Generative Adversarial Networks

By Bernd Prach
2 May 2018

Abstract

This essay is about Generative Adversarial Networks, short GANs. They are a framework which can be used to generate samples following some kind of complicated distribution, for example it can be used to produce pictures of human faces. The only information about the distribution we need for this is a good number of real samples from it.
GANs consist of two parts, the generator and the discriminator. We want to train the generator to learn to produce samples from the distribution we are interested in, and the discriminator will be trained to distinguish between real samples and samples generated by the generator.
This setup can be thought of as a student - teacher scenario: The generator produces pictures and the discriminator tries to figure out how the fake images differ from the real ones and tells the generator how to improve.
Usually GANs are used to generate pictures without any constraints, but with some modifications the framework can also be used for image inpainting, the task of filling in missing parts of a picture. We will talk about two approaches to do this in the last two chapters.

Contents

1 Introduction to GANs **1**
1.1 Global Optimality of $\mu_g = \mu_{data}$ 1

2 Introduction to Deep Learning **3**
2.1 Model 3
2.2 Loss Function 4
2.3 Learning 4
2.3.1 Gradient Descent 4
2.4 Convolutional Layers 6
2.5 Further Methods 8
2.5.1 Adam Algorithm 8
2.5.2 Batch Normalisation 9

3 Architecture of the DCGAN **12**
3.1 Model of the DCGAN 12
3.2 Loss Function of the DCGAN 13

4 Own Application of a GAN to Semantic Inpainting **15**
4.1 Alternated Model and Loss Function 15
4.2 Global Optimal G under Ideal Conditions 18
4.3 GAN Setting Compared to a Network with a Loss Function 21

5 More Advanced Semantic Image Inpainting **23**
5.1 Architecture of this Method 23
5.2 Poisson Blending 24
5.3 Results 24

6 Conclusion **27**

1 Introduction to GANs

As mentioned in the abstract, the GAN framework consists of two parts: The first part is the generator, $G(\,\cdot\,;\theta_g)$, a function parametrised by θ_g that maps from a random variable to some output space. A common choice for the random input is a vector of size 100 consisting of independent, identically distributed uniform variables on $[-1,1]$. The codomain is usually $\mathbb{R}^n$ for some n. For example, if we want the generator to produce $d_1 \times d_2$ RGB-pictures, the range would be $[0,1]^{d_1 \times d_2 \times 3}$. The second part is the discriminator $D(\,\cdot\,;\theta_d) : \mathbb{R}^n \to (0,1)$. Here $D(x;\theta_d)$ represents the probability that x came from the real data and not from G.
When we say we train G or D, we mean some process of consecutively updating θ_g or θ_d to minimise some loss function. We will leave away the parameters in the notation from now on to improve readability.
In the GAN framework, we want to train G to minimise the probability that the pictures are fake, which is $1 - D(x)$. More precisely, we would like to minimise the expectation over all values of z of $\log(1 - D(G(z))$. For D, we want to maximise $\log(D(x))$ if x is from the real samples and $\log(1 - D(x))$ if x is generated by G. We will write μ_{data} for the distribution we are interested in and μ_z for the distribution used for the random input. Further, for later use we define μ_{g} to be the distribution of $G(z)$ when $z \sim \mu_z$. We will use p_{data}, p_{z} and p_{g} for their PDFs. Then, the setup described above corresponds to the two-player minimax game with value function

$$V(G,D) = \mathop{\mathbb{E}}_{x\sim\mu_{\text{data}}} \left[\log\left(D(x)\right)\right] + \mathop{\mathbb{E}}_{x\sim\mu_z} \left[\log\left(1 - D(G(z))\right)\right], \tag{1}$$

where one player tries to find θ_g to minimise V and the other player seeks θ_d to maximise V.

1.1 Global Optimality of $\mu_{\text{g}} = \mu_{\text{data}}$

In the following chapter we want to establish the theoretical result that, if both G and D are not parametric but can be arbitrary functions, the minimax game given by (1) has the global minimum $\mu_{\text{g}} = \mu_{\text{data}}$, i.e. the generator creates pictures from the distribution we are interested in.
For this we first try to find the optimal D for any given G:

Proposition 1. *For G fixed, the optimal discriminator D^*_G is given by*

$$D^*_G(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}. \tag{2}$$

Proof. D^*_G is given by the function D that maximises $V(G,D)$. We get that

$$\begin{aligned} V(G,D) &= \mathop{\mathbb{E}}_{x\sim\mu_{\text{data}}} \left[\log\left(D(x)\right)\right] + \mathop{\mathbb{E}}_{x\sim\mu_{\text{g}}} \left[\log\left(1 - D(x)\right)\right] \\ &= \int_x \Big(p_{\text{data}}(x)\log(D(x)) + p_{\text{g}}(x)\log(1 - D(x)) \Big) dx. \end{aligned} \tag{3}$$

Considering the part inside the integral and differentiating gives

$$\begin{aligned} &\frac{d}{dD(x)} \Big(p_{\text{data}}(x)\ \log(D(x)) + p_{\text{g}}(x)\ \log(1 - D(x)) \Big) \\ &= \frac{p_{\text{data}}(x)}{D(x)} - \frac{p_{\text{g}}(x)}{1 - D(x)} = \frac{p_{\text{data}}(x) - D(x)\big(p_{\text{data}}(x) + p_{\text{g}}(x)\big)}{D(x)(1 - D(x))}. \end{aligned}$$

This is positive if and only if $D(x) < \frac{p_{\text{data}}(x)}{p_{\text{data}}(x)+p_{\text{g}}(x)}$, so D^*_G is given by (2). □

Theorem 1. *In this setting,* $\max_D V(G, D)$ *is minimised by* G *if and only if* $\mu_g = \mu_{data}$*. In that case* V *achieves the value* $-\log\ 4$*.*

Proof. By (2) we have $\max_D V(G, D) = V(G, D^*_G)$ and from (3):

$$\begin{aligned} V(G, D^*_G) &= \int_x \left(p_{\text{data}}(x) \log \left(\frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_{\text{g}}(x)} \right) + p_{\text{g}}(x) \log \left(1 - \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_{\text{g}}(x)} \right) \right) dx \\ &= \int_x -H \left(\frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_{\text{g}}(x)} \right) (p_{\text{data}}(x) + p_{\text{g}}(x)) dx, \end{aligned} \tag{4}$$

for $H(t) = -t\ \log(t) - (1-t)\ \log(1-t)$, the entropy function.
For $0 < t < 1$, we have

$$H'(t) = -\log(t) + \log(1-t), \tag{5}$$

and

$$H''(t) = -\frac{1}{t} - \frac{1}{t-1} < 0. \tag{6}$$

So, $H(t)$ is maximal for $t = \frac{1}{2}$ and $H(t) \leqslant -\log \frac{1}{2} = \log\ 2$ with equality if and only if $t = \frac{1}{2}$.
This gives

$$V(G, D^*_G) \geqslant \int_x -(\log 2)\big(p_{\text{data}}(x) + p_{\text{g}}(x)\big) dx = -2 \log 2 = -\log 4, \tag{7}$$

with equality if and only if $\frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_{\text{g}}(x)} = \frac{1}{2}$ almost everywhere, which is equivalent to $\mu_{\text{g}} = \mu_{\text{data}}$ □

This is a promising result, because we can hope that even in the real setup G will end up producing samples from a distribution similar to μ_{data}.
The main difference in the real setup is that we do not have access to the μ_{data} distribution, otherwise there would not be a point using a GAN to sample from it. Instead, we only have a number of samples from the distribution, so we have to use the sample mean of $\log(D(x))$ instead of its expectation under μ_{data} for our real loss function.
Unfortunately replacing μ_{data} with the discrete distribution of our samples in the proof above shows that the optimal μ_{g} in this case is exactly this discrete sample distribution. As this is not what we are interested in, we will have constraints on G and D, and those constraints will be that both G and D are fixed parametric functions. We will describe in the next two chapters what those functions look like.

2 Introduction to Deep Learning

In this chapter we will go through some of the basics of artificial neural networks to figure out more about how GANs work. The focus in this section is on methods actually used in GANs.

2.1 Model

The first part of a neural network is the model. Usually, the goal of such a network is to approximate some function f^* of the input. This could for example be the correct label for an input of a picture of a handwritten digit, or, in case of the discriminator whether the input is from the real data or generated by G.
To approximate f^* we use some function $f(\,\cdot\,;\theta)$, and our goal is to find a value of the parameter θ such that $f^* \sim f(\,\cdot\,;\theta)$.
The model describes the structure of f.

