Master's thesis

2019

Erlend Gjesteland Ekern

Master's thesis

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology and Electrical
Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Erlend Gjesteland Ekern

Information Maximizing and Pattern-Producing Generative Adversarial Networks

August 2019

NTNU
Norwegian University of
Science and Technology

Norwegian University of
Science and Technology

Information Maximizing and Pattern-Producing Generative Adversarial Networks

Erlend Gjesteland Ekern

Computer Science
Submission date: August 2019
Supervisor: Björn Gambäck

Norwegian University of Science and Technology
Department of Computer Science

Erlend Gjesteland Ekern

Information Maximizing and Pattern-Producing Generative Adversarial Networks

Master's Thesis in Computer Science, Spring 2019

Data and Artificial Intelligence Group
Department of Computer Science
Faculty of Information Technology and Electrical Engineering
Norwegian University of Science and Technology

Abstract

Recent years have seen a great deal of advances and progress in the field of generative models. These models often utilize neural networks to approximate the statistical properties of some kind of reference data like images, music, text, etc. The progress in this field has been especially evident in the image domain, where recent models have been able to generate images that exhibit a level of realism and detail that makes it hard to determine if a given image is computer-generated or not.

Most of these models take the same approach to image generation, namely generating an entire image at once. There is a more novel approach to image generation that pertains to generating only one pixel of an image at a time, effectively allowing for infinite resolution images. A neural network architecture that is based on this approach is the Compositional Pattern-Producing Network (CPPN). After a CPPN has been successfully trained, it can generate new images at arbitrary resolutions independent of the dimensions of the training data. Moreover, the synthetic images can exhibit different visual properties at different resolutions, all dependent on how the CPPN has been configured.

One of the most common ways to train generative models is using the adversarial framework offered by Generative Adversarial Networks (GAN). This framework effectively pits two neural networks against each other, where one, the discriminator, is concerned with recognizing generated (“fake”) outputs, while the other, the generator, tries to generate samples that the discriminator believes come from the training data (“real”). After successful training of a GAN, the generator is effectively an implicit approximation of the statistical properties of the training data, and should thus be able to generate new images that are similar to the data used when training the GAN.

While new architectures and techniques continue to improve the visual quality of images generated by GANs, the nature of generating entire images at once leads to increased computational requirements and necessitate access to large datasets of high-resolution images.

This thesis revolves around establishing the key properties of CPPNs, exploring how this more novel image generation approach can be taken when training GANs, and how such an approach can have merits both in terms of computational creativity and efficiency. Similar work exists, but the work presented here introduces experiments on previously untested datasets, as well as state-of-the-art results in terms of visual quality and realism of the generated images.

Sammendrag

En rekke teknologiske fremskritt og innovasjoner har dukket opp innenfor forskningsområdet rundt generative modeller i løpet av de siste årene. Denne typen modeller benytter seg ofte av kunstige nevrale nettverk som et verktøy for å trene en modell til å tilnærme seg en sannsynlighetsfordeling definert av referansedata som f. eks. bilder, musikk, tekst, osv. Fremskrittene innenfor dette området har vært spesielt synlige i bildedomenet, hvor nye modeller klarer å generere bilder som er så realistiske og detaljerte at det nesten er umulig å se at bildene er laget av en datamaskin.

De fleste slike modeller bruker samme fremgangsmåte når det kommer til generering av bilder, nemlig å generere hele bildet om gangen. Det finnes en annen metode som istedenfor genererer bare én bildepiksel om gangen, noe som gir mulighet for å generere bilder av uendelig høy oppløsning. En type nevrale nettverk som baserer seg på denne metoden er kalt Mønster-Genererende Nevrale Nettverk (CPPN). Etter at et slikt nettverk er ferdig trent, kan det generere nye bilder av vilkårlig oppløsning — uavhengig av oppløsningen på treningsdataene. Dessuten kan disse syntetiske bildene vise ulike visuelle egenskaper på ulike oppløsninger, alt avhengig av hvordan nettverket er konfigurert.

En av de vanligste måtene å trene generative modeller på er ved å bruke det adverserielle rammeverket som tilbys av Generative Adverserielle Nettverk (GAN). Dette rammeverket setter to nevrale nettverk opp mot hverandre, hvor det ene, diskriminatoren, prøver å gjenkjenne genererte ("falske") bilder, mens det andre, generatoren, prøver å generere bilder som diskriminatoren forhåpentligvis vil tro kommer fra treningsdataene ("ekte"). Etter vellykket trening av en GAN vil generatoren være en implisitt tilnærming av sannsynlighetsfordelingen som underligger treningsdataene, og vil dermed være i stand til å generere nye bilder som er veldig like bildene som ble brukt under treningen.

Mens nye arkitekturer og teknikker stadig forbedrer den visuelle kvaliteten av bildene som blir generert av GAN-er, fører den tradisjonelle tilnærmingen til bildegenerering til at det stilles stadig større krav til datakraft og tilgang til store sett treningsdata med høyoppløselige bilder.

Denne avhandlingen dreier seg i hovedsak om å etablere nøkkelegenskapene til CPPN-er, undersøke hvordan denne alternative tilnærmingen til bildegenerering kan brukes i sammenheng med GAN-er, og hvordan en slik tilnærming kan ha fordeler både når det kommer til kreativitet og effektivitet. Lignende arbeid finnes, men arbeidet som legges frem i denne avhandlingen introduserer eksperimenter på tidligere uprøvde sett med treningsdata, så vel som de beste resultatene når det gjelder visuell kvalitet og realisme i de genererte bildene.

Preface

This Master's Thesis is one of the necessary requirements for obtaining the degree of Master in Science in Computer Science from the Norwegian University of Science and Technology (NTNU). The work has been conducted at the Department of Computer Science (IDI) under the supervision of Björn Gambäck.

The thesis is based on an open-ended research project related to Computational Creativity.

I would like to thank my supervisor Björn Gambäck for valuable feedback along the way, and for giving me very free reins in terms of deciding the main topics of the thesis.

Erlend Gjesteland Ekern
Trondheim, 11th August 2019

Contents

List of Figures

List of Tables

1 Introduction

The usage of and research on generative models such as Generative Adversarial Networks (GANs) (Goodfellow et al., 2014) have seen enormous growth in the artificial intelligence community in recent years. GANs offer an adversarial framework for training generative models, where two networks, the discriminator and the generator, are pitted against each other: the discriminator is tasked with distinguishing between images from the dataset and synthetic ("fake") images from the generator, while the generator tries to generate samples that the discriminator classifies as coming from the training data ("real"). After successful training, the generator should be able to generate new samples that are very similar to the data the GAN was trained on. New and improved methods, architectures and applications are very frequently being introduced, and the quality of the generated results is constantly being improved. As an example, recent work (Karras and Aila, 2018) is able to generate large images of human faces that exhibit a level of realism that makes it almost impossible to tell if the images are real or computer-generated. An example of such generated images are shown in Figure 1.1.

Most of the mainstream research on generation of high-resolution images seem to revolve around the same approach, namely generating an entire image at once. An alternative approach is offered by the Compositional Pattern-Producing Network (CPPN) (Stanley, 2007), which generates the value of one pixel at a time and effectively allows for the generation of infinite-resolution images. While some previous work (Ha, 2016a, 2016b, 2016c; Metz and Gulrajani, 2017) has experimented with usage of CPPNs in GANs, this approach to image generation has not been studied much in the context of GANs. The results of previous work have been far away from reaching the quality possible using state-of-the-art methods. CPPNs do, however, exhibit properties that might have merits in other areas such as computational creativity and efficiency.

1.1 Motivation

In the early years of GANs, the networks were usually limited to training on and generation of images with small dimensionality, e.g. 28x28 pixels. In recent years, however, various innovations have made it possible to generate very detailed and realistic images of increasingly higher resolution. In practice, this has been done by introducing and utilizing newer, and perhaps more complex, architectures and techniques, and training the networks on larger datasets for longer periods of time. As proposed architectures become larger, and the required datasets become larger in size and of increased resolutions, many of these innovations lead to increased hardware requirements as well. As such, a lot of the current research in the field of GANs is focused on generating realistic images

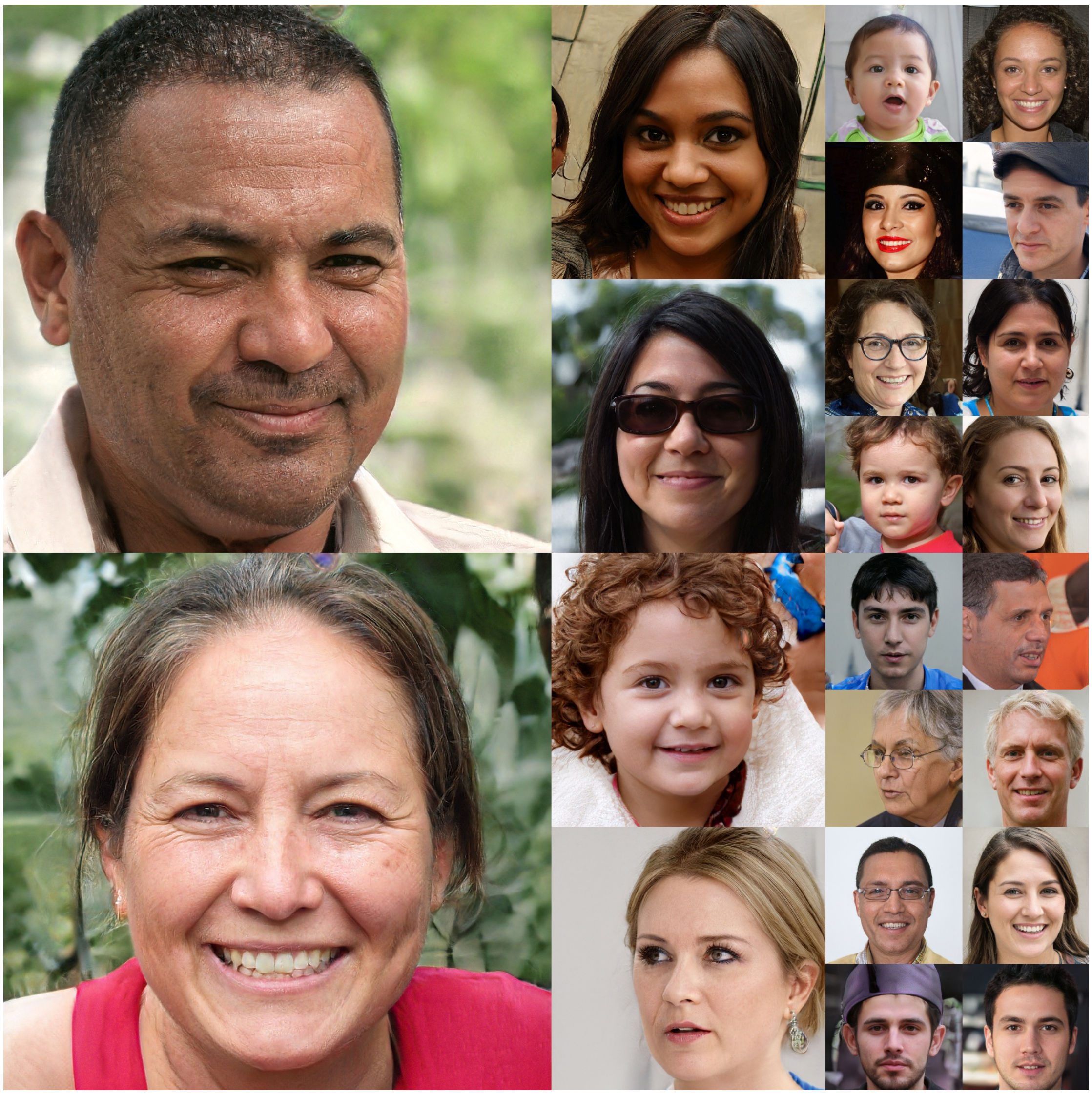


Figure 1.1: Images generated by the StyleGAN. (Adapted from Karras and Aila, 2018, reprinted with permission)

at increasingly higher resolutions. Computer-generated evaluation metrics such as the Fréchet Inception Distance (FID) (Heusel et al., 2017) and Inception Score (IS) (Salimans et al., 2016) are often used as a basis when introducing a new state-of-the-art architecture or technique.

Some novel architectures and approaches might be overlooked by, or never reach the attention of, the majority of the research community because they are not directly aligned with the overarching goals of improving the state-of-the-art in image quality and realism defined by some well-defined metric. An architecture that has seen little research in the context of GANs is the Compositional Pattern-Producing Network (CPPN). In short, CPPNs are able to generate images of infinite resolution regardless of the dimensions of the images they have been trained on. The state-of-the-art methods capable of generating images at resolutions such as 1024x1024 require both a dataset containing images of the aforementioned dimensions and well over a month of training time on a single powerful graphics processing unit (GPU) [1]. Conversely, a CPPN can be trained on much smaller datasets for a fraction of the training time, and afterwards be able to generate images of arbitrary dimensions. The dimensions of the training data naturally serve as an upper bound on the level of detail the generated images can contain, but in lieu of extra detail, other novel shapes, patterns and visual features may arise in the high resolution images. A successfully trained CPPN can thus not only approximate the visual characteristics of the training data at the original, low resolution, but also transfer these characteristics to very high resolutions while also exhibiting visual properties that are not otherwise apparent. This might additionally have merit in terms of computational creativity — a growing field of research that revolves around how computer systems can display behavior that can be deemed creative (Lamb et al., 2018).

Overall, this illustrates how exploring less-researched architectures that might perform worse than the state-of-the-art in hard metrics can still be worthwhile and net interesting results in the light of computational creativity and efficiency. The combination of CPPNs and GANs is such an architecture.

1.2 Goal and Research Questions

Thesis goal *Explore the usage of an unconventional generator architecture in a Generative Adversarial Network in the context of computational creativity and efficiency*
An architecture containing a CPPN as the generator in a GAN will be constructed and explored in the context of computational creativity and efficient generation of synthetic high-resolution images. This architecture will additionally utilize concepts from information theory to gain control over parts of the latent input, and the resulting architecture is introduced as an Information Maximizing and Pattern-Producing GAN (IMPGAN). A necessary precursor to this architecture includes the implementation of a performant and relatively stable GAN architecture using state-of-the-art architectures and techniques. This will lead to a technical and theoretical foundation that should prove quite crucial for improving the performance of the IMPGANs, where an unconventional generator architecture is utilized.

Research question 1 *How does the performance of an Information Maximizing and Pattern-Producing Generative Adversarial Network compare to its conventional counterpart?*
Several experiments will be carried out where GANs and IMPGANs with similar configurations are trained on the same datasets. The results will be compared, and similarities and differences will be highlighted and evaluated, with a focus

[1]Training time on a single Nvidia Tesla V100, as reported at `https://github.com/NVlabs/stylegan`.

on training stability, feature disentanglement and visual quality of low-resolution samples. Moreover, many of the techniques and architectures that have been proposed to improve the training stability and performance of GANs assume that the generative models used are of the conventional sort. It is thus of interest to see how a more unconventional generator, the CPPN, fares under such proposed schemes.

Research question 2 *How do different architectures, datasets and parameters affect images generated by an Information Maximizing and Pattern-Producing Generative Adversarial Network?*
A great deal of experiments will be conducted to analyze the effects different architectures and parameters have on the generated high-resolution images of an IMPGAN. Experiments using multiple datasets of different complexity will also be carried out. Additionally, experiments pertaining to stand-alone, untrained CPPNs will be conducted to establish the foundational properties of such networks. The key findings from the CPPN-only experiments should prove advantageous when constructing and evaluating more complex, trained architectures.

Research question 3 *How can the level of detail and realism be increased for the generated images of an Information Maximimizing and Pattern-Producing Generative Adversarial Network?*
After a solid technical and theoretical foundation has been established, experiments will be conducted with the goal of improving the level of realism and detail in the generated high-resolution images of an IMPGAN.

1.3 Contributions

1. *Introduced state-of-the-art results in terms of level of realism and detail for images generated by a trained Compositional Pattern-Producing Network.*

2. *A thorough study on how different architectures and parameters materialize in distinct visual features in the images generated by a Compositional Pattern-Producing Network.*

3. *A literary review of state-of-the-art methods for training Generative Adversarial Networks.*

4. *Implemented and introduced an architecture called Information Maximizing and Pattern-Producing Generative Adversarial Networks that perform well across a wide range of datasets and configurations.*

5. *Implemented a highly flexible architecture, including a suite of powerful visualization tools, that can serve as the basis for interesting future work.*

6. *Improved on related work by combining techniques for training stabilization with architectures aimed at achieving disentangled latent representations.*

1.4 Thesis Structure

- **Chapter 1** introduces the thesis, including the motivation behind the work, the overarching goal and research questions, and the contributions made.
- **Chapter 2** introduces the main background theory that underpins the subsequent chapters.
- **Chapter 3** presents earlier, related works that are similar in nature to the main contents of the thesis.
- **Chapter 4** describes the architecture that has been implemented and used to carry out the experiments, as well as the methodology that has been applied during the implementation phase.
- **Chapter 5** presents the experiments that have been conducted, accompanied by their purpose, configurations, results and brief evaluations.
- **Chapter 6** introduces an overall evaluation of the experimental results, a discussion on the merits of the work performed, as well as reflections on the extent to which the thesis goal and research questions have been met and answered, respectively.
- **Chapter 7** summarizes the key findings of the thesis, and outlines potential future work.

2 Background Theory

This chapter will introduce the background theory that underlies the subsequent chapters. Theory that might be very specific and useful in only one or few sections of the thesis are deferred to and introduced in its respective, later sections. The theory will mainly pertain to different types of neural networks, Generative Adversarial Networks (GAN) and relevant architectures, as well as practical details regarding the training of neural networks. As GANs are closely coupled with probability distributions and metrics, a brief section on relevant probability theory will be presented. Moreover, a key part of the work in this thesis is concerned with utilizing various concepts from information theory, and an introduction to the necessary concepts is thus required.

In terms of mathematical notation, lowercase, bold lowercase and bold uppercase letters refer to scalar values, vectors and matrices, respectively:

$$
\begin{aligned}
x &= 1 \\
\mathbf{y} &= \begin{pmatrix} 0 & 1 \end{pmatrix} \\
\mathbf{Z} &= \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}
\end{aligned}
$$

2.1 Computational Creativity

According to ProSecco (2018), computational creativity is an emerging field of research where the goal is to explore how computers can act as creators and co-creators by autonomously exhibiting behavior that outputs creative results. A general definition of creativity is quite hard to come up with given that it must encompass criteria that are valid in all domains where creativity occurs, e.g. sports, science, art and music. A common property of creativity is nonetheless the generation of something novel and surprising, and one of the main focus areas of computational creativity is to identify behaviors that lead to such results. This is in stark contrast to how computers traditionally have been used as mere tools to assist humans' creative endeavors. The level of human interaction involved with these systems determine the degree to which they are autonomous, that is, a system relying heavily on human input will only be semi-autonomous, while a system requiring little or no human input can be considered fully autonomous. Thus, the essence of the field is to explore how computers can act as more than mere tools to assist humans' creative endeavors, and actually be creative in their own rights (ProSecco, 2018).

2.2 Information Theory

The field of information theory was effectively started after Shannon's influential paper in 1948 (Cover and Thomas, 2006, p. xv). Its genesis was initally motivated by efficient transmittal of signals over noisy channels, e.g. sending messages over radio. Many concepts from information theory are also actively being applied to solve problems in machine learning, and what follows is an introduction to some of them.

2.2.1 Entropy

In the context of information theory, entropy is defined as the uncertainty of a random variable (Shannon, 1948). Given a random variable X, the entropy $H(X)$ is given by the following formula

$$H(X) = -\sum_{x \in X} P(x) \log(P(x))$$

The entropy can be regarded as the necessary amount of bits needed to describe a random variable: the higher the entropy, the more information the variable contains. The entropy is in fact a lower bound of the average number of bits necessary to describe a random variable (Goodfellow et al., 2016, p. 74).

2.2.2 Cross-Entropy

Given two probability distributions p, the true distribution, and q, an estimation of p, the cross-entropy between the two is given by

$$H_q(p) = -\sum_{x} p(x) \log q(x)$$

The cross-entropy is defined as the entropy of the distribution p given the coding scheme of q.[1] The cross-entropy $H_q(p)$ will always be larger than or equal to the entropy $H(p)$. If distributions p and q are equal, the cross-entropy will be equal to $H(p)$.

The concept of cross-entropy is quite useful in machine learning, and it is often used when training neural networks for classification tasks.

2.2.3 Relative Entropy

Relative entropy is a measure of the difference between two probability distributions. It is not a true distance metric, but it does, however, have some appealing properties: it is non-negative and zero iff the two distributions are identical to each other. Given two

[1] This is a non-standard notation for cross-entropy. The standard notation is $H(p, q)$, which is also used for the different concept of joint entropy. Thus, to avoid ambiguity this non-standard notation will be used when referring to cross-entropy.

probability distributions p and q, the relative entropy is given by

$$\begin{aligned} D(p||q) &= -\sum_{x\in X} p(x)\log\frac{q(x)}{p(x)} \\ &= -\sum_{x\in X} p(x)\log q(x) + \sum_{x\in X} p(x)\log p(x) \\ &= H_q(p) - H(p) \end{aligned}$$

where $H_q(p)$ is the cross-entropy between p and q. Thus, minimizing the cross-entropy with regards to q is equivalent to minimizing the relative entropy, as the term $H(p)$ is not dependent on q.

2.2.4 Mutual Information

The reduction in uncertainty for one random variable due to knowing another another random variable is known as mutual information (Cover and Thomas, 2006, p. 21). Given two random variables X and Y, the mutual information I between them can be defined in terms of entropy as

$$\begin{aligned} I(X;Y) &= H(X) - H(X|Y) \\ &= -\sum_{x} P(x)\log P(x) - (-\sum_{x,y} P(x,y)\log P(x|y)) \end{aligned} \tag{2.1}$$

Equation 2.1 will be zero iff X and Y are independent variables.

The concept of mutual information has successfully been applied to Generative Adversarial Networks (GAN) in the form of the so-called InfoGAN (Chen et al., 2016), and it effectively allows for control of certain visual features in the output — as opposed to the generation of entirely random images.

2.3 Probability Theory

This section introduces some key concepts relating to probability distributions and different ways of measuring the difference between them. This theory underpins many generative models, including the variational autoencoder (VAE) (Kingma and Welling, 2013) and Generative Adversarial Networks (GAN) (Goodfellow et al., 2014).

2.3.1 Distributions

A probability distribution describes the likeliness of each possible state of a random variable.

Some common probability distributions include:

- **Categorical distribution**
  A discrete probability distribution where a random variable X has k possible categories, each with their own probability. The probabilities of all of the categories must sum to 1, that is $\sum_{i=1}^{k} P(X = x_i) = 1$.

- **Gaussian (or normal) distribution**
  A continuous probability distribution that often arises in nature. The distribution is symmetric around its mean μ and has a spread defined by its standard deviation σ.

- **Uniform distribution**
  A distribution, either continuous or discrete, where each value in a given interval has equal probability. Often used for generating random numbers in computer programs.

2.3.2 Distance Measures

In can often be useful to compare the dissimilarity or distance between probability distributions. In general, a function is a distance measure iff it is non-negative, symmetric, satisfies the triangle inequality and is zero iff the values being compared are equal. A divergence only satisfies the first and last of these constraints, but it can be useful to think of it as a type of distance even though this is not strictly true. Some notable divergence and distance functions include:

- **Kullback-Leibler divergence**
  The Kullback-Leibler (KL) divergence is the same as the relative entropy in Section 2.2.3.
  $$D_{KL}(p||q) = H_q(p) - H(p)$$

- **Jensen-Shannon divergence**
  The Jensen-Shannon (JS) divergence (Lin, 1991) is an extension to the KL divergence that results in a symmetric and smoother function.
  $$D_{JS}(P\|Q) = \frac{1}{2}D_{KL}(P\|M) + \frac{1}{2}D_{KL}(Q\|M)$$
  where $M = \frac{1}{2}(P+Q)$.

- **Earth-Mover distance**
  The Earth-Mover (EM) distance is defined as the minimum work needed to transform one probability distribution into another. This measure has been shown to exhibit nicer properties in the context of machine learning compared to both KL- and JS divergence (Arjovsky et al., 2017). Given two probability distributions P_r and P_g, the EM distance between them is defined as
  $$\begin{aligned} W(P_r, P_g) &= \inf_{\gamma \in \Pi(P_r, P_g)} \int_x \int_y \|x-y\|\, \gamma(x,y) dy\, dx \\ &= \inf_{\gamma \in \Pi(P_r, P_g)} \mathbb{E}_{(x,y)\sim\gamma}[\, \|x-y\| \,] \end{aligned} \tag{2.2}$$
  where $\Pi(P_r, P_g)$ is the set of all joint distributions $\gamma(x,y)$ whose marginals are P_r and P_g, respectively, and each γ can be considered a transport plan for transforming P_r into P_g. That is, for all x, y move $\gamma(x,y)$ mass from x to y. The EM distance is then the cost of the optimal transport plan.

2.4 Artificial Neural Networks

This section introduces the foundations of artificial neural networks (ANN), as well as some relevant architectures.

2.4.1 Foundations

Historically artificial neural networks (ANN) have been used as an abstraction for how the neurons in the brain operate, but advances in neuroscience have proven this to be a major simplification. Mathematical models based on this simple abstraction have, however, proven to have many useful properties in terms of distributed computation, tolerance of noisy inputs and learning ability, and it has been and continues to be an effective tool in the field of machine learning (Russell et al., 2010, p. 728).

An ANN is a network of computational units known as perceptrons. In such a network the perceptrons, or just units, are often organized in layers. When a network contains at least two such layers of units, it is called a multilayer perceptron (MLP). A simple example of an MLP is shown in Figure 2.1, where there is one layer of inputs, one hidden layer of units, and one final output layer of units. In an MLP, each unit[2] in a layer i is connected to every unit in the next layer $i + 1$ in a so-called fully-connected manner. Additionally data only flows in one direction, which in Figure 2.1 would mean from left to right. Each unit in an MLP usually has both weights associated with all of its inputs, as well as a weight associated to what is called the bias — a term with a constant value of 1 that increases the expressiveness of each unit. The output of each unit in an MLP is strictly defined by its inputs (and bias), the weights associated with these values, as well as the activation function used. When speaking of ANNs (often shortened to neural networks in the context of artificial intelligence) in the general sense, an MLP is usually implied — either fully- or sparsely-connected. Furthermore, it has been proved that an MLP can represent or very closely approximate any given function (Russell et al., 2010, p. 732). For a function f it might be computationally infeasible to construct and train an MLP to represent this function using current methods, but it does nonetheless theoretically exist. This further underlines the generality and expressive power of these networks.

Some kind of objective must be construced in order to train a neural network. The objective is often based various error or loss functions that approximates the performance of the network in some kind of way. As an example, an objective in the case of training a neural network to correctly classify images as containing either a dog or a cat can be defined as a $E(y, \hat{y})$, where E is some function that calculates the error between the true label y of the input image and the network's predicted label $\hat{y}$. For n labeled input images and predicted labels, one type of error function can be defined by the mean-squared error (MSE), $E_{MSE}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$. In practice most classification networks use the softmax function on the outputs to normalize them in the range $[0, 1]$ such that they sum up to 1, and each predicted label can as such be considered the probability of the

[2]Or each input in the case of the input layer.

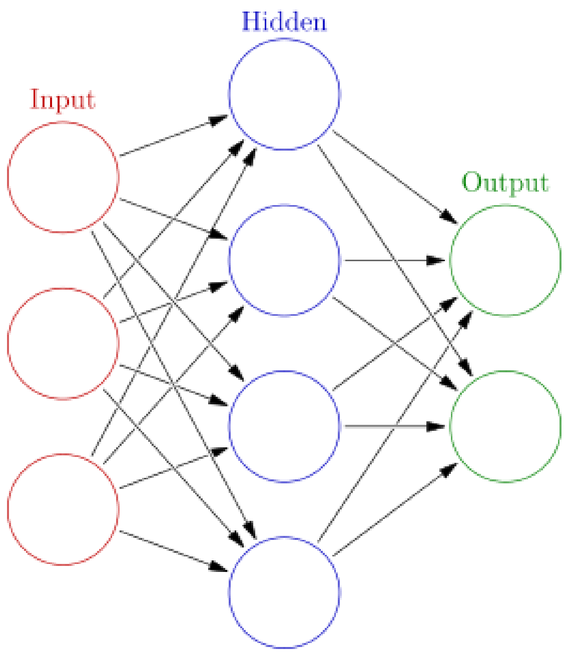


Figure 2.1: An example of a simple MLP with an input layer, a hidden layer and an output layer. Bias terms are not shown. (Reprinted from Glosser.ca, 2013 with permission)

given label. The cross-entropy between these predicted labels and the true labels are then calculated as the final output of the objective function.

An effective method for training neural networks to minimize a loss function is called backpropagation (Russell et al., 2010, p. 733). This method propagates the calculated output error backwards through the layers by using a technique called gradient descent, which entails changing the weights of the network in the direction that decreases the loss function the most. The size of these weight adjustments are determined by the learning rate and optimization process: a straight-forward implementation can utilize a fixed learning rate to adjust the weights. If the learning rate is set too high or too low, however, the loss function may never reach its minimum. The learning rate is a very sensitive parameter, and there exist many different optimization algorithms that improve training stability and speed by decreasing the learning rate over time, using different learning rates for each parameter in the network and using past gradients to influence the current update. Some of the more popular algorithms incorporating such techniques are the Adam, AdaGrad and RMSProp optimizers (Goodfellow et al., 2016, p. 309).

Training a neural network usually involves dividing the dataset into n mini-batches of equal size, and using an optimization algorithm on these mini-batches (as opposed to the entire dataset).

There exist some ambiguity in the literature when referring to many of these concepts, so a short clarification on the relevant terminology that will be used in this thesis is as follows:

- A **batch** is a mini-batch.
- The **batch size** is the size of each batch.
- For a dataset of 100 000 samples and a batch size of 200, one **epoch** is complete when the network has been trained on $\frac{100\,000}{200} = 500$ batches, i.e. a full cycle of the dataset. Training is often set up to run for a specified number of epochs, e.g. 400, after which training will stop.
- The **training step** is the total number of batches that the network has been trained on.

An important thing to note is that for an MLP to be able to represent non-linear functions, it needs to use non-linear activation functions (Goodfellow et al., 2016, p. 168). Historically, non-linear functions such as sigmoid or hyperbolic tangent have been used as activation functions for the units. Most neural networks today consist of rectified linear units (ReLU). They are simply units using the activation given by $f(x) = \max(0, x)$ (Goodfellow et al., 2016, p. 171). ReLUs have a big advantage of being easy to optimize given their similarity to linear units (Goodfellow et al., 2016, p. 189). They can, however, lead to training issues if a lot of the activations end up being 0. There exist a generalized version of ReLU called leaky ReLU (LReLU) that improves upon this situation by defining the activation as $f(x) = \max(0, x) + \alpha \min(0, x)$, where α is set to some small number (e.g. $\alpha = 0.2$). This results in usable gradients even if x, which the activation function is used on, is less than or equal to zero.

Another key concept in improving the training of ANNs is regularization. Regularization is a modificiation to the learning algorithm with the goal of improving the network's generality, possibly at the expense of an increased training error (Goodfellow et al., 2016, p. 120). Commonly used regularization schemes involve adding a penalty of sorts to the objective function. An example is using L^2 regularization on the weights of the neural network in order to encourage a model with lower weights.

Furthermore, the initial values of the parameters in the networks can have a large effect on the training, and can in the worst case cause the network to never converge. If the initial parameters are too large, it may lead to exploding values, or saturation of units containing activation functions like sigmoid or tanh. Weights are usually set according to a random distribution and possibly scaled and normalized, while biases are often initialized to a constant value (Goodfellow et al., 2016, p. 302–303). In practice biases are often initialized to 0 while weights are usually initialized close to 0. Some commonly used weight initialization schemes are Xavier[3] (Glorot and Bengio, 2010), He (He et al., 2015) or simply drawing from a Gaussian distribution. Both Xavier and He use the number of connections associcated with a unit to dynamically set the initial weights,

[3]The Xavier weight initialization scheme is sometimes referred to as Glorot.

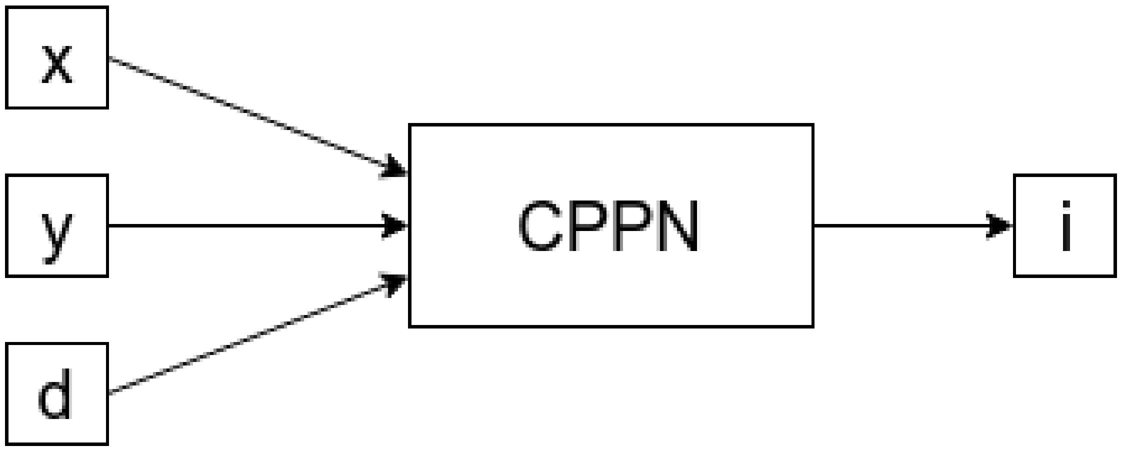


Figure 2.2: An example of the architecture of a CPPN. x and y are pixel coordinates, while d is the distance between these coordinates and the center of the image. For a greyscale image, i is the pixel intensity for the input pixel.

but Xavier takes both incoming and outgoing connections into account, while He only considers the incoming.

Finally there exist a great deal of miscellaneous techniques and algorithms that have been shown to improve the optimization process. Some recent normalization schemes that have been shown to improve the training and performance of neural networks include spectral normalization (Miyato et al., 2018), batch normalization (Ioffe and Szegedy, 2015) and layer normalization (Ba et al., 2016).

