Improved Learning of Joint Distributions using Soft-Coupled GANs

Master Thesis by

Markus Hald Juul-Nyholm
Patrick Abildgaard Alminde

Aalborg University
Technical faculty of IT and Design

IT and Design
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Improved Learning of Joint Distributions using Soft-Coupled GANs

Project Period:
Spring Semester 2020/21

Author(s):
Markus Hald Juul-Nyholm
Patrick Abildgaard Alminde

Supervisor(s):
Thomas Dyhre Nielsen

Page Numbers: 102

Date of Completion:
June 12, 2020

Abstract:

In this project we analyse the joint distribution learning framework Coupled GAN [34] and find that its imposed weight sharing constraint restricts the generators in learning the joint distribution over noisy and diverse datasets such as MNIST2SVHN, apple2orange, and horse2zebra. Through an experimental approach we propose to replace the strict weight sharing constraint with a softer coupling between generators in the shape of four regularisation terms. We call this type of model Soft-CoGAN (SCoGAN). These regularisation terms are (1) a feature regulariser which enforces generators to learn similar features, (2) a semantic loss based on classification of generated images such that the content of images are of the same class, (3) cycle consistency [44] between latent vectors and (4) a perceptual loss which is a more advanced version of the feature regulariser using features from a pretrained deep classifier. Through experiments on the aforementioned datasets we find that combinations of our proposed regularisers are able to provide a softer coupling that learns the joint distribution on MNIST2SVHN. However our approaches only achieve similar performance as CoGAN on the apple2orange, horse2zebra and CelebA datasets. We discuss the implications of this and provide arguments for our approaches showing greater promise on these datasets than CoGAN.

Summary

This is the mandatory summary of this master thesis.

For this master thesis we have worked on applying generative adversarial networks to joint distribution learning. The project is an extension of our 9th semester project in which we worked with both generative adversarial networks and variational autoencoders. This project was originally motivated mainly by our own interest in generative models and through literature review we found two important use cases, namely image-2-image translation and domain adaptation. Both of which rely on joint distribution learning. Examining this subject further we find that an array of SotA approaches [44, 16, 42, 30] express issues with an approach named Coupled GAN(CoGAN) presented in [34]. Specifically we see in [44] that CoGAN produces unappealing images on various image-to-image translation tasks, and [16, 30] report that the CoGAN model fails to learn the joint distribution over the digit datasets MNIST and Street View House Numbers(SVHN).

Further more we find that only little effort has been put into explaining the issues thoroughly or alleviating these to improve the CoGAN architecture. This may be a missed opportunity as CoGAN has the favourable property that it does not require paired data as needed in the Pix2Pix [19, 39] variants. Practically this means that we do not need a dataset representing the joint distribution we are trying to learn (dataset consisting of tuples of corresponding images in different domains), but only require images from the marginals of the domains. This allows us to learn the joint distribution between domains where paired data does not exist or is hard to come by. Furthermore CoGAN is interesting as most existing approaches [44, 7, 19, 16] utilise conditional GANs [35] while CoGAN instead couple GANs together such that each GAN learn a specific domain. We are therefore interested in whether the CoGAN proposal is a lost cause as indicated in SoTA literature, or if the core idea behind the proposal can be salvaged.

We therefore perform an experimental analysis of the CoGAN framework and discover two main issue of which we select one that we focus on. Namely that CoGANs fail to learn the joint distribution over dataset that contain a lot of noise in the shape of e.g. inconsistent placement of content across images, varying contexts on images and multiple objects on each image. We hypothesise that this is due to the way CoGAN learns the joint distribution, which is via a weight constraint on the coupled generators. In the case of a dataset as described above this constraint works more as a restriction on each generator such that it cannot learn it specific marginal.

We propose to replace the weight constraint with a softer coupling and propose 4 regularisation terms to enforce this coupling which are (1) a feature regulariser which enforces generators to learn similar features, (2) a semantic loss based on classification of generated images such that the content of images are of the same class, (3) cycle consistency [44] between latent vectors and (4) a perceptual loss which is a more advanced version of the feature regulariser using features from a pretrained deep classifier.

From experimentation with these terms we find that we are able to surpass or at least match the performance of CoGAN on various datasets.

Contents

Preface

We would like to thank the ITS department at Aalborg University for providing access to and support for the compute infrastructure used in this project (CLAAU-DIA) consisting of an ssh accessible server connected to two shared DGX-2 nodes. We would also like to thank the company Ambolt for providing a space in their offices to use as our group room.

Aalborg University, June 12, 2020

Markus Hald Juul-Nyholm
<Mjuuln15@student.aau.dk>

Patrick Abildgaard Alminde
<Palmin15@student.aau.dk>

Chapter 1 Introduction

Generative models are often used for data generation tasks such as image, music, speech, and text generation. In this project, we focus on image generation and for this task the Generative Adversarial Network (GAN) [12] has in recent years achieved state of the art performance by producing highly realistic images [25, 5]. However, in its original form, the GAN framework is only capable of learning marginal distributions i.e. learning images from one domain and not conditional or joint distributions. While extending GANs to learn either of these types of distributions is interesting we choose to focus on joint distribution learning of multi-domain images, as conditional GANs are already widely in use [44, 19, 35].

When dealing with multi-domain images we consider a joint distribution as a probability density function, that provides a density value for each occurrence of corresponding images. An example of a such joint distribution over two marginals consisting of digit images is shown in Figure 1.1. Here the marginals contain non-corresponding digit images in a specific style and the joint distribution consist of tuples of corresponding images in the respective styles. Using the same principle a joint distribution over facial images would consist of tuples of the same face with different attributes.

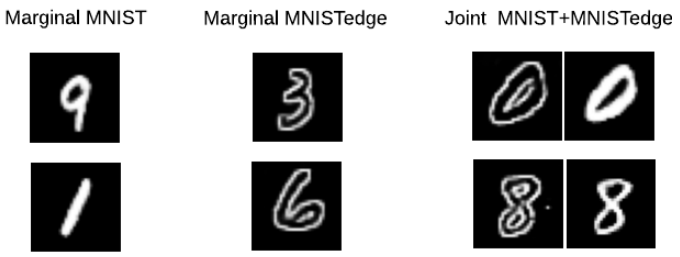


Figure 1.1: Example of data points from two marginal distributions and their joint distribution

Joint distribution learning has some important use cases such as domain adaptation and image-to-image translation. In domain adaptation, we wish to adapt a classifier trained in one domain to another domain while maintaining high classification accuracy. Domain adaptation is useful since labelled data for training classifiers is not always readily available and training a classifier on data from one domain and using it to classify data from another does not always yield good results. Learning a joint distribution over the two domains can aid domain adaptation models to understand the common features of the two domains.

In image-to-image translation, we wish to apply the style, e.g. the colours and tex-

tures, of a source image to the content, e.g. objects and shapes, of another image. Image-to-image translation networks are useful as ordinary GAN architectures, focusing on novel image generation, offer little to no control over the style or content of the generated images, besides their resemblance to the training data. Therefore if you wanted to generate a specific image you might have to generate multiple images from these models before randomly stumbling across the desired output. On the contrary, GANs focused on image translation tasks are concerned with the ability to alter an existing image through some learned transformation. Some examples of real-world cases could be satellite images to maps, which can be used for applications such as google maps, or sketches to detailed objects, which can be used by many kinds of design tasks to provide fast mock-ups of low fidelity ideas. Thus an image-to-image model can be viewed as learning the joint distribution over content(sketch) and style(detailed object) images.

While reviewing literature on state of the art methods that perform joint distribution learning we find that [44, 16, 42, 30] express issues with an approach named Coupled GAN(CoGAN) presented in [34]. Specifically we see in [44] that CoGAN produces unappealing images on various image-to-image translation tasks, and [16, 30] report that the CoGAN model fails to learn the joint distribution over the digit datasets MNIST and Street View House Numbers(SVHN).

Upon delving further into the related literature on CoGAN we find that little effort has been put into explaining the issues thoroughly or alleviating these to improve the CoGAN architecture. This may be a missed opportunity as CoGAN has the favourable property that it does not require paired data as needed in the Pix2Pix [19, 39] variants. Practically this means that we do not need a dataset representing the joint distribution we are trying to learn (dataset consisting of tuples of corresponding images in different domains), but only require images from the marginals of the domains. This allows us to learn the joint distribution between domains where paired data does not exist or is hard to come by. Furthermore CoGAN is interesting as most existing approaches [44, 7, 19, 16] utilise conditional GANs [35] while CoGAN instead couple GANs together such that each GAN learn a specific domain. We are therefore interested in whether the CoGAN proposal is a lost cause as indicated in SoTA literature, or if the core idea behind the proposal can be salvaged. We begin by examining this further through our initial problem statement:

How does CoGAN learn the underlying relationship of image domains to represent the joint distribution between them and which problems are experienced in this approach? Can we replicate the known issues with this architecture and understand the underlying causes in order to propose a solution to some of these issues?

Chapter 2 Preliminary theory

Before exploring the initial problem statement we present the necessary theory to sufficiently understand the remainder of this project. The two main areas that we cover here are: (1) Neural Networks in general and some important architectural components which we make use of, and (2) Generative Adversarial Networks (GAN) as originally proposed in [12] which are a key component in the CoGAN architecture. Additionally, we describe two main problems experienced with GANs and present existing approaches which we utilise later in this project to mitigate these problems as they arise in our own models. Finally this chapter also contains an overview of the datasets used throughout the project.

2.1 Neural Networks

In this section we provide a theoretical explanation of neural networks and their associated training procedure. Alongside this we describe different components that are used in neural networks, and that we utilise in this project. No single source has been used while writing this section. The information provided here has been accumulated through several semesters of working with neural networks and represents our theoretical understanding of the presented subjects. However in previous semester we have used [11] as a source on the presented subjects.

Neural networks are function approximators, meaning that they can be used for an array of problems in which some unknown function must be learned. Examples of such problems could be prediction and classification tasks. Structurally a neural network can be represented as a directed graph where the nodes(neurons) are arranged in layers as shown on Figure 2.1. Here the neural network consists of an input layer(blue), a single hidden layer(green), and an output layer(orange). As shown on the figure each layer following the input layer has an associated bias neuron with edges to all neurons in its associated layer. All edges are weighted and updating these weights is what we consider as learning in a neural net. This is done through a process referred to as training which utilises the backpropagation algorithm. This process is further explained in subsection 2.1.2.

Due to this layered structure neural nets can be viewed as a chain of transformation functions in which each layer constitutes a transformation of its input. for a neural net f with the structure shown on Figure 2.1 the chain of transformations is written as follows: $f(\mathbf{x}) = f^{(2)}(f^{(1)}(\mathbf{x}))$. Here $\mathbf{x}$ is the input vector containing 4 values and each f^i represents a layer. To demonstrate how data is passed through a neural

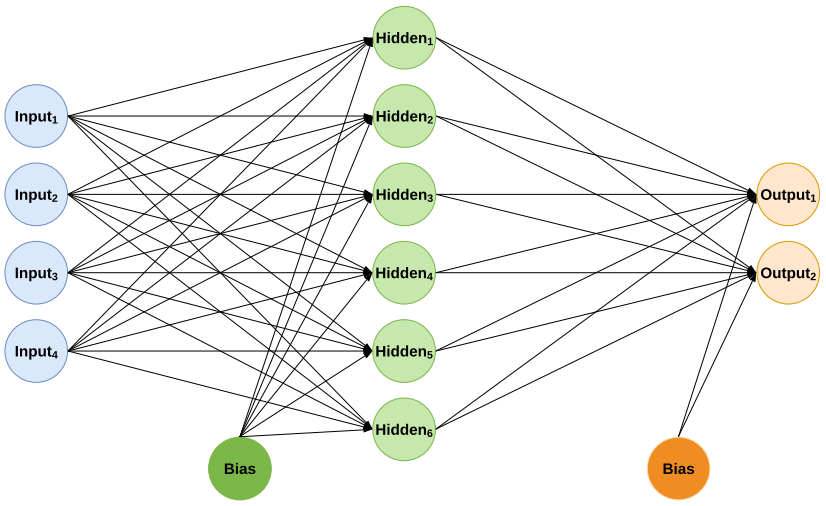


Figure 2.1: Simplified neural network model

net we examine an arbitrary neuron. The input, z_i, to neuron i is calculated as: $z_i = (1 \cdot bw_i) + \sum_{j=1}^{m} Input_j w_j$, where m is the total amount of inputs to neuron i, w_j is the weight on the edge connecting $Input_j$ to the neuron and bw_i is the weight on the edge connecting the bias node and the neuron i. This intermediate value z_i is then activated, $a_i = \sigma(z_i)$, before it is passed on to the next layer. Here σ is called an activation function.

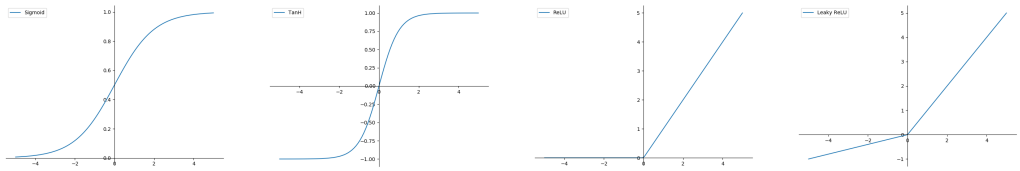
Activation functions are used in neurons to introduce non-linearity into the neural network. This is important since otherwise a neural network would simply be a sequence of linear transformations on the input data, which in itself is just a linear transformation. As such it would not be able to model complex non-linear functions. Different variants of activation functions that we use in this project are presented in the following subsection.

2.1.1 Activation functions

Sigmoid
The Sigmoid activation function is defined as $f(x) = \frac{1}{1+e^{-x}}$ it scales input values to be in the range [0,1] and is therefore often used for binary classification problems. A main issue with the Sigmoid activation function is that it suffers from the well known vanishing gradients problem. This problem occurs if input values are either very large or very small leading to outputs very close to 0 (small input values) or 1 (large input values), which in turn causes the gradients to approach 0, providing

little feedback and halting the training process. This can be seen on Figure 2.2a as the function stagnates.

Hyperbolic Tangent
Another widely used activation function is the hyperbolic tangent (TanH) which is a scaled version of the Sigmoid activation function and is defined as $f(x) = \frac{2}{1+e^{-2x}} - 1$. The TanH function scales input to be in the range [-1, 1] and as shown on Figure 2.2b it provides steeper derivatives than Sigmoid because of its scale. An important distinction is that TanH is a zero-centered function, while Sigmoid is centered at 0.5, which means that it can be better at handling both negative and positive inputs. Like Sigmoid, this function also suffers from vanishing gradients.



a) Sigmoid b) Tanh c) ReLU d) Leaky ReLU

Figure 2.2: Activation function plots

Rectified Linear Unit (ReLU)
The ReLU activation function is defined as $f(x) = max(0, x)$. The function returns x if $x > 0$ otherwise it returns 0. A negative or 0 input results in a derivative with value of 0, which provides no feedback to the backpropagation process and therefore neurons that experience this become passive as their learning is stopped. This can be seen on Figure 2.2c as the function is constant for inputs below or equal to 0. This is an issue when many of the neurons are shut off leading to the "dying ReLU problem" where large parts of the neural networks becomes passive, leading to a decrease in the complexity of the functions that can be approximated by the neural net.

Leaky ReLU
Leaky ReLU [33] is defined as $f(x) = max(\alpha \cdot x, x)$ and is a response to the aforementioned "dying ReLU problem". The issue is solved by introducing a small positive slope, determined by α, on negative values. This ensures that negative values does not result in a 0-derivative which happens for the ReLU activation function. The functions plot is shown on Figure 2.2d. Another version of Leaky ReLU is parametric ReLU (PReLU) [14] where the α value is learned during training.

2.1.2 Backpropagation

As previously mentioned neural network training uses the concept of backpropagation. This is a gradient based algorithm that utilises the fact that the negative gradient of a function points towards where the function decreases most rapidly. The weights and biases are therefore updated based on the negative gradient of a loss function in an attempt to minimise the loss. This approach is also called gradient descent. In order to understand how this works we describe one backward pass of a simple neural network, shown in Figure 2.3 and afterwards we generalise the procedure. The simple network consists of four layers: 1 input(blue), 2 hidden(green) and 1 output(orange), each layer has a single bias neuron(purple) and one weight between each of these neurons. We assume that the values of the weights and biases comes from some initialisation procedure and that one forward pass has been performed on some input resulting in the output value displayed on the figure. The disconnected neuron(pink) is the target label of the input data.

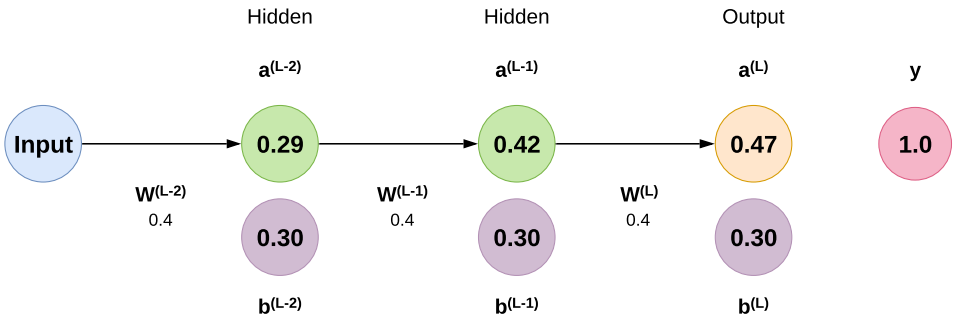


Figure 2.3: Single neuron neural network for backpropgation example

In this example we use the squared error loss function. The loss is therefore given by: $C_0(\ldots) = (a^{(L)} - y)^2 = (0.47 - 1)^2 = 0.28$ where $a^{(L)}$ is the value after activation of the layer L and y is the input label. $a^{(L)}$ is calculated through $a^{(L)} = \sigma(z^{(L)})$ as previously shown. For this example we use ReLU as the activation function in every layer.

Now we want to know how changes to the weight $w^{(L)}$ affects the outcome of our loss function $C_0(\ldots)$. We therefore want the ratio between changes of these two values given by $\frac{\partial C_0}{\partial w^{(L)}}$, however C_0 and $w^{(L)}$ are not directly correlated, as can be seen on Figure 2.4 so we use the chain rule to formulate the following $\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}}$. We then calculate these ratios as the derivatives of the original functions. Therefore $\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)} = 0.42$ and $\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)}) = ReLU'(0.47) = 1$

and $\frac{\partial C_0}{\partial a^{(L)}} = 2(a^{(L)} - y) = 2(0.47 - 1) = -1.06$, which when substituted in gives us $\frac{\partial C_0}{\partial w^{(L)}} = a^{(L-1)}\sigma'(z^{(L)})2(a^{(L)} - y) = -1.06 \cdot 0.42 \cdot 1 = -0.45$. This is the sensitivity of the loss function with respect to changes in $w^{(L)}$. We then carry out the same procedure to calculate the term with respect to $b^{(L)}$, which results in $\frac{\partial C_0}{\partial b^{(L)}} = 1\sigma'(z^{(L)})2(a^{(L)} - y)$.

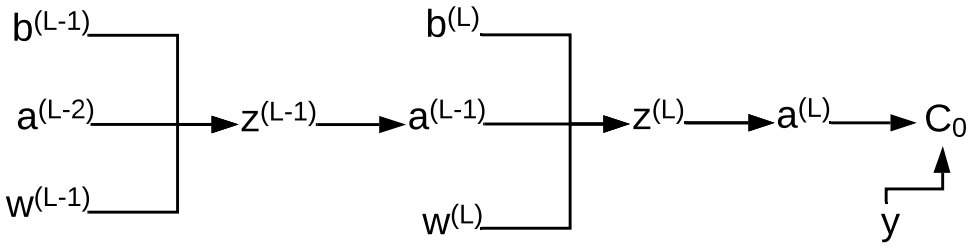


Figure 2.4: Illustration for explaining chain rule

This can then be applied to calculate the sensitivity of the loss function in regards to all weights and biases, with the small change that each step further from the loss requires an extra step in the chain-rule. For example: If we were to calculate $\frac{\partial C_0}{\partial w^{(L-1)}}$ we need to calculate the following chain $\frac{\partial C_0}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial z^{(L-1)}}{\partial w^{(L-1)}}$ where the first term must be extended as another chain to form: $\frac{\partial C_0}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial C_0}{\partial a_L} \frac{\partial a^{(L)}}{\partial z_L}$. These chains can also be followed on Figure 2.4 once again.

Finally these calculations are used to produce the gradient vector containing the gradient for each weight and bias in the network $\nabla C_0 = \left| \frac{\partial C_0}{\partial w^1} \quad \frac{\partial C_0}{\partial b^1} \quad \dots \quad \frac{\partial C_0}{\partial w^{(L)}} \quad \frac{\partial C_0}{\partial b^{(L)}} \right|$. Once the gradient vector is obtained the weights and biases of all layers are updated corresponding to their specific gradient values. An example of updating the weights of layer L is $\hat{w}^{(L)} = w^{(L)} - \alpha \cdot \frac{\partial C_0}{\partial w^{(L)}}$ where alpha is the learning rate, which regulates the size of the update.

We now generalise these calculations to a more realistic neural network with multiple neurons in each layer. First the loss is calculated as a sum of losses for each output neuron of the network, in our case the sum of squared error loss: $C = \sum_{j=1}^{J} (a_j^{(L)} - y)^2$ where J is the total amount of neurons in layer L(output layer).

Secondly, the input to a single neuron, $z_j^{(L)}$, is now dependant on multiple inputs from the previous layer $L - 1$. Therefore to calculate any $z_j^{(L)}$ the sum of its ingoing weights times the outbound activations of the previous layers are used like

so: $z_j^{(L)} = b_j^{(L)} \sum_{k=1}^{K} w_{j,k}^{(L)} a_{j,k}^{(L-1)}$. Here K is the amount of neurons in the previous layer.

Thirdly, as any given neuron is now affected by multiple neurons in the previous layer this changes how the gradients are calculated. Here we can look at the dual chain example from before, where we are interested in calculating $\frac{\partial C_0}{\partial w^{(L-1)}} = \frac{\partial C_0}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial z^{(L-1)}}{\partial w^{(L-1)}}$. Recall the calculation of the first term $\frac{\partial C_0}{\partial a^{(L-1)}}$ which was solved through the chain-rule. Now we must take the sum of the same chain rule, for each of the connected neurons in the next layer, so we get that: $\frac{\partial C}{\partial a_k^{(L-1)}} = \sum_{j=1}^{J} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial C}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}}$.

Usually these network updates are performed on batches of data using mini-batched stochastic gradient descent(SGD). In this project we use an extended version of mini-batch SGD called Adaptive Moment Estimation (ADAM) [27]. This method uses an adaptive learning rate meaning that given a vector w_t containing the weights of a layer at timestep t each weight has its own learning rate. This is achieved by keeping track of a decaying average of past gradients, $m_t' = \beta_1 m_t + (1 - \beta_1) g_t$, and a decaying average of past squared gradients, $v_t' = \beta_2 v_t + (1 - \beta_2) g_t^2$. Where the beta values($\beta_1$ and β_2) are originally proposed to be 0.9 and 0.999 respectively [27] and g_t represents the gradients of w_t with regards to the loss. These averages are initialised with zeroes and therefore tend to be biased towards 0 in the beginning of training if the beta values are close to 1 [27]. It is therefore necessary to compute bias corrected versions of these averages: $\hat{m}_t = \frac{m_t'}{1-\beta_1^t}$ and $\hat{v}_t = \frac{v_t'}{1-\beta_2^t}$. The update rule for the ADAM optimiser is then $w_t' = w_t - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$ in which the learning rate α is adapted according to the two averages and ϵ is a very small number used to avoid division by 0.

2.1.3 Weight initialisation

As mentioned neural nets require weights to be initialised before training. When initialising it is important to control the magnitudes of weights such that they are not disproportionate to each other. This imbalance can have a large impact on the convergence time of the neural network. There exists several approaches to initialise neural network weights however we mention only two, as these are the ones used in this project. The first is simply drawing all weights from a normal distribution with mean 0 and a constant standard deviation. This approach can suffer from disproportional weight values when the standard deviation is high. The second is Xavier initialisation [9], which also draws from a normal distribution with mean 0, but for each layer it uses a different standard deviation. We use the

Keras implementation[1] of this approach in which the standard deviation for each layer is computed as: $stddev = \sqrt{\frac{2}{layer_{in}+layer_{out}}}$. Here $layer_{in}$ is the amount of ingoing edges and $layer_{out}$ is the amount of out-going edges.

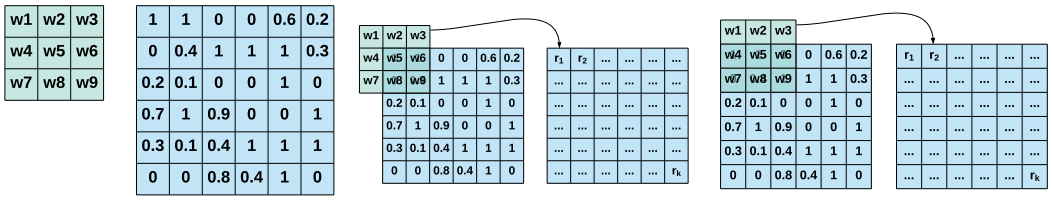
2.1.4 Convolution and Transposed convolution

So far we have only considered fully connected layers i.e. layers where all neurons of a layer are connected to all neurons of the subsequent layer, however an important layer type used in neural networks when working with image data is the convolution and transposed convolution layers. These layers capture underlying feature representations, pixel dependencies and reduce complexity of neural networks to increase training speed [28].

Convolution
A convolution operation is primarily used to scale down the size of input images while retaining the features of the image. As such a convolution layer may reduce a 64x64x3 RGB image to 32x32x6 with 3 additional feature representations. To achieve this the convolution layer uses kernels that contain learnable weights. A kernel is an $n \times n$ matrix that is passed over an $m \times m$ input matrix and aggregates multiple pixel values into a single value according to the weights. The step size of the kernel when passing over the input image is dictated by a stride value. Furthermore zero-padding of the input matrix is often used to control the size of the output. A convolution layer has a kernel for each feature representation that the layers has to produce. Therefore a single convolution layer is able to capture many different feature representations since all the kernels have separate learnable weights.

As an example we see on Figure 2.5a a green 3x3 matrix(kernel) and a blue 6x6 matrix(input). With $stride = 1$ and $padding = 1$ the first step is to place the mask as seen on Figure 2.5b and compute the first pixel of the output as the sum of all overlapping weights(from kernel) and indices(from input data): $output_1 = w5 \cdot 1 + w6 \cdot 1 + w7 \cdot 0 + w8 \cdot 0.4 + w9 \cdot 1$. Then we stride the kernel 1-step to the right and make the same computation again and so on until we have covered the entire input matrix. This produces a new matrix as seen on Figure 2.5c that is passed onto the next layer of the network. This specific example preserves the height and width dimensions of the input matrix, however a different combination of stride and padding can reduce these dimension if needed.

