rijksuniversiteit groningen / faculty of mathematics and natural sciences / artificial intelligence

Department of Artificial Intelligence
University of Groningen, The Netherlands

Data Set Extension with Generative Adversarial Nets

Master's Thesis

Luuk Boulogne
S2366681

Primary supervisor:	Dr. M.A. Wiering
Secondary supervisor:	K. Dijkstra, MSc.

Groningen, Friday 20th July, 2018

Abstract

This thesis focuses on supplementing data sets with data of absent classes by using other, similar data sets in which these classes are represented. The data is generated using Generative Adversarial Nets (GANs) trained on the CelebA and MNIST data sets. In particular, this thesis involves Coupled GANs (CoGANs), Auxiliary Classifier GANs (AC-GANs) and a novel combination of the two, Coupled Auxiliary Classifier GANs (CoAC-GANs). The abilities of these GANs to generate image data of domain-class combinations that were removed from the training data are compared. Classifiers are trained on the generated data to investigate the usefulness of the generated data for data set extension. The results show that AC-GANs and CoAC-GANs can be used successfully to generate labeled data from domain-class combinations that are absent from the training data. Furthermore, they suggest that the preference for one of the two types of generative models depends on training set characteristics. Classifiers trained on the generated data can accurately classify unseen data from the missing domain-class combinations.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Background

Machine learning (ML) is the science in which computers, data, and statistical methods are used to optimize certain criteria. It is a vividly alive research field that has had many break-throughs in recent years. The performance measures ML algorithms optimize are diverse, which allows them to be used to tackle different types of problems. In reinforcement learning, algorithms learn to complete tasks by optimizing some reward function. In supervised learning, they learn to label examples as accurately as possible and in unsupervised learning, they learn to find hidden structures in data.

Whichever criterion is optimized, ML algorithms need data with some structure to be trained. For unsupervised learning, any data with an underlying structure can be used. For reinforcement learning, the data consists of experiences that take the shape of past examples and the rewards they resulted in. For supervised learning the data consists of labeled examples.

The solutions ML algorithms find are often regarded as black boxes, since they are incomprehensible to humans. Generally, problems should therefore only be solved with ML when other solutions are too challenging or more expensive to develop. Because of this, ML is often used to tackle highly complex problems using sophisticated models. Large amounts of high quality training data are vital for these models to perform well. Many of the great successes in machine learning were only possible because of the large amount of structured data that has become available over the past decades. Using more training data does not only result in a better classification accuracy, but also helps to reduce overfitting on specifics of the training data in complex ML models [31]. However, collecting data is often expensive and/or labour intensive.

1.1.1 Generative models

A currently popular area of research within machine learning is data generation. This area concerns itself with generating new data for a distribution of interest. Different methods for data generation exist to date.

Variational Auto-Encoder

One of these data generation methods is the Variational Auto-Encoder (VAE) [29]. Like regular auto-encoders, VAEs are trained to first encode data to points in a lower dimensional latent space and then decode this point back to the original data. This restricts them in such a way that most of the information from the original data is retained in the latent code. Different from regular auto-encoders, the encoder part of a VAE is not directly trained to generate latent codes. Instead, it is trained to generate the means and standard deviations of a multivariate Gaussian model from which the latent code is sampled. After the VAE is trained, new data can be generated by inputting samples of that Gaussian model to the decoder. Improvements on VAEs have also been made (e.g. [57]).

Generative adversarial net

Another relatively recently developed type of generative model is the Generative Adversarial Net [17] (GAN). A GAN consists of a generator and a discriminator that are trained in a competitive setting. For a data generation task, the generator is trained to produce realistic fake data from some distribution of interest in order to confuse the discriminator. The discriminator is trained to discriminate between real data and data produced by the generator. By iteratively updating both the generator and the discriminator, an equilibrium should be reached in which the discriminator is no longer able to learn the differences between real data and the data produced by the generator. Although, at this point, the discriminator is no longer necessary for data generation, during the training process it has learned characteristics of the real data.

1.1.2 GAN research

Improved training

A difficulty for early GANs was producing high resolution, high quality images. One approach partially overcame this problem by generating images in a course-to-fine fashion [10]. In this work, a sequential approach based on the Laplacian pyramid is used. A first generator builds a low resolution image from noise and subsequent generators each improve on the previous stage by outputting a higher resolution difference image. For this, they use the upsampled image outputted at the previous stage as input. A large improvement has also been made by using a two-stage approach where the noise distribution from which the input for the generator is sampled is learned from an embedding of a textual description that the generated image should match [65].

An observed downside of GANs in their original form is that they exhibit massive training instabilities [2, 51, 55]. Therefore, the training dynamics of GANs have been theoretically investigated [2]. Architectural constraints have been proposed that are aimed at stabilizing training for deep convolutional GANs [51].

A well known failure mode for GANs is when they learn to only produce part of the desired distribution. The problem of having a generator map all its inputs to (a) single (mode of) output is called mode-collapsing. Both the training instability and the mode-collapsing problems have been partially solved by keeping to specific architectural features and training procedures [55]. Wasserstein GANs (WGAN) [3], that minimize the Earth Mover (EM) distance between the generated and real distributions instead of the Kullback-Leiber divergence as the original GANs do, have also shown to counter training instability and mode-collapsing. Next to this, the loss in the form of the EM distance of WGANs correlates well with the observed quality of generated images, which is useful for debugging and hyperparameter searches. The original WGAN suffers from optimization difficulties, but an improved version of the WGAN exhibits state-of-the-art performance by generating realistic looking images [18].

Class conditional information

Multiple ways have been proposed to turn the GAN architecture into a conditional model [41, 7, 48, 43] in order to allow the generator to generate specifically labeled data. The first GAN extension of this type is the conditional GAN [41] (cGAN). With this extension, conditional variables are used as input to both the generator and the discriminator besides their standard input. Because of this, the generator needs to produce class conditional images to fool the discriminator.

Instead of presenting the conditional variable to the discriminator by concatenating it directly to its input, the variable has also been presented at a later stage in the discriminator's processing by concatenating it to features that the discriminator extracts from its input [43].

Conditional information can also be incorporated into the discriminator by taking the inner product between an embedding of the variable and high level features extracted from the input

[43]. This method is referred to here as projection GAN (pGAN). The method has a strong theoretical foundation and its assumptions hold for real-world class-conditional image generation and super-resolution image generation tasks.

The conditional variable can also be incorporated into the training process by adding it into the performance criterion in other ways. This is done in Triple GANs [7] by adding a classifier as a third player to the cGAN optimization game. As with the regular cGAN, the generator and discriminator receive the conditional variable as input and are trained in an adversarial setting. The classifier is tasked to predict the conditional variable for real data and is, like the generator, trained together with the discriminator in an adversarial setting as well. The framework was shown to achieve great performance in semi-supervised learning.

With Auxiliary Classifier GANs [48] (AC-GANs) the conditional variable is incorporated into the loss function in another way. Like cGANs, the generator is presented with a conditional label during training, but with AC-GANs the discriminator is not. Instead, it is tasked to predict the label for both real and fake data. The generator and discriminator are both optimized to maximize the classification accuracy of the discriminator.

It was shown that the data produced with pGANs is more similar to real data than the data produced by cGANs and AC-GANs [43].

InfoGAN [5] took using conditional variables in GANs a step further by making the GAN itself discover what continuous and discrete input variables should represent by maximizing the mutual information between observations and these latent variables.

Architectural changes

GANs have also been used for image-to-image translation in e.g. [24, 26, 6]. Instead of generating new images from only noise and optionally a class label, such models alter existing images from one domain so that they seem to originate from another domain or they alter existing images of some class to appear as if they have another class label.

E.g. DiscoGAN [26] bijectively maps some domain of images to another domain. It was found that by doing so, DiscoGAN can find cross-domain relations and can be used to transfer styles from one domain to another. Also, forcing the GAN to represent a one-to-one mapping helps to prevent mode-collapsing.

Furthermore, StarGAN [6] is able to translate images in some domain to appear to have a class label for which no data in that domain is available by using additional data from another domain in which the particular class label is represented.

CoGAN [37] consists of multiple GANs that all share the weights in the first layers of the generators and last layers of the discriminators. These layers are the ones that learn the high-level semantics of images. By presenting a single input to coupled generators, CoGAN is able to generate corresponding images laying in multiple domains. Furthermore, CoGAN achieved excellent results in unsupervised domain adaptation, which concerns adapting a classifier from one domain to another, where there are no labels available in the new domain.

Other uses

GANs have been used for image inpainting of faces and cars [64]. Here, part of an image is missing and the task is to realistically reconstruct the original image. Similar work has been done without GANs by sequentially predicting pixels of images of more complex scenes [49]. Also, video with static background has been produced using spatio-temporal convolutional GANs [62]. Although promising results have been obtained, the current results are not yet life-like.

The concept of imposing a prior distribution in an auto-encoder (like with VAEs) has also been implemented by partially training the encoder as the generator part of a GAN [40]. Resulting architectures show competitive performance in semi-supervised learning.

1.2 Research questions

Any image data set only covers a fixed domain. This severely limits the abilities of classifiers trained on them. E.g. classifiers are unable to classify classes that do not exist in their training sets. They also often lack in accuracy when tested on data sets different from its training set with an overlapping set of classes, since the data set specifics such as image style are rarely identical to the specifics of the training set.

Domain adaptation is the research area devoted to solve the problem of classifying data from some domain $\mathcal{C}$ where no class labels are available by using data from a different domain $\mathcal{D}$ in which labeled data are available. Similar to domain adaptation this work involves using data from some domain to expand classifier capabilities in another. We consider the setting with two domains of labeled data, where in one of the domains, for one or more classes, no data are available.

More specifically, consider image domains $\mathcal{A}$ and $\mathcal{B}$ that produce the samples in data sets A and B respectively. Furthermore samples in A have classes from set C_A and samples in B have classes from set C_B. Let there be a class c such that $c \in C_B$, but $c \notin C_A$, and furthermore $C_A \subset C_B$. From here on, the set of samples in domain $\mathcal{A}$ of class c are denoted by $\mathcal{A}^c$. For this work, Extra Domain Data Generation (EDDG) is defined as the problem of generating samples from $\mathcal{A}^c$. An example of this is generating fives in the style of A if A and B contain images of digits, with B containing images of fives and A not containing images of fives.

The goal of this thesis is to tackle EDDG with the use of generative models. It is aimed to answer the following research questions.

1. Can original data be generated that appears to originate from some domain, but is of a class of which there is no real data available for that domain?

2. Is it possible to use this data to train machine learning models which classify data from some domain where for some class(es) all data is missing in the training data?

To answer these questions, GANs are trained on the CelebA [66] data set and a combination of the MNIST [33] and MNIST-edge [37] data sets to generate original images. CoGANs, AC-GANs and a novel combination of CoGANs and AC-GANs are trained to generate data from domain-class combinations that are not present in their training sets. This work also shows that using the data generated by these models, classifiers can be trained that are able to accurately classify this missing data.

1.3 Outline

All generators and discriminators that make up the GANs trained for this thesis are implemented as Neural Networks (NNs). Therefore, chapter 2 describes this type of model. After a short introduction to NNs, it constructively explains Adaptive moment estimation (Adam), which is the gradient descent method this work uses to optimize NNs. The chapter then describes the processing components of which the NNs used in this work consist. These components are activation functions of which different variants of the Rectified Linear Unit (ReLU) are highlighted, regular and transposed convolution layers, and two layer input normalization techniques.

Chapter 3 describes GANs. It explains formally how they minimize the distance between a data distribution generated by a generator and a target data distribution. In order to do so, it introduces the Jensen-Shannon divergence, which is the distribution similarity measure that GANs optimize. The chapter then proceeds to describe the training instabilities regularly observed in GANs and the different variants of GANs that this work builds upon, which include DCGANs, AC-GANs, CoGANs and WGANs. In order to obtain a thorough understanding of the latter type of GAN, Lipschitz continuity and the EM distance, which is the distribution similarity measure WGANs optimize, are also explained. Lastly, the chapter describes different methods that can be used to measure GAN performance qualitatively.

Consecutively, chapter 4 discusses the suitability of CoGANs and AC-GANs for EDDG. It continues by introducing Coupled Auxiliary Classifier GANs (CoAC-GANs), which is a novel combination of these two GAN variants. The usefulness of CoAC-GANs for EDDG is also discussed here.

In chapter 5, experiments designed to test how these methods perform at EDDG are described. It starts of by describing the MNIST [34], MNIST-edge [37] and CelebA [66] data sets that represent the data distributions of interest for this work. The chapter describes experiments for EDDG, as well as Extra Domain Classification (EDC). It also states the implementation details of the trained models.

The results of the experiments on EDDG and EDC are reported and discussed in chapter 6. In chapter 7 conclusions are drawn and some possibilities for future work are suggested.

Chapter 2

Neural Networks

Consider some data in which an underlying relation is present that can be described as $y = g(x)$. Here, x and y are some variables that the particular data contains and the function g describes the relation between these variables.

For e.g. classification and regression problems, it is useful to have access to g or to a function that approximates g so that y can be inferred or approximated when a new value for x is encountered. However, many collections of data contain variables that depend on each other in complex ways, while the functions that describe such relations are unknown and cannot be obtained using insightful modeling. In these cases, it is favourable to use types of models that lack introspection, but can be tuned to represent arbitrary functions, i.e. that are suitable for universal approximation [1].

A popular type of universal function approximator is the artificial Neural Network (NN) [1, 16]. Its structure holds a weak resemblance to that of the brain. NNs consist of individual processing units called neurons, each of which holds an activation value. The activation values of neurons influence each other through directed weighted connections between neurons.

A simple form of an NN is the perceptron [1]. A perceptron produces an output $y \in \mathbb{R}$ from an input vector $\mathbf{x} \in \mathbb{R}^n$. Its parameters are a weight vector $\mathbf{w} \in \mathbb{R}^n$ and a bias b. The output of the perceptron is calculated as $y = \mathbf{w}^\top \mathbf{x} + b$.

When a perceptron instead produces an output vector $\mathbf{h} \in \mathbb{R}^m$, it can be described by the matrix multiplication $\mathbf{h} = W\mathbf{x}$ where the columns of the $m \times n$ matrix W are the different weight vectors for each element in $\mathbf{h}$. A perceptron can be regarded as a single-layer NN. Multi-layer NN architectures can be created by using the output of one layer as the input for another. Such multi-layer NNs are called feed-forward NNs, because of their uni-directional processing flow. While perceptrons with dense connectivity are commonly used as NN layers, different types of layers exist.

NNs are not restricted to having a feed-forward architecture, but since such different architectures are not used in this work, they are not described here.

2.1 Gradient descent

Training a model usually involves minimizing the expected value of a loss function $f(\theta)$ where θ is the vector containing all learnable parameters of the model that is being trained [16]. When $f(\theta)$ is differentiable with respect to all parameters θ, this model can be trained with gradient descent based training algorithms.

Gradient descent algorithms use the gradient of $f(\theta)$ with respect to θ, $\nabla f(\theta)$, to gradually increase a models performance. They do so by sequentially updating θ. Although θ and with that $f(\theta)$ constantly change during training, for regular gradient descent, $f(\theta)$ remains static when evaluated for the same value for θ. This form of gradient descent corresponds to full batch learning. With full batch learning, f is not only a function of θ, but also of a set of training data for which f is optimized.

It is also possible for $f(\theta)$ to change at each update, even for the same value of θ. In this case, one speaks of stochastic gradient descent (SGD). For SGD, f at update step t is denoted as f_t. SGD corresponds to mini-batch learning, where at each time step a different batch of samples of the training data is presented to the model that is being trained. Consequently, for minibatch learning, each individual update is less time-consuming. Even though a mini-batch is an imperfect representation of the complete data set, mini-batch learning still converges, because most of the time it provides sufficient information to move θ in an admissible direction. f is minimized, because with a small learning rate, gradients average out over successive mini-batches [59].

Typically the SGD update is realized by nudging θ in the direction opposite to $\nabla f_t(\theta)$, which corresponds to directly changing the position of θ in parameter space [53]:

$$\Delta\theta_t = -\eta \nabla f_t(\theta_t) \tag{2.1}$$

and

$$\theta_{t+1} = \theta_t + \Delta\theta_t.$$

Here θ_t denotes θ at time step t and η denotes the learning rate.

A useful aid in performing SGD when θ comprises of the learnable parameters of an NN is the backpropagation algorithm [53]. In this algorithm, the chain rule is performed sequentially in order to compute $\nabla f_t(\theta)$.

2.1.1 Momentum

With regular SGD, it is possible that $\nabla f_t(\theta)$ changes dramatically for subsequent times t. This can result in large oscillations of the values in θ over time, which in turn slows down convergence [1]. To counter this, a momentum term can be added to equation 2.1. SGD with momentum corresponds more to nudging the velocity of θ in the direction opposite to $\nabla f_t(\theta)$, than to directly changing its position [53]:

$$\Delta\theta_t = \alpha\Delta\theta_{t-1} - \eta\nabla f_t(\theta_t).$$

Here, α is generally taken between 0.5 and 1.0 [1].

2.1.2 Root mean squared backpropagation

When performing full batch gradient descent, learning can become more robust by ignoring the magnitude of the partial derivative to f with respect to each learnable parameter and by only regarding their sign instead. With Resilient backpropagation [52] (Rprop), the size of a parameter update depends on the agreement of the signs of the correponding partial derivatives over multiple time steps. At any time step, this size becomes larger when the current and previous gradient signs agree for the regarded parameter and smaller if they do not. Although this is a robust algorithm for full-batch learning, it does not work when training with mini-batches [59]. This is attributed to the stochastic nature of f_t that results from using mini-batches.

Root Mean Squared backropagation (RMSprop) [59] is the mini-batch version of Rprop. It solves the problem stated above by normalizing the gradient for each of the learnable parameters at each update, while also keeping the normalization term similar in subsequent updates. It accomplishes this by dividing the gradient for each learnable parameter by the accumulated magnitude of the gradient, which improves learning [59]:

$$\Delta\theta_t = -\eta\frac{\nabla f_t(\theta_t)}{v_t},$$

where

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla f_t(\theta_t))^2. \tag{2.2}$$

Here β_2 is a hyperparameter and the division and squaring operations denote their elementwise versions.