2.4.2 Various Architectures

In addition to MLPs, there exist many other neural network architectures. This section will introduce the essential and basic neural network architectures that will be utilized later in this thesis. More complex architectures are deferred to Section 2.5.

Compositional Pattern Producing Networks

Compositional Pattern-Producing Networks (CPPNs) were introduced by Stanley (2007) in the context of evolutionary computation. The network architecture was proposed as a novel abstraction of how natural development occurs. In short, Stanley presents six general patterns that are found in nature, and he argues that CPPNs are capable of producing all of these, thus justifying CPPNs as valid abstractions. A CPPN's ability to generate the aforementioned patterns is exemplified by the generation of images containing these patterns, but Stanley notes that the visual depictions are for illustrative purposes only — arguing that CPPNs can potentially function as general pattern generators across a wide range of domains, not only in the image domain.

A CPPN in its simplest form can be described as taking a coordinate as an input, and outputting a value for that coordinate. The values for each coordinate in an arbitrary coordinate system can then be calculated by inputting each coordinate, one by one, into the CPPN. CPPNs are very similar to neural networks in that they utilize inputs and inter-connected units with weights and activation functions to calculate an output, but Stanley makes a clear distinction between them: CPPNs are related to natural development, while neural networks represent a simplified model of mechanisms found in the human brain. While the components of a CPPN are very similar to the ones found in neural networks, the hierarchy of layers, direction of edges, etc., can otherwise be arbitrary. CPPNs have traditionally been iteratively evolved from simple networks to networks of increasing complexity using evolutionary algorithms (Stanley, 2007). Each new generation of networks will then be a mutated crossover of several parent networks, with new units, edges and activation functions being added at random. Evolving CPPNs in this way often leads to sparsely-connected network structures much less organized than in traditional neural networks.

The remainder of this thesis will focus on CPPNs as generative models in the image domain, and the term CPPN will be used to describe neural network architectures whose main purpose is to output a pixel intensity or color given a single coordinate as input. Moreover, fixed as opposed to evolved CPPN architectures will be explored, and as such they will for all intents and purposes be normal neural networks.

As an example of image generation, a two-dimensional greyscale image can be generated by a CPPN by constructing a neural network with two inputs x and y (coordinates) and a final output i (pixel intensity). The inner architecture of the network can be arbitrary. In order to generate a 28x28 image, the CPPN would then be tasked with generating the pixel intensity for each pixel at the desired image resolution — in this case, a total of 28^2 times. A high-level example of a CPPN is shown in Figure 2.2. In this example an additional term d is given as input, defined as the distance between the input coordinates and the center of the coordinate system. The inclusion of this term provides a bias towards symmetry in the output (Stanley, 2007).

Different activation functions in a CPPN give rise to different patterns in the output (Stanley, 2007): using a Gaussian or absolute activation function will give rise to symmetric structures, while a sine activation function will give rise to repeating structures. Likewise, various other functions contribute to the sharp edges, circles, etc. found in the generated output image. One of the most interesting properties of CPPNs is the fact that they can generate images of infinite resolution. This can be done by transforming pixel coordinates — e.g. $(0, 0)$ or $(10, 200)$ — to a fixed continuous space, e.g. in the interval of $[-1, 1]$. Thus, in order to increase the resolution of the images generated by the CPPN, one only needs to increase the granularity of the input coordinates (Stanley, 2007). Figure 2.3 shows the coordinates that would be input to a CPPN in order to generate a 3x3 images. Note that in order to generate a 100x100 image, the only difference would be the level of detail of the input values. Instead of dividing the values of the interval $[-1, 1]$ between three evenly spaced values, the same would be done for hundred values: $[-1, -0.97979798, -0.95959596, ..., 0.95959596, 0.97979798, 1]$.

Figure 2.3: An example of the (x, y) input coordinates required to generate a 3x3 image using a CPPN. The coordinates are normalized to $[-1, 1]$.

$$\begin{bmatrix} (-1,-1) & (0,-1) & (1,-1) \\ (-1,\ \ 0) & (0,\ \ 0) & (1,\ \ 0) \\ (-1,\ \ 1) & (0,\ \ 1) & (1,\ \ 1) \end{bmatrix}$$

Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a neural network that contains one or more convolutional layers. These types of networks have been around for a long time, and they were specifically designed to improve recognition of 2D-shapes by exhibiting invariance with respect to transformations and distortions of the inputs (LeCun et al., 1998). After a CNN achieved considerably higher accuracy than the previous state-of-the-art in image classification in 2012 (Krizhevsky et al., 2012), the usage of, interest in and research on these types of networks, and deep learning in general, have seen a great increase (Goodfellow et al., 2016, p. 371).

In the case of image classification using a multilayer perceptron (MLP), there is usually one input for every pixel in the image. For a 512x512 image, this would result in 262 144 inputs. If the first hidden layer contains 100 units, this would result in roughly 2.6 million weights. These classification networks usually have a much deeper architecture than this, and given that the network is fully-connected, the number of needed to be stored in memory is going to be very large. CNNs have fewer edges and weights than traditional neural networks with similarly-sized layers, and thus pose less computational requirements and are easier to train (Krizhevsky et al., 2012).

While a traditional neural network layer uses matrix multiplication, a normal convolutional layer can usually be divided into three stages (in the given order): convolution of input, application of non-linear activation function, and finally downsampling (Goodfellow et al., 2016, p. 335). Convolution in the context of CNNs involves sliding a small matrix — often called a filter or kernel — over the entire input and calculating the dot product between the filter and the values at each window position. A filter works as a feature extractor, and the parameters of a filter are learned during the training of the network. In the first layers of the network, these filters will usually learn to recognize edges, while filters deeper in the network will learn to recognize higher-order features such as circles, eyes, ears, etc. depending on the training data. Many different filters can be applied to the same input.

The output of sliding this filter over the input results in what is often called a feature map. After the feature map has been calculated, a non-linear activation function such as ReLU is usually applied, followed by a pooling, or subsampling, operation. Pooling is used to downsample the data, and effectively summarize the feature map. This operation makes the layer more invariant to various translations of the inputs. The most often used

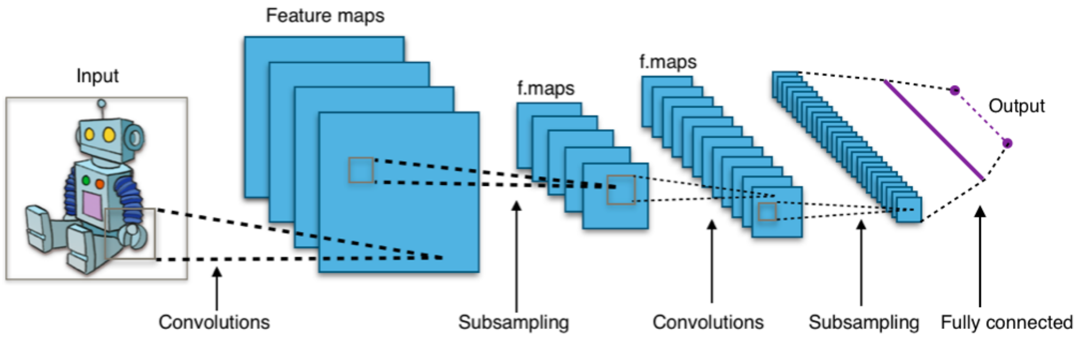


Figure 2.4: An typical CNN architecture. (Reprinted from Aphex34, 2015 with permission)

type pooling is called max-pooling, but other variants also exist (Dumoulin and Visin, 2016).

During both convolution and pooling, the distance between two consecutive windows is called the stride. A higher stride will result in reduced dimensionality compared to a lower stride.

An example of a typical CNN architecture is given in Figure 2.4, where convolutional and pooling layers are used, while the final hidden layer is fully-connected that typically uses softmax as an activation function to predict the class probability of the image.

In short, due to their architecture, CNNs are much more computationally efficient than fully-connected feed-forward neural networks (Goodfellow et al., 2016, p. 330). This, in addition to being invariant to scale, rotation, location and noise, leads to very high performance in image classification (Krizhevsky et al., 2012).

Additionally, the pipeline in a convolutional layer can be reversed by what can be referred to as a transposed convolutional layer. Instead of downsampling the input, as is often done in a convolutional layer, the transposed convolutional layer upsamples its input. Such a layer usually employs some sort of unpooling to upsample the input, followed by an application of a non-linear activation function and then finally convolution (Dumoulin and Visin, 2016). These layers can be beneficial in the context of image generation.

Residual Neural Networks

Historically it has been problematic to train deep[4] neural networks because of exploding and vanishing gradients, which make it difficult for the network to learn and reach convergence. Proposed weight initialization schemes and normalization techniques have improved such issues, but even with useful gradients the performance of deep networks tends to degrade as the depth increases (He et al., 2016). He et al. propose a deep residual learning framework to combat this degradation issue. The framework introduces

[4]A deep neural network is simply a network containing many layers.

the concept of a residual block, which is a stack of usually two or three layers where the input to the first layer is also passed directly to the last layer, allowing the block to skip the intermediate layers using a so-called skip- or shortcut-connection. If the weights of the layers in the residual block are initialized close to 0, the initial output of the block is close to the identity function of the input. As training progresses the weights of the intermediate layers inside the residual blocks are adjusted to have more or less influence over the output of the block.

A simple example of an MLP layer architecture versus a residual block is given in Figure 2.5. The output of the final layer in Figure 2.5a is solely determined by its inputs, weights, biases and activation function. In the residual block in Figure 2.5b, however, the input is added to the final layer before the result is being passed through the activation function. In practice a there might be various normalization layers inside the residual block, as well as a resizing layer that make the dimensions of the skip-connection match the dimensions of the final output.

A residual neural network can be constructed by stacking multiple residual blocks on top of each other, and He et al. show that such deep residual networks perform better than their standard neural network counterparts.

2.5 Generative Adversarial Networks

This section will introduce the Generative Adversarial Network (GAN), as well as various architectures associated with this adversarial framework.

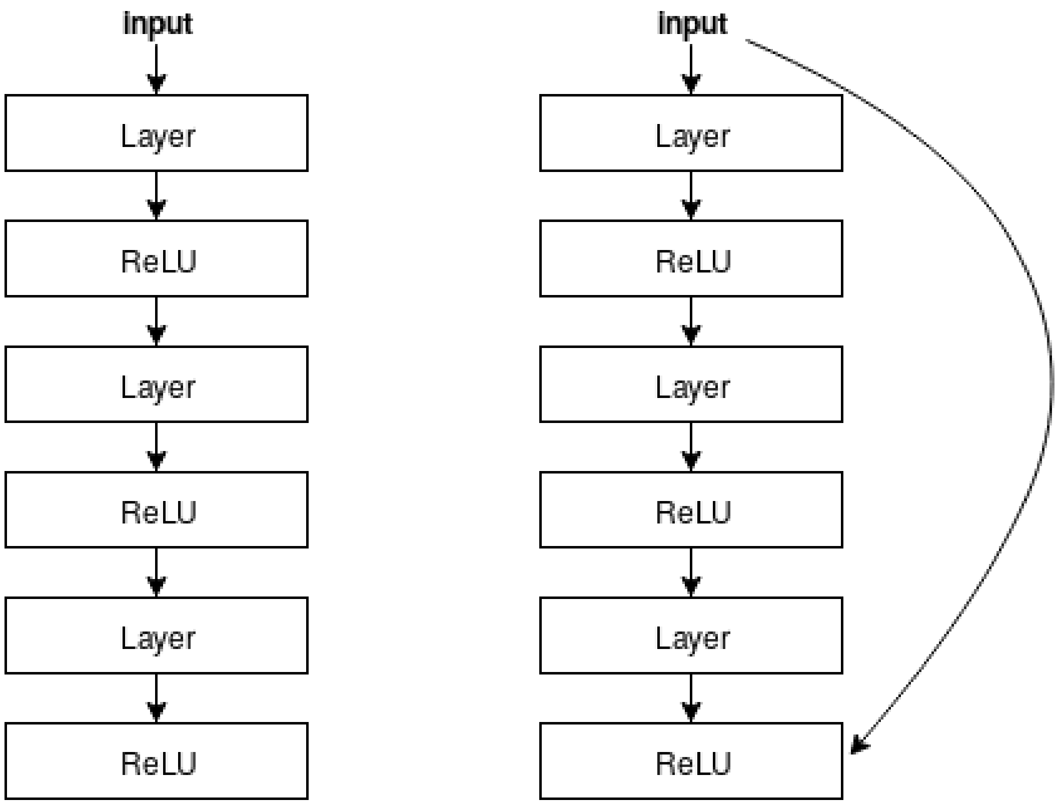
2.5.1 Original GAN

The term "original" here refers to the initial formulation of the Generative Adversarial Network.

A recent and influential framework for training generative models is the Generative Adversarial Network (GAN) (Goodfellow et al., 2014). A GAN pits a generative model, the generator, against a discriminative model, the discriminator, in what corresponds to a minimax two-player game. The value function for this minimax game is shown in Equation 2.3. The generator aims to learn the probability distribution of the training data, while the discriminator aims to classify samples as coming from either the training or the generator distribution. As such, GANs are a framework for training generative models through an adversarial process. In the original GAN paper they use MLPs for both the discriminator and generator (Goodfellow et al., 2014).

$$\min_G \max_D V(D,G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))] \tag{2.3}$$

In Equation 2.3, G is the generator function and D is the discriminator function. The output of the discriminator for a given input x, $D(x)$, is a real number $D(x) \in [0, 1]$. If the output is close to 0 or 1, the discriminator believes the input comes from p_g, the generator distribution, or p_{data}, the data distribution, respectively. The distribution of



(a) A standard neural network containing three layers (b) A neural network using a skip-connection between the layers

Figure 2.5: The operations in a given layer is limited to multiplying the weights with the inputs and adding biases. A separate block has been designated for the ReLU activation function.

the generator is implicity defined by the samples generated by $G(z)$, where $z \sim p_z$. In practice, a simple distribution like a unit Gaussian can be used as a prior such that $z \sim \mathcal{N}(0, 1)$. The idea behind using a GAN is then to train the discriminator and generator such that p_g is a good estimate of the actual data distribution p_{data}. This is effectively done by training the generator to map an n-dimensional latent noise vector z to samples that are similar to the ones found in the training data.

An informal example of the training of GANs is as follows: Imagine you want to start out in the lucrative business of art forgery. You are not particularly good at it when you first begin, and an art expert can easily differentiate between an original piece and your forgery. However, given that you have access to the art expert's reasons for labeling your work a forgery, you can use this knowledge to improve your future forgeries. This will hopefully make them a bit more realistic and harder to label as fake each time. At the same time, imagine that the art expert is not only exposed to your counterfeits, but also to more and more pieces of original art, thus improving his skill even more in distinguishing between originals and counterfeits. The art expert will thus be increasingly harder to fool, forcing you to improve your forgery skills. This process will continue until the art expert is unable to tell if any given piece of art is an original or a counterfeit produced by you, the forger. At this point, the forgeries are identical to original pieces in the eyes of the expert.

A problem with the minimax objective in Equation 2.3 is that as the discriminator improves its classification accuracy, the gradients provided to the generator will become increasingly smaller, thus making it harder for the generator to learn well (Goodfellow et al., 2014). Especially in the early stages of training this is problematic, as the discriminator is able to distinguish between a real and a generated sample with fairly high confidence. This can be improved by changing the objective function such that the generator maximizes $\log D(G(z))$ instead of minimizing $\log(1 - D(G(z))$ (Goodfellow et al., 2014). This objective will be referred to as the non-saturating objective. In using this modified objective, the GAN can no longer be considered a minimax game, but the formulation in Equation 2.3 does, in any case, lend itself well to theoretical analysis. Among other things, it has been used to show that learning in a GAN resembles the minimization of the Jensen-Shannon divergence (Section 2.3.2) between p_{data} and p_g (Goodfellow et al., 2014).

Newer research, however, suggests that despite changing the objective function, the gradients supplied to the generator tend to become worse as the discriminator becomes better (Arjovsky and Bottou, 2017). Additionally, a problem referred to as mode collapse has been widespread in GANs since their inception. When a mode collapse occurs, the generator maps many different latent noise vectors to the same generated output in order to approximate the diversity of p_{data} (Goodfellow et al., 2014). In such a case the generator might still be able to fool the discriminator, since the quality of the generated samples might be comparable to the ones in the training data. The generator will, however, lack the diversity of the training data. As an example, if mode collapse occurs when training a GAN on a dataset containg images of the numbers $1 - -9$, this could lead to the generator only being able to produce images belonging to a subset of these ten digits — maybe only 1s. This is a big problem, as the generator distribution is obviously not a good approximation of the data distribution if it is only capable of producing a subset of the original data.

2.5.2 Deep Convolutional GANs

A Deep Convolutional GAN (DCGAN) is a proposed set of architectural guidelines with the purpose of stabilizing the training of GANs (Radford et al., 2015). In contrast to standard GANs where MLPs are employed, a DCGAN models both discriminator and generator as CNNs. Traditional CNN architectures, however, have not worked well in GANs, and Radford et al. identify and propose a set of architectural guidelines that result in a stable training process across a wide variety of datasets, all the while allowing for the generation of high-resolution images and usage of deep networks. The insights brought forward by the DCGAN architecture have had a big influence on the field of GANs, and most GANs are in some way based on the DCGAN architecture (Goodfellow, 2016).

Radford et al. propose the following architectural guidelines:

1. Use convolutions instead of pooling layers (Section 2.4.2) in the discriminator.
2. Use transposed convolutions instead of pooling layers in the generator.
3. Use batch normalization (Section 2.4.1) in most layers.
4. Remove fully-connected layers.
5. Use ReLU (Section 2.4.1) in all layers of the generator except for the output, which uses tanh.
6. Use LReLU (Section 2.4.1) in all layers of the discriminator except for the output, which uses sigmoid.
7. Use the Adam optimizer (Section 2.4.1) with tuned hyperparameters when training.

Another important contribution made by Radford et al. was demonstrating that the generator is capable of learning dense feature representations using unsupervised data. These representations are encoded in the latent noise vector $\mathbf{z}$ that is input to the generator, and vector arithmetic can be performed on these vectors to predicatably manipulate certain qualities of the generated samples. Finding the latent representation of a specific feature can be done by manually collecting n generated images that contain the feature in question, and then average the latent vectors that produced them [5]. As an example, let the average of three latent noise vectors for men with glasses be $\mathbf{z}_1$, men without glasses be $\mathbf{z}_2$, women without glasses be $\mathbf{z}_3$. Then, $\mathbf{z}_1 - \mathbf{z}_2 + \mathbf{z}_3 = \mathbf{z}_4$, where inputting $\mathbf{z}_4$ to the generator would result in an image of a woman with glasses. Even though the latent noise vectors contain various representations of different features, the features are entangled with random noise. There is no apparent way to find the vectors that represent features such as man, woman or glasses without manually going through individual samples containing these features, averaging their latent vectors and hoping that this averaged vector will generate an image representative of the desired features. This is in contrast to disentangled representations, where features are disentangled from random noise and represented in a more isolated way.

The vector arithmetic here is somewhat analogous to how recent techniques such as Word2vec (Mikolov et al., 2013) and GloVe (Pennington et al., 2014) are able to represent words as dense, low-dimensional and real-valued vectors that capture semantic similarities and relationships between words, and it demonstrates how such dense vectors can have powerful applications across a wide range of domains.

[5] In practice, using just a single image sample can turn out to net unpredictable results. Radford et al. use $n = 3$.

2.5.3 Wasserstein GANs

The Wasserstein GAN (WGAN) uses an objective function which approximates the minimization of the Earth-Mover (EM) distance[6] (Section 2.3.2), and it has displayed convergence across a wide variety of architectures and datasets without the need for careful tuning of hyperparameters (Arjovsky et al., 2017). It suffers from far less mode collapse than both the minimax and non-saturating GAN objectives, and the value of the WGAN objective empirically correlates to the quality of generated samples (Arjovsky et al., 2017). In short, the WGAN objective seems to be much more forgiving in terms of architectural, technical and hyperparameter choices compared to traditional GANs[7]: displaying a more stable training process and good convergence when other models suffer from mode collapse or fail to converge entirely.

The formulation of the EM distance in Equation 2.2 is highly intractable, but Arjovsky et al. present an efficient approximation of an equivalent formula. The output of the discriminator is changed from being a probability to a scalar, and it is as such no longer a binary classifier [8]. For certain mathematical properties to hold, the weights of the discriminator θ_D are forced be in an interval $\theta_D \in [-c, c]$, where c is a small, real number (Arjovsky et al., 2017). This constraint is enforced by clipping the weights after every weight update. Arjovsky et al. do, however, note that weight clipping is not a particularly good way of enforcing the mathematical constraints as it can lead to optimization problems, decreased capacity of the discriminator, and vanishing or exploding gradients.

An improved version of the WGAN, dubbed WGAN-GP, uses a regularization technique called gradient penalty instead of weight clipping (Gulrajani et al., 2017). An additional term is added to the objective function which penalizes the discriminator for having a gradient norm larger than 1, i.e. the discriminator is incentivized to keep the norm of the gradients bounded. Experiments show that WGAN-GP demonstrates better convergence and stability, reduced training speed and higher quality samples than the earlier WGAN (Gulrajani et al., 2017). Further experiments show that DCGANs are capable of producing somewhat better images than WGAN-GPs, but similar to WGANs, WGAN-GPs have a more stable training process and are more robust to changes in hyperparameters than many other variants of GANs.

As the discriminator in both WGAN and WGAN-GP function as an approximator of the EM distance, the more accurate the approximation, the higher the quality of the gradients fed back to the generator. For this reason Arjovsky et al.; Gulrajani et al. suggest at each training step to train the discriminator n times while only training the generator once. This is in contrast to traditional GAN training, where there has often

[6]The EM distance also goes under the name Wasserstein-1, hence the name of the architecture: Wasserstein GAN.

[7]Traditional GANs refer to minimax and non-saturating GANs where the architecture is based on MLPs or DCGANs.

[8]Arjovsky et al. use the term *critic* instead of discriminator to reflect its new role. To avoid potential confusion, however, the word discriminator will be used to describe a generator's adversary in a GAN — regardless of classifying capabilities or the lack thereof.

been a focus on not letting the discriminator overpower than the generator.

2.5.4 Information Maximizing GANs

As mentioned in Section 2.5.2, while DCGANs are able to perform vector arithmetic on latent vectors and obtain meaningful features in the results, the representations of these features are entangled with random noise. Finding the vector representations of various features is a manual and unpredictable process. Information Maximizing GANs (InfoGAN), however, are able learn meaningful, disentangled representations in a completely unsupervised fashion using concepts from information theory (Chen et al., 2016).

The idea behind InfoGAN is to encapsulate different features in a separate vector $\mathbf{c}$, called the latent code, that is input to the generator in addition to the latent noise vector $\mathbf{z}$. The latent code $\mathbf{c}$ contains a preselected number of variables, where each variable can be considered a "placeholder" for a feature that the generator is tasked to learn. There are two types of variables that can be used in the latent code: categorical and continuous, which are sampled from categorical and uniform distributions (Section 2.3.1), respectively. If no modifications are made to the objective function in Equation 2.3, however, the generator can simply disregard $\mathbf{c}$, and effectively only use $\mathbf{z}$ to generate its output. Such a scenario would lead to the same entangled representations associated with standard GANs. To avoid this, Chen et al. propose adding a regularization term to the objective that maximizes the mutual information (Section 2.2.4) between the latent code $\mathbf{c}$ and the output of the generator. This can be done be maximizing the following equation

$$I(\mathbf{c}; G(\mathbf{z}, \mathbf{c})) = H(\mathbf{c}) - H(\mathbf{c}|G(\mathbf{z}, \mathbf{c})) \tag{2.4}$$

where I is the mutual information, G is the generator function and H is the entropy. Maximizing Equation 2.4 directly requires knowing $P(\mathbf{c}|\mathbf{x})$, the probability distribution for a latent code $\mathbf{c}$ given an image $\mathbf{x} \sim G(\mathbf{z}, \mathbf{c})$, which makes the optimization impractical. Instead, Chen et al. approximate this distribution with a neural network $Q(\mathbf{c}|\mathbf{x})$. This network usually reuses most of the layers of the discriminator, which results in only a small additional computational cost compared to allocating an entire stand-alone network for the task. In practice the added regularization term leads to faster convergence compared to standard GANs (Chen et al., 2016), so the added computational cost is negligible. Chen et al. further derive a lower bound of the mutual information, and propose the following GAN objective for its maximization

$$\min_{G,Q} \max_{D} V_{InfoGAN}(D, G, Q) = V(D, G) - \lambda L_I(G, Q) \tag{2.5}$$

where $V(D, G)$ refers to the original GAN objective from Equation 2.3, λ is a constant and $L_I(G, Q)$ is the regularization term.

As a practical example, if training an InfoGAN on a dataset containing images of the numbers 1–9, a latent code $\mathbf{c}$ consisting of one categorical and two continuous variables could be utilized. As there are 10 distinct numbers in the training data, using a categorical variable with 10 categories might correspond to digit type. The two continuous variables can be sampled from a uniform distribution in the interval $[-1, 1]$ and might correspond to features such as tilt and thickness of the digits[9]. After successful training, varying the value of the categorical variable usually results in changing which number the generator outputs. Thus, the generator has to a certain degree learned that the categorical variable corresponds to classes. Similarly, changing the value of the two continuous variables might correspond to changes in the tilt and thickness, respectively, of the generated numbers. An important thing to note is that the variables in $\mathbf{c}$ are not entangled with any random noise, allowing various features of the output to be predictably controlled.

2.5.5 Stabilizing Training of GANs

Since the inception of GANs in 2014, a multitude of architectures, objective functions, regularization and normalization techniques have been proposed to improve the training stability and performance of these networks. Some of the most influential ones have been the DCGAN architecture (Radford et al., 2015), the WGAN (Arjovsky et al., 2017) and WGAN-GP (Gulrajani et al., 2017) objectives, as well as batch normalization (Ioffe and Szegedy, 2015) and the recent spectral normalization (Miyato et al., 2018).

Recent work (Miyato et al., 2018) indicate that using a convolutional architecture, spectral normalization in the discriminator, batch normalization in the generator and the Adam optimizer can net training stability and solid performance across a wide range of architectures and hyperparameters. Some further experiments also suggest that using spectral normalization in combination with the WGAN-GP objective can improve over the WGAN-GP baseline (Miyato et al., 2018).

Compared to many other objective functions, the WGAN-GP can be very robust to hyperparameter changes (Fedus et al., 2017). One of the drawbacks of using this objective, however, is the added computational cost of the imbalanced training scheme where the discriminator and generator are trained according to a $n : 1$ ratio. A technique called the two time-scale update rule (TTUR) (Heusel et al., 2017) removes the need to balance the discriminator and generator by using different learning rates for the two networks: instead of updating the discriminator n times at each training step, the discriminator simply uses a higher learning rate compared to the generator. Training a WGAN-GP using TTUR has been shown to lead to better results compared to using the ratio-based training scheme (Heusel et al., 2017).

Some studies (Fedus et al., 2017; Kurach et al., 2019) suggest that the non-saturating objective from the original GAN paper (Goodfellow et al., 2014) can outperform many of the alternative objectives that have been proposed, especially when coupled with spectral normalization and/or the gradient penalty regularizer introduced by Gulrajani et al.. Miyato et al. show that large learning rates and β_1 parameter for the Adam optimizer can have a detrimental effect on GANs trained using the WGAN-GP objective. A better

[9]Which features the different variables of the latent code will correspond to after successful training is not known a priori. Categorical variables do often correspond to discontinuous features, e.g. different classes, while continuous variables correspond to more continuous features, e.g. rotation, width, lighting, etc.

theoretical understanding of GANs, and various methods for improving the training and performance of them, continues to be a highly active field of research. The various findings presented in this section should help in guiding the construction of a GAN that can perform well across a wide range of architectures and hyperparameters.

2.6 Tools

This section introduces some of the tooling and practical methodology that is involved with training neural networks. This will mainly pertain to the introduction of image datasets that are commonly used when training generative models, as well as a primer on hardware and software methodology in the context of machine learning.

2.6.1 Datasets

A dataset is a collection of data points, or examples in more colloquial terms, of any representation. The dataset might have labels associcated with the different data points as a way to describe relevant features or properties. A dataset containing images of cats and dogs might have labels associated with each image describing the type of the animal it contains. Similarly, a dataset for language processing might contain movie reviews labeled with their overall sentiment.

There exist a great deal of standardized datasets to use for training neural networks to classify and/or generate images. One that is often used in the developmental phases of a model is the MNIST dataset. This dataset contains 70 000 28x28 grayscale images of handwritten digits in the range 0–9 (LeCun et al., 2010). Due to its low complexity it is relatively easy to achieve good performance on the MNIST dataset, and if a model performs poorly on it, the model will likely fare worse on more complex datasets. In general, a model should achieve good scores on the MNIST dataset before training on other datasets. Another dataset, with a somewhat higher complexity than the MNIST dataset, is the Fashion MNIST dataset (Xiao et al., 2017). This dataset contains 70 000 28x28 greyscale images of 10 classes of fashion items: pants, sweaters, dresses and shoes to name a few.

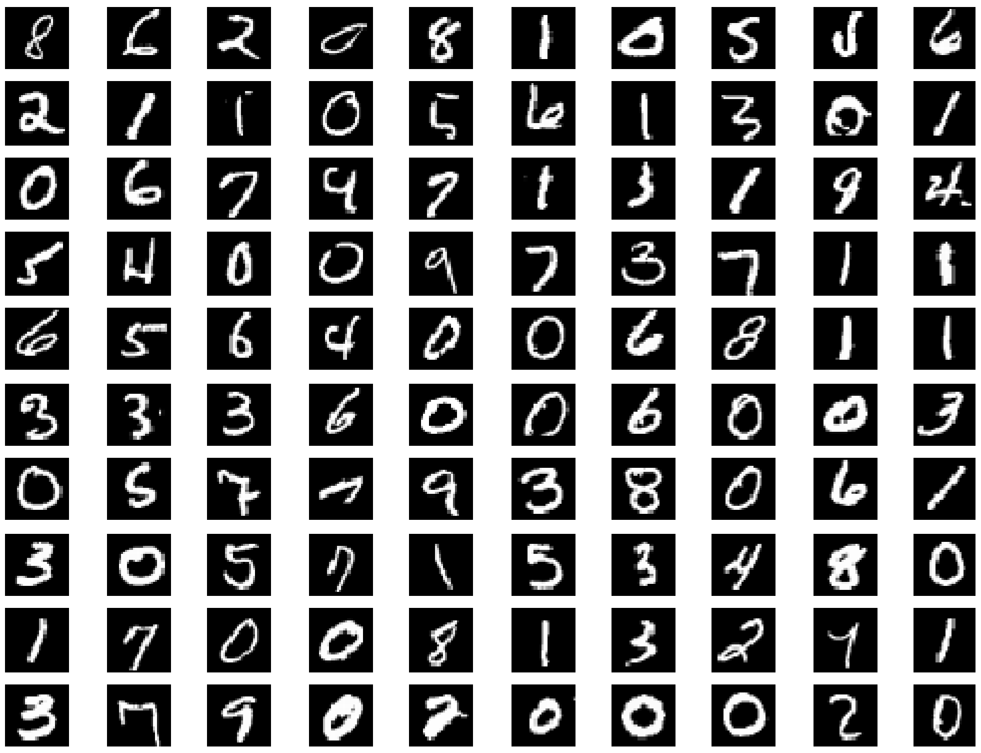
In terms of RGB images, a widely used datasets is the CIFAR-10 dataset. This dataset contains 60 000 32x32 images of ten mutually exclusive object classes, e.g. dogs, airplanes, cars, frogs, etc (Krizhevsky and Hinton, 2009). This is a significantly harder dataset to train on compared to the MNIST dataset, as there are now three color channels as opposed to one, and the images are more complex and diverse. Another popular RGB dataset is the CelebA dataset (Liu et al., 2015). This dataset contains more than 200 000 richly annotated 218x178 images of celebrity faces.

Random samples from these four datasets are shown in Figure 2.6.

2.6.2 Software Libraries

Most neural networks today harness the power of graphics processing units (GPUs) for efficient training (Goodfellow et al., 2016, p. 444). In contrast to the traditional, general-purpose central processing units (CPUs), GPUs are specialized hardware traditionally optimized for processing computer graphics, and they are particularly well-suited to efficiently perform large numbers of matrix and vector operations in parallel. As such there is a lot of overlap in the type of computation done in computer graphics and neural network algorithms. Several companies have developed software platforms that offer direct access to the GPU, the most widely used being NVIDIA's CUDA platform.

It is entirely possible to implement a simple neural network from scratch in an arbitrary programming language using just the language's standard library, but making such an implementation performant is a very non-trivial task. As the complexity of the model increases and optimizations are needed, one is better off using a machine learning software library. Three widely used libraries for working with neural networks are TensorFlow, PyTorch and Keras. These libraries offer the tools necessary to set up, train, evaluate, save and restore neural networks. They are optimized for both CPUs and GPUs, and they include a lot of common operations and functions needed in neural network algorithms. Their core functionality is implemented in highly-optimized low-level languages such as C++ and CUDA, but this functionality is wrapped in high-level languages, like Python or Javascript, to simplify the process of building and training the models. In short, these libraries let users set up and train neural networks without needing to worry about low-level optimization and implementation of commonly used operations, and they offer a lot of auxiliary functionality that can be useful, i.e. visualization tools, pre-made implementations of various algorithms, etc.