[1] `https://keras.io/api/layers/initializers/`



(a) 3x3 kernel matrix(Green) and a 6x6 input matrix(Blue)

(b) First convolution of kernel onto input matrix

(c) Striding the kernel 1 step to left for the next operation

Figure 2.5: Convolution process with a 3x3 kernel, stride = 1 and padding = 1

Transposed convolution

Instead of using a kernel to compress multiple values to 1-value, the transposed convolution decompresses 1-value to multiple values instead, i.e. it upscales images. It utilises a kernel, stride and padding just like the ordinary convolution, but the calculations are different. If we look at Figure 2.6 we have an input- and a kernel matrix, both 3×3, we set $stride = 1$ and $padding = 0$. The kernel is multiplied with the first entry of the input, which produces the lightblue matrix on the right. The same is done for each entry of the input matrix, and all of the resulting matrices are summed to produce the output. Its clear that this operation is able to function with multiple kernels as well and that transposed convolutions are capable of producing an output with a higher resolution than its input.

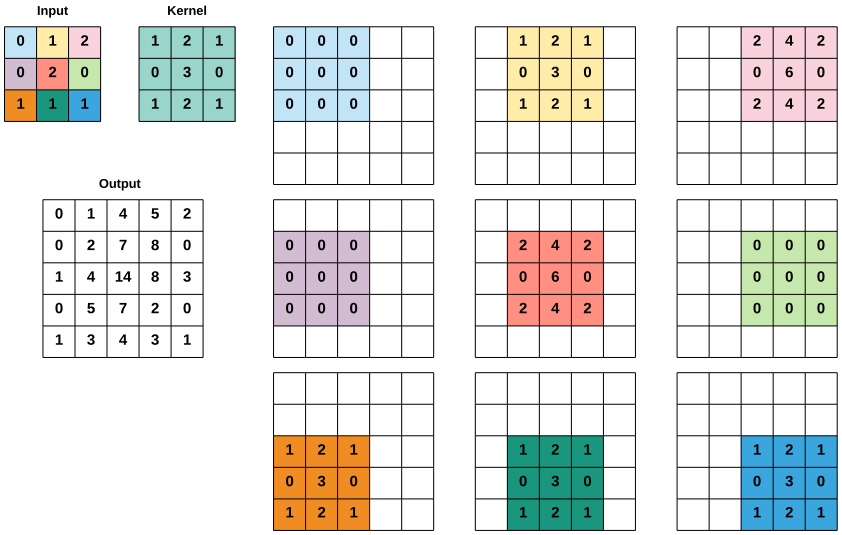


Figure 2.6: Example of transposed convolution operation on 3×3 input with 3×3 kernel. $padding = 0$ and $stride = 1$

2.1.5 Pooling operations

Another useful operation when working with image data is pooling operations. Pooling operations are used for down scaling feature representations within a network. Instead of a kernel we define a matrix window of size $n \times n$ which does not contain any weights. The down scaling is performed by passing the window over an $m \times m$ input matrix. Here all pixel values contained in the window will be aggregated into a single pixel value in the down scaled matrix. Different aggregation methods exists, and some commonly used are minimum, maximum or average pooling. The window is then passed over the entire input matrix and for each pass a new value in the down scaled matrix is created. Just like the convolutions the window can be controlled through padding and stride. If we have a 4×4 input matrix and a 2×2 window which we pass over the input matrix with a stride of 2. The down scaled matrix will be of size 2×2.

2.1.6 Batch Normalisation

When using neural nets it is common practice to normalise the input data into small ranges such as [0,1], or [-1,1], or by ensuring that the data has mean = 0 and standard deviation = 1 (also called standardisation). This is done to control the magnitudes of input values such that large values do not by default have larger influence on the output of the network.

Batch normalisation(BN) [18] is proposed based on the argument that normalisation should not only be performed on the input data, but also between layers in the net as well. Batch normalisation is often used between the fully-connected/convolution layer and the activation, such the output of the previous layer is normalised and then activated. Introducing batch normalisation in neural nets have been shown to result in a wider range of hyperparameter settings that lead to successful training as well as reducing training time [18].

It is called *batch* normalisation because it normalises the feature representations across each batch in mini-batched SGD. Given a batch $B = [x_1, ..., x_m]$ of data samples where each x_i consists of k feature representations $x_i = [\mathbf{x}_{i,1}, ..., \mathbf{x}_{i,k}]$ batch normalisation is performed as standardisation on the feature representation j, by computing the mean: $\mu_j = \frac{1}{m}\sum_{i=1}^{m} \mathbf{x}_{i,j}$ and variance: $\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}_{i,j} - \mu_j)^2$ over a batch of size m. These values are then used to normalise the j'th feature representation across the batch: $\bar{\mathbf{x}}_{i,j} = \frac{\mathbf{x}_{i,j} - \mu_j}{\sqrt{\sigma_j + \epsilon}}$ where ϵ is a small number used to avoid division by 0. When all feature representations have been normalised they have mean of 0 and standard deviation of 1, however we do not always want these spe-

cific values for mean and variance since this could limit learning. Therefore BN allows for scaling and shifting of the normalised feature representations through the following equation: $\hat{\mathbf{x}}_{i,j} = \gamma\bar{\mathbf{x}}_{i,j} + \beta$ where γ and β are learned parameters that enable the network to decide how best to scale and shift activations in the batch for the given batch normalisation layer.

2.1.7 Layer Normalisation

The goal of layer normalisation [4] is the same as batch normalisation, however it is achieved by normalising each element of the batch independently from each other rather than normalising each feature representation in relation to each corresponding feature representation in all elements of the batch. Layer normalisation therefore instead calculates mean and variance over feature representations in the individual batch elements rather than across the batch. It follows a similar procedure of calculating first the mean: $\mu_i = \frac{1}{m}\sum_{j=1}^{m}\mathbf{x}_{i,j}$ and then the variance: $\sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m}(\mathbf{x}_{i,j} - \mu_i)^2$. This is used to normalise every feature representation in the i'th element in the batch $\bar{\mathbf{x}}_i = \frac{\mathbf{x}_i - \mu}{\sqrt{\sigma_i + \epsilon_i}}$. Similar to batch normalisation this approach allows for scaling and shifting using two learned parameters specific to each layer of layer normalisation: $\hat{\mathbf{x}}_i = \gamma\bar{\mathbf{x}}_i + \beta$.

2.2 Generative Adversarial Networks

In this section we present a theoretical explanation of the Generative Adversarial Network (GAN) framework, the issues that it faces, and three popular approaches to dealing with these issues. This section contains revised parts of the GAN section in our project report from last semester [1].

A GAN is a generative model in which two models compete with each other. A generator which generates images that are supposed to resemble images from a chosen dataset and a discriminator which estimates the probability that a data sample was created by the generator or belongs to the dataset. The generator, G, must learn to represent some empirical distribution, p_{data}, over a predefined dataset of unlabelled images. We call the approximate distribution defined by the generator, p_{model}. To achieve this, a prior distribution, p_z, over a latent space, is defined. p_z can be represented by any continuous distribution e.g. normal or uniform distributions. The generator learns to represent p_{data} as a mapping between p_z and the data space, which is denoted $G_\phi(\mathbf{z})$, where G is a neural network with trainable parameters ϕ, which takes as input $\mathbf{z} \sim p_z$ and produces an image. A

second neural network is defined for the discriminator, D, which is represented by $D_\theta(\mathbf{x})$, with trainable parameters θ. This net takes, as input, a sample $\mathbf{x}$ from either, p_{model}(generated image) or p_{data}(real image), and $D_\theta(\mathbf{x})$ describes the probability that $\mathbf{x}$ origins from p_{data}, rather than p_{model}.

GAN training is in [12] presented as a minimax game between the two models with the loss: $\min_G \max_D V(D,G) = \mathbb{E}_{\mathbf{x}\sim p_{data}} log(D(\mathbf{x})) + \mathbb{E}_{\mathbf{z}\sim p_z} log(1 - D(G(\mathbf{z})))$. In this setting, the generator is trained to minimise the log probability of the discriminator classifying generated images as fake, and the discriminator is trained to best distinguish between generated and real images. An important observation regarding the GAN loss is that the generator loss does not correspond with sample quality. Usually in machine learning, a decreasing loss means that the model is getting better, however, GANs do not adhere to this standard. This can be seen as the full GAN loss measures only how well the models are performing in relation to each other and the generator loss itself measures only its current ability to fool the discriminator. Therefore, a decreasing generator loss does not imply good samples since a poorly performing discriminator may be fooled by low-quality samples.

When using the minimax loss as the objective for training G and D, the learning in every discriminator step, i.e. for a fixed generator, resembles an approximation of the Jensen-Shannon divergence (JSD) between p_{data} and p_{model} defined as: $JSD(p_{data}||p_{model}) = \frac{1}{2}KLD(p_{data}||p_m) + KLD(p_{model}||p_m)$ where $p_m = \frac{1}{2}(p_{data} + p_{model})$ and $KLD(p||q) = \sum_{x\in X} p(x) log \frac{p(x)}{q(x)}$ for two probability distributions q and p over the same space X [12]. The Jensen-Shannon divergence and the Kullback-Leibler divergence (KLD) are distance measures between two probability distributions. This means that the discriminator loss provides an approximation of the distance between the two distributions and the better the discriminator is the more accurate the approximation. In the case where the discriminator can perfectly discern real and generated samples, the loss is exactly $-log(4) + 2JSD(p_{data}||p_{model})$ and thereby reaches its minimum of $-log(4)$ when $p_{data} = p_{model}$ where $JSD(p_{data}||p_{model}) = 0$. It is therefore desirable to update the discriminator more than once before each generator update to better approximate the JSD and provide the best possible gradients of the JSD to the generator [12].

This can be achieved through a training procedure that uses alternating gradient descent, where k steps of discriminator updates are performed, followed by one generator update. However, this theoretical promise of convergence is only applicable if optimisation is done directly on the probability density function for p_{model}, i.e. in function space, rather than in parameter space. Therefore, this does not hold in practice where neural networks are optimised in parameter space and the GAN training procedure is therefore implemented as seen on algorithm 1, where we see

the alternating updates on line 2-7 for the discriminator and line 8-10 for the generator. We further note that updates are made in parameter space at line 5-6 and 9-10 to parameters θ and ϕ. Furthermore, when the minimax loss is used in practice, it causes vanishing gradients when the discriminator is consistently and confidently rejecting generated samples. This is because when $D(G(\mathbf{z}))$ approaches 0 (confident rejection), so will $log(1 - D(G(\mathbf{z})))$, thus resulting in a weak gradient from this term [10].

Algorithm 1: Batched stochastic gradient descent for training generative adversarial networks. m is the batch size, α is the learning rate. Any gradient-based optimiser can be used.

1 **for** *t = 0,1,2,...,number of epochs* **do**
2 **for** *k* **do**
3 $\mathbf{z}$ = Sample batch of m prior samples from p_z
4 $\mathbf{x}$ = Sample batch of m samples from p_{data}
5 $g_\theta = \nabla_\theta \frac{1}{m} \sum_{i=1}^{m} [log(D_\theta(\mathbf{x}_i)) + log(1 - D_\theta(G_\phi(\mathbf{z}_i)))]$
6 $\theta = \theta + \alpha \cdot SGD(\theta, g_\theta)$
7 **end**
8 z = Sample batch of m prior samples from p_z
9 $g_\phi = \nabla_\phi \frac{1}{m} \sum_{i=1}^{m} log(1 - D_\theta(G_\phi(\mathbf{z}_i)))$
10 $\phi = \phi - \alpha \cdot SGD(\phi, g_\phi)$
11 **end**

Due to the issue with the minimax loss [12] presents a second loss called the *non-saturating* loss which does not cause the gradients to vanish. This loss alters the generators objective during training from minimising $log(1 - D(G(\mathbf{z})))$ to maximising $log(D(G(\mathbf{z})))$. This change results in GAN training no longer being a minimax game and the generator loss becomes: $\max_G L_G = \mathbb{E}_{\mathbf{z} \sim p_z} log(D(G(\mathbf{z})))$ and the discriminator loss, which is unchanged from the minimax loss:

$\max_D L_D = \mathbb{E}_{\mathbf{x} \sim p_{data}} log(D(\mathbf{x})) + \mathbb{E}_{\mathbf{z} \sim p_z} log(1 - D(G(\mathbf{z})))$. As this loss does not suffer from vanishing gradients it tends to perform better in practice than the minimax loss, even though, there are no theoretical guarantees of convergence.

2.3 Common issues in GANs

Throughout this project, we encounter two main issues that cause GAN learning to fail. Namely, the issue of the discriminator loss quickly moving to 0, inhibiting further learning, and the issue of mode collapse. We describe these issues and their suspected causes in the following subsections.

2.3.1 JSD instability under an optimal discriminator

When training the discriminator the maximum value of its loss should be $-log(4) + 2JSD(p_{data}||p_{model})$, however, in practice the loss will go to 0 when continuously training the discriminator to achieve a better JSD approximation. [2] proposes that this occurrence is caused by the JSD between p_{data} and p_{model} maxing out at a value of $log(2)$, i.e. becoming constant and inhibiting learning. This can occur when the supports of the two distributions do not overlap. In this case, there exists an optimal discriminator that can perfectly discern real samples from generated ones [2]. As such under an optimal discriminator the GAN loss becomes $-log(4) + 2JSD(p_{data}||p_{model}) = -log(4) + 2log(2) = 0$, which does not provide a useful gradient.

We illustrate the issue of the JSD maxing out on Figure 2.7. Here, we measure the JSD between p (mean=0, standard deviation=0.1) and multiple identical distributions with shifted means ranging from 0 to 20 and plot the development in the JSD. As such, the JSD plot consists of 20 distance measurements and we see that as the distributions move apart, the distance increases until they no longer overlap in which case it becomes constant. Examples of distributions used for this experiments are shown as $q1, q2, q3$. The Python script for reproducing this result can be found in Appendix C. As the JSD becomes constant minimisation through gradient descent is not going to yield any useful learning and, as we have seen, this is exactly what the discriminator training step approximates.

2.3.2 Mode collapse

The goal of a GAN is to represent a probability distribution by learning a mapping from the distribution p_z to p_{data}, and through this mapping, be able to generate samples from the entire support of p_{data}. When the GAN fails to learn the entire support by, e.g. mapping too many $\mathbf{z} \sim p_z$ to the same $\mathbf{x} \sim p_{data}$ it is called mode collapse and this reduces the variance of generated images greatly. We visualise mode collapse on a 2D dataset later in this subsection and show samples from a

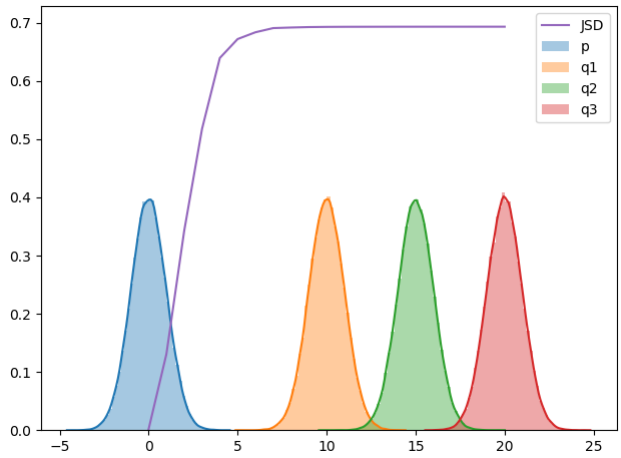


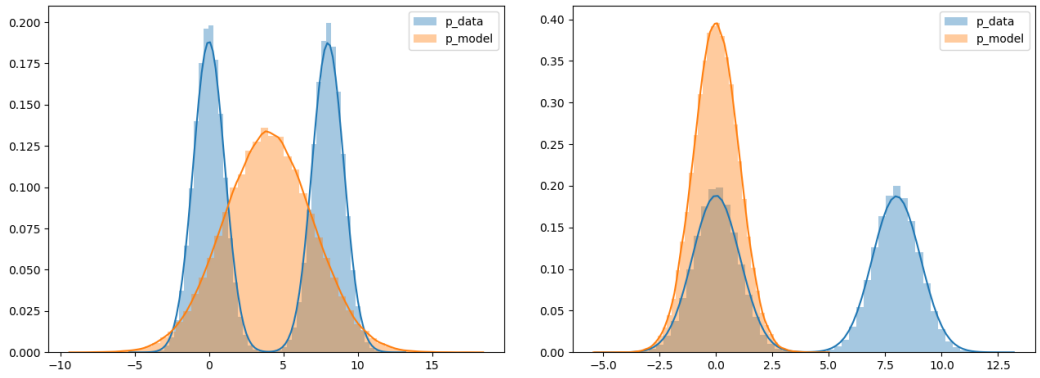
Figure 2.7: JSD behaviour when distributions are moved apart

mode collapsed CoGAN in section 3.3. Mode collapse can occur since the GAN loss does not contain any term that enforces learning the full distribution, instead the loss measures how well the generator is currently fooling the discriminator. Therefore, GAN learning can be viewed as optimising towards one specific image that is best at fooling the discriminator, as opposed to e.g. variational autoencoders [26], which enforces that data can be generated from the entirety of a parameterised distribution e.g. a Gaussian distribution.

A formal explanation of mode collapse is presented in [2] wherein we see that when using the non-saturating loss the gradient update to the generator for a fixed and optimal discriminator, D^*, corresponds to

$$\nabla_\phi \mathbb{E}_{\mathbf{z}\sim p_z} log(1 - D^*(G_\phi(\mathbf{z}))) = \nabla_\phi KLD(p_{model,\phi}||p_{data}) - 2JSD(p_{model,\phi}||p_{data}).$$

We see that the update contains a negated JSD computation between model and data distribution meaning that this term is actually pushing the distributions further apart. It also contains a KLD term which is the cause of the mode collapse. The KLD is asymmetrical meaning that $KLD(p_{model}||p_{data}) \neq KLD(p_{data}||p_{model})$. Figure 2.8 illustrates this asymmetry and how it relates to mode collapse. On this figure, we depict our p_{data} as a mixture of Gaussians that we wish p_{model} to approximate using minimisation of the KLD between them. In Figure 2.8b we see that using the KLD term present in the generator update for approximation yields a p_{model} that learns only one Guassian, this happens because this term places high cost on generating images outside p_{data} (due to division by 0 when $p_{data}(\mathbf{x}) = 0$), thus pushing the generator towards learning one mode really well, but dropping the others [10, 2].



a) $KLD(p_{data}||p_{model})$ b) $KLD(p_{model}||p_{data})$

Figure 2.8: KLD behaviour when used for approximation

As a result, GANs are prone to experience mode collapse where the variance of generated samples diminishes heavily, potentially to the point where the GAN generates the same image for every distinct latent vector. This could for example occur when the discriminator rejects the majority of generated samples, which pushes the generator to find an area in the latent space or perhaps a single latent vector that produces samples that are not rejected. Due to sample variance not being enforced, the generator may then choose to map all latent points towards this area yielding a mode collapsed GAN.

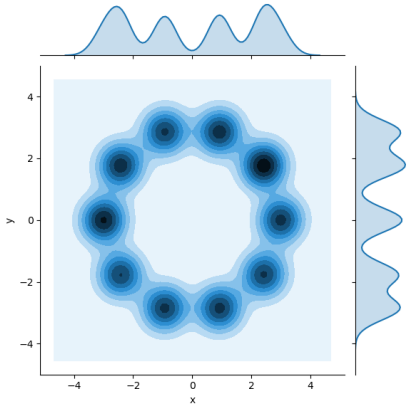


Figure 2.9: Toy dataset for mode collapse experiments

In our previous work [1] with GANs, from last semester, we conducted an experiment to visualise the effect of mode collapse and we reuse those results here. This experiment was conducted on a synthetic dataset consisting of 5100 two dimensional data points arranged in a circular shape as shown on Figure 2.9. The goal

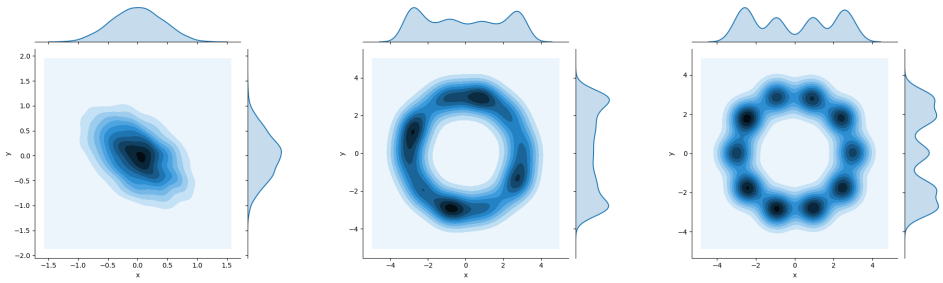
Generator	Activation	Neurons
Input	-	100
Fully connected	Tanh	128
Fully connected	Tanh	128
Fully connected	Tanh	128
Output	Linear	2

Discriminator	Activation	Neurons
Input	-	2
Fully connected	Tanh	128
Fully connected	Tanh	128
Fully connected	Tanh	128
Output	Sigmoid	1

Table 2.1: Neural net architectures used for exploring mode collapse in GANs

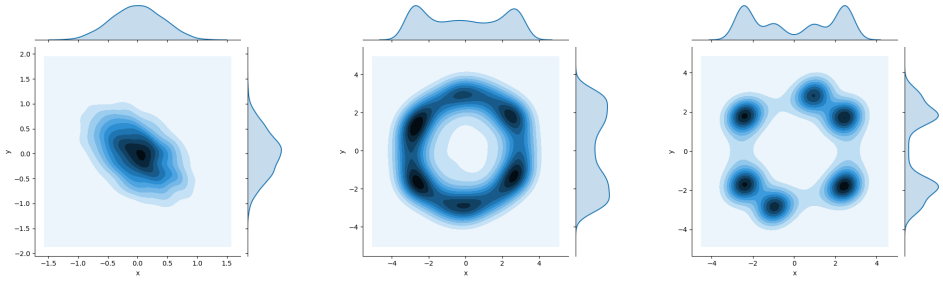
for the GAN is to match this distribution. Architecture details for this experiment is shown in Table 2.1. We show in Figure 2.10 that a GAN can successfully learn this distribution with the following hyperparameter settings: $gen\_lr = 0.0001, disc\_lr = 0.0001, z\_size = 10, k = 1, optimiser = ADAM, batch\_size = 100, iterations = 2000$.

Furthermore, we show in Figure 2.11 that when changing this configuration by increasing the amount of discriminator training iterations to 5 the GAN now mode collapses. According to the theoretical results regarding GANs, this change should improve the training as the discriminator now provides a more accurate approximation of the JSD between the two distributions, however, this is not the case in practice. Here we see that the discriminator potentially becomes too strong in comparison to the generator and even though at 1200 iterations the two GANs are roughly equal in their understanding of the data distribution it later resolves to only learn 6 out of 10 modes. This experiment further supports that the theoretical claims of GAN convergence do not hold in practice.



0 iterations 1200 iterations 2000 iterations

Figure 2.10: Experiment with hyperparameters for GAN that do not cause mode collapse



0 iterations 1200 iterations 2000 iterations

Figure 2.11: Experiment resulting in mode collapse when over training the discriminator (k=5)

2.4 Wasserstein GAN

The Wasserstein GAN (WGAN) is proposed in [3] with the purpose of alleviating the mode collapse and instability issues observed with the regular and non-saturating GAN losses. This is done by replacing the GAN loss with a function that approximates the Wasserstein distance (WD), rather than the JSD between p_{data} and p_{model}.

The motivation being that the WD does not become constant when the two distributions do not overlap, which is common, especially, in the beginning of training, and thus provides a better gradient in those situations. To show this we revisit the example from subsection 2.3.1 and we see on Figure 2.12 that the value of the WD does not stagnate like that of the JSD, which is a desirable property for optimisation through gradient descent. The WD is measured using the SciPy implementation[2] and the full Python script can be found in Appendix C. The following sections describe Wasserstein distance and how it is used as a loss function in GANs, as well as the improved version of WGAN called WGAN-GP which we utilise in our some of our models later in this project.

2.4.1 Wasserstein distance

The Wasserstein distance is a metric, which stems from the field of optimal transport and can be used to measure the distance between two probability distributions. It is also known as the Earth Movers distance, since it is often, informally, explained as corresponding to the minimum cost of moving and transforming one

[2] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.wasserstein_distance.html`

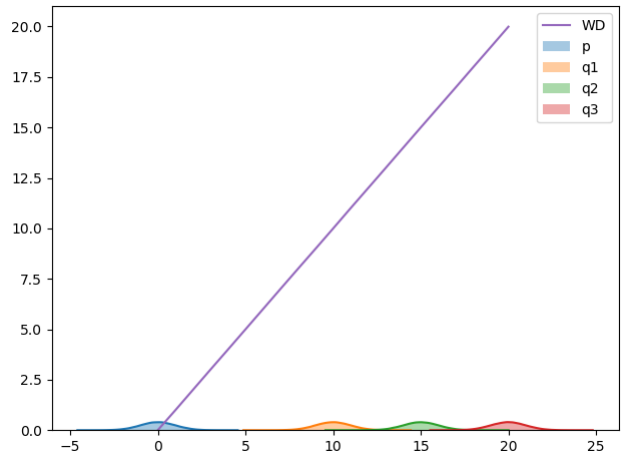


Figure 2.12: WD behaviour when distributions are moved further apart

pile of dirt, shaped as a distribution, to the shape of another distribution. The cost can be quantified as the amount of dirt moved times the distance it is moved [40].

To give an example of this intuition, we consider two discrete probability distributions P and Q that each has three piles of dirt and five shovelfuls distributed in the piles. These distributions could be defined as:

$$P_1 = 1, P_2 = 3, P_3 = 1$$
$$Q_1 = 2, Q_2 = 1, Q_3 = 2$$

Multiple solutions exist for transforming P into Q. Each solution is called a work plan, as it describes one way of performing the necessary work and the Wasserstein distance corresponds to the cheapest work plan. Here, the cheapest work plan would be moving one shovelful of dirt from P_2, to P_1 and P_3. However, another, more expensive, work plan could be to move one shovelful of dirt from P_1 to P_3 and then move two shovelfuls from P_2 to P_1. In the discrete case, the Wasserstein distance is calculated by first finding the cost for each pile and then summing the absolute value of those costs:

$$cost_i = cost_{i-1} + P_i - Q_i$$
$$WD_{discrete} = \sum_{j}^{m} |cost_j|$$

Here m is the total amount of piles. For this example, the cheapest cost is therefore:

$$cost_1 = 0 + 1 - 2 = -1$$
$$cost_2 = cost_1 + 3 - 1 = 1$$
$$cost_3 = cost_2 + 1 - 2 = 0$$
$$WD_{discrete} = \sum_{j}^{3} |cost_j| = 2$$

We can extend the discrete example such the P and Q are now continuous distributions in which case the Wasserstein distance is defined as

$WD(P,Q) = \inf_{\gamma \in \Pi(P,Q)} \mathbb{E}_{(x,y)\sim\gamma}||x-y||$. Here $\gamma(x,y)$ represents some work plan from a set of potential plans. In the continuous case, a work plan is a joint distribution with marginals P and Q and $\Pi(P,Q)$ is the set of all joint distributions. In this sense, $\gamma(x,y)$ describes how much mass must be moved from x to y, in order to transform P to Q. *inf* is short for infimum, which represents the greatest lower bound (minimum) for a set and is used for selecting the plan with the lowest cost.