Usually f is built up in layers, say k of them (for some $k \in \mathbb{N}$).
We start with $h^{(0)} = x$, for x the input of f, and then we have for $l = 1, \dots, k$:

$$h^{(l)} = g^{(l)}\left(W^{(l)^T} h^{(l-1)} + b^{(l)}\right), \tag{8}$$

where the $g^{(l)}$ are called the activation functions, $W^{(l)}$ are called the weight matrices and the $b^{(l)}$ are called the bias parameters. All those vectors $h^{(l)}$ are called hidden layers, and as in (8), each layer is given by a composition of some given function with a linear function of the previous layer. All the $W^{(l)}$ and $b^{(l)}$ are part of θ; they are parameters we want to optimise during training. The $g^{(l)}$ are functions determined by the model. Often, all the $g^{(l)}$ are the same function, common choices include elementwise application of one of the following functions:

- $g(z) = \max\{0, z\}$, called the rectified linear unit, or ReLU function.
- $g(z) = \begin{cases} z & \text{if } z \geq 0 \\ \delta z & \text{if } z < 0 \end{cases}$, for some $\delta \ll 1$, called the leaky ReLU.
- $g(z) = \dfrac{e^z}{1 + e^z}$, called the sigmoid or expit function.

The main reason why we need the activation function is because otherwise f would just be a composition of linear functions, i.e. a linear function.

The final part of the model is the output rule g^{out}, a map from the last hidden layer, $h^{(k)}$, to the output layer, which gives the output of f. Usually this map is chosen to represent the desired output of f. For example, for classification tasks we usually want f to generate a vector of probabilities for the different labels. Then, the softmax function can be a good choice:

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}. \tag{9}$$

This way we get an output vector with each entry in $(0, 1)$ that sums to 1.
The main criterion to choose the activation functions and the output rule is that we want them to be differentiable almost everywhere. Furthermore, it can be helpful if the gradient does not vanish as z grows. That is why the ReLU function is quite popular.

2.2 Loss Function

To evaluate how well f is approximating f^* we need a loss function.
One possible loss function is the log-likelihood. Suppose e.g. we have a classification task with softmax output rule. Then for input x with class $f^*(x) = l$ and with final layer $h^{(k)}$ we get

$$\begin{aligned} L\big(f(x), f^*(x)\big) &= \log\left(\sum_j \mathbb{1}_{\{f^*(x)=j\}} f(x)_j\right) = \log f(x)_l \\ &= \log \operatorname{softmax}\left(h^{(k)}\right)_l \\ &= \log \frac{\exp\left(h_l^{(k)}\right)}{\sum_j \exp\left(h_j^{(k)}\right)} \\ &= h_l^{(k)} - \log\left(\sum_j \exp\left(h_j^{(k)}\right)\right). \end{aligned} \tag{10}$$

Note that if $h_l^{(k)}$ is small compared to the other components of the vector, f will assign a low probability to the corresponding class, so f is bad at classifying x. But in this case, we also see that $h_l^{(k)}$ contributes little to the second part of the loss function. So, the derivative of the loss function with respect to $h_l^{(k)}$ is approximately 1. This, as we will see later, will lead to quick correction.
On the other hand, if f is good at classifying x, $h_l^{(k)}$ will be much larger than $h_j^{(k)}$ for $j \neq l$. Then $\log\left(\sum_j \exp\left(h_j^{(k)}\right)\right) \sim h_l^{(k)}$, so the derivative of L with respect to $h_l^{(k)}$ will be small. This behaviour is desired, as we already classify correctly, so there is no need to change the parameters.
This shows that the log-likelihood is a good loss function together with the softmax output rule.

There are also other loss functions one can use; the main requirement is again differentiability. For example, in our GAN setting the loss function for the generator is the log of the probability that the discriminator classifies wrong.

2.3 Learning

The part that is usually called learning refers to finding a good value for our parameter θ. Unfortunately, this is usually impossible to do analytically. Therefore, we have to use an algorithm to update θ stepwise until we arrive at a good value. The main algorithms used are based on gradient descent:

2.3.1 Gradient Descent

Gradient descent works by calculating the partial derivatives of the loss function with respect to the different components of θ and then updating θ corresponding to that.
The partial derivatives are calculated by an algorithm called **backpropagation**. It goes backwards through the model and calculates the partial derivatives using the chain rule:
We start with

$$\frac{\partial}{\partial y} L(y, f^*(x))\Big|_{y=f(x;\theta)}. \tag{11}$$

We can evaluate this because of our requirement that L is differentiable.
Next, we want to calculate the partial derivatives with respect to $h_i^{(k)}$ for all i. We know the output

rule g^{out} is differentiable, and

$$\frac{\partial}{\partial h_i^{(k)}} L = \frac{\partial f(x;\theta)}{\partial h_i^{(k)}} \left.\frac{\partial L(y, f^*(x))}{\partial y}\right|_{y=f(x;\theta)} = \frac{\partial g^{out}\left(h^{(k)}\right)}{\partial h_i^{(k)}} \left.\frac{\partial L\left(y, f^*(x)\right)}{\partial y}\right|_{y=f(x;\theta)}. \tag{12}$$

We already know the second factor, and can calculate the first one because of our requirement of g^{out} to be differentiable.
Next, we use that for all l we have: $h^{(l)} = g^{(l)}\left(W^{(l)^T} h^{(l-1)} + b^{(l)}\right)$. From that we get

$$\begin{aligned} \partial L(f(x;\theta), f^*(x)) &= \sum_i \frac{\partial L}{\partial h_i^{(l)}}\ \partial h_i^{(l)} \\ &= \sum_i \frac{\partial L}{\partial h_i^{(l)}}\ g_i^{(l)'}\left(W^{(l)^T} h^{(l-1)} + b^{(l)}\right)\ \partial\left(W^{(l)^T} h^{(l-1)} + b^{(l)}\right). \end{aligned} \tag{13}$$

We can use that to evaluate the partial derivative with respect to the parameters $W_{ij}^{(l)}$ and $b_i^{(l)}$ for different i and j, as well as the partial derivatives with respect to $h_i^{(l-1)}$, the previous hidden layer. So going backwards through the hidden layers we can compute the partial derivatives of $L(f(x;\theta), f^*(x))$ with respect to all of θ.
Now, knowing all of those, we update the parameter, using some previously chosen learning rate ϵ:

$$\theta^{(new)} = \theta^{(old)} - \epsilon \frac{1}{m} \sum_x \nabla_\theta L(f(x;\theta), f^*(x))\Big|_{\theta=\theta^{(old)}}, \tag{14}$$

where the sum is over all inputs and m is the number of inputs. Equation (14) shows why we do not want small gradients: In this case the parameter updates may be tiny, and we are not improving much. For example, if $W_{ij}^{(l)}$ is huge and $g^{(l)}$ is the sigmoid function, the derivative with respect to $W_{ij}^{(l)}$ is very small. So it will take a long time to reduce the size of $W_{ij}^{(l)}$.

This procedure is usually repeated until the derivative is close to 0, which means we are either close to the minimum if we are lucky, but we could also be at a local minimum or a saddle point.

The problem with this algorithm is that it can be very expensive to even compute one update if the training set is large. That is why in practice usually **stochastic gradient descent** (algorithm 1) is used.

In this algorithm, instead of going through the whole training set, in each step, we pick a subset of size m' uniformly from the samples, where m' is usually chosen match smaller than m. We usually denote this subset as $\mathbb{B} = \{x^{(1)}, \ldots, x^{(m')}\}$ and call it a **minibatch**.
Then, the gradient is estimated using only those samples.