(a) The MNIST dataset

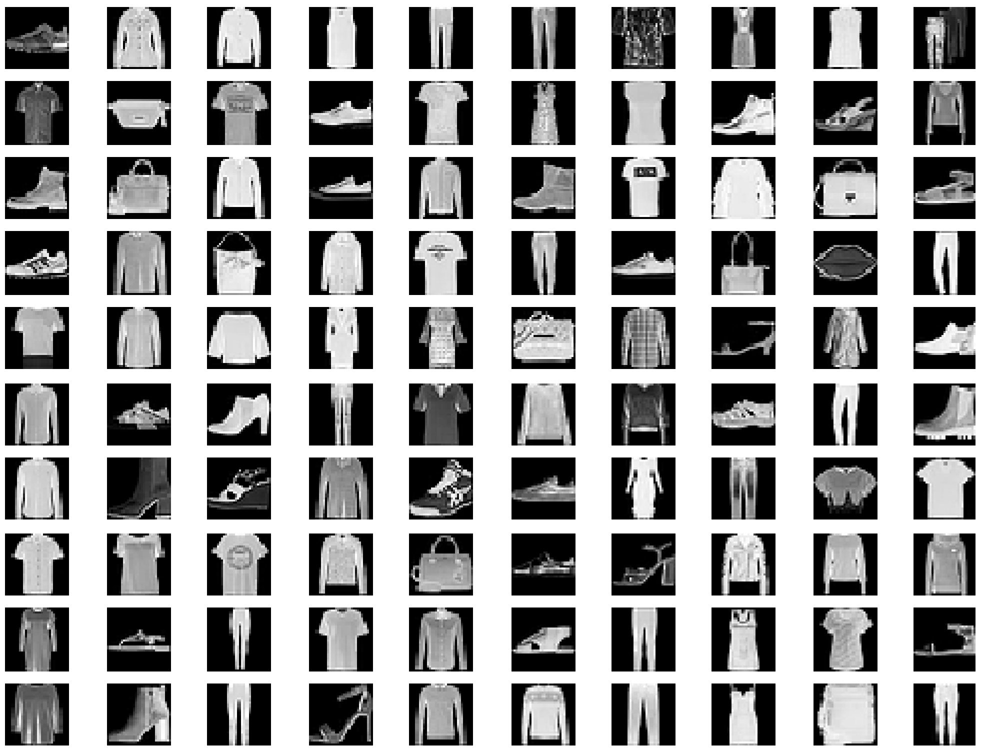
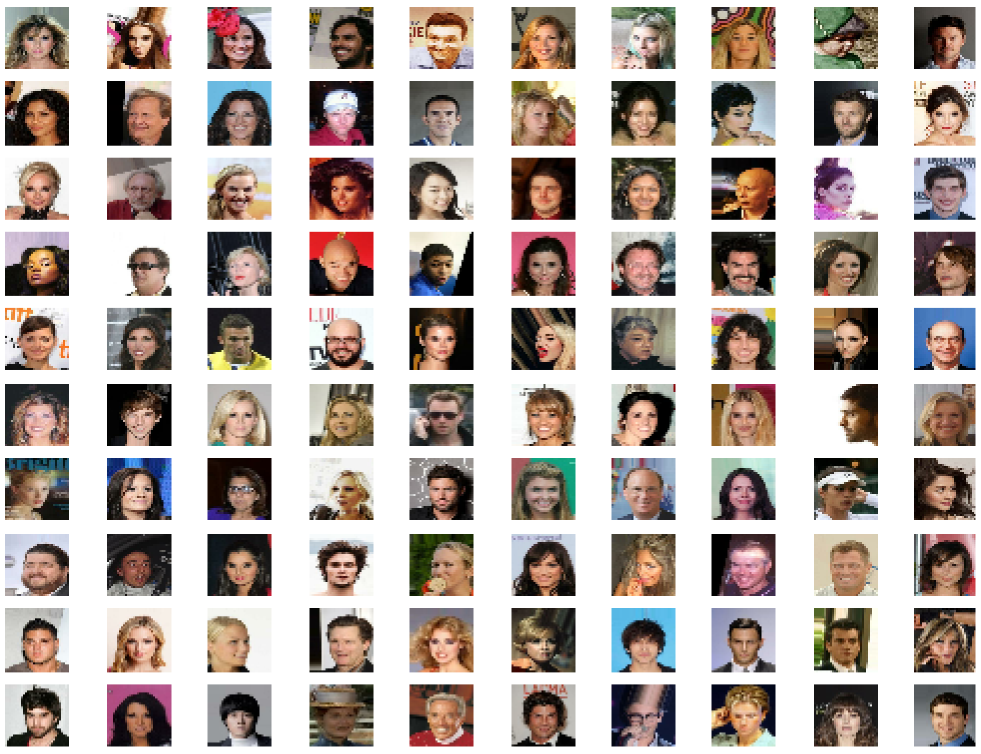

(b) The Fashion MNIST dataset

Figure 2.6: Random samples drawn from different datasets


(c) The CelebA dataset resized to 32x32

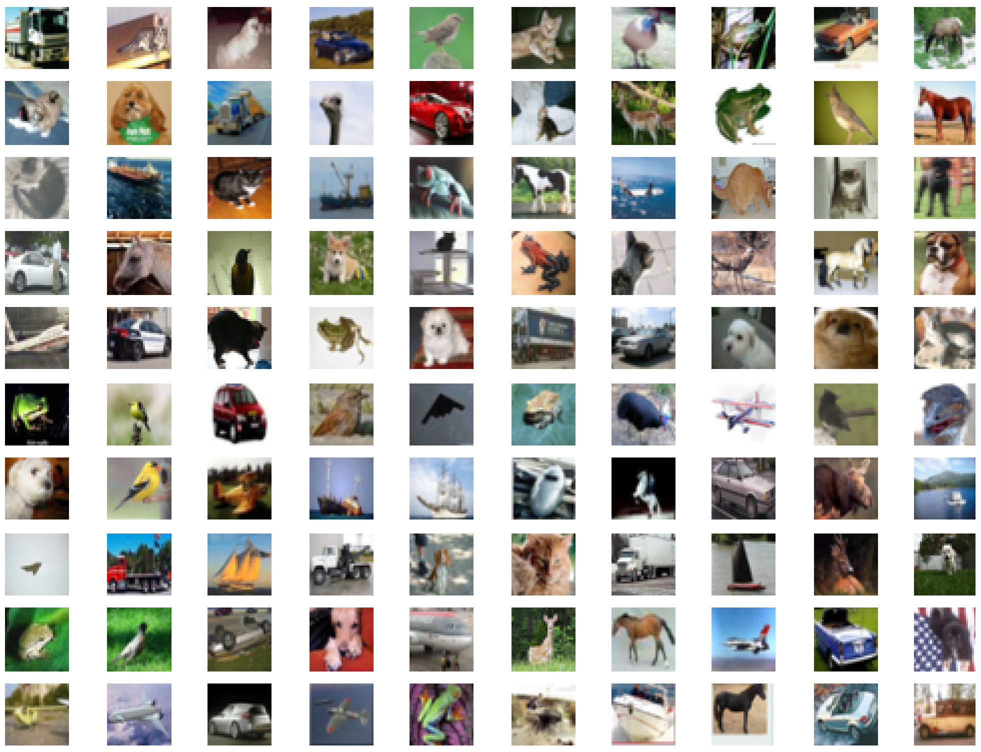

(d) The CIFAR-10 dataset

Figure 2.6: Random samples drawn from different datasets

3 Related Work

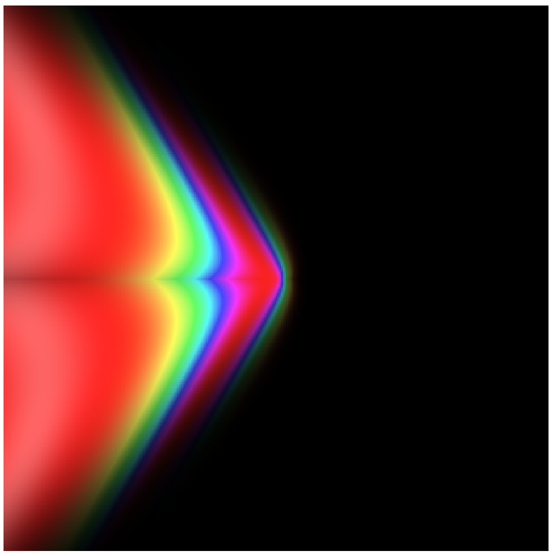
This chapter will introduce previous work that pertains to CPPNs in general, as well as the usage of CPPNs as generators in GANs (CPPN-GAN). There will be a focus on highlighting the motivations, findings and limitations of the various experiments conducted by the respective authors.

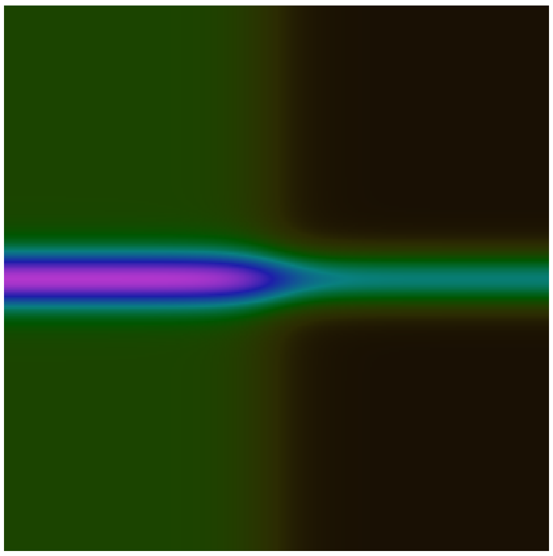
3.1 Early Applications of CPPNs

The earliest applications of CPPN-like[1] models appear in the context of interactive genetic art (Sims, 1991). Such applications revolve around a user being presented with a set of initial images generated by different models and then allowing the user to select which images, and thus implicitly models, to evolve. The images' visual characteristics are defined by the architectures and inner structure of the models generating them, and as such the user is able to dictate which patterns he or she would like to see more of in subsequent generations.

Picbreeder (Secretan et al., 2008) and Neurogram (Ha, 2015b) are two of the more influential projects that utilize CPPNs in such a manner. They utilize an evolutionary algorithm called NeuroEvolution of Augmenting Topologies (NEAT) (Stanley and Miikkulainen, 2002) to evolve user-selected models, with the models represented by CPPNs. An example of using Neurogram to evolve CPPNs is shown in Figure 3.1. The networks in Figure 3.1a and 3.1b are chosen to be evolved, and the result of this evolution is a new generation of CPPNs as shown in Figure 3.1c. In addition to Neurogram, Ha has conducted several other experiments on CPPNs in the context of interactive, generative and genetic art (Ha, 2015a; Ha, 2016e; Ha, 2016d). While some of these experiments are revolve around evolving the networks using NEAT, others pertain to image generation by randomly initialized CPPNs. Overall, the original CPPN paper (Stanley, 2007), Picbreeder and various experiments by Ha highlight some of the key properties of CPPNs, including the effect various activation functions and network structures can have on the output images. Some of these experiments (Ha, 2016e) further introduce a latent noise vector $\mathbf{z}$ as an input to the CPPN as a way to generate different images using a fixed network, and the application of a scaling factor s on all inputs to the CPPN.

[1]The term *CPPN-like* is used to refer to a superset of CPPNs, removing the constraint that they need to be implemented as a neural network. The characteristics are, however, still the same in that they represent a function $f(x, y)$ that takes a coordinate as input and outputs the value at the given coordinate — a pixel or color intensity in the case of images. The actual architecture of such models can be a neural network, LISP expressions, mathematical functions, etc.


(a) The first network that is to be evolved


(b) The second network that is to be evolved

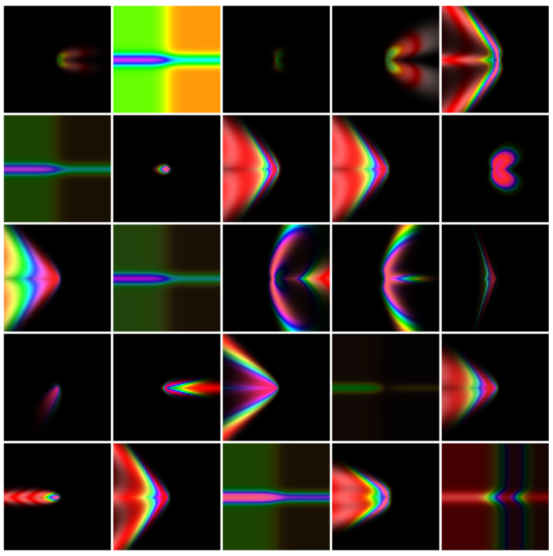

(c) The subsequent generation of networks after evolution

Figure 3.1: Evolving CPPNs using Neurogram (Ha, 2015b). The application is available at `http://otoro.net/neurogram/`. (The images are licensed under CC BY-SA 4.0)

In the aforementioned interactive systems the training of the CPPNs is being guided by the interacting user. It is, however, not possible to query the model for an image of a car. The process for generating a specific image instead revolves around actively seeking out patterns that might lead to results resembling such an object, and then evolving them using an evolutionary algorithm. This can, however, be quite a tedious process with no guarantee of success as the space of possible results is extremely large.

There have been some examples of training CPPN-like models on images (Karpathy, 2014), but such models have been restricted to generating only a single image. Extensions could be made to such models to work on two or more images, but at that point it would probably be more beneficial to utilize pre-existing and proven generative models that can be trained on large datasets. As such, it might be reasonable to experiment with CPPNs in the setting of existing generative models, such as GANs, before looking into ad-hoc solutions and architectures.

3.2 CPPN-GAN Architectures by Ha

This section contains a presentation of the different experiments performed by Ha in terms of CPPN-GAN architectures. Briefly, this includes a CPPN-GAN, CPPN-GAN with a variational encoder, and CPPN-GAN with a variational encoder and residual layers.

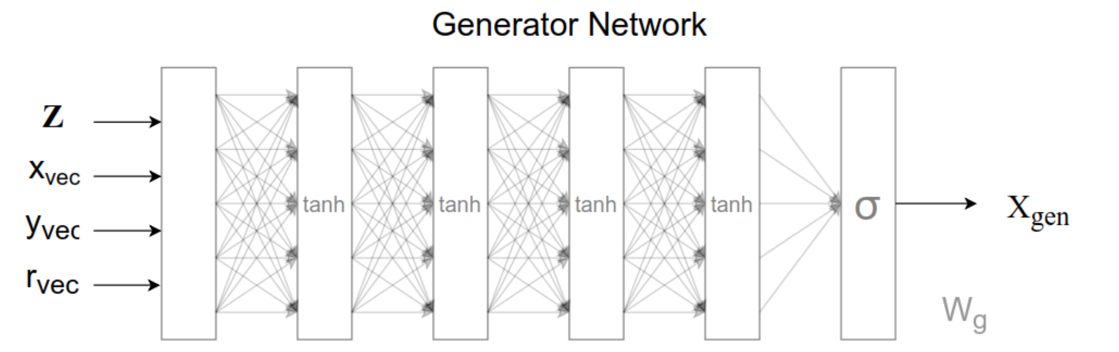
3.2.1 CPPN-GAN

A new application of CPPNs was introduced by Ha in 2016a that made it possible to train such networks on large, multi-class datasets. Ha experimented with and shared the results of training a CPPN by utilizing the adversarial framework offered by GANs. The overaching goals of his initial experiments with this architecture can be summarized as follows:

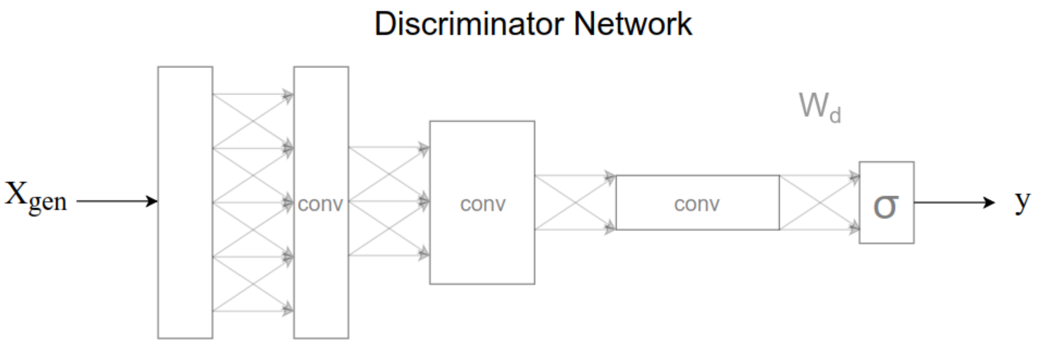
- Generate low-resolution images, e.g. 28x28, that are indistinguishable from the training data.
- Learn to generate a diverse set of samples, e.g. the ten different classes in MNIST dataset.
- Use the same generator to produce high-resolution images, e.g. 1200x1200, that exhibit interesting visual properties.

The architecture used in the initial experiments is shown in Figure 3.2. The inputs to the generator is a 32-dimensional latent vector $\mathbf{Z}$ of random Gaussian samples, a vector of x pixel-coordinates $\mathbf{x_{vec}}$, a vector of y pixel-coordinates $\mathbf{y_{vec}}$, and a vector of the distances between each (x, y) and the center of the image $\mathbf{r_{vec}}$. The output is a generated image $\mathbf{X}_{gen}$. Note that all of the pixel values for a given image are generated in parallel instead of one by one, which should speed up the training of the CPPN.

The CPPN consists of four hidden, fully-connected layers with hyperbolic tangent activation functions and 128 units in each layer. The final layer uses a sigmoid activation function, and outputs a single value in the interval $[0, 1]$. The discriminator is based partly on the DCGAN (Section 2.5.2) architecture and consists of three convolutional



(a) The generator is a CPPN using an MLP architecture



(b) The discriminator is a CNN

Figure 3.2: A CPPN-GAN architecture. (Reproduced from Ha, 2016a, reprinted with permission)

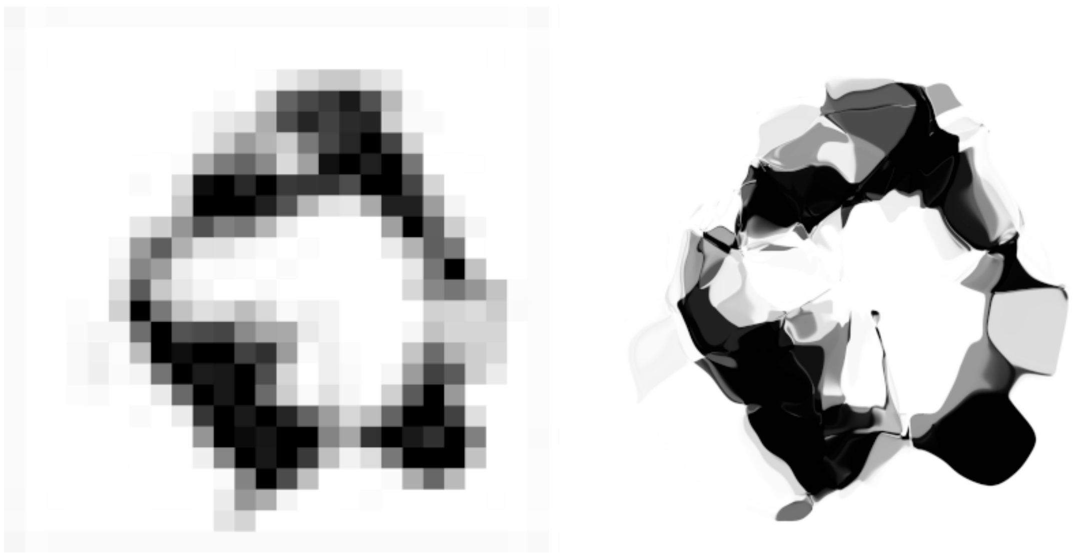
layers and one fully-connected layer. The output of the discriminator is a real number between 0 to 1, and the non-saturating GAN objective is used.

The CPPN-GAN was trained on an augmented version of the MNIST dataset. A collection of techniques were employed with an objective of slowing down the training of the discriminator, namely:

- Only updating the weights of the discriminator if the generator's loss is smaller than an upper boundary and the discriminator's loss is larger than a lower boundary.

- Every epoch, train the generator n times while only training the discriminator (potentially) once.

After training, the generator was able to generate the images shown in Figure 3.3. Ha notes that that the generator was only capable of generating a small subset of the ten classes found in the MNIST dataset.



(a) 26x26 generated image (b) 2160x2160 generated image

Figure 3.3: Images generated by a CPPN-GAN. The same latent vector was used for both images. (Adapted from Ha, 2016a, reprinted with permission)

3.2.2 CPPN-GAN with a Variational Encoder

One of the issues with the architecture presented in Section 3.2.1 is that the generator has no incentive to approximate the diversity found in the dataset. The generator will fool the discriminator as long as it is able to generate something that resembles anything from the training data: there is no mechanism in place to stop the generator from specializing in only generating plausible, in the case of the MNIST dataset, 1s and 2s, instead of approximating the true distribution underlying the diverse training data. There does, however, exist many different techniques to force diversity into the generator.

Ha experiments with adding the encoder from a variational autoencoder (VAE) (Kingma and Welling, 2013) to the CPPN-GAN architecture to improve this issue. In short, a VAE consists of an encoder and a decoder, both implemented as neural networks. The encoder's task is to convert an input, e.g. a 32x32 image, into an n-dimensional vector, e.g. $n = 100$. The idea is that this low-dimensional vector will contain a compact representation of the high-dimensional original input, learning to preserve relevant information while at the same time discarding irrelevant information. This vector is input into a decoder that tries to reconstruct the original input to the encoder based on this compressed version.

The resulting architecture of adding the variational encoder to the CPPN-GAN will be referred to as the CPPN-GAN-VAE architecture. In doing so, the variational encoder and the CPPN can effectively be considered a VAE, with the CPPN functioning as the decoder. The generator and discriminator architecture from Figure 3.2 remain the same, but the latent vector $\mathbf{z}$ is changed from being a randomly sampled vector of Gaussian noise to an encoded version of an input image as shown in Figure 3.4. At this point it

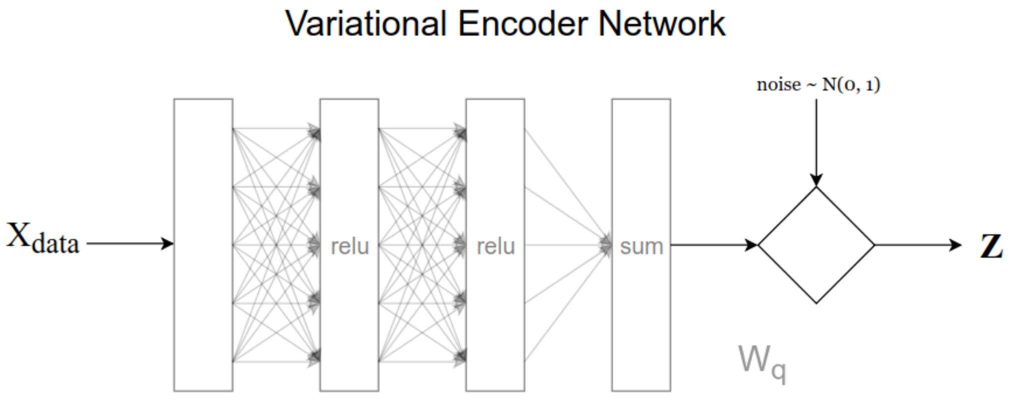


Figure 3.4: The architecture of the variational encoder used in combination with a CPPN-GAN. (Reproduced from Ha, 2016a, reprinted with permission)

would have been possible to simply use the CPPN as the decoder in a standard VAE, thus eliminating the need for a GAN architecture completely. This would likely alleviate some of the issues related to training stability. VAEs by themselves, however, tend to produce more blurry samples than GANs (Zhao et al., 2017). This also seems to be the case when using CPPNs as the decoder in a standard VAE, resulting in blurry and less novel images than the ones produced by CPPN-GANs (Ha, 2016a).

The training of the CPPN-GAN-VAE is similar to the training procedure presented in Section 3.2.1, but there are now three, instead of two, neural networks that are being trained simultaneously: the discriminator, the generator and the encoder. As the latent vector $\mathbf{z}$ is no longer sampled from a prior distribution, but is instead the encoded version of a real image from the dataset, the CPPN is additionally adjusting its weights based not only on the standard GAN objective, but also based on the pixel-by-pixel reconstruction loss between the real image and the generated image.

The encoder forces the generator to be more diverse, and it has the additional benefit of giving more control over the generated images: desired features in the output image can be controlled by using an image that contains these features as an input to the encoder, and subsequently feeding the encoding to the CPPN. Figure 3.5b shows a generated sample from the CPPN-GAN-VAE model.

3.2.3 Residual CPPN-GAN with a Variational Encoder

While the CPPN in the architecture in Section 3.2.2 is capable of more closely approximating the diversity of the training data, this comes at the expense of less novel high-resolution results (Ha, 2016b). Ha proposes the following changes to the CPPN-GAN-VAE architecture to encourage the generation of more novel and artistic images (Ha, 2016b):

1. Convert the discriminator into a classifier in an effort to eliminate the pixel-by-pixel

reconstruction loss associated with the variational encoder.

2. Increase the depth of the CPPN.

3. Increase the values of the initial weights of the CPPN.

Instead of the discriminator only classifying an image as real or fake, it is now also tasked with predicting the class of a given image. Thus, the discriminator is modified to output a softmax of eleven classes – one class for each unique digit in the MNIST dataset, and an additional class for images it believes come from the generator. This is an additional measure to force the generator to produce more diverse samples. The pixel-by-pixel reconstruction loss is replaced with a loss based on how well the generator is able to generate samples that the discriminator classifies as the class label of the (encoded) input image.

Moreover, Ha notes that, empirically, deeper architectures generate more interesting results. Traditionally, there have been many issues associated with the training of deeper networks, but the residual learning framework (Section 2.4.2) has improved on this situation. Ha uses 24 residual blocks to construct a deep CPPN architecture, the ResNet-CPPN-GAN-VAE. Each block contains five layers with six units in each, and the activation functions for the first four and the final layer is ReLU and tanh, respectively. Ha further notes that larger weights in the CPPN produce more interesting-looking images, and the initial weights in all of the intermediate layers of the residual blocks are thus set to large values. The weights of the final layer in each block, however, is initialized to values close to zero.

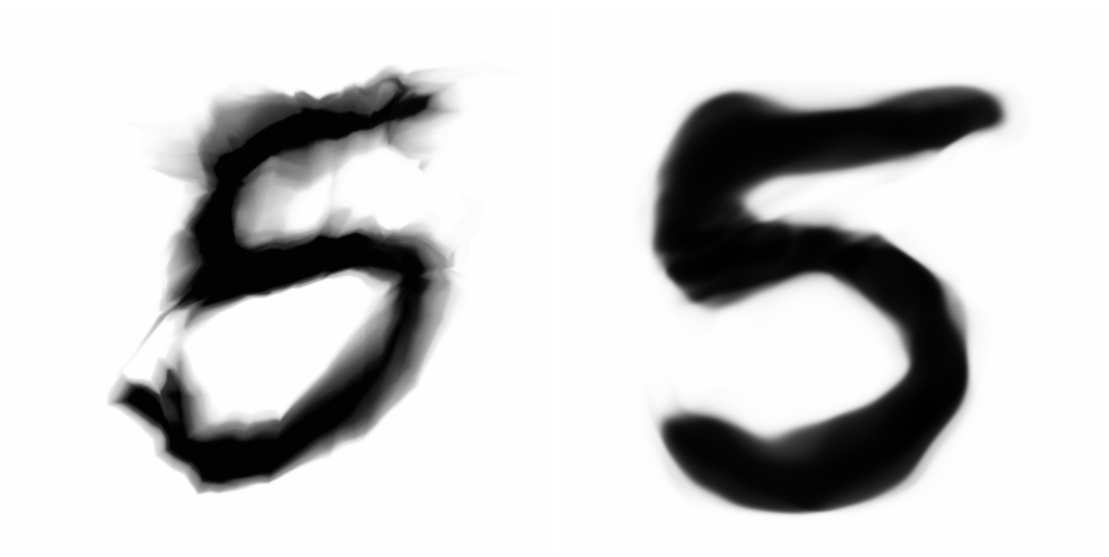
A comparison of the results from the ResNet-CPPN-GAN-VAE and CPPN-GAN-VAE is shown in Figure 3.5. Note the different visual properties of the two generated samples.

3.3 CPPN-GAN Architectures by Metz and Gulrajani

Metz and Gulrajani extend the work by Ha in Section 3.2 by employing new training techniques, and introducing new applications of CPPN-GAN architectures with a focus on exploring how such architectures can be used as creative tools (Metz and Gulrajani, 2017).

While the CPPN-GANs introduced in Section 3.2 were trained to approximate a single probability distribution, these architectures can also be trained on multiple different distributions (Metz and Gulrajani, 2017). A CPPN allows for the generation of images at both arbitrary scales[2] and resolutions. As such, a CPPN-GAN can be trained to approximate natural scenes at a low resolution, while circular patterns at a high resolution, or similarly natural scenes at a low scale and patterns on at high scale. To achieve this Metz and Gulrajani use multiple discriminators trained on different datasets, and train the CPPN to approximate multiple distributions at different scales or resolutions. An example of a CPPN trained to approximate different distributions at different scales can

[2]Scale refers to the scale of the input coordinates. For a scaling factor s, the input coordinates are in the interval $x, y \in [-s, s]$.



(a) 1300x1300 generated image from ResNet-CPPN-GAN-VAE (b) 3240x3240 generated image from the CPPN-GAN-VAE

Figure 3.5: Comparison between different CPPN-GAN models. (Adapted from Ha, 2016b, reprinted with permission)

be seen in Figure 3.6, where one discriminator is trained on the CIFAR-10 dataset and another is trained on a dataset of randomly generated circles. The CPPN is trained to approximate the first discriminator using a scaling factor $s = 1$, and the second using $s = 0.1$.

The architecture of this CPPN-GAN is very similar to the one presented in Section 3.2.1. Instead of the regular GAN objective, however, the WGAN-GP objective is used to increase the training stability, and layer normalization and batch renormalization is used for the discriminator and generator, respectively. Additionally, while the architectures presented in Section 3.2 have mostly used hyperbolic tangent and ReLU activation functions, a more diverse set of activation functions is explored, including absolute value and trigonometric functions (Metz and Gulrajani, 2017). These different functions are used to encourage different types of patterns in the high-resolution output.

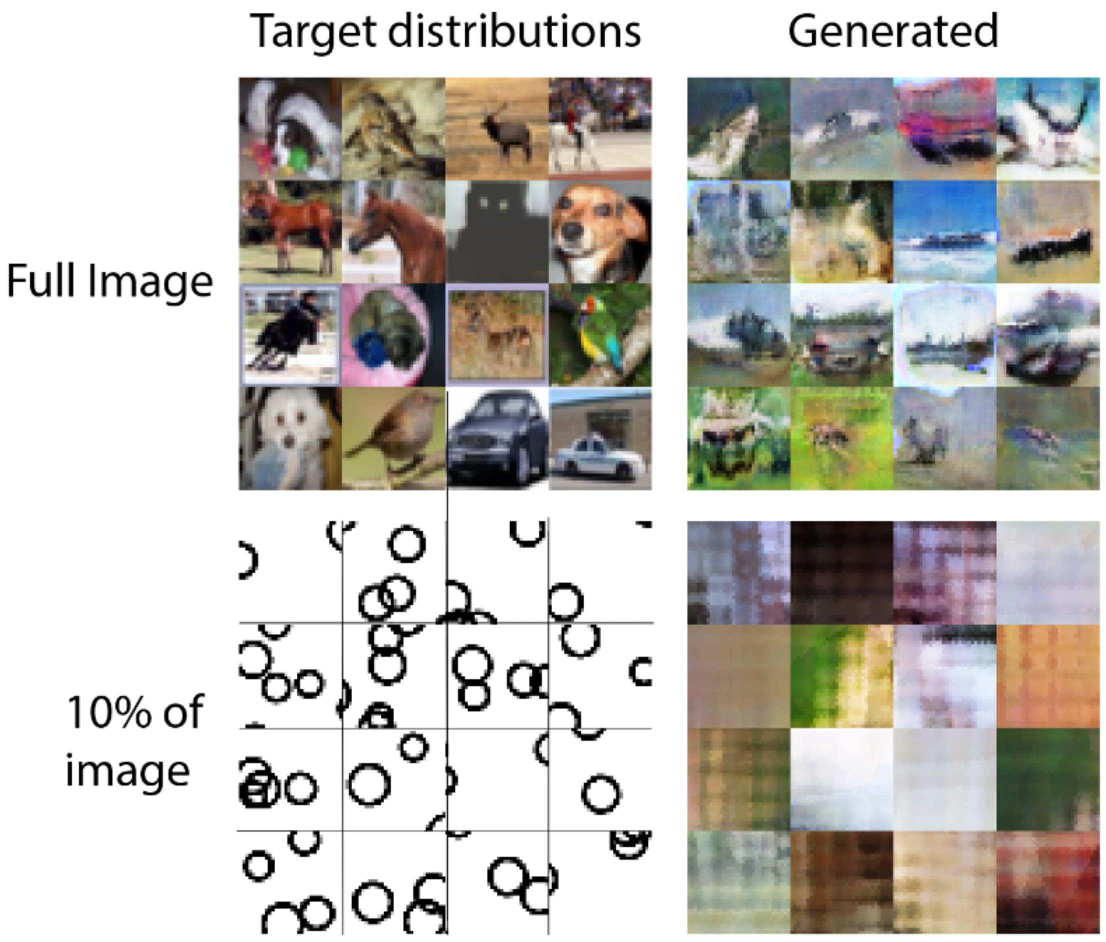


Figure 3.6: A CPPN-GAN trained to approximate different distributions at different scales. (Reproduced from Metz and Gulrajani, 2017, reprinted with permission)

4 Architecture and Methodology

This chapter will introduce the software used to implement the architecture, the methodology used during the implementation phase and finally an overview of the models the architecture supports. The term architecture will in this chapter be used to refer to the overall implemented system in general. A model, on the other hand, refers to a specific application of the architecture to construct a specific GAN model.

4.1 Software

The architecture has been developed using Python 3.6.6 and TensorFlow 1.13.1. TensorFlow was chosen as the neural network software library as it offers a flexible API and because there exist a great deal of reference implementations of various neural networks architectures using this library. TensorFlow 1.13.1 was the latest stable release when the main architecture was developed, and this library and other dependencies are all locked to specific versions to allow for easy cross-platform usage. A new version of TensorFlow, version 2.0, is currently in beta, and it approaches modeling in a different way than earlier versions and is not backwards-compatible with TensorFlow 1.x. In any case, using 1.13.1 and making heavy use of modularization should make it easier to port the architecture to the new TensorFlow 2.0, should that be of interest.

Parts of the architecture, e.g. convolutional layers and normalization schemes, relies on pre-existing implementations that come bundled with TensorFlow, while others, e.g. transposed convolutional and fully-connected layers, are implemented from scratch using TensorFlow's low-level Python API.

4.2 Methodology

This section will contain the methodology that has been used during the development of the architecture.