2.1.3 Adaptive moment estimation

Adaptive moment estimation (Adam) [28] is an SGD technique that keeps track of and uses bias-corrected lower-order moments. These lower-order moments are similar to the estimates used in SGD with momentum and RMSprop. Adam has three hyperparameters. These are the learning rate η and the decay rates for the first- and second-order moment estimates, which are denoted respectively by $\beta_1, \beta_2 \in [0, 1)$. The corresponding moments are given in equations 2.3 and 2.2.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla f_t(\theta_t) \tag{2.3}$$

Since m_t and v_t are initialized as vectors of zeros, they are biased towards zero. This is especially the case when t or their respective decay rates are small. The algorithm therefore uses bias-corrected variants of these moments:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

and

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}.$$

The update rule for Adam becomes:

$$\Delta\theta_t = -\eta \frac{\hat{m}_t}{\hat{v}_t + \epsilon}.$$

Here, ϵ is a small constant that avoids division by zero.

2.2 Activation functions

Activation functions are non-linear functions that are applied element-wise to the activation values of neurons in hidden layers. For multi-layer NN architectures, especially straightforward feed-forward ones, activation functions are often the NN's only source of non-linearity. Because of this, they are indispensable for NNs to have sufficient expressive power to tackle challenging problems. Since the optimization of an NN often involves computing its derivative with respect to every learnable parameter, many activation functions have the property that the derivative with respect to their input is easy to compute. Some popular activation functions also used in this work are the sigmoid function $f(x) = \frac{1}{1+e^{-x}}$ and hyperbolic tangent (tanh). A disadvantage of these functions is that they saturate for large absolute values of x. When x lies in a saturated regime the gradient vanishes, which slows down learning.

2.2.1 ReLU family

Another popular but also more modern activation function is the Rectified Linear Unit [44] (ReLU). It is defined as $f(x) = \max(0, x)$. Different from the sigmoid and tanh functions, it does not saturate for large values of x. Because of this, using it results in faster learning of deep NNs that are trained with gradient descent [31].

A disadvantage of the ReLU is that it does not provide any gradient when $x < 0$. A version of the ReLU that does not have this problem is the LeakyReLU [39], which is defined as

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \frac{1}{\alpha}x, & \text{otherwise,} \end{cases}$$

where α is a positive constant that can differ for different layers in an NN. In [39], α was set to the relatively large default value of 100 which caused the LeakyReLU to be very similar to the original ReLU. It was found that training NNs with smaller values for α such as 5.5 can increase classification accuracy [63].

Another version of the ReLU is the Parametrized ReLU [21] (PReLU). The PReLU has the same definition as the LeakyReLU. However, when using PReLU, α is no longer set to a fixed value, but treated as a learnable parameter instead. By letting the NNs learn the shape of their activation functions in this way, NNs surpassed human level performance on image classification [21].

2.3 Convolutional neural networks

For some processing tasks, the input to an NN consists of multiple values that represent the same feature or variable, but in a different position or at a different time. Such values can be the intensities of the color red of the pixels in an image or the air pressure deviations that make up an audio fragment. The spatial (or temporal) relations between these values are not intrinsically captured by the fully connected layers in an NN.

These relations are intrinsically captured by convolution (conv) layers [34]. Conv layers are therefore commonly used in NNs for a wide range of image processing tasks [16]. The convolution in conv layers is often implemented as a cross-correlation [16], which is the same as a convolution with a transposed kernel. The output of the conv layer is a series of feature maps in which the spatial relations between the different values are maintained. A feature map is computed as $B_q = A * H_q$, where $*$ denotes cross-correlation and q denotes the feature map index, A is the matrix that represents the input for the conv layer and H_i is a learnable kernel. Conv layers thus efficiently make use of the spatial relations in their input, by extracting the same features from different input areas.

Rather than as a cross-correlation, a conv layer can also be viewed as a sparse fully connected layer in which some weights are shared [12]. Instead of directly computing the layer output B as described above, A and H can be rolled out or flattened to become the vectors $\mathbf{a}$ and $\mathbf{h}$ respectively. Taking $\mathbf{b}$ to be the flattened version of B, the cross-correlation of A and H can then be implemented by computing $\mathbf{b} = W\mathbf{a}$. The sparse matrix W is thus defined in such a way that multiplying with it mimics cross-correlation. More specifically, if the entries in H and A that correspond to $\mathbf{h}_k$ and $\mathbf{a}_j$ respectively would be multiplied with each other in the process of computing the entry of B that corresponds to $\mathbf{b}_i$, then $W_{ij} = \mathbf{h}_k$. Otherwise, $W_{ij} = 0$.

Conv layers are often followed by pooling layers [34] in which the feature maps produced by the preceding layer are down-sampled. Convolution can also be implemented in such a way that the kernel shifts a fixed number of pixels $s \geq 1$ when it is convolved with the layer input. This is called strided convolution or convolution with a stride of s [16]. When taking $s > 1$, the convolution produces a smaller feature map than regular convolution. Therefore, strided convolution can also be used as a form of down-sampling in CNNs. When the conv layer input is multidimensional, e.g. a 2D image, different strides can be taken for the different directions (horizontal and vertical) of the convolution. Consequently the down-sampling for these directions is then different as well.

2.3.1 Transposed convolution layers

Performing backpropagation through a conv layer involves multiplying with $W^\top$, which is the transpose of the matrix W that describes the convolution that this layer implements in the forward pass [16]. The transposed convolution (TC) layer [38, 12] is defined as a regular conv layer in which W is swapped with $W^\top$ in the forward pass. Therefore, backpropagation through a TC layer now involves multiplying with $(W^\top)^\top = W$.

The forward pass of a TC layer can thus be implemented by computing its output $\mathbf{a} = W^\top\mathbf{b}$, where $\mathbf{b}$ is the input of the TC layer and W is defined as in section 2.3. TC layers were originally dubbed deconvolution layers [38]. Since, with respect to a regular conv layer, in a TC layer W is swapped with $W^\top$ in the forward pass, the dimensions of the input and output of a TC layer are also swapped. However, $W^\top$ is not necessarily equal to the inverse of W. In other words, $\mathbf{a} = W^\top\mathbf{b}$ does not imply that $\mathbf{b} = W\mathbf{a}$, which means that TC layers do not implement deconvolution. The name deconvolution layer is therefore depreciated [12].

TC layers should thus not be regarded as the inverse of regular conv layers. Instead, they could be viewed as a special case of regular conv layers. This is because every transposed convolution can also be implemented as a regular convolution [12]. Implementing TC layers in this way however needs zero padding around the input of the convolution, which is unnecessary when implementing the TC layer with a transposed convolution.

Fractionally strided convolution

The similarity between regular conv layers and TC layers does not stop at the vanilla conv layer variant. When W is defined for a strided convolution, again its transpose $W^\top$ can be used to define a corresponding TC. Such a TC can also be viewed as a special case of a regular strided convolution [12]: A convolution with a stride s corresponds to a TC that can be implemented as a convolution with a fractional stride of $\frac{1}{s}$. With such a fractional stride, before the convolution is performed, rows and columns of zeros are added in-between each adjacent pair of rows and columns of the input s. Although this is a useful way to gain insight in the way strided TC layers operate, such an implementation of TC layers would add many unnecessary multiplications with zero.

Because strided TC layers can be implemented in the way described above, TC layers have also been referred to as fractionally strided conv layers [12]. Similar to the way strided conv layers can be used to let NNs learn their own downsampling, TC layers can be used to let NNs learn their own upsampling. This makes them useful for e.g. image generators in Generative Adversarial Nets [51].

2.4 Layer input normalization

2.4.1 Batch normalization

Formally, an update of a weight of an NN trained with a gradient descent optimization technique assumes the NN to be partially static. More specifically, when updating some specific weight of the NN, it assumes that all other weights that had an influence on the activation value of the neuron with which the regarding weight was multiplied during the forward pass and all weights that the regarding weight has an influence on remain unchanged. Since all weights are updated simultaneously in gradient descent algorithms used for training NNs, this assumption does not hold. However, gradient descent can still be used for training NNs when the assumption is approximated by taking a sufficiently small learning rate.

The quality of the computed gradients is still impaired by the fact that a small change in one of the first layers can have a large impact on layers further up in the forward pass, especially when the NN is deep. Batch normalization [23] (BN) alleviates this problem by controlling the mean and standard deviation of the distribution of inputs for the layer before which it is applied. When applying it before saturating nonlinearities, it can help by transforming their input to avoid saturated modes and the vanishing gradient problem.

Normalizing layer input

BN firstly controls the distribution of inputs to a layer by making it invariant to any shifting or scaling caused by preceding layers. Let $x^{(k)}$ be the data in the k^{th} dimension of the batch of inputs x. At each batch update each dimension, or analogously k^{th} activation value of the input to the regarded layer, is normalized to have a mean of zero and a standard deviation of one:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}] + \epsilon}}. \tag{2.4}$$

Here ϵ is a small constant that avoids division by zero. Note that for hidden and output layers in an NN, the input distribution of a layer, and with that its mean and variance, changes at each update, since the weights of preceding layers alter at each update as well.

BN heavily relies on the assumption that all input batches are sampled uniformly from the layer input distribution. If they are not sampled in this way, $\mathbb{E}[x^{(k)}]$ and $\text{Var}[x^{(k)}]$ will be poor estimates of the mean and variance of the actual distribution of data in the k^{th} dimension of the input to the regarded layer. Although this assumption can be easily met when sampling from the input distribution during training, it does not hold when purposefully crafted batches are presented to the NN during testing. To circumvent this problem, the running averages of $\mathbb{E}[x^{(k)}]$ and $\text{Var}[x^{(k)}]$ that are accumulated during training are used when testing the NN.

Transforming the normalized input

In order to always allow the BN transform to represent the identity transform, for each dimension k the normalized values in $\hat{x}$ are scaled and shifted by learned parameters $\gamma^{(k)}$ and $\beta^{(k)}$ respectively:

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}. \tag{2.5}$$

At first glance, it might seem useless to first normalize the activation values and then introduce parameters that allow these normalized values to be scaled and shifted. The effectiveness of BN lies in the way it reparameterizes NNs [16]. Instead of letting the mean and variance of the input to a layer depend on the transformations that any preceding layers implement, they solely depend on two learned parameters.

When applying BN after a conv layer, the input is not normalized, scaled and shifted per activation value, but per feature map instead. This is done to obey the spatial relations present in the feature maps produced by the conv layer.

2.4.2 Instance normalization

Instance normalization is a weight normalization method that has been shown to improve the results in image style transfer [11]. The method is similar to BN. In fact, it only differs from BN in that it normalizes over single input instances instead of complete batches of input. The normalization process is thus described by equations 2.4 and 2.5, where $x^{(k)}$, $\hat{x}^{(k)}$ and $y^{(k)}$ denote the data in the k^{th} dimension of respectively the original input, normalized input and transformed input per input instance instead of the data in the k^{th} dimension per batch of inputs.

Chapter 3

Generative Adversarial Nets

Generative Adversarial Nets (GANs) [17] have been proposed as a method for data generation. The method uses a competitive setting that is depicted in figure 3.1. The generator and discriminator in GANs are often implemented as NNs. More specifically, for image generation tasks, CNNs are a popular choice (e.g. [37, 41, 43, 6, 48, 3, 19]), although with capsule networks [54] promising results have also been obtained [50, 60].

The generator obtains the ability to generate data by learning a mapping from some distribution $p_{\mathbf{z}}$ to the distribution of interest p_{data}. This mapping is learned by the generator that gets as input a sample $\mathbf{z} \sim p_{\mathbf{z}}$ and outputs some fake data point $G(\mathbf{z})$. We denote the distribution of $G(\mathbf{z})$ as p_G.

To learn this mapping, a discriminator is trained together with the generator. The discriminator learns to predict the probability that its input $\mathbf{x}$ is sampled from the real data instead of it being produced by the generator. It learns this by maximizing its output when $\mathbf{x} \sim p_{\text{data}}$ and minimizing its output when $\mathbf{x} \sim p_G$. In contrast, the generator learns to produce samples $\mathbf{x} \sim p_G$ that maximize the output of the discriminator.

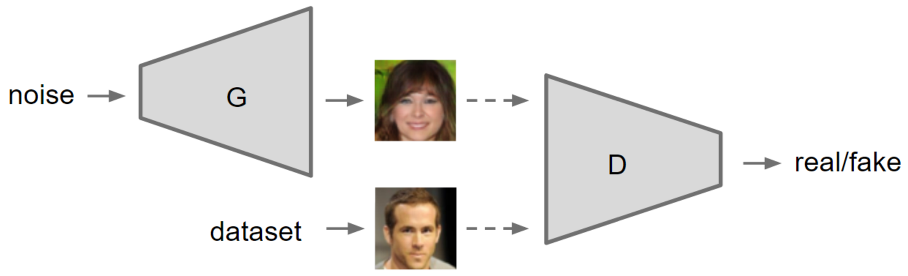


Figure 3.1: A simple graphical representation of the GAN setting. The generator has the task of generating convincing fake data from random noise. The discriminator gets as input either fake or real data and has to determine whether its input is real or fake.

3.1 Formal objective

The GAN objectives are captured in the value function:

$$V(D,G) = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[\log(1 - D(G(\mathbf{z})))]. \tag{3.1}$$

The discriminator needs to maximize this function, while the generator needs to minimize its right term. The complete training process is described by playing the following min-max game:

$$\min_G \max_D V(D,G) \tag{3.2}$$

3.1.1 Information theory

In order to understand why the min-max game defined in section 3.1 is played to optimize GANs, some knowledge about information theory is required. This section constructively introduces Jensen-Shannon (JS) divergence [36], which plays a large role in the optimization of the generator.

Self-information

Self-information denotes how much information is conveyed by finding out that an event occurs [16]. Self-information follows two intuitive notions. The first one is that finding out that likely events occur conveys little information, while finding out that unlikely events occur conveys much information. Secondly, self-information is also in accordance with the fact that finding out that two independent events occur together conveys as much information as the sum of finding out that those events occur separately. Following these notions, given some variable X and a value x for that variable, the amount of self-information present in the event that $X = x$ is:

$$I(x) = -\log P(x),$$

where $P(x) = Pr(X = x)$.

Entropy

The entropy of a random variable measures the uncertainty of that variable [8]. For the variable X, it does so by adding up the self-information for all values X can take, weighted by the probability that X takes those values. The entropy of a discrete random variable X is:

$$H(X) = -\sum_{x \in \mathcal{X}} P(x) \log P(x). \tag{3.3}$$

Here $\mathcal{X}$ denotes the distribution from which the value x for X is sampled. $H(X)$ is often denoted as $H(P)$ [8, 16].

Entropy can be better understood by considering trying to construct messages that are on average as short as possible, but could still be used to convey the value of a variable [8, 1]. The solution of this problem consists of finding the optimal way to encode the variable. In order to make the encoding as optimal as possible, the message length is allowed to vary for different values of X, which makes it possible to use shorter codes for more frequent values and longer codes for infrequent ones. The variable could e.g. be encoded using bits. In this case, the entropy denotes the minimum average message length in bits that that can be used so that X is encoded. Using bits as a unit of information for entropy corresponds to setting the logarithm in equation 3.3 to base 2.

Note that the definition in equation 3.3 can be easily adjusted to work with logarithms of other bases [8]. The base can be changed from a to b by multiplying the entropy with $\log_b a$. Changing the base then corresponds to scaling the unit of information in which the entropy is measured. Some common units of information [16] are bits, which have also been denoted as shannons, and nats, which correspond to the definition of entropy using the natural logarithm.

Equation 3.3 can be rewritten as

$$H(P) = \mathbb{E}_{X \sim P} \log P(x), \tag{3.4}$$

which generalizes to continuous variables as well [16].

Cross entropy

The cross entropy of P with respect to another probability distribution Q is defined as [16]:

$$H(P, Q) = -\mathbb{E}_{X \sim P} \log Q(x).$$

This is the average length (e.g. in bits for a logarithm with base 2) of messages that convey the value of a variable that has the distribution P, when the encoding of that variable is done using the code that would be optimal to convey the value of a variable that has the distribution Q.

Kullback-Leibler divergence

Intuitively, when P is similar to Q, $H(P,Q)$ is close to $H(P,P)$ and vice versa when P and Q are dissimilar. Because of this, a similarity measure for P and Q can be defined as the average inefficiency of encoding the value of a variable that has the distribution P with the optimal code for a variable that has the distribution Q [8, 16]. This average inefficiency is called the KL divergence [32]. It can be stated formally as:

$$KL(P\|Q) = \mathbb{E}_{X\sim P}[\log P(x) - \log Q(x)] = \mathbb{E}_{X\sim P}\log\frac{P(x)}{Q(x)}. \tag{3.5}$$

The KL divergence has some properties that make it useful as a similarity measure [8, 16]. Because KL divergence measures the encoding inefficiency of some code with respect to the optimal code, it can never be negative [8, 16]. Also, it is always nonzero except when $P = Q$ in the discrete case [8, 16] and it is zero when P and Q are equal almost everywhere in the continuous case [16]. However, it is not necessarily true that $KL(P\|Q) = KL(Q\|P)$ [8, 16]. Because of this, even though the KL divergence is an intuitive measure for the similarity of probability distributions, it is not a proper distance measure. Also, with using the KL divergence as a distance measure, there is no upper bound to the dissimilarity between distributions [36], which can also be undesirable.

It should be noted that optimizing Q so that it minimizes $H(P,Q)$, is the same as optimizing it so that it minimizes $KL(P\|Q)$ [16]. This is true because the only difference between cross-entropy and KL divergence is the term $\log P(x)$ in equation 3.5, which is independent of Q.

Jensen-Shannon divergence

A variant on the KL divergence that is more suitable as a distance metric than the KL divergence is Jeffreys divergence [46], which is defined as

$$J(P,Q) = KL(P\|Q) + KL(Q\|P).$$

Although Jeffreys divergence is clearly symmetric, it still has no upper bound [46]. There exists a variant of the KL divergence that does have an upper bound. This version is called the K-divergence [36]. It is defined as

$$K(P\|Q) = KL\left(P\middle\|\frac{P+Q}{2}\right).$$

Intuitively, the K divergence is thus the average inefficiency of encoding the value of a variable that has the distribution P with the optimal code for a variable that has the distribution that is the average of P and Q.