It can be shown (see [5]) that SGD converges to a local optimum under the following conditions:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \tag{16}$$

and

$$\sum_{k=1}^{\infty} \epsilon_k^2 < \infty. \tag{17}$$

Algorithm 1 Stochastic Gradient Descent (SGD)

Set: $n = \left\lfloor \frac{m}{m'} \right\rfloor$, the number of minibatches.
Require: Learning rate ϵ_t for each epoch.
Initialise: $\hat{\theta}^{(0,n)}$, consisting of the weight matrices and bias parameters, randomly. Shuffle the training samples and split them into minibatches $\mathbb{B}_1, \dots, \mathbb{B}_n$.
for each epoch $t = 1, 2, \dots$ **do:**
$\hat{\theta}^{(t,0)} = \hat{\theta}^{(t-1,n)}$
for $s = 1, 2, \dots, n$

- Compute $h^{(l)}$ for $l = 1, \dots, k$ as well as $f(x^{(i)}; \hat{\theta}^{(t,s-1)})$ for all inputs $x^{(i)} \in \mathbb{B}_s$. (**forward sweep**)
- Compute the contribution of $\big(x^{(i)}, f^*(x^{(i)})\big)$ to the partial derivative of the loss function for all $x^{(i)} \in \mathbb{B}_s$. (**backward sweep**)
- Update the parameters:

$$\hat{\theta}^{(t,s)} = \hat{\theta}^{(t,s-1)} - \epsilon_t \frac{1}{m'} \sum_{x^{(i)} \in \mathbb{B}_s} \nabla_\theta L(f(x;\theta), f^*(x))\Big|_{\theta=\hat{\theta}^{(t,s-1)}}. \tag{15}$$

end for
end for

So far, we have covered the basics of a neural network. There are a few more things necesarry to understand the currently popular versions of GANs.

2.4 Convolutional Layers

So far we have only considered fully connected layers, meaning that there is a separate parameter that describes how $h_i^{(l-1)}$ influences $h_j^{(l)}$ for all i, j.
In convolutional layers the same parameters are used more often. Formally, for a two-dimensional picture I as our input, and some two-dimensional kernel K, convolution on I and K is given by:

$$S(i,j) = \big(I * K\big)(i,j) := \sum_m \sum_n I(m,n) K(i-m, j-n), \tag{18}$$

where $*$ is called the convolution operator, and we set $I(m,n) = 0$ for m or n outside of the range of pixels.

Usually, we pick a K that has only a small number of non-zero entries, which will be optimised during training. This greatly reduces the number of parameters that need to be optimised.

One example of what convolution could do is that it can determine whether there is an edge in a given pixel, as this feature only depends on a few neighbouring pixels, and the dependence is the same for any position within the input.

Often convolutional layers also make use of a stride. This means $S(i,j)$ is not evaluated at every point (i,j), but only at every other point for example. This way the next layer has only half the length and width of the previous layer. This is for example used multiple times in the discriminator, where we start with a whole picture as the input and want a single probability value as the output. An example of a convolutional layer with a stride can be found in Figure 1.

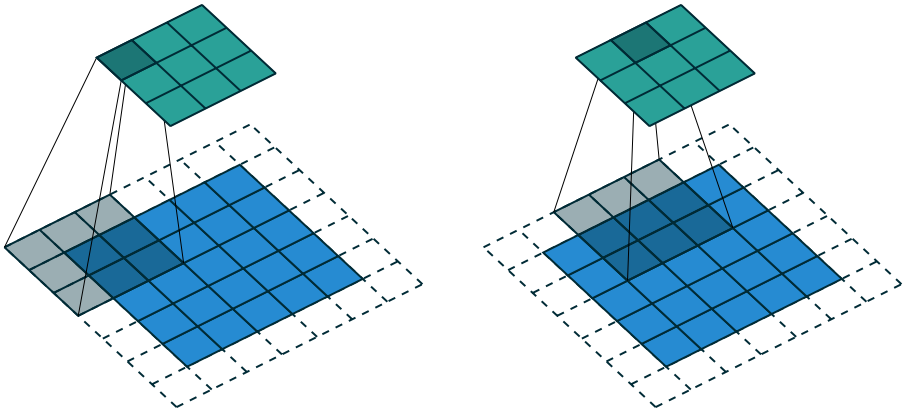

Figure 1: Illustration of **Convolution with stride 2** from the input (blue) to the output (green). This image is taken from [6].

Usually, our layers do not have only one value per pixel, but can have much more than that. For example, the input pictures for the discriminator have three values corresponding to the red, green and blue intensities at each pixel. This dimension is referred to as channels. In this case K has another input dimension, and potentially also gives a vector as output, and we can write the convolution as

$$S(i,j)_k = \sum_l \sum_m \sum_n I(m,n,l) K(i-m, j-n, l)_k, \tag{19}$$

for k referring to the channel of the output and l to the one of the input.

There is a similar layer to increase the number of pixels from one layer to the next, usually called a **transposed convolution**. It consists of adding rows and columns of zeros between any two original rows or columns, and then applying a traditional convolutional layer to it (with no stride). That way the dimensions are approximately doubled from one layer to the next. Transposed convolution is visualised in Figure 2.

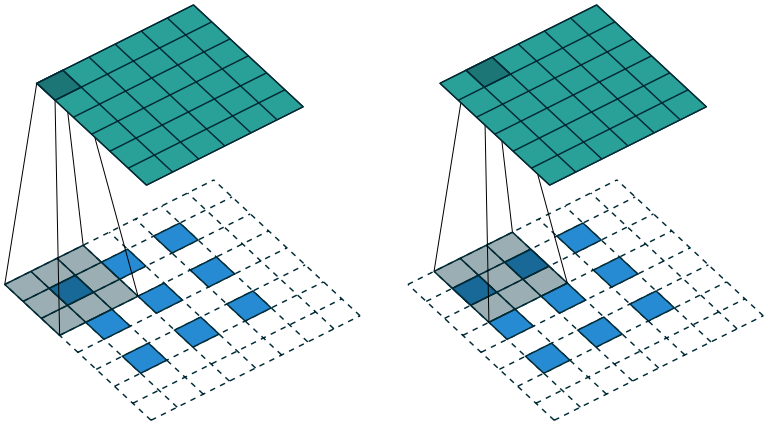

Figure 2: Illustration of **Transposed Convolution** from the input (blue) to the output (green). This image is taken from [6].

2.5 Further Methods

I will describe two more methods that are used in currently popular versions of GANs. They are not essential for the understanding of GANs themselves, but proved to work well with them.

2.5.1 Adam Algorithm

Adam was introduced in the paper [7]. It stands for adaptive moment estimation and it belongs to a class of algorithms that use momentum for stochastic optimisation.
The motivation behind momentum is that if we have one dimension with very noisy gradients and another dimension with small but consistent gradients we mainly want to continue updating in the second dimension. An example of this, showing that gradient descent may not work well in that case can be seen in Figure 3.

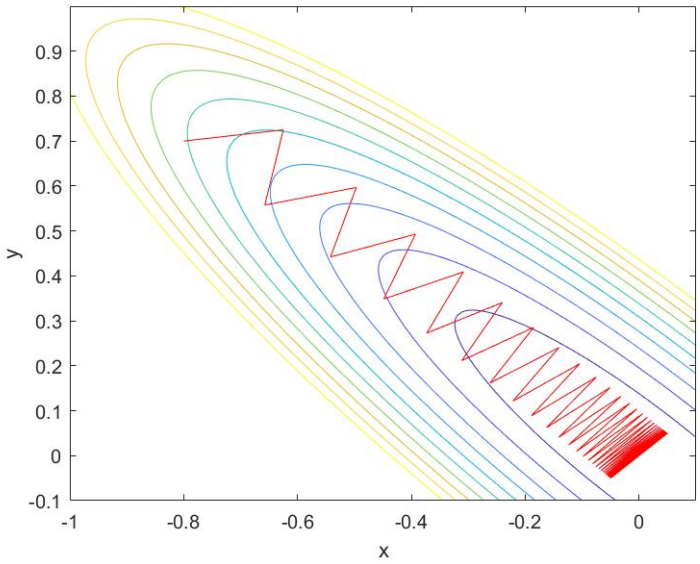

Figure 3: **Example where gradient descent does not work well.** Gradient descent applied to the function $f(x, y) = (x + y)^2 + 0.05 * (x - y)^2$ with $\epsilon = 0.5$ and starting point $(x, y) = (-0.8, 0.7)$.
We see that gradient descent struggles to fully converge to the minimum.
Code that created this figure can be found at [10]/figures_essay.

One can think of momentum as in the physical analogy, with a hockey puck sliding down an icy surface.

To apply momentum based optimisation, in addition to our learning rate ϵ, we also need another hyperparameter $\alpha \in [0, 1)$, which can be thought of as representing friction in our physical example. With this, the update rule is given by

$$\begin{aligned} v^{(t)} &= \alpha v^{(t-1)} - \epsilon \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L(f(x^{(i)}; \theta), f^*(x^{(i)})) \right) \Bigg|_{\theta=\theta^{(t-1)}} \\ \theta^{(t)} &= \theta^{(t-1)} + v^{(t)}. \end{aligned} \tag{20}$$

The Adam algorithm (presented in Algorithm 2) takes this idea further and also estimates the square of the gradient, denoted $w^{(t)}$, in the same way. Furthermore, it corrects for the bias introduced by initialising $v^{(0)}$ and $w^{(0)}$ as 0.
The update is then computed as

$$\Delta\theta = -\epsilon \frac{v^{(t)}}{\sqrt{w^{(t)}} + \delta}, \tag{21}$$

where all the operations are applied elementwise and the hyperparameter δ is $\ll 1$ and just there for numerical stability.

