant that they were a tried and tested solution with a good deal of example implementations that could be consulted in order to create our own implementation (see 4.2). Despite the GAN's ability to generalise well, they could still fail to produce plausible colouriations for many of the same images that CNN's had difficulties with.

GANs were only capable of generating single outputs for each greyscale input, and to solve this one project [5] proposed Variational AutoEncoders as another colourisation method which had the interesting ability to provide a diverse range of possible colourisation for each greyscale input. This presented a promising way to circumvent the difficulties related to certain images which neither GANs or CNNs could colourise well enough, as VAEs could present a variety of possible colourisations which were more likely to have a plausible output. We based our own implementation of a colourisation VAE (see 4.3) mainly on this project, although we were not able to implement some of the more complex features of their implementation such as the Mixture Density Network, or the loss function that accounts for uneven colour distributions.

3 Analysis

In this chapter we will give a high level explanation of the technology and data used to develop our colourisation models and why they were chosen.

3.1 Problem Formalisation

To illustrate the problem more clearly, and to succinctly explain our approach to the solution, we will express the inverse problem of colourisation and its general solution using mathematical notation.

3.1.1 General RGB Problem

Consider the direct problem of RGB to greyscale conversion (as explained in 2.1). We can express the conversion in matrix notation, where A is the coefficient matrix which, when multiplied by a 3D matrix X_{RGB} containing the RGB values, produces a 2D matrix Y containing the resulting greyscale values:

$$AX_{RGB} = Y. \tag{3.1}$$

The approximate solution to the inverse problem (see figure 2.2) can then be formalised as the process of finding a colourisation function $f(Y)$ with a set of parameters θ that minimises the difference between the colourisation's output $f_\theta(Y)$, and input's ground truth X_{RGB}:

$$\underset{\theta}{\operatorname{argmin}} |X_{RGB} - f_\theta(Y)|^2 \qquad s.t.\ AX_{RGB} = Y \quad \forall\, X_{RGB}, Y. \tag{3.2}$$

3.1.2 YUV Encoding Constraint

We can simplify and further constrain the problem significantly by converting X_{RGB} to a YUV encoding, X_{YUV}. Like RGB, the YUV model represents the image through 3 channels, the first channel Y encodes the brightness of the image while the other two channels U and V encode the image's green-blue and red-yellow projections (see figure 3.1). The benefit of this encoding is that the Y channel on its own is equivalent to a greyscale version of the image, this creates a dependency between the input X_{YUV} and output Y values, as the first channel of a YUV image will correspond to the greyscale conversion Y. The colour to greyscale conversion is then much simpler, as instead of multiplying X_{RGB} by a coefficient matrix A to acquire the greyscale output, we can simply multiply X_{YUV} by a part diagonal matrix that isolates the Y values.

We'll demonstrate this algebraically. The RGB to YUV conversion can be done by multiplying the X_{RGB} by C, a matrix containing the coefficients to convert RGB pixels to YUV:

$$CX_{RGB} = X_{YUV}. \tag{3.3}$$

Substituting this into left side of the conversion we get:

$$AC^{-1}X_{YUV} = Y. \tag{3.4}$$

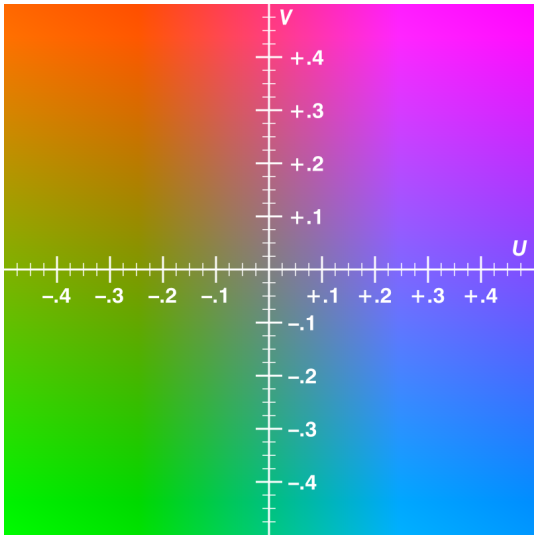


Figure 3.1: *UV plane for the YUV encoding, notice how U is the green-blue axis and V is the red-yellow axis. Every RGB colour can be acquired by sampling from points on this plane.*

A is kept on the left side and multiplied by C^{-1}, this results in the desired part diagonal matrix $\hat{I}$, that can isolate the Y values in X_{YUV} to obtain only the greyscale value matrix Y:

$$AC^{-1} = \hat{I}. \tag{3.5}$$

Substituting this diagonal matrix into the left side of the equation we get the direct problem in YUV form:

$$\hat{I}X_{YUV} = Y. \tag{3.6}$$

With this more constrained reformulation of the problem, the approximate solution which will be solved in this project entails finding a colourisation function $f(Y)$ and a set of parameters θ that minimises the difference between the colourisation's YUV output $f_\theta(Y)$, and the UV ground truth X_{YUV}:

$$\underset{\theta}{\operatorname{argmin}} |X_{YUV} - f_\theta(Y)|^2 \qquad s.t.\ \hat{I}X_{YUV} = Y \quad \forall\, X_{YUV}, Y. \tag{3.7}$$

We can further simplify the solution, due to the Y channel dependency between the input X_{YUV} and output Y values. The problem can now be solved more simply by finding a colourisation function $g(Y)$ that produces a UV output, and a set of parameters θ that minimises the difference between the function's output $g_\theta(Y)$, and the UV ground truth X_{UV}:

$$\underset{\theta}{\operatorname{argmin}} |X_{UV} - g_\theta(Y)|^2. \tag{3.8}$$

In this case the $\hat{I}X_{YUV} = Y$ constraint is satisfied by definition for any U and V.

3.2 Deep Learning Methods

Deep Learning (DL) is a subfield of ML which is also used to solve problems with difficult to define solutions. DL models allow us to tackle even more complex problems thanks to their greater ability to detect useful features in the data. As defined by Goodfellow, "DL models can learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined through its relation to simpler concepts"[12] (see figure 3.2).

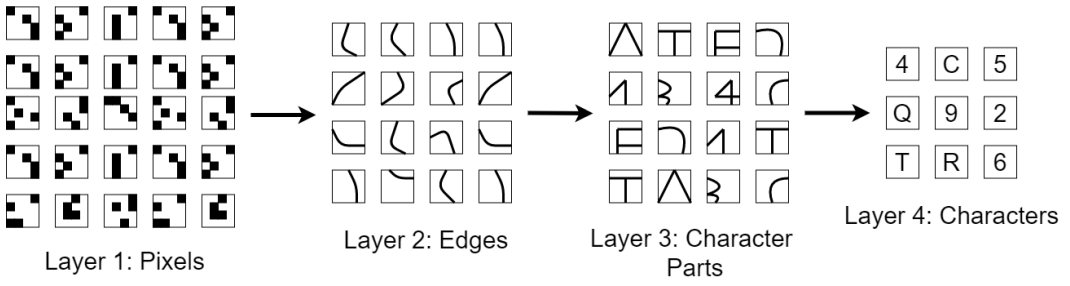


Figure 3.2: *Demonstration of how a Deep Learning model for character recognition would work: the model breaks the features of the character images into a hierarchy of abstraction. Lower layers analyse low level information like the individual pixels, then subsequent layers combine the information from previous layers to recognize, edges, parts, and eventually characters.*

The ability to extract hierarchies of features from data means DL is capable of finding useful feature priors in large amounts of image data, with relatively little tuning from the programmer compared to older ML models. This makes DL a good means with which to define an approximate solution to the problem of colourisation. The difficulty of manually defining a reliable set of priors to guide the colourisation process is solved as DL provides models which are very capable of automatically finding good priors and using them to guide the colourisation process. For an experimental demonstration of how feature recognition is employed by the models in this project to colourise, see 5.2.

3.2.1 Neural Networks

Neural Networks (NNs) are the main systems involved in DL. They are powerful and scalable models which are capable of tackling highly complex ML tasks. The main component of a NN is a Neuron, which takes a set of numerical inputs ($X1, ..., Xn$) sometimes along with a bias value (b) and produces a numerical output (Y).

Inside the neuron, a weighted sum of its inputs is calculated:

$$\Sigma = b + w1 * X1 + w2 * X2. \tag{3.9}$$

Then an activation function is applied to this sum to constrain its result within a given range. There are a variety of different activation functions, however, for this example we will use the common ReLU function:

$$Y = max(0, \Sigma). \tag{3.10}$$

In the case of a fully connected network, each neuron is connected to every other neuron in the previous and successive layers, receiving their inputs from them and propagating their outputs onward to others to eventually produce the expected result Y (see figure 3.4). This inter-dependency of information between the layers is what allows NNs to model hierarchies of concepts as mentioned above. Initial layers will be trained to extract useful information from low level details of the inputs which the next layer will then use to extract more relevant information from.

The network trains itself by tuning the weights and biases (both referred to as "trainable parameters") for each neuron in order to produce an optimal output. The optimal output is found through the use of a loss function which calculates a value representing the difference between the expected output and the output given by the network for a particular input. The backpropagation algorithm is then used to adjust the parameters of each neuron by computing the gradient of the loss function with respect to the parameters for each input. The gradient indicates how the

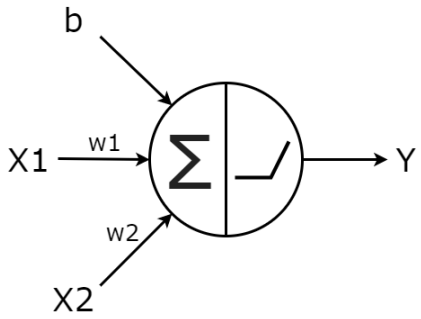


Figure 3.3: *The neuron calculates a weighted sum of its inputs (b, X1, X2), then applies an activation function (ReLU) to that sum and outputs the result (Y).*

parameters need to be changed in order to reach a minima of the loss function. Reaching a minima means that the network is outputting something similar or equal to the expected output.

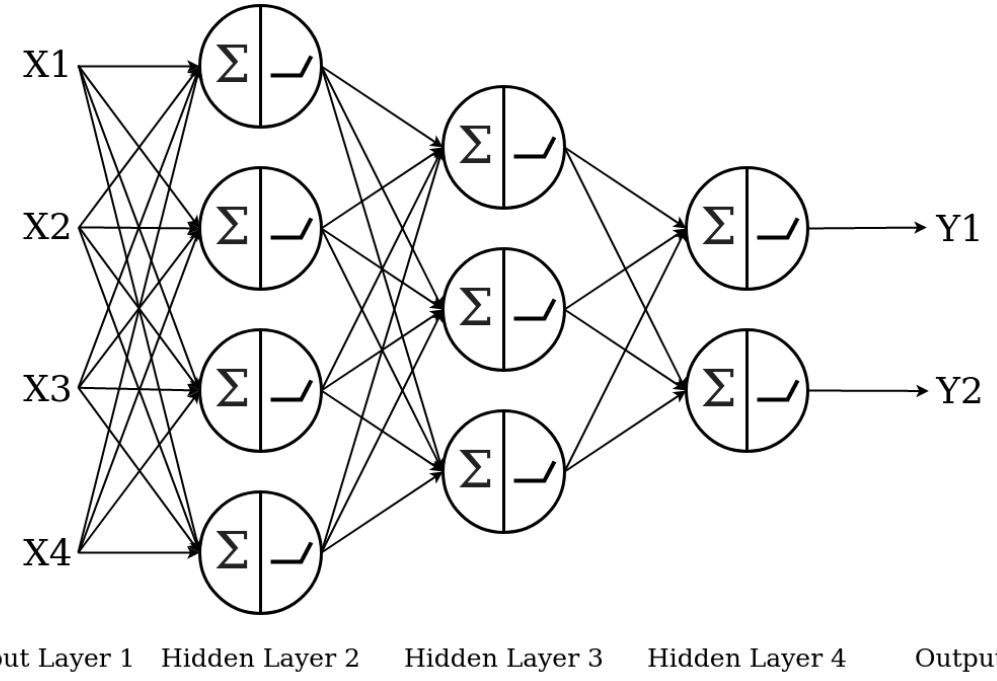


Figure 3.4: *Example of the dataflow structure of a NN. The number of X inputs, layers and Y outputs shown in this example are arbitrary and can vary in real NNs.*

3.2.2 Convolutional Neural Networks

One Deep Learning system used in this project which is particularly effective in handling problems involving image analysis is the Convolutional Neural Network (CNN). These networks are able to extract specific patterns in images and learn how these patterns can combine into meaningful features.

The main element of the CNN is the convolutional layer which contains neurons capable of learning weights and biases for particular objects in an image. The first convolutional layers of a CNN are able to recognise low level features present in specific parts of the image, which

can then be combined into higher level features for later layers to recognise (see figure 3.2). Unlike the fully connected layers of normal NNs, neurons in the first convolutional layer are only connected to particular pixels of the input image, then in successive layers, the neurons are only connected to particular neurons of the previous layer. These connections are made when the neurons or pixels of the previous layer or input fall within the range of a rectangular receptive field called the kernel.

The kernel performs the convolution operation over subareas of the image within its range in order to produce a convolved output. The convolution consists of a matrix multiplication between the values in the portion of the image the kernel passes over and the weights of the kernel itself plus a bias (see figure 3.5).

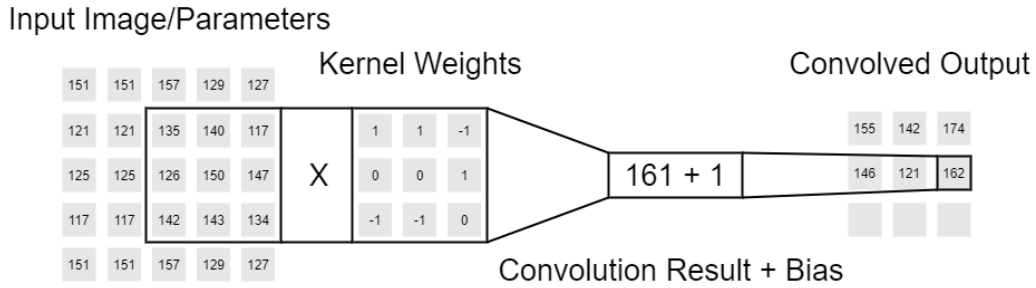


Figure 3.5: *Demonstration of a convolution, the kernel will eventually continue moving across the input matrix until it has convolved.*

The kernel will move over the entire image in strides, for instance 1 stride a turn means it will move along the image one pixel at a time, from left to right, top to bottom. Most convolutional layers have a stride greater than 1 that downsamples the size of the image with each convolution, this downsampling can isolate more important features of the image in the network, as portions which play a greater role in reducing the loss are more likely to be present in later layers after multiple downsamples. This process of downsampling also promotes "translational equivariance", which makes the model more tolerant to small translational changes in the input, i.e. making it easier for the model to notice an important feature in the image regardless of the feature's position.