The Jensen-Shannon (JS) divergence [36] is the symmetric variant of the K divergence similar to how Jeffreys divergence is the symmetric variant of the KL-divergence [46]:

$$JS(P,Q) = \frac{1}{2}\left(K(P\|Q) + K(Q\|P)\right).$$

The JS divergence thus keeps the useful properties for a distance measure of KL divergence, but it furthermore also has an upper bound and is symmetric [46].

3.1.2 Theoretical training objective of the generator

In [17] it is shown that when regarding a GAN with an unlimited capacity, for a fixed generator, the optimal discriminator is:

$$D^*_G(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_G(\mathbf{x})}. \tag{3.6}$$

As also shown in [17], imposing this on equation 3.1, one obtains a theoretical training criterion for the generator:

$$C(G) = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[\log D^*_G(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[\log(1 - D^*_G(G(\mathbf{z})))]. \tag{3.7}$$

It is shown in [17] that with some intermediate steps and by using equation 3.6 equation 3.7 can be rewritten as:

$$C(G) = -\log(4) + 2JS(p_G, p_{\text{data}}).$$

Therefore, with an optimal discriminator throughout the training process, the generator would be minimizing the JS divergence between p_G and p_{data}.

3.2 Training procedure

Algorithm 1 describes the regular training procedure for GANs [17] that can be used to play the min-max game described in section 3.1. Note that variants of SGD, such as Adam (see section 2.1.3), can also be used in this algorithm.

Algorithm 1 Training of GANs with stochastic gradient descent [17]. k is the number of times the discriminator is updated for each generator update, m is the mini-batch size used in SGD, w and θ are the parameters of the discriminator D and generator G respectively.

```
procedure TrainGAN
    for number of training iterations do
        initialize $w$ and $\theta$
        for $k$ steps do
            for $i = 1, ..., m$ do
                Sample a noise vector $\mathbf{z} \sim p_{\mathbf{z}}$
                Sample real data point $\mathbf{x} \sim p_{data}$
                $L_D^{(i)} \leftarrow -\log D(\mathbf{x}) - \log(1 - D(G(\mathbf{z}))$
            $w \leftarrow \text{SGD}\left(\nabla_w \frac{1}{m} \sum_{i=1}^m L_D^{(i)}\right)$
        Sample a mini-batch of $m$ samples $\{\mathbf{z}^{(1)}, ..., \mathbf{z}^{(m)}\} \sim p_{\mathbf{z}}$
        $\theta \leftarrow \text{SGD}\left(\nabla_\theta \frac{1}{m} \sum_{i=1}^m \log\left(1 - D\left(G\left(\mathbf{z}^{(i)}\right)\right)\right)\right)$
```

3.2.1 Instability

The training of regular GANs is found to be unstable [51, 55, 2]. An important source of this instability is the fact that the GAN min-max game involves a simultaneous optimization of two players with different objectives. This can cause the generator and discriminator to undo each other's progress during training, which can prevent an equilibrium from being reached [15, 55]. Here, the manner in which this type of instability most frequently occurs is discussed.

Mode collapse

The quality of samples generated by the generator depends on how well the min-max game described by equation 3.2 plays out. A well-known failure mode for this game is when the generator starts to produce samples that are similar to only a part of the real data distribution. This failure mode is called mode collapse [15]. Mode collapse can be investigated by regarding equation 3.2. This equation corresponds to GAN training where the generator tries to fool an optimal discriminator. This is also the assumption that underlies the theoretical training objective of the generator (see section 3.1.2). However, when using algorithm 1, this assumption does not necessarily hold, since the generator and discriminator perform gradient descent on their objective functions simultaneously. Instead, in some settings, the training process might be better described by the max-min game:

$$G^* = \max_D \min_G V(D, G),$$

where G^* is the optimal generator acquired by the training procedure [15]. In this game, G^* is trained to map its entire input domain to a single fake data sample [15].

Mode collapse can occur in varying degrees of severeness. When it only occurs to a mild extent, the training process is not severely hindered and thus still converges. However, the distribution that can be sampled from using the generator after training does not cover the entire real data distribution.

When severe cases of mode collapse occur, the generator starts to produce only few samples or even only a single sample [55, 51]. When this happens, the training process does not continue normally. Instead, the discriminator learns to only label the small diversity of samples generated by the generator as fake [55]. When the generator is updated, it then learns to map its input domain to a single or only a few points in its output space, since doing so has become an easy way to fool the now mode-collapsed discriminator. This repeating process in which the discriminator pushes the output of the generator around then prevents the min-max game from converging.

The mode-collapse scenario in which the generator learns to limit itself to generating only a few samples is common, but a complete collapse resulting in a generator that maps its entire input domain to only a single sample is rare [15]. In [15] it is also argued that GANs are prone to cover less modes than they are limited to.

Variational Auto-Encoders [29] (VAEs), which minimize the KL divergence between a real and fake data distribution, produce more blurry samples than GANs, but produce a better coverage of the target domain. It therefore used to be popular belief that training to minimize the JS divergence between the real data and the generated data results in sharp samples, but that this comes at the cost of mode collapse. In contrast, minimizing the KL divergence would cause the entire domain to be represented by the output samples, at the cost of these samples being blurry. This has been shown to be false through the introduction of f-GANs [47], which have been developed to allow GANs to minimize other f-divergences than the JS divergence. f-GANs that minimize the KL divergence between the real and generated data produce sharp samples, but such models still suffer from mode collapse [15].

3.3 Deep convolutional generative adversarial nets

Originally, GANs were trained to generate images and were implemented using multi-layer perceptrons [17]. In [51], some architectural constraints were found that result in stable GAN training. This family of architectures was developed for the generation of images. The generator and discriminator are both implemented as deep CNNs and architectures following the constraints are named Deep Convolutional GANs (DCGANs). The architectural constraints that define a DCGAN are the following:

- No pooling layers are used in the GAN. Instead, strided conv layers are used in the discriminator and fractionally strided conv layers are used in the generator (see section 2.3).
- Batch normalization is used in the generator and in the discriminator (see section 2.4.1).
- Fully connected hidden layers are removed for deeper architectures.
- Except for after the last layer, all activation functions in the generator are ReLUs. The last layer is followed by a tanh activation function.
- All activation functions in the discriminator are LeakyReLUs.

3.4 Wasserstein generative adversarial nets

As described in section 3.1.2, the original GAN min-max game is played to minimize the JS divergence. However, in some cases, the JS is non-continuous and fails to provide useful gradients, which causes it to fail to converge [3]. Because of this, Wasserstein GANs [3] (WGANs) were introduced. Instead of minimizing the JS divergence, this type of GAN minimizes the Earth Mover (EM) distance between two probability distributions. WGAN and its improved variant [19] improve upon the original GAN in multiple ways [3, 19] and are robust to mode collapse [3, 19]. This section first offers some theory on Lipschitz continuity and then formally introduces the EM distance. Using this theoretical basis, the section ends by explaining how WGANs can be trained.

3.4.1 Lipschitz continuity

One could rate some function f on its degree of continuity i.e. how sudden its output changes based on small changes in its input. This degree of continuity relates to the Lipschitz continuity [13]. A function f is said to be Lipschitz continuous on domain I with Lipschitz constant L_f if $L_f \geq 0$ exists s.t. for all $x_1, x_2 \in I$

$$\|f(x_1) - f(x_2)\| \leq L_f \|x_1 - x_2\|. \tag{3.8}$$

L_f thus denotes how much $f(x)$ can change under a small change of x. When a function is Lipschitz continuous with Lipschitz constant L_f it is said to be L_f-Lipschitz.

When a function satisfies equation 3.8 it is said to be Lipschitz or globally Lipschitz. A weaker variant of Lipschitz continuity is local Lipschitz continuity [56]. To define local Lipschitz continuity, first the neighborhood U of some point x_0 on I needs to be defined. If I lies in some space with dimensionality n, this neighborhood U is the n dimensional ball with center x_0. f is locally Lipschitz on I if for all $x_0 \in I$, there exists a neighborhood U of x_0 s.t. f is globally Lipschitz on U.

3.4.2 Earth mover distance

The EM distance has also been called the Wasserstein-1 distance or Kantorovich-Rubinstein distance [61]. It denotes the minimal cost of transforming one probability distribution p_1 into another p_2 and it can be explained intuitively when these probability distributions are regarded as heaps of earth with specific shapes. E.g. $p_1(x)$ then denotes the amount of earth in location x according to p_1. In order to transform the shape of one heap into the shape of the other, a transference plan γ is needed. $\gamma(x, y)$ then denotes the amount of earth that has to be moved from location x to y in order to transform the shape of p_1 to that of p_2. The cost of transporting one unit of earth from x to y equals the distance between x and y. The cost of transforming p_1 into p_2 thus depends on the optimality of γ. The EM distance is defined as the transport cost that corresponds to the optimal γ.

Formally, for probability distributions p_1 and p_2, the EM distance is [3, 61]

$$EM(p_1, p_2) = \inf_{\gamma \in \prod(p_1, p_2)} \mathbb{E}_{(x,y)\sim\gamma} \left[\|x - y\|\right]. \tag{3.9}$$

Here inf stands for the infimum or greatest lower bound. $\prod(p_1, p_2)$ denotes the set of all valid transference plans, which is the set of joint distributions with marginals p_1 and p_2.

While equation 3.9 offers an intuitive physical description of the EM distance, it cannot easily be minimized in a useful way for GANs. Through the Kantorovich-Rubinstein duality [3, 61] a definition of the EM distance that is useful for the problem at hand is obtained [3]. The Kantorovich-Rubinstein duality which is as follows:

$$EM(p_1, p_2) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x\sim p_1}[f(x)] - \mathbb{E}_{x\sim p_2}[f(x)]. \tag{3.10}$$

sup here stands for the supremum or least upper bound. It is taken over all 1-Lipschitz functions f.

3.4.3 Optimization

EM distance versus JS divergence

Let $G(\theta, \mathbf{z})$ be a generator in the GAN setting implemented as an NN with parameters θ that gets as input the uniformly sampled noise variable $\mathbf{z}$. It is shown in [3] that since $G(\theta, \mathbf{z})$ is locally Lipschitz, $EM(p_{\text{data}}, p_G)$ is continuous everywhere and differentiable almost everywhere. This lets the EM distance provide useful gradients during learning. It is also shown in [3] that this does not hold for the JS divergence by providing a counterexample for a simple learning problem. In a GAN setting where the generator is implemented as an NN, it may thus be favorable to use the EM distance instead of the JS divergence.

Formal definition of the original WGAN

Since p_G is defined as the distribution of $G(\mathbf{z})$ with $z \sim p_{\mathbf{z}}$, $\mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[D(G(\mathbf{z}))] = \mathbb{E}_{\mathbf{x}\sim p_{\mathbf{G}}}[D(x)]$. Thus, by equation 3.10, the EM distance between p_{data} and p_G is:

$$EM(p_{\text{data}}, p_G) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[f(\mathbf{x})] - \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[f(G(\mathbf{z}))]. \tag{3.11}$$

In [3] the WGAN version of the original GAN min-max game (equation 3.2) is defined as:

$$\min_G \max_{D\in\mathcal{D}} \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[D(\mathbf{x})] - \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[D(G(\mathbf{z}))]. \tag{3.12}$$

Here, D is referred to as the critic instead of the discriminator since it is not trained to classify [3, 19]. $\mathcal{D}$ is the set of all functions that are 1-Lipschitz. Similar to how the original min-max game minimizes the JS divergence when the discriminator is optimal (see section 3.1.2), for an optimal critic, the WGAN min-max game minimizes the EM distance [19]. Note that this becomes apparent by regarding equation 3.12. It shows that for an optimal critic, the generator minimizes equation 3.11.

In order to minimize the EM distance, in the WGAN min-max game, the critic thus has to be constrained to be 1-Lipschitz. In the original WGAN paper [3] the critic was implemented as an NN and constrained to be K-Lipschitz by clipping its weights to the interval $[-c, c]$, where the value of K depends on c. They note that constraining D to be K-Lipschitz instead of 1-Lipschitz remains a valid way of optimizing the EM distance, since this only changes the EM distance by a factor K.

Gradient penalty

Clipping the critic weights has multiple downsides [3, 19]. It would therefore be better to restrict the critic to be Lipschitz continuous without clipping its weights. WGAN with gradient penalty [19] (WGAN-GP) achieves this by adding a gradient penalty to the value function of the critic. The WGAN critic value function is:

$$V_W = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[D(\mathbf{x})] - \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[D(G(\mathbf{z}))] - \lambda \mathbb{E}_{\hat{\mathbf{x}}\sim p_{\hat{\mathbf{x}}}}\left[(\|\nabla_{\hat{\mathbf{x}}} D(\hat{\mathbf{x}})\|_2 - 1)^2\right]. \tag{3.13}$$

The generator is trained to maximize $\mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[D(G(\mathbf{z}))]$. Here, the coefficient of the gradient penalty λ has a standard value of 10. $p_{\hat{\mathbf{x}}}$ contains samples taken uniformly from lines between points from p_{data} and points from p_G, which is the distribution of fake data generated by the generator. The critic is thus not enforced to be Lipschitz continuous everywhere, but only for points from $p_{\hat{\mathbf{x}}}$. However, the gradient of an optimal critic has a gradient with norm 1 on all straight lines between coupled points from p_{data} and p_G [19]. This motivates only enforcing the constraint on samples from $p_{\hat{\mathbf{x}}}$ [19]. The WGAN-GP training algorithm is presented in algorithm 2. Unlike the weight clipping constraint in the original WGAN, the gradient penalty is only a soft constraint, since the critic is not fully restricted to be Lipschitz continuous anywhere. Also, for a function to be 1-Lipschitz, it is sufficient for the norm of its gradient to never be larger than 1. However, the gradient penalty also penalizes critics with gradient norms smaller than 1. Still, it was found that

Algorithm 2 The WGAN-GP training algorithm [19]. For this algorithm, the default hyperparameters for Adam (see section 2.1.3) are the learning rate $\eta = 0.0001$ and first- and second-order moment estimates $\beta_1 = 0$ and $\beta_2 = 0.9$. Like in the original GAN algorithm (algorithm 1), k is the number of times the critic is updated for each generator update, m is the mini-batch size, and w and θ are the parameters of the critic D and generator G respectively. For the WGAN-GP training algorithm, k has a default value of 5.

```
procedure TrainWGAN-GP
    for number of training iterations do
        initialize $w$ and $\theta$
        for $k$ steps do
            for $i = 1, ..., m$ do
                Sample a noise vector $\mathbf{z} \sim p_{\mathbf{z}}$
                Sample real data point $\mathbf{x} \sim p_{data}$
                Sample a random number $\epsilon \sim U[0, 1]$
                $\tilde{\mathbf{x}} \leftarrow G(\mathbf{z})$
                $\hat{\mathbf{x}} \leftarrow \epsilon x + (1 - \epsilon)\tilde{\mathbf{x}}$
                $L_D^{(i)} \leftarrow -\left(D(\mathbf{x}) - D(\tilde{\mathbf{x}}) - \lambda(\|\nabla_{\hat{\mathbf{x}}} D(\hat{\mathbf{x}})\|_2 - 1)^2\right)$
            $w \leftarrow \text{Adam}\left(\nabla_w \frac{1}{m} \sum_{i=1}^{m} L_D^{(i)}\right)$
        Sample a mini-batch of $m$ samples $\{\mathbf{z}^{(1)}, ..., \mathbf{z}^{(m)}\} \sim p_{\mathbf{z}}$
        $\theta \leftarrow \text{Adam}\left(\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} -D\left(G\left(\mathbf{z}^{(i)}\right)\right)\right)$
```

WGAN-GP achieves a better performance than the regular WGAN for different generator and critic architectures [19].

Since the WGAN-GP algorithm is used to train all WGANs in this work, from here on, the abbreviation 'WGAN' is used instead of 'WGAN-GP' for brevity.

3.5 Auxiliary classifier generative adversarial nets

With Auxillary Classifier GANs (AC-GANs) [48], in addition to the noise vector $\mathbf{z}$, the generator receives a class label $\mathbf{c}$ as input. The discriminator does not receive $\mathbf{c}$ as input. Instead, an auxiliary classifier is implemented in the discriminator, tasking the discriminator with predicting this $\mathbf{c}$, in addition to its task of predicting whether its input comes from the real data set. Figure 3.2 shows a graphical representation of an AC-GAN.

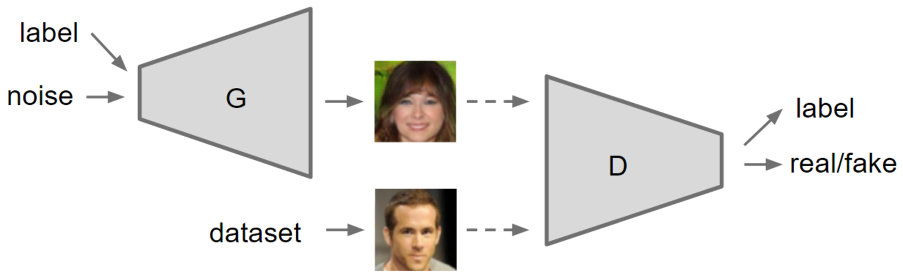


Figure 3.2: Graphical representation of an AC-GAN [48].

The discriminator thus has two outputs. The first one is $P(S = \text{real} \mid \mathbf{x})$, which is the probability that the source, S, from which the discriminator input, $\mathbf{x}$, is sampled is the real data set and not the distribution generated by the generator. Note that:

$$P(S = \text{fake} \mid \mathbf{x}) = 1 - P(S = \text{real} \mid \mathbf{x}).$$

The second output of the discriminator contains $P(C \mid \mathbf{x})$ for all classes C, which is a vector that contains the probability distribution over the class labels.