Algorithm 2 The Adam Algorithm

Require: ϵ: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: δ: small constant used for numerical stabilisation, $\delta \sim 10^{-8}$
Require: $\theta^{(0)}$: initial parameter
Set: $v^{(0)} = 0$, $w^{(0)} = 0$ (Initialise 1^{st} and 2^{nd} moment vector)
for $t = 1, 2, \ldots$ **do:**
Sample minibatch of m' samples $\{x^{(1)}, \ldots, x^{(m')}\}$ with corresponding labels $\{f^*\big(x^{(1)}\big), \ldots, f^*\big(x^{(m')}\big)\}$.
$g^{(t)} = \frac{1}{m} \nabla_\theta \sum_{i=1}^{m'} L(f(x^{(i)}; \theta), f^*(x^{(i)}))\Big|_{\theta=\hat{\theta}^{(t-1)}}$ (compute gradient)
$v^{(t)} = \beta_1 v^{(t-1)} + (1 - \beta_1) g^{(t)}$ (Update biased 1^{st} moment estimate)
$w^{(t)} = \beta_2 w^{(t-1)} + (1 - \beta_2) {g^{(t)}}^2$ (Update biased 2^{nd} moment estimate)
$\hat{v}^{(t)} = v^{(t)}/(1 - \beta_1^t)$ (correct for bias)
$\hat{w}^{(t)} = w^{(t)}/(1 - \beta_2^t)$ (correct for bias)
$\Delta\theta = -\epsilon \frac{v^{(t)}}{\sqrt{w^{(t)}} + \delta}$ (compute update, all operations applied elementwise)
$\theta^{(t)} = \theta^{(t-1)} + v^{(t)}$ (apply update)
end for

The advantages of that update rule is that we have an upper bound for the update. That way we can not lose too much of our progress by having one bad minibatch that is not representative of the data, or if we happen to be on a cliff with very steep gradient (see Figure 4).
Furthermore, if we are near the optimum we expect the stochastic gradient to contain a high proportion of noise, which will cause $\hat{v}^{(t)}$ to be much smaller than $\sqrt{\hat{w}^{(t)}}$. Therefore, the stepsize will automatically be reduced by the algorithm, making if fairly robust to the choice of hyperparameters. Figure 5 shows the Adam algorithm used for the same problem as above.

2.5.2 Batch Normalisation

One problem with updating all parameters at the same time according to their partial derivatives is that we get second- and higher-order effects that could potentially make a difference if the network has a lot of layers, and we would like to prevent that from happening. To address this problem, we introduce **batch normalisation**.
Batch normalisation tries to normalise the output of each layer before applying the activation function. This solves our problem, but also means that early layers have less influence in general.
To be able to do this we also need to change our model slightly. We used to think of our model as a function from a single input vector to some output. To be able to do batch normalisation we need

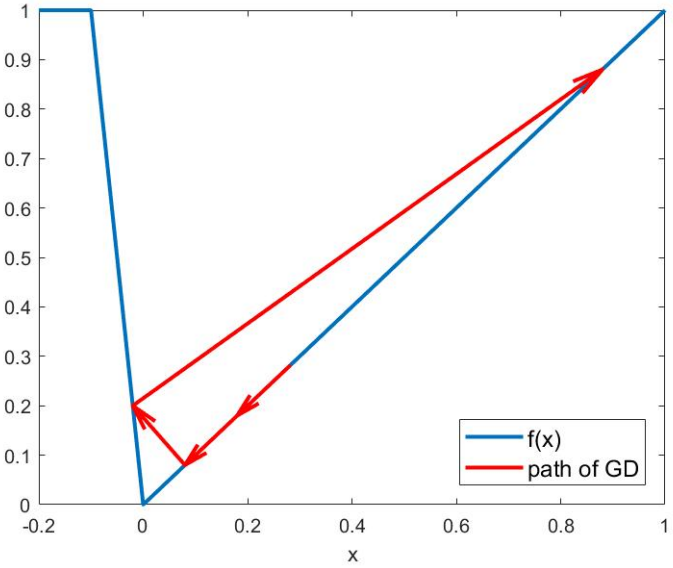

Figure 4: **Gradient Descent not doing well because of a cliff.** Here, f is the piecewise linear function given by the points $(-\frac{1}{10}, 1)$, $(0, 0)$ and $(1, 1)$. We start gradient descent at $x = 0.28$ with $\epsilon = 0.1$. We see that gradient descent can lose a lot of progress if it ends up on a steep cliff.
Code that created this figure can be found at [10]/figures_essay.

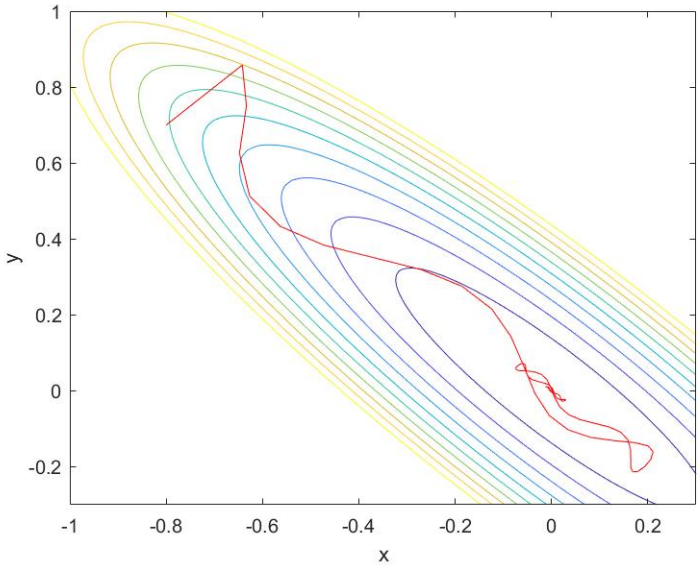

Figure 5: Same example as in Figure 3, with **Adam algorithm** applied to it this time.
This time, the code clearly converges to the minimum.
Code that created this figure can be found at [10]/figures_essay.

to change that function to be from a minibatch of input vectors to the same number of outputs.
For notation, write $S^{(l)}(x^{(j)})$ for the output of the linear map of hidden layer l, when evaluated with input (of the first layer) $x^{(j)}$.
With that notation, our previous model can be written as $h^{(0)}(x^{(j)}) = x^{(j)}$ and for $l = 1, \dots, k$

$$\begin{aligned} s^{(l)}\left(x^{(j)}\right) &= {W^{(l)}}^T h^{(l-1)}\left(x^{(j)}\right) + b^{(l)} \\ h^{(l)}\left(x^{(j)}\right) &= g^{(l)}\left(s^{(l)}\left(x^{(j)}\right)\right). \end{aligned} \tag{22}$$

For the new model, we keep the first equation but replace the second one with

$$h^{(l)}\left(x^{(j)}\right) = g^{(l)}\Bigg(\mathrm{BN}^{(l)}\left(s^{(l)}(x^{(j)});\ s^{(l)}(x^{(1)}), \dots, s^{(l)}(x^{(m')})\right)\Bigg), \tag{23}$$

where

$$\mathrm{BN}_i^{(l)}\left(s^{(l)}(x^{(j)});\ s^{(l)}(x^{(1)}), \dots, s^{(l)}(x^{(m')})\right) = \frac{s_i^{(l)}(x^{(j)}) - \mu_i^{(l)}\Big(\ s^{(l)}(x^{(1)}), \dots, s^{(l)}(x^{(m')})\Big)}{\sigma_i^{(l)}\Big(\ s^{(l)}(x^{(1)}), \dots, s^{(l)}(x^{(m')})\Big)}. \tag{24}$$

During training $\mu^{(l)}$ and $\sigma^{(l)}$ are usually sample mean and sample standard deviation over the whole minibatch,

$$\begin{aligned} \mu^{(l)} &= \frac{1}{m'}\sum_{j=1}^{m'} s^{(l)}\left(x^{(j)}\right) \\ \sigma_i^{(l)} &= \sqrt{\delta + \frac{1}{m'}\sum_{j=1}^{m'}\left(s_i^{(l)}\left(x^{(j)}\right) - \mu_i^{(l)}\right)^2} \end{aligned} \tag{25}$$

for some small δ, e.g. $\delta = 10^{-8}$.
At test time we usually use a running average of the values given above evaluated during training time for μ and σ.