In general, the architecture was developed by implementing the simple, foundational and shared functionality first and then extending it when needed. This development process has improved the modular nature of the code and has made the architecture flexible in use. The architecture makes it easy to run models with different configurations with minimal manual setup. Almost every relevant aspect of the architecture can be controlled using command-line arguments: discriminator and generator architectures, activation and loss functions, learning rates and datasets to mention a few. It is also possible to use a normal generator instead of the CPPN, which can be useful for comparing the performance of traditional GANs and CPPN-GANs, and to establish a baseline in terms of what conventional GANs can achieve in terms of visual quality.

4.2.1 Flexibility

The architecture has been developed with a focus on making most relevant aspects highly configurable. This is done by allowing the user to supply command-line arguments for every configurable argument. Sensible default parameters are set to avoid needlessly defining every aspect of each model, but it offers great flexibility should the need arise.

4.2.2 Reproducibility

Reproducibility in the field of artificial intelligence is a widespread issue, especially in terms of being able to reproduce the results of others (Gundersen and Kjensmo, 2018). This can often be attributed to lack of documentation and/or no reference implementation supplied by the original authors. It can also be a result of lack of care in terms of the randomness associated with neural networks: the weights and biases of neural networks are randomly initialized, and especially for volatile models such as GANs the initial values of the parameters can have a large effect on the model's convergence.

Five major steps have been taken to improve the reproducibility of the results obtained using the architecture:

- **Saving the initial configuration**
  As the majority of the architecture can be controlled using command-line arguments, all of the relevant configuration is saved to file in a human-readable format when a new model is created. This makes it very easy to retrospectively analyze and compare the architecture of any given model. Moreover, this file is used to programmatically set up the model when a user wants to load a given model for inference or to resume training.

- **Setting random seeds**
  When generating random numbers on a computer, the random-number generator can be initialized with a user-specified seed. For a given seed a random-number generator will produce the same sequence of random numbers. As such, using fixed seeds for the relevant libraries, i.e. NumPy, TensorFlow and Python's standard libraries, is helpful to eliminate randomness. If two identical models are set up using the same seed, the models should have the same initial parameters and the same progress and results during training.

- **Frequent saving of models**
  During training, the parameters, e.g. weights, biases and other relevant values of a given model, are frequently being saved to disk in the form of a checkpoint. This allows for restoration of a previously trained model at any of the saved checkpoints.

- **Architecture versioning**
  Each time the architecture is changed, the version number is increased to reflect

potential compatibility issues with earlier architectures. This is achived by using Git[1]. for version control, and it makes it easy to roll back changes and train/do inference on models created using an older architecture.

- **Dataset versioning**
  TensorFlow Datasets is a library of public datasets which provides streamlined access to a large collection of various datasets. The library abstracts away many of the operations usually associated with using a new dataset, e.g. manually downloading, extracting, preprocessing, etc., and presents the data in a format TensorFlow can work with out-of-the-box. All the datasets in the library are versioned, such that any changes in a given dataset will be reflected by a change in the version number.

4.2.3 Visualization

A great deal of effort has been aimed towards developing a wide variety of visualization tools that can be used both during and after the training of a model. The architecture offers extensive functionality for performing inference on any given model, and visualizing the results in meaningful ways. This includes creating movies from images generated by using interpolated latent vectors as input, creating grids of random samples, or grids of images where various variables of the latent code are varied, while others fixed, as a way to more easily assess their effect on the output. For the models that utilize a CPPN, the architecture also supports the generation of the same image at different resolutions and stacking them side-by-side for easy comparison of potential different visual features at different resolutions.

Most of the aforementioned visualization methods are also made compatible with TensorBoard — TensorFlow's toolkit for visualization. TensorBoard is a dashboard that can be used to monitor different aspects of a model both while it is training and after training has ceased. All of the statistics available in the dashboard is saved to disk, which makes it is possible to closely examine and compare multiple previously trained models at different steps of the training process. During training, the architecture has been configured to frequently upload various statistics and image samples to the dashboard: this includes one grid of samples for each variable in the latent code, a grid of random samples, plots of the values of the relevant loss functions over time, and, in the case of CPPNs, a random sample generated at both low- and high resolution. Moreover, the distributions and histograms of the biases, weights, pre-activation values and outputs of each layer of the generator in any given model is also frequently updated in the dashboard. All of this data allows for a holistic understanding of a given model's performance. It has proven very useful in understanding why any given model is converging or failing to converge, and perhaps stopping the training of models that seem to not converge. Finally, the computational graph of any given model is available in the dashboard, which can be useful to delve into and analyze in case there is any uncertainty surrounding the exact underlying architecture of the model.

[1]Git is a version-control system for tracking changes in and managing different versions of a codebase.

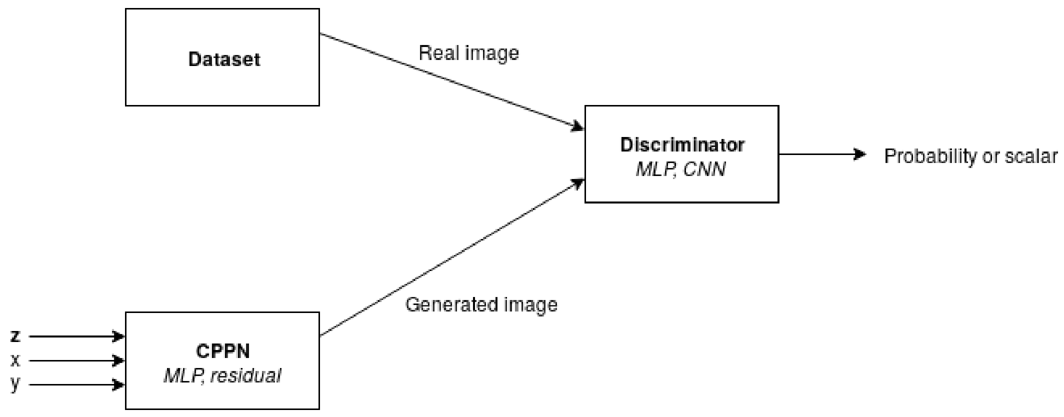


Figure 4.1: A CPPN-GAN model

4.3 Architecture Configurations

In general, the architecture allows for constructing GANs that use either an MLP, a CNN or a CPPN as the generator. The CPPN can be configured either as an MLP or a residual neural network. The discriminator can be configured as either an MLP or a CNN with a customizable number of layers and layer sizes. The auxiliary net is set up to reuse most of the layers of the discriminator, but an arbitrary number of fully-connected layers can be stacked on top of these when configuring the model. The architecture furthermore supports training GANs with both the non-saturating, non-saturating with gradient penalty, WGAN and WGAN-GP objectives. Four common normalization schemes can be applied to the layers of the networks in a model, including spectral and layer normalization, as well as batch normalization and renormalization.

The auxiliary net is only used if one of the objectives is to maximize the mutual information between the latent code **c** and the generated images. If a model does not use the auxiliary net, there will only be two neural networks as shown in Figure 4.1. The output of the discriminator is a probability or scalar based the GAN objective used. The discriminator in a WGAN or WGAN-GP model outputs a scalar which approximates the Earth-Mover (EM) distance (Section 2.3.2) between the training data and the implicit generator distribution, while networks using the non-saturating objective output the probability of a given image coming from the generator or the training data. If the auxiliary net is included there will be three neural networks in the model as shown in Figure 4.2. There is then an additional input **c**, the latent code, for the CPPN, and an auxiliary net that is concerned with reconstructing the latent code based on an input image.

When the architecture is configured to use a CPPN and an auxiliary net, the general model will be referred to as an Information Maximizing and Pattern-Producing GAN (IMPGAN).

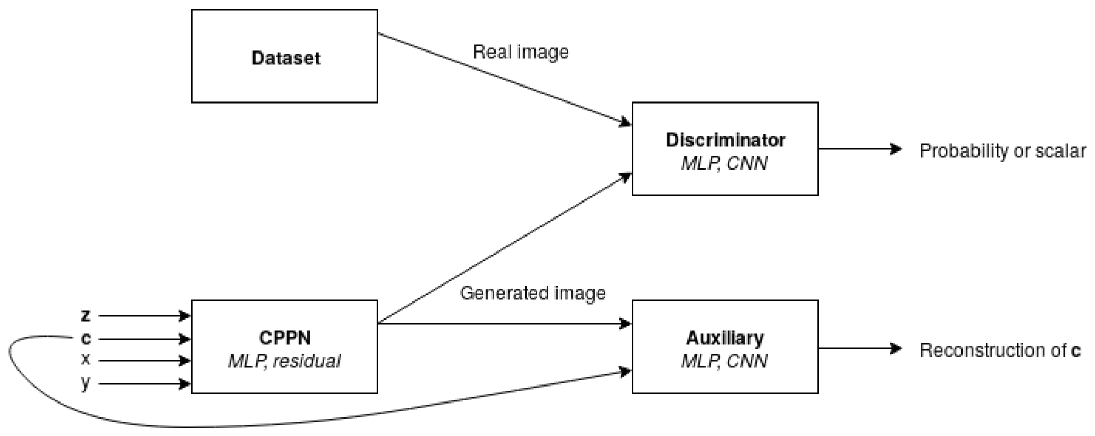


Figure 4.2: A CPPN-GAN model with an auxiliary net

5 Experiments and Results

In this chapter the majority of the experiments that have been carried out will be presented. Section 5.1 introduces the overarching plan for all of the experiments, followed by a short summary of the principal purposes and aims of the different experiments. Section 5.2 primarily revolves around describing the shared configurational aspects of the experiments, and the reasoning behind the default configuration. Finally, Section 5.3 describes the specific experiments and presents their respective results, as well as a short discussion and evaluation of each of them. Additionally, deviations from the more general experimental setup as well as specific extensions to the default configuration of the experiments will be described in this section if necessary. A more holistic and summarizing evaluation of the results is deferred to Chapter 6.

5.1 Experimental Plan

What follows is a short presentation of the key experiments that will be conducted, as well as the overarching aims and goals of them. The experiments will start out with models of low complexity, and the results of these simpler models will be used to guide later, more complex experiments in both a technical and theoretical manner. As such, a bottom-up approach will be taken during the experimental phase such that a solid foundation is established at each level. The key learnings from each level should prove beneficial as complexity increases in the later stages.

There exist an endless number of possible configurations in terms of learning rates, network depths, activation functions, etc., for most of the experiments, and given the time-consuming task of training and/or evaluating any given model, it is naturally infeasible to perform experiments on all possible combinations of architectures and parameters. The experiments will instead focus on the subset of configurations that are most likely to have a large effect on the training stability and generated images, guided by current literature, related work and potential key takeaways from previous experiments.

The first experiments will revolve around different architetures and parameters for CPPNs in order to assess how various configurations affect the generated images. The networks will merely be set up and initialized, and then sampled from and evaluated. In other words, there is no GAN or training involved in these experiments. The aim is to establish a practical understanding of the key properties of CPPNs, and utilize these learnings in the more complex experiments involving both CPPNs and GANs. The experiments are further motivated by the fact that it will be difficult to know in which ways a CPPN should be tweaked in order to improve the optimization process in a GAN, if one is not already familiar with how CPPNs in their basic form are affected by certain architectures and parameters.

Following this are experiments pertaining to what can be referred to as “conventional” GANs — that is, GANs with an MLP or CNN as the generator. The main goal is to construct an architecture that is able to achieve solid performance and training stability across a multitude of datasets. The architecture underlying the successful results will serve as the technical foundation for ensuing experiments. Moreover, as the CPPN in an IMPGAN is not expected to outperform a conventional GAN in terms of the visual quality of low-resolution images, the results can serve as an approximate upper bound of the visual quality attainable by the IMPGANs.

Finally the main experiments will be presented. These are concerned with training Information Maximizing and Pattern-Producing GANs (IMPGAN) across a wide range of configurations. The main purpose of these experiments is to generate images that approximates the original dataset at a low resolution, e.g. 32x32, but exhibits different visual properties and/or a relatively high degree of realism at higher resolutions, e.g. 1200x1200. The experiments will mainly be carried out by training IMPGANs on different datasets, and experimenting with different architectures, activation functions and various parameters in order to gauge how they affect the visual properties of the high-resolution images. Most of the CPPNs in the IMPGANs will be based on configurations that are expected to perform relatively well in an an optimization setting.

5.2 Experimental Setup

This section contains the experimental setup shared by most of the experiments. Due to the sheer number of experiments conducted, many of the configurations that are highly specific to a single experiment will instead be detailed together with the respective experiment in Section 5.3 to increase readability. Most experiments do, however, share a great deal of setup in terms of datasets, hardware, architectures, parameters and other configurational aspects, and this shared setup will be presented here. Any potential deviations from this setup will be mentioned in Section 5.3 if necessary.

5.2.1 Datasets

Four different datasets have been used in the experiments which involve the training of a neural network. These datasets are the MNIST, Fashion MNIST, CelebA and CIFAR-10 datasets. The datasets cover a diverse set of examples across many domains and levels of complexity. The CelebA dataset is resized to either 32x32 or 64x64 to decrease training time and computational requirements, while the other datasets are kept at their original resolutions. In certain cases, e.g. if convergence is hard to reach when using the full dataset, a given dataset might be filtered to only include images of a single class.

The raw representations of each image in the datasets are normalized to lie in the interval $[-1, 1]$ as opposed to alternatives such as $[0, 1]$ or $[0, 255]$.

5.2.2 Hardware

Most experiments have been run on an NVIDIA Tesla P100 GPU with 16GB of memory. Two of these GPUs have been available, allowing multiple computationally expensive experiments to be run in parallel. Any single experiment is, however, only run on a single GPU – that is, the computations have not been distributed across the two GPUs. Some experiments of lower complexity, often involving smaller models and architectures, and possibly no training, have been conducted on an NVIDIA GeForce GTX 750 Ti with 2GB memory. Additionally, when generating large images using a CPPN architecture, the processing is performed on the CPU as to not be limited by the amount of GPU memory.

The hardware is being accessed remotely through secure shell (SSH) connections, and all experiments are set up to run in persistent sessions using tmux, a terminal multiplexer. If the connection to the server for some reason is terminated, this allows the experiments to continue running on the hardware, and a connection to the persistent session can be re-established at any given time.

5.2.3 Default Architectures and Parameters

All of the configurational aspects presented here are the ones used for all the experiments unless otherwise mentioned in Section 5.3.

Hyperparameters

A common set of hyperparameters are used for the majority of the experiments, and their values are shown in Table 5.1. The values of these parameters were initially derived mainly from related work and current literature, but have subsequently been tweaked to improve the stability of the training process and the level of convergence reached for the experiments involving GANs and IMPGANs.

Initial experiments showed that setting a small scaling factor $s = 1$ resulted in similar results compared to $s = 10$ when training with small initial weights. When large initial weights were used, however, the network with the lower scaling factor seemed to converge slower and produce worse results. These results can be found in Appendix 1. Based on this, the scaling factor has been set to $s = 10$. A much larger scaling factor would likely not be beneficial in an optimization context, as the size differences between the latent inputs $\mathbf{z}$ and $\mathbf{c}$, and the coordinate inputs would be very large.

While the TensorFlow implementation of the Adam optimizer defaults to using $\beta_1 = 0.9$, recent work (Miyato et al., 2018) have shown that when using the Wasserstein distance with gradient penalty (WGAN-GP) as the main objective in a GAN, a smaller value of the β_1 parameter can result in better results compared to larger ones. Based on this, a lower $\beta_1 = 0.5$ is used in the default configuration.

The learning rates for the discriminator and generator are taken from the paper that introduced the two time-scale update rule (TTUR) (Heusel et al., 2017), where they had success with using TTUR when training a GAN using the WGAN-GP objective.

The learning rate for the auxiliary network is set to be the same as the generator in an effort to not let the maximization of mutual information dominate the optimization of the generator — that is, the generator should first and foremost aim to approximate the distribution of the training data.

While some recent work (Heusel et al., 2017; Miyato et al., 2018) utilizes a batch size of 64 when training various GANs, others (Brock et al., 2018) suggest that increased batch sizes result in generated images of higher visual quality. The extent to which larger batch sizes improve the training and performance of GANs is, however, still an open problem (Odena, 2019). For these reasons, the batch size in the experimental setup is set to 256 to strike a balance between the lower and higher values found in the literature. Moreover, the current architecture is set up such that the CPPN generates all pixel values in parallel as opposed to one-by-one. A much larger batch size can then be problematic due to excessive computational requirements. Training a traditional generator with a batch size of N requires the network to generate N images, while a CPPN using the same batch size is activated $N \cdot ImgDim^2$ times — once for each pixel in each output image of the batch.

The number and sizes of the categorical and continuous variables in the latent code **c** are inspired by one of the configurations used in the original InfoGAN paper (Chen et al., 2016). Additionally, this choice is further motivated by the intuition that a larger latent code can be harder or impossible for the auxiliary net to reconstruct, thus leading to a more unstable training process. It seems plausible that most datasets contain examples that overall exhibit at least some distinct properties, e.g. distinct classes, colors, sizes, etc., that can be captured by one categorical variable and two continuous variables.

The size of the latent noise vector **z** is set in accordance with current literature (Miyato et al., 2018; Kurach et al., 2019).

Finally, the random seed was early on set to an arbitrary fixed number, and the same seed has subsequently been used in all experiments and has worked well.

Optimization

The default GAN objective used is WGAN-GP. The loss function for the discriminator is then an approximation of the Earth-Mover (EM) distance between the data distribution and the generated samples. The generator loss, on the other hand, is concerned with maximizing the discriminator's loss. The auxiliary net uses a loss that is based on how well it is able to reconstruct the latent code — that is, the original code **c** that was provided as an input to the generator — of a generated image. The loss of the categorical variables in the latent code is calculated by the cross-entropy (Section 2.2.2) of the softmaxed outputs. The loss of the continuous variables is calculated using mean-squared error (MSE). The total loss for the auxiliary net is then the sum of these two losses, and during training both the auxiliary net and the generator's parameters are adjusted with a goal of minimizing the value of this loss.

The Adam optimizer is commonly used in the literature (Heusel et al., 2017; Miyato et al., 2018), and it has been used as the optimizer for all experiments involving the minimization of loss functions. The optimizer adjusts the parameters of any given

Table 5.1: Common hyperparameters across most experiments.

Batch size	256
Adam optimizer β_1 auxiliary net	0.5
Adam optimizer β_1 discriminator	0.5
Adam optimizer β_1 generator	0.5
Learning rate auxiliary net	0.0001
Learning rate discriminator	0.0003
Learning rate generator	0.0001
CPPN scaling factor	10
Size of latent noise vector	128
Size of latent code vector	12
Number of continuous variables	2
Number of categorical variables	1
Size of categorical variables	10
Random seed	9

network after each training step. The optimizations are performed in the following order: discriminator, generator and, if applicable, the auxiliary net. The default number of training steps used when training a model is roughly 100 000.

Architectures

The default architectures of the auxiliary net and discriminator are presented in Table 5.2 and Table 5.3, respectively.

Since all of the training data is normalized, the images input to the discriminator lie in the range $[-1, 1]$. The generator produces images in the aforementioned range by utilizing a tanh activation function at the output layer.

Similar to the related work in Section 3.2, the CPPN architecture used in the experiments generates all pixels in parallel instead of one by one.

When the generator in a GAN is an MLP or a CNN, the input is the latent noise vector $\mathbf{z}$ and, depending on the inclusion or exclusion of the auxiliary net, the latent code $\mathbf{c}$. This goes for the CPPN as well, with the addition of the coordinate inputs $\mathbf{x}$, $\mathbf{y}$ and possibly the distance term $\mathbf{r}$.

When residual blocks are used in the generator, there is an additional fully-connected layer used after the input layer that resizes the inputs such that their dimensions match the size of the final layer of the block. This resizing layer does not use any activation function.

The architecture of the MLP and CNN generators can be found in Appendix 2.

Biases and Weights

The weights of the auxiliary and discriminator network, as well as MLP and CNN generators, are initialized according to the Xavier weight initialization scheme (Section 2.4.1),

Table 5.2: The default architecture for the auxiliary net.

Layers	**Details**	**Normalization**	**Activation function**	**Output size**
All layers of the discriminator except the last	-	-	-	-
Fully connected	-	Spectral normalization	LReLU	128
Fully connected	Output size and activation function is dependent on the type of latent code used	Spectral normalization	Softmax or none	12

Table 5.3: The default architecture for the discriminator.

Layers	**Details**	**Normalization**	**Activation function**	**Output size**
Convolutional	Kernel size 4, stride 2	Spectral normalization	LReLU	64
Convolutional	Kernel size 4, stride 2	Spectral normalization	LReLU	128
Convolutional	Kernel size 4, stride 2	Spectral normalization	LReLU	256
Fully connected	-	Spectral normalization	None	1

while the biases are all initialized to 0. Any given CPPN is, however, initialized to either "small" or "large" weights. In the context of small initial weights or a weight initialization scheme leading to such weights, the weights are sampled from a Gaussian distribution $\mathcal{N}(0, 0.02)$. Similarly, when speaking of large initial weights, the weights are, again, drawn from a Gaussian distribution, but with a larger standard deviation: $\mathcal{N}(0, 1.0)$.

Sampling

Both the latent noise vector $\mathbf{z}$ and the continuous variables of the latent code $\mathbf{c}$ are sampled from a uniform distribution $U \sim [-1, 1]$, while the categorical variables of the code are sampled from a categorical distribution.

To evaluate the effects of a given categorical variable, a grid of samples is visualized where each column uses the same latent noise vector $\mathbf{z}$, while each row corresponds to a different value of the categorical variable. All continuous variables are set to 0, and all other categorical variables are fixed at their first category. A similar process is used in order to analyze the effects of the continuous variables. A grid of samples is visualized, where the value of the continuous variable in question varies from -2 at the leftmost column to 2 at the rightmost column. Moreover, each categorical variable is fixed to the value of its first category and the remaining continuous variables are set to 0. Each row uses the same latent noise vector $\mathbf{z}$. Note that the interval used when varying a continuous variable, i.e. $[-2, 2]$, includes values greater than what it was trained on — that is, $[-1, 1]$. This pushes the visual features the variable controls to extremes, which can make it easier it to discern which visual properties the variable controls.

The high-resolution samples generated by a CPPN, either as a stand-alone network or as the generator in a GAN, are generated at a resolution of 1200x1200. During training, however, all types of generators produce images that are of the same dimensions as the training data — that is, at relatively low resolutions.

5.3 Experimental Results

This section contains the results from the conducted experiments, experiment-specific setup descriptions, and short discussions and evaluations of the results. Any deviations from the default experimental setup presented in Section 5.2 will be mentioned if necessary.

The figures and the results they contain are often used as a convenient way to refer to the underlying networks instead of constructing schematics and separate sections for each network, and phrases such as "[...] The network in Figure *x.x* [...]" will be heavily used even though the respective figure does not contain any actual network descriptions, but merely the results generated by one or multiple networks.

Unless otherwise mentioned, only the hidden layers will be considered when referring to the layers of a given network in phrases such as "[...] The network consists of four layers [...]". The reason for this is that the output layers are fixed in terms of the activation functions, sizes of initial weights and number of output units across all experiments. The same goes for the the resizing layer that is added when using residual blocks in a CPPN — that is, this layer is not considered unless explicitly noted.

5.3.1 Compositional Pattern-Producing Network

There are a couple of specific hyperparameters associated with the usage of CPPNs that can be tweaked to vary the visual properties of the generated images. These include a scaling factor s that scales the inputs to lie in an interval $[-s, s]$, and the inclusion, exclusion and type of distance term $\mathbf{r}$. Moreover, the network depth and architecture, as well as activation functions and weight initialization schemes used, can all have an effect on the features exhibited by the generated images. Multiple experiments with CPPNs of various configurations have been run in order to examine the causal relationship between a given configuration and the visual characteristics of the generated images. None of the CPPNs in this first set of experiments have been trained. The networks are constructed with specific architectures and parameters, initialized with random weights, and then tasked with generating images.

By default, the layer size is 128, and the weights of each layer are initialized to large values. The scaling factor s is only applied to the coordinate inputs — that is, $\mathbf{x}$, $\mathbf{y}$, and the distance term $\mathbf{r}$, if included. Both the latent noise vector $\mathbf{z}$ and code $\mathbf{c}$ are input to the CPPNs when generating images, and they are randomly sampled according to the default configuration. For a given image, these input vectors stay fixed for all pixel values generated by the network. The distance term $\mathbf{r}$ is by default included as an input, and it is calculated by using the Euclidean distance measure.

Scaling Factor and Initial Weights

The generated images from a CPPN using different scaling factors s are shown in Figure 5.1. The scaling factor controls the interval in which the coordinates $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{r}$ lie. The scaling factor effectively controls the zoom of an image: relative to a fixed s, a larger scaling factor leads to a zoomed out image, while a smaller value lets you zoom in. This behavior can be explained by the fact that a large scaling factor leads to a coordinate system that is a superset of all coordinate systems produced by smaller scaling factors, with the difference being that the smaller coordinate systems are represented at a lower level of granularity. Conversely, a smaller scaling factor leads to a smaller coordinate system where the details of the current coordinates become finer compared to images generated by higher scaling factors. As an example, a scaling factor $s = 1$ results in a coordinate system that spans the interval $[-1, 1]$, while $s = 50$ results in a larger coordinate system defined by the values in $[-50, 50]$.

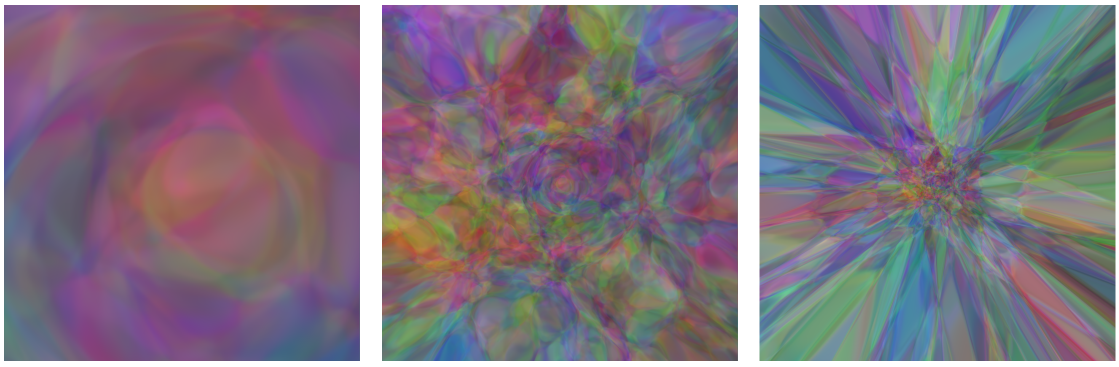

(a) Scale $s = 1$ (b) Scale $s = 10$ (c) Scale $s = 50$

Figure 5.1: Generated images from CPPNs with four layers and tanh activation functions

Figure 5.2 shows the results of experimentation with different weight initialization schemes. With the exception of the scheme used, the underlying CPPNs are identical. It seems that the level of sharpness and detail in the images becomes higher as the sizes of the initial weights become larger. In standard neural networks it is beneficial to initialize weights close to 0 to improve the optimization process, but when generating images with CPPNs the results end up being clearer and more detailed when the weights are initialized to larger values than normal.

(a) Initial weights sampled from $\mathcal{N}(0, 0.5)$ (b) Initial weights sampled from $\mathcal{N}(0, 1.0)$ (c) Initial weights sampled from $\mathcal{N}(0, 5.0)$

Figure 5.2: Generated images from CPPNs with one layer and tanh activation function

Activation Functions

In traditional neural networks it is normal to use activation functions that exhibit desirable properties in terms of optimization and that have proven themselves to improve the learning process. In CPPNs, however, the activation functions might be chosen based on the visual characteristics that their presence manifest in the generated images. These functions might perform very poorly in an optimization setting, e.g. leading to saturation of units, weak gradients, etc., but they can nonetheless be used to produce different patterns and shapes that are inherently unique to the functions used.

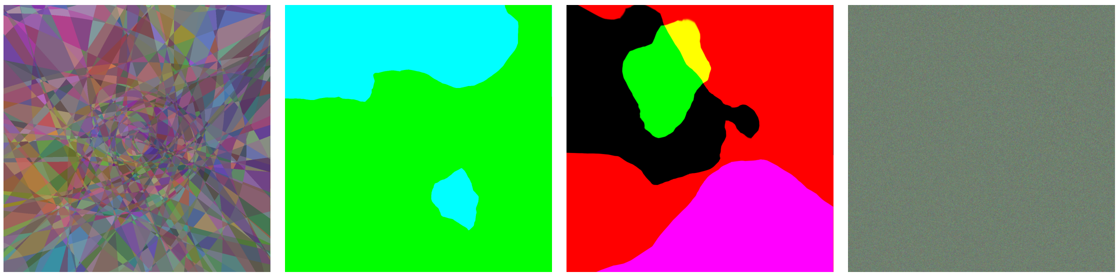
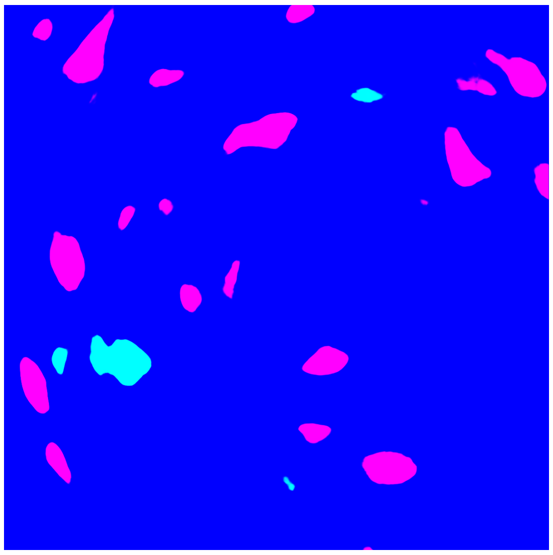
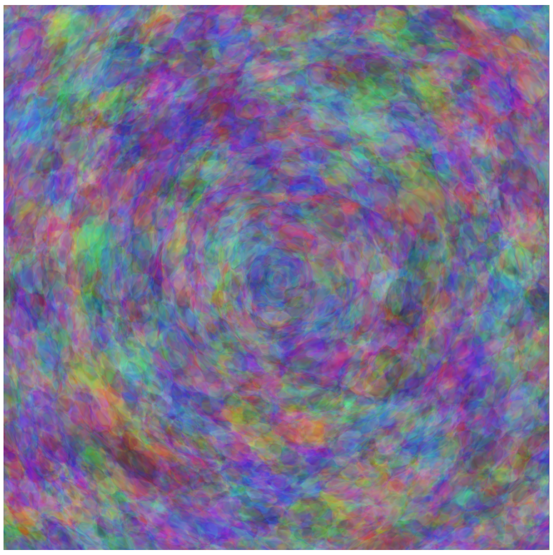
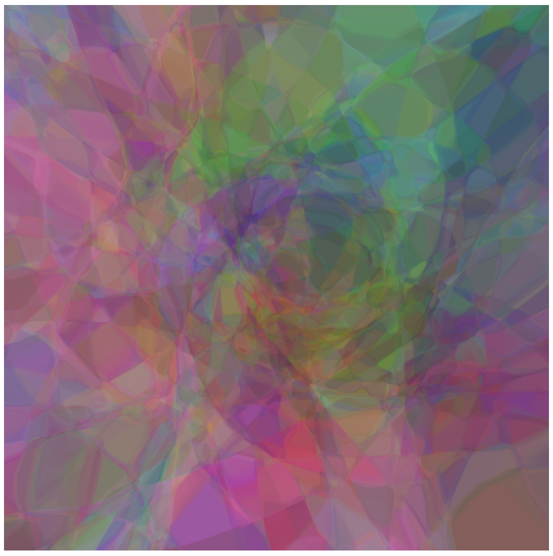

(a) Activation functions are all sign function (b) Activation functions are all absolute function (c) Activation functions are all LReLU (d) Activation functions are all sin

Figure 5.3: Generated images from CPPNs with four layers and with various activation functions

Figure 5.3 shows generated images from four CPPNs containing only sign, absolute, LReLU and sin functions, respectively. There are clear visual differences between the images generated by these networks. Using a sign function results in sharp lines and patterns. The patterns are similar to the ones presented in Figure 5.1b, which is likely due to the fact that the tanh function used there can, to a certain extent, be considered


(a) Activation functions are sin and remaining LReLU


(b) Activation functions are sin and remaining tanh


(c) Activation functions are alternating LReLU and tanh

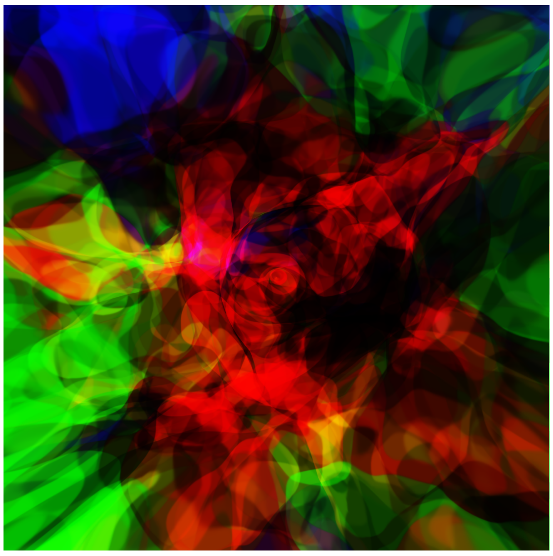

(d) Activation functions are alternating tanh and LReLU

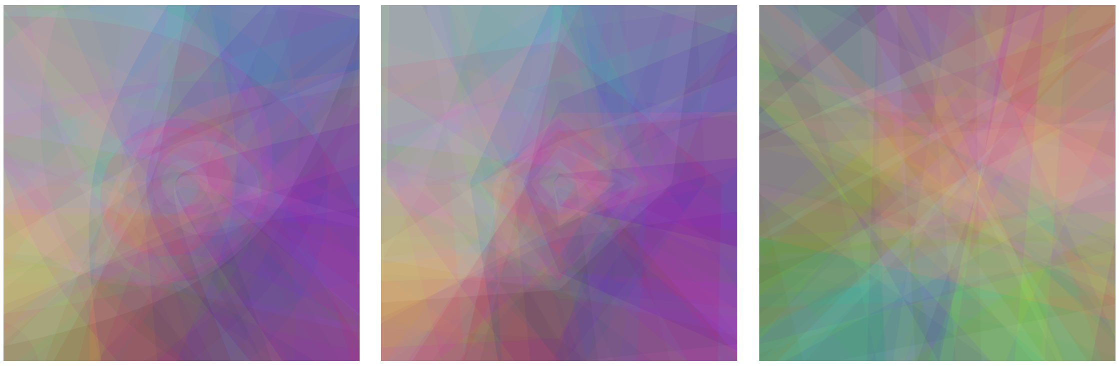
Figure 5.4: Generated images from CPPNs with four layers and with a mix of activation functions

a smooth version of the sign function[1]. As sin is a periodic function, the use of it as an activation function gives rise to repeating patterns — a behavior that is more evident in Figure 5.4.