With this notation, the log-likelihood for predicting S, described in equation 3.1, becomes:

$$V_S = \mathbb{E}_{\mathbf{x},\mathbf{c}\sim p_{\text{data}}}[\log P(S = \text{real} \mid \mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}},\mathbf{c}\sim p_{\mathbf{c}}}[\log P(S = \text{fake} \mid G(\mathbf{z}, \mathbf{c}))]. \tag{3.14}$$

The log-likelihood for predicting the correct class is formulated as:

$$V_C = \mathbb{E}_{\mathbf{x},\mathbf{c}\sim p_{\text{data}}}[\log P(C = \mathbf{c} \mid \mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}},\mathbf{c}\sim p_{\mathbf{c}}}, [\log P(C = \mathbf{c} \mid G(\mathbf{z}, \mathbf{c}))]. \tag{3.15}$$

Both the generator and the discriminator are trained to maximize V_C. AC-GANs are thus trained by letting the discriminator maximize $V_C + V_S$ and letting the generator maximize $V_C - V_S$. AC-GANs are easily extended to work with multiple class labels by adding additional V_C terms to these value functions.

3.6 Coupled generative adversarial nets

Coupled GANs (CoGANs) [37] are able to learn to sample from a joint distribution, while being trained only using multiple marginal distributions, where each marginal distribution describes a different domain.

CoGANs are implemented as multiple GANs, each of which learns to generate samples from a different marginal distribution. The GANs are interdependent by sharing the weights in the first layer(s) of their generators and in the last layer(s) of their discriminators. When CoGANs are implemented as deep feed-forward NNs, high level image semantics are expected to be encoded in these shared weights and low level details are expected to be encoded in the weights that are not shared. This is the case because of the hierarchical way features are represented in the layers of this type of model. When a single noise vector is presented to all the GANs that make up a CoGAN, the high level semantics of the generated images will be the same, while the low level details will be different. Since each of these GANs produces images from a different domain, the tuple of images generated by presenting a single noise vector to CoGANs is the approximation of a sample of a joint distribution of these different domains.

Consider N marginal distributions $p_{\text{data}i}$ where $i \in [1..N]$. A CoGAN consisting of N GANs, GAN$_i$, is trained to generate samples from the joint distribution that contains N-tuples of samples of $p_{\text{data}i}$. For each $i \in [1..N]$, GAN$_i$, with discriminator D_i and generator G_i, is trained to produce samples from $p_{\text{data}i}$. The value function for CoGANs is:

$$V_{Co} = \sum_{i=1}^{N} V_{Coi}, \tag{3.16}$$

where:

$$V_{Coi} = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}i}}[\log D_i(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[\log(1 - D_i(G_i(\mathbf{z})))].$$

CoGANs can be trained with the WGAN-GP objective by replacing V_{Coi} in equation 3.16 with:

$$V_{WCoi} = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}i}}[D_i(\mathbf{x})] - \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}}[D_i(G_i(\mathbf{z}))] - \lambda \mathbb{E}_{\hat{\mathbf{x}}\sim p_{\hat{\mathbf{x}}i}}\left[(\|\nabla_{\hat{\mathbf{x}}} D_i(\hat{\mathbf{x}})\|_2 - 1)^2\right].$$

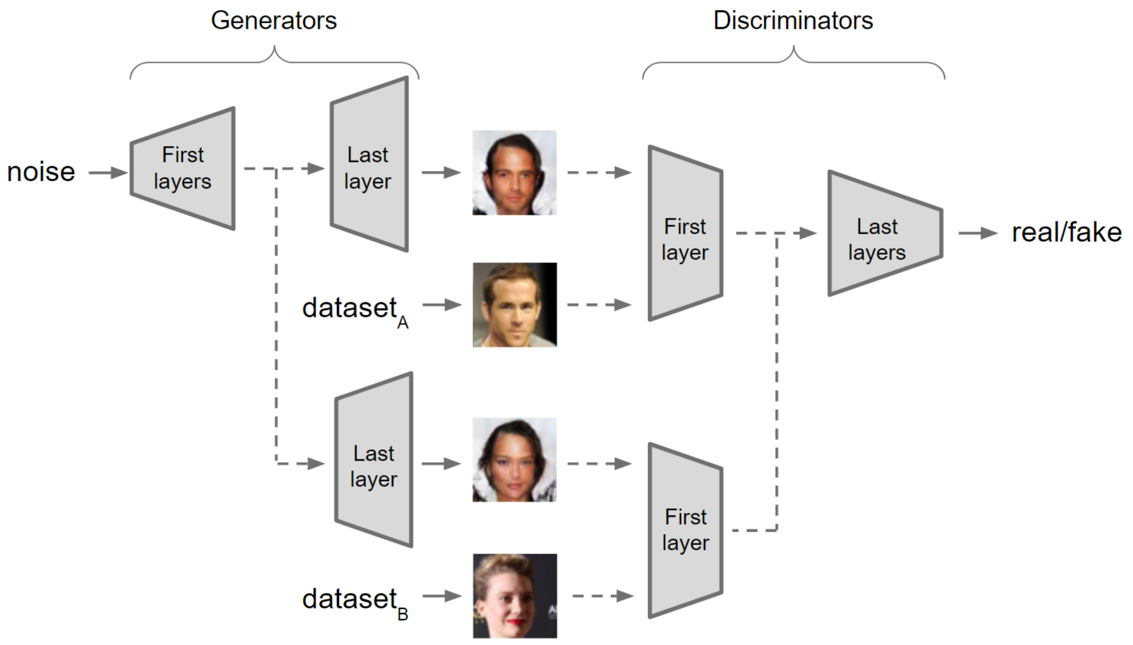


Figure 3.3: Graphical representation of a CoGAN [37] consisting of two GANs.

3.7 Performance measures

It is difficult to assess the quality of data generated by GANs [55, 15, 4, 25]. This also causes it to be challenging to accurately compare the quality of data produced by different GAN architectures, algorithms and hyper-parameter settings.

One way to measure the performance of generative models is an evaluation by humans. However, next to being time-consuming and expensive, this method also varies under evaluation conditions [55]. Specifically, the evaluation setup and motivation of the annotators affect the scoring. Furthermore, when annotators are given feedback, they learn from their mistakes and make fewer errors.

The (part of) the output of the discriminator that indicates whether the generated data is regarded as real could be used to monitor the convergence of GANs. However, for any specific discriminator, this output heavily depends on the generator that it is trained with. Therefore, the discriminator output cannot be used in a trivial way to quantitatively evaluate the quality of generated data.

A way to evaluate generated images based on the output of an NN has been proposed in the form of the Inception score [55]. This method uses a fixed NN (the Inception-v3 model [58], which was trained on ImageNet [9]) to obtain a conditional label distribution $p(y|\mathbf{x})$. This is the probability distribution of the labels y that the Inception model outputs when it is presented with the generated data $\mathbf{x} \sim p_G$. The Inception score is defined as

$$\text{Score}(G) = \exp\left(\mathbb{E}_{\mathbf{x}\sim p_G} KL(p(y|\mathbf{x})\|p(y)\right).$$

Intuitively, this method thus measures the quality of generated data by evaluating the similarity between $p(y|\mathbf{x})$ and $p(y)$, which is the actual distribution of these labels. The authors reported that the metric correlates very well with human judgment.

Originally, the Inception score was used to compare models trained on CIFAR-10 [30], which has a label distribution similar to that of ImageNet. Since then, the Inception score has been used to evaluate models trained on other data sets than Imagenet and CIFAR-10 as well [4], including CelebA [66]. However, argued from the theory underlying the score, it has been recommended to not use the Inception score to evaluate models trained to generate images from different data sets than ImageNet [4]. For any other data set a different Inception-v3 model, trained on that data set, should be used to obtain the Inception score. Furthermore, it was noted that the score overlooks overfitting and is not robust to different initial weights of the Inception-v3 model, unlike its classification accuracy [4].

The Fréchet Inception Distance [22] (FID) is proposed as an improvement upon the Inception score. To compute the FID between a real and fake data distribution, first the feature embeddings at the last pooling layer of the Inception-v3 model are computed for all images in these distributions. By assuming that the resulting distributions of embeddings each follow a multidimensional Gaussian, one can compute the distance between them by computing the Fréchet distance between these Gaussians. The Fréchet distance is computed as:

$$\mathrm{FID}(p_{\mathrm{data}}, p_G) = \|\mathbf{m}_{\mathrm{data}} - \mathbf{m}_G\|_2^2 + \mathrm{Tr}(C_{\mathrm{data}} - C_G - 2(C_{\mathrm{data}} C_G)^{1/2}),$$

where Tr denotes the trace of a matrix and $(\mathbf{m}_{\mathrm{data}}, C_{\mathrm{data}})$ and $(\mathbf{m}_G, C_G)$ denote the means and covariances of the embeddings of p_{data} and p_G. It was found that the FID improves upon the Inception score by capturing the similarity between image data distributions for other data sets than ImageNet and CIFAR-10.

The Geometry score (GS) [25] has been proposed as another metric to evaluate generative models. The method compares topological properties of the manifolds underlying the real and generated data. A downside of the GS is that, because it only regards topological properties, it is likely not suitable to be used by itself to assess the quality of generated image data. Future work will have to point out whether the GS correlates well with human judgment of image quality.

It was also shown that the discriminator value function without gradient penalty (see equation 3.13), which is an approximation of the EM distance, correlates well to sample quality [3, 19]. It can therefore be used to monitor the learning process.

Chapter 4

Extra Domain Data Generation

So far, GAN training algorithms and architectures have been covered that were designed to generate new samples from one or more domains that are defined by real data distributions. Some of these architectures are able to exploit additional information in the form of class labels. This section is devoted to tackling the problem of Extra Domain Data Generation (EDDG) as defined in section 1.2 using generative models.

4.1 Unsuitability of CoGAN

As has already been explained in section 3.6, the weights of the first layers of CoGAN generators are shared and weights of higher layers are not. These first layers encode the high level features of the generated data. When certain classes occur in both domains, the representation of these classes are expected to be encoded in the earlier layers.

Using the definitions of domain $\mathcal{A}$, $\mathcal{B}$ and class c as defined in section 1.2, consider some CoGAN that is trained to generate data from a joint distribution that consists of the domains $\mathcal{A}$ and $\mathcal{B}$. We denote its GAN that is trained to generate data from $\mathcal{A}$ as GAN$_{\mathcal{A}}$ and its other GAN as GAN$_{\mathcal{B}}$. It would be interesting to make it possible for this CoGAN to generate samples from $\mathcal{A}^c$.

One could test the ability of a CoGAN that has been trained in the regular fashion, as explained in section 3.6, for its ability to perform this task. This approach has the following downsides:

1. As described in section 3.6, discriminators in a CoGAN are trained to only classify samples from their input distribution of real data as real and to classify all input produced by the generator as fake. Furthermore, the discriminator of GAN$_{\mathcal{A}}$ never gets a real sample from $\mathcal{A}^c$ as input during training. Therefore, for the discriminator of GAN$_{\mathcal{A}}$, fake samples generated by the generator that are similar to samples from $\mathcal{A}^c$ should be easier to reject than fake samples generated by the generator that are similar to other samples from $\mathcal{A}$. This makes it harder for the generator to fool the discriminator by producing samples from $\mathcal{A}^c$, which in turn makes it more likely for the generator to find some optimum where it is unable to generate samples from $\mathcal{A}^c$.

2. When training is done, even if GAN$_{\mathcal{A}}$ is able to generate samples from $\mathcal{A}^c$, there is no simple way to obtain them. In order to generate the desired samples from $\mathcal{A}^c$ one has to use this generator to generate samples randomly, until a sample from $\mathcal{A}^c$ is found.

4.2 Suitability of AC-GAN

Instead of using different generators for different domains, one could use an AC-GAN and present a label that specifies from which domain to sample as input to its single generator. In this case,

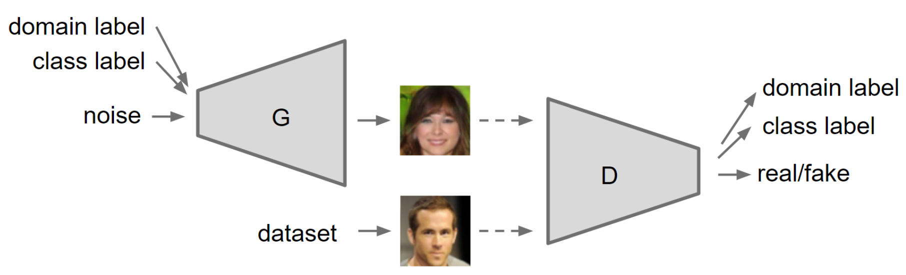


Figure 4.1: Graphical representation of an AC-GAN trained with both a domain label and a class label.

its discriminator would learn to recognize whether data it receives as input is real or fake and from which domain this input data is sampled. However, using this approach would not relieve either of the downsides stated in section 4.1; The generator would still learn that producing data from $\mathcal{A}^c$ is unfruitful and sampling from $\mathcal{A}^c$ would remain tedious.

However, both issues can be eliminated when both a class label and a domain label are presented as conditional input to the generator and used as variables that the discriminator has to predict (see figure 4.1). Evidently, this resolves the second issue, since after training, the generator can be primed to generate data from a specific class in a specific domain. This makes sampling from $\mathcal{A}^c$ more straightforward.

When training such an AC-GAN, the first issue is also taken care of when during training, the generator is presented with class labels from C_A as input when it is primed to generate data from $\mathcal{A}$ and with labels from C_B when generating data from $\mathcal{B}$. This way, the generator is never tasked to generate samples from $\mathcal{A}^c$ during training. It thus never gets to see that fooling the discriminator with data from $\mathcal{A}^c$ is difficult.

Although in this way the AC-GAN is never trained to specifically generate data from $\mathcal{A}^c$, it is trained to generate domain specific data depending on its input domain label and class specific data depending on its input class label. As long as the complexity of the function that it learns to do so is sufficiently restricted, there should be a large overlap in the features that the AC-GAN uses to generate data from different classes from the same domain. Similarly, there should be a large overlap in the features that the AC-GAN uses to generate data from the same class, but from different domains. Therefore, even though the generator never generates samples from $\mathcal{A}^c$ during training, it would still likely have the ability to generate samples of $\mathcal{A}^c$ after training.

One can condition a generator of such an AC-GAN twice with the same noise and class label, but with two different domain labels. As long as the complexity of the learned function is again sufficiently restricted, it can be expected that the generated data would be a sample from a joint distribution of the two different domains.

4.3 Coupled auxiliary classifier generative adversarial nets

Another GAN architecture that might be able to generate samples from $\mathcal{A}^c$ is the combination of the CoGAN and AC-GAN architectures. This combination (CoAC-GAN) can be used for the generation of labeled instances that look similar, but lie in different domains. Figure 4.2 shows its architecture. CoAC-GANs consist of a CoGAN where the generators are conditioned with a class label and the discriminators predict this class label. This introduces some new weights in the generator that connect the new input to its first hidden layer and some new weights in the discriminator that connect its last layer to its new output. The GANs that make up the CoGAN share these newly introduced weights. This is consistent with the regular CoGAN architecture, where the weights of the first layers in the generators and in the latter layers of the discriminators are shared.

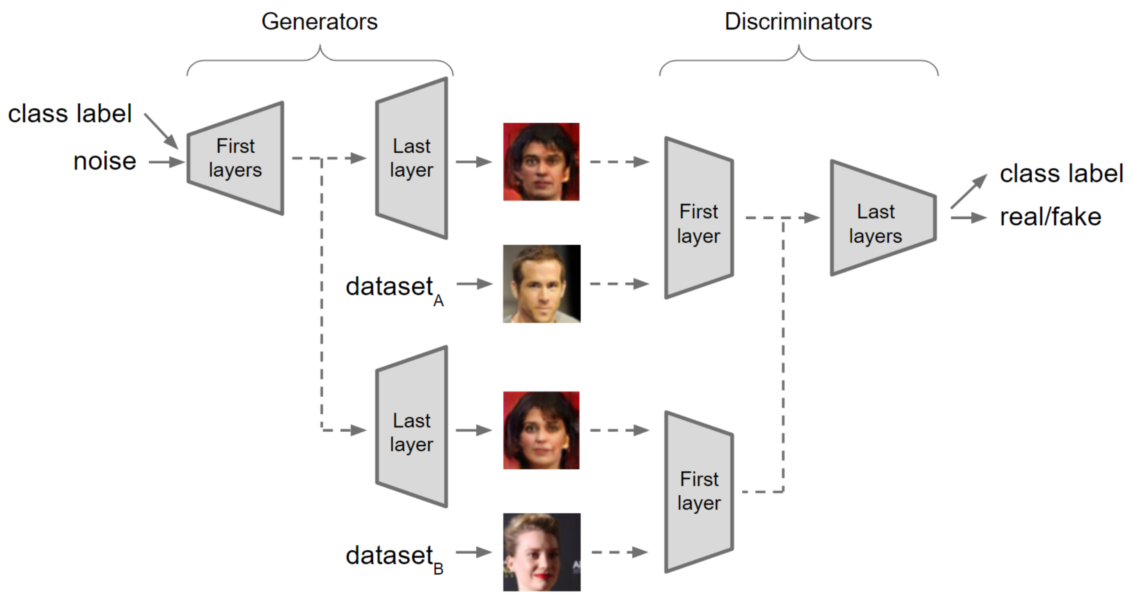


Figure 4.2: The CoACGAN architecture. Dashed connections indicate that data flows through only one of the arrows. For solid connections, data flows through all arrows.

As with a regular AC-GAN, because the generators of a CoAC-GAN have a conditional input variable that determines the class of the generated samples, it is possible to constrain the generator from generating samples from $\mathcal{A}^c$ during training in the same way as described in section 4.2, which avoids the first downside stated in section 4.1. Similarly, the second downside is also avoided, because when training is done, the class variable can be used to specify that a sample from $\mathcal{A}^c$ has to be generated using the generator of GAN$_{\mathcal{A}}$.