The size of the kernel is typically an odd dimension (often 3x3 or 5x5) as that allows it to visit particular pixels more than once as it convolves the image. This is useful as sometimes adjacent pixels may be part of features that are dependent on neighbouring pixels on different sides. For example, if a 2x2 kernel has a stride of 2, each stride will visit unique pixels meaning that it will not be able to model any kind of dependency between pixels in the different areas it visits. On the other hand, a 3x3 kernel with a stride of 2 will visit the same pixels on the edges of each area multiple times meaning that it can model a dependency between these pixels on the borders if it needs to.

3.2.3 Generative Adversarial Networks

Generative Adversarial Networks (GANs) are Deep Learning frameworks capable of generating data through an adversarial process. Two networks are trained in parallel, one being the generator and the other the discriminator. The generator generates new outputs from the training data while the discriminator calculates the probability that a sample came from the training data or is the output of the generator[13].

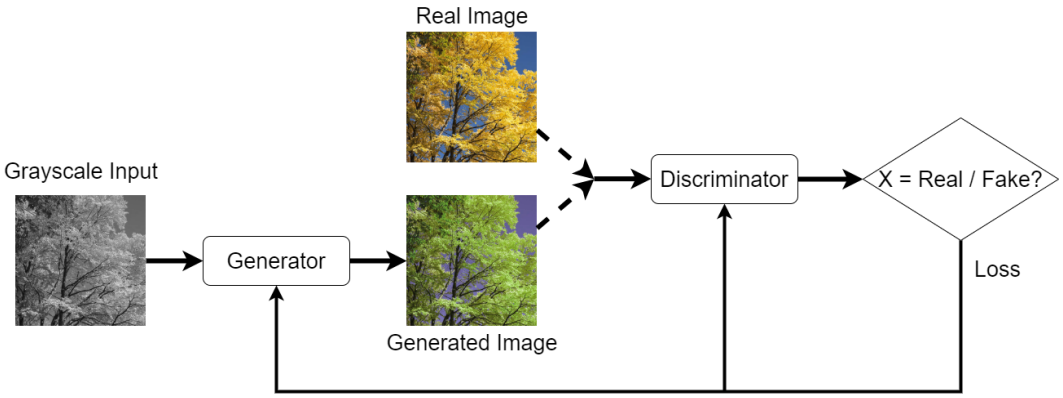


Figure 3.6: *General structure of a GAN used for image colourisation.*

Typically the generator network will be trained to generate candidates from a latent feature space. However for the purpose of image colourisation the generator is a CNN trained to produce colour images from greyscale inputs. The generator will train in order to maximise the probability of the discriminator classifying an image it produces as being a valid colour output. The discriminator is another CNN which maps a colour image to a single output probability representing the image's validity. This discriminator will initially be trained on the real images so it can recognise them, and thereafter will be further trained based on its ability to correctly validate images produced from the generator, attempting to minimise the probability of it making a mistake. Over time through training, the generator should produce better images while the discriminator will become more skilled at recognising generated images.

GANs are an effective method for providing generalised outputs due to the manner in which they are trained. Typical NNs are trained to produce a specific output for each given input, and thus have a tendency to "overfit". A model overfits when it is trained in a manner which conditions it too much to only recognise the data it has been trained on, and is then incapable of producing sufficient outputs for unseen data. When GANs train on the other hand, they use the adaptable output of another network, the discriminator, to determine whether what the generator is producing is valid. This process impedes it from overfitting to a specific dataset as much as a typical NN would due to the constantly changing form of the data it trains on.

GANs are a very attractive method for modelling the process of image colourisation due to their ability to generalise better than typical NNs, allowing them to account for more of the important priors involved in colourisation, across a more diverse range of images. A downside to them however is the difficulty involved in training them, as they can be very sensitive to hyperparameter changes, and often the relationship between the generator and discriminator can be imbalanced and lead to overfitting.

3.2.4 Variational Autoencoders

Variational Autoencoders (VAEs) are another form of Deep Generative model which generate data by sampling from a latent feature space. An encoder network is used to encode a given input to a compressed representation stored in the latent space Z, and then a decoder is used to decompress samples of this latent space in order to reconstruct equivalent outputs[11].

In standard Autoencoders (AEs) the latent space Z is a fixed vector. The larger this vector is, the greater the AE's ability to compress more details. The defining characteristic of VAEs, which sets them apart from AEs, is that the latent space is a distribution composed of a mean vector and standard deviation vector as opposed to a single fixed vector. With a fixed vector latent space,

sampling from each entry of Z will always produce the same corresponding output Y, however with a distribution, latent space samplings can have a range of potential values within a respective standard deviation. In more formal terms, a VAE encodes an input X to a latent distribution $Q(Z|X)$, and decodes a sample from this distribution $P(X|Z)$ to produce a reconstruction of the input.

VAEs are trained with a loss function composed of two terms. The first term is the reconstruction loss, which an AE would also use. This measures how effectively the decoder has learned to reconstruct an input image by sampling from its latent representation. The second term of the loss is the KL divergence which learns how far removed the distribution is from a normal Gaussian, this tries to force the distribution to stay close to a mean of 0 and a standard deviation of 1. Due to the fact that backpropagation cannot run through the neurons that sample from this distribution, VAEs sample from the latent space using a method known as the "Reparametrisation Trick". This trick takes a sample z as the sum of a fixed mean μ plus a standard deviation σ, then multiplied by a value ϵ which is always a standard gaussian with mean 0 and standard deviation 1:

$$z = \mu + \epsilon\sigma, \quad \epsilon \sim \mathcal{N}(0, 1). \tag{3.11}$$

The μ and σ are thus the only variables that need to be trained and which backpropagation needs to run through.

For the task of image colourisation, the VAE needs a second conditional encoder which can take a greyscale image and produce a corresponding latent space representation that is equal or approximate to the colour image's latent representation (see figure 3.7). During training, the training encoder is given a colour image X_{YUV} and the conditional encoder is given the greyscale version of the same image X_Y. The training encoder trains as normal to produce the latent space representation $Q(Z|X_{YUV})$, while the conditional encoder trains to replicate this same latent space for each greyscale encoding $Q(Z|X_Y)$. After the network has been trained, the VAE uses the conditional encoder's latent distribution $Q(Z|X_Y)$ in place of the training encoder's distribution $Q(Z|X_{YUV})$. A greyscale input can then be coloured by passing it to the conditional encoder that produces a latent space mapping, which can then be fed to the decoder to produce a colourised output $P(X_{YUV}|Z)$ (see figure 3.8).

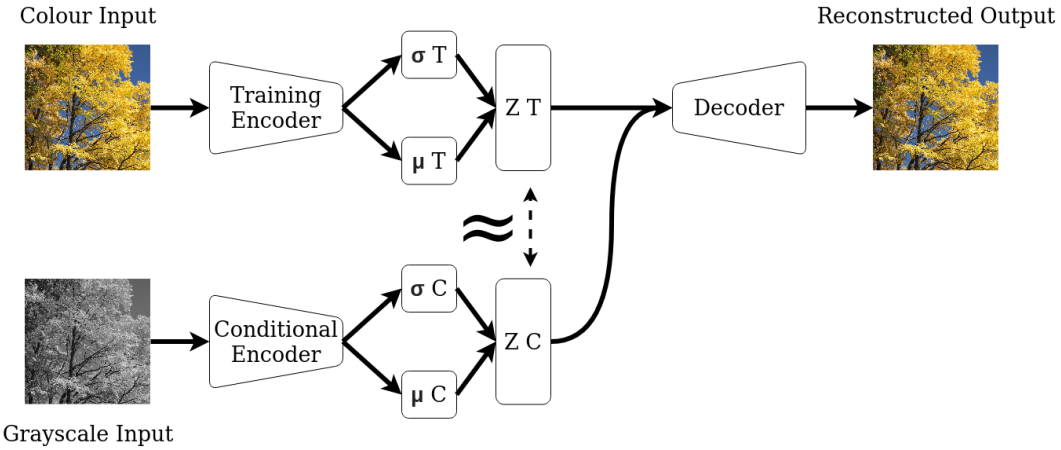


Figure 3.7: *Structure of a colourisation VAE for training.*

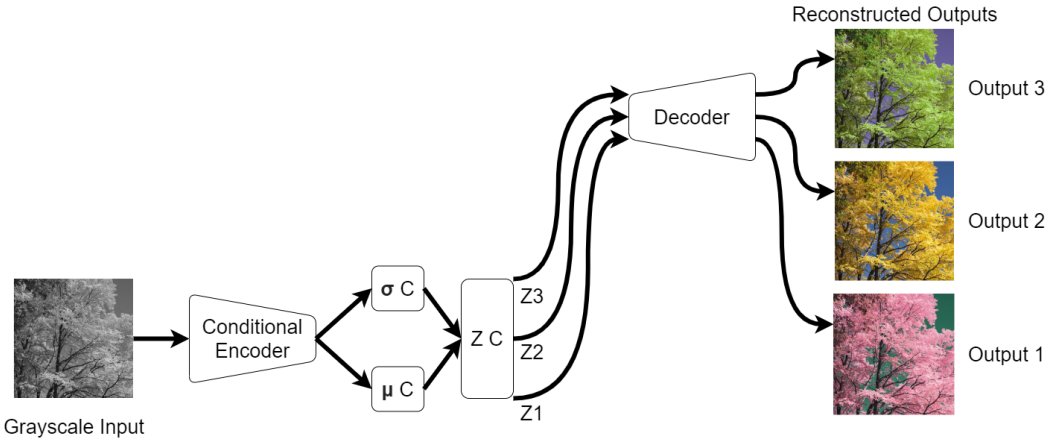


Figure 3.8: *Demonstration of how a colourisation VAE uses the conditional encoder to provide a range of colourisations.*

VAEs are an interesting tool for image colourisation as they are capable of modelling the diverse range of possible colourisations a greyscale image can feasibly correspond to. For each greyscale input, the VAE can map a range of potential latent representations with mean μC and deviation σC, each of which can then produce a different output when fed to the decoder.

3.3 Datasets

Due to the complexity of the problem, the effectiveness of a model's ability to colourise is heavily dependent on the dataset it is trained on. Even very effective DL models can be useless if they are not trained on datasets that are appropriate for the problem. A range of datasets were used to train and test with, each having their own benefits and disadvantages in terms of verifying the effectiveness of the model.

3.3.1 Shape Set

Shape Set is a small dataset that was made specifically for this project. The dataset is designed in order to demonstrate the mechanism NNs use to perform colourisation, which is explained in 3.2. It can specifically show how NNs colourise, by first learning to recognise the features in an image, and then assigning colours corresponding to each feature.

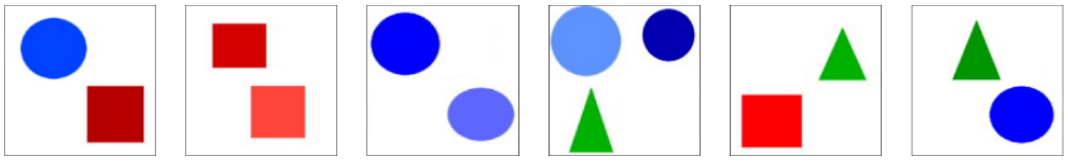


Figure 3.9: *Some sample images from Shape Set*

The images in the Set are composed of circles, squares, and triangles, each corresponding to the colours blue, red, and green respectively. These shapes represent the features that the NN has to learn to recognise in order to determine what colour should be assigned to it. The shapes can appear in any position in the image, meaning that the NN will have to learn how to recognise each one regardless of its location, demonstrating the concept of translational equivariance explained in 3.2.2.

The dataset also includes a couple of extra challenges for the NN. Firstly, the shapes are all presented with different dimensions, for example, no two squares have the same height and width. This forces the NN to recognise features based on an approximate form, as in real images, specific features will often appear in multiple different ways. Another challenge presented by the dataset is that all the colours appear in various shades, which can show whether the NN has learnt to account for the shading, as opposed to simply assigning the exact same colours for each feature.

3.3.2 Cifar-10

Cifar-10 is a widely used collection of 60,000 colour images spread equally amongst 10 different classes. Each image is 32x32 pixels large and thus the entire dataset is very compact at only 163MB. Due to its wide usage, the Keras library contains a simple to use function to conveniently download the dataset to the current workspace. Its small size and easy to use Keras setup made it an ideal dataset to quickly train models with, while also providing a large number of images with a good diversity of features which the network could train on.

Its division into separate equally sized classes also allows the convenient selection of specific classes to train on. This limits the network to training with a more constrained set of images having more features in common, helping it to better identify the most useful features for a given class. Training on a specific class is often a better way to evaluate a network, as datasets with too much diversity can make it difficult for any network to produce a satisfactory output.

The low resolution of the images however makes it an insufficient dataset to train any kind of model that would be used in a realistic scenario, as such a resolution is so low that a lot of smaller features begin to be blurred and unnoticeable. A colourisation model intended for wider use would likely need to colour images with much more detailed features and thus would need to be trained on a higher resolution dataset where such features are more prominent and can be identified by the network.

3.3.3 Faces

Faces is a set of 750 cropped face images taken of people of different genders and ethnicities. The images are all of varying resolutions but for the purpose of training were resized to 128x128, this kept the image small enough to optimise training speed while at the same time keeping most of the main features clearly present in the image.

The people in these images are all facing different directions but generally towards the camera so that their face is clearly visible in each image. This means every image contains the same clear prior of a human face with the same facial features (nose, eyes, mouth, etc.). Training on this dataset is a good test to determine whether the network is capable of identifying a more complex set of symbolic priors than the ones contained in the symbol set in order to aid colourisation, and also serves as a demonstration of how constrained datasets of well categorised images can provide more reliable training results from most models.

The dataset is also a useful training set to evaluate models with after testing them on Cifar-10. The higher resolution of the images means the model has to train on a more complex dataset, however the important features that aid colourisation (facial features) are clearly present in every image. This makes the dataset a good in-between to test a model's ability to handle higher resolutions before presenting it with a challenging diverse dataset of more distinctive images.

3.3.4 Imagenette

Imagenette is a subset of 10 image classes taken from the Image-Net database[3]. Each image is 160x160 pixels large but for the purpose of maximising training speed were resized to 128x128 as was the case for Faces.

The classes of Imagenette are very disparate, with categories ranging from animals to buildings. This makes training on the entire dataset difficult. However, training on each class is a good way to measure how each model adapts to particular categories of images and which categories present more of a challenge to the model.