3 Architecture of the DCGAN

So far, we have introduced some ideas of deep learning, in this section we will describe how they are put together to create a GAN. We will focus on the architecture of a currently popular version of a GAN, the **Deep Convolutional GAN** or DCGAN.
The description is based on the paper [8] and the code from [9].
The version we will describe is for the use with the *celebA* dataset, a set of more than 200.000 pictures of human faces.

3.1 Model of the DCGAN

We will start with the generator network. It takes a vector of size 100 as input, and uses 5 hidden layers to generate the output from it. As input, the generator is supplied with independent, identically distributed samples from a uniform distribution on $(-1, 1)$. They are passed to the first layer, a fully connected layer with batch normalisation, i.e. just a linear map form $\mathbb{R}^{100}$ to $\mathbb{R}^{1024*4*4}$ where each component of the output vector is normalised to have mean 0 and standard deviation 1. ReLU is applied to it afterwards.
From now on the codomain is considered as the space of 4×4 pictures with 1024 channels.

The next three layers are transposed convolutional layers as described in Section 2.4. That way both the length and width of the pictures are doubled from each layer to the next. The number of channels is halved between any two layers, both effects together causing the number of variables to be doubles from each layer to the next.
The Kernel used for this operation only has a 5×5 set of non-zero values, so that each value for each pixel can only be obtained using a limited number of pixel values from the previous layer.
After each of those layers batch normalisation and the ReLU function are applied.

Now we have a tensor of size $32 \times 32 \times 128$. One more transposed convolutional layer is used to map it to the space with our output dimension, $64 \times 64 \times 3$. Here, 64×64 are the dimensions of the pictures we want to generate, and we have 3 channels because we need red, green and blue values. Note that batch normalisation or the ReLU function is not applied after this layer, because we want the generator to be able to learn to produce pictures with the same mean and standard deviation as our target distribution.

Finally, as the output rule tanh is used, where $\tanh(x) = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$.
This is basically a scaled, shifted version of the sigmoid function and ensures the values are in $(-1, 1)$, representing relative colour values. A linear map is used to map the colour values of the sample pictures to the same interval. This improves the speed of training.

The whole architecture of the generator is shown in Figure 6.

The discriminator has basically the same architecture in reverse order.
The first layer is a convolutional layer with stride 2, where the output is chosen to have 64 channels. That way the $64 \times 64 \times 3$ input is mapped to the space of $32 \times 32 \times 64$ tensors.
Batch normalisation is not used after this layer, but an activation function is used. Leaky ReLU has proven to work well with the discriminator.

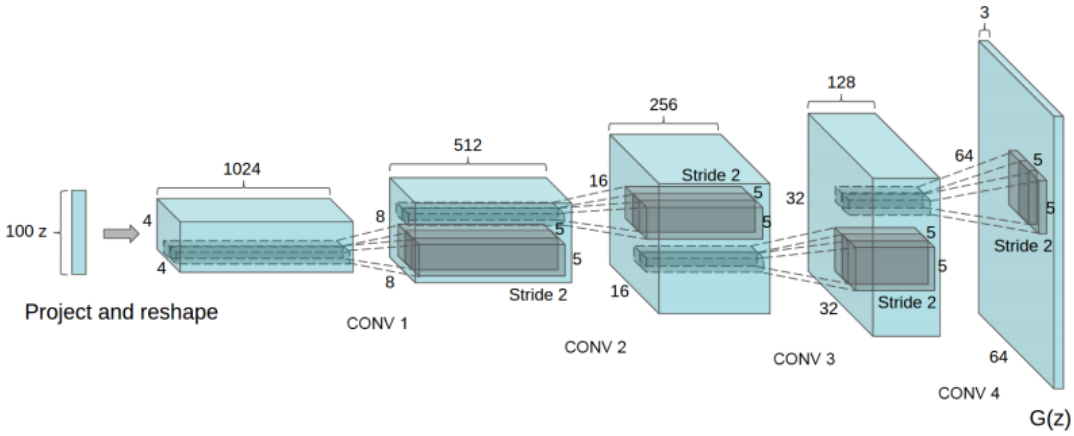

Figure 6: **DCGAN generator** mapping from a 100-dimensional input vector to a 64×64 picture. This image is taken from [8].

This layer is followed by another three convolutional layers, each with stride 2 and doubling the channels from one to the next one. After each of those batch normalisation is applied and leaky ReLU is used as the activation function.

This results in a tensor of dimension $4 \times 4 \times 512$. This is then reshaped into a vector, and a fully connected layer is used to map it to a single value.

Finally, as output rule the sigmoid function is used. This way we end up with a value in $(0, 1)$, which is interpreted as the probability of the input coming from the real data distribution.

3.2 Loss Function of the DCGAN

The next parts we need for our networks are the loss functions.
For the discriminator, as described in the first chapter we want to minimise the following function:

$$- \mathop{\mathbb{E}}_{x \sim \mu_{\text{data}}} \log D(x) - \mathop{\mathbb{E}}_{z \sim \mu_z} \log \big(1 - D(G(z))\big). \tag{26}$$

This is done by using the loss function

$$- \frac{1}{m'} \sum_{j=1}^{m'} \log D\left(x^{(j)}\right) - \frac{1}{m'} \sum_{j=1}^{m'} \log \left(1 - D\left(G\big(z^{(j)}\big)\right)\right), \tag{27}$$

where $x^{(1)}, \dots, x^{(m')}$ are a minibatch of real samples and $z^{(1)}, \dots, z^{(m')}$ are realisations of vectors of $[-1, 1]$ uniform random variables. For this θ_g is treated as a constant.
For the generator, we want (26) to be maximised. So we want to pick our loss function to be

$$\frac{1}{m'} \sum_{j=1}^{m'} \log D\left(x^{(j)}\right) + \frac{1}{m'} \sum_{j=1}^{m'} \log \left(1 - D\left(G\big(z^{(j)}\big)\right)\right). \tag{28}$$

Note that it is not clear that it is possible to use this function as a loss function because of our requirement of it to be differentiable. But we know that we can get the derivative of $D(x)$ with

respect to x by backpropagation, so this whole function is differentiable with respect to $G(z)$, so one is able to use it as the loss function.

To train the two networks the authors of the paper use the Adam optimiser described in section 2.5.1, whereby θ_d and θ_g are updated alternatingly in the process, and the other parameter is always treated as a constant.

4 Own Application of a GAN to Semantic Inpainting

Semantic inpainting is concerned with the task of taking an image with large parts missing, and filling those in a way that makes the whole image look realistic. I have tried to apply the GAN framework to solve this problem. My code (available at [10]) is based on the one available at [9]. The code from [9] basically uses the DCGAN architecture described above with some minor changes. For example, some of the dimensions are slightly different, and there are two updates of θ_g for every update of θ_d.
Furthermore, it includes a separate architecture built for the greyscale images from the *mnist* dataset, which consists of scans of handwritten digits. The main differences in this architecture are that it uses a layer less, and that it supplies additional channels with the labels of the digits at each layer.

The scenario I considered was that there was a bit less than the lower half of a picture, and I wanted to regenerate the full picture. We will use the subscript 0 to denote that lower half, for example we will write $x_0^{(j)}$ for that lower part of $x^{(j)}$.

4.1 Alternated Model and Loss Function

In my first attempt I changed the generator to take the pixel values of x_0 as input further to z. I also changed the first layer to be a fully connected layer from both z and x_0 to the same tensor as in the original version. Furthermore, I used the neural network only to generate the upper half of a picture, and then added x_0 afterwards and provided the whole picture to the discriminator.

This architecture managed to generate some pleasing results for the *mnist* dataset, some are shown in Figure 7.

Figure 7: **DCGAN where the generator is forced to use x_0.**
Image was generated running my code as: `main.py --dataset mnist --input_height=28 --output_height=28 --epoch=10 --train --gen_use_img`

For the *celebA* dataset the generator managed to improve for a bit at the start. But after the discriminator adjusted to that and managed to distinguish those from real samples, the generator did not seem flexible to change, and it started generating deformed pictures. See Figure 8 for an example.