4.3.1 Formal definition

By combining the value functions of CoGANs and AC-GANs, we propose Coupled Auxiliary Classifier GANs (CoAC-GANs). For the i^{th} GAN in a CoAC-GAN with a discriminator that outputs the tuple $D_i(\mathbf{x})$, $P_i(C \mid \mathbf{x})$ the log-likelihoods from (3.14) and (3.15) respectively become:

$$V_{Si} = \mathbb{E}_{\mathbf{x},\mathbf{c}\sim p_{\text{data}i}}[\log D_i(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}},\mathbf{c}\sim p_{\mathbf{c}i}}[\log(1 - D_i(G_i(\mathbf{z},\mathbf{c})))]$$

and

$$V_{Ci} = \mathbb{E}_{\mathbf{x},\mathbf{c}\sim p_{\text{data}i}}[\log P_i(C = \mathbf{c} \mid \mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}},\mathbf{c}\sim p_{\mathbf{c}i}}[\log P_i(C = \mathbf{c} \mid G_i(\mathbf{z},\mathbf{c}))].$$

A CoAC-GAN consisting of N GANs is trained by letting the discriminators maximize:

$$\sum_{i=1}^{N} V_{Ci} + V_{Si} \tag{4.1}$$

and letting the generators maximize:

$$\sum_{i=1}^{N} V_{Ci} - V_{Si}. \tag{4.2}$$

CoAC-GANs can be trained with the WGAN-GP objective by replacing the V_{Si} terms in equations 4.1 and 4.2 with:

$$V_{WSi} = \mathbb{E}_{\mathbf{x},\mathbf{c}\sim p_{\text{data}i}}[D_i(\mathbf{x})] - \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}},\mathbf{c}\sim p_{\mathbf{c}i}}[D_i(G_i(\mathbf{z},\mathbf{c}))] - \lambda\mathbb{E}_{\hat{\mathbf{x}}\sim p_{\hat{\mathbf{x}}i}}\left[(\|\nabla_{\hat{\mathbf{x}}}D_i(\hat{\mathbf{x}})\|_2 - 1)^2\right].$$

4.3.2 Comparison with AC-GAN

CoAC-GANs and AC-GANs as described in section 4.2 have similar architectures. Their main difference lies in the way they handle domain specific information.

As with regular CoGANs, the generators of a single CoAC-GAN produce data from different domains and these generators only differ from each other in their last non-shared layer(s). When a CoAC-GAN is used to generate data from these domains, the domain specific information is thus forced to be encoded in the last generator layer(s). Because of the hierachical way features in feed-forward NNs are encoded, the last layer(s) of the generator are likely to encode low level details.

When the domains from which one wishes to generate data only differ in low level details, it might thus be beneficial to use CoAC-GANs. This is because in this case, there is a match between the parts of the generators where the domain specific information is forced to be encoded and the parts of the generators where domain specific information is likely to be encoded given the nature of feed-forward NNs. On the other hand, it is also possible that the domains from which one wishes to generate data differ not solely in their low level details. In this case, using the CoAC-GAN architecture might be harmful for the quality of the generated data.

On the contrary, in AC-GANs, the domain specific information is presented to the generator as a part of its input. This means that, differently from CoAC-GANs, the encoding of the domain specific information is likely to occur in the first layers and might occur in all layers of the AC-GAN. Since AC-GAN allows for the domain specific information to occur in any layer, its architecture should not hinder the generation of data from multiple domains that differ in high level details.

There is no difference between the way an AC-GAN processes domain labels and the way it processes class labels. Also, class labels are decoded by AC-GANs and CoAC-GANs in the same way. CoAC-GANs can thus encode class-specific information in the shared weights of their generators. It should be noted that the generators are not restricted to encode class label information in the non-shared weights. Especially when class labels describe low-level details, the decoding of these labels could be done in the last layer(s) of the generators in both AC-GANs and CoAC-GANs.

Chapter 5

Experiments

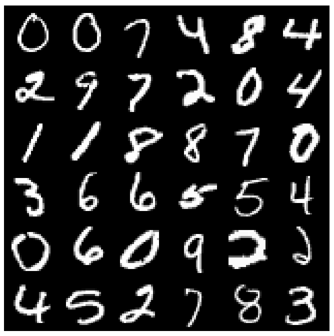
5.1 Data sets

This section describes the data sets used for the experiments performed for this work. All data sets are normalized so that their pixel values lay in the domain [-1,1].

5.1.1 Digits

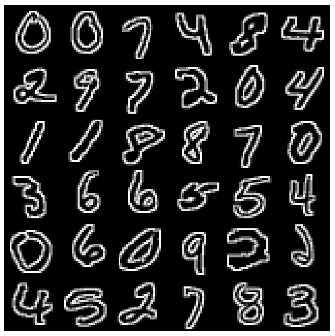
MNIST The first data set used for the experiments in this work is the MNIST data set [33]. This data set consists of black and white images of handwritten instances of the digits 0-9. The data set comprises of a training set of 60,000 images and a test set of 10,000 images. The digits in the training and test sets were written by disjoint sets of writers. The size of the MNIST images is 28×28 pixels. Figure 5.1a shows a sample of the MNIST data set.

Following [37], this work also uses a version of the MNIST data set where all images have been modified so that they depict the edges of the digits. For each image, the modification consists of first dilating the image and then substracting the original image from this dilated image. The resulting data set is denoted as MNIST-edge in this work. Figure 5.1b shows a sample of it.



(a) MNIST

,



(b) MNIST-edge

Figure 5.1: Corresponding images from the MNIST and MNIST-edge data sets.

Street View House Numbers The Street View House Numbers (SVHN) data set [45] consists of images of labeled house-number digits. Many of these images contain distracting digits to the sides of the digit of interest. The resolution of the images is 32×32 pixels. The data set contains 73,257 digits for training and 26,032 digits for testing. Figure 5.2 shows a sample of this data set.

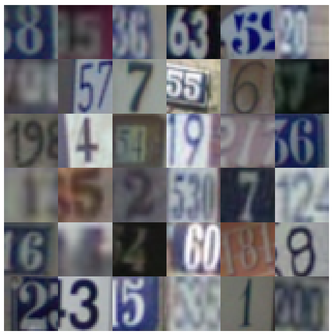


Figure 5.2: Images from the SVHN data set.

5.1.2 Faces

Another data set that is used to train the models in this work is CelebA [66]. Figure 5.3 shows a sample. CelebA contains more than 200,000 celebrity face images of more than 10,000 different identities. The images are annotated with 40 binary attributes such as gender and whether the depicted face is smiling. CelebA has been extensively used in research on GANs (e.g. [37, 6, 50, 26]).

In this work the aligned version of the data set is used. Furthermore, the images were cropped to a size of 160 × 160 pixels and consecutively re-sized to 64 × 64 pixels.

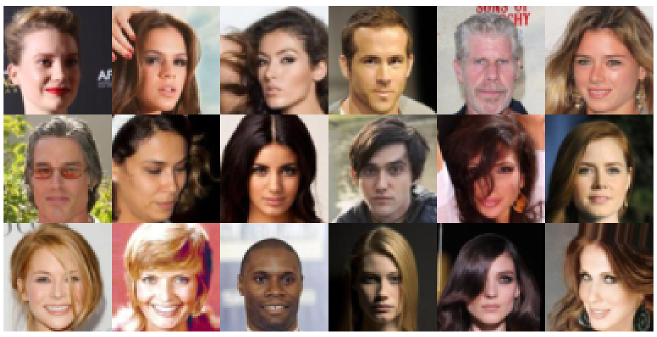


Figure 5.3: Sample of the CelebA data set

5.2 Extra-domain data generation

Here, the data generation experiments are described that are done to determine whether it is possible for a GAN to generate data from domain-class combinations that are not present in the data on which it is trained. The image generation is done with CoGANs, which do not use class labels and are therefore generate unlabeled data, and with CoAC-GANs and AC-GANs, which do use class labels. The GANs are trained on a combination of the MNIST and MNIST-edge data sets, combinations of the MNIST and SVHN data sets, and on the CelebA data set. For the digit data set, the MNIST, MNIST-edge and SVHN data sets each represent a different domain. For the CelebA data set, the image domain is determined by gender and the classes are 'smiling' and 'not smiling'.

Data sets with missing classes are emulated. For the face image generation task, no smiling males are presented to the discriminator during training. This data set will be denoted with Celeba^1.

Similarly, for the data generation task with the MNIST and MNIST-edge domains, some digits from the MNIST-edge are not presented during training. Experiments are performed where the digit '5' is missing and where the digits '5'-'9' are missing. These data sets are denoted MNIST^1 and MNIST^2 respectively.

For the data generation tasks with the MNIST and SVHN domain, the digits '0'-'4' are present in both the MNIST and SVHN domain. The only other data available during training is all data of one single digit from MNIST. The resulting data sets are denoted as SVHN^1_n where n is the digit that is only available in the MNIST domain. When these data sets are referred to in general without specifying which digit is missing from the SVHN domain, the data sets are denoted as SVHN^1.

GANs are trained with the data sets described above to either minimize the regular GAN objective described in section 3 or to minimize the EM distance with gradient penalty described in section 3.4, which will be referred to as the WGAN objective.

5.3 Extra-domain classification

The experiments described here are performed to learn whether data of missing classes of domain $\mathcal{A}$ can nevertheless be classified accurately when encountered during testing. In order to do so, firstly, CoAC-GANs and AC-GANs trained as described in section 5.2 are used to generate data from all possible domain-class combinations, including the ones missing from the training set. This fake data is used to construct two types of training sets. One consists only of fake data and one consists of both fake and real data. For both of these types of data sets, for both the GAN and WGAN objective, five CoAC-GANs and five AC-GANs are trained with and tested on their ability to generate the missing data from MNIST^1 and MNIST^2. This results in a total of $2 \times 2 \times 2 \times 2 \times 5 = 80$ generative models. For each of these generative models, an individual classifier is trained.

Class label distributions

The class label distribution of a data set consisting of only fake data used for classifier training is taken from the matching real data set. It is assumed that the frequency in which the missing domain-class instances occur is either known or can be estimated well. Missing domain-label frequencies thus follow the corresponding real data sets as well.

For a data set that contains both real and fake data the distribution of class labels for the real data is simply that of domain $\mathcal{A}$. The distribution of class labels for the fake data only differs from that of the data set containing only fake data in that the amount of data with labels that only occur in $\mathcal{B}$ is doubled. This is done to mimic the expected class label distribution during testing, assuming that the missing classes in domain $\mathcal{A}$ will occur proportionally to those of domain $\mathcal{B}$.

Baselines

The accuracy of naively trained classifiers on class-domain combinations that were not presented during training is also examined. These classifiers are trained on only data from domain $\mathcal{B}$ and on the junction of $\mathcal{A}$ and $\mathcal{B}$, where again labels of classes that are represented in both $\mathcal{A}$ and $\mathcal{B}$ are sampled with equal probability during training. Five classifiers were trained for each of these types of training sets for both MNIST^1 and MNIST^2. The performance of the CoAC-GANs and AC-GANs are compared with these baselines.

5.4 Model details

5.4.1 GANs

Architectures

The architectures of the GANs trained with CelebA images in this work follow the architectural constraints of DCGAN [51], which are repeated in section 3.3. Tables 5.1 and 5.2 show the

Table 5.1: Generator for GANs trained with CelebA images.

Layer	Preceding layer	Type	Output dimensions	Description
0	-	Input	$100 + d$	Concatenation of noise and class label
1	0	Fully connected	$512 \times 4 \times 4$	batchnorm, ReLU
2	1	Transposed convolution	$256 \times 8 \times 8$	kernel size 5, stride 2, pad 2, batchnorm, ReLU
3	2	Transposed convolution	$128 \times 16 \times 16$	kernel size 5, stride 2, pad 2, batchnorm, ReLU
4	3	Transposed convolution	$64 \times 32 \times 32$	kernel size 5, stride 2, pad 2, batchnorm, ReLU
5	4	Transposed convolution	$3 \times 64 \times 64$	kernel size 5, stride 2, pad 2, tanh

Table 5.2: Discriminator for GANs trained with CelebA images. For the LeakyReLUs, $\alpha = 5$.

Layer	Preceding layer	Type	Output dimensions	Description
0	-	Input	$3 \times 64 \times 64$	Real or fake data
1	0	Convolution	$64 \times 32 \times 32$	kernel size 5, stride 2, pad 2, LeakyReLU
2	1	Convolution	$128 \times 16 \times 16$	kernel size 5, stride 2, pad 2, batchnorm, LeakyReLU
3	2	Convolution	$256 \times 8 \times 8$	kernel size 5, stride 2, pad 2, batchnorm, LeakyReLU
4	3	Convolution	$512 \times 4 \times 4$	kernel size 5, stride 2, pad 2, batchnorm, LeakyReLU
5	4	Convolution	$1 \times 1 \times 1$	1 kernel of size 4, sigmoid
6	4	Convolution	$c \times 1 \times 1$	c kernels of size 4, softmax

architecture details. Unless otherwise specified, in all experiments, instance normalization [11] (section 2.4.2) replaces batch normalization in the WGAN discriminators. Tables 5.3 and 5.4 show the architectures of GANs trained with digit images. These architectures were also used in [37]. Although these architectures do not follow all DCGAN constraints, they have been shown to perform well in digit image generation tasks [37]. Following the original WGAN paper [3], the sigmoid activation in the last layer of all WGAN discriminators is omitted.

Class labels are represented as one-hot vectors. For Tables 5.2 and 5.4 layer 6 and 7 respectively is only added for AC-GANs or CoAC-GANs. When predicting multiple class labels, multiple instances of these layers are present in the architectures. For each of these labels, c denotes the length of the corresponding one-hot vector. d in Tables 5.1 and 5.3 denotes the length of the concatenation of all one-hot vectors that are presented to a generator. i denotes the number of image channels of the in- or output images. When training AC-GANs to generate images from the MNIST and SVHN domains, the gray-scale MNIST images are converted to color images by simply repeating the intensity value thrice for each pixel.

Training

The GANs are trained with the Adam optimization algorithm [28] with $\eta = 0.0002$, $\beta_1 = 0.5$, and $\beta_2 = 0.999$. The batch size is set to 64 samples. For a CoGAN or CoAC-GAN, a single update consists of a forward pass and backward pass through each individual GAN that the model is composed of, which results in an effective batch size of 128 samples for their shared layers. All AC-GANs are trained for 100 epochs. CoGANs and CoAC-GANs are trained for 125,000 batches

Table 5.3: Generator for GANs trained with digit images.

Layer	Preceding layer	Type	Output dimensions	Description
0	-	Input	$(100 + d) \times 1 \times 1$	Concatenation of noise and class label
1	0	Transposed convolution	$512 \times 4 \times 4$	kernel size 4, batchnorm, PReLU
2	1	Transposed convolution	$256 \times 7 \times 7$	kernel size 3, stride 2, pad 1, batchnorm, PReLU
3	2	Transposed convolution	$128 \times 13 \times 13$	kernel size 3, stride 2, pad 1, batchnorm, ReLU
4	3	Transposed convolution	$64 \times 25 \times 25$	kernel size 3, stride 2, pad 1, batchnorm, ReLU
5	4	Transposed convolution	$i \times 28 \times 28$	kernel size 6, pad 1, tanh

Table 5.4: Discriminator for GANs trained with digit images.

Layer	Preceding layer	Type	Output dimensions	Description
0	-	Input	$i \times 28 \times 28$	Real or fake data
1	0	Convolution	$20 \times 24 \times 24$	kernel size 5
2	1	Max pooling	$20 \times 12 \times 12$	kernel size 2
3	2	Convolution	$50 \times 8 \times 8$	kernel size 5
4	3	Max pooling	$50 \times 4 \times 4$	kernel size 2
5	4	Convolution	$100 \times 1 \times 1$	kernel size 4, PReLU
6	5	Fully connected	1	kernel size 1, sigmoid
7	5	Fully connected	c	kernel size 1, softmax

Table 5.5: Classifier trained with generated images to classify domain-class combinations that are missing during training.

Layer	Preceding layer	Type	Output dimensions	Description
0	-	Input	$i \times 28 \times 28$	Real or fake data
1	0	Convolution	$20 \times 24 \times 24$	kernel size 5
2	1	Max pooling	$20 \times 12 \times 12$	kernel size 2, ReLU
3	2	Convolution	$50 \times 8 \times 8$	kernel size 5
4	3	Max pooling	$50 \times 4 \times 4$	kernel size 2, ReLU
5	4	Fully connected	500	ReLU
6	5	Fully connected	c	softmax

on digit data sets and for 150,000 batches on CelebA. For each five batches with a discriminator update, one generator update is done.

5.4.2 Classifiers

The architecture of the models trained for extra-domain classification is a variant of the LeNet [35] architecture. It is shown in table 5.5. The classifiers were trained with SGD with a learning rate of 0.01 with 64 samples per batch. To keep the classifiers from overfitting, 10,000 samples are removed from the training set and used as a validation set. During training, the error on the validation set is monitored at each epoch. When the validation error is not improved upon for 10 epochs, training is terminated and the model at the epoch with the lowest validation error is used for testing.

5.5 Fréchet inception distance

For the MNIST^1, MNIST^2 and SVHN^1 data sets, experiments were performed to investigate the relation between the extra domain data generation and classification performance and the Fréchet Inception Distance (FID) between real and generated data. For each model trained on MNIST^1, MNIST^2 or SVHN^1 data sets, the FID was computed between a generated data set of 50,000 images of digits of missing domain-class combinations and all digits in the test set with corresponding class labels.

Chapter 6

Results

6.1 Extra-domain data generation

6.1.1 CoGAN

Digits

Figure 6.1 shows images produced by CoGANs and WCoGANs trained on MNIST^1. The digit ‘1’ is depicted in a disproportionally large share of the images shown in figure 6.1a. Training a CoGAN with the regular GAN objective can thus eventually result in a mode collapse. At the optimal point during the training of this model, the CoGAN produces images that cover a larger variety of digits. Interestingly, for some input noise vectors the CoGAN is also able to produce the digit ‘5’, even though this was missing in the training data of the MNIST-edge domain.

Training the WCoGAN did not result in a mode collapse. After training is completed, the model is able to produce the digit ‘5’ in the MNIST domain. However, the corresponding images in the MINST-edge domain do not resemble ‘5’s. Instead, the model produces loosely coupled images that resemble other digits.