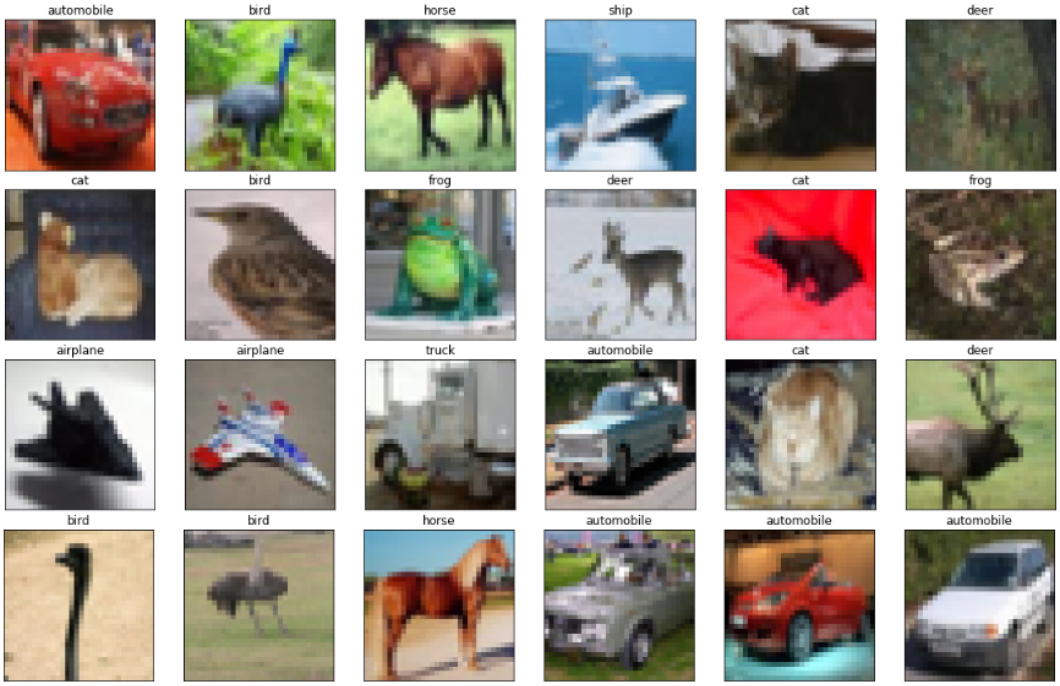
The dataset is also very noisy even for the same class as can be seen in figure 3.10c. The main object of the image varies greatly in terms of orientation, lighting, and size. This makes the dataset very challenging for most networks, even when it is only looking at a particular category. Identifying the important features that can aid colourisation is made much more difficult by the noise and variation of the images.

3.4 Google Colab

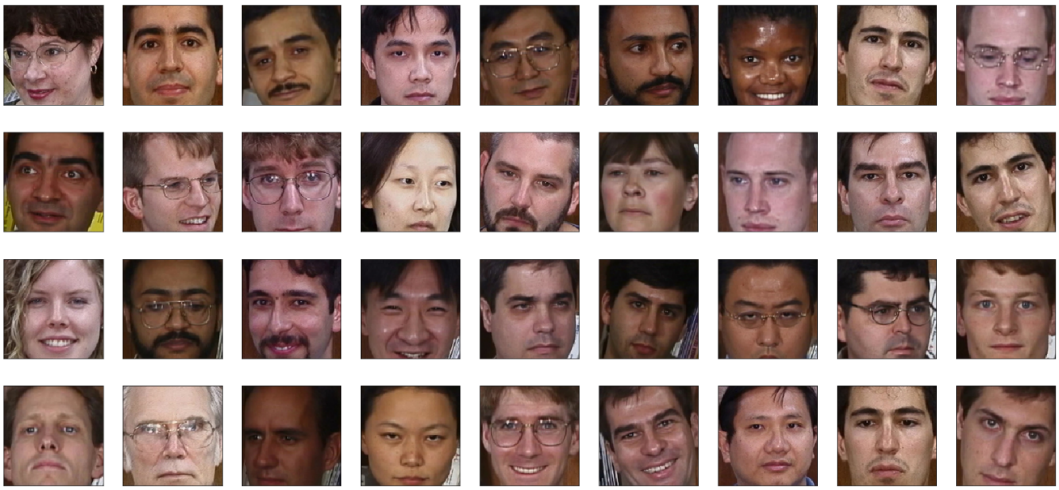
Google Colab is a free, browser-based, Python notebook environment that allows the user to run their code on Google's own cloud hardware. Each model developed for this project was written, trained, and evaluated using Colab as it provided a convenient and free platform on which to develop and run Machine Learning code.

Because Colab is browser-based, each notebook written on it is conveniently accessible from any machine. The notebooks themselves are saved on the user's Google Drive account, and the user is also able to use their Drive to save and load data from within the notebook's code. This Drive integration provided a useful platform on which to host datasets, as well as a space to save and load trained models.

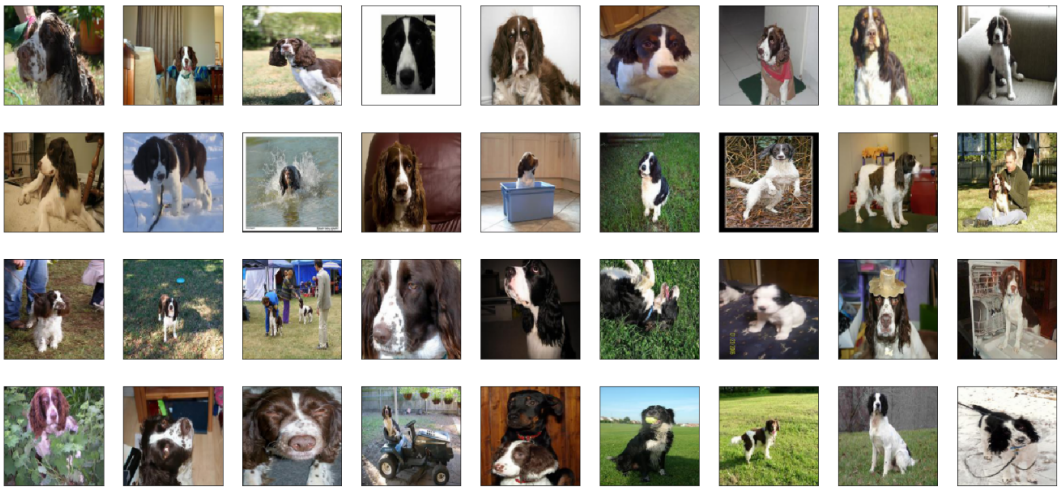
Code run on Colab can use Google's own GPU and TPU hardware accelerators, making it a good platform for NN training which involves a great number of tensor operations that can be carried out faster with such hardware. Code containing training loops can be conveniently run in the background, however a maximum of only two notebooks can be run at any point in time. Notebooks can also only run so long as a browser created session is still running, so closing the browser which a notebook is currently running on will end the training loop. These limitations made it difficult to train models with large datasets or a large number of parameters as the browser needed to be kept open for extensive periods of time.



(a) *Sample images from Cifar-10 with corresponding classes above.*



(b) *Sample images from Faces.*



(c) *Sample images of the Dog category from Imagenette.*

Figure 3.10: *Third-party dataset samples.*

4 Implementation

In this chapter we will explain the designs of each model and the reasoning behind their design choices.

4.1 Convolutional Neural Network

The first model which was developed was a Convolutional Neural Network (CNN), which is a Neural Network (explained in 3.2.1) made using convolutional layers (explained in 3.2.2). This was a deterministic model capable of producing a single corresponding colourisation for each unique greyscale input. This network was developed first as it was the simplest viable DL solution, and because it could then be reused as part of the later generative models after a suitable design for it was found.

As explained in 3.1.2, the colourisation model needs to take a 3-dimensional tensor as input, representing a greyscale image in the form of the Y channel of a YUV image, the model must then find an algorithm to produce the corresponding U and V colour channels of that YUV image. This means that the input to the network will be a rank 3 tensor (Shape: Width, Height, 1) and the output will be another rank 3 tensor (Shape: Width, Height, 2).

4.1.1 Architecture

Typically when implementing ML algorithms, a range of candidate parameters are chosen and then an exhaustive search is performed over each subset in order to find the optimal configuration that optimises the model. When designing Neural Networks this approach is not feasible due to the large amount of parameters involved as well as the long periods of time necessary to train most models. Designing a model ultimately had to be done by repeatedly testing and tuning models in order to finally find one with the best performance that overfits or underfits the least.

Models overfit or underfit based on their "capacity" meaning their ability to fit a wide variety of functions. Low capacity models can underfit the dataset as they lack the capacity to memorise important information, while high capacity models can overfit by memorising too many properties of the training data that won't help their performance on unseen data[12, p.110]. The primary parts of a CNN which govern its capacity are the number of hidden layers it has, the number of filters present in these layers, and the kernel size and strides for each layer which determine the downsampling (explained in 3.2.2). These main elements were the most important parts of the CNN that needed to be decided through repeated testing before other parameters could be determined.

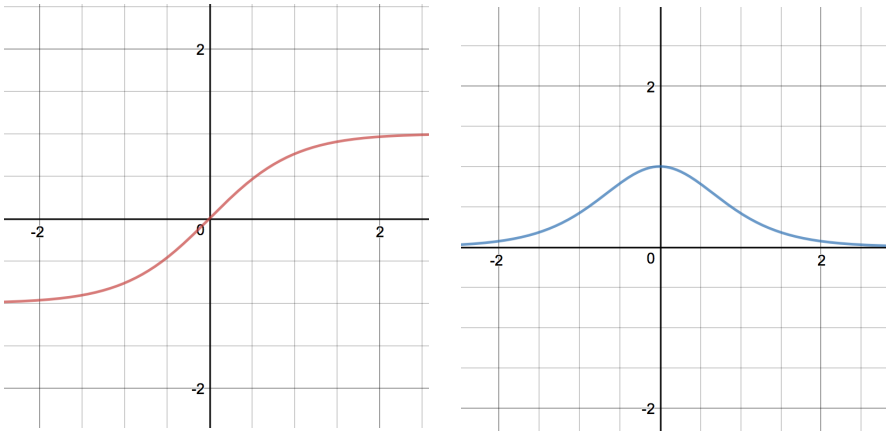
Along with the general structure of the network, there were some important parameters that also needed to be decided before the model could be trained. A loss function first needed to be chosen to indicate the error between the output of the network and the ground truth. The task the network had to perform can be seen as a form of regression, where it estimates the relationship between an image's greyscale features and a colour encoding, and for most regression problems the Mean Squared Error (MSE) is the most commonly used loss function. MSE calculates the

loss as the mean of the squared sum of the differences between n output values y' and n expected values y:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_i')^2. \tag{4.1}$$

Another popular regression loss function was Mean Absolute Error (MAE), which is a similar formula however takes the absolute difference of the sum of output and expected values as opposed to the squared root. MAE was not used as it penalises larger errors less than MSE does, leading to less precise output values.

Activation functions needed to be chosen for each hidden layer along with a precision activation for the output layer. Using computationally expensive activations in the hidden layers would slow training considerably for diminishing improvements on performance, a more complex activation on the output layer however is affordable as the precision of the final result matters more than the accuracy of the many feature activations. ReLU functions (explained in 3.10) were chosen for the inner layers of the network as they were the simplest and thus the quickest for the network to calculate. For the output layer, tanh functions were chosen, which are defined as:



(a) $tanh(x) = \frac{2}{1+e^{-2x}} - 1$ **(b)** $tanh'(x) = 1 - tanh(x)^2$

Figure 4.1: *$tanh(x)$ activation function with its derivative.*

Tanh functions have steep derivatives that help the network minimise its contribution to the loss relatively quickly. This is because the backpropagation algorithm adjusts the network's weights in the direction of the steepest descent surface defined by the total error observed, so larger derivatives correspond to more significant weight updates[1].

An optimiser also needed to be chosen, which refers to a function used to find minima of the loss function. As a preliminary optimiser, Stochastic Gradient Descent (SGD) was used, as it is the most basic optimiser and could be used as a good baseline from which to measure the performance of other optimisers when parameter tuning later on.

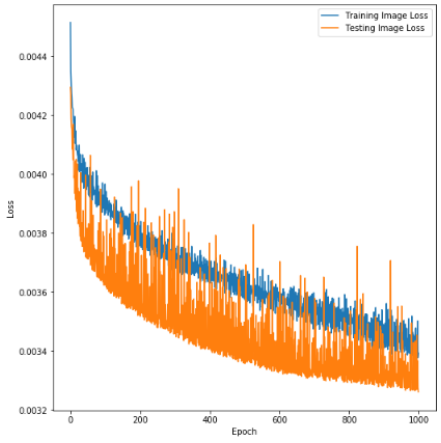
To design the network, two models were initially chosen, one very shallow model with a small number of layers and filters (see figure A.3) and a contrasting very deep model with a large number of layers and filters (see figure A.4). The models were both trained on the same data and the output of the loss over training was plotted in order to determine how far each model was from a well fitting design.

After repeating the two model method with an increasingly shallow deep model and an increasingly deep shallow model, an in-between model was eventually found that achieved the best loss

Layer (type)	Output Shape	Param #
conv2d_48 (Conv2D)	(None, 30, 30, 8)	80
conv2d_49 (Conv2D)	(None, 15, 15, 16)	1168
conv2d_50 (Conv2D)	(None, 8, 8, 32)	4640
conv2d_51 (Conv2D)	(None, 4, 4, 128)	36992
up_sampling2d_18 (UpSampling	(None, 8, 8, 128)	0
conv2d_52 (Conv2D)	(None, 8, 8, 128)	147584
up_sampling2d_19 (UpSampling	(None, 16, 16, 128)	0
conv2d_53 (Conv2D)	(None, 16, 16, 32)	36896
conv2d_54 (Conv2D)	(None, 16, 16, 16)	4624
up_sampling2d_20 (UpSampling	(None, 32, 32, 16)	0
conv2d_55 (Conv2D)	(None, 32, 32, 2)	290

Total params: 232,274

(a)



(b)

Figure 4.2: *Final basic model with accompanying loss. Notice how the training and testing loss are continuing to descend, with enough epochs the training loss will go below the validation loss however the important thing is that both the losses closely descend together. This means that the network has just enough capacity to store the most important information necessary for the problem that can help it generalise without overfitting.*

output for both the training and testing dataset (see figure 4.2). With the basic structure decided, the next step was tuning the higher level parameters.