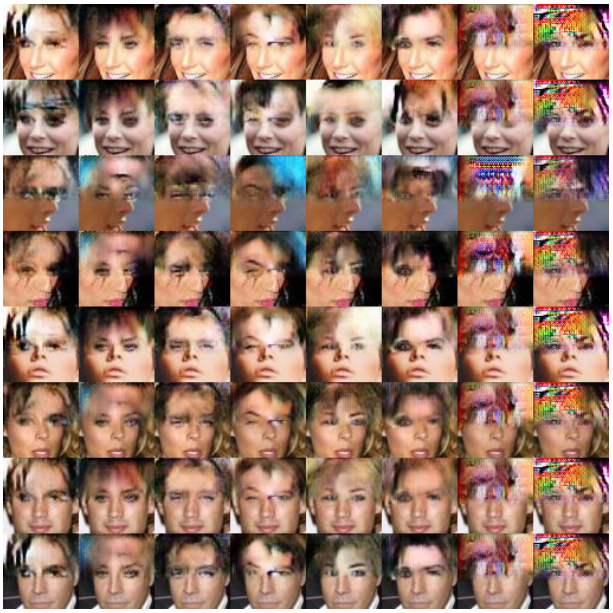

Figure 8: Images created by the generator with the settings described above. All pictures in a line are produced with the same input. The columns are (from left to right) the output of the generator after $1, \dots, 8$ epochs of training.
One can see some improvement at the start, but eventually the discriminator becomes too good, and the generator starts to produce deformed pictures.
Image was generated running my code as: `main.py --dataset celebA --input_height=108 --crop --epoch=8 --train --gen_use_img`

After trying out multiple ideas I ended up with the following version: To make the generator more flexible, I let the generator create the full picture. Then, as the loss function I use a sum of two different parts: I use the discriminator to encourage generation of real looking pictures. To make sure the generator creates pictures similar to the input, I also use the L_1-loss between the pixel values of the generated image and the ones of the supplied part of the real picture. So the new loss function for the generator is given by

$$\frac{1}{m'} \sum_{j=1}^{m'} \left(\log D\left(x^{(j)}\right) + \log\left(1 - D\left(G\left(x_0^{(j)}, z^{(j)}\right)\right)\right) + \lambda \overline{L_1}\left(G\left(x_0^{(j)}, z^{(j)}\right)_0 - x_0\right) \right) \tag{29}$$

where $\overline{L_1}$ stands for the average absolute value, taken over all pixels and channels. The hyperparameter λ is used to balance the two losses, in my experiments $\lambda = 100$ seemed to work well.
Note that I use the picture x_0 is taken from also as the input for D.

With these settings the generator seemed to be flexible enough to adjust to changes of the discriminator without losing too much of the progress made earlier.
Examples can be found in Figure 9 and Figure 10.

Figure 9: **Semantic Inpainting on samples of the *mnist* dataset.** The images were coloured after generation for visualisation. **Green:** original digits **Blue:** The part of the picture where the generator had the real values. **White:** Part added by the generator. Image was generated running my code as: `main.py --dataset mnist --input_height=28 --output_height=28 --epoch=10 --train`

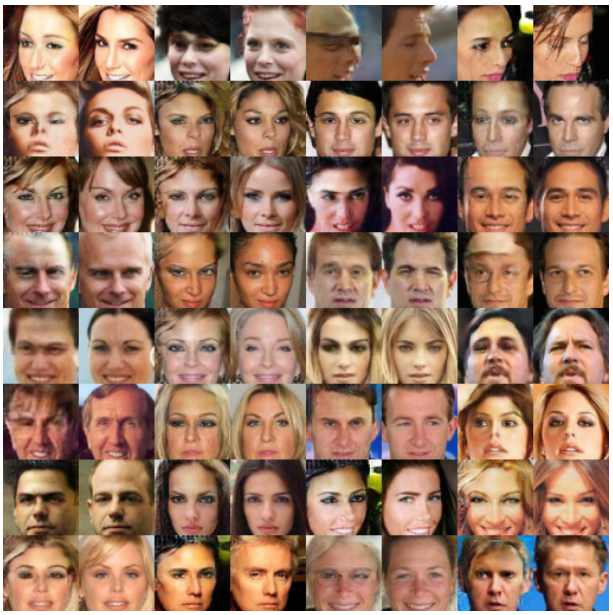

Figure 10: **Semantic Inpainting on samples of the *celebA* dataset.** The odd numbered columns are pictures generated by the GAN, the even ones show the pictures that gave the input. All of these pictures are 64×64 pixels in size, and the lowest 24 rows were supplied to the generator.
Image was generated running my code as: `main.py --dataset celebA --input_height=108 --crop --epoch=8 --train`

Note that in these figures we have shown the pictures as they were generated by the generator. Technically this is not really inpainting, as we do not match the input perfectly on the part supplied. But, in contrast to training, where we want to pass the whole image from the generator to the discriminator, at test time we can just replace parts of the generated picture with x_0. With the right choice of λ the generator will have reduced the L_1-loss enough so that this merged picture looks natural and the transitions nearly vanish. Figure 11 gives examples of those merged pictures.

Figure 11: **Generated pictures merged with given parts.** The pictures still look natural and it is hard to see transitions between supplied and generated part.
Image was generated running my code as: `main.py --dataset celebA --input_height=108 --crop --epoch=8 --train`

It is important to note that this method does not try to reproduce the full picture from the input. It just tries to find a possible picture that matches x_0 on this part.

In fact, we will show that under ideal conditions, for every possible x_0 and an optimal G, we have that $G(x_0, \cdot)$ exactly reproduces the distribution of μ_{data} restricted to pictures that contain x_0.

4.2 Global Optimal G under Ideal Conditions

Denote the distribution of μ_{data} restricted to pictures that contain x_0 by $\mu_{\text{data}|x_0}$ and also write $\mu_{\text{g}(x_0)}$ for the distribution of $G(x_0, z)$ when $z \sim \mu_z$. Then the following theorem holds.

Theorem 2. *Define*

$$V(G,D) = \mathop{\mathbb{E}}_{x\sim\mu_{data}} \left[\log\left(D(x)\right)\right] + \mathop{\mathbb{E}}_{x\sim\mu_{data}} \mathop{\mathbb{E}}_{z\sim\mu_z} \left[\log\left(1 - D(G(x_0,z))\right)\right], \tag{30}$$

Write D_G^* for the function that maximises $V(G,D)$ for fixed G. Then, for any $\lambda > 0$, the function that minimises

$$\mathcal{L}(G) := V(G, D_G^*) + \lambda \mathop{\mathbb{E}}_{x\sim\mu_{data}} \mathop{\mathbb{E}}_{z\sim\mu_z} \overline{L_1}\left(G\big(x_0,z\big)_0 - x_0\right) \tag{31}$$

fulfils $\mu_{g(x_0)} = \mu_{data|x_0}$ for all possible x_0.

Proof. Write μ_{g} again for the distribution of $G(x_0, z)$, this time for $x_0 \sim \mu_{\mathrm{data}}$ and $z \sim \mu_z$. Then, as in Proposition 1, we get

$$D_G^*(x) = \frac{p_{\mathrm{data}}(x)}{p_{\mathrm{data}}(x) + p_{\mathrm{g}}(x)}. \tag{32}$$

For the optimisation of G, note that $\mathcal{L}(G)$ consists of two parts.
The first part, $V(G, D_G^*)$ is minimised if and only if $\mu_{\mathrm{g}} = \mu_{\mathrm{data}}$ by the same proof as in the first chapter.
The second part is minimised if and only if $G(x_0, z)_0 = x_0$ for all possible x_0, this means that μ_{g} restricted to pictures that contain x_0 is exactly the same distribution as $\mu_{\mathrm{g}(x_0)}$, the distribution of outputs of $G(x_0, z)$ when $z \sim \mu_z$.

It follows that, if we want both conditions above to be fulfilled, we need that

$$\mu_{\mathrm{g}(x_0)} = \mu_{\mathrm{data}|x_0} \tag{33}$$

for almost every x.

In fact, this is also a sufficient condition for both conditions to hold, and therefore sufficient for G to be optimal: It is clear that the second condition follows from (33), and we will further show that it implies $\mu_{\mathrm{g}} = \mu_{\mathrm{data}}$:

$$\begin{aligned} p_{\mathrm{g}}(x') &= \int_x p_{\mathrm{g}(x_0)}(x'|x_0) p_{\mathrm{g}}(x_0) dx \\ &= \int_x p_{\mathrm{data}|x_0}(x'|x_0) p_{\mathrm{g}}(x_0) dx \\ &= p_{\mathrm{data}}(x'). \end{aligned} \tag{34}$$

□

Not that (34) only works because the distribution used for the input of G is the same as the one used to provide real samples to D.