2.4.2 Wasserstein distance as a GAN loss

The Wasserstein distance cannot be directly used as a loss, since computing the infimum over all possible work plans is often intractable in practice as it requires us to compute all possible joint distributions in $\Pi(P,Q)$. Therefore, [3] proposes to use a reformulation of the Wasserstein distance found by applying the Kantorovich-Rubinstein duality and is defined as $WD(P,Q) = \sup_{f} \mathbb{E}_{\mathbf{x}\sim P}[f(\mathbf{x})] - \mathbb{E}_{\mathbf{x}\sim Q}[f(\mathbf{x})]$. Exactly how this is done, is described in chapter 6 of [38].

Here, the supremum is over all 1-Lipschitz functions, $f : X \to \mathbb{R}$, where in our case X is the space of all images. In order to calculate the Wasserstein distance, we have to find the function f that maximises the expression. A real valued function $g : \mathbb{R} \to \mathbb{R}$ is K-Lipschitz if there is a constant $K \geq 0$ where $|g(x_1) - g(x_2)| = K \cdot |x_1 - x_2|$ for any $x_1, x_2 \in \mathbb{R}$. Intuitively for a function to be K-Lipschitz means that there is some limit K on the slope of the function. Assuming that f origins from a family of K-Lipschitz functions parameterised by θ, $\{f_\theta\}_{\theta\in\Theta}$, where Θ is a compact space, we can estimate f by solving the following maximisation problem: $\max_{\theta} \mathbb{E}_{\mathbf{x}\sim P}[f_\theta(\mathbf{x})] - \mathbb{E}_{\mathbf{x}\sim Q}[f_\theta(G(\mathbf{x}))]$. Finding θ such that $f_\theta = f$ yields an approximation of the Wasserstein distance between the two distributions up to a multiplicative constant, namely $K \cdot WD(P,Q)$.

This is employed as a GAN loss:

$\min_G \max_D V_{WGAN}(G, D) = \mathbb{E}_{\mathbf{x}\sim p_{data}}[D(\mathbf{x})] - \mathbb{E}_{\mathbf{z}\sim p_z}[D(G(\mathbf{z}))]$. Here we solve the maximisation problem to estimate f using the discriminative model, which, as a result, has to be K-Lipschitz. As f is a real valued function the discriminator is no longer trained to discriminate. Instead, it outputs a value in the range $[-\infty, \infty]$ interpreted as the realness of a given image, rather than a classification. The implications of this, in a practical implementation, is simply that the last Sigmoid layer used for binary classification is removed. As a result, the discriminator is renamed the critic in [3], but for simplicity we stick to the term discriminator in the remainder of this report, as well as, use D to represent it. To ensure that the discriminator is K-Lipschitz [3] proposes to clip the weights θ of the discriminator after every update to ensure that they lie in a compact space. In [3], they propose to use the clipping range $[-0.01, 0.01]$. Training is performed in the same way as shown in algorithm 1 using alternating gradient descent with weight clipping after each discriminator update. When training in practice, [3] proposes to update the discriminator five times for each generator update.

Seeing as the discriminators loss approximates the Wasserstein distance it is in our interest to update the discriminator multiple times to improve this approximation. Opposed to the minimax and non-saturating losses that fail to provide good gradients when the discriminator is overtrained, the WGAN loss benefits from this, and due to the continuous nature of this distance measure provide better gradients for the generator with a better approximation. An additional improvement that WGAN brings to the GAN framework is that the loss now correlates with sample quality, which is not the case for the minimax and non-saturating losses. This is an important contribution as it enables another, although not perfect, method of evaluating trained GANs, rather than only visual inspection of generated samples. Furthermore, WGANs alleviates the mode collapse issue as the formal description presented in subsection 2.3.2 is no longer true for WGAN loss.

2.5 Wasserstein GAN with gradient penalty

While the WGAN loss does provide improvements to the GAN framework, both theoretical and in practice, the authors acknowledge that: *"Weight clipping is a clearly terrible way of enforcing a Lipschitz constraint"* [3]. This approach inhibits learning and limits the discriminators capability of modelling more complex functions [3, 13]. Furthermore, the optimal 1-Lipschitz function to maximise the WGAN loss has gradient norm of 1 and when training a WGAN the gradient norms tend to either explode or vanish depending on the clipping range [13]. In cases where the

clipping parameter is not well tuned, WGAN will not achieve an accurate approximation of the Wasserstein distance and therefore not provide accurate gradients for the generator to follow. It is therefore desirable to find another way of enforcing the Lipschitz constraint.

With this motivation, [13] proposes to add a regularisation term to the WGAN loss that penalises the gradients of the discriminator with respect to its input whenever the gradient norm strays from 1. As it is intractable to compute for the gradients for every possible real and fake input and measure the gradient norm they utilise the fact that the discriminator must also have a gradient norm equal to 1 for any interpolated sample $\hat{\mathbf{x}} = interpolate(\mathbf{x}, \mathbf{y}, \alpha) = \alpha\mathbf{x} + (1 - \alpha)\mathbf{y}$ where $\mathbf{x} \sim p_{data}$, $\mathbf{y} \sim p_{model}$ and α is drawn from a uniform distribution with range [0,1]. These interpolations lie on straight lines between real and generated samples and α decides the exact position along the line. Figure 2.13 illustrates this idea of interpolating images along a line.

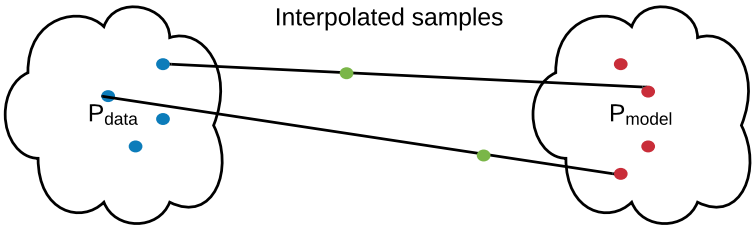


Figure 2.13: Interpolated samples along straight lines. The position of the green interpolations are decided by α

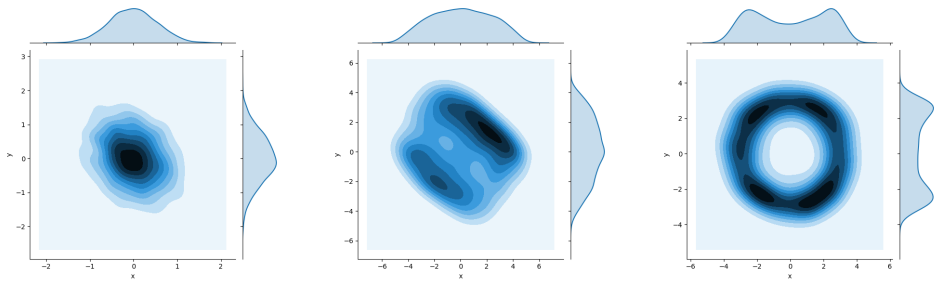
As such, this regularisation term penalises the gradient norm at randomly sampled interpolations $\hat{\mathbf{x}}$. Equation 2.1 shows the WGAN loss with the gradient penalty term added. Here, λ is a hyperparameter that we must set and its purpose is to scale the effect of the penalty. [13] recommends $\lambda = 10$. The L2 norm of the discriminators gradients with respect to its input is represented as $||\nabla_{\hat{\mathbf{x}}} f(\hat{\mathbf{x}})||_2$.

$$\min_G \max_D V_{WGAN-GP}(G, D) = V_{WGAN}(G, D) + \lambda \mathbb{E}_{\mathbf{x} \sim p_{data}, \mathbf{y} \sim p_{model}} (||\nabla_{\hat{\mathbf{x}}} f(\hat{\mathbf{x}})||_2 - 1)^2 \tag{2.1}$$

$$where \quad \hat{\mathbf{x}} = interpolate(\mathbf{x}, \mathbf{y}, \alpha)$$

In practice there are two key differences when using the WGAN-GP approach. The first is that batch normalisation should no longer be used in the discriminator. This is because the regularisation term penalises the gradient norm with respect to each independent input and batch normalisation creates correlations between samples in batches. They instead recommend using layer normalisation as this approach does not create such correlations. Secondly, they propose a new set of hyperparameter settings for stable training which are as follows: ADAM parameters beta1 and beta2 should be set to 0 and 0.9 respectively and the learning rate should be set to 0.0001.

This approach has been empirically shown to retain the improvements made by WGAN such as meaningful loss and reduced mode collapse. To display these properties gained by introducing this loss in GAN training we reuse the example regarding mode collapse shown in subsection 2.3.2. We train the mode collapsed configuration now using the WGAN-GP loss and its recommended hyperparameter settings and display results in Figure 2.14. Here we see that WGAN-GP stabilises training and after 3000 iterations the model has learned the shape of the distribution, however, it is less precise in the modes it has learned. Furthermore Figure 2.15 shows how the discriminator loss, i.e. the Wasserstein estimate, decreases as the model learns to represent the correct distribution.



0 iterations 2000 iterations 3000 iterations

Figure 2.14: Experiment showing WGAN-GP alleviating mode collapse on GAN configuration that collapses with non-saturating loss

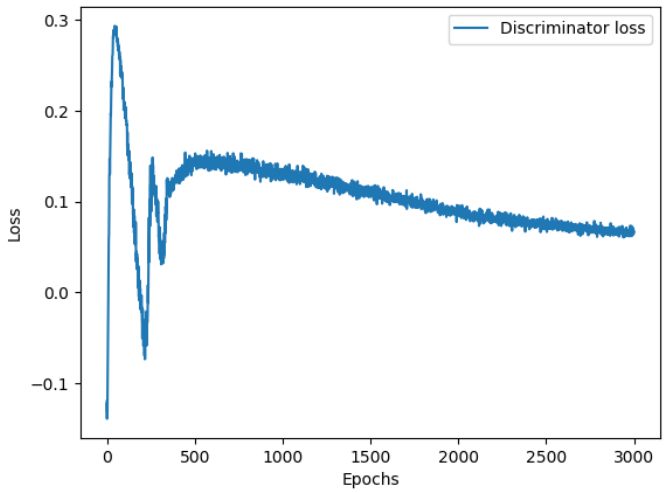


Figure 2.15: Discriminator loss (WD estimate) for WGAN-GP training

2.6 Deep Convolutional GAN

When GANs were first proposed in 2014, the generator and discriminator models were both implemented as neural nets that consisted solely of fully connected layers and pooling layers. However, in 2015, this changed when Deep Convolutional GANs (DCGANs) were proposed in [37]. DCGANs are based on knowledge about model design in computer vision tasks and describes how convolutional layers can be incorporated in GAN architectures to stabilise training and achieve superior image quality. This improved performance using convolutional architectures is shown empirically, rather than based on mathematical proofs. [37] presents 5 rules for building DCGAN architectures which are as follows:

1. Replace pooling layers with strided convolutions in the discriminator and transpose convolutions in the generator.
2. Use batch normalisation in both generator and discriminator.
3. Remove fully connected hidden layers for deeper architectures.
4. Use ReLU activations in the generator for all layers except the output, which uses Tanh.
5. Use LeakyReLU activations in all layers in the discriminator.

By training models that follow these rules for architecture design on multiple datasets they found that the adaptive learning rate optimiser ADAM [27] with the following hyperparameter settings performs the best: $\beta_1 = 0.5$ and $\beta_2 = 0.999$, the learning rate should be set to 0.0002, LeakyReLU leak should be set to 0.2, all weights should be initialised from a normal distribution with standard deviation of 0.2 and mean 0, batch size should be set to 128 and input images must be normalised to [-1,1], to fit the Tanh activation. DCGANs have, since their publication, been adopted as the default pattern for designing GAN architectures and have been used as baselines in many branches of GAN research such as [3, 13, 22, 32, 6].

2.7 Datasets

This section provides an overview of the datasets used in this project: MNIST, MNISTedge, MNISTrotate, Street View House Numbers (SVHN), CelebA, apple2orange and horse2zebra.

MNIST

MNIST is a well known and often used dataset in the field of machine learning. It consists of 60.000, 28x28 images of handwritten white digits(1-9) on a black background. In this project, whenever we use MNIST we scale the images to 32x32. It was proposed in [41] as a downloadable dataset, but has since been made available through Tensorflow Datasets for easier access which is what we have used. Figure 2.16 shows examples of MNIST images.

Figure 2.16: Randomly selected samples from the MNIST dataset.

MNISTedge and negative

These two datasets are transformations of the MNIST dataset. MNISTnegative is created by negating colours of original MNIST images as shown on Figure 2.17. MNISTedge can be created by using the dilate function in the python library,

OpenCV and subtracting the original MNIST image from the dilated version. Resulting images are shown on Figure 2.18. These datasets are used in conjunction with the MNIST dataset for joint distribution learning in which MNIST represents one domain and edge/negative represents the other. We therefore select 50% of the MNIST dataset and convert it into either edge/rotate and thus the size of MNIST and edge/negative is 30.000 images each.

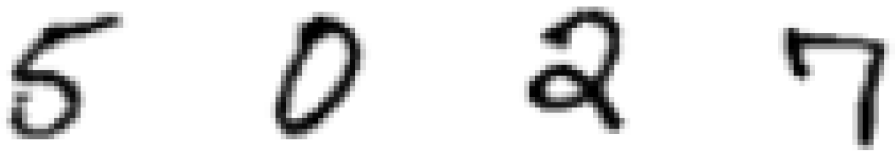


Figure 2.17: Randomly selected samples from MNISTnegative.

Figure 2.18: Randomly selected samples from MNISTedge.

Street View House Numbers(SVHN)

This dataset consists of 73.257, 32x32 real-life images of house numbers(0-9). While this dataset is much alike MNIST in image size and main content SVHN images are generally noisier and blurrier. Some images may also contain more than one digit where the centred digit is what the image is labelled as. Examples of SVHN images can be seen on Figure 2.19 in which the images from left to right are labelled as: 1,1,0,3 even though three of the images contain multiple numbers. This dataset is available for download through `http://ufldl.stanford.edu/housenumbers/`, however we acquired it through the Tensorflow Datasets API. We use a pruned version of this dataset where we remove a subset of blurry images such as the rightmost one on Figure 2.19. This is done to improve quality of generated images. We pruned the dataset by training a classifier(details found in chapter 5) and removing all samples where the highest prediction confidence is below or equal to 40%. This resulted in a dataset of size 70.262 with fewer blurry images.

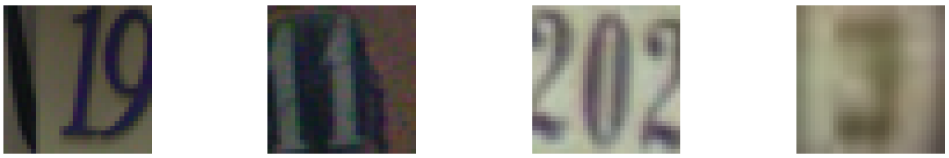


Figure 2.19: Randomly selected samples from the SVHN dataset.

CelebA

CelebA consists of 202.599 face images of celebrities and was first proposed in [31]. Each image is annotated with 40 binary attributes, some examples of attributes are: big_nose, brown_hair, and goatee. Samples are shown on Figure 2.20. Through the attributes the dataset can be split into multiple domains e.g. one domain would only contain images with blond hair and the second domain would only contain images with brown hair. The dataset is made available for download through `http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html` which is what we used to collect it. All images are of size 178x218 and unless otherwise stated when we use the CelebA dataset we scale the images to 128x128. In this project we run experiments on three different subsets of this dataset namely (1) faces with and without glasses called CelebEyeglasses, (2) faces with blond and non blond hair called CelebBlond and (3) faces that are smiling and not smiling called CelebSmiling. The size of each dataset is shown in Table 2.2.

Attribute	CelebSmiling	CelebBlond	CelebEyeglasses
With attribute	97,669	29,983	13,193
Without attribute	104,930	172,616	189,406

Table 2.2: Overview of CelebA dataset sizes

Figure 2.20: Randomly selected samples from the CelebA dataset

CycleGAN datasets

In 2017 the CycleGAN [44] paper was released and it introduced an array of 2-domain datasets, which are ideal for the image-to-image translation tasks that CycleGAN was solving. CycleGAN datasets contain fewer images than the other datasets presented in this chapter and all the images are of size 256x256. We have used two of their proposed datasets, namely *apple2orange* and *horse2zebra*. Apple2orange consists of 995 images of apples and 1019 images of oranges in different noisy contexts. Examples are shown on Figure 2.21. The horse2zebra dataset consists of 1067 horse images and 1334 zebra images, also in noisy contexts. Examples are shown on Figure 2.22. All CycleGAN datasets are available through the Tensorflow Datasets API which is what we have used to collect them.

Figure 2.21: Randomly selected samples from the Apple2Orange dataset

Figure 2.22: Randomly selected samples from the Horse2Zebra dataset

Chapter 3 Problem analysis

In this chapter, we describe and experimentally analyse Coupled Generative Adversarial Nets (CoGAN) [34]. CoGAN is an extension of the GAN framework which enables GANs to represent joint probability distributions and not just marginal distributions. With our own implementation, we reproduce some of the results presented in the original paper and show two main issues that this approach suffers from. Furthermore we describe the evaluation metric we use to quantify the performance of trained models since we use it as part of the experimental analysis.

3.1 Coupled Generative Adversarial Networks

The main feat of CoGAN is that this approach manages to learn a joint distribution over multiple image domains using only the marginal distributions from each domain, rather than the joint distribution itself [34]. This is, especially, attractive since it can be difficult to construct datasets consisting of pairs of corresponding images to represent the joint distribution. Furthermore, CoGANs have been shown to be applicable within the fields of image-2-image translation and unsupervised domain adaptation [34].

CoGAN consists of multiple GANs coupled together by a weight sharing constraint between the generators and discriminators respectively. Specifically, it requires one GAN per domain in the joint distribution we wish to learn. As such, CoGANs can be scaled up to learn k domains by coupling k GANs together. However, in this project we focus mainly on the case of learning a joint distribution over two domains.

CoGANs rely on a core idea that specific layers in neural networks decode specific levels of detail in images. Thereby when the weights of specific layers in the GAN that decodes high-level features, i.e. the general shapes and larger features, are shared between models, these are forced to perform this decoding in the same way and obtain a shared high-level image representation. By not weight sharing the layers that decode low-level features each GAN can style this shared representation to its specific domain. In the case of facial images, the high-level features may be the shape of the face and placement of eyes, ears, mouth and so on. While the low-level features may be eye or hair colour, or facial expression.

Weights are shared between generators and discriminators respectively as shown on Figure 3.1. For the generators the weight sharing can be written, in the case

of learning a joint distribution over two domains, as follows: Let $G1$ and $G2$ be the two generators of GAN_1 and GAN_2. Both GANs are implemented as neural networks and can therefore be written as a sequence of transformation on an input vector $\mathbf{z} \sim p_z$ as seen below. Here, $G_1^{(i)}$ and $G_1^{(i)}$ represents the i'th layer in the neural net and m_1 and m_2 are the total number of layers.

$$G_1(\mathbf{z}) = G_1^{(m_1)}(G_1^{(m_1-1)}(...(G_1^{(2)}(G_1^{(1)}(\mathbf{z}))), \qquad G_2(\mathbf{z}) = G_2^{(m_2)}(G_2^{(m_2-1)}(...(G_2^{(2)}(G_2^{(1)}(\mathbf{z})))$$

In the generators the first layers decode high-level features [34] and later layers decode low-level features. Therefore, the weight sharing between the generators occur in the first layers such that $\phi_{G_1^{(i)}} = \phi_{G_2^{(i)}}$ for $i = 1, 2, 3, ..., k$ where i represents the total amount of shared layers and $\phi_{G_1^{(i)}}$ and $\phi_{G_2^{(i)}}$ are the trainable weights of layer i in each model.

The discriminators are represented in a similar manner as shown below. Here, $\mathbf{x}_1 \sim p_{d1}$ and $\mathbf{x}_2 \sim p_{d2}$ are images from the two domains that we wish to learn a joint distribution over. n_1 and n_2 are the total amount of layers in either discriminator D_1 and D_2.

$$D_1(\mathbf{x_1}) = D_1^{(n_1)}(D_1^{(n_1-1)}(...(D_1^{(2)}(D_1^{(1)}(\mathbf{x_1}))), \qquad D_2(\mathbf{x_2}) = D_2^{(n_2)}(D_2^{(n_2-1)}(...(D_2^{(2)}(D_2^{(1)}(\mathbf{x_2})))$$

In discriminative models, the last layers are responsible for encoding high-level features and therefore the discriminators share the weights of those layers such that: $\theta_{D_1^{(n_1-j)}} = \theta_{D_2^{(n_2-j)}}$ for $j = 0, 1, 2, ..., l-1$, where l represents the total amount of shared layers and $\theta_{D_1^{(j)}}$ and $\theta_{D_2^{(j)}}$ are the trainable weights of layer i in each model.

It is possible for the neural nets to be of different sizes, $m_1 \neq m_2$ and $n_1 \neq n_2$, in this case we can only share layers corresponding to $minimum(m_1, m_2)$ and $minimum(n_1, n_2)$. Additionally, layers that share weights have to share structure as well i.e. if we wish to share weights between layer 1 of G_1 and G_2 they must be the same type of layer and have the same amount of weights. This also applies to the discriminators.

Figure 3.1 shows how the two GANs are employed in the CoGAN architecture. Both GANs are given the same input vector, $\mathbf{z} \sim p_z$, as in regular GAN training, which is passed into the shared layers of each generator to produce the shared representation of the generated image. This is then passed to the unshared layers

to add details to each image, thus styling them to the specific domain. In the figure, we use MNIST and MNISTedge as the two domains. The generated numbers from either generator must have the same high-level features, i.e. the number itself and different low-level features, i.e. either full line numbers of edge versions. As with regular GAN training, the discriminator is passed either a generated or a real image and in the case of CoGAN the real images passed to both discriminators do not necessarily need to have the same high-level features. The discriminators then evaluate the image they receive as either real or fake.

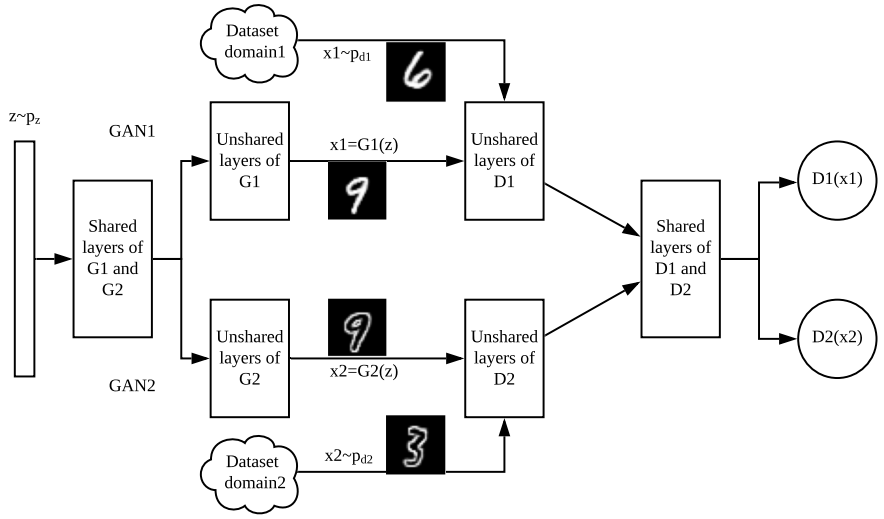


Figure 3.1: CoGAN architecture for joint distribution with two domains

$$\min_{G_1,G_2} \max_{D_1,D_2} V_{CoGAN}(G_1,G_2,D_1,D_2) = \mathbb{E}_{\mathbf{x}_1\sim p_{data1}} log(D_1(\mathbf{x}_1)) - \mathbb{E}_{\mathbf{z}\sim p_z} log(1-D_1(G_1(\mathbf{z}))) + \mathbb{E}_{\mathbf{x}_2\sim p_{data2}} log(D_2(\mathbf{x}_2)) - \mathbb{E}_{\mathbf{z}\sim p_z} log(1-D_2(G_2(\mathbf{z})) \tag{3.1}$$

Training in the CoGAN framework is very similar to regular GAN training in that it is performed using alternating gradient descent. The main difference is that we have to account for the weight sharing and ensure that updates performed on shared layers are the same between the two models. We show how this can be implemented using Tensorflow and Keras in chapter 6. The loss function for CoGANs, shown in Equation 3.1, is therefore also quite similar to that of regular GANs. Here, we see that it has simply been expanded to include the second GAN. With the important detail that $V_{CoGAN}(D_1,D_2,G_1,G_2)$ is subject to the weight sharing constraints on the generators and discriminators, which were defined earlier.

Intuitively, the generator and discriminators of CoGAN perform the same objective as in regular GAN training i.e. the generator for one domain attempts to best fit its given data distribution and the discriminator attempts to discern between generated and real images. In the case of CoGAN, the training can now be viewed as teams of generators and discriminators that collaborate through weight sharing to best complete their respective objectives. Note that, with this combined loss it is still possible to employ the non-saturating loss or the Wasserstein loss as well as add penalties such as the gradient penalty as previously described. As an example of this, Equation 3.2 shows the CoGAN loss function using the Wasserstein loss.

$$\begin{aligned}\min_{G_1,G_2}\max_{D_1,D_2} V_{WCoGAN}(G_1,G_2,D_1,D_2) = \mathbb{E}_{\mathbf{x}_1\sim p_{data1}} D_1(\mathbf{x}_1) - \mathbb{E}_{\mathbf{z}\sim p_z} D_1(G_1(\mathbf{z})) \\ + \mathbb{E}_{\mathbf{x}_2\sim p_{data2}} D_2(\mathbf{x}_2) - \mathbb{E}_{\mathbf{z}\sim p_z} D_2(G_2(\mathbf{z})\end{aligned} \tag{3.2}$$

3.2 Metric for model evaluation

In order objectively evaluate model performance we are interested in a quantitative metric which measures similarity between images. However quantitative metrics for evaluating image generation models is currently a subject that is under development. Metrics currently used in literature [15, 29, 43] are based on comparison between feature representations, extracted from images. These metrics require sampling a large amount of images in an attempt to evaluate samples from the entire support of the model distribution. Relying on random sampling to cover the entire distribution causes uncertainty in how accurate the estimated score is, however we find no other approaches.