Faces

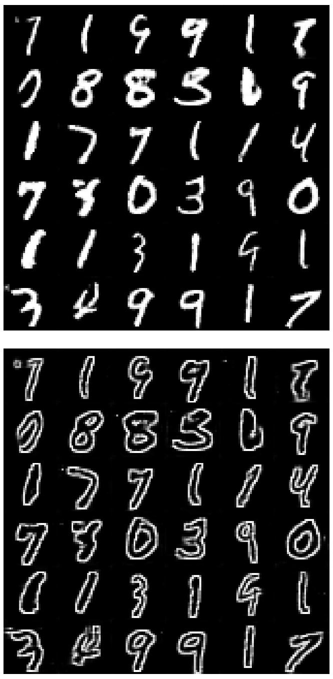
Figure 6.2 shows images produced by CoGANs and WCoGANs trained on CelebA^1. The GANs trained to produce females in these models are able to produce smiling and non-smiling faces. However, when the same noise vectors are presented to the corresponding GANs trained to generate male face images, none of the resulting images depicts a smiling face.

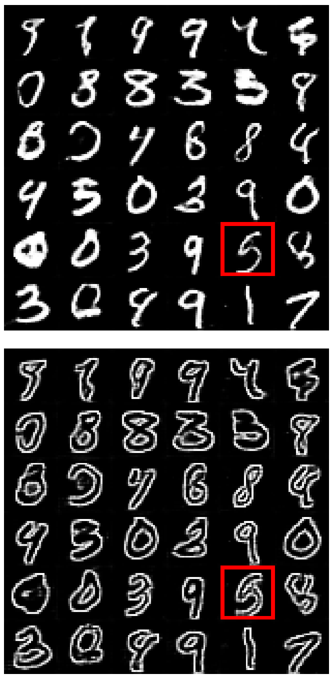
6.1.2 AC-GAN and CoAC-GAN

Digits

Figures 6.3 and 6.4 show the images produced by AC-GANs, WAC-GANs, CoAC-GANs and WCoAC-GANs trained on MNIST^1 and MNIST^2. All models are able to generate convincing samples of class-domain combinations that were present in the training data, although generally the output domains of models trained with the WGAN objective cover a larger variation of styles.

The models were also conditioned to generate images of the missing classes of the MNIST-edge domain. All models trained on MNIST^1 produce images of the missing class that are recognizable as the digit ‘5’. Of the models trained on MNIST^2, the images of missing classes produced by CoAC-GANs and WCoAC-GANs also clearly express their input class label. However, missing images produced by AC-GANs and WAC-GANs trained on MNIST^2 are often unrecognizable. For both MNIST^1 and MNIST^2, the missing class-domain samples produced by CoAC-GAN and WCoAC-GANs are of higher quality than those produced by AC-GANs and WAC-GANs.


(a) CoGAN results after 125,000 batches of training.


(b) CoGAN results after 25,000 batches of training.

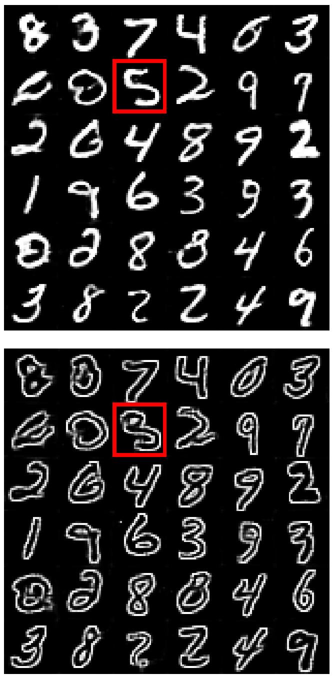

(c) WCoGAN results after 125,000 batches of training.

Figure 6.1: Images generated by CoGANs and WCoGANs trained on MNIST[1]. For each figure, the top images show digits produced by the GAN trained on MNIST data. The images below them show the corresponding digits produced by the GAN trained on MNIST-edge data. These digits were produced from the same input noise vectors. The noise vectors used to produce the digits in figures 6.1a and 6.1b are also identical.

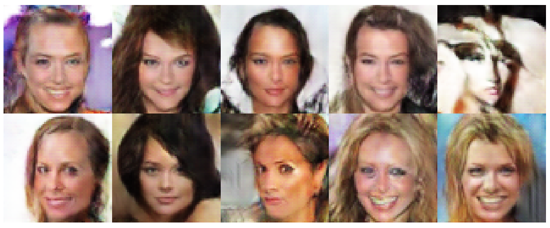
Figures 6.5 and 6.6 show images produced by models trained on the SVHN[1] data sets. The generative models are able to produce recognizable digits in the SVHN domain of classes that are also available in the MNIST domain. However, the models rarely produce recognizable digits from SVHN. In one case, an AC-GAN converged such that it produces digits from the MNIST domain regardless of the domain label it is primed with.

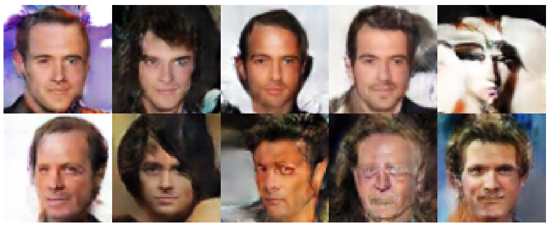
Faces

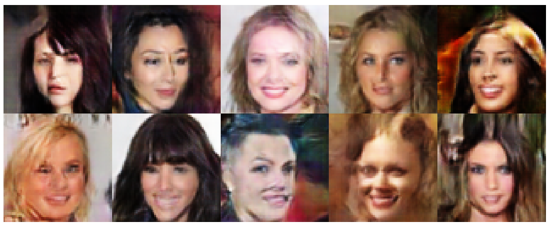
Figure 6.7 shows the images generated by GANss trained with CelebA[1]. CoAC-GANs trained with the regular GAN objective were unable to generate images of the missing domain-class combination, which is males that are smiling. The quality of the generated images is inferior to those generated by a CoGAN in the same setting (see figure 6.2). The CoAC-GAN is also prone to mode collapse, which clearly shows in all smiling mouths generated by CoAC-GANs. The images of smiling males generated early in the training process in figures 6.7a and 6.7b show color artifacts in the mouth area that are not represented in the training data. Furthermore, the difference between non-smiling males and females that is present in early stages of training has almost completely disappeared at the end of the training process.

WCoAC-GANs are able to produce images of smiling males, especially in the early stages of training, as can be seen in figure 6.7f. In figure 6.7h color artifacts can again be seen around the mouths of the images of smiling males produced at the end of training. As with CoAC-GANs trained with the regular GAN objective, these images also show more female characteristics than those obtained earlier in the training process.

These images produced by AC-GANs and WAC-GANs show none of the issues of CoAC-GANs


(a) Females, CoGAN


(b) Males, CoGAN


(c) Females, WCoGAN

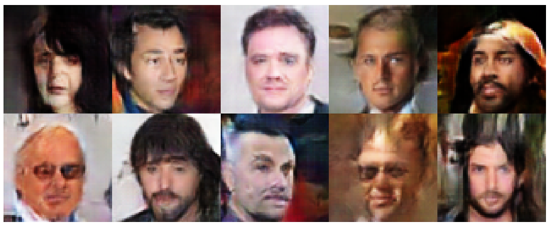
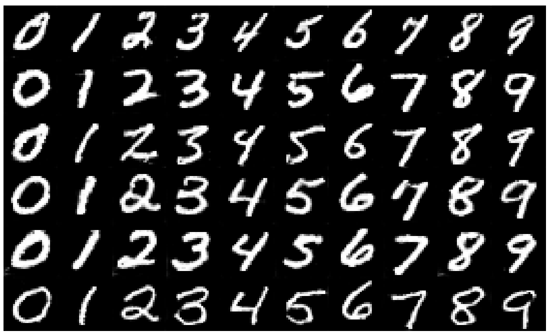
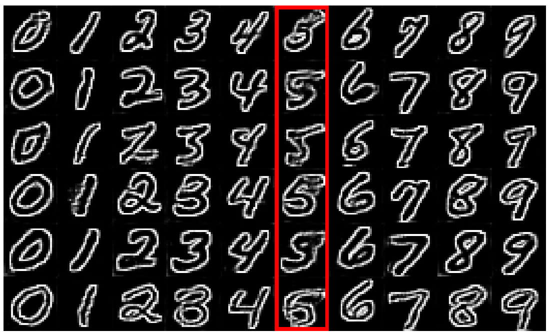
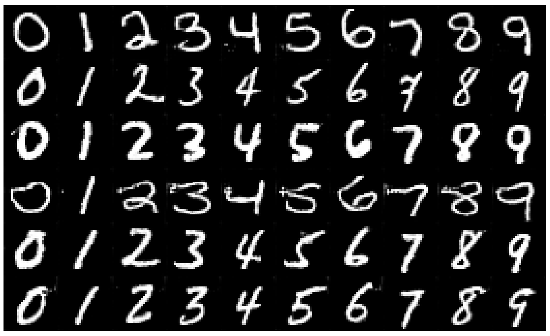

(d) Males, WCoGAN

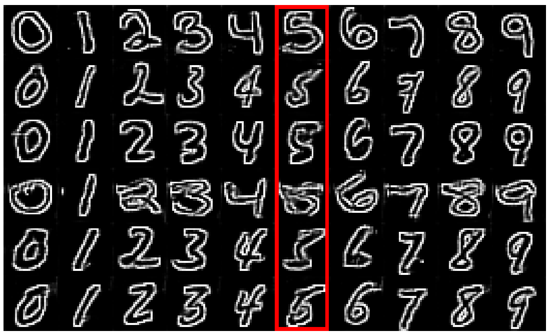
Figure 6.2: Images generated by CoGANs and WCoGANs trained on CelebA[1].

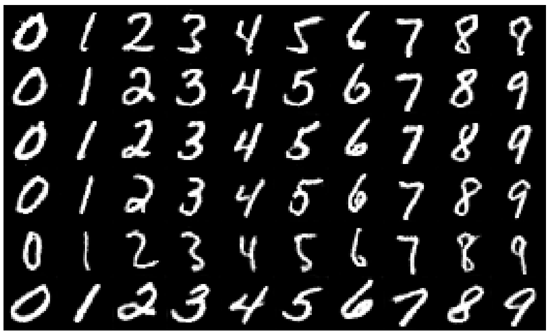
and WCoAC-GANs described above. The AC-GANs and WAC-GANs are able to produce images of smiling males even though they have not been presented with this domain-class combination during training.

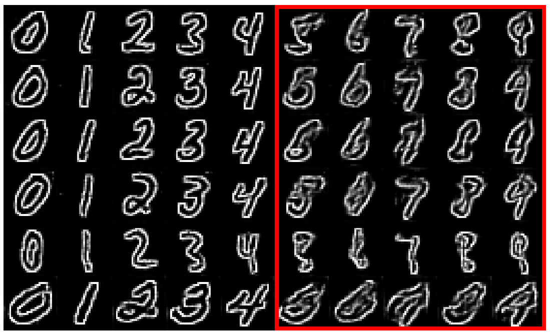

(a) MNIST^1, MNIST, AC-GAN

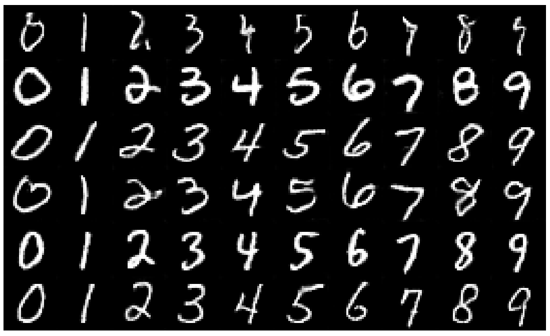

(b) MNIST^1, MNIST-edge, AC-GAN


(c) MNIST^1, MNIST, WAC-GAN


(d) MNIST^1, MNIST-edge, WAC-GAN


(e) MNIST^2, MNIST, AC-GAN


(f) MNIST^2, MNIST-edge, AC-GAN


(g) MNIST^2, MNIST, WAC-GAN

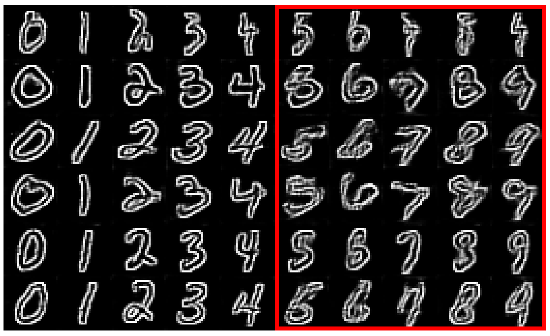
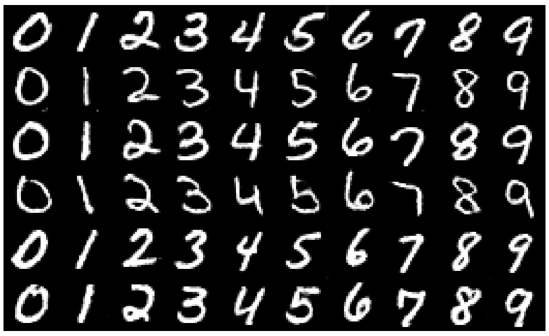
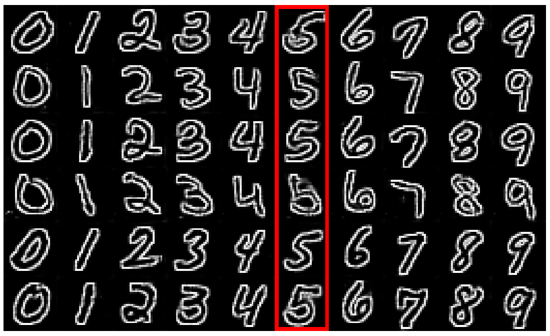
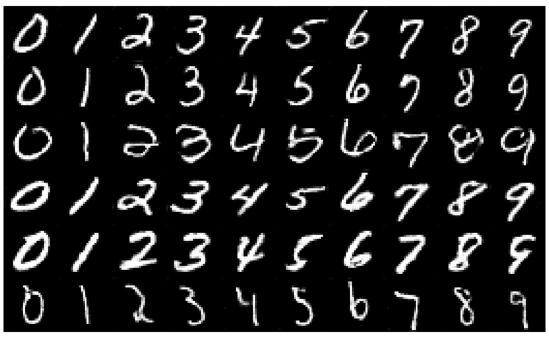
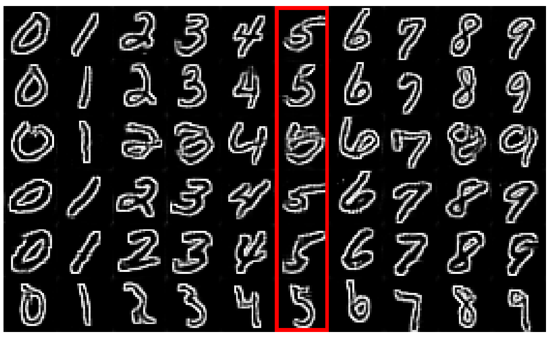

(h) MNIST^2, MNIST-edge, WAC-GAN

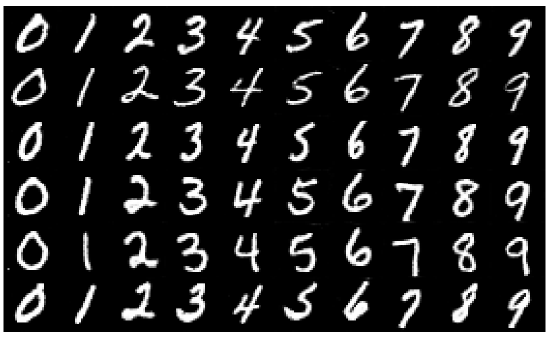
Figure 6.3: Images generated by AC-GANs and WAC-GANs trained on MNIST^1 and MNIST^2.

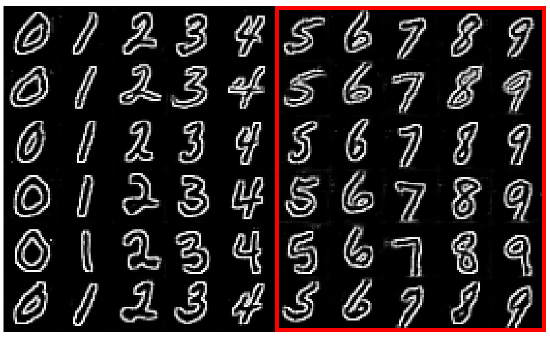

(a) MNIST^2, MNIST, CoAC-GAN

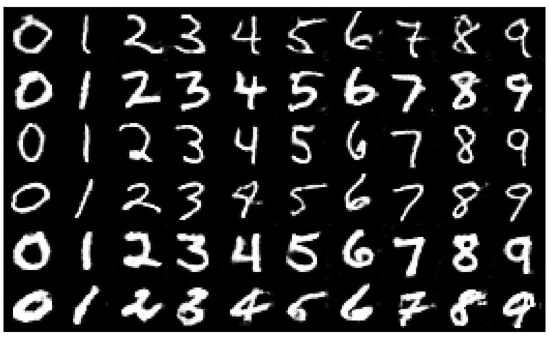

(b) MNIST^2, MNIST-edge, CoAC-GAN


(c) MNIST^2, MNIST, WCoAC-GAN


(d) MNIST^2, MNIST-edge, WCoAC-GAN


(e) MNIST^2, MNIST, CoAC-GAN


(f) MNIST^2, MNIST-edge, CoAC-GAN


(g) MNIST^2, MNIST, WCoAC-GAN

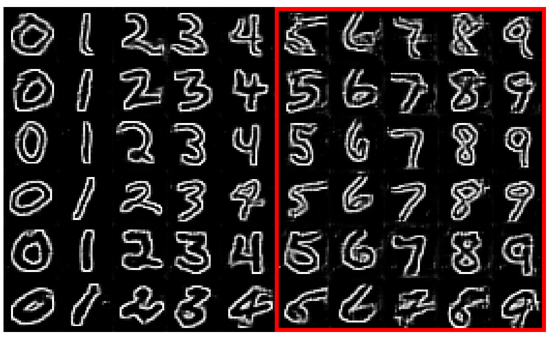
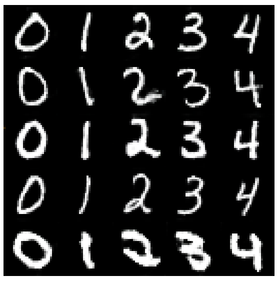
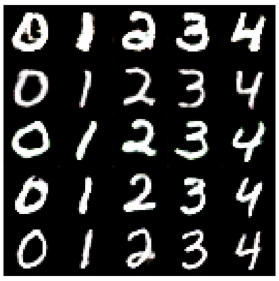
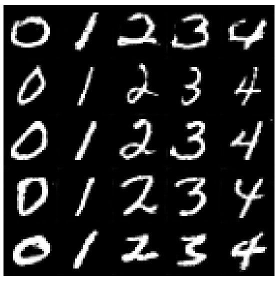
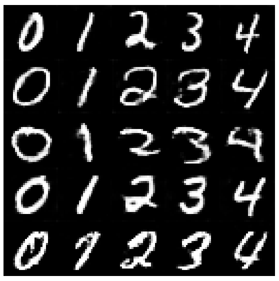

(h) MNIST^2, MNIST-edge, WCoAC-GAN

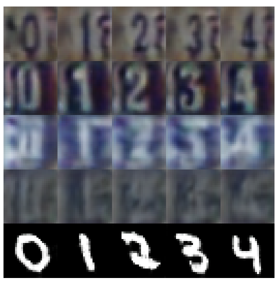
Figure 6.4: Images generated by CoAC-GANs and WCoAC-GANs trained on MNIST^1 and MNIST^2.