Additionally, using a CPPN with a mix of different activation functions gives rise to patterns that exhibit characteristics of all the functions in the mix. Four different CPPNs were set up with combinations of sin, LReLU and tanh functions in various orders of appearance in the network hierarchy. The results of these experiments are presented in Figure 5.4. The periodic nature of the sin function is apparent both when mixing it with LReLU and tanh functions, leading to repeating patterns in both cases. Moreover, the combinations of alternating LReLU and tanh activation functions similiarly produce results that contain a blend of the distinct visual properties that have been evident in experiments where only one of them is present.

Distance Term

Most earlier works with CPPNs have included a distance term $\mathbf{r}$ as an input to the network in addition to the coordinate vectors $\mathbf{x}$ and $\mathbf{y}$. For each coordinate, this term represents the distance between the respective coordinate and the center. As an example, when using a scaling factor $s = 1$, the coordinate corresponding to the upper left corner of the image would be given by $(x, y) = (-1, -1)$. The Euclidean distance between this coordinate and the center $(0, 0)$ would then be calculcated as $r = \sqrt{(-1-0)^2 + (-1-0)^2} = \sqrt{2}$. The nature of a distance measure then leads to many pairs of coordinates sharing the same distance r. As such, this term assists the network in generating symmetric shapes, and supplies the network with a type of global context as opposed to the strictly local context of $\mathbf{x}$ and $\mathbf{y}$.



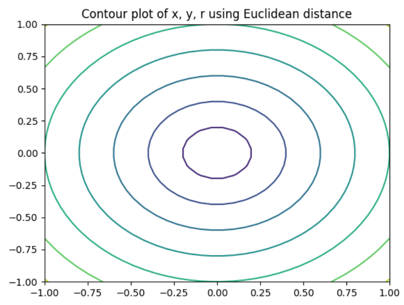
(a) Euclidean distance from center is included as an input

(b) Manhattan distance from center is included as an input

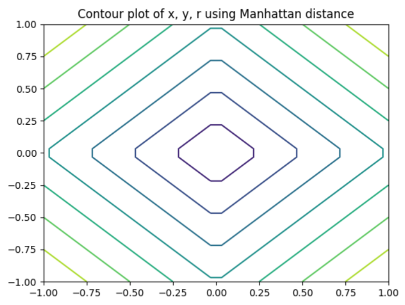
(c) Distance term is not included as an input

Figure 5.5: Generated images from CPPNs with one layer and with sign activation function

[1]Note how $tanh(kx)$ approximates $sign(x)$ as $k \to \infty$.



(a) **r** is calculated using the Euclidean distance from center



(b) **r** is calculated using the Manhattan distance from center

Figure 5.6: Contour plots of different types of distance measures

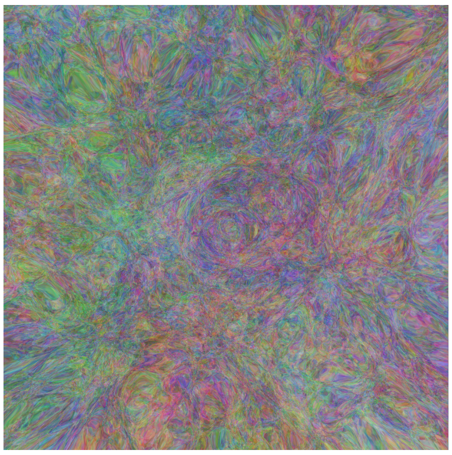
Multiple experiments have been carried out where the effects of both inclusion and exclusion of the distance term as an input, as well as the type of distance measure used, have been visualized. The results of these experiments are presented in Figure 5.5, and they show that including a distance term results in a distinct, symmetric pattern emanating from the center of the respective images. When using the Euclidean distance, this pattern is represented as a circular shape, and there is a great deal of curved lines found in the image. Using the Manhattan distance, however, leads to a diamond shape and straight lines. This behavior further corresponds with the characteristics of the contour plots of the two distance measures presented in Figure 5.6. The plots show how distance terms of different types change based on the input coordinates **x** and **y**. The shapes of the contour plots are quite similar to the ones found in the CPPN-generated images in Figure 5.5 that incorporate the distance term. Finally, excluding the distance term altogether results in an image that lacks a bias towards symmetry and is more defined by the characteristics of the activation function used, as shown in Figure 5.5c.

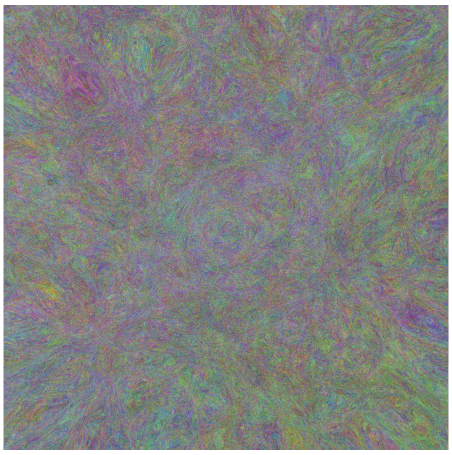
Architectures

The following experiments pertain to CPPNs of varied architectures, including CPPNs with varying network depths, as well as CPPNs containing residual blocks.

The results from varying the network depth of three fully-connected CPPNs with tanh activation functions are shown in Figure 5.7. Since the weights are initialized to large values, and given the nature of the activation function used, it seems likely that the layers become more saturated as the depth increases. If this is the case, it would lead to a higher number of activations in the network across inputs, which would explain the lack of distinct details in the resulting images as the depth increases.

The experiments using residual blocks are presented in Figure 5.8. The intermediate layers of each block have large initial weights, while the weights of the final layer are initialized to small values. Each residual block contains three layers, and each CPPN contains five such blocks. The images in Figure 5.8a and 5.8b are generated by the same


(a) 8 tanh layers.


(b) 12 tanh layers.

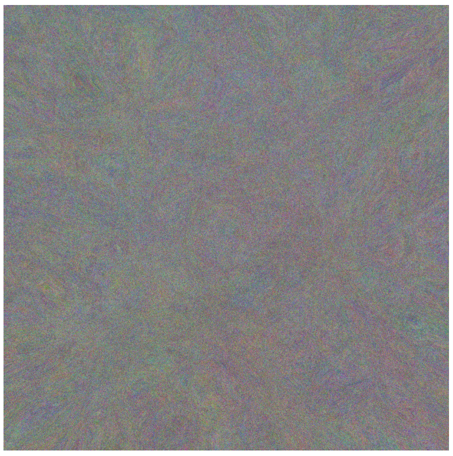
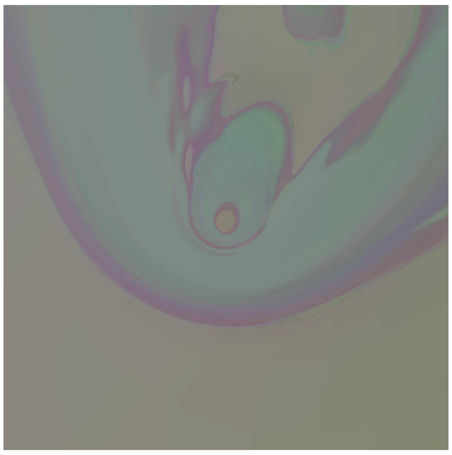
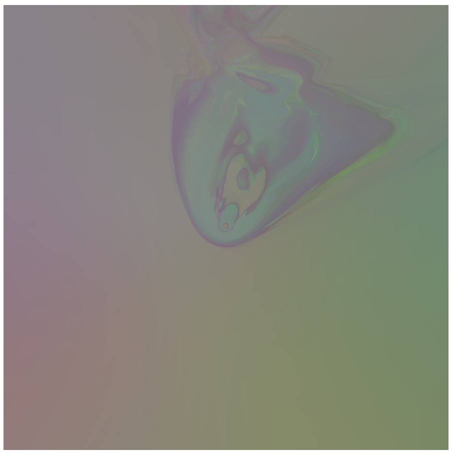

(c) 16 tanh layers.

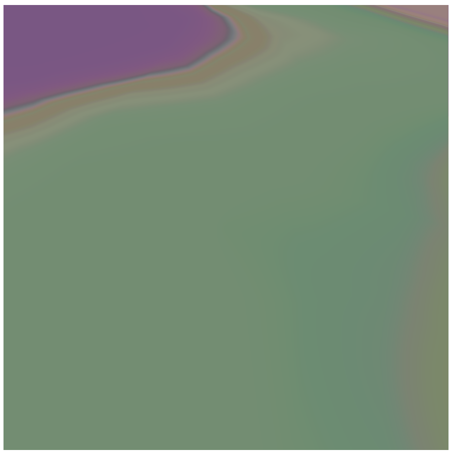
Figure 5.7: Generated images from CPPNs with tanh activation functions and different network depths.

network, only with different scaling factors. The network uses tanh activation functions for all layers in the residual blocks. The same is true for the images in Figure 5.8c and 5.8d, with the difference being the utilization of LReLU and tanh in the intermediate layers and final layer of each residual block, respectively. Since the final layer of each residual block uses a small weight initialization scheme, the output of each block is more defined by the initial input to the block compared to the intermediate layers[2] It is, however, evident that the activation functions of the intermediate layers have an effect on the images generated by the networks: using only tanh leads to more smooth, circular

[2]The output from the last intermediate layer will be multipled with small weights and summed up. The result will likely be small due to the nature of the weights. The initial input to the block is added directly to this small sum using the skip-connection, and the final output of the block should then be more characterized by the initial input compared to the intermediate layers.


(a) Residual blocks with tanh. Scale $s = 10$.


(b) Residual blocks with tanh. Scale $s = 50$.


(c) Residual blocks with LReLU and tanh. Scale $s = 10$.

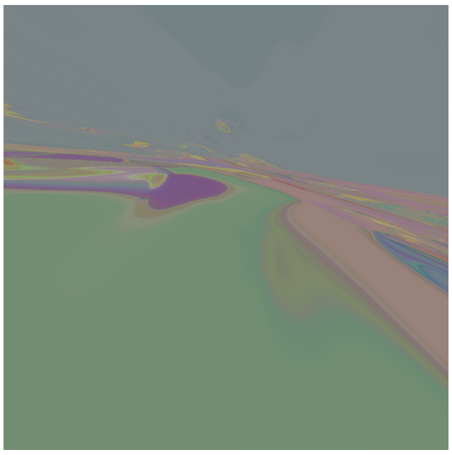

(d) Residual blocks with LReLU and tanh. Scale $s = 50$.

Figure 5.8: Generated images from CPPNs with residual blocks.

shapes and curved lines, while using LReLU leads to sharper and straight lines.

5.3.2 Generative Adversarial Network

A GAN containing a non-CPPN generator, e.g. a multi-layer perceptron (MLP) or a convolutional neural network (CNN), is expected to perform better than one containing a CPPN. The main reason for this is that these conventional generators incorporate more context when generating the images, and such contexts have proven themselves useful in the image domain where the content of one pixel is usually dependent on the neighbouring pixels. Moreover, these traditional generators generate an entire image at once, while a CPPN generates one pixel at a time. As such, a CPPN has no context at inference time, and the network itself has to be trained to capture and function as the context.

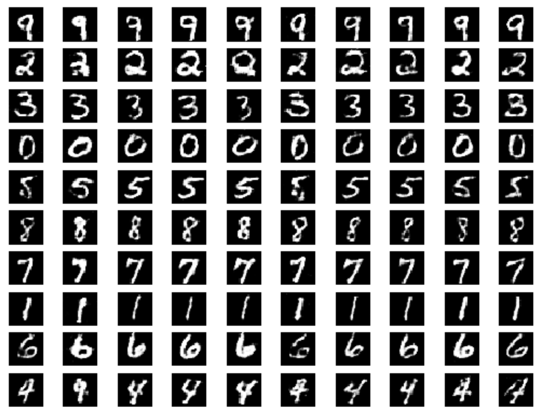
It is beneficial to train GANs with MLP or CNN generators as a way to establish an upper bound for the visual quality of generated results at low resolution. If such a GAN is unable to approximate the data distribution, it is unlikely that a GAN using a CPPN will. Moreover, the results from these more conventional GANs can be used to evaluate the performance of GANs that uses CPPNs as the generator.

Finally, the experiments conducted on traditional GANs will be beneficial in establishing a performant configuration that can be used as a base for later experiments with IMPGANs.

Maximimization of Mutual Information

The results from experimenting with the inclusion and exclusion of the auxiliary net when training on the MNIST dataset is shown in Figure 5.9. The generator is configured as an MLP, and the architecture used can be found in Appendix 2.

When the auxiliary net is included, the parameters of both the auxiliary net and the generator are adjusted to maximize the mutual information between the latent code **c** and the generated images. When the net is excluded, however, the generator is not insentivized to treat the latent code any different than the latent noise vector **z**: the generator can effectively ignore or attach little significance to the latent code. This behavior is evident in Figure 5.9, where one network is penalized for not maximizing the mutual information, while the other one is not. Varying the latent code in the network that is trained without the auxiliary net has little or no discernible effect on the generated images.


(a) With auxiliary net.

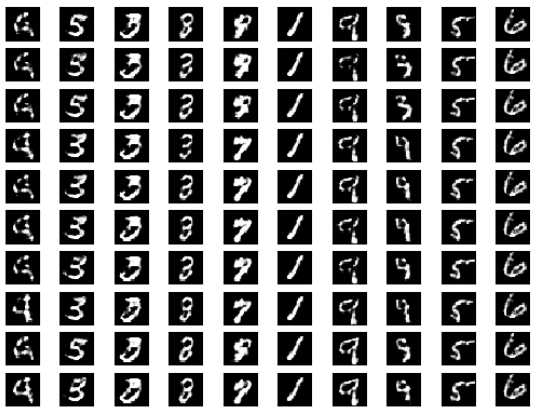

(b) Without auxiliary net.

Figure 5.9: Generated images from GANs with and without an auxiliary net for mutual information maximization, where each row corresponds to a different value of the categorical variable.

Different Datasets

The results from training GANs on different datasets are presented in Figure 5.10. All of the networks are identical except the sizes of the convolutional layers in the discriminator and generator as a result of different image dimensions. The results should serve as a baseline for comparing the quality of the results from subsequent experiments with IMPGANs. The generator is based on a convolutional architecture, and specific details can be found in Appendix 2. The results from varying the latent codes can be found in Appendix 4.

5.3.3 Information Maximizing and Pattern-Producing GAN

In this section, the experiments pertaining to an Information Maximizing and Pattern-Producing GAN (IMPGAN) will be presented. The term IMPGAN is used to describe a GAN that uses a CPPN, hence pattern-producing, as its generator, as well as mutual information maximization for disentangled representations of the latent space. All CPPNs in this section are trained in the context of an IMPGAN.

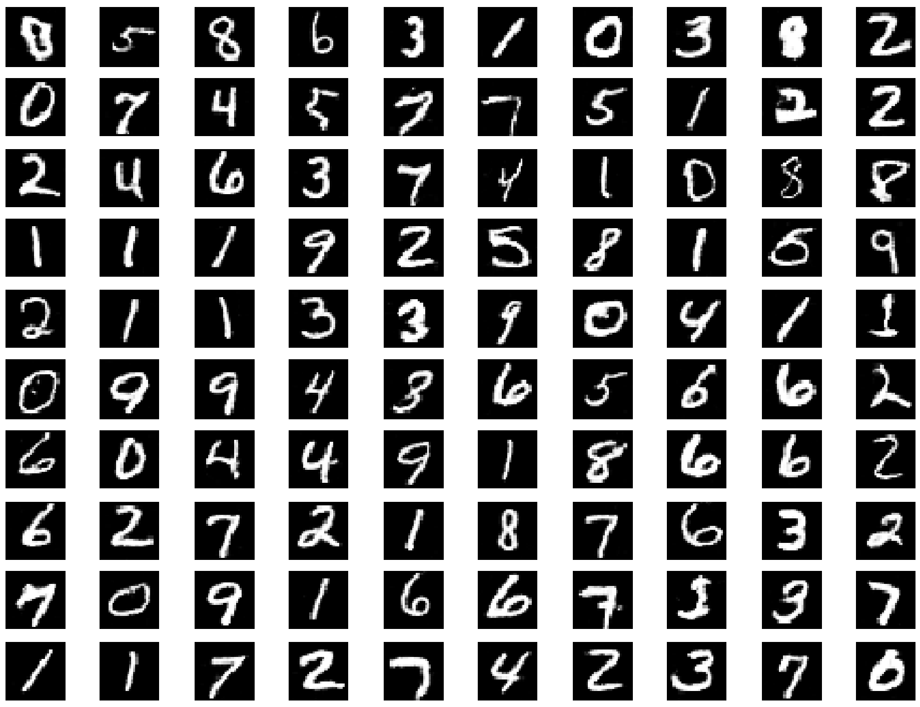
The experiments will be performed across multiple datasets of different complexity. This includes the MNIST, Fashion MNIST, CelebA and CIFAR-10 datasets.

Unless otherwise specified the size of any given layer is 128.

MNIST

The low level of complexity exhibited by the MNIST dataset makes it a good candidate for a multitude of experiments that might not be feasible on other datasets. As an example, some experimental configurations, e.g. large weight initializations and/or unconventional activation functions, might be particularly ill-suited in the context of optimization and reaching convergence. Since the MNIST dataset is relatively easy to train on, however, the network might still be able to reach a certain degree of convergence using such configurations. This might not be the case for other datasets, where successful training using such combinations of activation functions, hyperparameters, etc., might require excessive computational resources or not be possible at all. As such, the experiments pertaining to the MNIST dataset can serve as a testing ground for more exotic configurations. Finally, the results presented in this section might prove to be more effective at highlighting the principal effects of different configurations compared to experiments conducted on more complex datasets containing more color channels and a higher degree of diversity in both objects and shapes.

Activation Functions
Generated images from four different CPPNs with different activation functions are shown in Figure 5.11, where the functions are presented in the order in which they appear in the respective network. These networks were all trained on the MNIST dataset and initialized with large weights.


(a) Trained on the MNIST dataset for $\sim 100\,000$ steps.

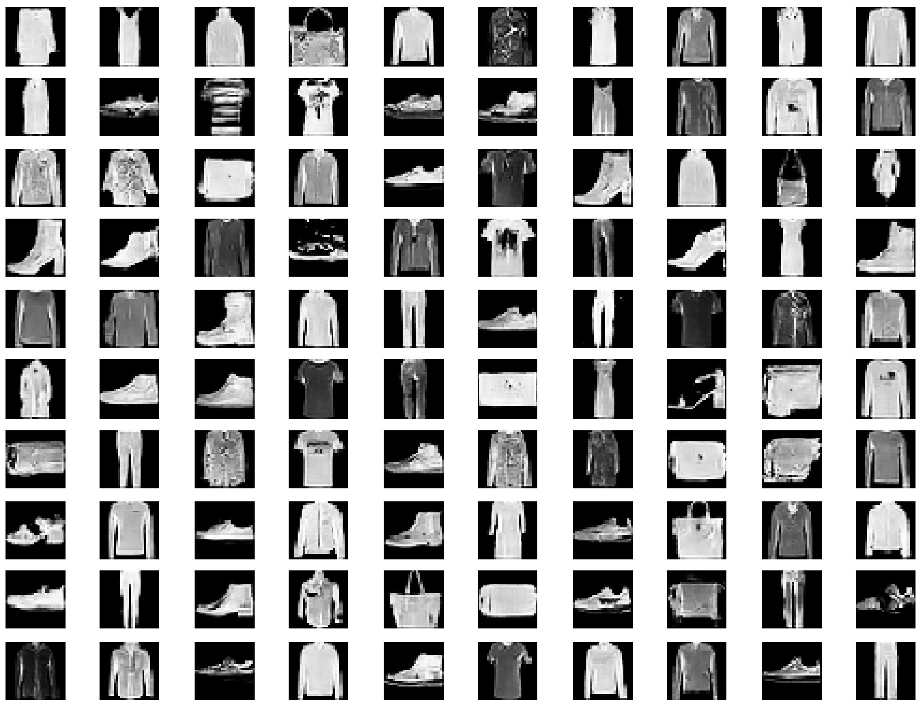
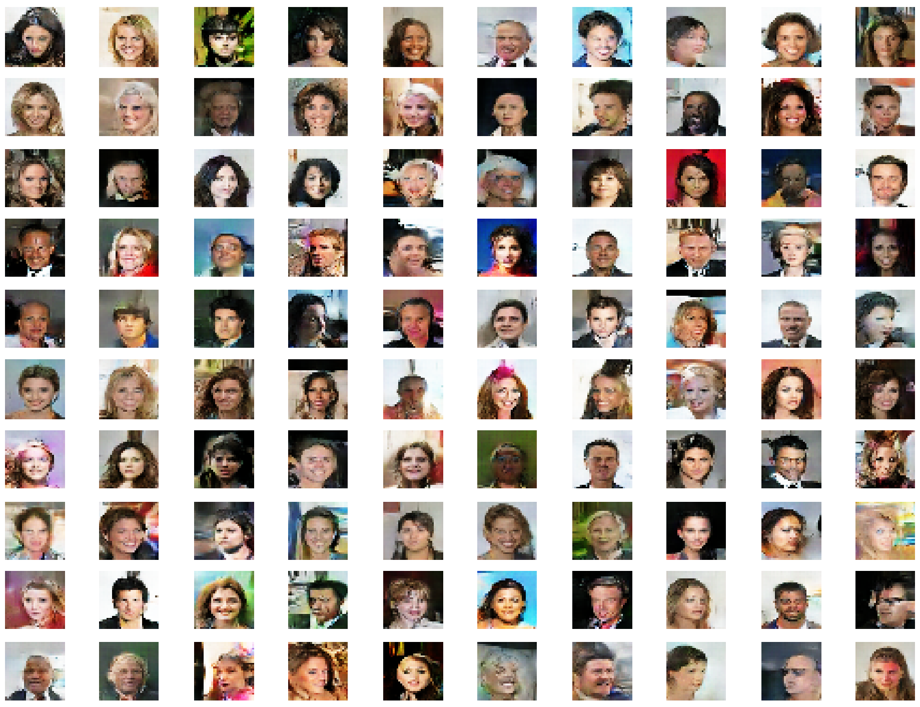

(b) Trained on the Fashion MNIST dataset for $\sim 100\,000$ steps.

Figure 5.10: Generated images from GANs trained on different datasets.


(c) Trained on the CelebA dataset for $\sim 250\,000$ steps.

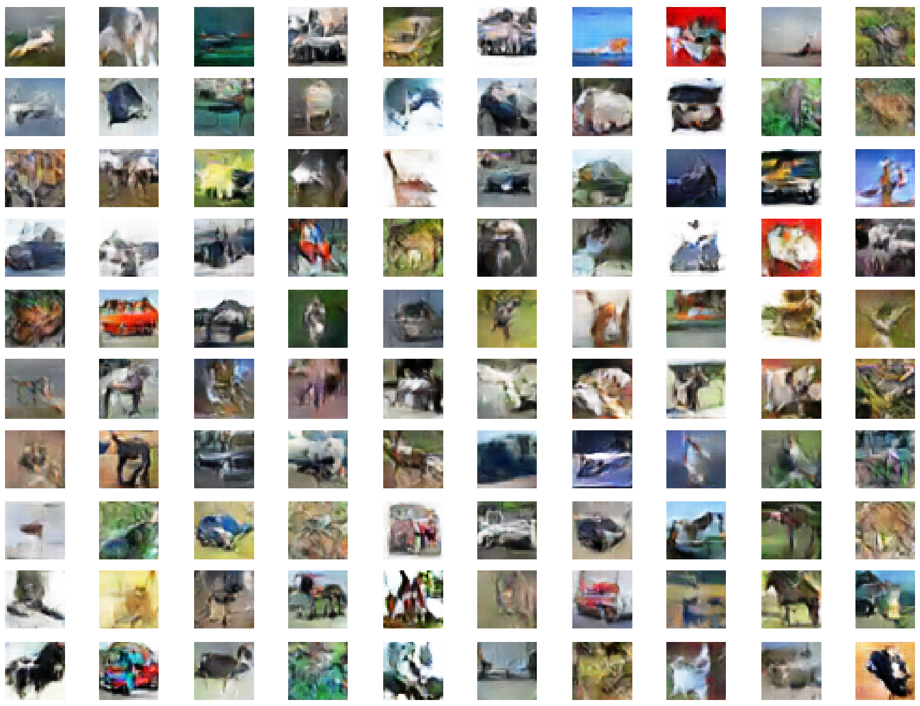

(d) Trained on the CIFAR-10 dataset for $\sim 250\,000$ steps.

Figure 5.10: Generated images from GANs trained on different datasets.

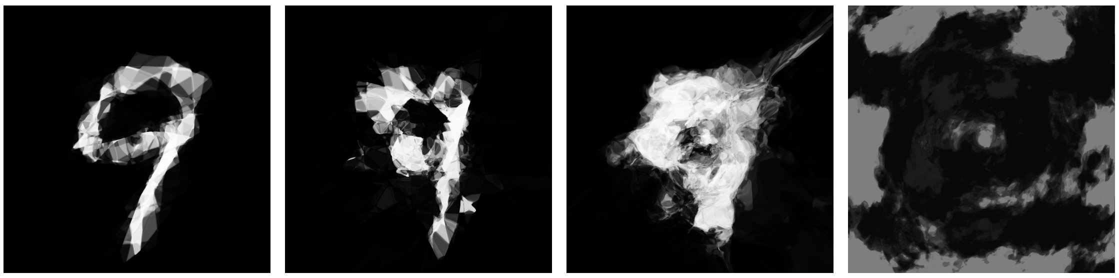

(a) LReLU and three tanh layers (b) LReLU and six tanh layers (c) Seven tanh layers (d) Sin and six tanh layers

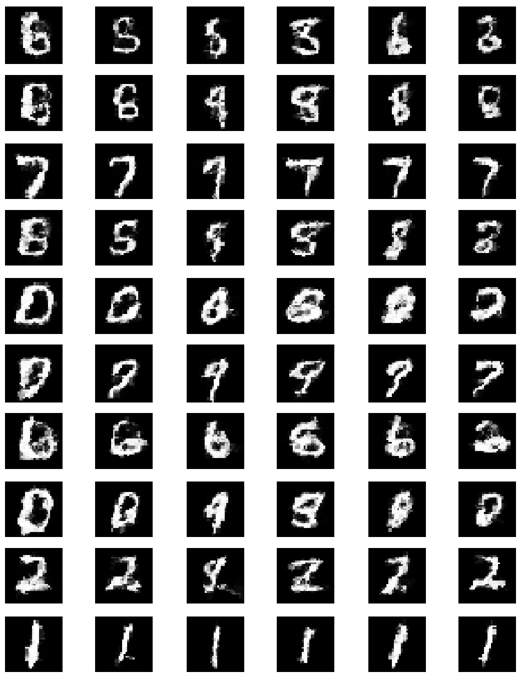
Figure 5.11: IMPGANs with various activation functions trained on the MNIST dataset

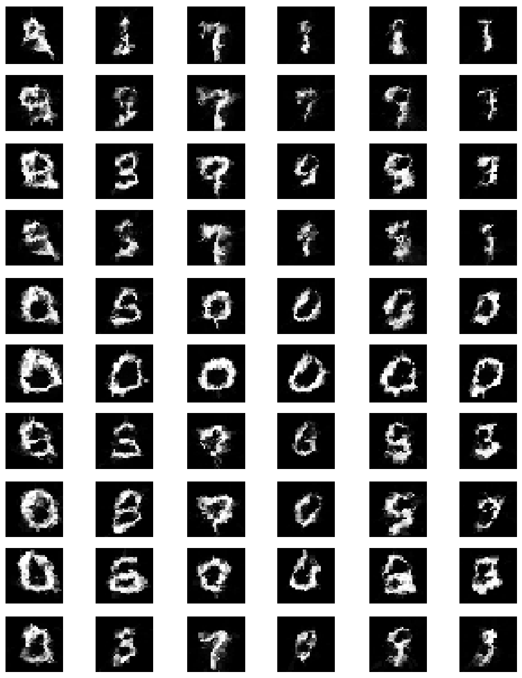
Figure 5.11a shows how using a combination of LReLU and tanh functions leads to characteristics of both of these functions being apparent in the generated images. Increasing the depth of such a network (Figure 5.11b) adds more complexity to the images. While a deeper network increases the training time, some activation functions may lead to one deep network converging faster than another: in Figure 5.11b, the seven-layer network using both LReLU and tanh is producing samples that are closer to the distribution of the training data compared to the network only using tanh in Figure 5.11c. Finally, using sin as an activation function in the first layer makes it hard to reach any convergence at all, as shown in Figure 5.11d. The repeating nature of this periodic function, together with the high initial weights, likely makes the network very difficult to optimize.

Grids of low resolution, 28x28 images from the first three CPPNs of Figure 5.11 are shown in Figure 5.12. As evidenced, even though the different networks generate images which exhibit large differences at high resolution, these differences are much less pronounced at the original resolution of the dataset. The complexity of a given network does, among other factors, have an effect on the number of steps it must be trained for in order to reach a certain level of visual quality. These networks are, however, all trained for the same amount of training steps even though they are all of different complexity. As such, even though there are some visual differences at low resolution, such differences are expected to gradually decrease as the training progresses and the networks more accurately approximate the distribution of the original dataset. The distinct visual properties at high resolution will in this case be less pronounced, but they should in any case still be visible.

Initial Weights

The results from experiments using various weight initialization schemes are shown in Figure 5.13, and with the exception of the sizes of the initial weights, these networks are the exact same as the ones presented in Figure 5.11a and Figure 5.11d, respectively. The network in Figure 5.13a uses small initial weights in all of its layers, and successfully converges and is able to generate all digits 0–9. The generated images at high resolution,


(a) LReLU and three tanh layers.


(b) LReLU and six tanh layers.

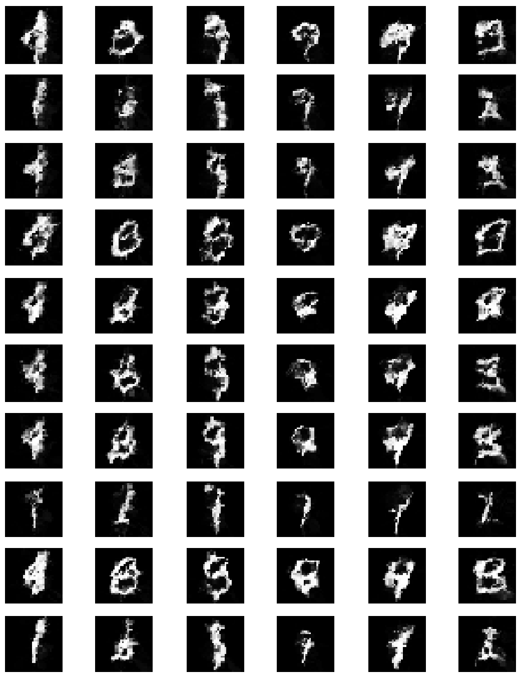
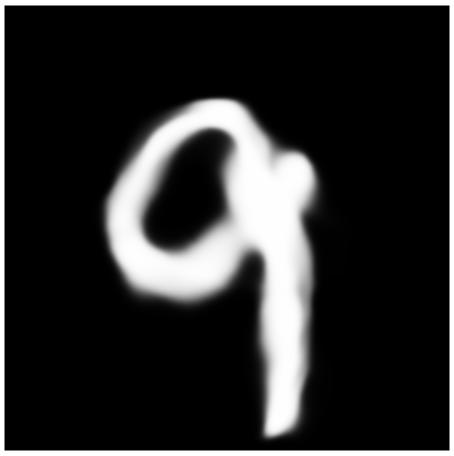

(c) Seven tanh layers.

Figure 5.12: Varying the categorical variable on IMPGANs trained on the MNIST dataset.


(a) LReLU and three tanh layers with small initial weights

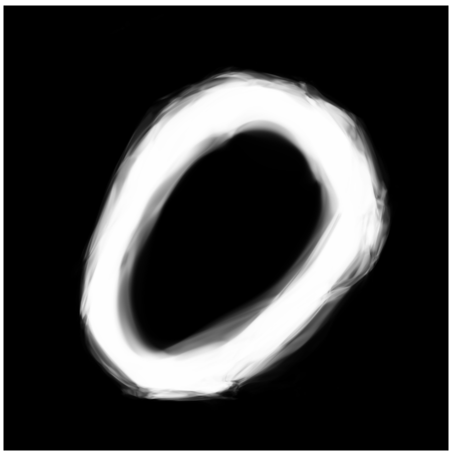
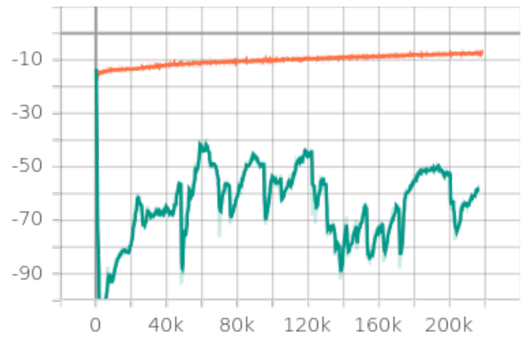

(b) Sin and six tanh layers with small and high initial weights, respectively

Figure 5.13: IMPGANs with different initial weights trained on the MNIST dataset

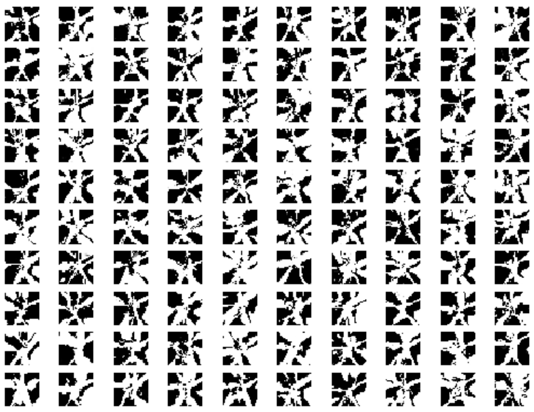
however, lack sharp details, and the images have a somewhat blurry and smooth look. The size of the initial weights are likely the reason for this, as the weights will not grow any larger than needed in order to improve on the training objective — which is concerned with generating convincing images at the original 28x28 resolution. The ensuing experiment uses small initial weights only in the first sin layer, while large initial weights are present in the remanining tanh layers. The network converges, as opposed to the similar network in Figure 5.11d, and gives of a relatively sharp look at high resolutions. The only difference between this network and the one using sin in Figure 5.11d is that the initial weights of the current sin layer are smaller.