When analysing the CoGAN framework we therefore use a combination of visual inspection and Learned Perceptual Image Patch Similarity(LPIPS) distances [43] to assess and discuss the capabilities and issues of the framework. LPIPS distance is a measurement for perceptual similarity between images that is aimed at simulating human visual perception. This means that LPIPS assigns a low score to an image pair if they perceptually resemble each other. This metric measures the difference in feature maps extracted by a pretrained classifier on two input images. If the difference between the feature maps is small the similarity of images is higher, which means that if an image pair has LPIPS distance of 0, they are perceptually very similar. We choose to use this metric, since it has been used by other papers within the field of style transfer [17, 8], and it provides a measurement on the similarity between images that is more complex than per-pixel difference.

LPIPS distance is measured as $d(\mathbf{x}, \mathbf{x}_0) = \sum_l \frac{1}{H_l W_l} \sum_{h,w} ||w_l \odot (\delta(\mathbf{x})^l_{hw} - \phi(\mathbf{x}_0)^l_{0hw})||_2^2$ where $\mathbf{x}$ and $\mathbf{x}_0$ are input images, l is the number of layers extracted by δ, which is a pretrained classifier(VGG, AlexNet, or SqueezeNet) used as a feature extractor. H_l and W_l are height and width dimension of feature map l. w_l is a scaling vector, $\odot$ is the pairwise multiplication operator and $\delta(\mathbf{x})^l_{hw}$ is the channel value of feature map l extracted by δ with input $\mathbf{x}$. Lastly, it is noted that δ also normalises the feature maps. All reported LPIPS distances are calculated through a cloned version of the official implementation[1]. We use the 0.1 version of the official implementation with all default settings.

3.3 CoGAN analysis

In this section, we present an experimental analysis of the CoGAN framework. To perform this analysis, we implement a training framework in Python using Tensorflow that allows us to easily define model architectures and train CoGANs with different hyperparameter settings. Details of our implementation of the training framework can be found in chapter 6. We train CoGANs on a variety of two-domain problems.

In these experiments, we use architectures identical to those used in the official Github implementation of CoGAN[2], and presented in Table 3.1 and 3.2. The BatchNorm and Activation columns describe whether batch normalisation is applied after the given layer in a row and which activation function is then applied. If a layer is followed by batch normalisation, it is always applied before the activation function. A '-' in the activation column means that the identity function is applied as the activation. To avoid cluttering the following chapters and sections with large tables of all used architectures, every time there is a reference to an architecture it can be found in Appendix A. We also use the same optimiser, hyperparameter settings (ADAM optimiser with DCGAN settings), and training time of 25000 iterations [3] In all experiments conducted in this project, we scale pixel values from [0,255] to [-1,1].

Resulting samples of the experiments on the MNIST variants are shown in Figure 3.2, where each image in the top half is generated with separate latent vectors and belong to one domain and the bottom half is the corresponding image in the

[1] `https://github.com/richzhang/PerceptualSimilarity`

[2] `https://github.com/mingyuliutw/CoGAN`

[3] However, we note that during CoGAN training on the MNIST variants, we already see similar features and whole digits around 2500-4000 iterations in. This is also shown in `https://github.com/palminde/P10Project`

Generator	Shape	Kernel	Stride	BatchNorm	Activation	Shared
Input	100	-	-	-	-	yes
Fully Connected	16384	-	-	-	-	yes
Reshape	4x4x1024	-	-	-	-	yes
ConvTranspose	4x4x1024	4	1	yes	PRelu(0.25)	yes
ConvTranspose	4x4x512	3	2	yes	PRelu(0.25)	yes
ConvTranspose	8x8x256	3	2	yes	PRelu(0.25)	yes
ConvTranspose	16x16x128	3	2	yes	PRelu(0.25)	yes
ConvTranspose	32x32x3	6	1	yes	PRelu(0.25)	no

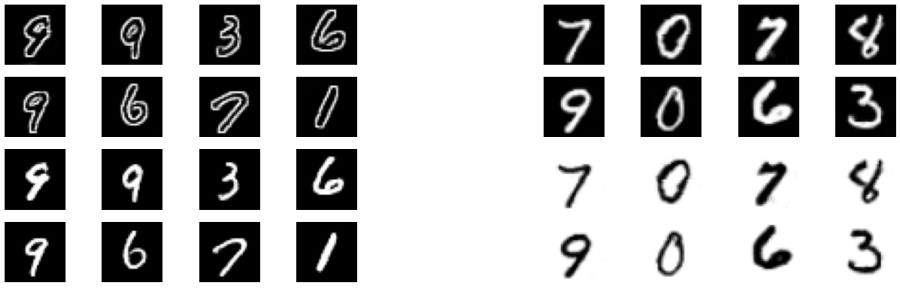
Table 3.1: CoGAN generator architecture for 32x32 image experiments

Discriminator	Shape	Kernel	Stride	BatchNorm	Activation	Shared
Input	32x32x3	-	-	-	-	no
Conv	32x32x20	5	1	-	-	no
MaxPool	16x16x20	2	2	-	-	no
Conv	16x16x50	5	1	-	-	yes
MaxPool	8x8x50	2	2	-	-	yes
Fully Connected	500	-	-	-	PRelu(0.25)	yes
Dropout(0.5)	500	-	-	-	-	yes
Fully Connected	1	-	-	-	Sigmoid	yes

Table 3.2: CoGAN discriminator architecture for 32x32 image experiments

second domain. In this way, the first image in row one is generated from the same latent vector as the first image in row three, and the second image in row one corresponds to the second image in row three and so on. When we display image samples for models that learn joint distributions they will always be presented following this pattern. We see that our implementation of CoGAN can successfully learn joint distributions over the digit datasets and is capable of producing pairs of corresponding images from both domains.

We did not manage to fully reproduce the results on the CelebA dataset. Using the architecture and hyperparameter settings described in the paper yielded results shown on Figure 3.3a. These samples do, clearly, not match those presented in the CoGAN paper and after thoroughly inspecting their Caffe [20] implementation, a library with which we are unfamiliar, we found several inconsistencies between their paper and the implementation. Firstly, the architectures used in their code did not fully match those shown in the paper differing in the first layer of the discriminator and with several dropout layers of the discriminator being left out completely. Additionally, they use L2 weight decay, which adds the following regularisation term to the loss: $c \sum_{i=0}^{I} w_i^2$. This term consists of the sum of squared



a) MNIST2edge b) MNIST2negative

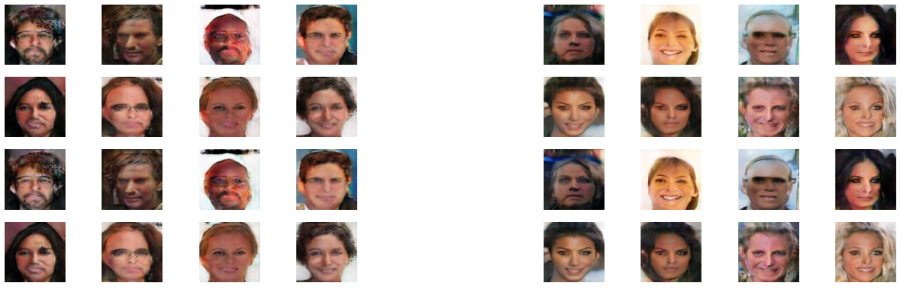
Figure 3.2: Reproduced CoGAN results with our own implementation

weights multiplied by some constant, in this case $c = 0.0001$. I is equal to the total number of weights in the model. L2 weight decay continuously squeezes the weights with the intuition of diminishing weight values that are high due to noise in the data. These will not, as easily, rise again as high weight values that identify actual features of the data.

Correcting these inconsistencies in our implementation improved the sharpness of the generated samples to a point where they are visually comparable to those presented in the CoGAN paper [34], as shown on Figure 3.3b. On these images, the top half is the domain with no attribute and the bottom half is the domain with the attribute. Although we see similar high-level features between the two domains we do not, as in the CoGAN paper, see the difference in low-level features that we aim for with one generator producing faces with glasses and one without. Additionally, we have trained on CelebSmiling and CelebBlond, shown on Figure 3.4, and see the same tendency of sharp corresponding images, but no distinct feature is learned on either dataset. The updated architecture used for the CelebA experiments can be found in Table A.1 and Table A.2.

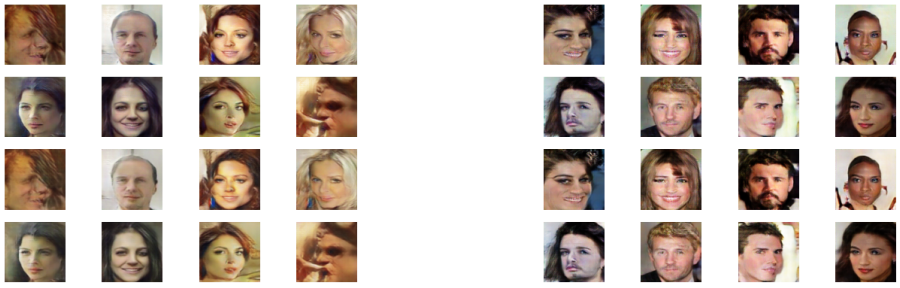
Since we are able to reproduce some results as well as achieve comparable image quality, we do not believe that the error origins from our implementation of CoGAN training, but rather from incomplete architecture or hyperparameter information about their CelebA experiments provided by the CoGAN paper and potentially us not being able to extract it from their implementation. All further experiments with CoGANs are therefore performed using our own implementation. We discuss this further in section 7.2.

Beyond our reproduction issues, we identify two underlying problems with the CoGAN framework, which are based on an important observation about its learn-



a) CelebEyeglasses b) CelebEyeglasses

Figure 3.3: Failed reproduction CoGAN results with our own implementation. a) is achieved using the architecture described in the CoGAN paper. b) is achieved using the architecture found in their Caffe implementation. Samples are taken after 35k iterations of training



a) CelebSmiling b) CelebBlond

Figure 3.4: Failed reproduction of CoGAN results with our own updated implementation.

ing capabilities. Namely, that to be able to learn a joint distribution over two domains, the domains are required to share high-level features [34]. While CoGAN performs reasonably well on the MNIST variants and CelebA, we show in the following sections that when the images become too large (256x256) or when the domains become too 'different' in either the content or the style of images CoGANs fail to learn anything meaningful. We name these issues the *scaling issue* and the *domain issue*.

3.3.1 Scaling issue

We train CoGANs on the horse2zebra and apple2orange datasets in order to evaluate the models performance on larger image sizes. For these experiments, we use the 128x128 architectures, but add an extra transposed convolution layer to the generators in order to generate 256x256 images. The architectures are shown in Table A.3 and A.4. The models are trained for 35k iterations using DCGAN hyperparameters. Samples from each model are shown on Figure 3.5 in which we see that CoGAN manages to produce a coupling between generated images in different domains, however quality of the images is very low and it is clear that the images are not generated as corresponding pairs.

An example of the learned coupling is seen on Figure 3.5a, where similar horse images maps to similar zebra images. On the horse2zebra dataset, it does not seem like the model manages to learn any correct shapes, although some of the zebra and horse styles are visible. As this can be difficult to visually assess, we calculate the average LPIPS distance between generated and real samples as shown in Table 3.3. Here, each generated and real set consists of 1000 samples, since we are limited by the small size of the real horse2zebra dataset. From these distance measures, we see that each generator is generating images that are more similar to their respective targets, which supports our visual assessment. The same observations can be seen for the apple2orange where samples are shown on Figure 3.5b. Here we also calculate LPIPS distance, seen in Table 3.4, and again the same pattern supports the visual inspection of the samples.

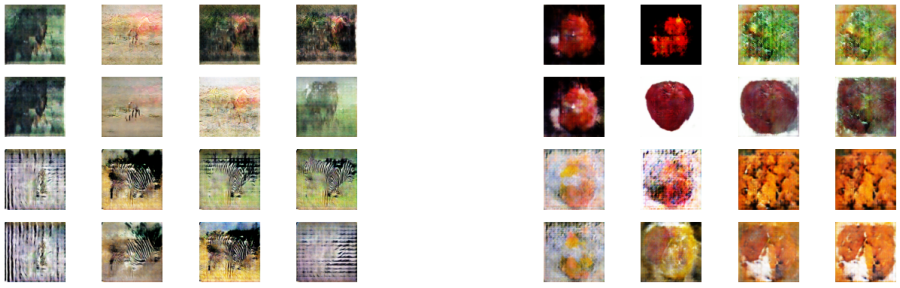
Gen./Real	Real horses	Real zebras
Gen. horses	0.7792	0.8185
Gen. zebras	0.7728	0.7546

Table 3.3: LPIPS distances between 256x256 horse2zebra dataset with CoGAN

Gen./Real	Real apples	Real oranges
Gen. apples	0.8388	0.8693
Gen. oranges	0.9005	0.8944

Table 3.4: LPIPS distances between 256x256 apple2orange dataset with CoGAN

This image quality issue that occurs when scaling up CoGAN is most likely caused by the fact that GAN architectures are not easily scalable to higher resolution images (>256x256), rather than a direct issue in the CoGAN framework. This is a well-known problem in GAN literature, and [24, 23, 5] are works that aim to enable GANs to generate larger images through architectural changes. As such this issue could most likely be solved by introducing one of these methods.



a) horse2zebra b) apple2orange

Figure 3.5: Samples from CoGAN trained on 256x256 CycleGAN datasets

3.3.2 Domain issue

As we saw in the previous section, CoGAN does not perform very well when training on 256x256 images. Therefore, we scale down the horse2zebra and apple2orange datasets to 128x128, the same size as CelebA, and train a CoGAN model using the same generator and discriminator architectures as for CelebA (Table A.1 and A.2). The goal of this experiment is to understand whether CoGAN solely fails on these datasets due to the image size, or if there are other contributing factors. The models are trained for 35k iterations with the DCGAN hyperparameter settings. Samples from this experiment are shown on Figure 3.6a and Figure 3.7a.

Shapes that may indicate body and legs are now visible in the zebra samples, but it is more difficult to distinguish any horse features in the horse domain and we also see the same issues of non-correspondence as found on the 256x256 images persists. Additionally, the images in the zebra domain are very similar across different latent vectors, which could be a sign of the model mode collapsing. The same is seen to a lesser extent in the horse domain. The calculated LPIPS distances on Table 3.5, is indicating that there is a larger difference between generated horses and real samples from both domains, than what is seen of generated zebras. This is expected as the collapsed modes of the horse domain does not resemble neither horses nor zebras, while the mode collapsed zebra domain has some of the correct zebra styles and some shapes that could resemble horse/zebra-like features.

When looking at the apple2orange samples on Figure 3.7a we notice a slight improvement, especially, on the shapes and texture of oranges, and on the fact that some of the apples appear to have a form of stem. However, the image quality is definitely not up to par with that of models trained on CelebA. From the LPIPS

distances seen in Table 3.6 we also see these improvements across all scores, when comparing to those obtained by the 256x256 case reported in Table 3.4.

Gen./Real	Real horses	Real zebras
Gen. horses	0.8906	0.9594
Gen. zebras	0.6923	0.7024

Table 3.5: LPIPS distances between 128x128 horse2zebra dataset with CoGAN

Gen./Real	Real apples	Real oranges
Gen. apples	0.7345	0.7756
Gen. oranges	0.7402	0.7169

Table 3.6: LPIPS distances between 128x128 apple2orange dataset with CoGAN

Due to the observed mode collapse on the horse2zebra dataset, we train two CoGANs using the WGAN-GP loss described in section 2.5 on both horse2zebra and apple2orange. Due to the nature of this loss, we replace all batch normalisation layers in our discriminator with layer normalisation and use WGAN-GP hyperparameter settings mentioned in section 2.5. Using this loss quadrupled the training time due to the added calculations of the gradients of the interpolated images and we therefore only train the WGAN-GP models for 25k iterations.

Samples from these models are shown in Figure 3.6b and 3.7b. We notice a definite decrease in mode collapse, which impacts the LPIPS distances. On Table 3.7, we report new LPIPS distances on horse2zebra and without mode collapse, we now see the correct pattern in the distances. When comparing LPIPS distances without WGAN-GP (Table 3.5 and 3.6) to those with WGAN-GP (Table 3.7 and 3.8) we see a slight improvement in the similarity across all configurations, which could indicate that WGAN-GP improves the models. However, we still note that all produced images are of extremely low quality when considering the quality of images achieved in [34] on the CelebA dataset.

Since the downscaled CycleGAN datasets still proved difficult for CoGAN, despite the fact that CoGAN has achieved success on the 128x128 CelebA dataset, indicates that these datasets offer a different challenge, other than just image size. Later in this section we hypothesise what this challenge may be, but first we show another experiment to further illustrate the domain issue.

We experiment with the MNIST2SVHN dataset using the architectures previously shown in Table 3.1 and 3.2 and DCGAN hyperparameter settings. These models are trained for 25k iterations. The goal of this experiment is to investigate the weight sharing constraint and its effect on the capabilities of each GAN. We train four CoGANs which each have 4, 3, 2, and 1 shared layers between their generators.

a) Non-saturating GAN loss b) WGAN-GP loss

Figure 3.6: 128x128 CoGAN samples trained on horse2zebra

a) Non-saturating GAN loss b) WGAN-GP loss

Figure 3.7: 128x128 CoGAN samples trained on apple2orange

We here wish to find out whether gradually reducing the number of shared layers in the generator improves the individual GANs ability to capture the distributions of each dataset. If this is the case, we believe that the weight sharing constraint is restricting the generators and may be a key factor as to why CoGAN fails on these noisier datasets.

Figure 3.8 shows samples from the four models of this experiment. From these we confirm that the CoGAN does not learn a joint distribution on this dataset as reported in [16, 30]. Additionally we see that as the number of shared layers is decreased (figure A to D), the GAN responsible for the SVHN domain becomes better at learning its domain. According to the assumption that CoGANs can utilise high-level features between image domains, this implies that MNIST and SVHN do not share any high-level features. However, both datasets contain images of digits i.e. the same content and only differ in the styles in which the content is displayed.

Gen./Real	Real horses	Real zebras
Gen. horses	0.6676	0.7472
Gen. zebras	0.7101	0.6824

Table 3.7: LPIPS distances between 128x128 horse2zebra dataset with CoGAN: WGAN-GP

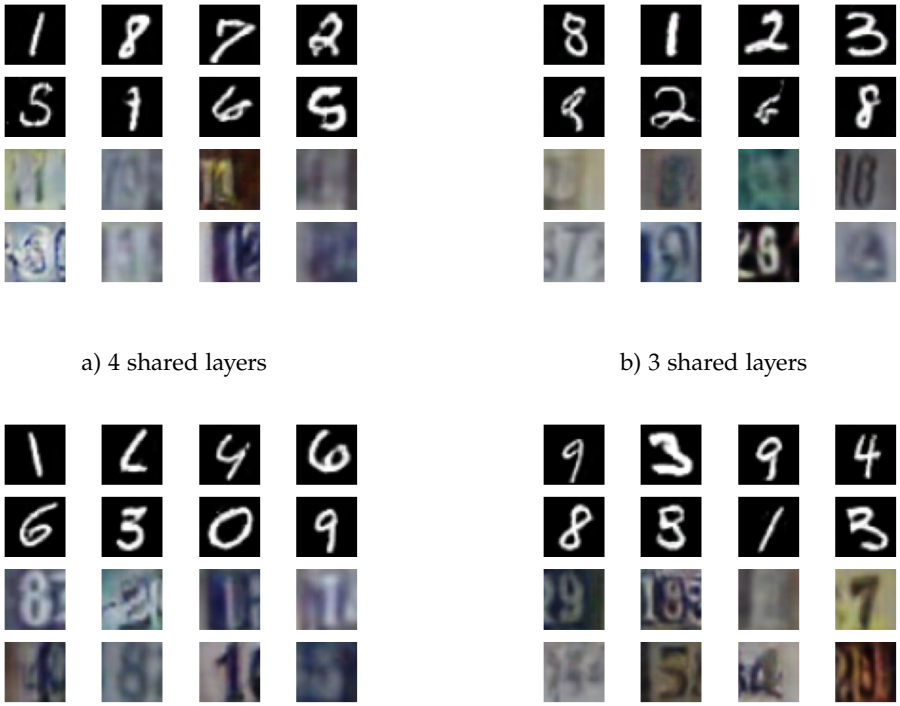
Gen./Real	Real apples	Real oranges
Gen. apples	0.6931	0.7414
Gen. oranges	0.7069	0.6968

Table 3.8: LPIPS distances between 128x128 apple2orange dataset with CoGAN: WGAN-GP

So, intuitively there should exist some kind of high-level features representing the shapes of the digit objects that CoGAN could learn. We examine how similar the generated pairs of images are through the LPIPS distance measures. We calculate the average distance between 50.000 generated MNIST images and 50.000 generated SVHN images for the configurations furthest apart, namely, 4 shared layers and 1 shared layer. The calculated average for 4 shared layers is 0.3541 and for 1 shared layer is 0.3423, which indicates that more similarity between generated samples is achieved when relaxing the weight constraint, however this relaxation is not enough to achieve proper joint distribution learning.

We believe the reason that CoGAN succeeds on MNIST variants and CelebA, and fails on datasets like horse2zebra, apple2orange, and MNIST2SVHN, is that the images in both domains of the MNIST variants and CelebA variants are very similar compared to those of horse2zebra, apple2orange, and MNIST2SVHN. By similar, we mean that the content of images in successful datasets is regularly placed (centred) and contains low noise.

In the case of the SVHN, the content(numbers) occurs in many different styles such as changing colours as well as the occurrence of multiple numbers on one image. As such it is not very similar to MNIST which has very little noise. For the CycleGAN datasets, the content occurs in many different contexts and is often not always positioned in the same spot across images. Additionally, multiple content objects may occur on each image as with SVHN. These are potential factors that cause these datasets to be more noisy and may therefore contribute to why CoGAN does not perform well on these datasets. As further support of our claims in this section, we refer to [44, 30, 16] where CoGAN exhibits similar issues.



a) 4 shared layers

b) 3 shared layers

c) 2 shared layers

d) 1 shared layer

Figure 3.8: Reproduced CoGAN results with our own implementation

Based on our discussion of the domain issue, as well as the observations in the aforementioned literature, we form the hypothesis that as the content or style, or both, of the domains, become less similar the weight sharing constraint may work more as a restriction on the generators such that they cannot fully learn their specific domain.

3.4 Problem statement

Through our examination of CoGAN, we identify two main issues with the framework:
The domain issue regarding the learning of joint distributions over domains without a certain degree of similar high-level features.
The scaling issue centred around the scaling of CoGANs to learn distributions of more complex data, i.e. larger images.

Since we are interested in working with issues specific to the CoGAN framework, we choose to work with the domain issue, where we are interested in the fact that weight sharing is restricting the generators, to a point where even the marginals cannot be learned on noisy datasets. This problem is focused on the fundamental approach in CoGAN, whereas the scaling issue is more based on general GAN improvements. We are also interested in whether WGAN-GP is necessary to prevent mode collapse when deviating from the CoGAN approach, with the goal of learning joint distributions over noisier datasets. With these choices and interests in mind, we state the following general problem statement for further working with the domain issue of CoGAN, along with some specific problems that we want to address in this project:

How can we address the domain issue experienced with CoGANs in a way that improves the models capability of learning joint distributions, while still only training on marginal datasets?

- How can we relax the weight sharing constraint that restricts the individual generators, in a way that increases the CoGANs capability of learning the joint distribution between image domains?
- How can we utilise the core idea of learning similar high-level features presented by CoGAN?
- Will we still need WGAN-GP, as previously seen, to avoid mode collapse on noisier datasets?

Chapter 4 Approaches

In this chapter, we first present an initial approach aimed at solving our problem statement. Through experimentation, we learn that this approach alone is not sufficient and therefore we present three additional methods that aim to improve the results seen from our initial approach. All presented methods are in the shape of regularisation terms, which we add to the loss of both generators. We disregard the need for replacing the weight sharing constraint between the discriminators, as they in [34] report little difference when altering the weight sharing of these. Implementation details of these proposed regularisation terms can be found in chapter 6.

4.1 Feature regularising

Our idea is to completely remove the weight sharing constraint and instead introduce a softer constraint, which provides more freedom for the generators' weights to deviate from each other. As such, we still wish to couple two GANs, but with a softer coupling. We call these GANs Soft-Coupled GAN (SCoGAN).

In [34], the goal is to generate pairs of images in two respective domains that share high-level features. They achieve this goal implicitly via the weight sharing constraint that they impose between the generators and discriminators. We are instead interested in achieving this goal by explicitly enforcing similar learned features between the generators. Therefore, we initially propose a feature regularisation(FR) term that is added to each of the generator losses, aimed at this explicit enforcing of similar features. The feature regularisation term measures similarity between intermediate outputs of both generators at select layers and is defined in the generator's losses as seen on Equation 4.1. Here, K is the number of shared layers, $G_1^k(\mathbf{z})$ is the intermediate feature map output of generator1 at layer k and $G_2^k(\mathbf{z})$ is the corresponding output of generator2. Finally, α is for weighting the term and by setting K, you can select a number of layers you wish to regularise between. These variables are considered tunable hyperparameters. This follows the intuition described in [34] that certain layers of neural networks are responsible for high-level feature generation and other layers are responsible for low-level feature generation.

$$
\begin{aligned}
L_{G_1}(G_1) &= \mathbb{E}_{\mathbf{z}\sim p_z} log(1 - D_1(G_1(\mathbf{z}))) + \alpha F_{diff} \\
L_{G_2}(G_2) &= \mathbb{E}_{\mathbf{z}\sim p_z} log(1 - D_2(G_2(\mathbf{z}))) + \alpha F_{diff} \\
\text{where } F_{diff}(G_1, G_2) &= \frac{1}{K}\sum_{k=1}^{K}(G_1^k(\mathbf{z}) - G_2^k(\mathbf{z}))^2
\end{aligned}
\tag{4.1}
$$

We hypothesise that the feature regulariser provides the network with more freedom to use its weights to learn the appropriate joint distribution since it does not put a restriction directly on the weights. However, a concern with the FR term is that it might create similar training dynamics, to what is seen when using weight sharing. Consider the fact that FR must be lower, when weights are more similar, which means that optimising for a low feature regulariser term could encourage similar weights. Because of this, we conduct an initial experiment with a SCoGAN that employs the FR term (SCoGAN-FR) and a CoGAN to examine the behaviour of generator weights during training.