Although this result does not apply directly, as we can not use the expectation over μ_{data} in our loss function, it is promising. It encourages us to hope that the generator generates the whole distribution of possible inpaintings, and not just one sample of this distribution.

In my experiments, this does indeed work very well with the *mnist* dataset, examples are shown in Figure 12, where we can see $G(x_0, z)$ for z split into two component vectors randomly, and one of them varied (between -1 and 1) horizontally and the other one vertically.

With the *celebA* dataset, the same setup did barely show any variation. I assume this was because x_0 has much more components in that case, as the image resolution is bigger and in addition we have 3 channels to work with, so the influence of z compared to x_0 is just too small. But I reduced the height of x_0 and increase the dimension of z to get sizes of similar order of magnitude, and with that there was definitely some improvement. A selected good example can be seen in Figure 13, with z varied in the same way as described above.

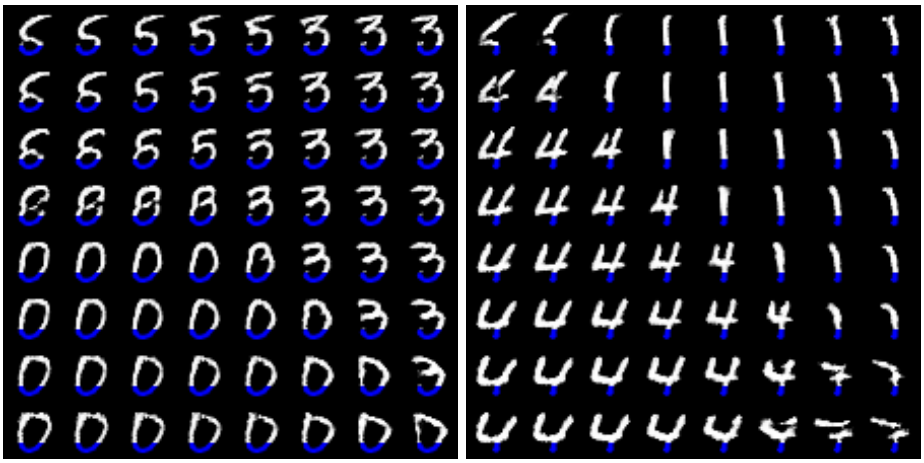

Figure 12: $G(x_0, z)$ **with** x_0 **fixed and** z **varied.** One can see that G generates multiple different digits from one given input depending on the value of z.
Image was generated running my code as: `main.py --dataset mnist --input_height=28 --output_height=28 --epoch=25 --train`

Figure 13: $G(x_0, z)$ **with** x_0 **fixed and** z **varied.** Here x_0 was marked for visualisation.
One can see both the haircut and the eyes changing with z.
Image was generated running my code as: `main.py --dataset celebA --input_height=108 --crop --epoch=8 --train --z_dim=4000 --img_height=12`

4.3 GAN Setting Compared to a Network with a Loss Function

In this section we will compare our approach to the more standard approach of using a neural network together with some loss function for the task of inpainting.
More precisely, we use a network identical to the generator to generate pictures, but then instead of using the discriminator for the loss function, we compare the output to the picture that has provided the input, and use the L_1 norm of their difference as the loss function.

Here, in my opinion, the main advantage of our method is that it creates a complete distribution of pictures, as we have shown above.

Another difference between the two approaches is that the pictures generated with just a loss function were usually blurry when used with the *celebA* dataset, which can be seen in Figure 14. This ability of GANs to produce sharp pictures is generally considered a big advantage of them compared to other methods.

A considerable upside of using the loss function was that the generator made all its progress much quicker, usually within 1 epoch. So training is much less costly with this version.
Finally, as expected there was very little variation with z.

Figure 14: **Generator with just L_1 loss.** One can see good results, but the generated upper half appears blurry in most pictures.
Image was generated running my code as: `main.py --dataset celebA --input_height=108 --crop --epoch=8 --train --drop_discriminator`

Summarising, the method I used gave some promising results, in my opinion especially obtaining a whole distribution can definitely be useful. Even when the aim is just to produce one picture, this can help, as one would be able to pick a value of z that makes the picture look as close to real as possible (either from a human's point of view, but it would also be possible to automate that using the discriminator.)

The main drawback of this method is though that the missing parts of the picture need to be known in advance of training. If we, for example wanted to generate a digit knowing the upper half of it, we would have to train a whole new model for exclusively that task.

There do exist methods which do not depend on this, where we only need to train the model once, and then we can inpaint pictures with arbitrary missing parts, and I will describe one of them in the next chapter.

5 More Advanced Semantic Image Inpainting

In this chapter we will introduce a more flexible algorithm to do semantic inpainting with a GAN. The method we will describe was introduced in paper [11].
The idea behind it is that we first train some kind of GAN, for example the DCGAN introduced in Section 3. Then, given some incomplete picture y we want to do image inpainting on, we try to find a vector $\hat{z}$ such that $G(\hat{z})$ is similar to y on the given parts of the picture. At the same time, we also want $G(\hat{z})$ to look real, at least from the point of view of the discriminator. As ideally G produces the whole distribution of e.g. human faces, we hope that we will be able to find some $\hat{z}$ such that $G(\hat{z})$ is very realistic and at the same time very close to our input. As a last step, we merge y and $G(\hat{z})$ to generate our final result.

One big advantage of this model compared to other currently used ones is that it uses not only the given part of the image to construct the output, but we also code information about what a human face looks like into the model by training the GAN on the corresponding data before.

Another advantage is that we do not need to know which part of the image is missing ahead of training. Also, we can use the model to inpaint images with different parts missing without having to train the model again.

5.1 Architecture of this Method

As mentioned above the first thing we need is a trained GAN, that means we have found θ_g and θ_d such that G produces pictures that look realistic, and we have the function D that can determine the probability that a picture is real. We will take θ_g and θ_d as fixed from now on, and will not change them for the rest of the algorithm. To make notation easier, for $d_1 \times d_2$ the dimension of the input we define the mask $M \in \{0,1\}^{d_1 \times d_2}$ to be a matrix of the same size, where $M_{ij} = 1$ if we know the value of that pixel, and $M_{ij} = 0$ if we want to generate it. Then, for a corrupted picture y, we will try to find $\hat{z}$ such that $M \odot y + (1 - M) \odot G(\hat{z})$ gives us a good reconstruction of the image, where $\odot$ denotes elementwise multiplication.

To find a good vector $\hat{z}$ we use a loss function with two different parts. The first one is a contextual loss making sure $G(\hat{z})$ and y are similar on the pixels we know about. One version possible is

$$||M \odot y + (1 - M) \odot G(\hat{z})||_1, \tag{35}$$

where $|| \cdot ||_1$ is the L_1 norm, with the sum taken over all pixels and channels.
This version is used in the code from [12] that I will be using to present some results.

Another option, which is used in the paper, is to weight this loss function, where the weight for pixel i is coming from the ratio of unknown pixels in a given neighbourhood $N(i)$.
It is inspired by the fact that one wants the pictures to be similar in areas with missing pixels to be able to fill them in well, and cares less about having the same background for example. For this option, the weighted mask W is defined by

$$W_i = M_i \sum_{j \in N(i)} \frac{1 - M_j}{|N(i)|} \tag{36}$$

for all pixel indices i.
Then, the contextual loss function is taken to be

$$\mathcal{L}_c(z; y) := ||W \odot G(z) - W \odot y||_1. \tag{37}$$

The second part of the loss function, the prior loss, is there to penalise unrealistic pictures. The discriminator is used for this:

$$\mathcal{L}_p(z) := \log\big(1 - D(G(z))\big). \tag{38}$$

The loss function is then defined to be

$$\mathcal{L}(z; y) = \mathcal{L}_c(z; y) + \lambda \mathcal{L}_p(z) \tag{39}$$

for some $\lambda \in \mathbb{R}^+$ to balance the two parts.
Note that the loss function is a composition of differentiable functions, so by backpropagation it is possible to obtain $\frac{\partial}{\partial z}\mathcal{L}(z; y)$. Knowing this derivative then makes it possible to use gradient descent based algorithms to find a good value of z. In the paper the authors use 1500 iterations of the Adam optimiser described in chapter 2.5.1, with z restricted to $[-1, 1]$.