Latent Codes

The results from using two categorical variables of size 10 and three continuous variables are shown in Figure 5.14, where the plot represents the loss of the discriminator over time: the orange line is the loss of an identical network with the default latent code configuration, while the green line is the current experiment. The x-axis represents the number of training steps, while the y-axis represents the value of the loss. Note that all of the plotted values are on the negative side of the y-axis. Both of these networks are using seven layers of tanh with large initial weights, and are trained for roughly 200 000 training steps. Using a larger latent code results in a network that fails entirely to converge, as evidenced by the quality of the images in Figure 5.14b, and a highly unstable training process compared to using a smaller latent code. There are many possible explanations for this behavior. First, the generator's weights are optimized to both generate realistic images and associate meaningful differences between the different variables in the latent code. These two objectives might not only be contradictory at times, but there might not be enough underlying categories in the dataset to actually create meaningful associations for all of the categorical variables.



(a) The loss of the discriminator plotted over time



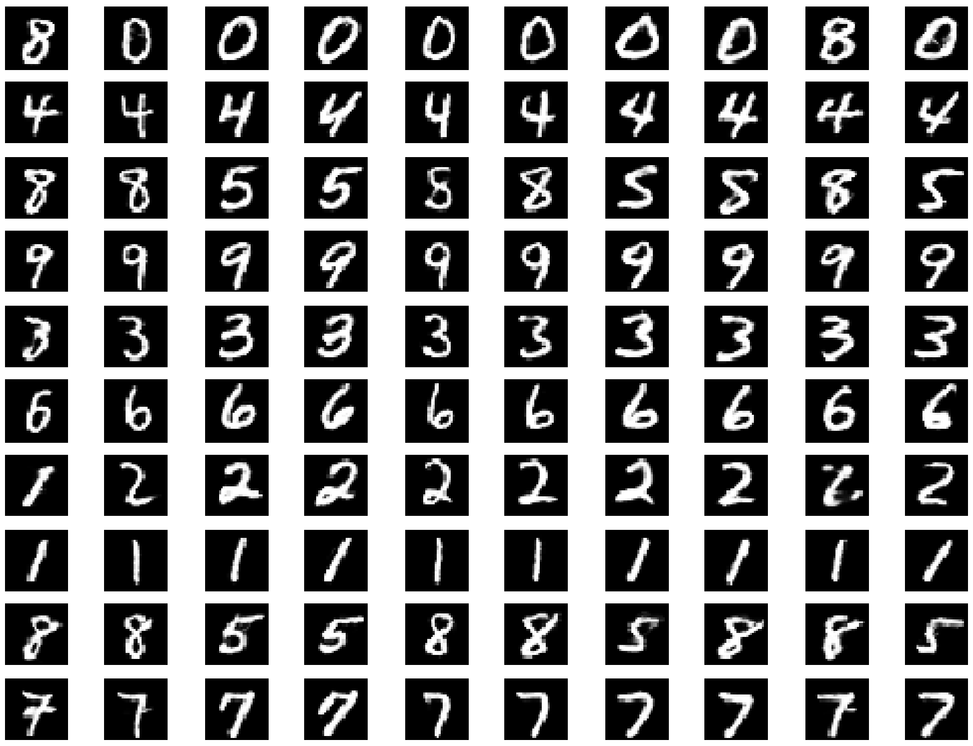
(b) A grid of randomly generated samples

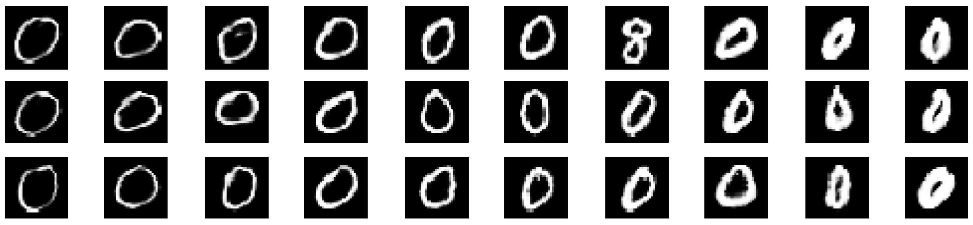
Figure 5.14: Latent code experiments on an IMPGAN trained on MNIST

The results from training a network using the default latent code configuration, and varying the values of the different variables, are shown in Figure 5.15. The network consists of one LReLU and three tanh layers with small initial weights. The generator seems to have learned to map each category of the categorical variable to a distinct digit class. There is a bit of overlap between similar digits, e.g. 8s and 0s sometimes end up in the same category due to their similarity, but this situation should be improved as the training progresses. The two continuous variables seem to correspond to digit thickness and rotation, respectively.

Distance Term

The inclusion or exclusion of the distance term **r** as an input has also been experimented with on this dataset, but it does not seem to have a very pronounced effect on the generated images. The effects are most visible in the early training phases, but as training progresses, the effect is decreased. The the network might adjust its weights as to make the effects of the distance term smaller over time if its inclusion is not beneficial. Some generated samples still exhibit a small degree of circular radiation from the center that is consistent with the contour plots in Figure 5.6. Some examples of this can be seen in Figure 5.11c and 5.11d. The visual effects of including the distance term seem to be more pronounced if coupled with large initial weights. Moreover, the exclusion of the distance term seems to have little or no effect on the level of convergence of the network or quality of the generated images compared to an identical network with this term included.


(a) Varying the value of the categorical variable


(b) Varying the value of the first continuous variable

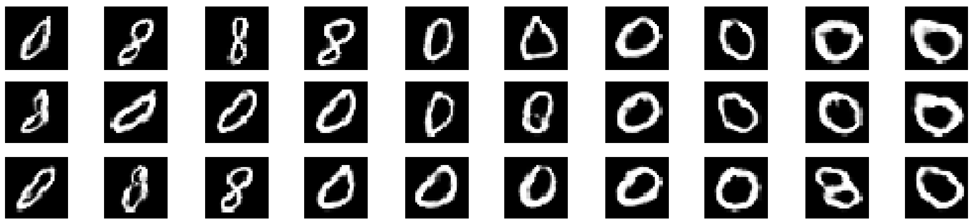
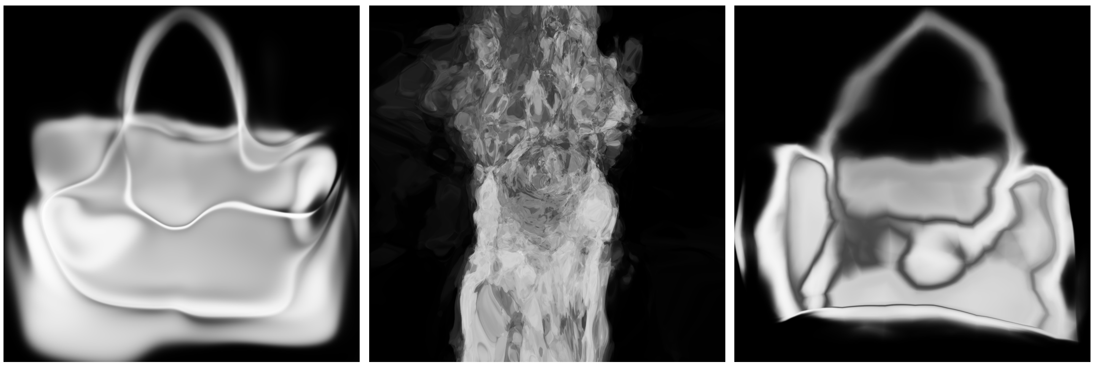

(c) Varying the value of the second continuous variable

Figure 5.15: Varying the latent code in an IMPGAN trained on MNIST



(a) Eight layers of tanh with size 64 and small initial weights (b) Seven layers of tanh with large initial weights (c) Five residual blocks with layer sizes 64 (tanh), 32 (tanh) and 64 (LReLU), and small initial weights

Figure 5.16: IMPGANs trained on the Fashion MNIST dataset

Fashion MNIST

The Fashion MNIST dataset marks a step up in complexity from the MNIST dataset. The dataset contains images of the same dimensions and number of color channels as the MNIST dataset, but the shapes and objects found in the images are more complex.

Experimentation with different weight initialization schemes, network depths and architectures, and activation functions are shown in Figure 5.16. The network using large initial weights in Figure 5.16b is, for the fixed amount of training steps used, not able to closely approximate the training data. The two networks using small initial weights and varying depth, however, both reach a very similar level of visual quality. The different architectures used in these two networks lead to various visual differences in the generated images. In Figure 5.16a, the overall properties are characterized by smooth and curved lines, while Figure 5.16c exhibits a stronger bias towards sharper lines and shapes.

The distance term is included for all experiments in Figure 5.16, but it does not seem to have any large discernible effects on the networks using small initial weights. The visual effects of the term's inclusion are more pronounced in the network using large initial weights, as shown in Figure 5.16b.

The categorical variable in the latent code usually corresponds to the different object classes found in the dataset, e.g. sweaters, shoes, purses, pants. This separation was more successful when using the residual architecture, which might be attributed to the depth of the network. One of the continuous variables consistently corresponds to changing the color of an item from black to white, or vice versa, while the other variable has been seen to control the size of a given item.

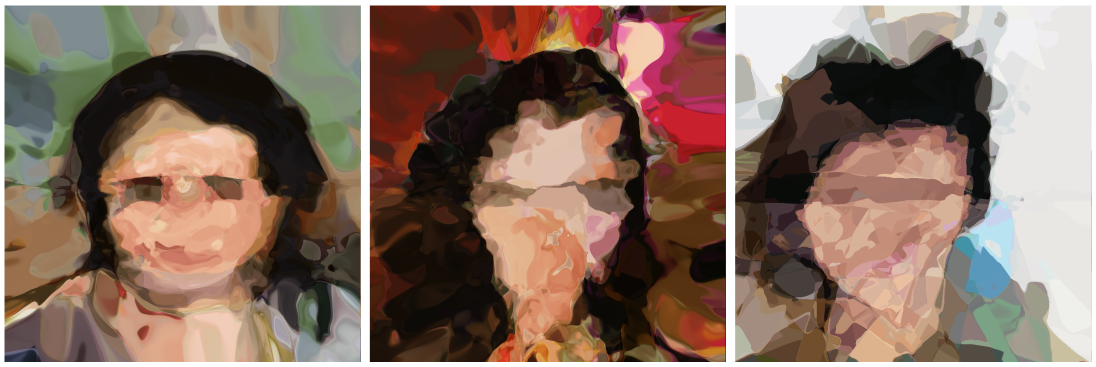

(a) Eight layers of tanh. (b) Eight layers of tanh. (c) Eight layers of alternating LReLU and tanh.

Figure 5.17: IMPGANs trained on the CelebA dataset

CelebA

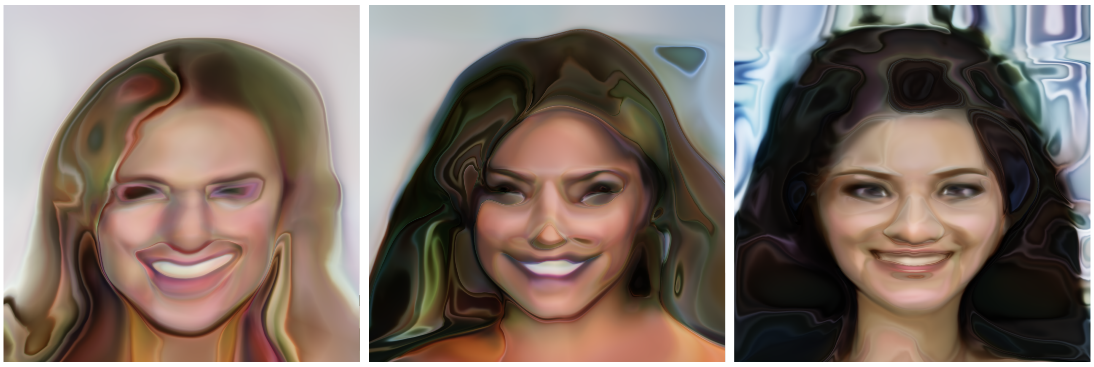
The CelebA dataset is much more diverse than the greyscale MNIST and Fashion MNIST datasets. The images contain faces of celebrities, use three color channels and contain a wide variety of colors. The experiments in this section will mainly revolve around deep architectures using MLPs and residual blocks. The experimental configurations will converge to results that exhibit a relatively high degree of realism compared to previous CPPN-generated results.

The networks in this section are by default trained on a version of the CelebA with images resized to 32x32. The experiments are run for approximately 250 000 training steps as opposed to the default 100 000.

MLP Architectures

The images in Figure 5.17 are generated by three different CPPNs with eight layers of sizes 128, 128, 128, 64, 64, 64, 32 and 32. All weights in the networks are initialized to large values, and $\beta_1 = 0.0$ is used in all of the Adam optimizers. The networks in Figure 5.17a and 5.17b only use tanh activation functions, while the Figure 5.17c alternates between LReLU and tanh for each layer. Only the network represented in Figure 5.17a uses the distance term as an input, and layer normalization is applied in the auxiliary net of this model instead of the default spectral normalization. Additionally, the networks in Figure 5.17b and 5.17c both use five continuous variables instead of two in their latent code.

The two first networks are able to obtain a high level of maximization of mutual information, i.e. varying the latent code predictably controls various features in the generated images. This is in contrast to the third network which, despite being of very similar architecture and with the principal difference being alternating LReLU and tanh layers, as opposed to only tanh, fails to attribute meaning to the latent code. In any case, the effects of the two meaningful latent codes in the successful networks are somewhat



(a) Residual network trained on 32x32 CelebA (b) Residual network trained on 32x32 CelebA (c) Residual network trained on 64x64 CelebA

Figure 5.18: IMPGANs trained on the CelebA dataset

hard to discern given that the networks have not been trained to a level of convergence that allows generation of high detailed images. The codes do, however, seem to mainly affect background colors, rotation, and face styles and sizes.

The effects of including or excluding the distance term are visible in Figure 5.17a, where the image has the distinctive circular pattern in the center of the image and an inherit bias towards symmetry compared to the two other images where the distance term is not included.

Finally, the network in Figure 5.17c that uses both LReLU and tanh functions exhibits more sharp lines and patterns compared to the other two networks.

Residual Architectures

In an attempt to increase the visual quality of the generated images at high resolution, experiments on deeper architectures utilizing residual blocks have been conducted. The main results from these experiments are shown in Figure 5.18, where each generated image is associated with a different CPPN.

The image in Figure 5.18a is generated by a network containing five residual blocks consisting of layers of sizes 64, 32 and 64. The activation functions used are tanh, tanh and LReLU, respectively, and the weights are initialized to small values. The result is a relatively sharp image with a mix of curved and sharp lines. The generator has clearly learned the concept of a face, but some finer details related to facial expression and shadowing are somewhat lacking.

In order to further improve the expressiveness of the generator, the same network is extended to contain ten residual blocks with layers of sizes 128, 32 and 128. An image generated by this network is shown in Figure 5.18b, and it has a higher level of detail compared to the smaller network. The increased expressiveness of the network manifests itself especially in the areas of the eyes, nose and lips. The low resolution of the training data naturally sets an upper bound on the amount of detail the generator will be able to produce.

The final image in Figure 5.18c is generated by a network trained on a 64x64 version of the CelebA dataset in order to increase the level of detail available for the generator to learn. Besides the dimensions of the dataset, and limiting the number of residual blocks to seven, the network is the same as the one responsible for the generation of Figure 5.18b. There is an increased amount of detail in the generated image compared to the two networks trained on smaller versions of the dataset. This is especially apparent in the eyes, where this network is actually able to generate visible pupils, and in general, the details of the facial features are finer. There is, however, a higher level of visual artifacts in this result. These artifacts are likely caused by the distance term and/or the activation functions used, and they might be less pronounced after more training.

The categorical variables for all of the networks seem to correspond to distinct visual looks based on a mixture of visual features, e.g. categories such as “woman with long hair”, “person with short hair” and “person smiling”, although some categories are harder to properly define than others. The two continuous variables, on the other hand, consistently correspond to the simultaneous rotation of faces and changing of color. A common occurrence is that one of the variables rotates the face inwards, while the outer rotates the face outwards. As such, the two continuous variables mostly control the same visual property, but at different numerical values. The results from varying the latent codes of the network in Figure 5.18c can be found in Appendix 5.

CIFAR-10

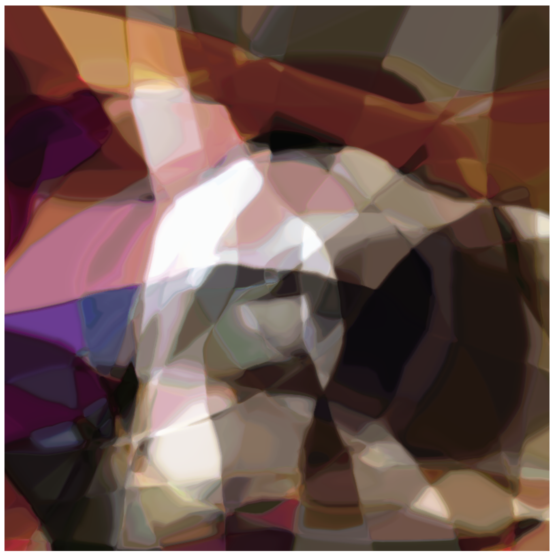
The CIFAR-10 dataset is perhaps the most complex dataset used in any of the experiments due to the great diversity of objects and shapes. The dataset spans a wide range of diverse objects such as airplanes, cars, dogs, frogs and horses, to name a few. The CelebA dataset, on the other hand, mainly contains images of human faces. As such, the majority of the experiments in this section will revolve around deep, residual architectures that might be more accommodating to a dataset of higher complexity.

The results from training an eight-layered CPPN on the CIFAR-10 dataset is shown in Figure 5.19. Figure 5.19a and Figure 5.19b are generated at different stages of the training, and they exhibit different visual properties. The overall level of activation in the network, as well as the effects of the distance term, seem to be decrease as training progresses. While the network fails to properly capture the distribution underlying the dataset, it has been able to approximate the distribution of colors and various shapes, as shown in Figure 5.19c.

In an effort to increase the expressiveness of the CPPN, a deep network consisting of five residual blocks is constructed, with each block containing three layers of size 128, 32, 128 with activation functions tanh, tanh and LReLU, respectively. All the weights in the network are initialized to small values. Images generated by this network is shown in Figure 5.20. The network does reach a certain level of convergence, as evidenced by the quality of the image grid in Figure 5.20a. The network is able to generate samples that approximate examples from the original dataset to a certain degree, as indicated by the numerous images that depict various shapes of cars, trucks and four-legged creatures. The approximation to the training data is not perfect by any means, but the overall distribution of colors, objects and shapes implies that parts of the distribution have been learned. The categorical variable of the latent code corresponds to different object classes, e.g. cars, trucks and horses in this case, but some of the categories are harder to make out than others. The two continuous variables both seem to have a large effect on the color of the images, as well as various transformations of the main object in a given image.

In a further attempt to improve the visual quality of the generated images, experiments where the auxiliary net is removed from the model have also been conducted. As such, the generator will no longer have its parameters adjusted with the objective of maximizing the mutual information. The latent code will, for convenience sake, still be input to the network, but as the generator is not being optimized to associcate the latent code with anything in particular, it should effectively serve as an extension of the latent noise vector $\mathbf{z}$. In other words, even though the latent code is still input, it should not have any discernible effect on the generated images. Moreover, the distance term is removed as an input in case it might restrict the CPPN's ability to approximate the diverse set of shapes found in the dataset. The results shown in Figure 5.21 are generated by the same network as in Figure 5.20, with the exception of the layer sizes of each residual block changed to 128, 64, 128, and the exclusion of the distance term and auxiliary net. The network is able to generate a wide variety of images, as evidenced in the grid of random samples, and many of the objects resemble cars, airplanes and various animals. Some of the generated images do not resemble anything in particular, especially at high resolutions, as shown in Figure 5.21c. The removal of the distance term does not seem to have a large effect on the generated images, and the results are quite similar to the ones presented in Figure 5.20.

(a) An image generated at training step ~ 20 000


(b) An image generated at training step ~ 250 000

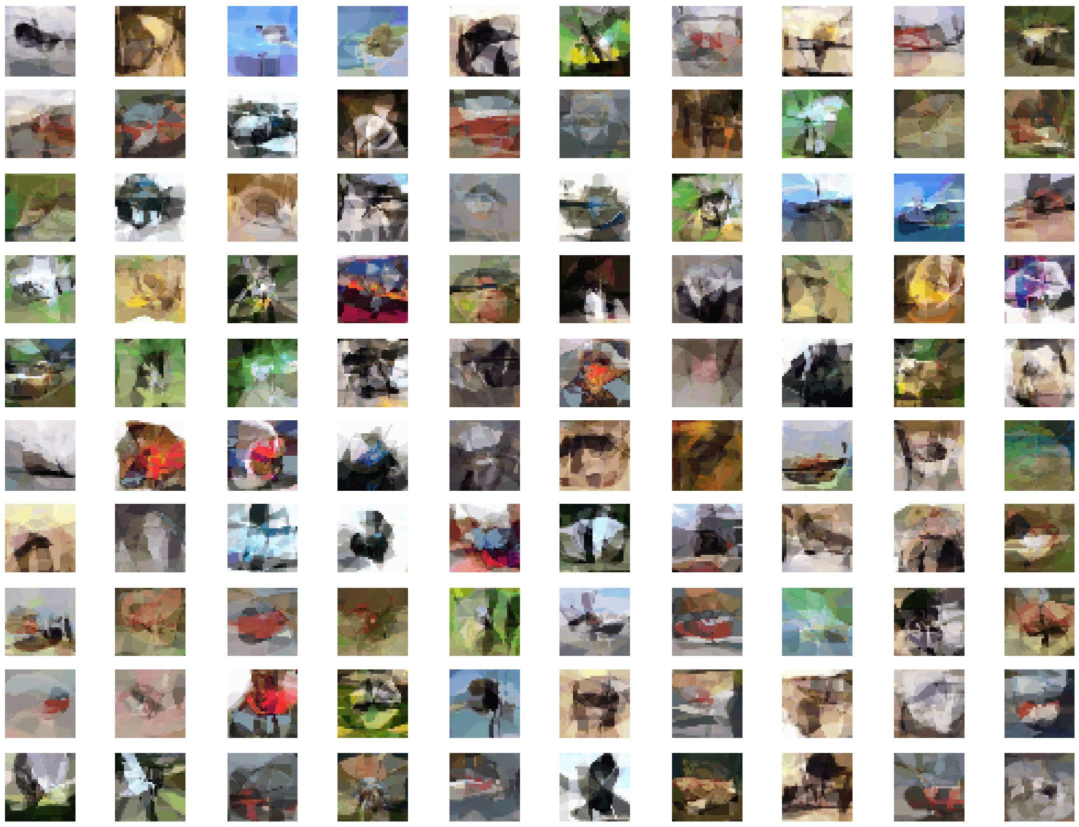
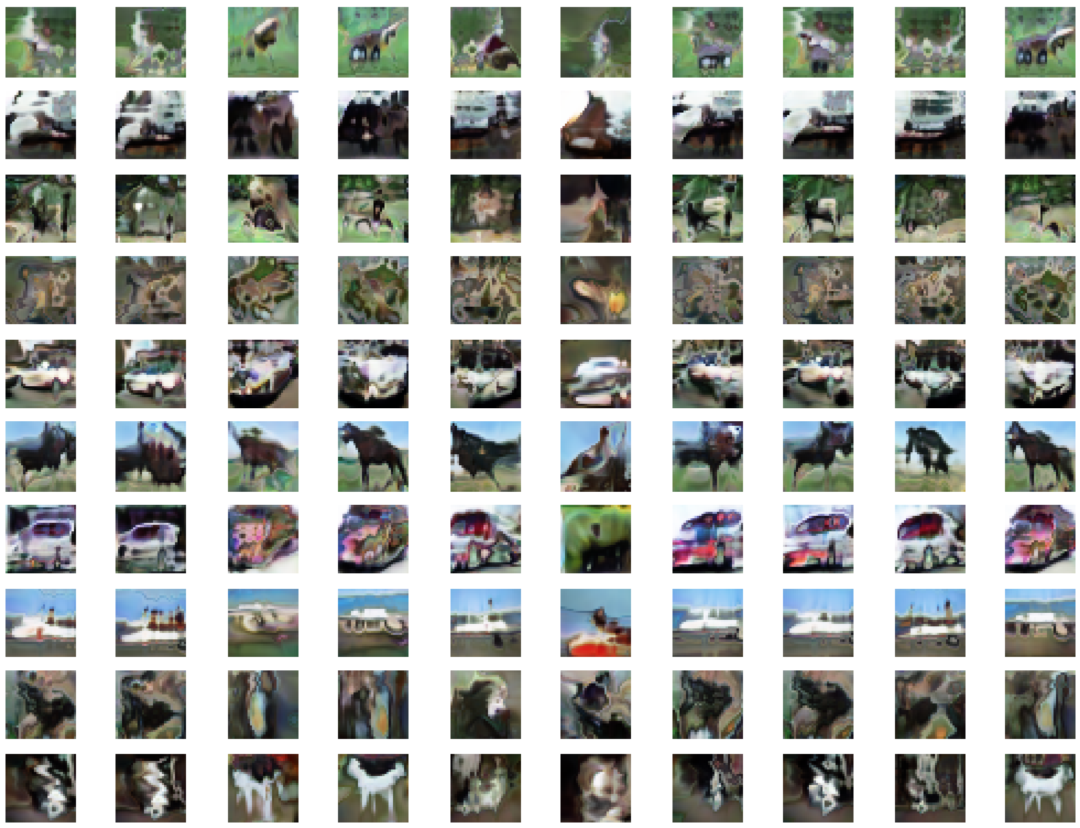
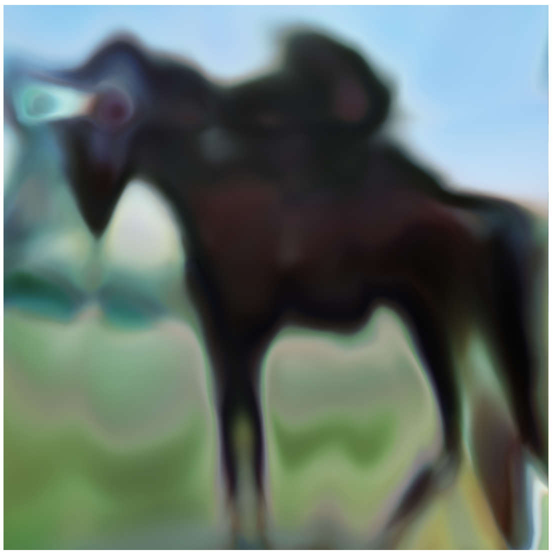

(c) A grid of samples generated at training step ~ 250 000

Figure 5.19: The CPPN in the IMPGAN uses eight layers of LReLU and remaining tanh.


(a) A grid of samples; each row with a different value of the categorical variable


(b) A high-resolution sample

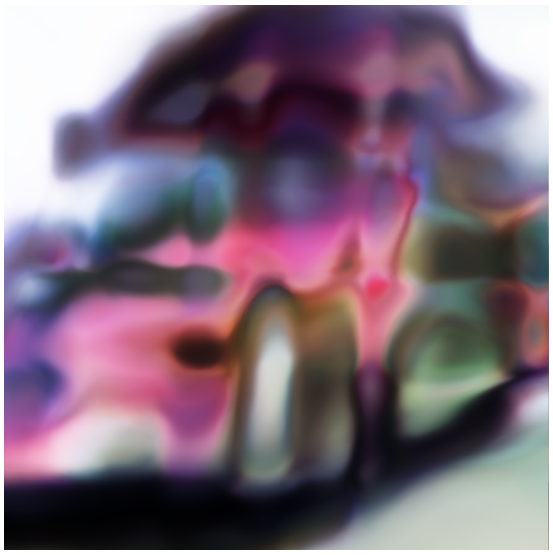
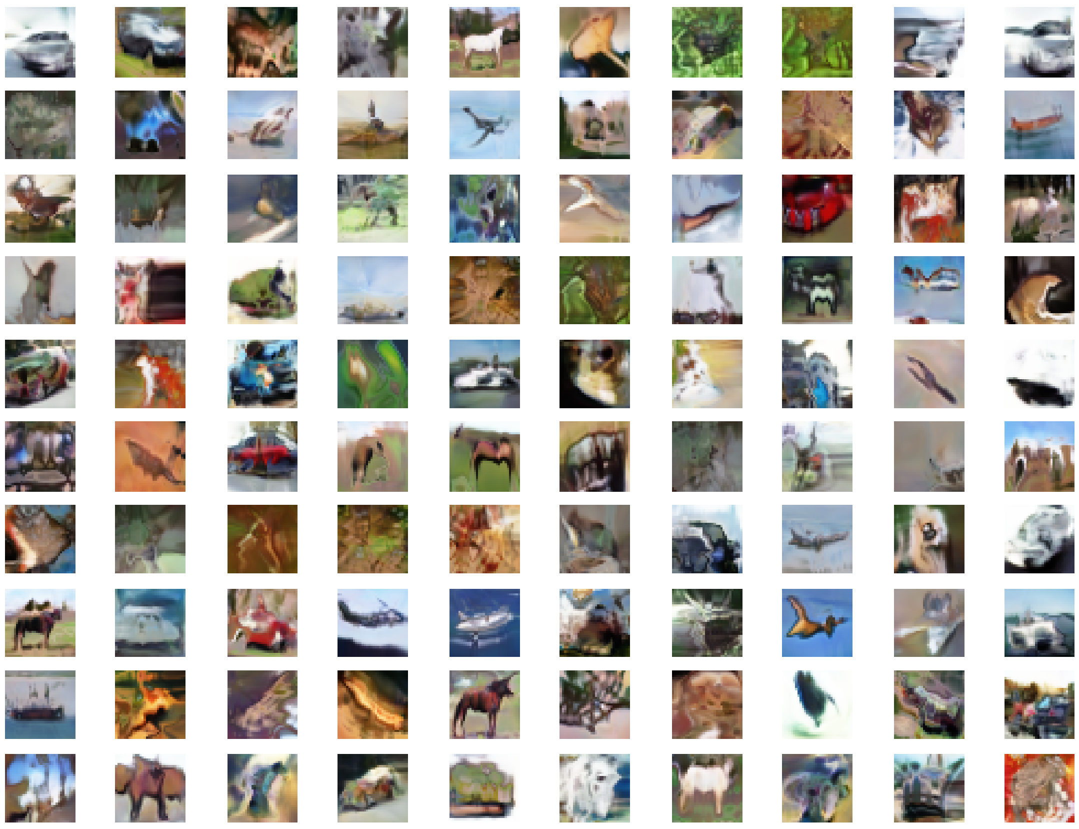
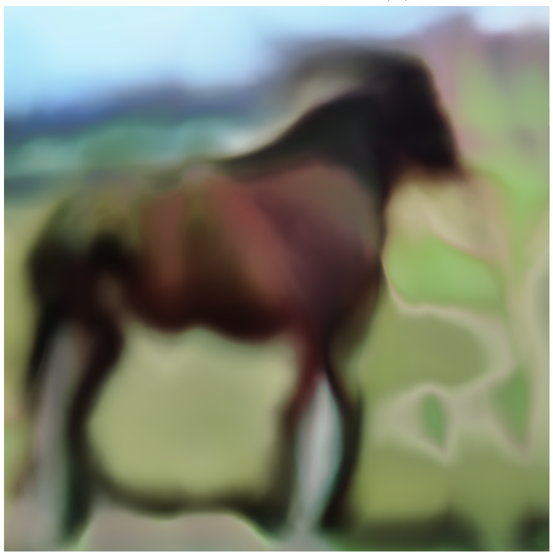

(c) A high-resolution sample

Figure 5.20: IMPGAN trained on the CIFAR10 dataset for $\sim 160\,000$ steps


(a) A grid of random samples


(b) A high-resolution sample

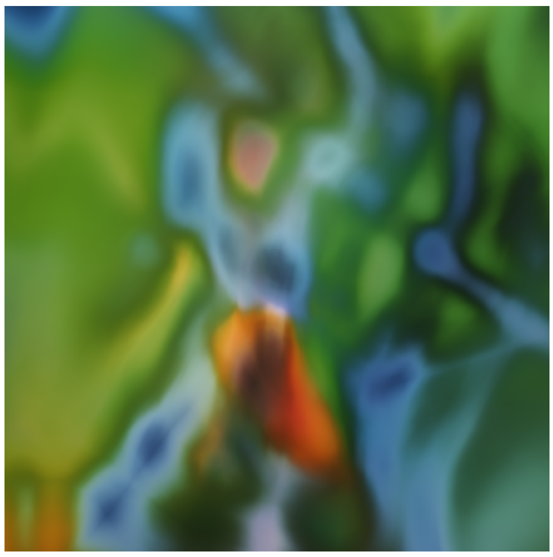

(c) A high-resolution sample

Figure 5.21: IMPGAN trained on the CIFAR10 dataset for $\sim 250\,000$ steps

6 Evaluation and Discussion

In this chapter the results from Chapter 5 will be evaluated and summarized in a broad manner, with a focus on highlighting similarities, differences and possible explanations for them. Additionally, the work done and results obtained will be discussed in the light of the thesis goal and research questions, as well as in the context of potential merits, key limitations and related work.

6.1 Evaluation

This section contains an overall evaluation of the key results from Chapter 5. This includes an evaluation of the visual quality of the generated images, as well as the potential merits of IMPGANs in terms of computational creativity and efficiency. Finally an overall evaluation of how various aspects of both stand-alone CPPNs and CPPNs in IMPGANs can be configured to have an effect on the output images will be presented.