4.1.2 Hyper-Parameter Tuning

Unlike the structure of the neural network, the hyper-parameters can feasibly be tested using an exhaustive search. Unfortunately however, the available libraries (Hyperas and ScikitLearn) that could do this for Keras models were both incompatible with Google Colab. These libraries needed to be run on local runtimes where they had access to many of the libraries and files that were being used within the code, which is not possible on Colab's hosted runtime environment. Hyper-parameters thus had to be chosen by running repeated simulations on ranges of potential values.

The first hyper-parameter that had to be chosen was batch size, which is the number of samples the network loads and trains on at each iteration. This value was conditioned mostly by the amount of RAM available on Google Colab, as even a relatively low memory dataset like Cifar-10 could exhaust Colab's available RAM when trained on a large enough network. The batch size thus needed to be at most 5,000 in order to train most models. The choice of batch size also has a significant impact on model learning; lower batch sizes result in less accurate updates, as the model will calculate gradients based on the batch it is currently working on which often is not representative of the wider training dataset. This impact that low batch sizes can have results in a more stochastic training process and in some situations can be beneficial if the network can get stuck in local optima or overfit easily. Small batch sizes however can significantly slow down training, as more batches need to be transferred from storage to RAM in order to feed to the model.

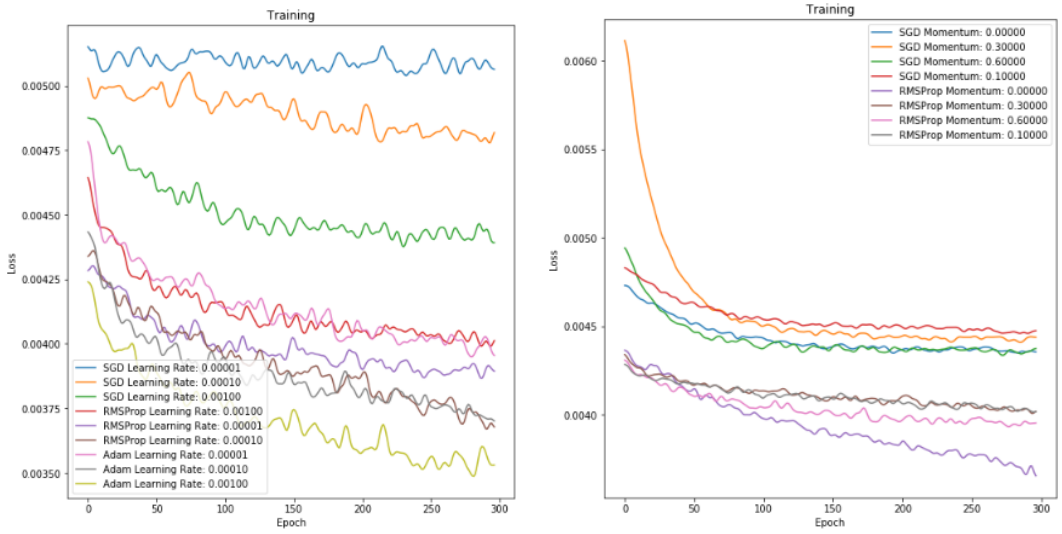
Although lower batch sizes lead to a better loss value after the same amount of epochs, the time required to train on lower sizes got significantly higher for diminishing improvements on the final loss. The final batch size that was chosen was 512, as it provided a fairer balance between training time and performance.

A better optimiser also needed to be chosen, and there were two which both seemed like good

Table 4.1: *Impact of different batch sizes on loss and training time.*

Batch Size	Final Loss after 100 Epochs	Training Time
1024	0.0039	2 min 27 sec
512	0.0038	2 min 32 sec
128	0.0036	3 min 43 sec
32	0.0035	5 min 30 sec
16	0.0034	7 min 6 sec

potential candidates: RMSProp, and Adam. Both of these are popular extensions to the classic SGD and can often achieve better results[9]. An important parameter that needed to be tested alongside the optimiser was the learning rate. Different optimisers can have very different performances based on their learning rates and so these two things needed to be tested together for a more reliable comparison. The learning rate dictates how much the model changes its parameters at each iteration in order to minimise the loss. Larger learning rates train faster, but can result in the model converging too quickly and overfitting before being capable of accounting for the less evident features that can help it generalise. If the learning rate is too small though it can result in the loss plateauing early as it may be incapable of making significant changes to the model. A general range of different rates were tested on the network for each optimiser (see figure4.3a) and the most successful configuration proved to be Adam with a learning rate of 0.001. It made sense that Adam proved so effective as it is commonly employed for Computer Vision tasks[10]. This is likely due to the fact that Adam is particularly effective for problems with "noisy gradients", meaning the network's parameters can still be tuned effectively even if the tuning information given by the backpropagation algorithm can vary greatly as it can for this problem.



(a) *Effect of different learning rates and optimisers on training the DCNN. The optimiser that achieved the best results was Adam with a learning rate of 0.001.*

(b) *Effect of different momentum values and optimisers on training the DCNN. Momentum does not seem to provide any noticeable benefit for training with any of the optimisers.*

Figure 4.3

Although Adam seemed to be the best optimiser, RMSProp has a particular hyper-parameter called "momentum" which in some cases can be very effective at improving model performance.

Momentum helps prevent the network getting stuck in local minima by also recording the direction its gradients previously moved towards. Higher momentum means that steps will follow a direction which is closer to the direction the previous step took. Tests were also run to determine if RMSProp with momentum could outperform Adam (see figure 4.3b), however any amount of momentum greater than 0 only worsened training. This is likely because of the high dimensionality of the data, as there would probably be a large variation in the ways the gradient could move in order to decrease the loss. Constraining it to a more fixed direction thus only prevented it from moving in the correct direction.

4.1.3 Data Augmentation

Data augmentation was used to help prevent the model from overfitting by expanding the training dataset with slightly modified versions of the original images. This technique helps the model as the modified images will still contain the same important features however will be altered in such a way that the network has the opportunity to recognise them presented in another manner, promoting translational equivariance (explained in 3.2.2). The choice of how this data is augmented is important as it can also contribute to overfitting if not done correctly. If the modifications do not change the image significantly enough then the network will be exposed to a very similar image multiple times.

The data augmentation techniques that were tested for training were:

1. Horizontal flipping: the columns of each image were reversed.
2. Rotation: each image was randomly rotated within a range of 60 degrees.
3. Width and Height shifting: the pixels of each image were moved either horizontally or vertically by 20%.
4. Brightness: images were randomly darkened or brightened within a 50%-150% range.
5. Zooming: randomly zooms in or out on images within a 50%-150% range.

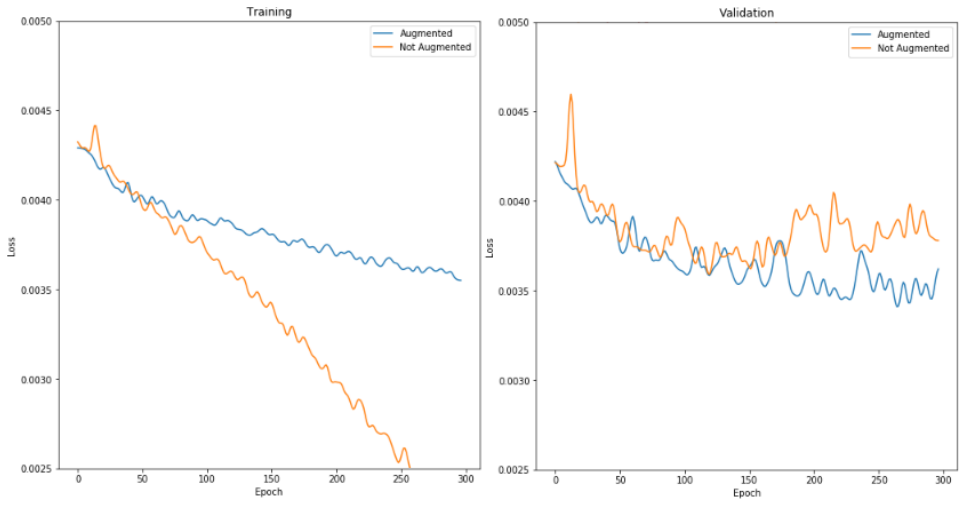


Figure 4.4: *Effect of data augmentation on training and validation loss. Augmentation results in the validation loss much more closely approximating the training loss, meaning that the model is generalising better with it.*

These different techniques were tested individually to see what impact they had on the model's performance (see figure A.1), and after the best augmentation techniques were found they were tested together to find which combinations worked best (see figure A.2). In the end, Horizontal

Flipping, Width and Height shifting, and Rotation were used to augment the dataset used by the CNN. These forms of augmentation had a significant impact on improving the model's ability to generalise (see figure 4.4).

4.1.4 Regularisation

One final technique used to encourage the model to generalise was regularisation, which refers to a set of techniques used to constrain the network to learning sparse features, preventing it from simply memorising specific details of the training dataset.

The first regularisation technique that was tried was L1 and L2 regularisation. Both of these methods work by adding an extra penalty term to the loss function to prevent large weight changes in the model, this penalty term being the norm of the model's weight vector (regularisation for the model's biases also exists however its impact on training was small). These forms of weight regularisation are named after the particular norm with which they're calculated.

The L1 norm is defined as the sum of the magnitudes of the weights:

$$||w||_1 = |w_1| + |w_2| + ... + |w_N|. \tag{4.2}$$

The main attraction to L1 regression is that it shrinks the less important features' coefficient to zero, removing some features altogether. This works well for feature selection like colourisation as we have a huge number of features to consider with some that may not aid the colourisation process at all.

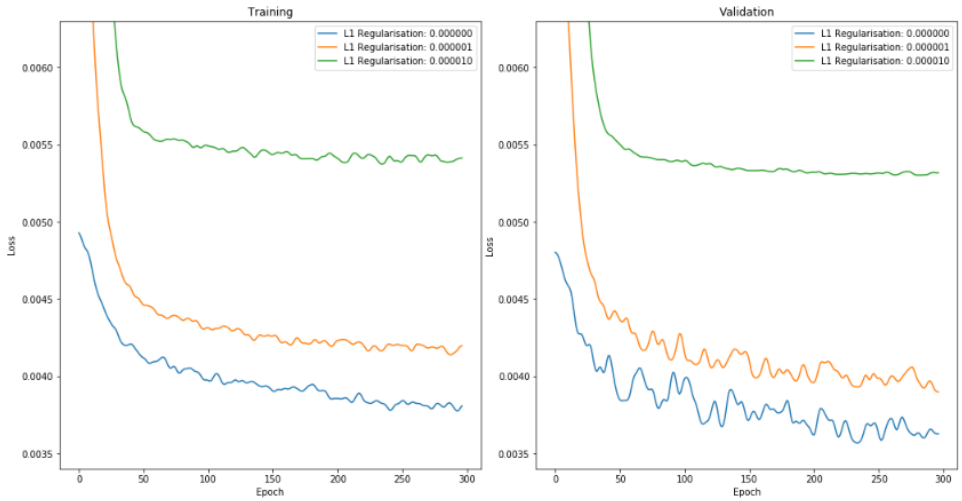


Figure 4.5: *Training and Validation loss of various L1 regularisation values on the CNN. Any amount of L1 regularisation only appeared to worsen the both the training and validation loss.*

The L2 norm is defined as the square root of the sum of the square weights:

$$||w||_2 = \left(|w_1^2| + |w_2^2| + ... + |w_N^2|\right)^{1/2}. \tag{4.3}$$

L2 regression is less severe, as it never decreases any of the features to 0 (although some can come very close), this is ideal in situations where there is always some level of dependency between the features which should be accounted for. To be certain that either could improve the model's performance they were both tested (see figure 4.5 and 4.6).

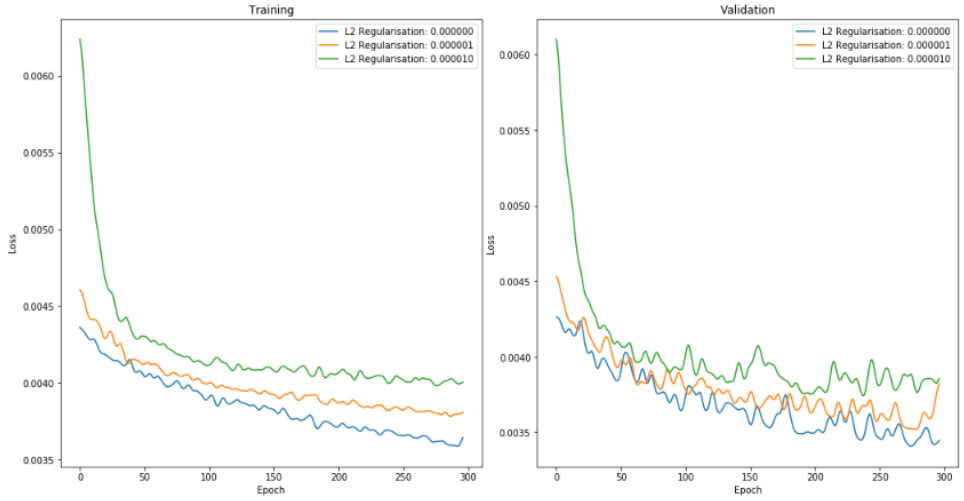


Figure 4.6: *Training and Validation loss of various L2 regularisation values on the CNN. Just like L1 regularisation, L2 regularisation only worsened the performance of the model.*

Another regularisation method was Dropout, which entails excluding random sets of neurons from each iteration of the training process based on a given probability. This can be very effective at reducing overfitting because it prevents neurons from becoming overly dependent on other neurons in previous layers, thus helping them recognise more general features by forcing them to consider information from the neurons that were not excluded. A reasonable range of dropout values were tested (see figure 4.7) and a final small dropout value of 0.2 was used.