For this experiment, we use SCoGAN with the same architecture as shown in Table 3.1 and 3.2, without any shared layers. From our effort on reproducing the CoGAN results on MNIST2edge, we found that after $\sim$ 3000 iterations the CoGAN had learned similar features and proper creation of the digits, therefore we choose to train for 4000 iterations. We use DCGAN hyperparameter settings, set the weight $\alpha = 10$, and use non-saturating GAN loss. For the feature regulariser, we set $K = 3$, meaning that we compare intermediate feature maps on the first 3 layers of the generators. Plots are shown on Figure 4.1, where we see that the weight difference, during training, increases much faster for the feature regulariser, than for CoGAN. From this, we confirm that SCoGAN-FR and CoGAN do not have similar training behaviour.

Next, we choose to conduct an experiment on SCoGAN-FR to see whether it is able to learn a joint distribution like CoGAN. This experiment uses the same configurations as the weight difference experiment above. We train two different versions of this SCoGAN architecture on the MNIST2edge dataset. The first version is a SCoGAN without any regularising terms and acts as a baseline, which should show no correlation between samples from either domain. The second version uses the feature regularising term in the loss function of each generator.

On Figure 4.2a, we see samples from the baseline model at 4000 iterations. It is clear that the model only learns the marginal distributions as expected. On 4.2b, the feature regulariser proves to be efficient at enforcing similar generated features between the two models, and therefore is learning the joint distribution.

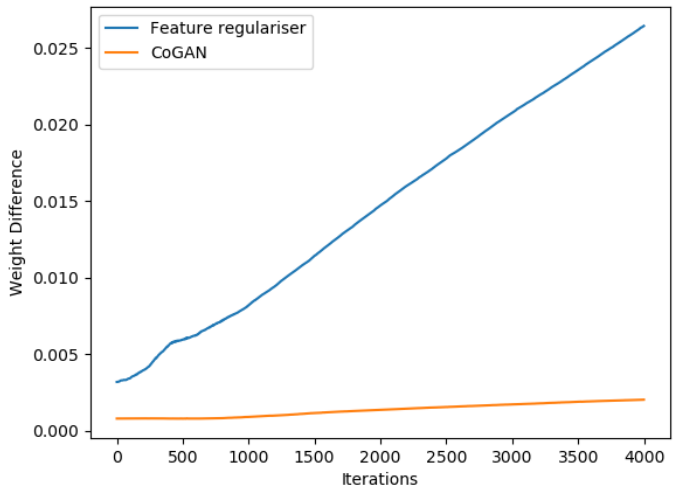
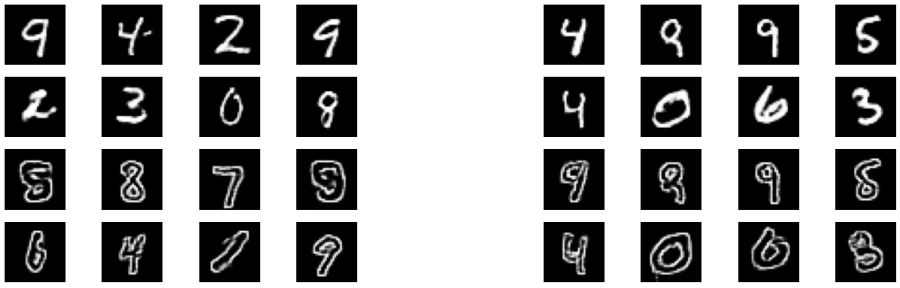


Figure 4.1: Measured weight difference during training for SCoGAN with feature regulariser, and CoGAN

Since SCoGAN-FR achieves comparable results to CoGAN on the low noise and clear object dataset of MNIST2edge, we decide to experiment further with the MNIST2SVHN dataset. The same architectures, hyperparameters, and loss function as the previous MNIST2edge experiment are used. The model is trained for 20000 iterations, as the joint distribution of this dataset is seemingly harder to learn. On Figure 4.3a, we see that SCoGAN-FR is not able to learn the joint distribution with $\alpha = 10$. Therefore, we attempt to increase the weight $\alpha = 100$, to emphasise the importance of learning similar features.

Samples from this are seen on Figure 4.3b, and it clearly shows that this enforces similar high-level features, but we lose the ability to generate SVHN style. Examining the discriminator loss for these experiments, as seen on Figure 4.3a&b, revealed that the discriminator is unstable when the feature regulariser term is weighted this high. We tested further with weight values(10,50,75) but found that a balance between learning similar high-level features and learning the style of SVHN was difficult to achieve without causing the SVHN discriminator to become unstable.

Due to this instability, we introduce the WGAN-GP loss as presented in section 2.5. We replace the non-saturating loss with the WGAN loss and employ gradient penalty with $\lambda = 10$ and perform 5 discriminator updates, on each discriminator, before updating the generators. We use the same architecture and hyperparameter configuration as previously. On Figure 4.4a, we see samples from this



a) SCoGAN - 4000 iterations b) SCoGAN-FR - 4000 iterations

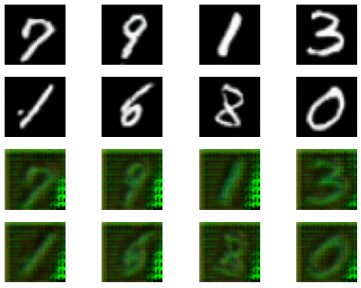
Figure 4.2: Samples from feature regulariser experiments

model, where SCoGAN-FR is vastly improved by the WGAN-GP loss and is now vaguely capable of learning a joint distribution, albeit with some flaws. The top rows of the MNIST2SVHN samples contain similar content, while the bottom rows are different. Observing how samples from the same latent vector seed developed during training, we notice that the features of the top row images change accordingly such that corresponding digits are shown[1]. We notice that the FR introduces SVHN styling onto the MNIST samples, which is especially seen on the third sample of the second row in 4.4a. This is likely due to a balancing issue between the weight and the number of layers considered by the feature regulariser. From the results, we would like to increase the weight even further to capture more similarity between content features, but this will most likely lead to enforcing additional styling features. To avoid this, we attempt to reduce the layers that are part of the feature regulariser term from $K = 3$ to $K = 2$. This targets even higher level content features and might avoid the low-level style features. The result of this can be seen on Figure 4.4b, and it is clear that this does not contribute to a better learning of the joint distribution, as there are no correlations between MNIST and SVHN samples.

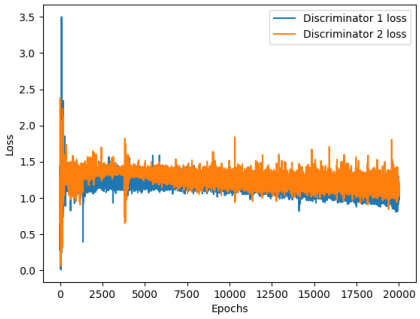
From the results obtained with the feature regulariser approach, we are confident that a solution focusing on the similarity between features, is achievable. However, the feature regulariser is only able to capture the joint distribution of MNIST2SVHN to some degree, and therefore we look further into other ways of controlling and enforcing feature similarity that either supports or replaces the feature regulariser in the training process.

[1]https://github.com/palminde/P10Project

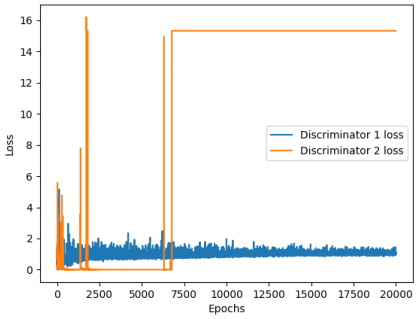
a) SCoGAN-FR $\alpha = 10$



b) SCoGAN-FR $\alpha = 100$



c) Discriminator loss for $\alpha = 10$



d) Discriminator loss for $\alpha = 100$

Figure 4.3: SCoGAN-FR on MNIST2SVHN dataset

4.2 Controlling learned features

As we are focusing on controlling the learned features between the two GANs, we propose two methods to be used in conjunction with the feature regulariser. These are: (1) a semantic loss which enforces that pairs of generated images should be classified identically by an auxiliary classifier, (2) a cycle consistency loss between latent vectors which indirectly enforces feature similarity such that generated images pairs can be encoded back to the latent vector that was used to generate them. Additionally we propose a more advanced alternative to the feature regulariser called the perceptual loss. This term directly enforces content similarity between generated pairs of images and style similarity between generated and real images. To achieve this we use a pretrained classifier as a feature extractor and compares features from multiple intermediate layers.

a) SCoGAN-FR $\alpha = 100$ and $K = 3$ b) SCoGAN-FR $\alpha = 100$ and $K = 2$

Figure 4.4: SCoGAN-FR with WGAN-GP loss on MNIST2SVHN

4.2.1 Semantic loss

Our semantic loss is an altered version of the semantic loss presented in [16]. Here they provide their model with a source image that must be transformed into the style of a target image. Their semantic loss enforces that the transformed source image and the original source image must be classified identically by a classifier pretrained on the source domain with the goal of retaining the content features through the style transformation.

To adapt this to our case in which we have no source image, we compare classifications of generated image pairs, as shown in Figure 4.5, ensuring that for the same latent vector both generators produce images that can be classified identically. The classifier should preferably be trained on a mixed dataset of both domains to achieve the best possible classifications to guide the two GANs toward generating similar images. However if the two domains have similar high-level features, such as MNIST2edge, a classifier trained on one domain may be sufficient.

Our semantic loss does not enforce similar features but rather learns to map specific explicit classes in different domains. As an example, when this approach is used on MNIST2edge the classifier is trained to classify MNIST and MNISTedge images, and thus this regularisation term pushes towards a mapping between ones, twos, threes, and so on. As such for a given latent vector both generators produce ones, but as similar features are not enforced, digits generated from the same latent vector may have, for example, different orientation in the two domains. Because of this, the semantic loss term can ideally be coupled with the feature regulariser, to aid with producing paired content, such that the feature regulariser is more focused on making the orientation/style/etc. of the content identical.

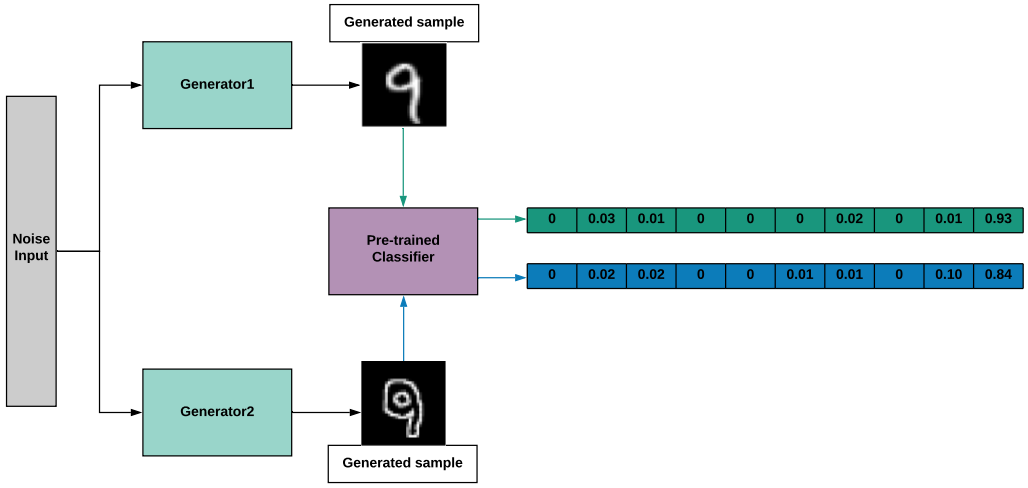


Figure 4.5: Generator setup with semantic loss

Equation 4.2.1 shows the generator loss when employing the semantic loss as a regularisation term. Here, we pretrain δ as a classifier network for each dataset case that we experiment on. Besides the added semantic loss term, we add α as a tuneable weight.

$$\begin{aligned} L_{G_1} &= \mathbb{E}_{\mathbf{z}\sim p_z} log(1 - D1(G_1(\mathbf{z}))) + \alpha L_{sem} \\ L_{G_2} &= \mathbb{E}_{\mathbf{z}\sim p_z} log(1 - D2(G_2(\mathbf{z}))) + \alpha L_{sem} \\ \text{where } L_{sem} &= \frac{(\delta(G_1(\mathbf{z})) - \delta(G_2(\mathbf{z})))^2}{2} \end{aligned} \tag{4.2}$$

The main restriction with this approach is that it requires to pretrain a classifier that is able to classify at least one of the two domains. As such, atleast one dataset must be labelled and each dataset must contain the same amount of classes so that the classifier can be taught to match corresponding classes in each domain. This restricts the types of datasets that can be used. An example of a dataset that abides by these points is MNIST2SVHN. Both datasets(MNIST and SVHN), consist of the same type of objects and therefore a single classifier can be trained on either one of the datasets or a mixture of both. An example of a dataset that does not work in this setting is horse2zebra, since each domain consists only of one class, either horse or zebra and a classifier on these datasets therefore would not help to guide the type of objects to generate.

4.2.2 Cycle consistency

This proposal is inspired by the main contribution of CycleGAN [44], namely the cycle-consistency loss (CCL). Intuitively, the CCL enforces that the stylised output image of one GAN can be reversed back to its original style through another trained GAN. This enforces that the two GANs generate the same content in different styles such that the image can be reversed [44, 7]. These two GANs are trained together to learn this cyclic behaviour. CCL has been adopted in many conditional style transfer models [16, 7, 30], where style transferring back and forth between two domains is desirable. Because of its popularity within this field and its relation to joint distribution learning, we are interested in the application of this loss for the SCoGAN architecture in conjunction with both our semantic loss and our feature regulariser loss.

Our proposal is a slightly different interpretation of cycle consistency. Rather than measuring the reconstruction loss between original and stylised images we measure reconstruction between latent vectors as SCoGAN does have any input image to compare to. On Figure 4.6, we illustrate the SCoGAN generator architecture for generating images with the cycle consistency loss. Consider the generation of an image $\mathbf{x} = G1(\mathbf{z})$ and an image $y = G2(\mathbf{z})$. We train an encoding network F, alongside $G1$ and $G2$, which learns to encode generated images back to the initial latent vector, $\mathbf{z}$, by minimising the loss $L_{enc}(F, h, \mathbf{z}) = ||F(h) - \mathbf{z}||_2^2$. Here, h is either $\mathbf{x}$ or $\mathbf{y}$. This encoding network is used in the cycle consistency loss for the SCoGAN architecture seen as part of the generator losses at Equation 4.3. Here, the CCL L_{cyc} enforces that both generators generate images that can be encoded back to the latent vector that originally generated them. This means that both images must contain some shared information that allows the encoder to create the same latent vector again.

$$L_{G1}(G_1, G_2, F) = \mathbb{E}_{\mathbf{z}\sim p_z} log(D_1(G_1(\mathbf{z}))) + \alpha L_{cyc} \tag{4.3}$$

$$L_{G2}(G_1, G_2, F) = \mathbb{E}_{\mathbf{z}\sim p_z} log(D_2(G_2(\mathbf{z}))) + \alpha L_{cyc} \tag{4.4}$$

$$\text{where } L_{cyc}(G_1, G_2) = \mathbb{E}_{\mathbf{z}\sim p_z} ||F(G_1(\mathbf{z})) - \mathbf{z}||_2^2 + ||F(G_2(\mathbf{z})) - \mathbf{z}||_2^2$$

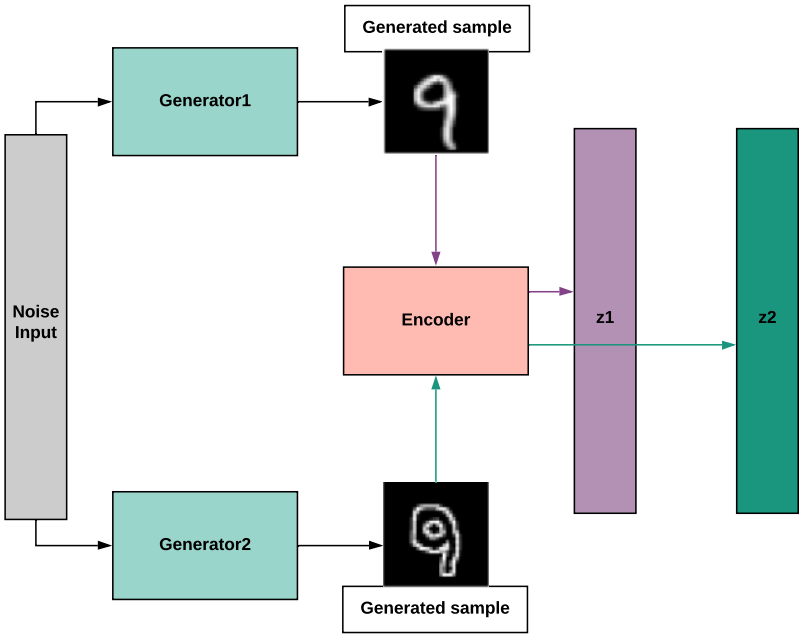


Figure 4.6: Generator setup with cycle concistency loss

4.2.3 Perceptual loss

Our final proposition on how to solve our problem statement is a more advanced approach to feature regularisation, which uses an altered version of the perceptual loss introduced in [21]. The perceptual loss also relies on an auxiliary network, but here it is used as a deep feature extraction network, instead of a classifier. The perceptual loss in [21] consists of two terms, namely the Feature Reconstruction Loss(FRL) and the Style Reconstruction Loss(SRL). The intuition behind this is the same as used in CoGAN: neural network classifiers extract high-level features in later layers and low-level features in the early layers. Thus, the FRL measures high-level feature similarity between images by comparing feature maps extracted from the auxiliary net at a later layer, and SRL measures low-level feature similarity by extracting feature maps at multiple intermediate layers. Originally, this approach is used in a style transfer model, where a network is trained to transform the style of an input image, to match the style of the target image. However as SCoGAN is not a conditional model we utilise another scheme.

We employ the FRL term between features extracted from the two generated images as to enforce the generated images to share similar high-level features. We use the SRL term between generated and real images in the respective domains to aid the generators in maintaining the correct style in each domain. This task is

partly shared with the adversarial loss which ensures that the generated images match images in the target domain in order to fool the discriminator. However, the discriminator is concerned with a full view of the image in order to identify real/fake samples, and therefore focuses on a mixture of features containing both content and style. The feature extraction is illustrated on Figure 4.7. Here, we extract both style and feature maps with the pretrained VGG19 network from the generated images, which is indicated by the green arrows. While also extracting style maps for the real target images indicated by the red arrows.

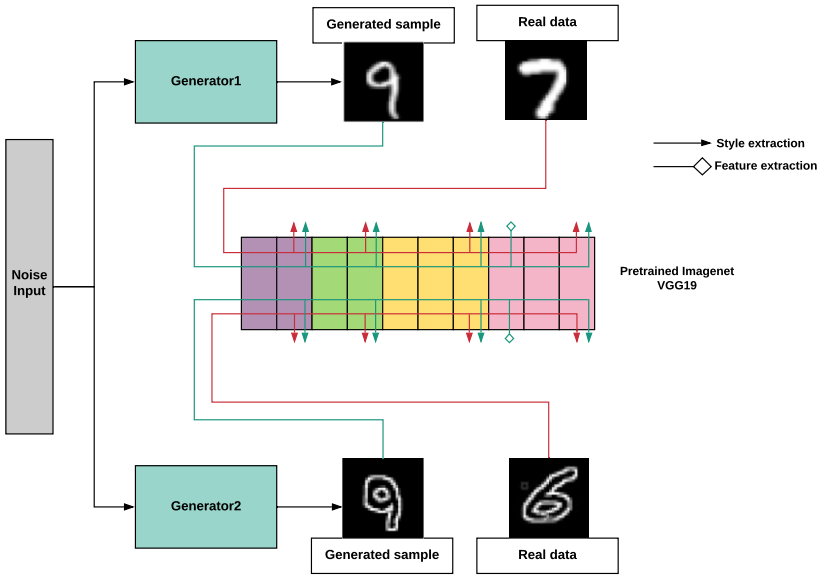


Figure 4.7: Generator setup with perceptuel loss

These alterations result in the new loss functions for the generators seen on Equation 4.5. Here, $\mathbf{x}_1 \sim p_{data1}$ and $\mathbf{x}_2 \sim p_{data2}$ where p_{data1} and p_{data2} are distributions over the two domains respectively. δ_j is a pretrained VGG16 network that provides the output of layer j and $H_j W_j C_j$ are the height, width, and channel dimensions of this output. In the SRL term, $Gram_j$ is a $C_j \times C_j$ gram matrix, where each element (c, c') of the matrix is calculated as $Gram_j(\mathbf{k})_{c,c'} = \frac{1}{H_j W_j C_j} \sum_{h=1}^{H_j} \sum_{w=1}^{W_j} \delta_j(\mathbf{k})_{h,w,c} \delta_j(\mathbf{k})_{h,w,c'}$. Finally, we add weighting terms as before via the α and β variables.

$$
\begin{aligned}
L_{G1} &= \mathbb{E}_{\mathbf{z}\sim p_z} log(D_1(G_1(\mathbf{z}))) + \alpha FRL(G_1(\mathbf{z}), G_2(\mathbf{z})) + \mathbb{E}_{\mathbf{x}_1\sim p_{data1}} \beta SRL(G_1(\mathbf{z}), \mathbf{x}_1) \\
L_{G2} &= \mathbb{E}_{\mathbf{z}\sim p_z} log(D_2(G_2(\mathbf{z}))) + \alpha FRL(G_2(\mathbf{z}), G_1(\mathbf{z})) + \mathbb{E}_{\mathbf{x}_2\sim p_{data2}} \beta SRL(G_2(\mathbf{z}), \mathbf{x}_2) \\
&\text{where } FRL(\mathbf{x}, \mathbf{x}') = \frac{(\delta_j(\mathbf{x}') - \delta_j(\mathbf{x}))^2}{H_j W_j C_j} \\
&\text{and } SRL(\mathbf{x}, \mathbf{x}') = (Gram_j(\mathbf{x}') - Gram_j(\mathbf{x}))^2
\end{aligned} \tag{4.5}
$$

In contrast to the semantic loss combination, this method is more general and is applicable to any combination of datasets as it does not require labelled data. However we note that the quality of the features may vary depending on the pretrained feature extractor.

Chapter 5 Experiments

In this chapter, we conduct and present the results of experiments with our proposed solutions. The chapter is split into four sections, one for each dataset, and within these sections, we present the samples for visual inspection and quantitative evaluations of each approach on the given dataset. All experiments are conducted on 1x NVIDIA Tesla V-100 GPU.

We quantify the performance of our models using the LPIPS-distance presented in section 3.2. For each dataset, we calculate the LPIPS distance between the real samples of both domains as a baseline for how similar these two datasets are. For each experiment, we report three different measurements of LPIPS distances. First, we measure LPIPS between generated samples from both domains, which has a correspondence to how well high-level features are shared between the generators. This is compared to the baseline LPIPS for the given dataset and should be lower, as comparing pairs of images that, if the joint distribution has been correctly learned, share similar high-level features. Even so, the score must not be too low, as this would indicate that the images are too similar, meaning that no domain-specific features have been learned. Providing a limit for this is difficult and therefore we rely on visual inspection of samples to fully interpret this score. The additional samples we use for visual inspection as well as samples taken during training of different SCoGAN models can be found here `https://github.com/palminde/P10Project`.

The second and third measurements compare batches of generated and real samples in the respective domains, e.g. generated MNIST images with real MNIST images. These distances correspond with each GANs ability to generate images that retain features from its target domain. For these two scores lower is better as a lower score indicates more similarity between the target domain and the generated samples. We calculate these three measurements for both SCoGAN and CoGAN models to have a more objective comparison method between models than visual inspection.

Before presenting the experiments, we would like to state that we found the perceptual loss at a later stage of the project and therefore did not have time to perform any tuning of hyperparameters or experiment with which layers in the pretrained net are optimal to extract features from, on the different datasets. Training with the perceptual loss requires the setting of the weight α_{PL} and β_{PL} for the FRL and SRL term respectively. In the official implementation of [21] $\alpha_{PL} = 1$ and $\beta_{PL} = 1250$ which produces huge values that are disproportionate to our adversarial loss term. We therefore decide to use the weight values $\alpha_{PL} = 0.001$ and $\beta_{PL} = 0.00000000001$

such that they correspond better with our adversarial loss. These values were found after a very small amount of initial testing and we discuss the implications of this choice and how we would have approached this problem if we had had more time. The experiments performed with this proposal, therefore, shows only a rough proof of concept, rather than a fully fletched solution. However, as it alleviates the main issue with the semantic loss, as discussed in subsection 4.2.3, we deemed it important to include. For all perceptual loss experiments, we use a VGG19 network pretrained on the Imagenet dataset [1] as the feature extractor.

5.1 MNIST2edge

In [34] experiments are conducted on several different alterations of the MNIST dataset. We choose to only include one of these alterations in our experiments, namely MNIST2edge, because of the simplicity of these datasets. As seen in chapter 4 SCoGAN-FR can sufficiently learn the joint distribution between the two domains, however, we are interested in whether our three remaining proposals are likewise able to create a soft coupling between the GANs and produce similar results.

All experiments in this section are conducted with the architectures shown in Table 3.1 and 3.2 without any shared layers. We follow optimiser, learning rate, input normalisation and batch size recommendations for DCGAN training as presented in section 2.6. Weights are initialised using Xavier initialisation and the latent vector consists of 100 values sampled from a uniform distribution with $min = -1$ and $max = 1$. Lastly, we use weight decay with a decay rate of 0.0005. These settings mirror those used in the official Pytorch implementation of CoGAN for training on MNIST2edge. If not otherwise specified, all models are trained with the WGAN-GP loss with weight $\lambda = 10$ for the gradient penalty weight. All models are trained for 20.000 iterations.