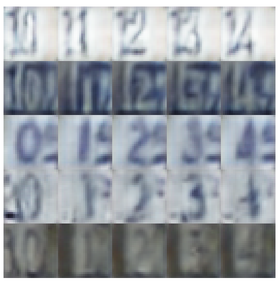

(a) MNIST, AC-GAN

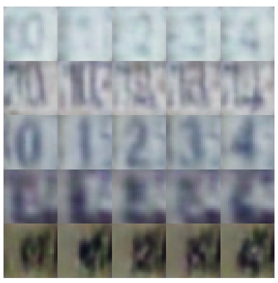

(b) MNIST, WAC-GAN


(c) MNIST, CoAC-GAN


(d) MNIST, WCoAC-GAN


(e) SVHN, AC-GAN


(f) SVHN, WAC-GAN


(g) SVHN, CoAC-GAN

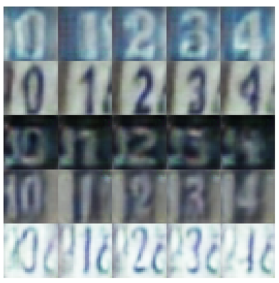
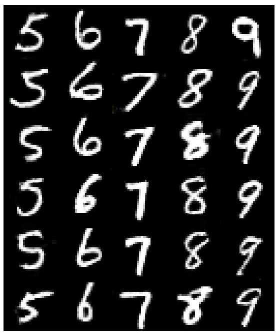
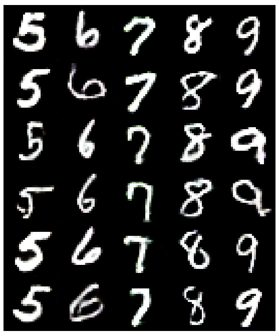
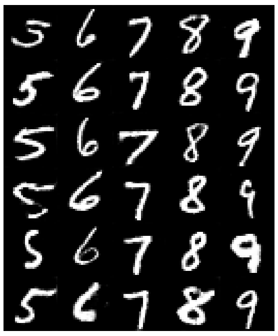
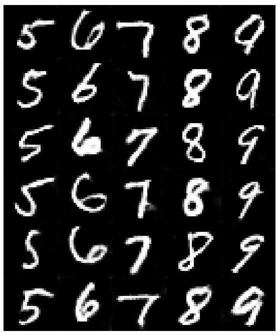

(h) SVHN, WCoAC-GAN

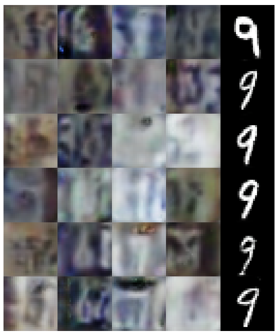
Figure 6.5: Images generated by AC-GANs, WAC-GANs CoAC-GANs and WCoAC-GANs trained on the SVHN[1] are shown here. In order to generate these results, at test time, the GANs were only primed with class labels that were represented in both the MNIST and SVHN domain. No images of missing domain-class combinations are shown here. From top to bottom, the images in the rows in each subfigure are produced by generators trained on data where the digit '5', '6', '7', '8', or '9' was missing. For the generation of images that share a row, the respective generator was primed with the same noise vector. The generation process of images in vertically adjacent subfigures differs only in the target domain. From top to bottom, the images in the rows of each subfigure show digits generated by the same generator as the top rows of the images in the subfigures of figure 6.6 from left to right. For the generation of these corresponding digits, the respective generator was primed with the same noise input.

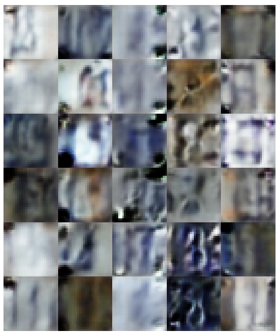

(a) MNIST, AC-GAN

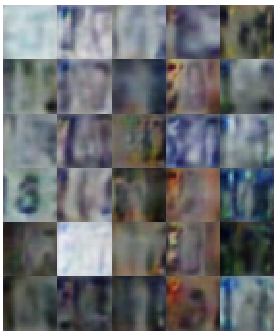

(b) MNIST, WAC-GAN


(c) MNIST, CoAC-GAN


(d) MNIST, WCoAC-GAN


(e) SVHN, AC-GAN


(f) SVHN, WAC-GAN


(g) SVHN, CoAC-GAN

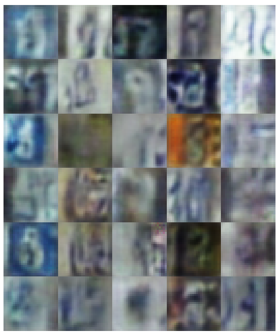
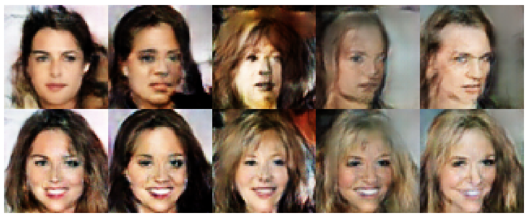
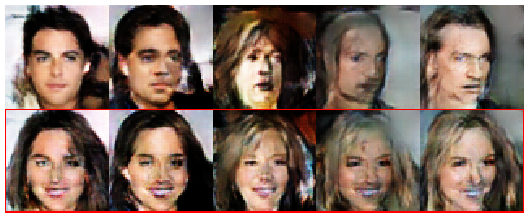
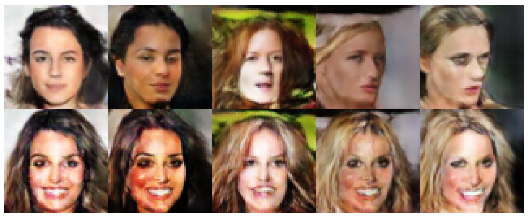
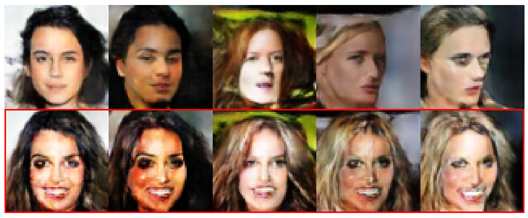

(h) SVHN, WCoAC-GAN

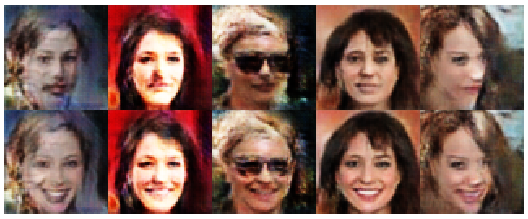
Figure 6.6: Images generated by AC-GANs, WAC-GANs CoAC-GANs and WCoAC-GANs trained on the SVHN[1] data sets are shown here. In order to generate them, the GANs were primed with the class labels that were not represented in the SVHN domain. Images in the same column of a subfigure were generated by the same generator. The generation process of images in vertically adjacent subfigures differs only in the target domain.

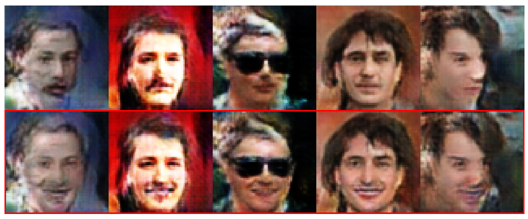

(a) Females, CoAC-GAN, 25,000 batches.

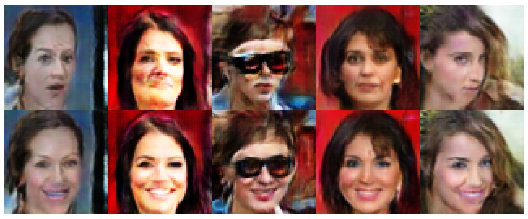

(b) Males, CoAC-GAN, 25,000 batches.

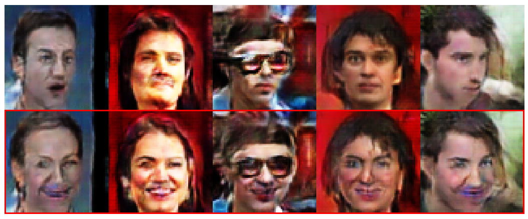

(c) Females, CoAC-GAN, 150,000 batches.

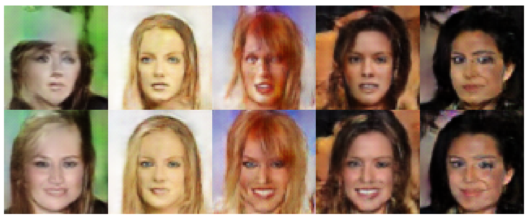

(d) Males, CoAC-GAN, 150,000 batches.

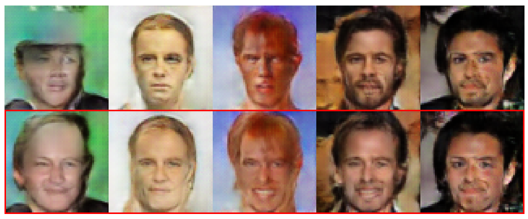

(e) Females, WCoAC-GAN, 25,000 batches.

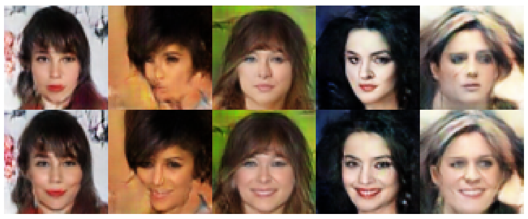

(f) Males, WCoAC-GAN, 25,000 batches.


(g) Females, WCoAC-GAN, 150,000 batches.


(h) Males, WCoAC-GAN, 150,000 batches.


(i) Females, AC-GAN, 150,000 batches


(j) Males, AC-GAN, 150,000 batches


(k) Females, WAC-GAN, 150,000 batches

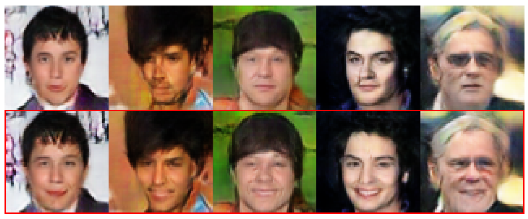

(l) Males, WAC-GAN, 150,000 batches

Figure 6.7: Images generated by CoAC-GANs, WCoAC-GANs, AC-GANs, and WAC-GANs trained on CelebA[1].

6.2 Extra-domain classification

Tables 6.1 and 6.2 show the accuracy of classifiers tested on real data of missing domain-class combinations. These classifiers were trained on only real data, on only fake data, and on hybrid data sets. In both tables, the baseline classifiers trained on only real data are clearly more accurate than randomly assigning labels based on prior class probabilities. This indicates that features these classifiers learned from the MNIST data set generalize to the MNIST-edge data set.

Training on data produced by CoAC-GANs rather than on data produced by AC-GANs results in a better classifier performance in all experiments. In general, training only on fake data also results in a better accuracy than training on hybrid data with both real and fake samples.

For the classifiers tasked to classify missing images from MNIST^1, the performance of the baseline is surpassed by all classifiers trained on fake or hybrid data sets. For classifiers tasked to classify missing digits from MNIST^2, this only holds when the fake training data originates from CoAC-GANs. In this case, classifiers trained on fake data generated by AC-GANs have poor accuracy.

The fact that a better accuracy is obtained by classifiers trained on data generated by CoAC-GAN with respect to those trained on data generated by AC-GANs coincides with the quality of the visual results shown in section 6.1.2.

Table 6.1: Classifier accuracy on the missing data from MNIST^1 ('5's from MNIST-edge). The real training data on which the classifiers are trained originates from MNIST^1. The fake training data is produced by GANs trained with MNIST^1. The best performing classifier is printed in bold.

Train data source	Training objective	Architecture	Accuracy (%): mean ± std
real MNIST	-	-	40.45 ± 7.42
real MNIST + real MNISTEDGE	-	-	53.34 ± 6.82
fake MNISTEDGE	WGAN	AC-GAN	94.69 ± 4.22
		COAC-GAN	97.00 ± 1.05
	GAN	AC-GAN	97.71 ± 0.40
		COAC-GAN	$\mathbf{98.03 \pm 0.70}$
real MNISTEDGE + fake MNISTEDGE	WGAN	AC-GAN	89.75 ± 6.44
		COAC-GAN	96.82 ± 0.82
	GAN	AC-GAN	94.13 ± 1.58
		COAC-GAN	97.13 ± 0.92

Table 6.2: Classifier accuracy on the missing data from MNIST^2 (digits 5-9 from MNIST-edge). The real training data on which the classifiers are trained originates from MNIST^2. The fake training data is produced by GANs trained with MNIST^2. The best performing classifier is printed in bold.

Train data source	Training objective	Architecture	Accuracy (%): mean ± std
real MNIST	-	-	67.74 ± 24.69
real MNIST + real MNISTEDGE	-	-	45.14 ± 14.11
fake MNISTEDGE	WGAN	AC-GAN	68.04 ± 14.50
		COAC-GAN	$\mathbf{95.82 \pm 1.60}$
	GAN	AC-GAN	56.26 ± 29.80
		COAC-GAN	93.27 ± 3.80
real MNISTEDGE + fake MNISTEDGE	WGAN	AC-GAN	57.14 ± 18.47
		COAC-GAN	94.33 ± 1.77
	GAN	AC-GAN	39.16 ± 27.16
		COAC-GAN	92.45 ± 4.12

6.3 Fréchet inception distance

Table 6.3 shows the Fréchet inception distances (FIDs) between real data of the missing domain-class combinations from emulated data sets and corresponding generated data. Table 6.4 shows FIDs between real data of the missing domain-class combinations from emulated data sets and the data of corresponding classes from the MNIST domain.

When comparing the results in the top two rows of tables 6.1 and 6.2 with the results in table 6.4 and the results in the rest of tables 6.1 and 6.2 with the results in table 6.3, a negative correlation can be observed between the FID and classifier accuracy for the MNIST^1 and MNIST^2 datasets. This relation can also be observed between the FID and the quality of data generated from missing domain-class combinations.

For the SVHN^1 data sets, this latter relation also holds, considering the black spots in images produced by WAC-GANs and the fact that the AC-GAN trained on SVHN$^1_9$ failed to generate images from the SVHN domain entirely (see figure 6.6).

Table 6.3: FIDs between generated data of missing domain-class combinations and real data with corresponding domain and class labels.

Data set	Training objective	Architecture	FID: mean ± std
MNIST^1	WGAN	AC-GAN	62.49 ± 18.51
		CoAC-GAN	12.24 ± 1.70
	GAN	AC-GAN	35.71 ± 4.77
		CoAC-GAN	10.35 ± 1.42
MNIST^2	WGAN	AC-GAN	78.64 ± 2.09
		CoAC-GAN	14.23 ± 1.42
	GAN	AC-GAN	65.77 ± 5.48
		CoAC-GAN	10.31 ± 1.55
SVHN^1	WGAN	AC-GAN	134.45 ± 8.24
		CoAC-GAN	57.16 ± 20.37
	GAN	AC-GAN	160.17 ± 28.16
		CoAC-GAN	67.24 ± 13.07

Table 6.4: FIDs between real data of missing domain-class combinations in the specified data sets and real MNIST data with corresponding class labels. The results shown here are analogous to those in the top two rows of tables 6.1 and 6.1. For the comparison between the missing data from MNIST^1 and the corresponding data from MNIST, only one FID can be computed. The same holds for MNIST^2. Therefore no standard deviation is given for the MNIST^1 and MNIST^2 datasets. A mean and standard deviation are given for the distances computed between MNIST and the variants of SVHN^1 in which different digits are missing.

Data set	FID
MNIST^1	108.57
MNIST^2	81.67
SVHN^1	214.65 ± 9.94

6.4 Discussion

6.4.1 Domain similarity

The models with the CoAC-GAN architecture and the AC-GAN architecture were both able to generate meaningful images of missing domain-class combinations when trained on MNIST^1 and MNIST^2. Only models with the AC-GAN architecture were able to generate meaningful images of the missing domain-class combination of the CelebA^1 data set. Furthermore, in images of the missing domain-class combination produced with models trained on the SVHN^1 data sets the class labels were rarely properly encoded. For both the CelebA^1 and the SVHN^1 data sets, some models failed to incorporate the correct domain characteristics into the generated images. These results suggest a positive correlation between the performance of the generative models on extra domain data generation and the resemblance between the domains on which they are trained.