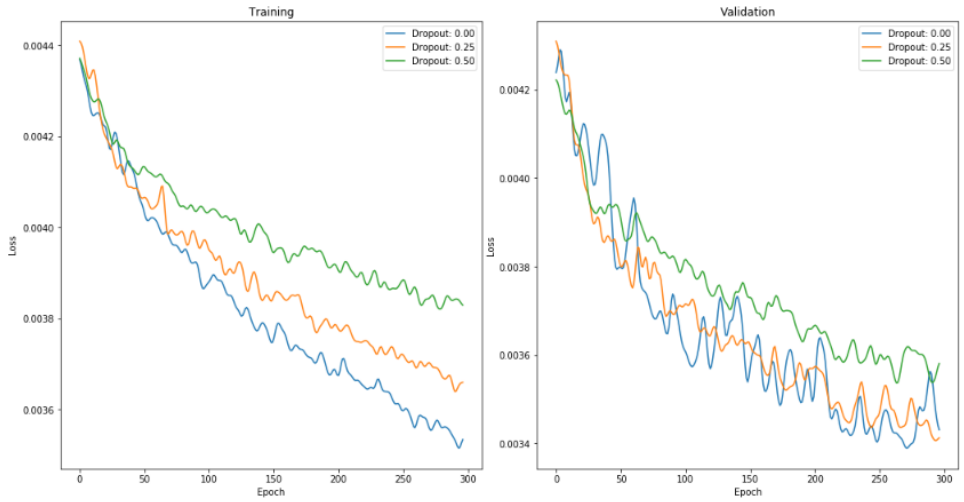


Figure 4.7: *Training and Validation loss of various Dropout probabilities on the CNN. Higher dropout values caused the training loss to plateau earlier however the middle dropout value of 0.25 did seem to have a small noticeable improvement on the stability of the validation loss.*

4.2 Convolutional Generative Adversarial Network

After an adequate CNN design was found, it could then be used as the base model for the generator of the Convolutional Generative Adversarial Network (CGAN). The CGAN is an

implementation of the GAN (explained in 3.2.3) designed to solve computer vision problems through the use of convolutional layers. It was the first generative model that was implemented due to its relatively widespread use for the task of image colourisation, and so there were a substantial number of reference projects available that could be consulted in order to support our own implementation.

Like the CNN, the CGAN needs to take a 3-dimensional tensor as input, containing the Y channel of a YUV image, and is trained to model general mappings for U and V colour channels of that YUV image. It thus has the same input and output shapes as the CNN.

The repeated tuning and testing process that was used to design the CNN couldn't be relied on when deciding the structure and parameters of the CGAN, due to the considerably longer amount of time needed to train it. Many of the design choices thus had to be made using suggestions and based on previously used methods that were successful in other projects. The relatively long amount of timed required to train the CGAN is demonstrated in 5.1.

4.2.1 Discriminator

The first step was to design a discriminator that could accurately validate an image as being real or generated. The discriminator model takes as input a 3D tensor containing the U and V channels of a YUV image (Shape: Width, Height, 2), and outputs a prediction between 1 and 0 representing the degree of belief for whether the image is real or fake. It was implemented as a CNN, but unlike the generator, it convolves from the UV tensor to a single value output as opposed to convolving from a Y tensor to a UV tensor.

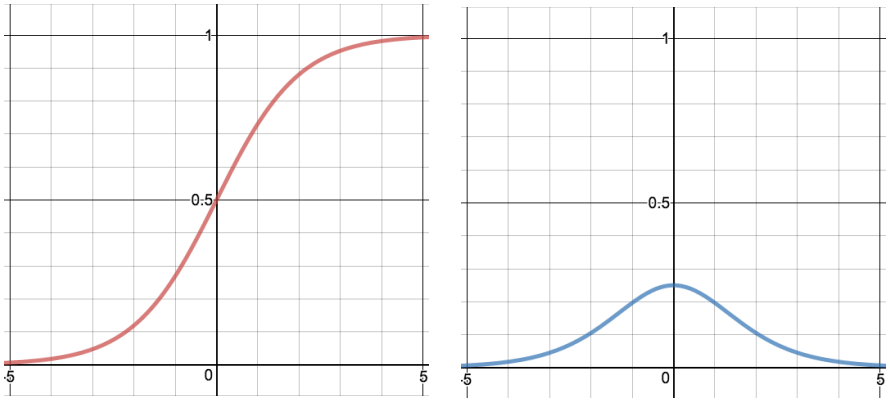
The most important thing that had to be considered when designing the capacity of the discriminator was that it should only be effective in proportion to the generator, if the discriminator is too good then its outputs will be close to the two extremes of either 1 or 0 and the generator will struggle to tune itself proportionate to the error it is making. On the other hand, if the discriminator is not accurate enough, its evaluations can be unreliable and it could mislead the generator to producing poor outputs. The discriminator was made to be of a higher capacity than the generator so it could give quality information without being too accurate.

With regards to the loss function, the task the discriminator had to perform can be seen as a form of classification, as it classifies an image as being real or fake. For most classification tasks, Cross-Entropy is used, and as this only had to perform a binary classification, a more specific Binary Cross-Entropy (BCE) loss function could be used. BCE can be defined with the following equation, where y is a binary value representing the correct prediction, and p is the prediction probability:

$$BCE = -\left(y \log(p) + (1 - y) \log(1 - p)\right). \tag{4.4}$$

Like the CNN, Adam was chosen as the optimiser for the discriminator. This is because the same computer vision challenges which Adam helps the CNN with also apply to the discriminator, as both networks rely on effective feature extraction to produce optimal outputs.

For the activation functions, ReLU's were also used for the hidden layers for the same reasons they were used in the CNN, but for the output a Sigmoid function was used. The sigmoid function is defined as:



(a) $sig(x) = \frac{1}{1+e^{-x}}$ *(b)* $sig'(x) = sig(x)(1 - sig(x))$

Figure 4.8: *$sig(x)$ activation function with its derivative.*

Although the sigmoid's gradients are not as sharp as tanh's, its outputs are bound to a 0–1 range which is ideal for providing the needed binary output.

4.2.2 Generator

While the CNN we already designed served as a good base model for the CGAN's generator, some alterations had to be made in order for it to be viable as a generator.

Batch normalisation was added at particular points in between convolutional layers, which standardises the input of each layer so that it has a mean of 0 and a standard deviation of 1. This means that very diverse images can be made part of a closer distribution which the network can work with more easily, helping it generalise better. Batch normalisation is strongly recommended for CGANs[6], as it stabilises learning and helps deal with training problems due to poor initialisation, helping gradients to flow. Batch normalisation was only applied to most of the layers after the input and before the output layer, as applying it to every layer lead to instability.

The generator was also compiled without a loss function. This is because the generator is not trained based on its own performance, but on the performance of the discriminator. More updates are made to the generator when the discriminator is giving correct evaluations, and less updates are made otherwise. The loss of the generator is given by the loss of the combined model which is explained in the next section.

Dropout was also removed from the generator. While in the case of the CNN it aided generalisation, for the CGAN it only added a further handicap to the generator, making it less capable of producing outputs which the discriminator would consider real. The general adversarial method which a CGAN trains with should be enough to aid the model's ability to generalise on its own, so regularisation methods like dropout were no longer necessary.

4.2.3 Combined Model

The final CGAN model was implemented by stacking the generator and discriminator networks into a combined model that allowed them to train together. Within the model, the generator can generate samples that are then passed as inputs to the discriminator, then the outputs of the discriminator are used to indicate the loss of the generator in order to update it.

For each training step, the discriminator is trained first, then the generator is trained after. The discriminator trains on a batch of the UV tensors from the training dataset based on the expected

output of them being classified as real, then it trains on the batch of UV tensors produced by the generator based on the expected output that they're fake. The discriminator's total loss is subsequently calculated as the sum of the BCE losses returned from training on the real and fake batches. The generator is then trained after as part of the combined model and the loss is calculated as the BCE loss between the generated images being evaluated as real and the discriminator's own evaluation. I.e. the y term of the BCE function is that each generated output is real and the p term is what the discriminator actually evaluated the output as.

Layer (type)	Output Shape	Param #
conv2d_158 (Conv2D)	(None, 32, 32, 16)	304
conv2d_159 (Conv2D)	(None, 16, 16, 16)	2320
conv2d_160 (Conv2D)	(None, 16, 16, 32)	4640
conv2d_161 (Conv2D)	(None, 8, 8, 32)	9248
conv2d_162 (Conv2D)	(None, 8, 8, 64)	18496
conv2d_163 (Conv2D)	(None, 4, 4, 128)	73856
conv2d_164 (Conv2D)	(None, 4, 4, 256)	295168
flatten_11 (Flatten)	(None, 4096)	0
dense_11 (Dense)	(None, 1)	4097

Total params: 408,129

(a) *Discriminator.*

Layer (type)	Output Shape	Param #
conv2d_165 (Conv2D)	(None, 30, 30, 8)	80
conv2d_166 (Conv2D)	(None, 15, 15, 16)	1168
batch_normalization_47 (Batc	(None, 15, 15, 16)	64
conv2d_167 (Conv2D)	(None, 8, 8, 32)	4640
batch_normalization_48 (Batc	(None, 8, 8, 32)	128
conv2d_168 (Conv2D)	(None, 4, 4, 128)	36992
up_sampling2d_33 (UpSampling	(None, 8, 8, 128)	0
batch_normalization_49 (Batc	(None, 8, 8, 128)	512
conv2d_169 (Conv2D)	(None, 8, 8, 128)	147584
up_sampling2d_34 (UpSampling	(None, 16, 16, 128)	0
conv2d_170 (Conv2D)	(None, 16, 16, 32)	36896
batch_normalization_50 (Batc	(None, 16, 16, 32)	128
conv2d_171 (Conv2D)	(None, 16, 16, 16)	4624
up_sampling2d_35 (UpSampling	(None, 32, 32, 16)	0
conv2d_172 (Conv2D)	(None, 32, 32, 2)	290

Total params: 233,106

(b) *Generator.*

Figure 4.9: *Final layer structure for the CGAN's discriminator and generator.*

The combined design was compiled with Adam and a learning rate of 0.0002 and beta of 0.5, which is recommended when training CGANs[2]. These parameters are different from the CNN because of the different way the loss is calculated, and because the added momentum from the beta value allows the generator to be more agile when reacting to evaluations from the discriminator. The data augmentation used in the CNN also was not employed for the CGAN, as there is not the same need to reduce overfitting due to the generalising ability of the adversarial model. The training batch size was 512 as it was successful for the CNN and would work fine with the CGAN for the same reasons.

4.3 Convolutional Variational Autoencoder

The final generative model to be implemented was a Convolutional Variational Autoencoder (CVAE), a convolutional implementation of the VAE (explained in 3.2.4) intended for computer vision tasks. The CVAE was developed after the CGAN, as its implementation was less documented and required more research, and because it has a relatively complex custom loss function that was difficult to implement. The output the CVAE produces is non-deterministic as it is drawn from a random sampling of the input's embedding in the latent feature space. Each output thus tends to be more or less different based on the way the model was trained.

When training the model, the training encoder takes a rank 3 tensor as input, containing every channel of a YUV image (Shape: Width, Height, 3), while the conditional encoder only takes a

tensor containing the Y channel for the same image (Shape: Width, Height, 1). During evaluation, the conditional encoder is given a Y channel tensor and will produce a latent embedding for it which can then be passed to the decoder to produce a corresponding YUV channel tensor for the image (Shape: Width, Height, 3).

The CVAE did not require the same extensive amounts of time to train as the CGAN did (see 5.1), so more of the design choices for it could be found through experimentation.

4.3.1 Training Encoder

The CNN was also used as a base for the training encoder, however only a portion of it could be reused. While the CNN would convolve from a Y tensor to a UV tensor, the training encoder needed to convolve from a YUV tensor to a latent space composed of a mean vector, a standard deviation vector, and a sampling vector. The first half of the CNN was thus taken as a base model, when it convolves the input down to its smallest representation and before it begins to upsample, then at this half point the latent space layers were added.

The training encoder takes the entire YUV representation, as opposed to only the UV channel like the CGAN does, because the Y channel will always be available anyway through training as it is needed by the conditional encoder. Having the extra Y channel input gives the training encoder more information to work with that might help it recognise more of the important features of the image.

The latent space was implemented as a layer that flattens the lowest convolution, then passes the data in 1-Dimensional form to a dense layer that distributes it to two other dense layers representing the mean and standard deviation latent vectors respectively. A third sampling dense layer was then added after that took data from both the latent space layers to produce a sample using the reparametrisation trick (explained in 3.11).

4.3.2 Conditional Encoder

The conditional encoder had the same structure as the training encoder, the only difference being that for its input it takes a Y tensor (Shape: Width, Height, 1), instead of a UV tensor (Shape: Width, Height, 2). It was given its own latent space structure which was trained to approximate the contents of the training encoder's latent space.

4.3.3 Decoder

The decoder takes a 1-dimensional feature input from the latent space, then reshapes it into a 3-dimensional representation. This representation is then passed to transposed convolution layers. These layers model the inverse of a convolution in order to upsample images based on a custom kernel which the layer develops through training. The transposed convolutional layers will gradually upsample the latent input to take the shape of a UV tensor, the corresponding Y tensor for the UV image will then be concatenated onto it to produce the resulting colour image.

The decoder only convolves to produce the UV tensor because the Y tensor will always be available to it, whether through sampling or training, as it is always needed as an input to the conditional encoder in either case. Leaving the decoder to produce the Y tensor as well would have only added another task to the model, making the problem more complex.

4.3.4 Loss Function

None of the independent networks were compiled or given loss functions as they would all be trained through the loss function of the combined VAE model. As explained in 3.2.4, the loss function for a VAE is composed of two terms, a reconstruction loss and a KL divergence. The

general equation for the loss is then the sum of these two terms, however in our implementation, the reconstruction term is also weighted by a variable W:

$$Loss = W\left(MSE(X_{YUV}, Y_{YUV})\right) + KL\left(p(z|X_{YUV}), q(z|X_Y)\right). \tag{4.5}$$

The reconstruction loss is calculated with MSE, where the loss is the MSE between the n inputs to the training encoder X_{YUV} and the n outputs from the decoder Y_{YUV}:

$$MSE(X_{YUV}, Y_{YUV}) = \frac{1}{n}\sum_{i=1}^{n}(X_{YUV} - Y_{YUV})^2. \tag{4.6}$$

The weighting imposed by W on the reconstruction term is used to determine how varied the distribution should be for each input. Having an unweighted reconstruction loss means more consistent outputs, while having a weighting that decreases the reconstruction loss results in a greater diversity of possible outputs for a given input.

This technique is very similar to the one applied in disentangled VAEs (β-VAEs). These implement a hyper-parameter β inside their loss functions that weighs how much the KL divergence is present in the loss. This weighting discourages correlation between neurons in the latent space, pushing each one to learn something unique about the input data[14]. Our implementation weights the reconstruction loss on the other hand, as that has a much larger value than the KL divergence, and so its weighting has a bigger impact on diversity.