LPIPS distances for models trained on this dataset are measured as the average distance between 30.000 pairs of images. This amount is chosen as it is the total size of MNISTedge. The baseline average LPIPS distance between real MNIST and MNISTedge samples is 0.1627. Table 5.1 summarises all LPIPS scores gained on the MNISTedge dataset using CoGAN and SCoGAN with feature regulariser(FR), semantic loss(SL), cycle consistency(CC), and perceptual loss(PL). In order to provide a better understanding of these scores and put them into perspective, we see some examples on Figure 5.2. These are the lowest(a) and highest(b) scoring pairs generated by the baseline CoGAN architecture on the MNIST2edge dataset. In

[1] `https://www.tensorflow.org/api_docs/python/tf/keras/applications/VGG19`

this case, even the highest scoring pair is quite similar, most likely due to the fact that the second domain(MNISTedge) is created from the first domain(MNIST) and therefore generated pairs inherently has many similar features.

a) LPIPS Score = 0.0183 b) LPIPS Score = 0.2937

Figure 5.1: Samples generated from CoGAN model trained on MNIST2edge. Leftmost image in each pair belongs to MNIST domain and rightmost belongs to MNISTedge

Configuration	CoGAN	FR	SL	CC	PL
Gen. MNIST vs Gen. edge	0.1026	0.1227	0.1039	0.1092	0.1019
Gen. MNIST vs Real MNIST	0.1157	0.1217	0.1141	0.1148	0.1110
Gen. edge vs Real edge	0.1157	0.1429	0.1100	0.1092	0.1064

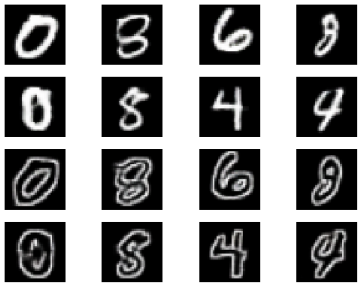
Table 5.1: All LPIPS scores obtained on the MNIST2edge dataset

Semantic Loss

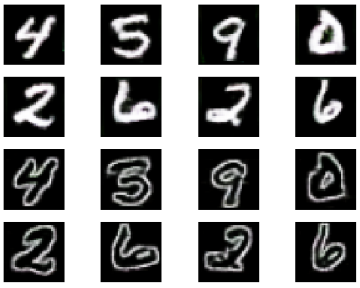
As a prerequisite, we train a classifier whose architecture is seen in Table B.1. The classifier is trained on the standard MNIST dataset for 10 epochs, reaching an accuracy of $\sim$ 98% using a categorical cross entropy loss. We use this classifier when training the SCoGAN model with the semantic loss term (SCoGAN-SL). We set the weight $\alpha_{SL} = 10$. On Figure 5.2a, we show samples of images from this model. Based on visual inspection, the semantic loss term does learn the joint distribution over these domains. This is further supported by the measured LPIPS distances shown in Table 5.1, which are very close to those measured on the CoGAN architecture and surpasses the baseline MNIST2edge score.

Cycle Consistency

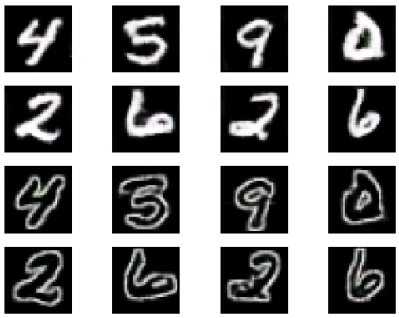
In this experiment, we train an SCoGAN model that is only regularised by cycle consistency (SCoGAN-CC). The weight of the CC term is set to 10. Samples from this model, seen on Figure 5.2b, show that SCoGAN-CC is equally good at learning the joint distribution and this is again supported by the LPIPS distances on Table 5.1.



a) SCoGAN-SL



b) SCoGAN-CC



c) SCoGAN-PL

Figure 5.2: Samples from SCoGAN-SL, SCoGAN-CC and SCoGAN-PL on MNIST2edge

Perceptual Loss

Training an SCoGAN using the perceptual loss (SCoGAN-PL) we see on Figure 5.2c that this model is equally capable of learning the joint distribution as our previously proposed approaches. This is again supported by the LPIPS measures as seen in Table 5.1.

Summary

These experiments confirm that all our approaches enforce learning of similar features between generators and can successfully learn the joint distribution over two low noise domains. All methods surpass the baseline LPIPS distance between MNIST and MNISTedge, thus, quantitatively, showing that generated samples generally do resemble each other and similarly low LPIPs scores are achieved for all

approaches when comparing generated and real images. Meaning that each separate GAN has managed to learn its respective domain.

5.2 MNIST2SVHN

In these experiments, we are interested in whether semantic loss, feature regularising, and cycle consistency can be combined in meaningful ways to solve the issues we see with the CoGAN and SCoGAN-FL on MNIST2SVHN in chapter 3 and 4. Unless explicitly specified in the following subsections, all experiments in this section are conducted with the same architecture, loss, and hyperparameter settings as described in section 5.1. Each model is trained for 25.000 iterations.

The average LPIPS distance is measured between 50.000 image pairs, and the baseline average LPIPS distance on MNIST2SVHN is 0.3304. Table 5.2 summarise all LPIPS distances of the models trained on MNIST2SVHN. We note that the generated MNIST vs real MNIST measure is comparable to the ones achieved on MNIST2edge, which is expected as the MNIST generator of all models produces compelling MNIST images. We also see that there is less similarity between generated images compared to the MNIST2edge case, which is also expected as the baseline LPIPS distance is larger for MNIST2SVHN. Finally, we note that CoGAN has a higher dissimilarity between generated images(0.3400) than our SCoGAN with the feature regulariser(0.2434). Again, this is expected as SCoGAN-FR produces MNIST images with SVHN styling, resulting in more similarity between the generated samples. Therefore, we hypothesise that a good model produces a generator vs generator LPIPS score in the range of $[0.2434, 0.3400]$ for this dataset.

Figure 5.3 shows pairs of images with low(a) and high(b) LPIPS score. These samples are taken from an SCoGAN with semantic loss and feature regulariser(SCoGAN-SLFR) trained on the MNIST2SVHN dataset. Here, it is easier to see the difference between high and low LPIPS scores than on the MNIST2edge dataset.

a) LPIPS Score = 0.0919 b) LPIPS Score = 0.6342

Figure 5.3: Samples generated from an SCoGAN-SLFR model. Leftmost image in each pair belongs to MNIST domain and rightmost belongs to SVHN

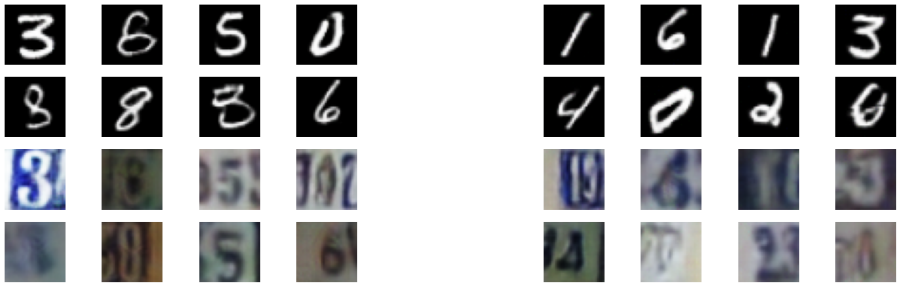
Configuration	CoGAN	FR	SLFR	SLCC	FRCC	PL
Gen. MNIST vs Gen. SVHN	0.3400	0.2434	0.3016	0.3093	0.3022	0.2948
Gen. MNIST vs Real MNIST	0.1126	0.1194	0.1162	0.1153	0.1146	0.1120
Gen. SVHN vs Real SVHN	0.2151	0.1941	0.2046	0.2083	0.2094	0.2117

Table 5.2: All LPIPS scores obtained on the MNIST2SVHN dataset

Semantic Loss and Feature Regulariser

We train a SCoGAN model with both the semantic loss and the feature regulariser (SCoGAN-SLFR), where $\alpha_{SL} = 10$ and $\alpha_{FR} = 100$. The classifier used for the semantic loss is trained on a mixed dataset containing both MNIST and SVHN images and uses the architecture found at Table B.1. It is trained for 10 epochs, reaching an accuracy of ∼92%. Samples from the SCoGAN-SLFR model can be seen on Figure 5.4a and shows visible similarities between high-level features in most pairs, which is an improvement over both the feature regulariser and over CoGAN. Furthermore, we see styles that match those of MNIST and SVHN in their respective samples.

The LPIPS distance between generated and real samples does not deviate a lot from what CoGAN and the feature regulariser achieved. However, the distance between generated samples from both domains is lower and actually lies within the range of the baseline and feature regulariser score indicating that the generators have managed to learn the shared high-level features of the two domains.



a) $\alpha_{SL} = 10$ and $\alpha_{FR} = 100$ b) $\alpha_{SL} = 10$ and $\alpha_{CC} = 10$

Figure 5.4: Samples from SCoGAN-SLFR (a) and SCoGAN-SLCC (b) on MNIST2SVHN

Semantic Loss and Cycle Consistency

We now train an SCoGAN model with the cycle consistency and semantic loss (SCoGAN-SLCC). We use the configuration described at the beginning of this section. We use standard values for the weights $\alpha_{CC} = 10$ and $\alpha_{SL} = 10$ and showcase samples from the trained model on Figure 5.4b. Here, we also see similarities between image pairs, and when comparing LPIPS scores the SCoGAN-SLCC is almost identical to the SCoGAN-SLCC. The reason that we see good results here, may be that the cycle consistency loss indirectly pushes toward feature similarity, as the feature regulariser does, by enforcing that both images can be encoded into the same latent vector. As such, it works well with the semantic loss since they enforce different aspects that are important to learn the joint distribution, namely similar features (CC) and similar content (SL).

Feature Regulariser and Cycle Consistency

We also train an SCoGAN model with the feature regulariser and the cycle consistency terms (SCoGAN-FRCC). The configuration used is the same as previously and the weights are set as $\alpha_{FR} = 100$ and $\alpha_{CC} = 10$. Figure 5.5a shows samples from this model. According to the LPIPS score, this combination of regularisation terms performs just as well as the other two, however through visual inspection of samples we see that this combination is unable to learn the joint distribution. This is a clear example of why the LPIPS score should be used in conjunction with visual inspection.

A possible explanation for these results is that both terms enforce feature similarity - the feature regulariser does it directly and cycle consistency does it indirectly - as such, this combination is missing a directed content element such as the one provided by the semantic loss. We discuss this further in section 7.3. As we see poor results from this combination and with the fact that the two previous combinations have yielded decent results, we choose to discontinue this combination and conduct no further experiments with it.

Perceptual Loss

Finally, we train a SCoGAN-PL model using the same configuration as previously with weights $\alpha_{PL} = 0.001$ and $\beta_{PL} = 0.00000000001$. Resulting samples are shown on Figure 5.5b, where we see similar behaviour to what is achieved by the feature regulariser in section 4.1. There is similarity between some of the digits such as

a) $\alpha_{FR} = 100$ and $\alpha_{CC} = 10$ b) $\alpha_{PL} = 0.001$ and $\beta_{PL} = 0.00000000001$

Figure 5.5: Samples from SCoGAN-FRCC (a) and SCoGAN-PL (b) on MNIST2SVHN

5, 9, and 0, and from our visual inspection, these appear clearer than previously generated MNISTSVHN pairs by our other approaches. This could indicate that targeting style and content separately can result in a clearer generation of content in the correct styling. We note that this is just speculation, but it could be an interesting subject to investigate as part of future work. The LPIPS metrics can be seen in Table 5.2, and are close to the other proposals.

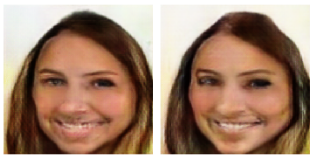
5.3 CelebA

We train and evaluate SCoGAN-SLFR, SCoGAN-SLCC, and SCoGAN-PL models on the CelebSmiling dataset, with the goal of learning the joint distribution over face images with different features. For these experiments, we use the architectures shown in Table A.1 and A.2 with no shared layers. We copy the hyperparameter settings from the CelebA experiments performed in [34], namely L2 weight decay with a decay rate of 0.0001, weights being initialised from a normal distribution with mean 0, and variance 0.02, the latent vector is of size 100 and is sampled from a uniform distribution with range $[-1, 1]$. Deviating from CoGANs configuration, we use the WGAN-GP loss with the recommend parameters proposed in [13] as mentioned in section 2.5. We train each model for 35.000 iterations.

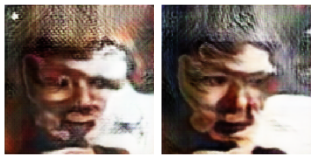
The pretrained classifier used for the semantic loss is now trained as a multi-label classification problem, rather than a multi-class classification problem. This is due to the fact that CelebA images do not belong to a single labelled class as is the case for MNIST and SVHN images. Instead, each CelebA image has 40 binary attributes describing features on the image which can be used as labels. As such, one image can have multiple labels. We choose to not use all 40 attributes as la-

bels since some are severely under-represented. We, therefore, select the labels that are represented by a minimum of 50.000 images. This results in the following 13 which are: Arched_Eyebrows, Attractive, Heavy_Makeup, High_Cheekbones, Male, Mouth_Slightly_Open, No_Beard, Oval_Face, Pointy_Nose, Smiling, Wavy_Hair, Wearing_Lipstick and Young. These labels cover the entire dataset and this selection, therefore, does not reduce the size of the dataset. Training a classifier for multi-label classification only requires slight alterations from training a multi-class classifier. We use the same architecture as for previously described semantic loss classifiers, but the softmax activation in the last layer must be replaced with a sigmoid activation to get a probability of every single label and not a probability vector over all labels and then we use the binary cross-entropy loss. The classifier is trained for 10 epochs achieving $\sim$87% accuracy.

On this dataset, we calculate the average LPIPS distance between 50.000 image pairs and the baseline dataset measure is 0.3955. Table 5.3 summarises all LPIPS scores obtained on this dataset. We note that the difference between image pairs generate by CoGAN is low, which is expected as our implementation of the CoGAN model generated very similar images with both generators. However we believe that the optimal LPIPS score on the CelebSmiling dataset is even lower than this. This is due to the fact that the perceptual difference should be lower, when the style difference between the two domains is smaller. Consider the example image on Figure 5.6a, where LPIPS is 0.0437 and here head shape, eyes, colourisation, and mouths are slightly different. So the score should be lower if this slight difference only applied to the mouths on the images. Figure 5.6 shows both of the samples from a trained SCoGAN-SLFR model with low(a) and high(b) LPIPS scores.



a) LPIPS Score = 0.0437



b) LPIPS Score = 0.3988

Figure 5.6: Samples generated from an SCoGAN-SLFR model trained on CelebSmiling. Leftmost image in each pair belongs to No-Smile domain and rightmost belongs to Smile

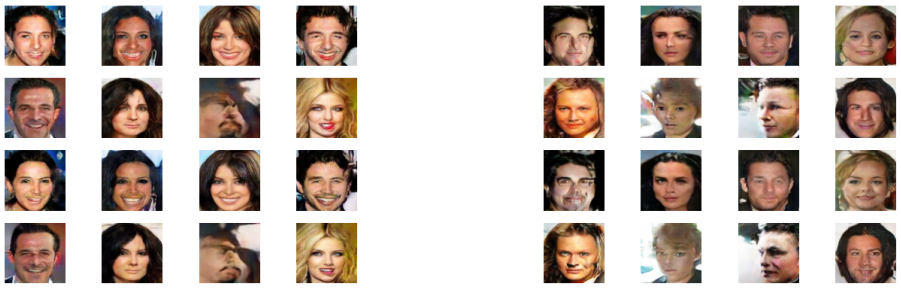
Configuration	CoGAN	SLFR	SLCC	PL
Gen. No-smile vs Gen. Smile	0.0433	0.1294	0.1708	0.0857
Gen. No-smile vs Real No-smile	0.3923	0.3875	0.3879	0.3907
Gen. Smile vs Real Smile	0.3877	0.3921	0.3881	0.3962

Table 5.3: All LPIPS scores obtained on the CelebSmiling dataset

Semantic Loss and Feature Regulariser

We train an SCoGAN-SLFR model with the aforementioned configuration and regulariser weights set to $\alpha_{SL} = 10$ and $\alpha_{FR} = 100$. Samples from the trained model can be seen in Figure 5.7a. As with our CoGAN models, we see no joint distribution learning, rather the two models have learned to make almost identical image pairs. Even though samples are similar to those presented by CoGAN, we see a larger difference between generated images in the LPIPS scores reported in Table 5.3. This indicates that SCoGAN-SLFR generates images that differs too much from each other, as the models should achieve low LPIPS scores on CelebSmiling.

This might be due to the way that the semantic loss enforces the mapping to the multi-label classification problem of CelebA. Consider that we have 13 attributes to express the complex features of a human face. It is easy to imagine that two completely different faces could be expressed by the same 13 binary attributes. If the mapping is perfect on all 13 attributes it is the job of the feature regulariser to then match exact styles of these attributes, which intuitively is much more complicated than the same task for MNIST2SVHN, where only the style of the single attribute should be matched. If the mapping is imperfect this tasks becomes even more difficult for the feature regulariser term.



a) $\alpha_{SL} = 10$ and $\alpha_{FR} = 100$ b) $\alpha_{SL} = 10$, $\alpha_{cc} = 10$

Figure 5.7: Samples from SCoGAN-SLFR (a) and SCoGAN-SLCC (b) on CelebSmiling

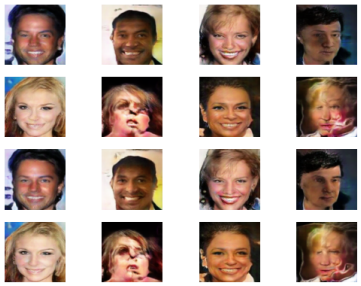
Semantic Loss and Cycle Consistency

We train an SCoGAN-SLCC model with weights set to $\alpha_{SL} = 10$, $\alpha_{cc} = 10$. Samples are shown in Figure 5.7b. Visual inspection of samples from this model shows that this combination of regulariser terms also fail to learn the joint distribution, and also seems to produce worse CelebA images. This model also sees the largest difference between samples from the two domains according to the LPIPS score which, following the logic of the previous experiment with SLFR, indicates that the indirect feature regularising of CC is even worse at handling this difficult mapping case that semantic loss presents.

Perceptual Loss

This experiment uses weights $\alpha = 0.001$ and $\beta = 0.00000000001$ and the configuration of architecture and hyperparameters mentioned at the beginning of this section. This model is trained on all three CelebA variants. Samples from all three models are shown on Figure 5.8. The SCoGAN-PL model does not successfully learn the joint distribution over any of the CelebA variants. However, from visual inspection of samples, we notice that the domain-specific feature is seen a lot in samples from both domains, e.g. we see many image pairs with blond hair when targeting this feature on the second generator.

We only report the LPIPS on the CelebSmiling case for comparison. Here we notice that similarity between generated images is very high compared to our other approaches. This indicates that the controlled targeting of features is important in these cases where the difference between the features of both domains are subtle. Since a smile could be considered a fairly low-level feature it would be interesting to change the FRL extraction layer to capture this earlier in the model.



a) CelebSmiling

b) CelebBlond

c) CelebEyeglasses

Figure 5.8: Samples from SCoGAN-PL on the CelebSmiling(a), Blond(b) and Eyeglasses(c) datasets

5.4 CycleGAN datasets

As mentioned in chapter 4, a major drawback of the semantic loss is that it is reliant on having labelled data from at least one, and preferably both domains, which reduces the number of datasets it can be successfully applied to. The perceptual loss does not suffer from this limitation and as such we show its performance on the CycleGAN datasets horse2zebra and apple2orange, on which the semantic loss cannot be applied. For these experiments, we resize the images from 256x256 to 128x128 and use the same architecture and hyperparameter settings as for the previous experiments with the CelebSmiling. Each model is trained for 20.000 iterations. We measure LPIPS distance between 1000 samples for horse2zebra, and 995 samples for apple2orange as these datasets are a lot smaller than the others we use. The baseline LPIPS distance for horse2zebra is 0.7062 and for apple2orange it is 0.7013.

Configuration	CoGAN(a2o)	PL(a2o)	CoGAN(h2z)	PL(h2z)
Gen. Domain1 vs Gen. Domain2	0.6081	0.5774	0.6238	0.6163
Gen. Domain1 vs Real Domain1	0.6939	0.7361	0.6691	0.6715
Gen. Domain2 vs Real Domain2	0.7021	0.7339	0.6844	0.6900

Table 5.4: All LPIPS scores obtained on the horse2zebra and apple2orange datasets

Samples from the trained SCoGAN-PL models are shown on Figure 5.9. Visual inspection shows that there are a small number of similarities between image pairs, however, it is not any more significant than what can be produced by a CoGAN. This could indicate that to be more successful in finding similar shapes on this dataset, we need to increase the weight on the FRL term as it seems that both models manage to capture their respective style domains i.e. we see brown horses and striped zebras as well as red apples and orange oranges.

These rather poor results are backed by high LPIPS distances shown in Table 5.4 signalling that the models do not learn similar features very well. Figure 5.10 and 5.11 shows samples from the two SCoGAN-PL models with low(a) and high(b) LPIPS scores on the horse2zebra and apple2orange datasets. On horse2zebra, we see that for image pairs with low LPIPS, the zebra pattern has been completely discarded, and seemingly no meaningful shapes are present. For the high scoring image pair, the zebra image somewhat resembles the neck and head of a zebra, but the corresponding horse image retains none of these features. Apple2orange has more visible shapes in the low scoring pair, however, it seems that the red colour has merged into the orange domain, a similar effect to what we saw on the CelebA variants using perceptual loss. Furthermore, on both datasets, we notice occurrences of 'checkerboard' artifacts. We discuss these results and the checkerboard artifacts further in section 7.4. We also look into the behaviour of the SRL and FRL terms during training.

a) Horse2zebra b) Apple2orange

Figure 5.9: Samples from SCoGAN-PL on horse2zebra and apple2orange

a) LPIPS Score = 0.4474 b) LPIPS Score = 0.8266

Figure 5.10: Samples generated from a SCoGAN-PL model with corresponding LPIPS score. Leftmost image in each pair belongs to the horse domain and rightmost belongs to the zebra domain

a) LPIPS Score = 0.3496 b) LPIPS Score = 0.7897

Figure 5.11: Samples generated from a SCoGAN-PL model with corresponding LPIPS score. Leftmost image in each pair belongs to apple domain and rightmost belongs to orange domain

Chapter 6 Implementation

As a part of our previous semester project, we implemented a GAN training framework that allows us to easily train different GANs on different datasets. For this thesis, we have expanded upon this framework such that it now includes CoGAN and SCoGAN models as well. The framework is written in Python3.6 using the machine learning framework Tensorflow2.0 to implement models and training procedures. Our framework is used for all the previously presented experiments. The Framework consists of nine modules: `Main`, `Data`, `Nets`, `Losses`, `Penalties`, `Gan_trainer`, `CoGAN_trainer`, `SCoGAN_trainer` and a utility module, `Utils`, which mainly contains helper functions for the Main and trainer modules. An illustration of the framework architecture is shown in Figure 6.1. In this figure, every module imports and utilises all the in-going connected modules.

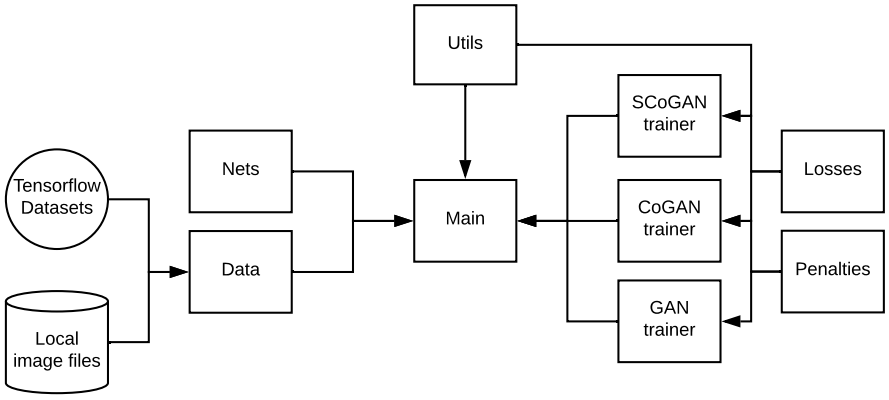


Figure 6.1: The structure of the GAN training framework which we have developed over two semesters

6.1 Main

The main module binds together the entire framework and is responsible for receiving user input and starting the correct training that corresponds to the user-provided configuration. Input about the training configuration is given to this module when launching it, as an array of arguments which specify the type of GAN, CoGAN or SCoGAN you wish to train, as well as on which dataset, with which architecture and which hyperparameter settings to use. An explanation of to run training using our framework is provided in our Github repository

`https://github.com/palminde/P10Project`.

To handle the input arguments, we use a Python package called `argparse` that parses the given arguments and stores them in a dictionary which we call `args`. When user arguments have been parsed, the `Main` module instantiates the correct Keras optimisers for the generative and discriminative models using the user-specified hyperparameter settings. When this is done, the necessary datasets are loaded and prepared for training in the `Data` module and upon request from the `Main` module, the necessary discriminator and generator nets are selected from the `Nets` module. When the correct dataset and neural nets have been found, a trainer is instantiated, which contains functionality for training either a GAN, CoGANs, or SCoGANs. When instantiated, the trainer selects the correct loss function and penalty settings based on user arguments. During training the trainer keeps a record of the loss values and samples images from the generator at a user-selected interval. The following sections go more in-depth with each module and how it works internally.

6.2 Data

This module is responsible for loading data from either local image files or the Tensorflow Datasets API and preprocessing it such that it can be used for either GAN or (S)CoGAN training. We use the Tensorflow Data API which allows us to build datasets that can be automatically batched and shuffled. To fetch and preprocess data for (S)CoGAN training, we use the function `select_dataset_cogan()`. On Listing 6.1 is a snippet of this method, where we use the Tensorflow Dataset API to load the correct dataset in line 4. The Tensorflow Dataset objects for the two domains are extracted from the data dictionary in line 5 while line 7 and 8 is for preprocessing the images. Tensorflow Dataset objects allow for an array of functions to be called on them such as map, filter, and reduce. In this case, we perform our preprocessing by mapping a function containing image transformations onto the dataset.

The preprocessing we apply with the function `format_example_scale` is shown in Listing 6.2. This function takes two inputs, an image and its label and returns a preprocessed image and its label. The preprocessing we perform on CycleGAN datasets is simply scaling pixel values to the range [-1, 1], however, many other operations can be added in this function such as image size scaling. Each dataset is then shuffled, repeated, and batched according to some batch size. Repeating a dataset enables us to continuously loop through it with an iterator without having to reset it and reshuffle.

```
...
elif args.cogan_data in ['apple2orange', 'horse2zebra']:
    # Domains
    data, info = tfds.load('cycle_gan/'+args.cogan_data, with_info=True,
    as_supervised=True)
    X1, X2 = data['trainA'], data['trainB']

    X1 = X1.map(format_example_scale)
    X2 = X2.map(format_example_scale)
    num_examples = info.splits['trainA'].num_examples

    X1 = X1.shuffle(num_examples).repeat().batch(args.batch_size)
    X2 = X2.shuffle(num_examples).repeat().batch(args.batch_size)
    shape = (None, 256, 256, 3)
...
```

Listing 6.1: Dataset selection and preprocessing - implemented in python

```
def format_example_scale(image, label):
    image = tf.cast(image, tf.float32)
    # Normalize the pixel values
    image = (image - 127.5) / 127.5
    return (image, label)
```

Listing 6.2: One preprocessing function - implemented in python

6.3 Nets

The `Nets` module contains all of our discriminator and generator architectures for regular GANs, CoGANs, and SCoGANs. The models are implemented using the Keras Model API available through Tensorflow as such we utilise the Keras implementation of all neural net components such as layers, activations, and weight initialisers. In this section, we show how we have implemented a GAN used for generating 32x32 images and how this GAN can be transformed into a CoGAN with weight sharing between generators and discriminators. Furthermore, we show the multi-output SCoGAN architecture necessary for the feature regulariser to extract features maps from intermediate layers. We do not show the single output SCoGAN implementation as this is simply two GANs with no relation to each other.