5.2 Poisson Blending

As the colours of $G(\hat{z})$ and y tend to be slightly different, often the transitions between the two are visible in the merged picture. Poisson blending can be used to hide them.
The idea behind poisson blending is not to use the pixel values from $G(\hat{z})$ to complete y, but to use the gradient instead.
More precisely, it is given by

$$\hat{x} = \arg\min_x ||\nabla x - \nabla G(\hat{z})||_2^2 \qquad \text{such that } x_i = y_i \text{ when } M_i = 1 \tag{40}$$

where ∇ is the discrete Laplacian operator, given by

$$(\nabla x)_{i,j} = 4x_{i,j} - x_{i+1,j} - x_{i-1,j} - x_{i,j+1} - x_{i,j-1} \tag{41}$$

Applying this will make the results look better, and it is harder to tell they were constructed merging two different pictures.

5.3 Results

I have used the code from [12] with some changes to show some results using this method. The changes contained adding another mask and a visualisation of the progress finding $\hat{z}$. Poisson blending was not used.
The first mask I have used hides a square in the centre of the picture. I trained the underlying GAN for 8 epochs, and then did 500 steps of optimising $\hat{z}$. The results can be seen in Figure 15.
Also, the process of finding the right $\hat{z}$ can be seen in Figure 16.

Furthermore, I also used this algorithm with the same input as in my own version. The resulting pictures can be found in Figure 17.

Figure 15: **Semantic Inpainting with a centre mask.** The parts inserted are very similar in terms of shape and colour, but the transitions are still visible.

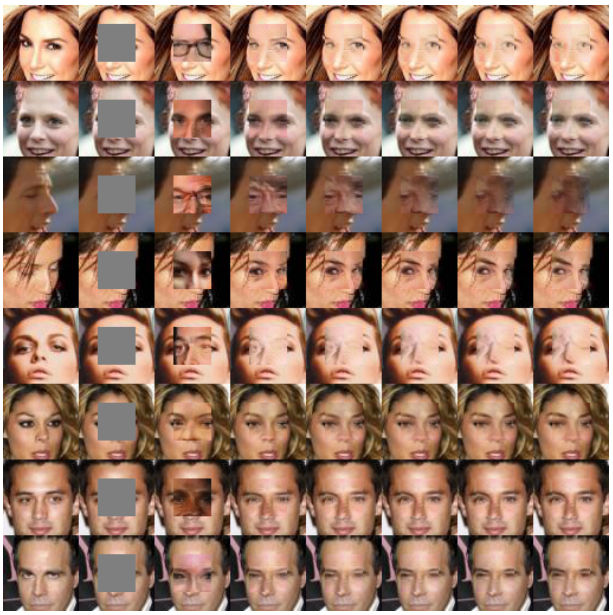

Figure 16: **Finding a good value of $\hat{z}$.** The first two columns give the original picture and the part that was used for semantic inpainting. The third column gives the result with a random vector $\hat{z}$, and the following columns give $G(\hat{z})$ after $100, 200, \ldots$ steps of improving $\hat{z}$.

Figure 17: **Semantic Inpainting with a top mask.** The underlying GAN was trained for 8 epochs and 800 steps were used for optimising $\hat{z}$. The mask was hiding about two-thirds of the picture from top.

6 Conclusion

In this essay I have described two methods that use a GAN to do inpainting. Both deliver in my opinion decent results. The results are compared in Figure 18.

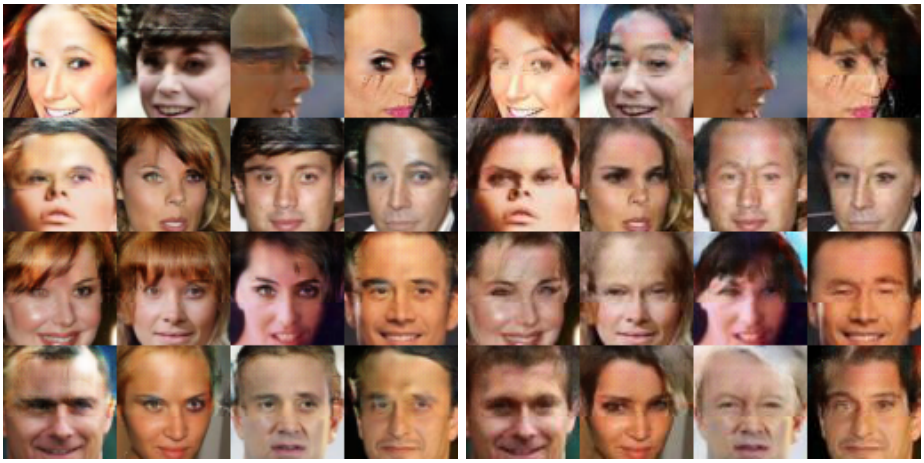

Figure 18: **Results from the two methods compared.** Results from the method from Section 4 are shown on the left, the ones from the method from [11] on the right.
The transitions are less visible on the left, and in my opinion these pictures also look slightly more realistic.

Although the method I used produces promising results, it has a few drawbacks.

One big drawback is that we need to know the missing parts ahead of training.

Furthermore, in the form I have used the method it does not scale very well. This is because we use the information about all the known pixels of the picture in the first layer, and this layer is fully connected. This results in needing a great many of parameters for this layer. This can be quite expensive memory-wise.
A solution to that could be to reduce the dimension of the input before supplying it to the GAN. For example, we could add a few convolutional layers to the generator to reduce the dimension of the non-random input, and then take the output vector of that layers together with a random vector for the rest of the GAN.

Remarkably, a similar setup to the one I used can be applied to solve a lot of other inverse problems as well:
Suppose we know how to corrupt pictures (write corruption as $x \to x'$), and we want to train a network to learn how to create realistic, uncorrupted pictures from corrupted ones. Then we can use a GAN, and supply the generator with corrupted versions of pictures x' to get some output, say y. Then, we base the part of the loss function of the generator that does not depend on the discriminator on the difference of x' and y', the input and the corrupted version of the output of the generator. We use the standard discriminator, and with that setup we can hope that the generator learns how to invert the corruption process.

The results in Section 4 show that this is possible, for corruption taken to be removing the upper half of a picture. Furthermore, I have alternated my code to show the same concept works for another kind of corruption: In this version I corrupt pictures by turning them into black and white versions. That way, the generator learns how to colour black and white pictures, the result can be seen in Figure 19.

Figure 19: **Colouring using a GAN.** Odd columns show the original, and even columns show the version coloured by the generator that took a black and white version as input.
Image was generated running my code from [10]/code_GAN_colouring for 1 epoch.

This suggests that the method I have used can prove useful in various scenarios, and has a great potential.

Acknowledgement

I would like to thank Dr Sergio Bacallado for setting the essay and providing the main references as well as giving helpful feedback during the writing process and afterwards.

References

[1] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. eprint: arXiv:1406.2661.

[2] Ian Goodfellow. *NIPS 2016 Tutorial: Generative Adversarial Networks*. 2016. eprint: arXiv:1701.00160.

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

[4] Dr Tengyao Wang. *Statistical Learning in Practice*. http://www.statslab.cam.ac.uk/~tw389/teaching/SLP18/notes.pdf. 2018.

[5] Jérôme Lelong. "A Central Limit Theorem for Robbins Monro Algorithms with Projections". In: 2005.

[6] Vincent Dumoulin and Francesco Visin. "A guide to convolution arithmetic for deep learning". In: *ArXiv e-prints* (Mar. 2016). Github: https://github.com/vdumoulin/conv_arithmetic. eprint: 1603.07285.

[7] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. eprint: arXiv:1412.6980.

[8] Alec Radford, Luke Metz, and Soumith Chintala. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. 2015. eprint: arXiv:1511.06434.

[9] Kim Taehoon. *DCGAN in Tensorflow*. https://github.com/carpedm20/DCGAN-tensorflow.

[10] Myself. *Semantic Inpainting using a GAN*. https://github.com/11BP11/Semantic_Inpainting_using_a_GAN.

[11] Raymond A. Yeh et al. *Semantic Image Inpainting with Deep Generative Models*. 2016. eprint: arXiv:1607.07539.

[12] Brandon Amos. *Image Completion with Deep Learning in TensorFlow*. http://bamos.github.io/2016/08/09/deep-completion. Github: https://github.com/bamos/dcgan-completion.tensorflow.