The implementation of our colourisation VAE has a special KL divergence function that accounts for the divergence of the conditional encoder's latent distribution as well as the training encoder's. The KL divergence can be formalised as follows:

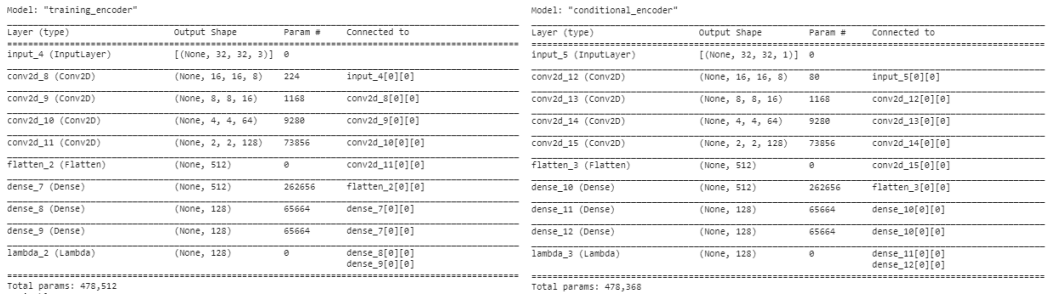
$$\begin{aligned} KL\left(p(z|X_{YUV}), q(z|X_Y)\right) &= -\int p(z|X_{YUV}) \log q(z|X_Y) dx + \int p(z|X_{YUV}) \log p(z|X_{YUV}) dx \\ &= \frac{1}{2}\log(2\pi\sigma_2^2) + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2}(1 + \log 2\pi\sigma_1^2) \\ &= \log\frac{\sigma_2}{\sigma_1} + \frac{\sigma_1^2 + (\mu_1 - \mu_2)^2}{2\sigma_2^2} - \frac{1}{2}, \end{aligned} \tag{4.7}$$

where $p(z|X_{YUV})$ is the probability distribution function of the training encoder's latent mapping z for a YUV input X_{YUV}, which can also be expressed as the latent normal distribution $N(\mu_1, \sigma_1)$. The pdf for the conditional encoder's z for a Y input X_Y is $p(z|X_Y)$ and can also be expressed as the normal distribution $N(\mu_2, \sigma_2)$.

A more in depth explanation of how the KL equation is formulated can be found at [4].

4.3.5 Combined Model

Like the CGAN, the different networks were combined into a model for the purpose of training. The model was compiled with the loss function and trained on in order to adjust the weights of the encoders and decoder. After training, the combined model was no longer necessary as sampling could then be done by passing data to the conditional encoder and the decoder. This is done by first feeding a Y tensor to the conditional encoder to retrieve a latent embedding, then passing it to the decoder along with the Y tensor to produce a YUV output.



Model: "training_encoder"

Layer (type)	Output Shape	Param #	Connected to
input_4 (InputLayer)	[(None, 32, 32, 3)]	0	
conv2d_8 (Conv2D)	(None, 16, 16, 8)	224	input_4[0][0]
conv2d_9 (Conv2D)	(None, 8, 8, 16)	1168	conv2d_8[0][0]
conv2d_10 (Conv2D)	(None, 4, 4, 64)	9280	conv2d_9[0][0]
conv2d_11 (Conv2D)	(None, 2, 2, 128)	73856	conv2d_10[0][0]
flatten_2 (Flatten)	(None, 512)	0	conv2d_11[0][0]
dense_7 (Dense)	(None, 512)	262656	flatten_2[0][0]
dense_8 (Dense)	(None, 128)	65664	dense_7[0][0]
dense_9 (Dense)	(None, 128)	65664	dense_7[0][0]
lambda_2 (Lambda)	(None, 128)	0	dense_8[0][0] dense_9[0][0]

Total params: 478,512

(a) Training encoder.

Model: "conditional_encoder"

Layer (type)	Output Shape	Param #	Connected to
input_5 (InputLayer)	[(None, 32, 32, 1)]	0	
conv2d_12 (Conv2D)	(None, 16, 16, 8)	80	input_5[0][0]
conv2d_13 (Conv2D)	(None, 8, 8, 16)	1168	conv2d_12[0][0]
conv2d_14 (Conv2D)	(None, 4, 4, 64)	9280	conv2d_13[0][0]
conv2d_15 (Conv2D)	(None, 2, 2, 128)	73856	conv2d_14[0][0]
flatten_3 (Flatten)	(None, 512)	0	conv2d_15[0][0]
dense_10 (Dense)	(None, 512)	262656	flatten_3[0][0]
dense_11 (Dense)	(None, 128)	65664	dense_10[0][0]
dense_12 (Dense)	(None, 128)	65664	dense_10[0][0]
lambda_3 (Lambda)	(None, 128)	0	dense_11[0][0] dense_12[0][0]

Total params: 478,368

(b) Conditional encoder.

Model: "decoder"

Layer (type)	Output Shape	Param #	Connected to
input_6 (InputLayer)	[(None, 128)]	0	
dense_13 (Dense)	(None, 512)	66048	input_6[0][0]
reshape_1 (Reshape)	(None, 2, 2, 128)	0	dense_13[0][0]
conv2d_transpose_7 (Conv2DTrans	(None, 4, 4, 128)	147584	reshape_1[0][0]
conv2d_transpose_8 (Conv2DTrans	(None, 8, 8, 64)	73792	conv2d_transpose_7[0][0]
conv2d_transpose_9 (Conv2DTrans	(None, 16, 16, 16)	9232	conv2d_transpose_8[0][0]
conv2d_transpose_10 (Conv2DTran	(None, 32, 32, 8)	1160	conv2d_transpose_9[0][0]
conv2d_transpose_11 (Conv2DTran	(None, 32, 32, 2)	146	conv2d_transpose_10[0][0]
input_5 (InputLayer)	[(None, 32, 32, 1)]	0	
concatenate_1 (Concatenate)	(None, 32, 32, 3)	0	conv2d_transpose_11[0][0] input_5[0][0]
conv2d_transpose_12 (Conv2DTran	(None, 32, 32, 8)	224	concatenate_1[0][0]
conv2d_transpose_13 (Conv2DTran	(None, 32, 32, 3)	219	conv2d_transpose_12[0][0]

Total params: 298,405

(c) Decoder.

Figure 4.10: *Final layer structure for the CVAE's encoders and decoder.*

As it was for the CGAN, dropout and data augmentation were removed as they was no longer necessary. The CVAE already had a relatively good ability to generalise through compressing inputs to latent representations that isolate the most important features. Including these extra measures would have further slowed down training, and would have made it harder to analyse how other aspects of the CVAE design impacted performance.

4.3.6 Hyper-Parameter Tuning

Due to the lesser amount of time required to train the CVAE compared to the CGAN, its parameters could be feasibly tuned through repeated testing.

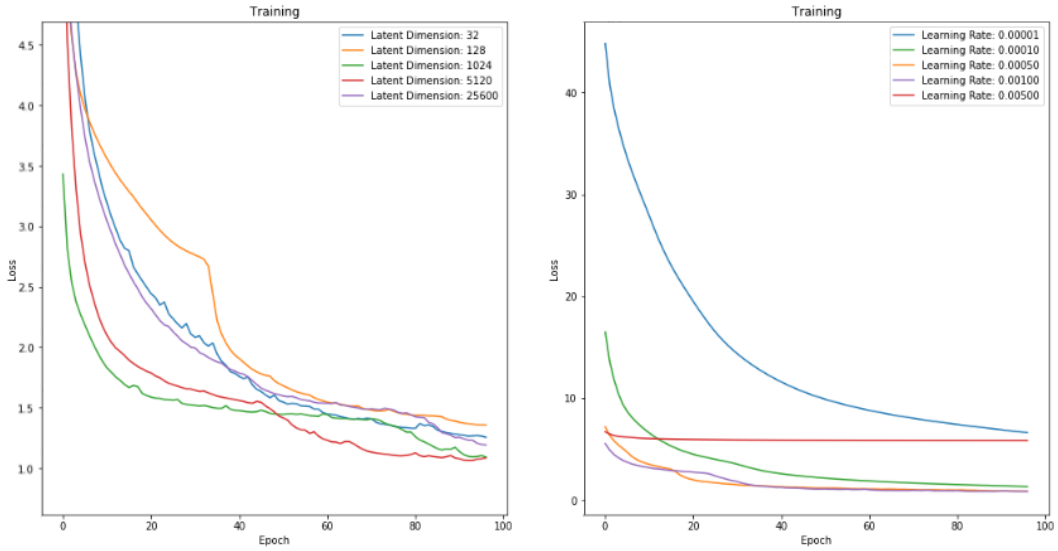
The first parameter to be tested was the batch size, which for the CVAE is also dependent on Colab's available RAM, training time, and the value of the final training loss.

Table 4.2: *Impact of different batch sizes on loss and training time.*

Batch Size	Loss after 100 Epochs	Training Time
1024	15.66	55 sec
512	5.78	1 min 6 sec
128	0.62	1 min 22 sec
32	0.21	3 min 8 sec
16	0.04	5 min 42 sec

As was the case for the CNN, lower batch sizes lead to a better final loss value but result in diminishing loss improvements at the cost of longer training times. The final batch size was 32, which seemed to offer a good balance between training time and performance.

The next parameter to be inspected was the latent dimension, which determines the size of the mean and standard deviation vectors of the latent feature space. VAEs with larger latent spaces are capable of storing more features. However having too large a latent dimension can cause overfitting as the optimiser model will be given too much space to store features in, leading to it memorising specific details of the training set. A latent dimension that is too small on the other hand will lead to the model underfitting because it lacks the storage necessary to memorise important general features. A variety of latent dimensions were tested and compared in order to find an optimal size (see figure 4.11a).



(a) *Effect of different latent dimensions on training the CVAE. A latent dimension in the range of 1024 to 5120 achieved optimal performance.*

(b) *Effect of different learning rates on training the CVAE. A learning rate of 0.001 appears to be optimal, as larger values after that (0.005) seem to decrease performance.*

Figure 4.11

The tests showed that a latent dimension in the range of 1024 to 5120 achieved optimal performance, so 1024 was chosen as the final latent dimension. As this was the smallest of the two, it was more likely to only encode more important general features than a larger dimension would.

Adam was chosen as the optimiser, as it was clear at this point that it was the best choice for image colourisation models, considering how well it had worked for both the CNN and CGAN. A range of learning rates were tested on the CVAE (see figure 4.11b) and it was found that a rate of 0.001 achieved the best continuous descent for the training loss and was thus used to train the final model.

5 Evaluation

In this chapter we will show testing results for each model and analyse the performance of each.

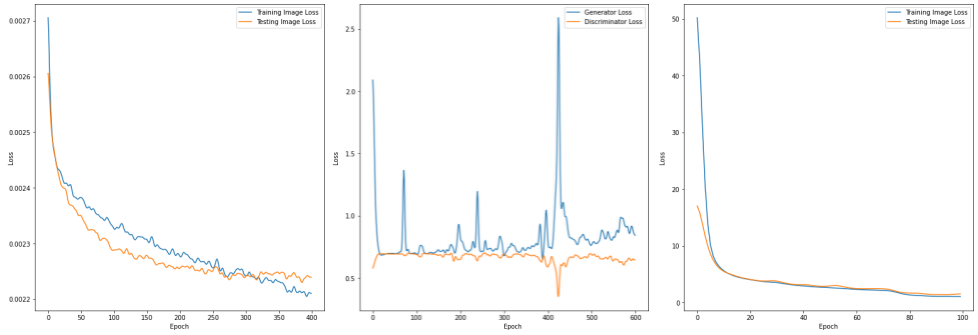
5.1 Computation Time

To gain a better understanding of how computationally expensive each model was, we evaluated the time it took for each one to train on a subset of Cifar-10 (see table 5.1).

Table 5.1: *Time required for each model to train on the Horse class of Cifar10.*

Model	Training Time
CNN	9 min 41 sec
CGAN	29 min 54 sec
CVAE	2 min 45 sec

The CVAE proved to be the fastest model to train, followed by the CNN, both of which finished training in under ten minutes. The CGAN however, took considerably longer than both at almost half an hour. This is primarily due to the precarious way in which it needs to be trained compared to the other two models.



(a) *CNN.* ***(b)*** *CGAN.* ***(c)*** *CVAE.*

Figure 5.1: *Learning curves of each model when trained on the Horse class of Cifar-10.*

The CNN was trained until it was clear that the model began to overfit. This point was evident when the training and testing loss of each model began to diverge (see figure 5.1a). The CNN was particularly susceptible to overfitting, and most of its design choices were made in an attempt to prevent it overfitting for as long as possible during training.

The CGAN did not have the same clear indicators for when training should end. Typically, the generator and discriminator's losses would continue to fluctuate around each other as long as training continued. In theory so long as these two losses were close to each other, some form

of learning was still happening. At some point, either the generator or discriminator would begin to outperform the other, and their respective losses would start to consistently diverge (see figure 5.1b). The CGAN was trained until this divergence became clear, which due to the inherent instability of the two loss functions, often required a large number of epochs to ensure the divergence was consistent, and not the product of a momentary spike.

Training for the CVAE was always performed up to the point where both the training and validation losses plateaued (see figure 5.1c). The fact that both the training and validation losses always stayed so close through training is a testament to how well the CVAE can fit the data.

5.2 Feature-Based Colourisation

To succinctly demonstrate the primary mechanism which Deep Learning models use to assign colours to areas of an image (explained in 3.2), the CNN was trained on images from the custom Shape Set (see figure 5.2).

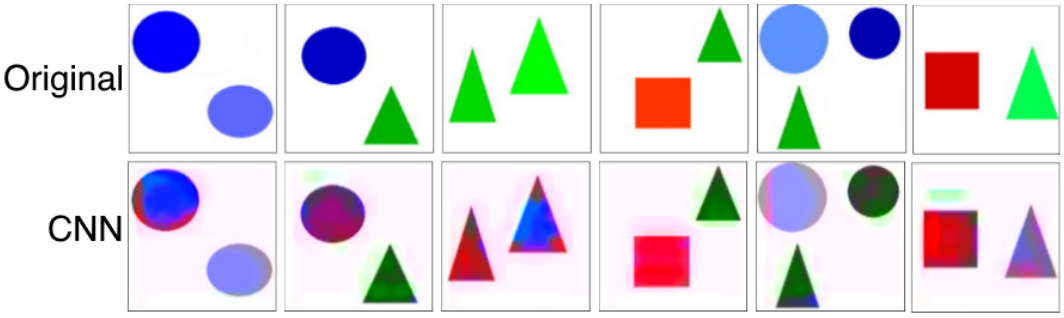


Figure 5.2: *Trained CNN results on unseen Shape Set images.*

The results show that the CNN has learnt to recognise many of the circles, squares, and triangles that were presented to it in a variety of different positions and dimensions. It has also understood that certain shapes correspond to particular colours, colouring many of the circles blue, the triangles green, and the squares red. It has learned that colours should generally be placed within the confines of shapes, however it has seemingly had difficulty using only one constant shade or colour for some shapes. It does show an ability to distinguish between shades, and even when it chooses the wrong colour for a shape it often is still able to give that colour a similar brightness to the original colour's.