The evaluation will focus mainly on the results from the experiments on IMPGANs, but other results may be brought in to highlight and/or discuss certain aspects. While preceding experiments revolving stand-alone CPPNs and traditional GANs have had a great deal of influence in establishing a foundation for the experiments with IMPGANs, they have mainly served as stepping stones to these more complex experiments.

6.1.1 Visual Quality

There are many different metrics that are commonly used to evaluate the visual quality of the generated images of a GAN. Two metrics often found in newer literature are the Fréchet Inception Distance (FID) (Heusel et al., 2017) and Inception Score (IS) (Salimans et al., 2016). There is, however, some overhead associated with using such metrics in that they require a separate, large and pre-trained classification network as part of their calculations. While IS is concerned with the general realism of the images, the FID metric is additionally taking into account the realism of an image in relation to the original dataset. In any case, the overarching goal of the experiments on IMPGANs has not been to achieve state-of-the-art visual quality at the original dataset resolution, but instead to strike a balance between relatively good visual quality at the original resolution, and distinct and novel visual properties, perhaps with a surprising degree of realism, at high resolution. Because of this, the traditional GAN evaluation metrics have not been utilized. What follows is an evaluation of the visual quality of images generated by IMPGANs at low and high resolutions.

Low-Resolution Images

The visual quality at low resolution can be evaluated in terms of how images generated by an IMPGAN compares to images generated by traditional GANs. As such, the evaluation of visual quality is done mainly by visual inspection.

Across a wide variety of datasets the visual quality of the low-resolution images generated by IMPGANs seem to be quite similar to the results obtained from training similarly configured traditional GANs. It should be noted that in some cases a CPPN might generate better images than the a similar MLP or CNN for the same amount of training steps. This can probably be attributed to the fact that during each training step, the CPPN is activated and computes gradients $BatchSize \cdot ImgDim^2$ times, while a traditional generator does the same only $BatchSize$ times. The training of a CPPN can then be considered as processing a batch within a batch — as also noted in previous, related experiments (Ha, 2016a). For some of the simpler datasets such as the MNIST dataset, this behavior does in fact seem to arise, where some of the IMPGANs can be argued to generate higher visual quality images compared to GANs using a CNN in the generator, especially in terms of the results where the latent code is varied (see Figure 5.15 versus Figure 5).

Overall, it can be argued that the IMPGANs are able to generate low-resolution images that are similar to the training data, and that exhibits a visual quality comparable to that obtained from similarly configured GANs.

High-Resolution Images

The visual quality at high resolution is harder to evaluate as the training data of all the IMPGANs consists of low-resolution images. Mainstream GAN models capable of generating images at high resolutions are trained on datasets containing images of resolutions hundreds of times greater than what the IMPGANs have been trained on, and these models are naturally capable of learning a much higher level of detail. As such, it does not make much sense to compare the CPPN-generated images to the images generated by these models — models such as StyleGAN (Karras and Aila, 2018) are naturally capable of generating images of much higher quality and with a very impressive level of detail.

The most realistic results from the experiments have arguably been obtained by training the IMPGAN on the CelebA dataset. The author has not been able to find any related work that has trained a CPPN on this dataset. The lack of any previous work with CPPN-generated images exhibiting similar levels of realism — regardless of dataset — seems to indicate an improved state-of-the-art in terms of realism and detail in the high-resolution images generated by CPPNs trained using GANs.

Moreover, a high-resolution video depicting the interpolation between different latent vectors using the model in Figure 5.18 was shared online[1], where Kenneth Stanley, the creator of the CPPN, NEAT and other evolutionary architectures and algorithms,

[1]`https://twitter.com/erlendekern/status/1154002479769456640`

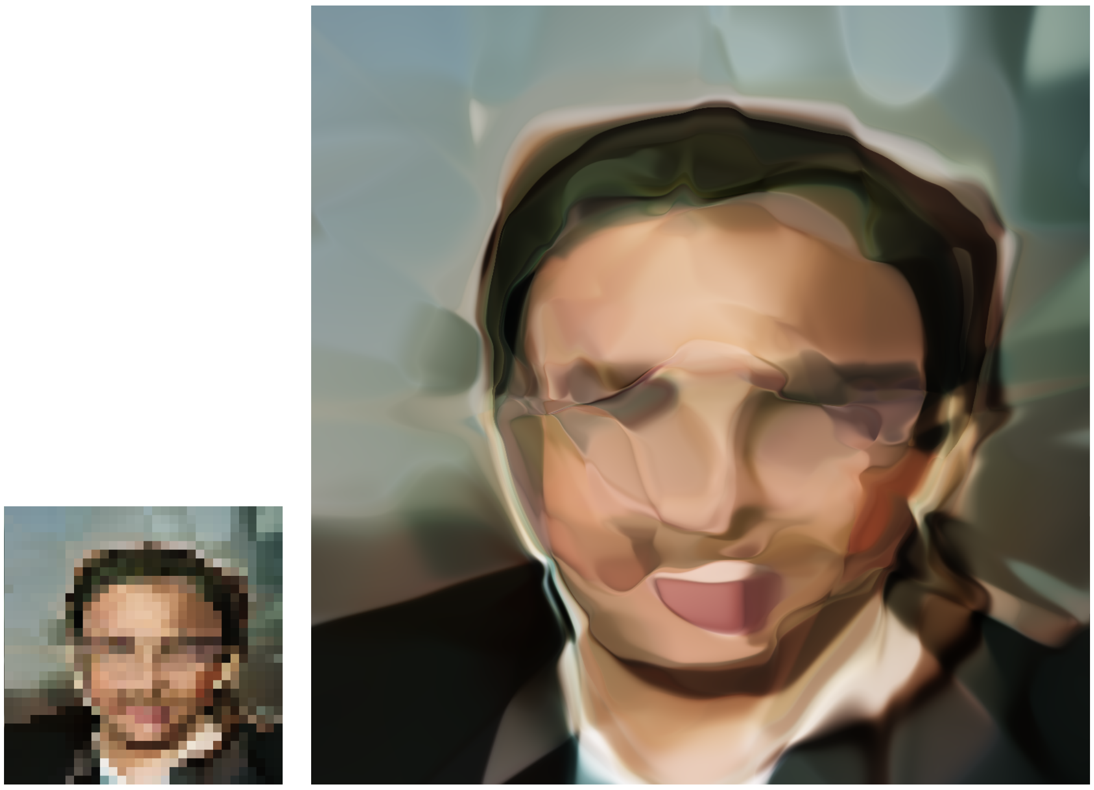
commented on the results, noting that he had not previously seen CPPNs trained with gradient descent generate images at such a level of realism.

As the quality of the high-resolution images can be somewhat hard to properly evaluate, it might be interesting to discuss their potential merits in the context of computational creativity (Section 6.1.2) as well.

6.1.2 Computational Creativity

Creativity is the ability to generate something that can be considered novel, surprising and valuable (Boden, 2004). This includes artifacts such as an idea, a well-crafted joke, artworks, music and so on. The novelty can be described in terms of the extent to which an artifact can be considered new and novel from the perspective of the creator — that is, the creator does not have any prior knowledge of similar artifacts. Boden further makes a distinction between different notions of surprise, two of these pertain to unexpected ideas that fit into an existing way of thinking, and ideas that feel impossible once encountered, respectively (Boden, 2004, p. 3). The notion of value is harder to properly define in a general way, but it has been suggested that value can be defined in terms of who the artifact should be valuable to (Lamb et al., 2018). With this in mind, a perhaps fitting description of value in the context of synthetic images is the visual quality compared to a set of target examples (training data). The value of the generated images can then be described in terms of their similarity to some reference data. Without evaluating the value, it might be argued that a trivial program that generates random images by simply sampling random values in an interval $[0, 255]$, arranging the results in a grid and saving it as an image is creative. By including the aforementioned definition of value, however, the generated images must additionally be vastly different from results generated by a purely random process.

The possible creative merits found in a CPPN trained using an IMPGAN can be argued using these three concepts as evaluative measures. The entire generating system will be evaluated as a whole by taking both the generating network and the final output images into account. As the network is only being trained on low-resolution images, the results obtained when generating images at very high resolutions are the product of the CPPN attempting to bridge the gap between the values it knows, i.e. a given coordinate system, and values far outside what it is familiar with it. This represents the novelty of the system — the network generates images that are of a different nature to compared to anything it has both seen and generated before. Moreover, it can be argued that the networks can elicit various forms of surprise. Image that a potential audience is presented with a low-resolution image that is very similar to a face, as in Figure 6.1a. A question could then be posed, asking the audience to predict how the image would look were they to make it hundreds of times greater, i.e. how would they imagine "filling in the blanks". and subsequently be shown the exact same image at a much larger resolution (Figure 6.1b) with different visual properties — the main essence of the image remaining constant. It is still an image of a face, but with visual properties the viewer would likely not have anticipated based on the small image. The value represented by the system can be argued based on its ability to generate low-resolution images that are similar to the



(a) Generated at 64x64 (b) Generated at 1200x1200

Figure 6.1: An image generated by an IMPGAN at different resolutions

training data.

6.1.3 Computational Efficiency

The computational efficiency here refers to how training a CPPN can pose less computational requirements, i.e. training time and memory usage, compared to other GANs in terms of the ability to generate high-resolution images.

Training two identically configured GANs, i.e. identical discriminators, learning rates, dataset, etc., with the only exception being one generator is a CPPN and the other an MLP, naturally leads to less training time for the more traditional GAN for a fixed number of training steps. The reason for this, as also mentioned in Section 6.1.1, is that at each training step the traditional generator would calculate the gradients N times, while the CPPN would need to do the same $N \cdot ImgDim^2$ times.

Despite this, the nature of the CPPN can make it compelling to compare different models based on the requirements they necessitate in order to train a model to generate a specific resolution. Training the model in Figure 5.18c for 250 000 training steps takes approximately 4 days on a Nvidia Tesla P100 GPU, while a state-of-the-art StyleGAN would need to be trained for around 41 days on similar hardware (Nvidia Tesla V100) to generate 1024x1024 images[2]. Admittedly, the visual quality of and level of detail in the images generated by the StyleGAN is much higher, but the training of such models do in any case necessitate more training time and training data of higher resolution compared to a CPPN. This demonstrates how a CPPN can be much more computationally efficient than other methods found in the literature in terms of generating high-resolution images. The CPPN can be trained on the CelebA dataset downsampled to 64x64, and subsequently generate images at 2000x2000. The obvious catch here is that there is a large difference in the amount of detail in 64x64 images compared to 1024x1024 images that models like StyleGAN are trained on.

Moreover, in an effort to speed up the training process of CPPNs, the network is configured to generate the values for all pixels in parallel as opposed to one by one. If memory constraints are an issue, however, the network can easily be modified to instead only generate one or a few pixels at a time. This means that the CPPN is able to generate large images on restricted hardware where other methods for high-resolution synthesis, like the StyleGAN, might be too computationally expensive.

6.1.4 Activation Functions

A diverse set of activation functions has been explored, and they have all, in some way, contributed to distinct visual effects in the generated images. It is apparent that using unconventional activation functions such as sin can have a detrimental effect on the convergence of a given network, especially if the layer in question is coupled with large initial weights. This serves to highlight how some functions are not well-suited to perform in an optimization process. It can still be possible to reach convergence using such functions on simpler datasets like the MNIST dataset, but due to the fact that they in some cases can hinder convergence, the majority of the experiments pertaining to the training of CPPNs utilize more conventional activation functions such as LReLU and tanh in an effort to increase the training stability. These two functions both contribute to distinct visual properties in the generated images, while also performing well in an optimization context.

By examining the plot of a given function, the visual effects that it will be responsible for in a CPPN-generated image can to a high degree be predicted a priori. If the function can be characterized by smooth curves, it will likely contribute to curves and circular shapes in the output, and conversely more linear plots will often result in sharper lines. The LReLU leads to a bias towards sharp and straight lines in the high resolution images, while tanh functions give rise to curved and smooth lines. The periodic nature of the sin function, on the other hand, leads to repeating patterns.

Increasing the depth of a CPPN combines the visual properties of each layer in a complex way, where stacking multiple tanh layers leads to an output with overlapping, curved lines in a sort of “bubble” pattern. Similarly, using different functions in different layers leads to an output that exhibits a combination of the visual features that can be attributed to each different function. The strength of these features can to some extent

[2]Taken from `https://github.com/NVlabs/stylegan`.

be controlled by initializing a given layer to smaller or larger weights to decrease or increase the effects, respectively.

6.1.5 Distance Term

The degree to which the distance term affects the generated images seems to vary based on the dataset used. The CelebA dataset, for instance, is mainly concerned with faces, and since faces exhibit a certain degree of symmetry, the distance term might be of use in providing symmetry to the generated images. The CIFAR-10 dataset, however, contains a wide range of shapes and objects, and there is no shared symmetry between most images in this dataset. In this case the distance term actually hinders convergence, as the CPPN is biased towards a symmetry that might not actually exist in the dataset. In theory, the network could learn to simply ignore the distance term if it is of no use. It seems plausible, however, that the networks utilize the included distance term in the early stages of training when the fidelity of the generated images is low. Since the learning rate used for the generator is relatively small, it might then be difficult to reverse this dependency as training progresses and finer details and shapes are to be generated.

The combination of including the distance term and using large initial weights seems to make the effects of the distance term more pronounced, and a distinct pattern of circular radiation is often visible in the center of the generated images.

It is possible to use a smaller scaling factor for the distance term compared to the remaining input coordinates $\mathbf{x}$ and $\mathbf{y}$ in an effort to lessen the initial influence it has over the generated images. The network would then have access to the distance term if needed, but in the early stages of training the output would be biased towards prioritizing the coordinate inputs $\mathbf{x}$ and $\mathbf{y}$ as they would contain relatively larger values.

6.1.6 Scaling Factor

The scaling factor s lets you zoom in on and out of an image by decreasing and increasing its value, respectively. It scales all of the input coordinates, i.e. $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{r}$, if included, to lie in a specific interval. All experiments involving training a CPPN have used a fixed scaling factor that has empirically worked. Both the latent vector $\mathbf{z}$ and code $\mathbf{c}$ lie in the interval $[-1, 1]$, and setting a relatively high scaling factor should result in the coordinates having more influence over the output compared to the latent inputs. Setting it to a very high value like $s = 50$, however, might end up destabilizing the training process, as the inputs are of vastly different sizes. Setting it to a small value, e.g. $s = 1$, seems to lead to slower convergence when combined with large initial weights.

After training, the scaling factor can freely be adjusted to control the zoom level of the generated images, and it can provide insight into the representations the network have learned. As an example, by using a large scaling factor and thus zooming out, one can visualize how the network generalizes when being tasked with generating images in coordinate systems far larger than the ones it was trained on.

The effects of using different scaling factors have mainly been studied in the context of untrained, randomly initialized CPPNs or generating images using an already trained

IMPGAN. While some experiments have been conducted on how different scaling factors affect the training of IMPGANs and how they might lead to different visual properties (see Appendix 1), these effects have not been studied in-depth.

6.1.7 Latent Code

Most of the experiments result in a generator that is able to achieve a certain level of disentangled representations in the latent code **c**. A latent code of one categorical and two continuous variables is utilized in the majority of the networks, and changing the values of these variables usually results in meaningful changes in the generated output. The different values of the categorical variable tend to correspond to specific categories. A category, in this sense, often represents a combination of visual features that can be considered unique compared to other combinations. In general, the categorical variable can be associated with visual properties that are more discontinuous in nature. In the case of the MNIST dataset, a category is a single digit class, and similarly for the Fashion MNIST dataset where each category usually corresponds to a unique fashion object.

The continuous variables, on the other hand, usually correspond to properties of a more continuous nature, such as rotation, size and color. The experiments show how two continuous variables often are the inverse of one another: if the first continuous variable rotates an object outwards, the second variable might rotate the object inwards. The variables are then effectively controlling the same visual effect, but at different values. In practice, however, there is often some additional property associated with these pair of inverses such that each variable is in fact able to control a unique set of features.

Optimizing the generator to maximize the mutual information between the latent code and the generated images can, however, have a large negative effect on the convergence and stability of the training process. The reason for this is likely that the generator's parameters are adjusted both in terms of generating realistic images and maximizing the mutual information. Optimizing for both of these objectives might at a given training step result in contradictory parameter adjustments, leading to a tug of war of sorts between the optimizers. This behavior has been apparent especially when utilizing more than one categorical variable, and the training might prove so unstable that convergence is impossible. Adding more continuous variables does not usually have the same detrimental effect, and it seems that it is easier for the auxiliary net to find (and for the generator to generate) continuous visual features compared to features of a more discontinuous nature. Moreover, even when only using one categorical and two continuous variables, throughout the training there is usually one or more sudden large increases in the loss function, but this is usually corrected quickly.

Additionally, there might not be enough inherent categories in a given dataset for the auxiliary net to sort them into meaningful categories. Similarly, the auxiliary net might not be expressive enough to be able to reconstruct large latent codes. Such a situation might be improved by making the auxiliary net a stand-alone network, as opposed to reusing most of the discriminator layers, or by making the discriminator, and thus implicitly the auxiliary net, larger.

Some of the training instability related to the latent code might be improved by using a lower learning rate for the auxiliary net compared to the generator.

6.1.8 Initial Weights

Most of the experiments pertaining to the sizes of the initial weights of CPPNs have very similar results despite revolving around a wide variety of network architectures and parameters. In the case of merely initializing a CPPN with large weights, and generating images from such an untrained network, the results are more dramatic and exhibit a sharper look as a result of the higher number and stronger level of activation in the network. Similarly, when training a CPPN with large initial weights, the patterns and shapes unique to the activation functions of the CPPN are more apparent at the early stages of training, and then gradually lessen their effect on the generated images as the training converges and the generator is more closely approximating the distribution of the dataset. In some cases, e.g. when using sin functions, large initial weights can hinder the convergence of a network altogether. A network initialized to small weights exhibits a more stable training phase that converges quicker, but generated high-resolution images usually have a relatively soft and smooth appearance. The weight initialization scheme used in a CPPN then reflects a trade-off between a high level of convergence and pronounced visual properties at high resolution. The images generated at the original, low resolution of the training data are usually quite similar regardless of the weight initialization scheme used in the networks.

For layers using the tanh activation function, large initial weights will likely lead to many saturated units in the layer. Most of the outputs will then be close to the minimum (-1) and maximum (1) of the tanh function. In an optimization context these saturated units can lead to vanishing gradients, which can impair the learning process. LReLU activation functions, on the other hand, do not lead to saturated units when used together with large initial weights, but the outputs can become very large in size[3], and might as such lead to saturation of downstream tanh layers. The different behaviors of LReLU and tanh layers when using small and large initial weights are shown in Appendix 3.

6.1.9 Architectures

In general, increasing the depth of a given CPPN gives rise to more complex output images as the value of the final output is dependent on a higher number of functions. When using an MLP, some of the deeper units might become saturated depending on the nature of the activation functions used. Saturated units should in turn lead to sharper and more complex images, as the output pixel values become more extreme. Moreover, a residual architecture seems to give rise to some more complex patterns, as each block can adjust its weights to let the intermediate layers have a smaller or larger effect on the output of the block.

[3]The nature of the LReLU function leads to larger positive values than negative, as a positive input is equal to the input itself, while negative inputs are multiplied by a small constant $\alpha = 0.2$.

It is nonetheless interesting that training similarly configured IMPGANs on different datasets can net very different results: training a CPPN with residual blocks and small weights on the CelebA dataset results in sharper images compared to training a similar network on the CIFAR-10 dataset. The reason for this is not clear, but it might be because the CIFAR-10 dataset contains a very wide range of objects, colors and shapes, and that the network output for any given pixel is pushed towards softer, more ambiguous values that than can better accommodate the higher diversity of the dataset. Each pixel might, in a sense, become more "general" in nature to be able to represent a wider range of potential shapes, objects and colors. This might explain the softer appearance of the images generated by residual networks trained on CIFAR-10.

6.2 Discussion

This section will contain a discussion of the work presented, and highlight the key findings, potential and limitations, as well the influential related work. Following this is a discussion of how well the overarching thesis goal has been met, and to which extent the research questions have been answered.

6.2.1 Potential, Limitations and Influential, Related Work

The work presented has demonstrated how the foundational properties of CPPNs affect the generated images, as well as how these networks can be trained to achieve a relatively high degree of realism on certain datasets compared to previous works on CPPNs. The findings presented has a lot of potential for future exploration. In short, the IMPGAN architecture can be trained on low-resolution, unlabeled datasets, and subsequently generate images that at the original resolution are similar to the training data, while at high resolution exhibit different visual properties and/or unexpected level of detail and realism. This sidesteps many of the requirements recent GAN methods have in terms of computational cost and high-resolution datasets. Moreover, instead of using the labels, if any, of a given dataset to guide the training, the IMPGAN instead aims to map a part of the latent input to controllable, isolated features in the output images. These disentangled representations can have a lot of potential for possible interactive systems using IMPGANs, where users can choose what kind of features they would like to see in the generated images. Finally, it can be argued that the work has merits within the field of computational creativity and efficiency.

Despite promising results, there are also many limitations associated with the work presented. Chiefly, the sheer number of possible combinations of various architectures, parameters and techniques is a natural limitation: it has only been feasible to focus on a subset of these combinations, and it can not be ruled out that other configurations can net better results. Other limitations are related to the fact that the field of GANs is still maturing, and solid theoretical foundations for various techniques, architectures and parameters are still somewhat lacking. Currently, a lot of the common GAN configurations are based on what has worked well for others in the past, but recent large-scale studies

(Kurach et al., 2019) can help shed light on which techniques actually have a verified, positive effect on training stabilization, and which are configurations the community has simply converged to without much solid reasoning.

Previous related work has been responsible for introducing a lot of insights in terms of various techniques and highlighting their properties and limitations. The work in this thesis has used such findings as a base to derive inspiration and further explore from. Especially the work on CPPN-GAN architectures by Ha (2016a; 2016b; 2016c) has been quite influential, as the author performed, in the context of CPPN-GANs, many pioneering experiments, shared and discussed key findings and limitations, and additionally included the source code for all relevant experiments.

Moreover, many existing reference implementations have been used as inspiration when developing the architecture, especially specific details concerning the proper implementations of various objective functions for GANs such as the Wasserstein distance (WGAN) (Arjovsky et al., 2017) and Wasserstein distance with gradient penalty (WGAN-GP) (Gulrajani et al., 2017).

6.2.2 Thesis Goal and Research Questions

The goal and research questions presented in Chapter 1 will now be repeated, and the extent to which the overarching goal has been reached and the research questions have been answered will be discussed and reflected upon. The research questions will either contain direct answers or, in an effort to increase readability, references to relevant sections where the answers can be found.

Research question 1 *How does the performance of an Information Maximizing and Pattern-Producing Generative Adversarial Network compare to its conventional counterpart?*
A wide variety of experiments have been conducted where IMPGANs and conventional GANs of similar architecture have been trained on the same datasets. The IMPGAN usually takes longer to train in terms of wall-clock time, but achieves similar visual quality to MLP and CNN generators when generating samples at low resolution. Moreover, in some cases the IMPGAN seems to produce better quality samples then conventional GANs, and the latent codes of an IMPGAN have been seen to correspond to more pronounced and discernable visual changes.

Research question 2 *How do different architectures, datasets and parameters affect images generated by an Information Maximizing and Pattern-Producing Generative Adversarial Network?*
Experiments pertaining to how all of these configurational aspects affect the images generated by an IMPGAN have been carried out, evaluated and discussed. Moreover, the accompanying results have not been evaluated in a vacuum, but instead in a holistic manner where similarities and differences between them, as well as key findings from other experiments have been utilized to support the arguments given.

In an effort to establish solid foundations that more complex experiments can build on, the entire experimental phase has revolved around establishing theoretical and technical foundations in a bottom-up manner. This approach has also been motivated by conducting the initial experiments on simple models and simple datasets as a way to remove possible disturbances that might arise in more complex experiments and test the effects of different configurations in a more isolated environment. These isolated findings, e.g. the effect of the distance term on untrained CPPNs, have been used to guide subsequent experiments and gain a better understanding of their results.

In short, the effects activation functions, different datasets, the scaling factor, distance term, latent code, size of initial weights, and architectures have on the generated images of an IMPGAN have all been established in Section 6.1.

Research question 3 *How can the level of detail and realism be increased for the generated images of an Information Maximimizing and Pattern-Producing Generative Adversarial Network?*
The IMPGAN responsible for the most realistic and detailed images succeeded in doing so for a couple of different reasons. First, the dataset that was used, the CelebA dataset, contains images of faces, and the dataset does not span many vastly different image classes. There are images of different genders, ethnicities, ages, different facial expressions, with various clothing accessories, etc., but the overall characteristics of a face is still relatively similar across these samples. Contrast this to the wide variety of shapes and objects found in the CIFAR-10 dataset.

Second, the CPPN is initialized to small weights, which should improve the optimization process relative to large initial weights.

Third, the network is constructed as a deep neural network using residual blocks. As such, the network has the possibility to adjust its parameters to utilize more or less of the intermediate layers of each block as it sees fit. Simply using a deep neural network by stacking fully-connected layers would force the network utilize any given layer. Additionally, the intermediate layers use a tanh activation function while the final layer in each block uses LReLU, which should allow the network to generate a wide variety of shapes — that is, curved and smooth, and linear patterns, as well as a combination of these.

Fourth, the distance term is included as an input to the network in an effort to bias the network towards generating symmetric patterns (as faces can be considered relatively symmetric).

Overall, the experiments utilizing the combination of these four configurational aspects led to the most detailed and realistic CPPN-generated images.

Thesis goal *Explore the usage of an unconventional generator architecture in a Generative Adversarial Network in the context of computational creativity and efficiency*
The overarching goal of this thesis has focused on training CPPNs using the

adversarial framework offered by GANs. As related work had demonstrated that CPPNs were able to generate images at arbitrary resolutions with perhaps novel and surprising visual results, it was a natural decision to frame the goal in the context of computational creativity and efficiency. This was further motivated by the fact that a lot of the research on GANs seemed to converge towards larger models and datasets, requiring more training time and better hardware, instead of exploring more novel approaches to image generation.

The goal can perhaps seem somewhat open-ended and ambigious, but this is intentional. The direction in which to focus the main efforts of the work was not known at the start, but has instead been influenced by the many experiments conducted and their accompanying results. The main idea was to start experimenting with a simple model trained on a simple dataset, and then continue exploration in whichever direction seemed promising based on these results. Moreover, the work has pertained to only a small subset of possible configurations in an effort to thoroughly document both the experiments and results.

Overall it can be argued that the goal has been reached: a great deal of experiments revolving around the usage of a CPPN as the generator in a GAN have been carried out, documented and evaluated. CPPNs have been trained on new datasets, with new configurations and with a focus on achieving disentangled representations — such experiments have, to the best of the author's knowledge, not previously been carried out by others. The work has been evaluated both in the context of computational creativity and as an efficient way to generate high-resolution images compared to the state-of-the-art. It must, however, be noted that there is a great deal of possible improvements, and alternative directions and configurations that are yet to explored.

7 Conclusion and Future Work

This thesis has shown that Compositional Pattern-Producing Networks (CPPN) (Stanley, 2007) can be trained using Generative Adversarial Networks (GAN) (Goodfellow et al., 2014) to approximate the underlying distributions of a wide variety of datasets. An Information Maximizing and Pattern-Producing GAN (IMPGAN) architecture has been constructed using state-of-the-art methods and techniques for training GANs, and it has demonstrated an ability to successfully train CPPNs of different configurations across a wide variety of datasets. The images generated by an IMPGAN have been shown to be of similar visual quality compared to results produced by more conventional generators. It has further been demonstrated how the IMPGANs are able to generate images not only similar to the original, and often low, resolution training data, but that it can be tasked with generating images at resolutions hundreds of times greater than what it has been trained on. The resulting high-resolution images have been shown to exhibit different, novel visual properties, and, in the context of CPPN-generated images, previously unseen levels of detail and realism.

Training a GAN to generate high-resolution images, e.g. 1024x1024, using state-of-the-art methods such as StyleGAN (Karras and Aila, 2018) require a lot of computational power, large models and access to high-resolution datasets. A CPPN, on the other hand, takes a novel approach to image generation that allows it to be trained on a low-resolution dataset and subsequently produce images at arbitrary resolutions. Naturally it seems unlikely that a CPPN trained on 32x32 images can create highly realistic 1000x1000 images due to the lack of detail in the training data, but for certain datasets and configurations the results can be quite interesting. Assuming that after a certain image resolution, e.g. 256x256, there are diminishing returns in terms of the amount of detail the generator in a GAN can learn to generate, a CPPN could then be trained at said resolution, and afterwards generate realistic images at much higher resolutions. This could be advantageous in settings where there is a lack of high-resolution datasets and/or excessive computational power, and the CPPN can be argued to have merits in the context of computational efficiency.

While it has been demonstrated that some configurations of CPPNs trained by an IMPGAN have failed to converge properly, the images generated by the networks are nonetheless based on an attempt to approximate the underlying distribution of the dataset. In this sense, the images are vastly different to purely random generated images, and the visual properties of the images can prove quite novel. In general, it can be argued that IMPGANs, together with the images they generate, exhibit certain characteristics that are computationally creative.

Overall, the work presented indicates that there is a great deal of potential for further exploration of this more novel approach to image generation, and some possible ideas and suggestions for future work are outlined in Section 7.1.

7.1 Future Work

In this section various future work will be outlined. This includes experiments that were not carried out due to time constraints, directions that have not been explored extensively, as well as some general ideas and suggestions.

7.1.1 Evaluation

The evaluation of visual quality in this work has mainly been based on visual inspection. It could be interesting to evaluate CPPN-generated images using common evaluation metrics such as the Fréchet Inception Distance (FID) and Inception Score (IS). This could be used to evaluate the images generated at both low and high resolutions. It would also allow for objective comparisons between the images generated between CPPNs and more traditional generators such as convolutional neural networks (CNN). Moreover, conducting surveys to evaluate the potential creativity in the CPPN-generated images could net interesting results.

7.1.2 Interactive System

A trained IMPGAN has several input parameters that can be tweaked to effect the output: chiefly the latent noise vector $\mathbf{z}$, with unpredictable output effects, and the latent code $\mathbf{c}$ with relatively predictable effects on the output images, as well as the zoom level and size of the generated images. These inputs could be interesting to further explore in an interactive setting.

An interactive system could be set up for generating images from a CPPN trained using an IMPGAN, where the user can control the latent inputs, as well as the scale and resolution of the generated images. This could be achieved by giving access to a trained model via an application programming interface (API), and constructing a web interface that communicates with this API. There exists various software that can help facilitate such access to TensorFlow models e.g. TensorFlow Serving.

7.1.3 Training on Different Resolutions and Datasets

The experiments conducted on CPPNs and GANs have revolved around approximating a single distribution at the fixed, original resolution of the training data and with a fixed scaling factor. Similar to the related work in Section 3.3, extensions can be made to the IMPGAN architecture to allow training the CPPN to approximate one dataset at a low resolution, and another at a higher resolution, or similar for the scaling factor. A CPPN can then be trained to approximate the CelebA dataset at a low resolution, and perhaps a dataset of abstract artworks at a higher resolution. The final images generated by the CPPN at high resolutions will then likely contain similarities to both of these datasets.

It could also be interesting to train the CPPN on one dataset at iteratively higher resolutions, similar to ProGAN (Karras et al., 2017) and StyleGAN. This might stabilize the training and net better overall visual results.

7.1.4 Various Configurations

Most of the experiments in this work have not utilized any normalization scheme, e.g. spectral normalization, batch normalization or layer normalization, in the CPPN, and given the positive effects such schemes have had on traditional generators, utilizing such schemes might net better results when training a CPPN as well.

Moreover, experimenting with transposed convolutional layers in the CPPN could also be of interest. The experiments in this thesis have mainly been concerned with residual blocks and fully-connected layers.

Overall, there are many different configurations that have yet to be tested extensively. This includes loss functions, learning rates, optimizer parameters, different batch sizes, auxiliary net and discriminator architectures, and normalization schemes, to name a few.

Bibliography

Aphex34. Typical cnn architecture, Dec 2015. URL `https://commons.wikimedia.org/wiki/File:Typical_cnn.png`.

Martin Arjovsky and Léon Bottou. Towards principled methods for training generative adversarial networks. *arXiv preprint arXiv:1701.04862*, 2017.

Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Margaret A Boden. *The creative mind: Myths and mechanisms.* Routledge, 2004.

Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale gan training for high fidelity natural image synthesis. *arXiv preprint arXiv:1809.11096*, 2018.

Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In *Advances in neural information processing systems*, pages 2172–2180, 2016.

Thomas M Cover and Joy A Thomas. *Elements of Information Theory, Second Edition.* John Wiley & Sons, 2006.

Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

William Fedus, Mihaela Rosca, Balaji Lakshminarayanan, Andrew M Dai, Shakir Mohamed, and Ian Goodfellow. Many paths to equilibrium: Gans do not need to decrease a divergence at every step. *arXiv preprint arXiv:1710.08446*, 2017.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

Glosser.ca. Artificial neural network with layer coloring, Feb 2013. URL `https://commons.wikimedia.org/wiki/File:Colored_neural_network.svg`.

Ian Goodfellow. Nips 2016 tutorial: Generative adversarial networks. *arXiv preprint arXiv:1701.00160*, 2016.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

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

Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. In *Advances in Neural Information Processing Systems*, pages 5767–5777, 2017.

Odd Erik Gundersen and Sigbjørn Kjensmo. State of the art: Reproducibility in artificial intelligence. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

David Ha. Neural network generative art in javascript, June 2015a. URL `http://blog.otoro.net/2015/06/19/neural-network-generative-art/`.

David Ha. Neurogram, July 2015b. URL `http://blog.otoro.net/2015/07/31/neurogram/`.

David Ha. Generating large images from latent vectors, April 2016a. URL `http://blog.otoro.net/2016/04/01/generating-large-images-from-latent-vectors/`.

David Ha. Generating large images from latent vectors - part two, June 2016b. URL `http://blog.otoro.net/2016/06/02/generating-large-images-from-latent-vectors-part-two/`.

David Ha. The frog of cifar 10, April 2016c. URL `http://blog.otoro.net/2016/04/06/the-frog-of-cifar-10/`.

David Ha. Interactive abstract pattern generation javascript demo, April 2016d. URL `http://blog.otoro.net/2016/04/24/interactive-abstract-pattern-generation-javascript-demo/`.