Listing 6.3 shows a regular GAN generator which takes as input a noise vector in line 4, performs a sequence of transformations on it from line 7 to 14, and finally outputs an image on line 16. This code defines a Keras model based on the sequence of transformations. This model can be trained and used to generate new images. To allow a generator with this architecture to share the weights of all its layers except the last with another similar generator is quite simple. All it requires

is for us to replace the final layer of Listing 6.3 (line 16) with the code shown in Listing 6.4. The output of the sequence of transformations up until line 14 is now given to two separate layers that each generates an image independent of each other. As we can see on line 7 of Listing 6.4, two models are now defined using the same input but producing different outputs. These two models will share the layers defined on and before line 14 and have separate output layers.

Listing 6.4 additionally shows how weight decay, weight initialisers, and bias initialisers are implemented in Keras models. This is done through the arguments `kernel_initialzer`, `kernel_regularizer`, and `bias_initializer` respectively. For each of these arguments, we have a corresponding value in the `args` dictionary that can be set by the user to define which type of initialisation and weight decay should be used. These arguments have been removed from future listings for simplicity, however, we do utilise weight decay, and the two types of initialisation in every layer in all generator and discriminator models.

```
1  def gan_generator_32(args):
2      channels = args.dataset_dim[3]
3
4      noise = tf.keras.layers.Input(shape=(args.noise_dim,))
5      model = tf.keras.layers.Dense(1024*4*4, kernel_initializer=args.w_init
       , kernel_regularizer=args.wd, bias_initializer=args.bi)(noise)
6      model = tf.keras.layers.Reshape((4, 4, 1024))(model)
7      model = (tf.keras.layers.BatchNormalization())(model)
8      model = (tf.keras.layers.PReLU(args.prelu_init))(model)
9
10     ... # Intermediate layers removed to keep listing shorter - See
       appendix A for full architecture
11
12     model = (tf.keras.layers.Conv2DTranspose(128, (3,3), strides=(2, 2),
       padding='same', kernel_initializer=args.w_init, kernel_regularizer=
       args.wd, bias_initializer=args.bi))(model)
13     model = (tf.keras.layers.BatchNormalization())(model)
14     model = (tf.keras.layers.PReLU(args.prelu_init))(model)
15
16     img = tf.keras.layers.Conv2DTranspose(channels, (6,6), strides=(1, 1),
        activation='tanh', padding='same', kernel_initializer=args.w_init,
       kernel_regularizer=args.wd, bias_initializer=args.bi)(model)
17
18     return keras.Model(noise, img)
```

Listing 6.3: 32x32 image GAN generator - implemented in Python

```
    # Generator 1
    img1 = tf.keras.layers.Conv2DTranspose(channels, (6,6), strides=(1, 1)
    , activation='tanh', padding='same', kernel_initializer=args.w_init,
    kernel_regularizer=args.wd, bias_initializer=args.bi)(model)

    # Generator 2
    img2 = tf.keras.layers.Conv2DTranspose(channels, (6,6), strides=(1, 1)
    , activation='tanh', padding='same', kernel_initializer=args.w_init,
    kernel_regularizer=args.wd, bias_initializer=args.bi)(model)

    return keras.Model(noise, img1), keras.Model(noise, img2)
```

Listing 6.4: Changes performed to GAN generator implementation create the 32x32 image CoGAN generators - implemented in Python

Listing 6.5 shows how we have implemented the discriminator part of this GAN. It consists of an input layer on which we perform some initial transformations. This is then provided to a predefined sequential model that contains all intermediate layers and the output layer. We do not have to make use of a sequential model like this to implement a discriminator, it could have followed the same structure as we saw in the generator, i.e. just have the layers continuously after each other. However, we choose this structure as it is easier to alter into a CoGAN discriminator. To make this alteration, we replace the input layers in line 5-7 with the code shown in line 2-9 of Listing 6.6 such that our model now has two separate input layers. Then we have to replace the code in line 15 and 16 of Listing 6.5 with line 11-13 of Listing 6.6. This code takes the output from each of the separate input layers and has the shared sequential model evaluate both separately and then defines two Keras models.

```
def cogan_discriminators_digit(args):
    img_shape = (args.dataset_dim[1], args.dataset_dim[2], args.
    dataset_dim[3])

    # Discriminator input
    img = tf.keras.layers.Input(shape=img_shape)
    x = tf.keras.layers.Conv2D(20, (5, 5), padding='same')(img)
    x = tf.keras.layers.MaxPool2D()(x)

    # Sequential model of intermediate layers and output
    model = keras.Sequential()
    model.add(tf.keras.layers.Conv2D(50, (5, 5), padding='same'))
    ... # Intermediate layers removed to keep listing shorter - See
    appendix A for full architecture
    model.add(tf.keras.layers.Dense(1))

    output = model(x, training=True)
```

```
16     return keras.Model(img, output)
```

Listing 6.5: Changes performed to GAN generator implementation create the 32x32 image CoGAN generators - implemented in Python

```
1      # Discriminator 1
2      img1 = tf.keras.layers.Input(shape=img_shape)
3      x1 = tf.keras.layers.Conv2D(20, (5, 5), padding='same')(img1)
4      x1 = tf.keras.layers.MaxPool2D()(x1)
5
6      # Discriminator 2
7      img2 = tf.keras.layers.Input(shape=img_shape)
8      x2 = tf.keras.layers.Conv2D(20, (5, 5), padding='same')(img2)
9      x2 = tf.keras.layers.MaxPool2D()(x2)
10
11     output1 = model(x1, training=True)
12     output2 = model(x2, training=True)
13     return keras.Model(img1, output1), keras.Model(img2, output2)
```

Listing 6.6: Changes performed to GAN generator implementation create the 32x32 image CoGAN generators - implemented in Python

Having shown how we implement GANs and CoGANs, we now move on to the multi-output SCoGAN model. On Listing 6.7, we create an array `output1`(line 2) onto which we append all intermediate outputs of the network (line 12-13, 16-17, and 22). We can then specify a Keras model with inputs `noise` and outputs as the list of outputs we have to create, namely `output1`.

```
1  def cogan_generators_digit_noshare(args):
2      output1 = []
3
4      noise = tf.keras.layers.Input(shape=(args.noise_dim,))
5      model = tf.keras.layers.Dense(1024*4*4)(noise)
6      model = tf.keras.layers.Reshape((4, 4, 1024))(model)
7
8      # Generator 1
9      model1 = tf.keras.layers.Conv2DTranspose(512, (3,3), strides=(2, 2),
       padding='same')(model)
10     model1 = tf.keras.layers.BatchNormalization()(model1)
11     features1_8x8 = tf.keras.layers.PReLU(prelu_init)(model1)
12     output1.append(features1_8x8)
13
14     model1 = tf.keras.layers.Conv2DTranspose(256, (3,3), strides=(2, 2),
       padding='same')(features1_8x8)
15     features1_16x16 = tf.keras.layers.PReLU(prelu_init)(model1)
16     output1.append(features1_16x16)
17
18     model1 = tf.keras.layers.Conv2DTranspose(128, (3,3), strides=(2, 2),
       padding='same')(features1_16x16)
19     img1 = tf.keras.layers.Conv2DTranspose(channels, (6,6), strides=(1, 1)
       , activation='tanh', padding='same')(model1)
```

```
20
21      output1.append(img1)
22      generator1 = keras.Model(noise, output1)
23
24      ... # Generator 2 below here
```

Listing 6.7: Changes performed to GAN generator implementation create the 32x32 image CoGAN generators - implemented in Python

6.4 Losses and Penalties

These two modules implement the loss functions (non-saturating, WGAN) and penalties (WGAN-GP, feature regulariser) that we have described in our report. The goal of these two modules is to make the losses and penalties dynamic such that a user can easily choose exactly which loss and penalty configuration to use for training, without having to rewrite the training procedure. Additionally, the modules are easily expandable such that extra losses and penalties can be added also without having to perform large changes in related modules.

The entire `Losses` module is shown in Listing 6.8. All losses are implemented according to their formulation shown in the respective papers. Each generator loss takes only one argument, namely the output of the discriminator on generated samples, i.e. $D(G(\mathbf{z}))$ and each discriminator loss takes two arguments, namely the discriminator output on real and generated samples, i.e. $D(\mathbf{x})$ and $D(G(\mathbf{z}))$. For the non-saturating loss, we apply the Sigmoid function to outputs of the discriminator as all our discriminator nets are implemented without a Sigmoid activation as the last layer for binary classification. This allows us to use both the WGAN and non-saturating loss on all our models.

```
1  import Tensorflow as tf
2
3  def non_sat_gen(fake_output):
4      G_loss = -tf.reduce_mean(tf.math.log(tf.math.sigmoid(fake_output)))
5      return G_loss
6
7  def non_sat_disc(fake_output, real_output):
8      real_loss = tf.math.reduce_sum(-tf.reduce_mean(tf.math.log(tf.math.
       sigmoid(real_output))))
9      fake_loss = tf.math.log(1 - tf.math.sigmoid(fake_output))
10     D_loss = real_loss + fake_loss
11     return D_loss
12
13 def wasserstein_gen(fake_output):
14     G_loss = -tf.reduce_mean(fake_output)
15     return G_loss
```

```
16
17 def wasserstein_disc(fake_output, real_output):
18     D_loss = tf.reduce_mean(fake_output) - tf.reduce_mean(real_output)
19     return D_loss
20
21 def set_losses(args):
22     if args.loss == 'ce':
23         return cross_entropy_disc, cross_entropy_gen
24     elif args.loss == 'wgan':
25         return wasserstein_disc, wasserstein_gen
```

Listing 6.8: Non-saturating and Wasserstein loss - implemented in python

The `Penalty` module consists of two classes called `DiscriminatorPenalties` and `GeneratorPenalties`. The `DiscriminatorPenalties` class contains our implementation of the gradient penalty proposed in [13], shown in Listing 6.9 and `GeneratorPenalties` contains an implementation of our proposed feature regulariser, shown in Listing 6.10. Our implementation of WGAN-GP follows the description given in the original paper. First, the interpolated image is computed. Then, Tensorflows `GradientTape()`, which defines a scope and records computations performed within it such that the gradients may be computed later is used to find the gradients of the discriminator output with regards to the interpolations. Finally, the L2 of the gradients is computed and we found how far they deviate from 1. Our implementation of the feature regulariser is quite simple, we iterate a predefined number of layers and sum over the means of the squared difference between features maps from the respective layers.

```
1  def wasserstein_gp(self, fake_data, real_data, discriminator):
2      # Interpolation constant
3      alpha = tf.random.uniform(shape=[real_data.shape[0],1,1,1], minval=0.,
        maxval=1.)
4      # Calculate interpolations
5      differences = fake_data - real_data
6      interpolated_images = real_data + (alpha * differences)
7      # Gradients of interpolated images
8      with tf.GradientTape() as gTape:
9          gTape.watch(interpolated_images)
10         disc_interpolates = discriminator(interpolated_images, training=
       True)
11     gradients = gTape.gradient(disc_interpolates, interpolated_images)
12     gradients += 1e-8
13     # Calculate penalty from image gradients
14     slopes = tf.sqrt(tf.math.reduce_sum(tf.math.square(gradients), axis
       =[1,2,3]))
15     gradient_penalty = tf.reduce_mean((slopes - 1.) ** 2)
16     return gradient_penalty
```

Listing 6.9: Gradient penalty presented in [13] - implemented in python

```
def feature_regularizer(self, g1_batch, g2_batch, shared_layers):
    distance = 0
    for idx in range(shared_layers):
        distance = distance + tf.reduce_mean(tf.math.squared_difference(
    g1_batch[idx], g2_batch[idx]))
    return distance
```

Listing 6.10: Our proposed feature regulariser - implemented in python

6.5 CoGAN_trainer

This module implements the CoGAN training procedure. It consists of a class called `Trainer`, shown in Listing 6.11, which implements the training procedure in the function `train()` as well as functionality for logging generator and discriminator losses, sampling images during training and clipping discriminator weights in the case where the WGAN loss is used. When instantiated it requires the two necessary datasets representing each domain and the two generators and discriminators. Furthermore, it contains instances of the `DiscriminatorPenalty` and `GeneratorPenalty` classes and four instance variables used for logging loss values for each model. In this section, we only cover the `train()` function as image sampling, and logging is deemed trivial and unnecessary to include.

```
class Trainer(object):
    def __init__(self, g1, g2, d1, d2, domain1, domain2):
        self.hist_g1 = []
        self.hist_g2 = []
        self.hist_d1 = []
        self.hist_d2 = []
        self.X1 = domain1
        self.X2 = domain2
        self.full_training_time = 0
        self.discPenal = p.DiscriminatorPenalties()
        self.genPenal = p.GeneratorPenalties()
        self.d1, self.d2 = d1, d2
        self.g1, self.g2 = g1, g2

    def train():
    def sample_images():
    def plot_losses():
    def clip_weights(self, clip):
```

Listing 6.11: Overview of Trainer class - implemented in python

The `train()` function retains the loop structure of algorithm 1, thus consisting of two loops of which the outer loop counts training iterations and the inner loop counts extra discriminator iterations.

Within the inner loop, we sample data from both real data domains, generate a noise vector, and use it to generate two batches of images, one for each domain. Using the generated and real image batches, we wish to update both our discriminators. For this, we again use Tensorflows `GradientTape()`. The update procedure for one CoGAN discriminator is shown in Listing 6.12. Each cycle in the inner loop consists of two of these, one for each discriminator. Within the scope of the `GradientTape()`, we apply the discriminator on both real and generate images and calculate the loss based on these in line 7. Next, the discriminator penalty is calculated and added to the loss. If no discriminator penalty is to be applied, then `calc_penalty()` will return 0. After a single update on both discriminators, we either move on to updating generators or run another cycle in the inner a new noise vector and new real images.

```
1  gen_batch1 = self.g1(noise, training=True)
2  with tf.GradientTape() as tape:
3      # Disc response
4      disc_real1 = self.d1(batch1, training=True)
5      disc_fake1 = self.d1(gen_batch1, training=True)
6      # Calc loss and penalty
7      d1_loss = d_loss_fn(disc_fake1, disc_real1)
8      gp1 = self.discPenal.calc_penalty(gen_batch1, batch1, self.d1, args)
9      d1_loss = d1_loss + (gp1 * args.penalty_weight_d)
10     #Calc gradients
11 gradients_of_discriminator = tape.gradient(d1_loss, self.d1.
       trainable_variables)
12 args.disc_optimizer.apply_gradients(zip(gradients_of_discriminator, self.
       d1.trainable_variables))
```

Listing 6.12: CoGAN discriminator training procedure - implemented in python

Generator updates are implemented similarly to the discriminator updates. We again use `GradientTape()` as shown on Listing 6.13. Using a new noise vector, we generate a batch of images and have the discriminate evaluate them in lines 2 and 4. The generator loss is calculated and a generator penalty is added if one is enabled. Then the gradients are calculated and applied, and the outer loop finishes with logging of all four model losses and runs again.

```
1  with tf.GradientTape() as tape:
2      gen_fake = self.g1(noise, training=True)
3      # Disc response
4      disc_fake = self.d1(gen_fake, training=True)
5      g1_loss = g_loss_fn(disc_fake)
6      # Calc loss and penalty
7      penalty1 = self.genPenal.calc_penalty(self.g1, self.g2, 20, args)
8      g1_loss = g1_loss + (penalty1 * args.penalty_weight_g)
9      #Calc gradients
10 gradients_of_generator1 = tape.gradient(g1_loss, self.g1.
       trainable_variables)
```

```
11 args.gen_optimizer.apply_gradients(zip(gradients_of_generator1, self.g1.
       trainable_variables))
```

Listing 6.13: CoGAN generator training procedure - implemented in python

6.6 SCoGAN_trainer

The SCoGAN_trainer is the main contribution to the framework in this project. It is developed as a prototype to explore our different proposals in combination with each other. The SCoGAN_trainer resembles the CoGAN_trainer, but with a few alterations. It contains more variables meant for logging different loss term values that are related to our proposals, during training, e.g. semantic loss value or cycle loss value. It also contains the implementation of all proposed approaches, besides the feature regulariser which is part of the `GeneratorPenalty` class presented in section 6.4. On Listing 6.14, we see the initial check of the `train` method, where we load auxiliary networks, lines 5 and 7 for semantic and perceptual loss respectively, or create additional neural networks as the encoder in line 3 for the cycle consistency loss.

```
1 def train(self, args):
2         if args.use_cycle:
3             self.encoder = n.encoder(args)
4         if args.semantic_loss:
5             self.classifier = tf.keras.models.load_model(args.
      classifier_path)
6         if args.perceptual_loss:
7             self.vgg_feature_model = self.feature_layers(self.style_layers
       + self.content_layers, args)
8         ... # Rest of the train method below here
```

Listing 6.14: SCoGAN train method - implemented in python

These extra networks are used when updating the generators later in the training method. We see on Listing 6.15, the beginning of the generator update method, and how we utilise three gradient tapes simultaneously to capture calculations that allows for updating generator1, generator2, and the cycle consistency encoder within one scope. This is done since all proposals rely on information from both generators. The first three lines in this `with` block constitute the adversarial loss for generator1 and here, we see a difference from the `CoGAN_trainer` as the input to the discriminator at line 4 is the last element of `gen1_fake`. This is seen throughout the SCoGAN trainer, as it uses the multi-output generators. Therefore, `gen1_fake` is a list of outputs, where the last element is the image output. From line 9, we see the implementation of the semantic loss term. This loss is calculated as the mean

of the squared difference between predictions on both domains. The value is then logged and added to the total generator loss.

```
with tf.GradientTape() as tape1, tf.GradientTape() as tape2, tf.GradientTape() as tape3:
    # Adv loss
    gen1_fake = self.g1(noise, training=True)
    disc1_fake = self.d1(gen1_fake[-1], training=True)
    g1_loss = g_loss_fn(disc1_fake)

    ... #Omitted adverserial loss for generator2

    if args.semantic_loss:
        domain1_pred = self.classifier(gen1_fake[-1])
        domain2_pred = self.classifier(gen2_fake[-1])
        diff = tf.reduce_mean(tf.math.squared_difference(domain1_pred, domain2_pred))
        # log semantic loss
        self.hist_semantic_loss.append(diff)
        g1_loss = g1_loss + diff * args.semantic_weight
        g2_loss = g2_loss + diff * args.semantic_weight
```

Listing 6.15: SCoGAN generator update with semantic loss - implemented in python

The implementation of the cycle consistency loss term is equally simple and can be seen on Listing 6.16. Here, we provide the extra encoder network with the image output of each generator. Then, we calculated the mean absolute error between each reconstruction and the original noise vector as seen in lines 4-5. The total reconstruction loss is simply the sum of each reconstruction task. This loss value is then multiplied by its respective weight and added to the total generator loss. This reconstruction loss is also the loss of the encoder minimises during training.

```
if args.use_cycle:
    noise_recon1 = self.encoder(gen1_fake[-1])
    noise_recon2 = self.encoder(gen2_fake[-1])

    noise_recon_loss1 = tf.math.reduce_mean(tf.math.abs(noise_recon1 - noise))
    noise_recon_loss2 = tf.math.reduce_mean(tf.math.abs(noise_recon2 - noise))

    total_recon_loss = noise_recon_loss1 + noise_recon_loss2

    # log cycle loss
    self.hist_cycle_loss.append(total_recon_loss)

    g1_loss = g1_loss + (total_recon_loss * args.cycle_weight)
    g2_loss = g2_loss + (total_recon_loss * args.cycle_weight)
```

Listing 6.16: SCoGAN cycle consistency loss - implemented in python

Finally, the perceptual loss is implemented through different steps. The first step is found on Listing 6.17, which shows the method `feature_layers(self, layer_names)`. This method is used to instantiate the feature extraction network from a pretrained VGG classifier. Here, we use the Keras Applications API to download and cache a pretrained VGG19 network in line 2. The input of the method `layer_names` is a list of names for the layers, we want to get intermediate outputs from. We build a list of output layers from the VGG network through these layer names in line 3. We create a model whose input is that of the VGG network and whose outputs are all these intermediate layers. The method `StyleContentModel(self, inputs)` is responsible for passing input through the network and returning separate style and content features. Here, all style feature maps are passed through the method `gram_matrix` at line 15. In this method, we calculate the gram matrix for the feature maps through the `einsum` method in line 23. The `input` style map has dimensions $B \times H \times W \times C$ and we specify that `einsum` should do matrix multiplication as $\sum_i \sum_j input[b,i,j,c] * input[b,i,j,d] = out[b,c,d]$. Finally, we normalise over the total number of values in the feature map.

```
1  def feature_layers(self, layer_names):
2      vgg = tf.keras.applications.VGG19()
3      outputs = [vgg.get_layer(name).output for name in layer_names]
4
5      model = tf.keras.Model([vgg.input], outputs)
6      return model
7
8  def StyleContentModel(self, inputs):
9      inputs = (0.5 * inputs + 0.5) * 255
10     inputs = tf.keras.applications.vgg19.preprocess_input(inputs)
11     feature_outputs = self.vgg_feature_model(inputs)
12     styles = feature_outputs[:self.num_style_layers]
13     content = feature_outputs[self.num_style_layers:]
14
15     style_outputs = [self.gram_matrix(style) for style in styles]
16
17     content_dict = {content_name:value for content_name, value in zip(self
       .content_layers, content)}
18     style_dict = {style_name:value for style_name, value in zip(self.
       style_layers,style_outputs)}
19
20     return content_dict, style_dict
21
22 def gram_matrix(self,input):
23     result = tf.linalg.einsum('bijc,bijd->bcd', input, input)
24     num_locations = input.shape[1] * input.shape[2]
25     return result/num_locations
```

Listing 6.17: SCoGAN perceptual loss setup methods - implemented in python

In the generator update section of the `train` method, we calculate the perceptual

loss through the code seen at Listing 6.18 line 1-9. Here, we pass generated and real images through the method explained in the previous listing to produce style and content features. In lines 7 and 8, we calculate the SRL and FRL with the `StyleContentLoss` method seen in line 13. This method calculates the SRL as the sum of mean squared differences between the respective style gram matrixes, and the FRL is calculated likewise. The values are normalised through the number of layers and multiplied by their respective weights.

```
1  ... //Snippet of train method
2  if args.perceptual_loss:
3  fake1_content, fake1_style = self.StyleContentModel(gen1_fake[-1])
4  fake2_content, fake2_style = self.StyleContentModel(gen2_fake[-1])
5  real1_content, real1_style = self.StyleContentModel(batch1)
6  real2_content, real2_style = self.StyleContentModel(batch2)
7
8  g1_style_loss, g1_content_loss = self.StyleContentLoss(fake1_style,
       real1_style, fake1_content, fake2_content, args)
9  g2_style_loss, g2_content_loss = self.StyleContentLoss(fake2_style,
       real2_style, fake2_content, fake1_content, args)
10
11 g1_loss = (g1_loss) + g1_style_loss + g1_content_loss
12 g2_loss = (g2_loss) + g2_style_loss + g2_content_loss
13 ...
14
15
16
17 def StyleContentLoss(self, style_fake, style_target, content_fake1,
       content_fake2, args):
18     style_loss = tf.add_n([tf.reduce_mean((style_fake[name]-style_target[
       name])**2) for name in style_fake.keys()])
19     content_loss = tf.add_n([tf.reduce_mean((content_fake1[name] -
       content_fake2[name])**2) for name in content_fake1.keys()])
20
21     style_loss *= args.style_weight / self.num_style_layers
22     content_loss *= args.content_weight / self.num_content_layers
23
24     return style_loss, content_loss
```

Listing 6.18: SCoGAN perceptual loss calculation - implemented in python

Chapter 7 Discussion

In this chapter we discuss the results achieved during experimentation as well as other aspects of the project. These discussion points are mainly focused on providing clearer thought behind some decisions or providing additional information that seemed out of place elsewhere.

7.1 Domain adaptation and style transfer

We motivate this project through the two use-cases domain adaptation and style transfer, which the original CoGAN is capable of doing [34]. However since our focus has been on the learning of joint distributions and not on the use cases of this, we have not directly shown that our proposed methods are equally capable of these tasks. Therefore we here note that all our proposals can be used for domain adaptation and style transfer in the same way as described in [34]. This is the case as we are not altering the input/output scheme, and we are preserving the joint distribution aspect of CoGAN.

7.2 Joint distribution learning on CelebA variants

Throughout this project, we experience issues with our implementation of CoGAN and SCoGAN on the CelebA variants. While both models learn to generate decent looking facial images, none manage to learn the joint distribution. We have investigated the following problems that we believe can cause such an issue: (1) error in data loading, (2) error in the training procedure, loss or penalty implementations, and (3) error in the network architecture implementation.

We confirmed that the data was being loaded correctly by inspecting dataset sizes after CelebA has been split into two domains and ensuring that the sizes in our implementation matched the ones specified in [34]. We extracted a subset of CelebA consisting of 3700 images for which we knew how many images belonged to each domain, and for this smaller test, we also see that the images are split correctly into two datasets. Additionally, we have performed a visual inspection of the first 200 samples from each domain for the full dataset without seeing any misplaced images in either domain.

Errors in the implementation of the training procedure, loss, and penalties are to us not very likely, as we are using the same implementation on the MNIST2edge and MNIST2SVHN datasets, where SCoGAN models do manage to learn the joint distributions. One extra step that we could take in ensuring correct implementation is unit testing our loss and penalty implementations. We performed the testing informally, comparing the output of our losses to the output of official implementations such[1] for WGAN-GP and found our implementation to produce the same values.

Lastly, we looked for errors in our implementation of the neural net architectures. We thoroughly examined the official implementation of CoGAN and built our architectures following the Caffe definition files they have made available. This included comparing Caffe and Keras documentation of e.g. batch normalisation and parametric ReLU layers to ensure that default hyperparameter settings were the same and if not, to include Caffe default values in our implementation. An example is the initial value of parametric ReLU being 0.25 in Caffe and 0 in Keras.