A lot of the CNN's difficulty producing results similar to the original images is likely due to the variation in dimensions of the shapes themselves, which probably leads to the most confusion. This is because it has already shown a good ability to distinguish the shapes regardless of position through translational equivariance, so it's likely to be the dimension of the shapes which still causes issues for it. The relatively small size of the dataset also definitely contributes to some of the poor results, if the model had more images to train off it would have undoubtedly been able to generalise better.

Despite its shortcomings, the CNN's results have shown how it uses the mechanism described in 3.2 to perform colourisation, which was the main purpose of the experiment.

5.3 Colourisation Results

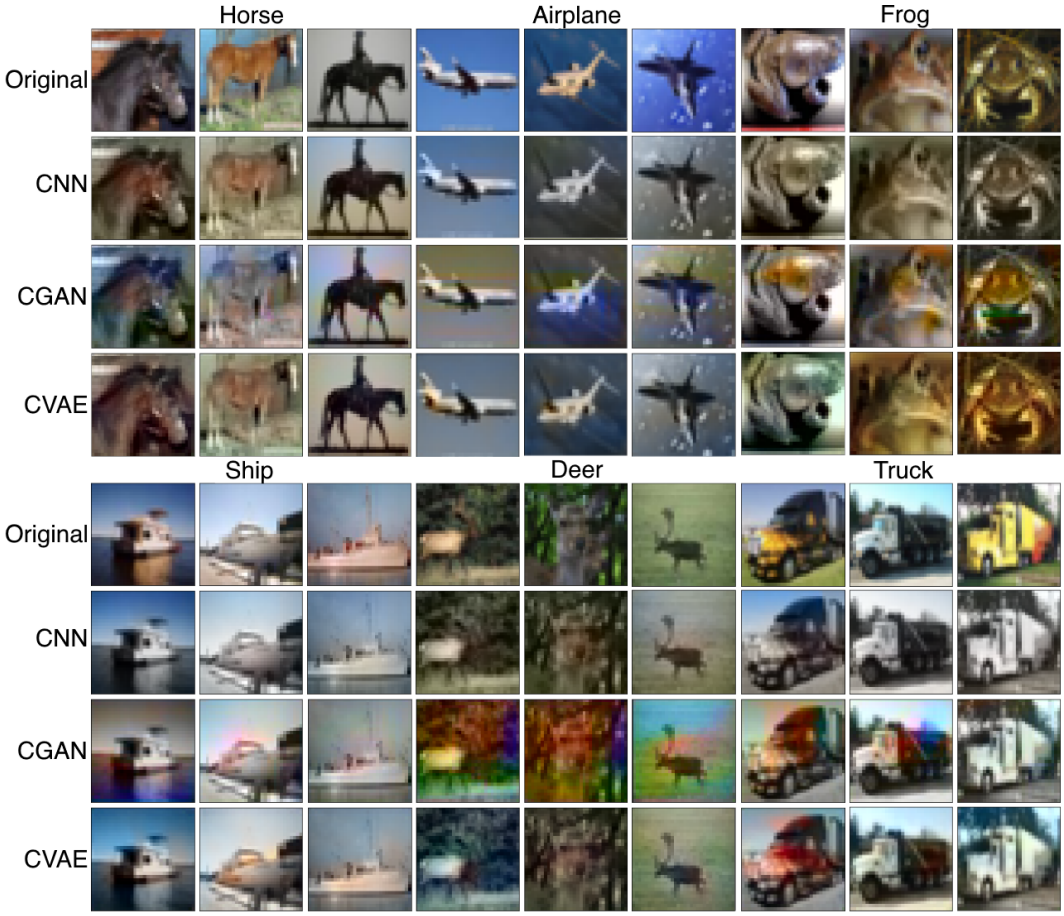


Figure 5.3: *Results on unseen Cifar-10 images from models trained for each specific class.*

Figure 5.4: *Trained model results on unseen Face images.*

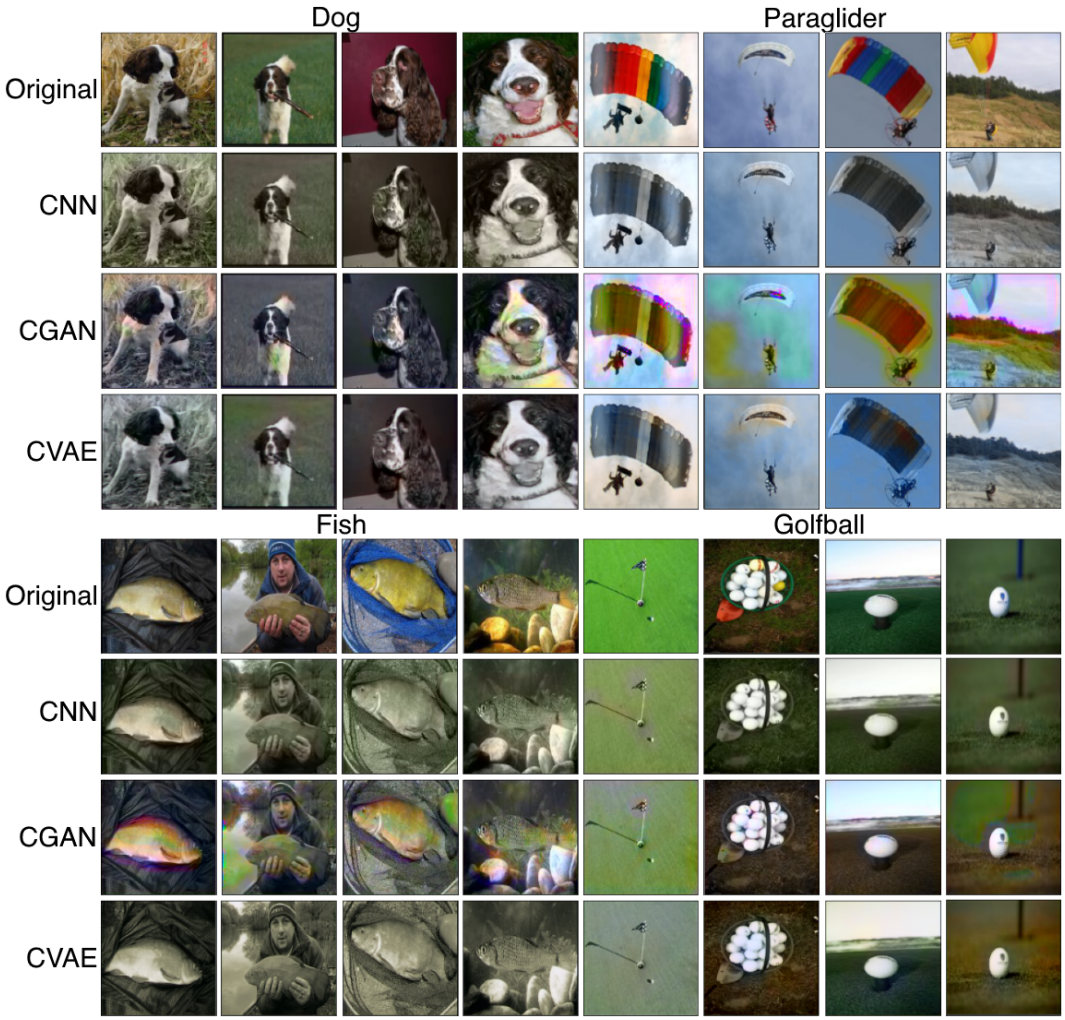


Figure 5.5: *Results on unseen Imagenette images from models trained for each specific class.*

It is important to note that every model was trained on each specific subclass of the Cifar10 and Imagenette datasets. This is because training on the entire dataset lead the models to produce results that were often too general and which made it difficult to determine the relative performance of each model. Training on the subclasses, on the other hand, allowed the models to produce more useful detailed results that could help us identify in what particular areas the network may be failing.

Certain classes also had interesting feature distributions that were worthy of closer analysis. For example Cifar10's Truck class was highly varied and challenging, containing images with many different colours and characteristics. Compare this to the Deer class, which consistently contains images of deers in front of green vegetation, giving the network a chance to identify a very clear and prevalent feature.

Also, the CVAE results shown are from a model with an unweighted reconstruction term, which causes most of its latent samples for a particular image to be very similar. The results shown here are thus almost the same as any other result it could have produced for each image. This was done so as to not give the CVAE an unfair advantage compared to the other models by exploiting its ability to provide diverse colourisations, which could then be cherry-picked in order to retrieve good outputs.

5.4 Discussion

5.4.1 CNN

On Cifar-10, the CNN performed adequately on classes where clear and repetitive priors could be identified, for example always being able to colour the sky blue for the Ship class, or colour the background green for the Deer class. Where such clear priors were not present however it tended to resort to using very muted colours, for instance assigning neutral grey colours to the main components in the Frog and Truck classes. This shows how the CNN is not very good at accounting for small details within images and tends to colour features only after it has been provided with a large amount of clear priors encouraging it to do so.

The CNN succeeded at producing plausible colourisations for the higher resolution Faces dataset, demonstrating that it is in fact capable of producing fairly detailed colourisations. Albeit, with the assistance of very constrained training data.

For the more challenging Imagenette dataset, the CNN had some of the same issues it had with Cifar-10. It was again only capable of producing soft colour results, and could only apply colour to areas having clear repetitive priors, such as the sky for the Paraglider class or the grass for the Golfball class.

The results demonstrate that the CNN is reliant on a great deal of clear prior information within its training data to allow it to provide strong colourisations to particular features, and that it tends to skip colouring details which it has not been provided enough training data relevant to. This makes it a very inconvenient solution to the problem, as to perform detailed colourisations it requires a large enough dataset tuned specifically to accomodate for the particular features it needs to recognise. However, one benefit to the simplicity of the CNN's results is that it is more likely to not colourise a feature that it is unsure about, as opposed to colouring it incorrectly. Strong incorrect colourisations can result in very jarring and more obviously incorrect results, as was a major issue with the CGAN.

5.4.2 CGAN

For Cifar-10, the CGAN tended to use very saturated bright colours in its colourisations, which is contrary to the CNN's tendency to produce soft colours for difficult to determine features. Many of these colours are inappropriately placed however, which can be seen in its coloursations of the Plane and Deer class. This shows that the CGAN struggled a lot with edge detection compared to the CNN.

On the Faces dataset it was able to better place colours on the image thanks to the more constrained nature of the dataset. However it had a similar issue with with its use of over-saturated colours, assigning an unnatural purple tone to many of the faces.

For the Imagenette dataset however, the CGAN produced some surprisingly good results. In cases like the Fish and Paraglider classes, it still struggled with edge detection, but it was capable of producing the most detailed colourisations where the other two models were unable to assign almost any colours.

The CGAN's issues with edge detection and colour intensity could be due to the fact that it is generalising too much and is accounting for many details which often are not present. This is likely due to poor training and overfitting, considering how difficult it is to determine how long it needs to be trained for and what the optimal hyper-parameters are for it. The fact that it was somewhat successful on a dataset as large and diverse as Imagenette however, could mean that it is best used when applied to problems with much more diverse priors in the training data, as opposed to the CNN's reliance on a large amount of specific priors. In either case, further

evaluation on other datasets is required to determine the true cause of these problems, and major improvements are still needed in order to make the CGAN a viable solution for colourisation.

5.4.3 CVAE

The CVAE appeared to perform the best of all three models when colouring Cifar-10. It struck a good balance between providing plausible colourisations, while still being fairly detailed. Certain classes of images like the Truck and Frog had results containing strong colours that were well placed within appropriate features, showing the CVAE had overcome the CGAN's issues with edge detection and the CNN's issues with inadequate colours. It also performed well on the Faces dataset, being able to produce slightly sharper results than the CNN did.

On the Imagenette dataset however it performed poorly, suffering from the same issue as the CNN in that it failed to recognise certain features and resorted to using muted colours. In some cases it even did worse than the CNN, as can be seen in the Golfball class. This indicates that the CVAE is only proficient at handling data of a complexity it is designed for and scales poorly to more challenging datsets with larger resolutions and more diverse features.

The CVAE's failure to colourise Imagenette images is not particularly bad however, considering it was designed and tested specifically on Cifar-10 data as were the other models. Its relative success with Cifar-10 is much more significant as it shows that it is the most capable model. It is clearly able to recognise the most features that can help it provide good results, while not accounting for too many that could lead to anomalies and poor generalisation.

A unique aspect of the CVAE also worth discussing is its ability to provide a variety of different colour outputs for a particular input. The CVAE was configured to increase the diversity of its outputs and trained on the Frog class of Cifar-10 in order to demonstrate this ability (see figure 5.6).

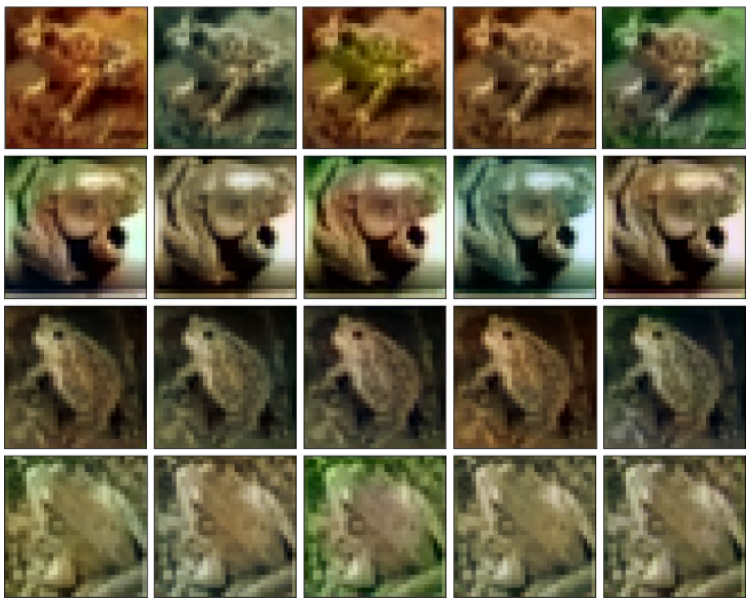


Figure 5.6: *Multiple outputs which a CVAE produced for unseen Cifar-10 images. The CVAE was trained with a low-weighted reconstruction loss to increase the diversity of each sample.*

This feature alone may make the CVAE the most ideal tool for colourisation. A greyscale image may have a range of potential colourisations that could be considered acceptable, and the CVAE is

capable of modelling this characteristic of the problem. Even if the CVAE cannot generate most of the colourisations that could be plausible, its ability to produce diverse outputs can still increase the likelihood of it providing a good colourisation. The CNN and CGAN on the other hand are only capable of producing the same output for any given input after having been trained, meaning for some images they can fail completely, while the CVAE might still be able to produce a satisfactory result.