6.4.2 Color artifacts

Images generated by CoAC-GANs of the domain-class combination that are missing in CelebA^1 show undesirable artifacts in the area that characterize the regarding class. This might be attributed to the CoGAN architecture. The domain specific information in the generators is forced to reside in the non-shared weights of the CoGAN, but this is not the case for class specific information. One explanation could therefore be that the decoding of information specific to the class label that is missing in domain $\mathcal{A}$ happens partly in the shared and partly in the non-shared generator weights. This is especially likely when the class label is not solely characterized by low level image details. However, the non-shared part of the generator that is tasked to generate images from domain $\mathcal{A}$ is never trained to further encode output of the shared layers when these are presented with the class label that is missing in domain $\mathcal{A}$. It would in this case be unable to transform this shared layer output into meaningful images.

Chapter 7

Conclusion

7.1 Extra domain data generation

In this thesis, it is researched whether it is possible to use Generative Adversarial Nets [17] (GANs) to generate new data that seems to originate from some domain, but characterizes classes of which no data is available for that domain. In order to do so, Coupled Auxiliary Classifier GANs (CoAC-GANs), a novel combination of Coupled GANs [37] (CoGANs) and Auxiliary Classifier GANs [48], are proposed as a method for data generation. The performance of CoGANs, AC-GANs and CoAC-GANs on EDDG is compared.

Results show that it is possible to generate labeled images of extra-domain classes using AC-GANs and CoAC-GANs trained with the regular GAN objective or with the Wasserstein GAN objective. In order to do so, data characterizing these classes must be available in some other similar domain.

Despite the theoretical downsides of using CoGANs for this task (see section 4.1), in some cases, CoGANs are able to generate data from the missing classes. However, since CoGANs cannot be presented with conditional class variables, they cannot generate labeled output.

The preference for AC-GANs or CoAC-GANs depends on the training data. CoAC-GANs seem to have a superior performance when the domains only differ in low level detail and for such domains seem to be able to cope better with multiple missing classes than AC-GANs. This is in line with theoretical expectations due to the difference in shared weights of the GANs that make up the architectures of CoAC-GANs or AC-GANs.

7.2 Qualitative evaluation

It has also been researched whether it is possible to use the generated data to train machine learning models which classify data from some domain where for some class(es) all data is missing in the training data.

The results show that data generated with AC-GANs and CoAC-GANs can be used to train classifiers that accurately classify the missing classes when they are encountered after training. The performance of these classifiers correlates with the quality of the generated images. The classification accuracy of such classifiers can thus be used as an indicator for qualitative evaluation of the EDDG performance of generative models. The results show that the Fréchet inception distance can be used as indicator for the EDDG performance of generative models as well.

7.3 Future work

This work can be extended upon in multiple ways. This section provides a non-extensive list of recommendations for future work.

7.3.1 Data sets

First of all, in this thesis all experiments were done with emulated data sets. This made it possible to asses qualitative performance measures for EDDG. Experiments could also be done with different data sets. Future work could be aimed at using EDDG to e.g. generalize data sets of some medical conditions to ethnic groups or age groups for which no data is available. Similar experiments to the ones performed here could also be done with other data than images.

7.3.2 GAN architectures

The usefulness for EDDG of different GAN architectures than those compared in this thesis could be researched. Firstly, other neural network architectures could be used for the generators and discriminators. Experiments could e.g. be performed with neural networks with residual blocks [20], which have shown to work well in GANs [19, 42]. Experiments could also be performed with spectral normalization [42], which is a recently developed input normalization technique that produces state-of-the-art results.

AC-GAN performance for domains that only differ in low level details might be increased if the domain specific information is presented at the last layer(s) of the generator. As with CoAC-GANs, this would force the class-specific features to be encoded in these latter layers.

Experiments could be performed in which the CoGAN architecture is combined with other GANs that can generate labeled data than AC-GANs, such as conditional GANs with a projection discriminator [43].

Experiments done with CoGANs suggest that optimal performance of CoGANs for regular data generation can be reached when only a single layer of the generator and a single layer of the discriminator is shared [37]. However, the influence of the number of shared layers in a CoAC-GAN trained for EDDG remains unknown and could be investigated.

Furthermore, it would be interesting to see what effect it would have to prime a CoGAN generator with both a domain and class label and let its discriminator predict both of these labels, similar to the way AC-GANs are trained for EDDG in this work. It would be useful to develop architectures for EDDG that do not depend on data set characteristics.

7.3.3 Image-to-image translation

It might also be possible to extend this work in such a way that the CoAC-GAN architecture can be used to perform image-to-image translation. Here, the target image could then be of a domain-class combination that is not present in the training data. One way to do this could be by implementing an operation that is the inverse of the non-shared layer(s) of the generator that generates images from the domain in which data of all classes is available. When this operation is applied to some real image of that same domain, the resulting output matches a latent code of features in the last shared layer of the generator. One could use this output as input for the non-shared layer(s) of the generator that generates images from the domain in which data of some class is missing. Performing both of these steps consecutively should result in image-to-image translation. This method could be used to generate images of a missing domain-class combination by using as input a real image that is of the missing class, but of the domain for which all classes are available.

The inverse operation described above has not been implemented in this thesis. It should be noted that for the generator architectures used in this work, the non-shared last layer(s) and this inverse operation cannot be bijective inverses of each other. This is because latent codes in the last shared layer of the generator have a larger dimensionality than the generated images. To implement the inverse operation in an exact way, an invertible convolution layer would have to be developed. Promising results in inverting convolutional neural networks have already been obtained [14]. Fully invertible 1×1 convolution layers have already been used successfully in generative models [27]. Future developments in this research area might be useful for the implementation of the inverse operation. It might also be useful to alter the generator architecture used in this work so that the dimensionality of the codes in the last shared layer of the generator is the same as that of the generated images. This would allow the non-shared generator layer(s) and the inverse operation to be bijective inverses of each other. Lastly, the reverse operation could also be learned by a separate machine learning model.

Bibliography

[1] Alpaydin, Ethem. *Introduction to Machine Learning.* MIT press, 2014. 6, 7, 13

[2] Arjovsky, Martin and Bottou, Léon. Towards Principled Methods for Training Generative Adversarial Networks. *arXiv preprint arXiv:1701.04862*, 2017. 2, 15

[3] Arjovsky, Martin and Chintala, Soumith and Bottou, Léon. Wasserstein GAN. *arXiv preprint arXiv:1701.07875*, 2017. 2, 12, 17, 18, 22, 30

[4] Barratt, Shane and Sharma, Rishi. A Note on the Inception Score. *arXiv preprint arXiv:1801.01973*, 2018. 21, 22

[5] Chen, Xi and Duan, Yan and Houthooft, Rein and Schulman, John and Sutskever, Ilya and Abbeel, Pieter. InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets. In *Advances in Neural Information Processing Systems*, pages 2172–2180, 2016. 3

[6] Choi, Yunjey and Choi, Minje and Kim, Munyoung and Ha, Jung-Woo and Kim, Sunghun and Choo, Jaegul. StarGAN: Unified Generative Adversarial Networks for Multi-Domain Image-to-Image Translation. *arXiv preprint arXiv:1711.09020*, 2017. 3, 12, 28

[7] Chongxuan, LI and Xu, Taufik and Zhu, Jun and Zhang, Bo. Triple Generative Adversarial Nets. In *Advances in Neural Information Processing Systems*, pages 4091–4101, 2017. 2, 3

[8] Cover, Thomas M and Thomas, Joy A. *Elements of Information Theory.* John Wiley & Sons, 2012. 13, 14

[9] Deng, Jia and Dong, Wei and Socher, Richard and Li, Li-Jia and Li, Kai and Fei-Fei, Li. ImageNet: A Large-Scale Hierarchical Image Database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009. 21

[10] Denton, Emily L and Chintala, Soumith and Fergus, Rob and others. Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks. In *Advances in neural information processing systems*, pages 1486–1494, 2015. 2

[11] Dmitry Ulyanov and Andrea Vedaldi and Victor S. Lempitsky. Instance Normalization: The Missing Ingredient for Fast Stylization. *CoRR*, abs/1607.08022, 2016. 11, 30

[12] Dumoulin, Vincent and Visin, Francesco. A Guide to Convolution Arithmetic for Deep Learning. *arXiv preprint arXiv:1603.07285*, 2016. 9, 10

[13] Eriksson, Kenneth and Estep, Donald and Johnson, Claes. *Applied mathematics: Body and soul: Volume 1: Derivatives and geometry in IR3.* Springer Science & Business Media, 2013. 17

[14] Anna C Gilbert, Yi Zhang, Kibok Lee, Yuting Zhang, and Honglak Lee. Towards Understanding the Invertibility of Convolutional Neural Networks. *arXiv preprint arXiv:1705.08664*, 2017. 45

[15] Goodfellow, Ian. NIPS 2016 Tutorial: Generative Adversarial Networks. *arXiv preprint arXiv:1701.00160*, 2016. 15, 16, 21

[16] Goodfellow, Ian and Bengio, Yoshua and Courville, Aaron. *Deep learning*, volume 1. MIT press Cambridge, 2016. 6, 9, 11, 13, 14

[17] Goodfellow, Ian and Pouget-Abadie, Jean and Mirza, Mehdi and Xu, Bing and Warde-Farley, David and Ozair, Sherjil and Courville, Aaron and Bengio, Yoshua. Generative Adversarial Nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014. viii, 2, 12, 14, 15, 16, 44

[18] Gulrajani, Ishaan and Ahmed, Faruk and Arjovsky, Martin and Dumoulin, Vincent and Courville, Aaron. Improved Training of Wasserstein GANs. *arXiv preprint arXiv:1704.00028*, 2017. 2

[19] Gulrajani, Ishaan and Ahmed, Faruk and Arjovsky, Martin and Dumoulin, Vincent and Courville, Aaron C. Improved Training of Wasserstein GANs. In *Advances in Neural Information Processing Systems*, pages 5769–5779, 2017. viii, 12, 17, 18, 19, 22, 45

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 45

[21] He, Kaiming and Zhang, Xiangyu and Ren, Shaoqing and Sun, Jian. Delving deep into rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015. 9

[22] Heusel, Martin and Ramsauer, Hubert and Unterthiner, Thomas and Nessler, Bernhard and Klambauer, Günter and Hochreiter, Sepp. GANs Trained by a Two Time-Scale Update Rule Converge to a Nash Equilibrium. *arXiv preprint arXiv:1706.08500*, 2017. 22

[23] Ioffe, Sergey and Szegedy, Christian. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv preprint arXiv:1502.03167*, 2015. 10

[24] Isola, Phillip and Zhu, Jun-Yan and Zhou, Tinghui and Efros, Alexei A. Image-to-Image Translation With Conditional Adversarial Networks. *arXiv preprint*, 2017. 3

[25] Khrulkov, Valentin and Oseledets, Ivan. Geometry Score: A Method For Comparing Generative Adversarial Networks. *arXiv preprint arXiv:1802.02664*, 2018. 21, 22

[26] Kim, Taeksoo and Cha, Moonsu and Kim, Hyunsoo and Lee, Jungkwon and Kim, Jiwon. Learning to Discover Cross-Domain Relations with Generative Adversarial Networks. *arXiv preprint arXiv:1703.05192*, 2017. 3, 28

[27] Diederik P Kingma and Prafulla Dhariwal. Glow: Generative Flow with Invertible 1x1 Convolutions. *arXiv preprint arXiv:1807.03039*, 2018. 45

[28] Kingma, Diederik P and Ba, Jimmy. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014. 8, 30

[29] Kingma, Diederik P and Welling, Max. Auto-Encoding Variational Bayes. *arXiv preprint arXiv:1312.6114*, 2013. 1, 16

[30] Alex Krizhevsky and Geoffrey Hinton. Learning Multiple Layers of Features from Tiny Images. Technical report, Citeseer, 2009. 21

[31] Krizhevsky, Alex and Sutskever, Ilya and Hinton, Geoffrey E. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. 1, 8

[32] Kullback, Solomon and Leibler, Richard A. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951. 14

[33] LeCun, Yann. The MNIST database of handwritten digits. *http://yann.lecun.com/exdb/mnist/*, 1998. 4, 27

[34] LeCun, Yann and Boser, Bernhard E and Denker, John S and Henderson, Donnie and Howard, Richard E and Hubbard, Wayne E and Jackel, Lawrence D. Handwritten Digit Recognition with a Back-Propagation Network. In *Advances in neural information processing systems*, pages 396–404, 1990. 5, 9

[35] LeCun, Yann and Bottou, Léon and Bengio, Yoshua and Haffner, Patrick. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 32

[36] Lin, Jianhua. Divergence Measures Based on the Shannon Entropy. *IEEE Transactions on Information theory*, 37(1):145–151, 1991. 13, 14

[37] Liu, Ming-Yu and Tuzel, Oncel. Coupled Generative Adversarial Networks. In *Advances in neural information processing systems*, pages 469–477, 2016. v, 3, 4, 5, 12, 20, 21, 27, 28, 30, 44, 45

[38] Long, Jonathan and Shelhamer, Evan and Darrell, Trevor. Fully Convolutional Networks For Semantic Segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015. 9

[39] Maas, Andrew L and Hannun, Awni Y and Ng, Andrew Y. Rectifier Nonlinearities Improve Neural Network Acoustic Models. In *Proc. icml*, volume 30, page 3, 2013. 8

[40] Makhzani, Alireza and Shlens, Jonathon and Jaitly, Navdeep and Goodfellow, Ian and Frey, Brendan. Adversarial Autoencoders. *arXiv preprint arXiv:1511.05644*, 2015. 3

[41] Mirza, Mehdi and Osindero, Simon. Conditional Generative Adversarial Nets. *arXiv preprint arXiv:1411.1784*, 2014. 2, 12

[42] Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral Normalization for Generative Adversarial Networks. *arXiv preprint arXiv:1802.05957*, 2018. 45

[43] Miyato, Takeru and Koyama, Masanori. cGANs with Projection Discriminator. *arXiv preprint arXiv:1802.05637*, 2018. 2, 3, 12, 45

[44] Nair, Vinod and Hinton, Geoffrey E. Rectified Linear Units Improve Restricted Boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010. 8

[45] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading Digits in Natural Images with Unsupervised Feature Learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, volume 2011, page 5, 2011. 27

[46] Nielsen, Frank. A Family of Statistical Symmetric Divergences Based on Jensen's Inequality. *arXiv preprint arXiv:1009.4004*, 2010. 14

[47] Nowozin, Sebastian and Cseke, Botond and Tomioka, Ryota. f-GAN: Training Generative Neural Samplers using Variational Divergence Minimization. In *Advances in Neural Information Processing Systems*, pages 271–279, 2016. 16

[48] Odena, Augustus and Olah, Christopher and Shlens, Jonathon. Conditional Image Synthesis With Auxiliary Classifier GANs. *arXiv preprint arXiv:1610.09585*, 2016. v, 2, 3, 12, 19, 44

[49] Oord, Aaron van den and Kalchbrenner, Nal and Kavukcuoglu, Koray. Pixel Recurrent Neural Networks. *arXiv preprint arXiv:1601.06759*, 2016. 3

[50] Pieters, Mathijs and Wiering, Marco. Comparing Generative Adversarial Network Techniques for Image Creation and Modification. *arXiv preprint arXiv:1803.09093*, 2018. 12, 28

[51] Radford, Alec and Metz, Luke and Chintala, Soumith. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *arXiv preprint arXiv:1511.06434*, 2015. 2, 10, 15, 16, 29

[52] Riedmiller, Martin and Rprop, I. Rprop-Description and Implementation Details, 1994. 7

[53] Rumelhart, David E and Hinton, Geoffrey E and Williams, Ronald J. Learning Representations by Back-Propagating Errors. *nature*, 323(6088):533, 1986. 7

[54] Sabour, Sara and Frosst, Nicholas and Hinton, Geoffrey E. Dynamic Routing Between Capsules. In *Advances in Neural Information Processing Systems*, pages 3859–3869, 2017. 12

[55] Salimans, Tim and Goodfellow, Ian and Zaremba, Wojciech and Cheung, Vicki and Radford, Alec and Chen, Xi. Improved Techniques for Training GANs. In *Advances in Neural Information Processing Systems*, pages 2234–2242, 2016. 2, 15, 16, 21

[56] Schaeffer, David G and Cain, John W. *Ordinary Differential Equations: Basics and Beyond*, volume 65. Springer, 2016. 17

[57] Sønderby, Casper Kaae and Raiko, Tapani and Maaløe, Lars and Sønderby, Søren Kaae and Winther, Ole. Ladder Variational Autoencoders. In *Advances in Neural Information Processing Systems*, pages 3738–3746, 2016. 2

[58] Szegedy, Christian and Vanhoucke, Vincent and Ioffe, Sergey and Shlens, Jon and Wojna, Zbigniew. Rethinking the Inception Architecture for Computer Vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016. 21

[59] Tieleman, Tijmen and Hinton, Geoffrey. Lecture 6.5-rmsprop: Divide the Gradient by a Running Average of its Recent Magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012. 7

[60] Upadhyay, Yash and Schrater, Paul. Generative Adversarial Network Architectures For Image Synthesis Using Capsule Networks. *arXiv preprint arXiv:1806.03796*, 2018. 12

[61] Villani, Cédric. *Optimal transport: old and new*, volume 338. Springer Science & Business Media, 2008. 17

[62] Vondrick, Carl and Pirsiavash, Hamed and Torralba, Antonio. Generating Videos with Scene Dynamics. In *Advances In Neural Information Processing Systems*, pages 613–621, 2016. 3

[63] Xu, Bing and Wang, Naiyan and Chen, Tianqi and Li, Mu. Empirical Evaluation of Rectified Activations in Convolutional Network. *arXiv preprint arXiv:1505.00853*, 2015. 8

[64] Yeh, Raymond and Chen, Chen and Lim, Teck Yian and Hasegawa-Johnson, Mark and Do, Minh N. Semantic Image Inpainting with Perceptual and Contextual Losses. *arXiv preprint arXiv:1607.07539*, 2016. 3

[65] Zhang, Han and Xu, Tao and Li, Hongsheng and Zhang, Shaoting and Huang, Xiaolei and Wang, Xiaogang and Metaxas, Dimitris. StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks. *arXiv preprint arXiv:1612.03242*, 2016. 2

[66] Ziwei Liu and Ping Luo and Xiaogang Wang and Xiaoou Tang. Deep Learning Face Attributes in the Wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015. 4, 5, 21, 28