Furthermore, we wished to confirm if the official CoGAN implementation could reproduce the shown CelebA results by running their code. However, the official repository only contains scripts for running MNISTedge experiments with no specification for how to train on CelebA and seemingly no data/data loading for CelebA datasets. We, therefore, do not know if the architecture described in the definition file available in their implementation is even capable of learning the joint distribution, which may not be the case. Additionally, as we previously described in section 3.3, their Caffe implementation does not follow the described architectures in their paper, and their PyTorch implementation only covers the MNIST cases. We, therefore, believe that some information could be missing from their paper or implementation about the celebA model and configurations. Another aspect that could cause these discrepancies is that we are not familiar with the Caffe framework, and there could be computational differences between the internal implementation of the Keras and Caffe framework, that we have been unable to find.

[1] `https://github.com/igul222/improved_wgan_training`

7.3 Combining regulariser terms

From the experiments on the MNIST2SVHN dataset, we found that both SCoGAN combinations with semantic loss produce compelling samples, while the combination of feature regulariser and cycle consistency did not. We elaborate on this here.

The feature regulariser is a softer coupling of the generators than the weight sharing constraint of CoGAN, however, the feature regulariser is still directly constraining the generators during training, i.e. we are in both cases trying to control the intermediate layers that the model uses to generate images, but with varying degrees of freedom. Meanwhile, the cycle consistency is an implicit way of enforcing feature similarity between generated images, but we are here unable to directly understand or influence how these feature similarities are made in this approach. A combination of these terms (FR and CC) provides the SCoGAN-FRCC with a multi-task learning objective where one task can be restrictive on the generation process(FR), and the other task is not directly related to producing similar high-level features or content, between generated samples(CC). This might be the reason why this combination performs poorly in relation to the ones with the semantic loss, as the semantic loss is a direct and non-restrictive way of enforcing similarity between the content of generated images. Therefore, the semantic loss can be directing either approach in their respective combinations.

7.4 Perceptual loss

As previously mentioned the perceptual loss was added late in the project and therefore tuning performed on this approach has been minimal. However we believe the intuition of separating learned features into style and content can yield good results and therefore we provide a discussion below on some of the choices and further aspects regarding this proposal.

7.4.1 Perceptual loss values and weight

Before implementing the approach ourselves we examined the official implementation[2] through code inspection and execution. Here we found that the loss values of the FRL and SRL term were many magnitudes larger than our adversarial loss term for the generators. For example the FRL term would take a value > 10.000 and

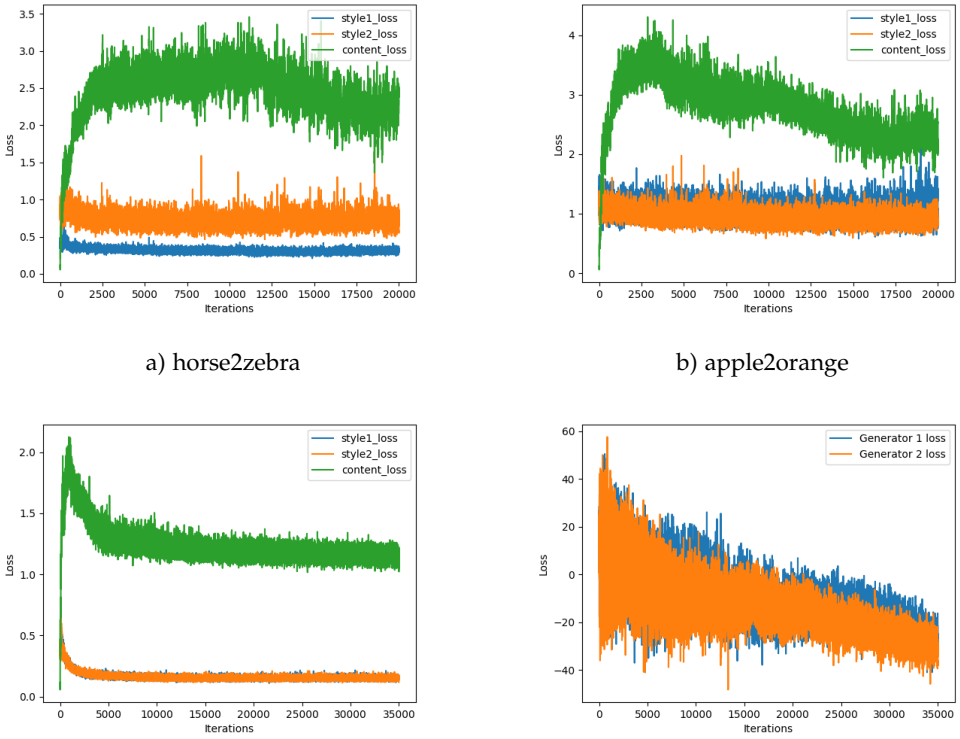
[2] `https://github.com/jcjohnson/fast-neural-style`

the SRL term would be $> 10.000.000$. These are the values before multiplying with the weight, which for the default setting was set to $w_{FRL} = 1$ and $w_{SRL} = 1250$. If we were to introduce these values into our total generator loss term we would severely diminish the effect, that the adversarial loss would have on the generator updates. Therefore we choose to use the weights to scale down the loss values to proportionately fit with the adversarial loss, such that each term of the generators loss would roughly contribute equally to the total value. Our adversarial loss term for the generators produces values approximately in the $[-100, 100]$ range, and therefore we set the weights $w_{FRL} = 0.001$ and $w_{SRL} = 0.00000000001$. These were the only tested weights for this method and as we see in our horse2zebra, apple2orange and CelebA experiments these values are seemingly not optimal.

One reason that we see so poor results on these datasets is the SRL term is constant through the entire training for the horse2zebra, apple2orange and the CelebA datasets, as shown on Figure 7.1a-c. Ideally this term should to steadily decrease throughout training as the model learns the domain specific feature and since it is constant, this learning clearly does not take place. Furthermore Figure 7.1d shows the generator loss for an SCoGAN-PL model trained on CelebSmiling. Here we see that the loss is decreasing while oscillating with rather large values compared to those seen in its corresponding SRL and FRL terms shown on Figure 7.1c. In this case these terms may not have a very large influence even if they were not constant. Apple2oranges and horse2zebra datasets similarly shows much larger generator loss. We therefore believe that further testing with different weight values as well as an exploration of the optimal amount of layers to extract feature maps from in the pretrained model is necessary in order to fully understand the capability of this method.

7.4.2 CycleGAN dataset results

Neither CoGAN nor SCoGAN-PL achieve stunning results on the horse2zebra and apple2orange datasets. We suspect that this is related to the architectures that we have used, rather than a failure of CoGAN or SCoGAN-PL. This is due to the fact that CycleGAN uses network architectures with 2 convolutional layers, followed by 6 residual blocks, followed by 2 transpose convolutions for learning 128x128 images. This points toward the necessity of using deeper networks on these datasets. We initially wanted to compare our results to those achieved by CoGAN and therefore used the architectures that they proposed, however to further investigate performance on the CycleGAN datasets a good place to start would most likely be updating the neural net architectures such that they follow the residual architecture used by CycleGAN.



a) horse2zebra

b) apple2orange

c) CelebSmiling

d) Generator loss for CelebSmiling

Figure 7.1: (a-c) SRL and FRL losses from training SCoGAN-PL on various datasets. (d) Generator loss for SCoGAN-PL on CelebSmiling

7.4.3 Checker board patterns in perceptual loss results

Inspecting samples from SCoGAN-PL models trained with the perceptual loss on the CycleGAN datasets we see that they exhibit checkerboard patterns. This issue may relate to the architecture depth issue discussed above. However another potential causes is proposed in [36] which establishes that such patterns are commonly observed and proposes that it is due to the use of transpose convolutions for upscaling the noise vector to an image in the generator. Transposed convolutions when upscaling images can suffer from overlapping kernels which amplify pixel values contained in the overlap. Since kernels are applied methodically across the entire image these overlaps will occur in a checkerboard pattern that depends on the kernel size and stride. [36] shows some very nice interactive figures of how this overlap looks. To solve this problem they propose to replace transposed convolutions with regular image resizing using nearest neighbour interpolation followed by a stride one convolution layer. While this may be a good place to start it does

not explain why the SCoGAN-PL models suffer from this problem, when a CoGAN with an equivalent architecture is not affect as heavily. This point towards an issue inherent in the perceptual loss approach which may be alleviated through further exploration as mentioned previously.

7.5 Further investigation of semantic loss

During our experiments on MNIST2SVHN we see that the two methods that show best performance both include the semantic loss. We therefore conduct a small additional experiment to examine whether the semantic loss alone can learn this joint distribution. We train an SCoGAN-SL model on MNIST2SVHN with the same configurations as mentioned in section 5.2. We set $\alpha_{SL} = 10$ and show samples from this training on Figure 7.2 and LPIPS scores in Table 7.1, which are very similar to those achieved by our other approaches on MNIST2SVHN. Visual inspection of samples shows us that this model manages to learn the joint distribution in similar fashion to the models combining semantic loss with the feature regulariser and cycle consistency respectively. Additional samples used for visual inspection can be found in our Github repository[3].

Figure 7.2: Samples from SCoGAN-SL trained on MNIST2SVHN with $\alpha_{SL} = 10$

[3] `https://github.com/palminde/P10Project`

Configuration	SL
Gen. MNIST vs Gen. SVHN	0.3018
Gen. MNIST vs Real MNIST	0.1140
Gen. SVHN vs Real SVHN	0.2053

Table 7.1: All LPIPS scores obtained on the MNIST2SVHN dataset

Chapter 8 Conclusion

Throughout this project we have worked with applying generative adversarial networks (GAN) to joint distribution learning over multi domain images. This work springs from an initial problem statement that is formulated based on a review of current literature within the field of joint distribution learning. Here we observe that an older GAN based model, called CoGAN [34], which when proposed showed impressive results and has the desirable property of being able to learn a joint distribution over multiple image domains requiring only data from the marginal distribution of each domain. Has been dismissed in literature due to lack of performance with no real effort in showing the problems that it suffers from. We therefore decide to investigate this model.

Through an experimental analysis we discover two main issues with CoGANs that may be the cause of this dismissal. The first issue is that CoGANs do not scale well to larger images. We find that this problem may relate more to the architecture of the GANs comprising the model, than the actual CoGAN framework, and therefore suspect that this issue can be solved by introducing a GAN architecture proposed for large scale image generation such as [23, 5, 24]. The second issue is that, as the image domains become less similar through either the appearance of background noise or content diversity the weight sharing constraint, used in CoGANs to learn the joint distribution, works as a restriction on its generators to the point where they cannot learn their respective domains.

We choose to focus on the second issue as it offers more opportunity for applying new approaches rather than implementing existing ideas. This leads us to our main problem statement: *How can we address the domain issue experienced with CoGANs in a way that improves the models capability of learning joint distributions, while still only training on marginal datasets?* and the following subquestions:

- How can we relax the weight sharing constraint that restricts the individual generators, in a way that increases the CoGANs capability of learning the joint distribution between image domains?
- How can we utilise the core idea of learning similar high-level features presented by CoGAN?
- Will we still need WGAN-GP, as previously seen, to avoid mode collapse on noisier datasets?

To solve this problem statement we propose to remove the strict weight sharing coupling used in CoGANs and instead aim to introduce a softer coupling which would allow each GAN more freedom over their respective weights to better learn their image domain. We call this type of model a Soft-CoGAN (SCoGAN). We propose to achieve the softer coupling through four different regularisation terms: (1 - feature regulariser) Regularise similarity between intermediate layer outputs of both generators, thereby soft coupling GANs to generate similar feature maps in the generation process. (2 - semantic loss) Use an auxiliary network to classify the image output of both generators and regularise similarity between these classification, thereby enforcing that the content on each image should come from the same class. (3 - cycle consistency) Simultaneously train an encoder network, that encodes generated images back to the original latent vector and regularise similarity between encoded and input latents. This implicitly enforces both generators to produce similar content that the encoder can use to produce the same output. (4 - perceptual loss) Use an auxiliary network to extract features of varying levels in the generated and real images to regularise high-level feature similarity between generated images in both domains, and low-level feature similarity between generated and real images in respective domains.

With our own implementation of the CoGAN architecture and our four proposed regularisation terms we conduct experiments on a variety of 32x32 and 128x128 datasets. Through a combination of qualitative visual inspection and quantitative LPIPS distance measures on samples from these experiments we find that all four regularisers approaches are able to learn a joint distribution over the low noise MNIST2edge dataset.

On the noisier dataset MNIST2SVHN we find that SCoGAN models that use the semantic loss, either alone or in conjunction with the feature regulariser or cycle consistency are capable of learning the joint distribution that CoGAN cannot. The perceptual loss manages to achieve similar results to those seen by the feature regulariser alone, but maintains more control over the generated style features on the MNIST2SVHN dataset.

On the CelebA datasets our implementation of both CoGAN and SCoGAN were unable to replicate the results seen in [34], which we discuss further in section 7.2. On this dataset both models learn similar high- and low-level features, and thus do not learn domain specific features. This indicates that the coupling between generators is too tight. This is seen through visual inspection of samples and the measured LPIPS distances. However the LPIPS distances also show that our proposed regulariser terms result in more diversity between generated samples than achieved by our CoGAN. Through this we see the benefits that our softer coupling provides, but still must conclude that the coupling is too tight.

Experiments on the two CycleGAN datasets apple2orange and horse2zebra, show that neither our proposed method nor CoGAN is able to achieve results that is comparable to those seen in CycleGAN. This could be explained by the deeper architecture of CycleGAN, which we discuss in subsection 7.4.2.

Additionally we show that WGAN-GP can be used to stabilise the learning of joint distributions in both CoGAN and SCoGAN, specifically in cases where the models face difficulties.

This concludes our work in this project and it is summarised in the following contributions: We show that the weight sharing constraint of CoGAN restricts the learning capabilities of the generators. Based on this we propose 4 different regularisation terms to provide a softer coupling between GANs that retain the ability to learn joint distributions, but are less constraining than the weight sharing of CoGAN. We then show that these methods can single-handedly or in combination create Soft Coupled GANs(SCoGANs) that either surpass or perform similarly to the CoGAN approach on joint distribution learning tasks. We specifically want to highlight our SCoGAN-SLFR and SCoGAN-SLCC models abilities to learn the joint distribution on MNIST2SVHN, which CoGAN is notorious for failing on. Through these regularisation terms we prove that the core intuition in CoGAN of coupling GANs on to achieve similar high-level features, can be utilised to create joint distribution learning models, that, in some cases, are better than the original proposed method.

Appendix A Generator and discriminator architectures

Generator	Output shape	Kernel	Stride	BatchNorm	Activation	Shared
Input	100	-	-	-	-	yes
Fully Connected	16384	-	-	-	-	yes
Reshape	4x4x1024	-	-	-	-	yes
ConvTranspose	4x4x512	4	2	yes	PRelu(0.25)	yes
ConvTranspose	8x8x256	4	2	yes	PRelu(0.25)	yes
ConvTranspose	16x16x128	4	2	yes	PRelu(0.25)	yes
ConvTranspose	32x32x64	4	2	yes	PRelu(0.25)	yes
ConvTranspose	64x64x32	4	2	yes	PRelu(0.25)	no
ConvTranspose	128x128x3	3	1	yes	Tanh	no

Table A.1: Original CoGAN generator architecture for 128x128 image experiments

Discriminator	Output shape	Kernel	Stride	BatchNorm	Activation	Shared
Input	128x128x3	-	-	yes	-	no
Conv	128x128x32	5	2	yes	PRelu(0.25)	no
Conv	64x64x64	5	2	yes	PRelu(0.25)	yes
Conv	32x32x128	5	2	yes	PRelu(0.25)	yes
Dropout(0.1)	32x32x128	-	-	-	-	yes
Conv	16x16x256	3	2	yes	PRelu(0.25)	yes
Dropout(0.3)	16x16x256	-	-	-	-	yes
Conv	8x8x512	3	2	yes	PRelu(0.25)	yes
Dropout(0.3)	8x8x512	-	-	-	-	yes
Conv	4x4x1024	3	2	yes	PRelu(0.25)	yes
Dropout(0.5)	4x4x1024	-	-	-	-	yes
Fully Connected	2048	-	-	yes	PRelu(0.25)	yes
Dropout(0.5)	2048	-	-	-	-	yes
Fully Connected	1	-	-	-	Sigmoid	yes

Table A.2: Original CoGAN discriminator architecture for 128x128 image experiments

Discriminator	Output shape	Kernel	Stride	BatchNorm	Activation	Shared
Input	-	100	-	-	-	yes
Input	100	-	-	-	-	yes
Fully Connected	32768	-	-	-	-	yes
Reshape	4x4x2048	-	-	-	-	yes
ConvTranspose	4x4x1024	4	2	yes	PRelu(0.25)	yes
ConvTranspose	8x8x512	4	2	yes	PRelu(0.25)	yes
ConvTranspose	16x16x256	4	2	yes	PRelu(0.25)	yes
ConvTranspose	32x32x128	4	2	yes	PRelu(0.25)	yes
ConvTranspose	64x64x64	4	2	yes	PRelu(0.25)	yes
ConvTranspose	128x128x32	4	2	yes	PRelu(0.25)	no
ConvTranspose	256x256x3	3	1	yes	Tanh	no

Table A.3: CoGAN generator architecture for 256x256 image experiments

Discriminator	Output shape	Kernel	Stride	BatchNorm	Activation	Shared
Input	256x256x3	-	-	yes	-	no
Conv	256x256x32	5	2	yes	PRelu(0.25)	no
Conv	128x128x64	5	2	yes	PRelu(0.25)	yes
Conv	64x64x128	5	2	yes	PRelu(0.25)	yes
Dropout(0.1)	64x64x128	-	-	-	-	yes
Conv	32x32x256	3	2	yes	PRelu(0.25)	yes
Dropout(0.3)	32x32x256	-	-	-	-	yes
Conv	16x16x512	3	2	yes	PRelu(0.25)	yes
Dropout(0.3)	16x16x512	-	-	-	-	yes
Conv	8x8x1024	3	2	yes	PRelu(0.25)	yes
Dropout(0.5)	8x8x1024	-	-	-	-	yes
Fully Connected	2048	-	-	yes	PRelu(0.25)	yes
Dropout(0.5)	2048	-	-	-	-	yes
Fully Connected	1	-	-	-	Sigmoid	yes

Table A.4: CoGAN discriminator architecture for 256x256 image experiments

Appendix B Classifier architectures for semantic loss

Digit Classifier	Shape	Kernel	Stride	Activation
Input	32x32x3	-	-	-
Conv	32x32x32	3	1	LeakyReLU(0.3)
Conv	32x32x64	3	1	LeakyReLU(0.3)
MaxPool	64x64x64	2	1	-
Dropout(0.25)	64x64x64	-	-	-
Fully Connected	128	-	-	LeakyReLU(0.3)
Dropout(0.5)	128	-	-	-
Fully Connected	10	-	-	Softmax

Table B.1: Pretrained classifier architecture used in the semantic loss for MNIST2edge and MNIST2SVHN

Digit Classifier	Shape	Kernel	Stride	Activation
Input	128x128x3	-	-	-
Conv	128x128x32	3	1	LeakyReLU(0.3)
Conv	128x128x64	3	1	LeakyReLU(0.3)
MaxPool	256x256x64	2	1	-
Dropout(0.25)	256x256x64	-	-	-
Fully Connected	128	-	-	LeakyReLU(0.3)
Dropout(0.5)	128	-	-	-
Fully Connected	13	-	-	Sigmoid

Table B.2: Pretrained classifier architecture used in the semantic loss for the CelebA variants

Appendix C Python script for WD and JSD experiment

```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import scipy

def JSD(P, Q):
    _P = P / np.linalg.norm(P, ord=1)
    _Q = Q / np.linalg.norm(Q, ord=1)
    _M = 0.5 * (_P + _Q)
    return 0.5 * (scipy.stats.entropy(_P, _M) + scipy.stats.entropy(_Q, _M
    ))


amount_of_samples = 100000
p_const = np.random.normal(0, size=amount_of_samples)
q1 = np.random.normal(10, size=amount_of_samples)
q2 = np.random.normal(15, size=amount_of_samples)
q3 = np.random.normal(20, size=amount_of_samples)

p_linspace = np.linspace(p_const.min()-1, p_const.max()+1,
    amount_of_samples)
p_pdf = scipy.stats.norm.pdf(p_linspace)

dists_jsd = []
dists_jsd2 = []
dists_w = []
for i in range(21):
    q = (np.random.normal(i, size=amount_of_samples))
    q_linspace = np.linspace(q.min() - 1, q.max() + 1, amount_of_samples)
    q_pdf = scipy.stats.norm.pdf(q_linspace)

    dist_jsd = JSD(p_pdf, q_pdf)
    dist_w = scipy.stats.wasserstein_distance(p_const,q)

    dists_jsd.append(dist_jsd)
    dists_w.append(dist_w)

sns.distplot(p_const, label='p')
sns.distplot(q1, label='q1')
sns.distplot(q2, label='q2')
sns.distplot(q3, label='q3')
plt.plot(dists_jsd, label='JSD')
plt.legend()
plt.show()

```

```
sns.distplot(p_const, label='p')
sns.distplot(q1, label='q1')
sns.distplot(q2, label='q2')
sns.distplot(q3, label='q3')
plt.plot(dists_w, label='WD')
plt.legend()
plt.show()
```

Bibliography

[1] Patrick Alminde and Markus H. Juul-Nyholm. "Deep Generative Models for Image Generation - An Investigation of GAN and VAE Models". In: *AAU library* (Jan. 2020).

[2] Martin Arjovsky and Léon Bottou. "Towards Principled Methods For Training Generative Adversarial Networks". In: *ICLR* (Jan. 2017).

[3] Martin Arjovsky, Soumith Chintala, and Léon Bottou. "Wasserstein GAN". In: *PMLR* (Dec. 2017).

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization". In: *ArXiv* (2016).

[5] Andrew Brock, Jeff Donahue, and Karen Simonyan. "Large Scale GAN Training for High Fidelity Natural Image Synthesis". In: *ICLR* (Sept. 2018).

[6] Xi Chen et al. "InfoGAN: Interpretable Representation Learning By Information Maximizing Generative Adversarial Nets". In: *NIPS* (July 2016).

[7] Yunjey Choi et al. "StarGAN: Unified Generative Adversarial Networks for Multi-domain Image-to-Image Translation". In: *IEEE CVPR* (2018).

[8] Yunjey Choi et al. *StarGAN v2: Diverse Image Synthesis for Multiple Domains*. 2019. arXiv: `1912.01865 [cs.CV]`.

[9] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *AISTATS* (2010).

[10] Ian Goodfellow. "NIPS 2016 Tutorial: Generative Adversarial Networks". In: *NIPS* (Apr. 2017).

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

[12] Ian Goodfellow et al. "Generative Adversarial Nets". In: *NIPS* (June 2014).

[13] Ishaan Gulrajani et al. "Improved Training of Wasserstein GANs". In: *NIPS* (Dec. 2017).

[14] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *ICCV* (Feb. 2015).

[15] Martin Heusel et al. "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium". In: *NIPS* (June 2017).

[16] Judy Hoffman et al. "CyCaDa: Cycle-Consistent Adversarial Domain Adaptation". In: *MLR* (Dec. 2019).

[17] Xun Huang et al. "Multimodal Unsupervised Image-to-Image Translation". In: *ECCV* (Aug. 2018).

[18] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *JMLR* (2015).

[19] Phillip Isola et al. “Image-to-Image Translation with Conditional Adversarial Networks”. In: *CVPR* (Nov. 2016).

[20] Yangqing Jia et al. “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *arXiv preprint arXiv:1408.5093* (2014).

[21] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. “Perceptual losses for real-time style transfer and super-resolution”. In: *European Conference on Computer Vision*. 2016.

[22] Alexia Jolicoeur-Martineau. “The relativistic generator: a key element missing from standard GAN”. In: *ArXiv* (Sept. 2018).

[23] Animesh Karnewar and Oliver Wang. “MSG-GAN - Multi-Scale Gradients for Generative Adversarial Networks”. In: *ArXiv* (Nov. 2019).

[24] Tero Karra et al. “Progressive Growing of GANs for Improved Quality, Stability and Variation”. In: *ICLR* (Feb. 2018).

[25] Tero Karras et al. “Analyzing and Improving the Image Quality of StyleGAN”. In: *ArXiv* (Dec. 2019).

[26] Diederik Kingma and Max Welling. “Auto-Encoding Variational Bayes”. In: *ICLR* (Dec. 2013).

[27] Diederik P. Kingma and Jimmy Lei Ba. “ADAM: A Method for Stochastic Optimization”. In: *NIPS* (Jan. 2017).

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *NIPS* (2012).

[29] Tuomas Kynkäänniemi et al. “Improved Precision and Recall Metric for Assessing Generative Models”. In: *NeurIPS* (Apr. 2019).

[30] Ming-Yu Liu, Thomas Breuel, and Jan Kautz. “Unsupervised Image-to-Image Translation Networks”. In: *NIPS* (2018).

[31] Ziwei Liu et al. “Deep Learning Face Attributes in the Wild”. In: *ICCV* (Dec. 2015).

[32] Mario Lucic et al. “Are GANs Create Equal? A Large-Scale Study”. In: *NIPS* (Oct. 2018).

[33] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. “Rectifier Nonlinearities Improve Neural Network Acoustic Models”. In: *JMLR* (2013).

[34] Liu Ming-YU and Tuzel Oncel. “Coupled Generative Adversarial Networks”. In: *NIPS* (2016).

[35] Mehdi Mirza and Simon Osindero. “Conditional Generative Adversarial Nets”. In: (Nov. 2014).

[36] Chris Olah, Vincent Dumoulin, and Augustus Odena. *Deconvolution and Checkerboard Artifacts*. https://distill.pub/2016/deconv-checkerboard/. 2016.

[37] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks". In: *ICLR* (Jan. 2016).

[38] Cédric Villani. *Optimal transport, old and new*. https://ljk.imag.fr/membres/Emmanuel.Maitre/lib/exe/fetch.php?media=b07.stflour.pdf. Springer, 2006.

[39] Ting-Chun Wang et al. "High-Resolution Image Synthesis and Semantic Manipulation with Conditional GANs". In: *IEEE CVPR* (2018).

[40] Lilian Weng. *From GAN to WGAN*. https://lilianweng.github.io/lil-log/2017/08/20/from-GAN-to-WGAN.html. 2017.

[41] Y. Bengio Y. LeCun L. Bottou and P. Haffner. "Gradient-based learning applied to document recognition". In: *IEEE* (1998).

[42] Zili Yi et al. "DualGAN: Unsupervised Dual Learning for Image-To-Image Translation". In: *ICCV* (Apr. 2017).

[43] Richard Zhang et al. "The Unreasonable Effectiveness of Deep Features as a Perceptual Metric". In: *IEEE* (Apr. 2018).

[44] Jun-Yan Zhu et al. "Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks". In: *ICCV* (2017).