6 Conclusion

Our project investigated the use of Deep Learning to solve the problem of automatic image colourisation. We gave an in depth analysis of the problem itself and why DL in particular was an attractive solution. We then looked at how best to present the training data to DL models in the form of a YUV encoding, and we chose a range of datasets that would best help us train and evaluate our models. We implemented a CNN, CGAN, and CVAE for colourisation, and evaluated each of them on our datasets in order to determine which one was the most viable solution.

Our CNN was designed from scratch and its architecture and hyper-parameters were optimised through repeated testing. We managed to produce a functioning colourisation model, though its results were often unimpressive. It was particularly susceptible to overfitting, and it would still overfit despite the many design choices that were made to try and prevent it from doing so. It also relied on a large amount of clear priors for specific features within its training data in order to get it to recognise and colour them. When it failed to recognise features, it resorted to using muted colours which often resulted in grey and weakly coloured outputs. It is a potential solution, however for it to be used in practice it would usually need to be trained on large and very specific datasets for it to produce plausible colourisations, which for many users won't be feasible.

The CGAN was implemented based mainly on suggestions from tutorials, as well as design decisions made in other colourisation CGAN projects. This was due to the extensive amounts of time needed to train the CGAN relative to other models, which was a problem considering the project's time constraints, as well as the limited resources Colab provides. While the CGAN is used in a number of other projects for colourisation and might well be a very capable of performing the task, the results from our model unfortunately were not very good. It had major issues with edge detection, feature recognition, and colour choice and requires further development in order to be a viable solution.

Our final model was the CVAE, which we were able to implement and tune through repeated testing better than the CGAN, thanks to its ability to train very quickly. It produced relatively detailed colourisations on our two simpler datasets, but began to perform poorly on more challenging data, even more so than the CNN. Its scalability issues are not particularly bad howevever, as its difficulties with the challenging dataset can probably be overcome by increasing its capacity, considering it was designed primarily to colourise images from the simpler datasets that it did so well on. It also possessed the very useful ability of being able to produce a diverse range of potential colourisations for a given input. This makes it a very good solution considering how most greyscale images can have more than one plausible colourisation which the CVAE is capable of accounting for. Its efficient training, tendency to fit the data well, ability to produce the most detailed results, and capacity for diverse colourisations, all make it the most promising model of the three. If time had permitted, it would undoubtedly have been developed further.

6.1 Future Developments

6.1.1 Better Training Environment

Google Colab was a very limiting work environment for this project, and was one of our main constraints. Its requirement that a browser session remained open throughout training made training models over a certain capacity as well as using large datasets almost impossible. An environment that provided the use of GPUs without these limitations, would allow us to do further work on resource and time intensive models like the CGAN, and furthermore allow us to implement even higher capacity models that are trained on bigger datasets.

6.1.2 Feature Extraction NNs

As has already been demonstrated (see 5.2), the primary mechanism which CNNs use to perform colourisation relies on feature extraction. To better guide this mechanism however, some state of the art projects like [21] also implemented dedicated feature extraction networks that fed feature data to the colourisation network, which lead to better results. Our own CNN could probably have performed much better if more of the feature extraction work it struggled with was left to dedicated networks. It would have also been interesting to see what different impacts low, mid, and high level feature extraction networks would have had on performance, as well as how they could be combined.

6.1.3 Improving Diverse CVAE Colourisation

The state of the art colourisation VAE [5] further capitalised on the VAE's ability for diverse colourisation by implementing the conditional encoder as a Mixture Density Network (MDN) that learns to map greyscale image data to multiple latent space embeddings. The MDN can take into account the one-to-many mapping between greyscale input and colour output, and allows the output latent representation to take multiple values conditioned on the same input. This provides a better selection of different latent embeddings to pass to the decoder. Their decoder's loss terms were also constructed in order to avoid blurry outputs and to take into account the uneven distribution of colours. These improvements would be worth trying on our own CVAE to improve its performance and its capacity for diverse outputs.

6.1.4 Image Restoration

One of the most common things image colourisation is used for is the restoration of old black and white photographs. The datasets that we used for our models mainly contained modern colour images that were then converted to greyscale for the purpose of training, but it would have been interesting to see how well our networks could handle colourising vintage photos as well. This also adds the extra challenge of not only colouring the image, but in many cases also restoring it. Old photos can often contain blemishes and damaged areas that Deep Learning models could feasibly also be used to fix through a process similar to colourisation.

A Appendices

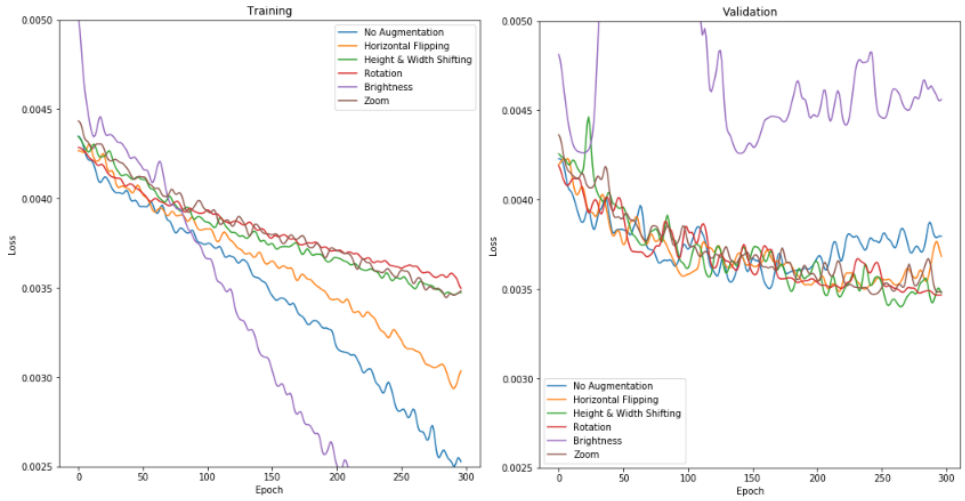


Figure A.1: *Impact of different forms of data augmentation on network Training and Validation loss. Horizontal Flipping, Shifting, Rotation, and Zoom all had positive impacts on helping the model to generalise. Zoom was the only form of augmentation that worsened training.*

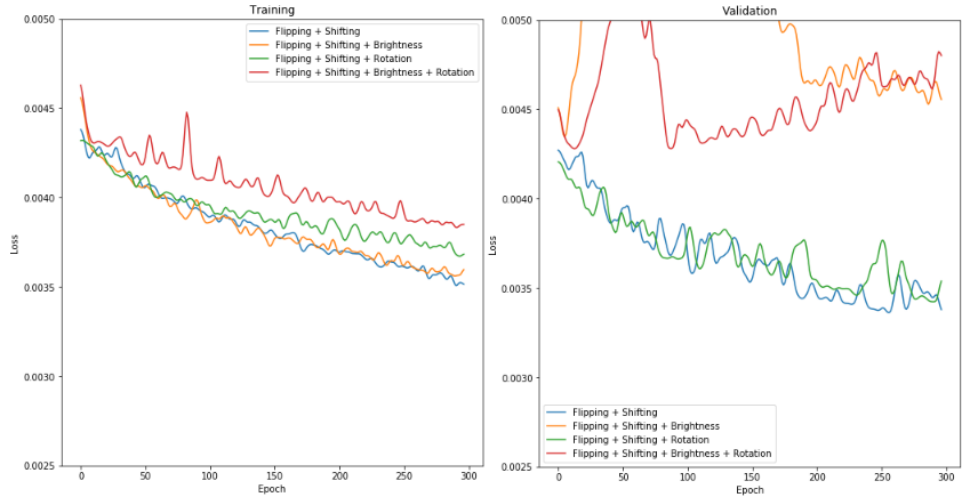
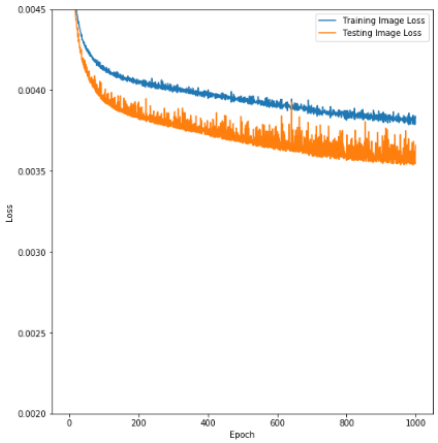


Figure A.2: *Impact of combinations of data augmentation methods on network Training and Validation loss. Flipping + Shifting + Rotation allowed the training and validation loss to match each other most closely and thus was chosen as the final set of augmentation techniques.*

Layer (type)	Output Shape	Param #
conv2d_62 (Conv2D)	(None, 31, 31, 8)	40
conv2d_63 (Conv2D)	(None, 16, 16, 16)	1168
conv2d_64 (Conv2D)	(None, 8, 8, 32)	4640
up_sampling2d_19 (UpSampling	(None, 16, 16, 32)	0
conv2d_65 (Conv2D)	(None, 16, 16, 16)	4624
up_sampling2d_20 (UpSampling	(None, 32, 32, 16)	0
conv2d_66 (Conv2D)	(None, 32, 32, 2)	290

Total params: 10,762

(a)



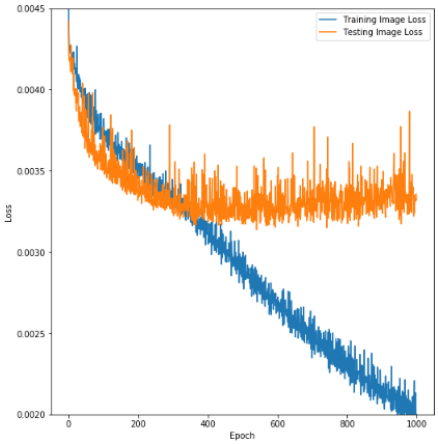
(b)

Figure A.3: *Low capacity shallow model designed for a 32x32 Cifar-10 input set with accompanying loss. The network is far too simple to be a feasible colourisation model however it does serve as a good start to analyse how poorly it performs in order to see how far the structure is from being a viable model. The model is incapable of storing enough information past a certain epoch and thus the loss for both training and validation data plateaus meaning a higher capacity model is needed.*

Layer (type)	Output Shape	Param #
conv2d_160 (Conv2D)	(None, 30, 30, 64)	640
conv2d_161 (Conv2D)	(None, 15, 15, 64)	36928
conv2d_162 (Conv2D)	(None, 15, 15, 128)	73856
conv2d_163 (Conv2D)	(None, 8, 8, 128)	147584
conv2d_164 (Conv2D)	(None, 8, 8, 256)	295168
conv2d_165 (Conv2D)	(None, 4, 4, 256)	590080
conv2d_166 (Conv2D)	(None, 4, 4, 128)	295040
up_sampling2d_50 (UpSampling	(None, 8, 8, 128)	0
conv2d_167 (Conv2D)	(None, 8, 8, 64)	73792
up_sampling2d_51 (UpSampling	(None, 16, 16, 64)	0
conv2d_168 (Conv2D)	(None, 16, 16, 32)	18464
conv2d_169 (Conv2D)	(None, 16, 16, 2)	578
up_sampling2d_52 (UpSampling	(None, 32, 32, 2)	0

Total params: 1,532,130

(a)



(b)

Figure A.4: *The contrasting deep model also designed for Cifar-10's 32x32 set with accompanying loss. The network is much deeper and downsamples the image to an even lower 4x4 resolution over more convolutional layers. The number of parameters here is excessive relative to the detail of the set and, while it will produce a detailed output for the training set, will inevitably overfit and be unable to generalise to unseen images. The loss for the training data continues to decrease however the loss for the validation data plateaus and gradually starts to go up again, meaning the model is overfitting and is memorising information from the traning data.*

6 Bibliography

[1] Activation Functions. https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html. Accessed: 05-Apr-2020.

[2] Tips for Training Stable Generative Adversarial Networks. https://machinelearningmastery.com/how-to-train-stable-generative-adversarial-networks/. Accessed: 08-Mar-2020.

[3] About ImageNet. http://image-net.org/about-overview. Accessed: 03-Apr-2020.

[4] Normal Gaussian Kulback Leibler Distance KL-distance KLdistance two normals Gaussians probability densities distributions KL2. http://allisons.org/ll/MML/KL/Normal/. Accessed: 07-Mar-2020.

[5] Aditya Deshpande, Jiajun Lu, Mao-Chuang Yeh, Min Jin Chong, David Forsyth. Learning Diverse Image Colorization. 2017.

[6] Alec Radford, Luke Metz, Soumith Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. 2015.

[7] Alex Yong-Sang Chia, Shaojie Zhuo, Raj Kumar Gupta, Yu-Wing Tai, David Cho, Ping Tan, Stephen Lin. Semantic Colorization with Internet Images. 2011.

[8] Anat Levin, Dani Lischinski, Yair Weiss. Colorization using Optimization. 2004.

[9] Ayoosh Kathuria. Intro to optimization in deep learning: Momentum, RMSProp and Adam. https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/. Accessed: 05-Apr-2020.

[10] Diederik P. Kingma, Jimmy Ba. Adam: A Method for Stochastic Optimization. 2014.

[11] Diederik P Kingma, Max Welling. Auto-Encoding Variational Bayes. 2013.

[12] Ian Goodfellow, Yoshua Bengio, Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[13] Ian J. Goodfellow, Jean Pouget-Abadie†, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio. Generative Adversarial Nets. 2014.

[14] Irina Higgins, Loic Matthey, Xavier Glorot, Arka Pal, Benigno Uria, Charles Blundell, Shakir Mohamed, Alexander Lerchner. Early Visual Concept Learning with Unsupervised Deep Learning. 2016.

[15] Jacques Hadamard. *Sur les problèmes aux dérivées partielles et leur signification physique*. Princeton University Bulletin, 1902.

[16] Kamyar Nazeri, Eric Ng, Mehran Ebrahimi. Image Colorization using Generative Adversarial Networks. 2018.

[17] Mario Bertero, Patrizia Boccacci. *Introduction to Inverse Problems in Imaging*. The Institute of Physics Publishing, 1998.

[18] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, Alexei A. Efros. Image-to-Image Translation with Conditional Adversarial Networks. 2017.

[19] Tomaso Fontanini, Eleonora Iotti, Andrea Prati. MetalGAN: a Cluster-based Adaptive Training for Few-Shot Adversarial Colorization. 2019.

[20] Tomihisa Welsh, Michael Ashikhmin, Klaus Mueller. Transferring Color to Greyscale Images. 2002.

[21] Zezhou Cheng, Qingxiong Yang, Bin Sheng. Deep Colorization. 2015.