David Ha. Generating abstract patterns with tensorflow, March 2016e. URL `http://blog.otoro.net/2016/03/25/generating-abstract-patterns-with-tensorflow/`.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 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.

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, Jun 2016.

Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. In *Advances in Neural Information Processing Systems*, pages 6626–6637, 2017.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL `http://arxiv.org/abs/1502.03167`.

Andrej Karpathy. Convnetjs demo: Image "painting", March 2014. URL `https://cs.stanford.edu/people/karpathy/convnetjs/demo/image_regression.html`.

Samuli Karras, Tero Laine and Timo Aila. A style-based generator architecture for generative adversarial networks. *arXiv preprint arXiv:1812.04948*, 2018.

Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*, 2017.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

Karol Kurach, Mario Lučić, Xiaohua Zhai, Marcin Michalski, and Sylvain Gelly. A large-scale study on regularization and normalization in gans. In *International Conference on Machine Learning*, pages 3581–3590, 2019.

Carolyn Lamb, Daniel G Brown, and Charles LA Clarke. Evaluating computational creativity: An interdisciplinary tutorial. *ACM Computing Surveys (CSUR)*, 51(2):28, 2018.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

Jianhua Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information theory*, 37(1):145–151, 1991.

Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

Luke Metz and Ishaan Gulrajani. Compositional pattern producing gan. In *NIPS 2017 Workshop on Machine Learning for Creativity and Design*, December 2017.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral normalization for generative adversarial networks. *arXiv preprint arXiv:1802.05957*, 2018.

Augustus Odena. Open questions about generative adversarial networks. *Distill*, 2019. doi: 10.23915/distill.00018. https://distill.pub/2019/gan-open-problems.

Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

ProSecco. Computational creativity as a topic and a discipline, December 2018. URL `http://prosecco.computationalcreativity.net/field`.

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

S.J. Russell, S.J. Russell, P. Norvig, and E. Davis. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Prentice Hall, 2010. ISBN 9780136042594. URL `https://books.google.no/books?id=8jZBksh-bUMC`.

Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in neural information processing systems*, pages 2234–2242, 2016.

Jimmy Secretan, Nicholas Beato, David B D Ambrosio, Adelein Rodriguez, Adam Campbell, and Kenneth O Stanley. Picbreeder: evolving pictures collaboratively online. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1759–1768. ACM, 2008.

Claude Elwood Shannon. A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423, 1948.

Karl Sims. *Artificial evolution for computer graphics*, volume 25. ACM, 1991.

Kenneth O Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic programming and evolvable machines*, 8(2):131–162, 2007.

Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

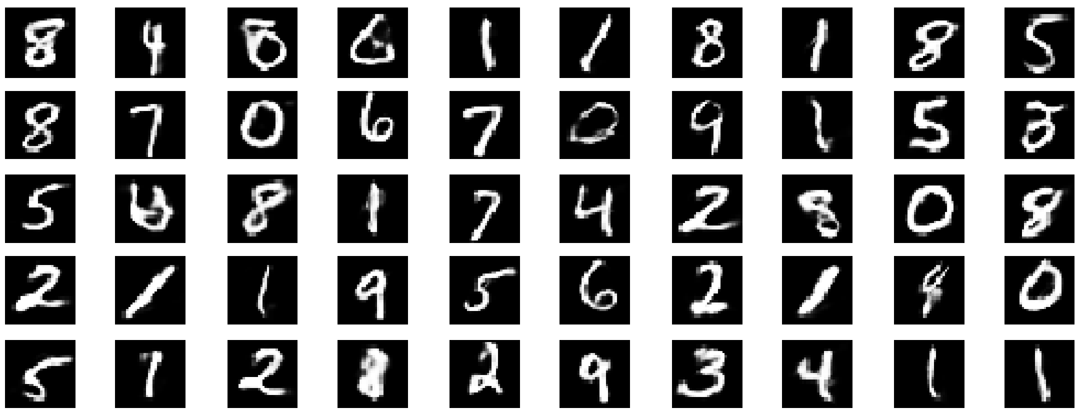
Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

Shengjia Zhao, Jiaming Song, and Stefano Ermon. Towards a deeper understanding of variational autoencoding models. *arXiv preprint arXiv:1702.08658*, 2017.

Appendices

1 Experimental Setup

Figure 1 and Figure 2 show the results from training an IMPGAN with different scaling factors and weight initialization schemes on the MNIST dataset.


(a) Scale $s = 1$

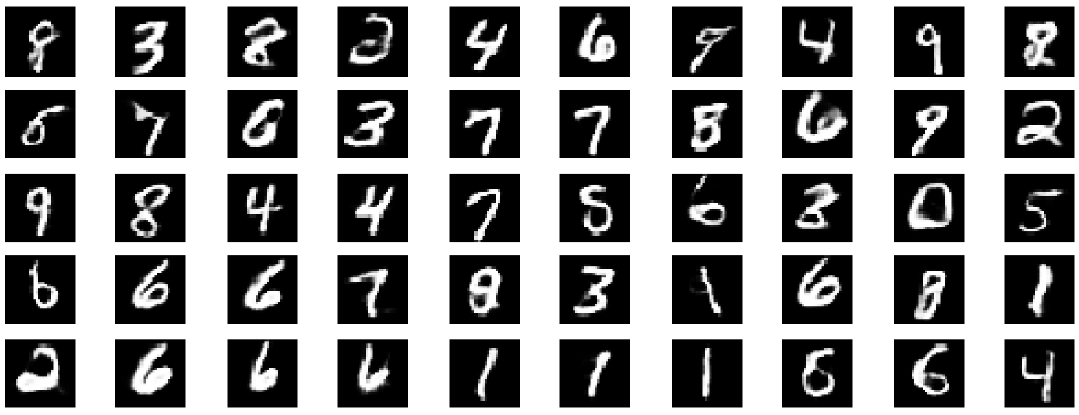
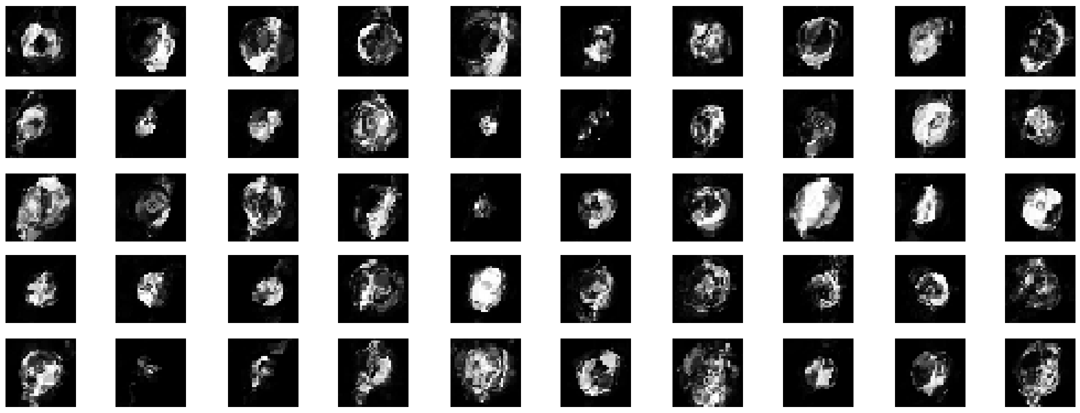

(b) Scale $s = 10$

Figure 1: Effect of scaling factor when using small initial weights


(a) Scale $s = 1$

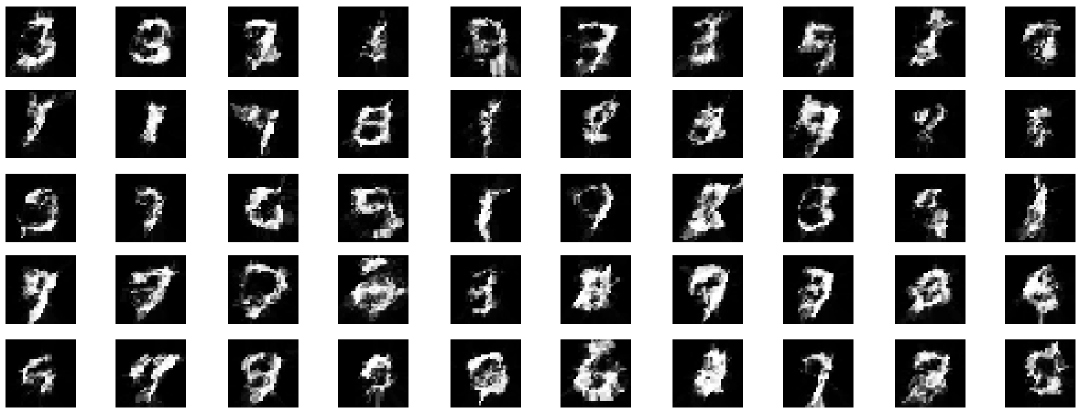

(b) Scale $s = 10$

Figure 2: Effect of scaling factor when using large initial weights

2 Generator Architectures

Table 1 and Table 2 show the architectures used for the conventional CNN and MLP generators, respectively.

Table 1: The default architecture for the CNN generator
The values are configured for the generation of 32x32 images with three color channels

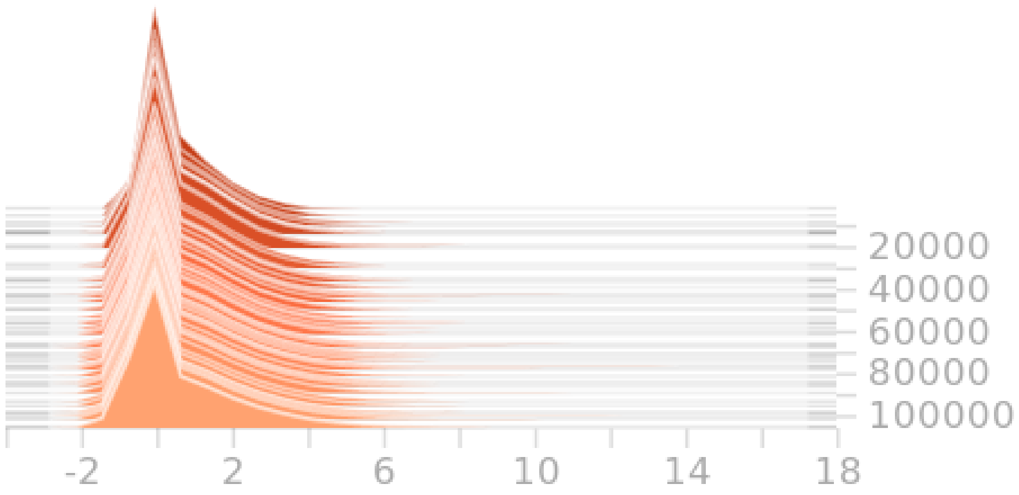
Layers	**Details**	**Normalization**	**Activation function**	**Output size/filters**
Fully connected	-	Batch normalization	ReLU	2048
Transposed convolution	Kernel size 4, stride 2	Batch normalization	ReLU	256
Transposed convolution	Kernel size 4, stride 2	Batch normalization	ReLU	128
Transposed convolution	Kernel size 4, stride 2	Batch normalization	ReLU	64
Transposed convolution	Kernel size 4, stride 2	-	tanh	3

Table 2: The default architecture for the MLP generator
The values are configured for the generation of 28x28 images with one color channel

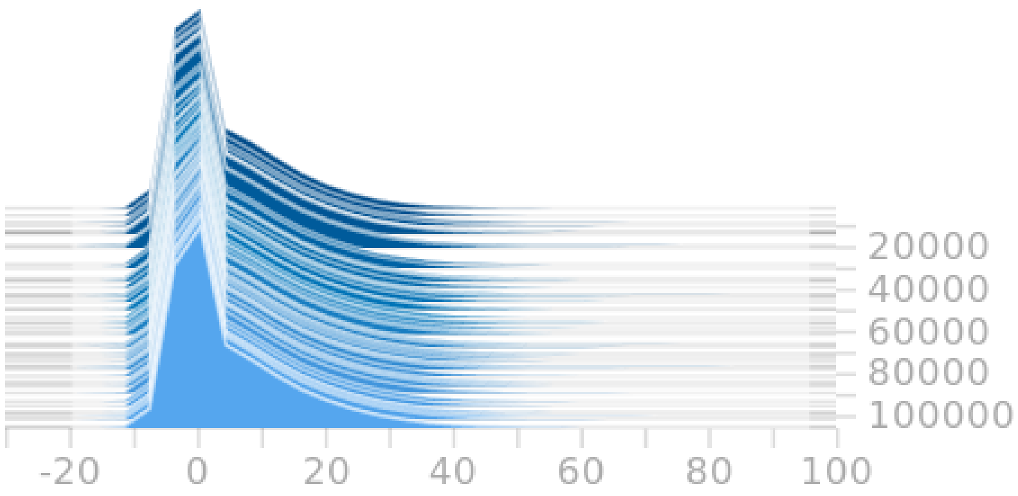
Layers	**Details**	**Normalization**	**Activation function**	**Output size**
Fully connected	-	Batch normalization	LReLU	128
Fully connected	-	Batch normalization	tanh	128
Fully connected	-	Batch normalization	tanh	128
Fully connected	-	Batch normalization	tanh	128
Fully connected	-	-	tanh	784

3 Histograms

Figure 3 and 4 respectively show the distribution of the outputs of the first and third layer of a CPPN over time. The x-axis represents the size of the output values, while the y-axis indicates the training step.

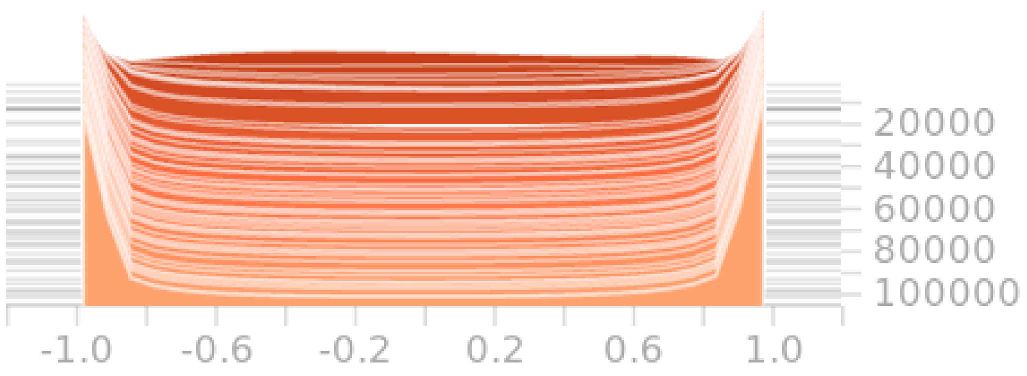


(a) Using small initial weights

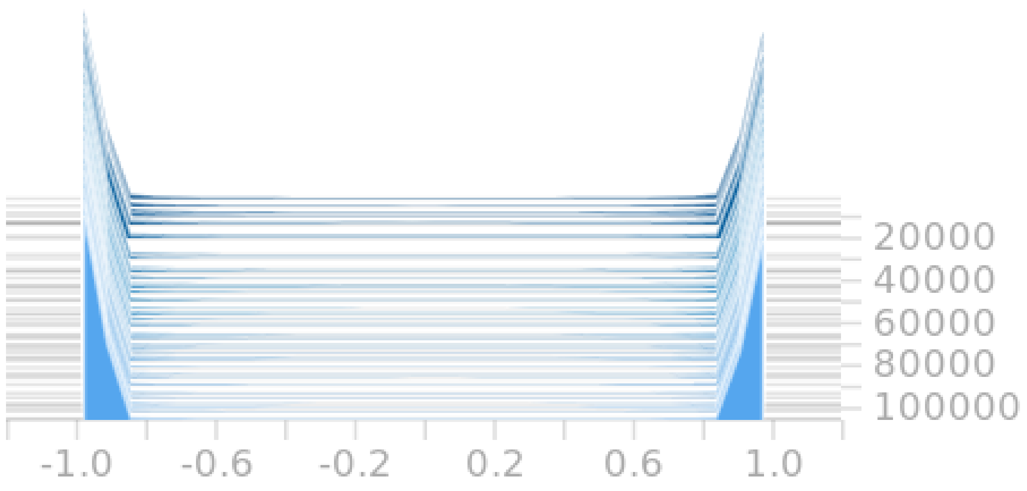


(b) Using large initial weights

Figure 3: The distribution of the outputs of the first layer in a CPPN. The CPPN contains four layers with activation functions LReLU and remaining tanh. The two figures use an identical CPPN with the exception of the sizes of the initial weights.



(a) Using small initial weights

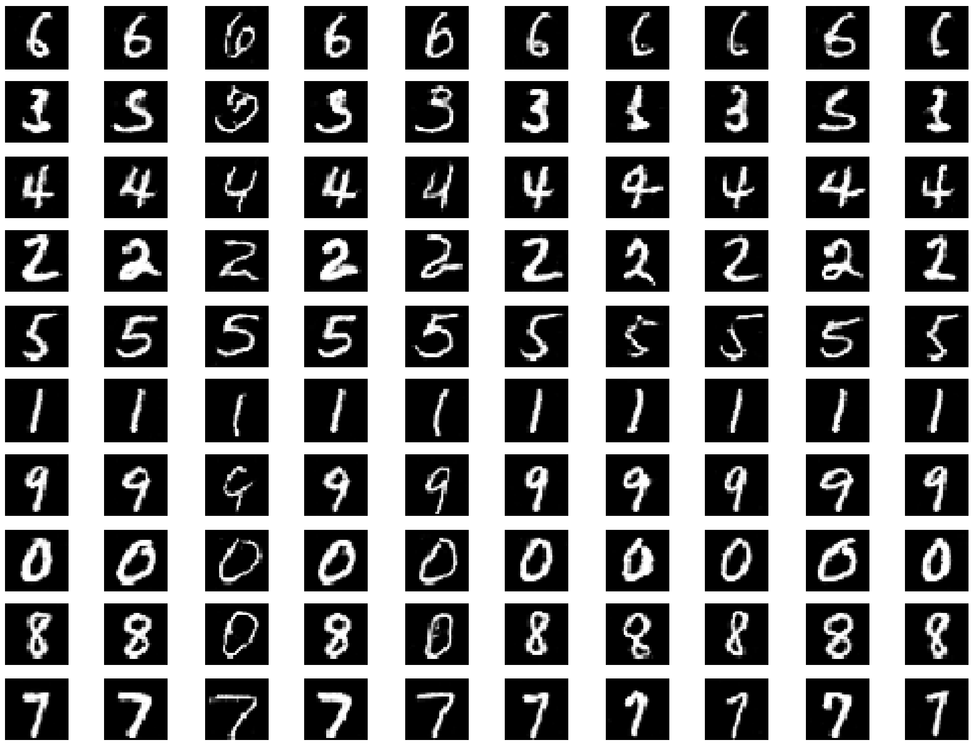


(b) Using large initial weights

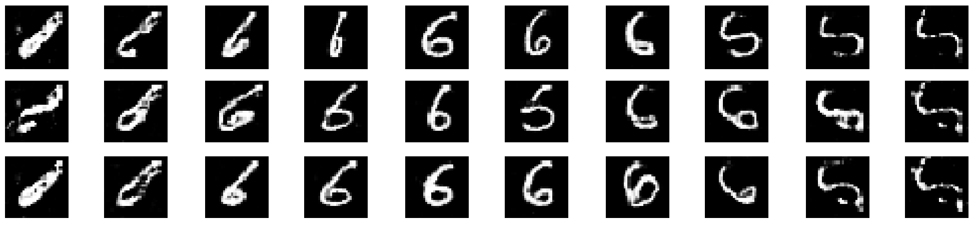
Figure 4: The distribution of the outputs of the third layer in a CPPN. The CPPN contains four layers with activation functions LReLU and remaining tanh. The two figures use an identical CPPN with the exception of the sizes of the initial weights.

4 Results from using CNN Generators

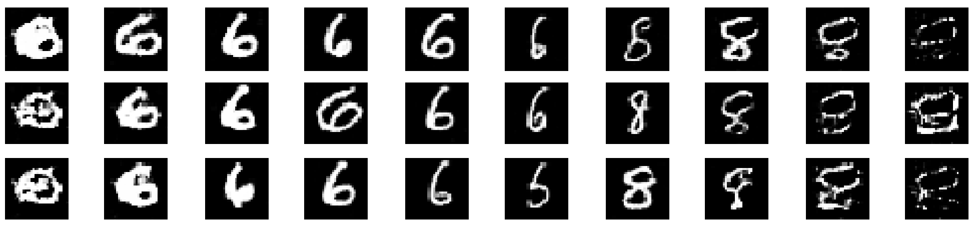
Figure 5, Figure 6, Figure 7 and Figure 8 show the results of varying the latent codes for GANs with CNN generators trained on the MNIST, Fashion MNIST, CelebA and CIFAR-10 datasets, respectively.



(a) Varying the categorical variable

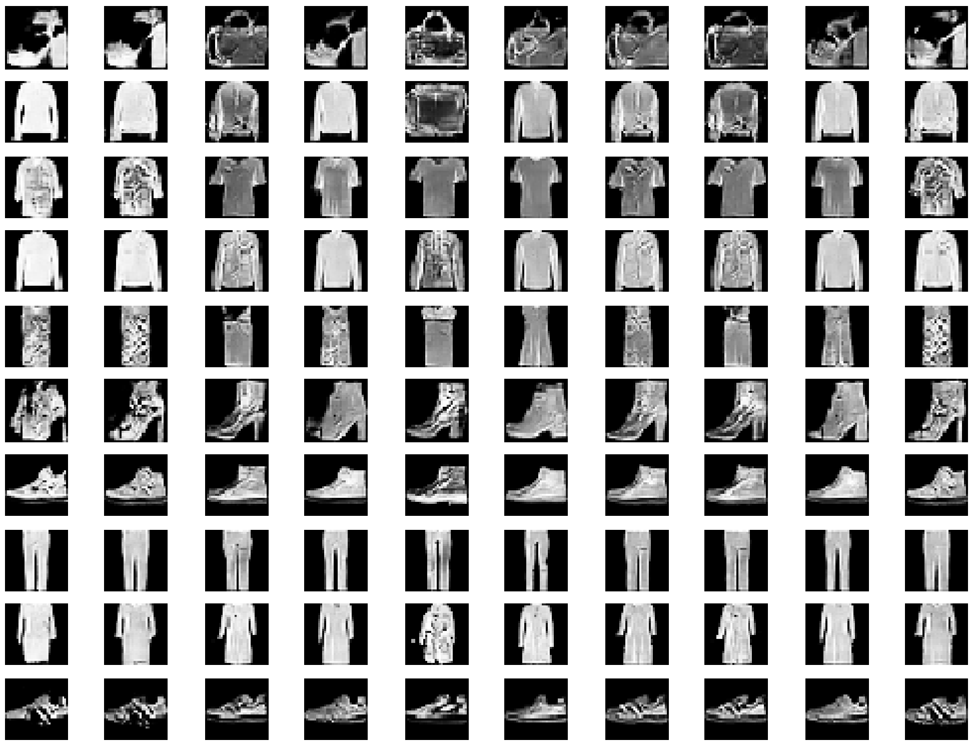


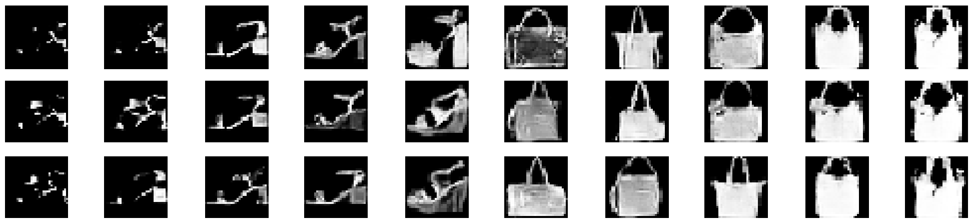
(b) Varying the first continuous variable



(c) Varying the second continuous variable

Figure 5: Generated images from a GAN with a CNN as the generator. The network is trained on the MNIST dataset for roughly 100 000 steps.


(a) Varying the categorical variable


(b) Varying the first continuous variable

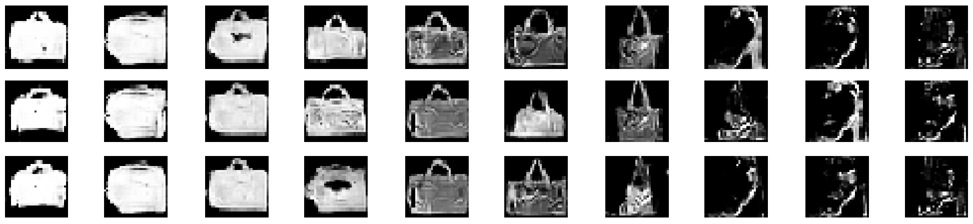
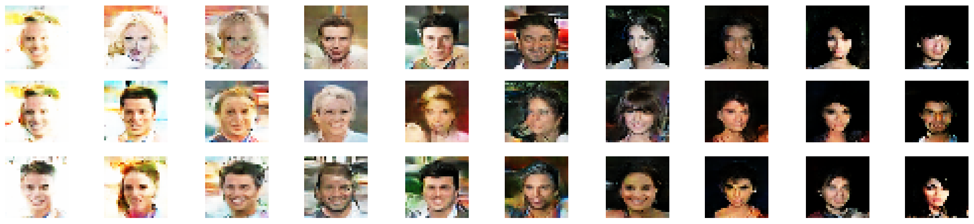
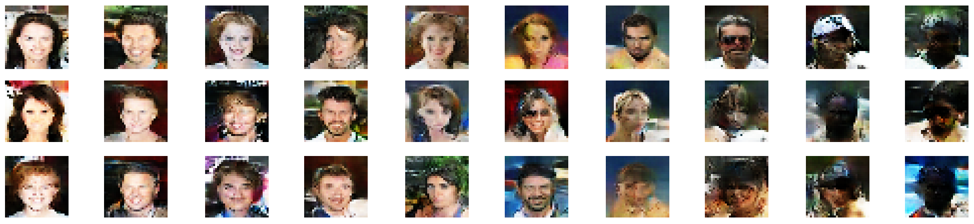

(c) Varying the second continuous variable

Figure 6: Generated images from a GAN with a CNN as the generator. The network is trained on the Fashion MNIST dataset for roughly 100 000 steps.

(a) Varying the categorical variable

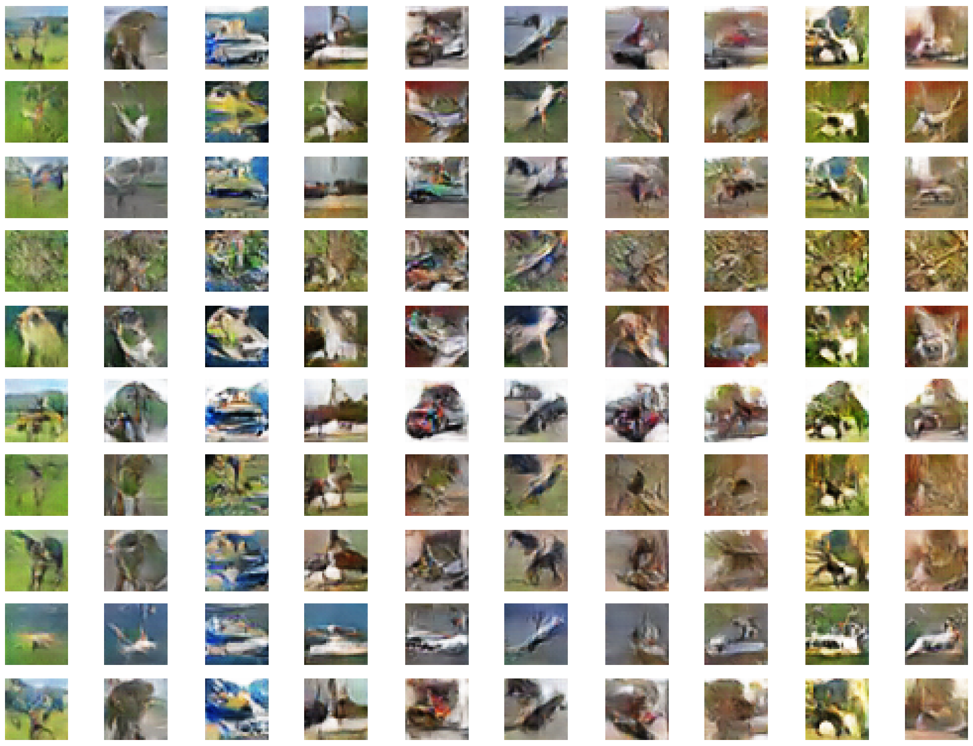


(b) Varying the first continuous variable

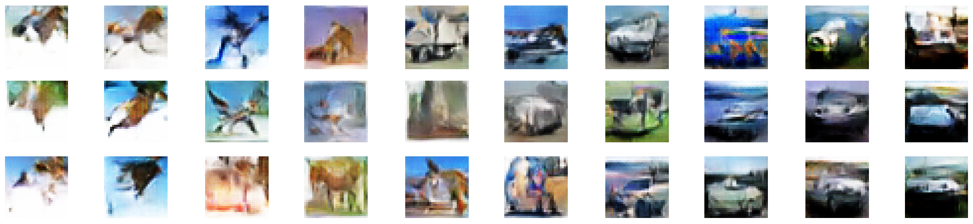


(c) Varying the second continuous variable

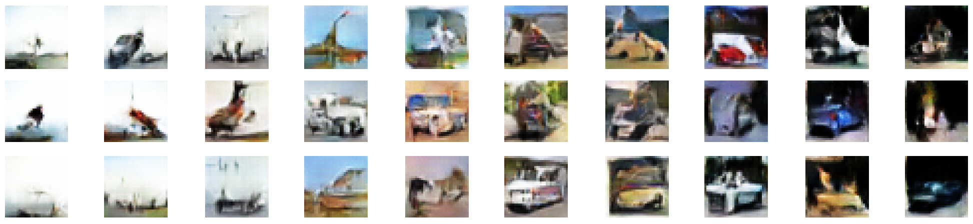
Figure 7: Generated images from a GAN with a CNN as the generator. The network is trained on the CelebA dataset for roughly 250 000 steps.



(a) Varying the categorical variable



(b) Varying the first continuous variable

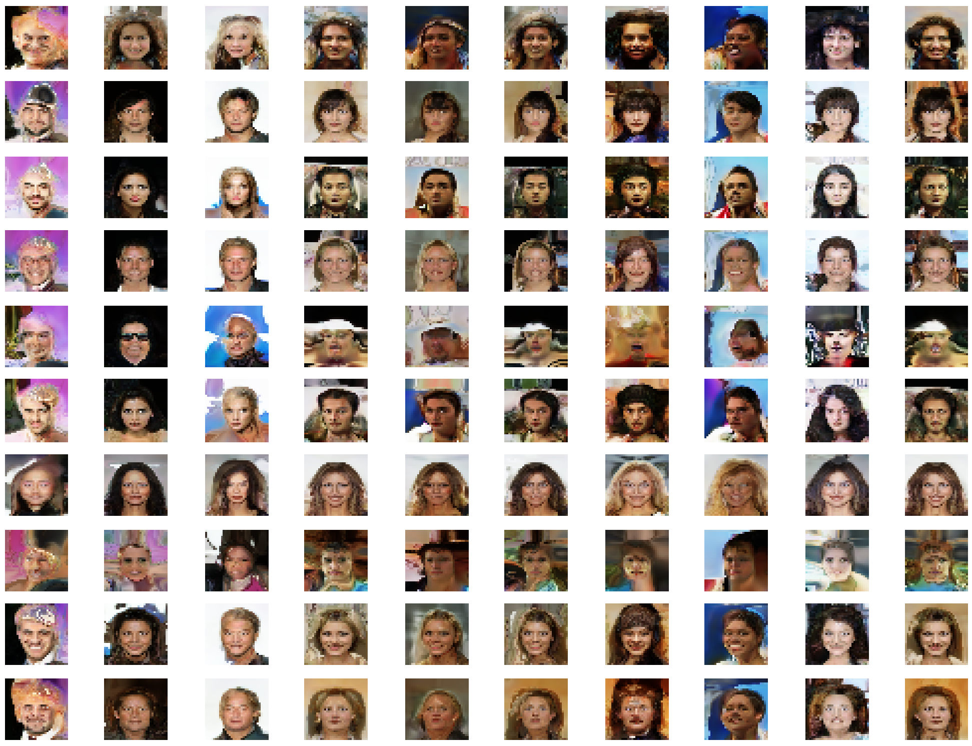


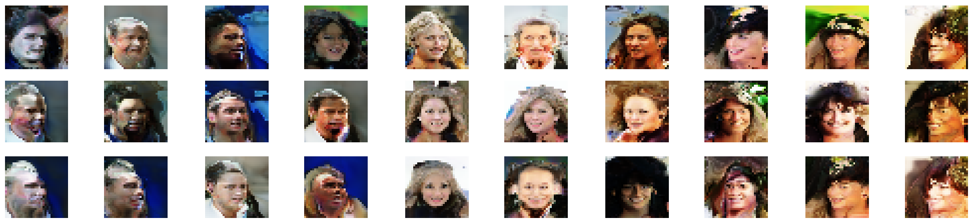
(c) Varying the second continuous variable

Figure 8: Generated images from a GAN with a CNN as the generator. The network is trained on the CIFAR-10 dataset for roughly 250 000 steps.

5 IMPGAN trained on the CelebA dataset

Figure 9 shows the results of varying the latent code on an IMPGAN trained on the CelebA dataset.


(a) Varying the categorical variable


(b) Varying the first continuous variable

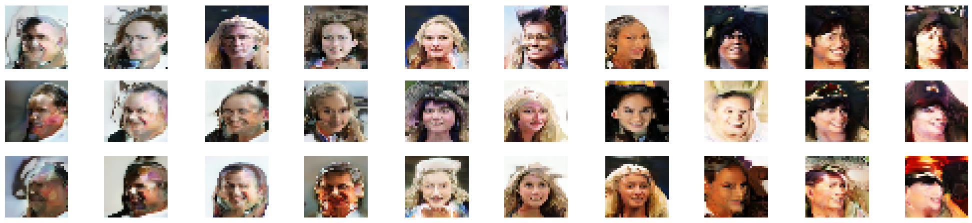

(c) Varying the second continuous variable

Figure 9: Generated images from an IMPGAN with a residual architecture. The network is trained on the CelebA dataset for roughly 250 000 steps.