Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

MASTER THESIS

A Study of Generative Adversarial Networks

Florian Krach
Advisor: Prof. Josef Teichmann

Department of Mathematics
ETH Zürich

March 20, 2019

Abstract

In this thesis we study generative adversarial networks (GANs). In the first part, the standard GAN and many other GAN variants are reviewed in detail. The focus lies on two classes, GANs with optimal transport loss functions and GANs with a different more sophisticated structure that extends the standard generator-discriminator design. In the second part, a new GAN variant is presented, which is an extension of the *sliced Wasserstein generator* (SWG) [22]. A theoretical justification for our model is provided and extensive tests are carried out to compare it with SWG. In the last part, GANs with fixed random or semi-random weights are studied. We come to the conclusion that the choice of the architecture is more important than the training of the weights and that training only 0.4% of all weights is enough to learn parts of the target distribution.

Contents

Acknowledgement **4**

1 Introduction **5**

2 Generative Adversarial Networks **7**
2.1 Standard GAN 7
2.2 Problems in training of GANs 9
2.3 Different OT loss functions for GAN training 11
2.3.1 Distances and divergences between probability measures 11
2.3.2 Application in GANs 13
2.4 Different architectures for GANs 20
2.4.1 Creating more structure on the latent space 21
2.4.2 Other architectural modifications to avoid mode collapse 27
2.4.3 Stacked generator functions 31
2.5 Comparison of different GAN variants 36

3 Combining the Johnson-Lindenstrauss Lemma with the SWG **40**
3.1 Johnson-Lindenstrauss Lemma 40
3.2 The generative model 41
3.2.1 JL-SWG 42
3.2.2 JL-SWGAN 46
3.3 Experimental Results 49
3.3.1 Toy dataset 50
3.3.2 JL-SWG on MNIST 52
3.3.3 JL-SWG on CIFAR-10 and CelebA 54
3.3.4 JL-SWGAN on MNIST 58
3.3.5 JL-SWGAN on CIFAR-10 59
3.3.6 JL-SWGAN on CelebA 63
3.4 Applying the Johnson-Lindenstrauss Lemma somewhere else . 67
3.5 Conclusion 68

4 GANs with fixed weights **70**
4.1 Overview of NN with random weights 70
4.2 Why considering GANs with random weights 71
4.3 Experimental Results 73
4.3.1 GANs with random weights 75
4.3.2 GANs with perturbed weights 78
4.3.3 Pre-training GANs on a subset of the data 81
4.4 Conclusion 84

References **85**

Appendices **92**

A JL-SWGAN architectures **92**

B DCGAN architectures for MNIST **93**

C Time improvement of GANs with random weights **93**

D Layer statistics and distributions of trained DCGAN **95**

Declaration of Originality **102**

Acknowledgement

I want to thank my advisor Professor Josef Teichmann for spending so much time on the numerous fruitful discussions we had. Furthermore, I thank him for pointing me in the right direction whenever I needed it, but giving me also the time and the academic freedom to delve into the subjects which were most interesting for me. It was a true pleasure working together with him.
Furthermore, I want to thank my parents for their unconditional support throughout my studies. They have been there for me whenever I needed them.
Finally, I want to thank Alina for being by my side and pulling me out of the "maths world" whenever it is necessary.

1 Introduction

In the past years the field of generative modelling has grown enormously. This is a dicipline of machine learning concerned with training a model that is able to generate data similar to given real world data (e.g. images, videos, audio, text). Especially in image generation the learning techniques are so advanced now that, for example, synthetic images of faces can't be distinguished from true images anymore [46].

Beside others, there are two very important approaches for deep generative models: *Generative Adversarial Networks* (GAN) and *Variational Autoencoders* (VAE). VAEs, first introduced in 2013 in [48], are a deep learning modification of autoencoders, which have been known (also under different names) in machine learning literature for at least 30 years (according to [69] autoencoders were proposed in [10]). The concept of GANs, on the other side, is still pretty new, as it was first introduced in 2014 in [30].
The difference between them, in a nutshell, is the following. In generative adversarial networks a deep neural network (DNN), called *generator* G, is trained to map random noise vectors Z to $G(Z)$ lying in the same space as the given (real world) data samples. The training goal is that the distribution of $G(Z)$ approaches the given data distribution. To make this possible, most of the GANs use a second adversarial deep network, called *discriminator* D, which is simultaneously trained to distinguish given data from generated samples.
On the contrary, variational autoencoders consist of an *encoder* E and a *decoder* D. The encoder maps given data samples, x, to a lower dimensional space "of codes" $\mathcal{Z}$ and is trained to match a given distribution Q on $\mathcal{Z}$. The decoder maps from $\mathcal{Z}$ back to the space in which the real samples live. Both, E and D, are trained to minimize the reconstruction error between x and $D(E(x))$. To generate new data with an VAE, one draws random samples on $\mathcal{Z}$ from the distribution Q and uses them as input for the decoder.

Both methods have their strengths and weaknesses. Generative adversarial networks can, when applied to image generation tasks, generate sharp images of high quality, but their training tends to be unstable [8, 9]. Furthermore, when the given dataset is highly diverse, mode collapse is a major problem in GAN training [9].
On the other hand, variational autoencoders are easy to train and usually don't suffer from mode collaps, but mostly produce blurry images [46]. This suggests that in generative modelling there always is a trade-off between sample quality and sample diversity (at least when the model sizes are bounded) as claimed for example in [14].
For both methods there exist many extensions trying to tackle their re-

spective problems without loosing their advantages. For GANs we give an overview of many such variations in Section 2.3 and 2.4. For VAE we list some references for further reading [48, 17, 57, 74, 81, 50, 36, 65]. Some of the extensions also combine GANs and VAEs [52, 76].

In Section 2 we give a more detailed introduction to GANs, together with an in-depth description of many GAN variants. The described GAN versions can be split in two categories, on the one hand GANs with optimal transport loss functions and on the other hand GANs with an structure that extends the standard generator-discriminator design. In Section 3 we introduce a new GAN variant which is an extension of the *sliced Wasserstein generator* (SWG) [22]. Tests on three of the most common datasets for training GANs, MNIST [53], CIFAR-10 [51] and CelebA [56], are conducted to compare our extension with SWG. In Section 4 we study GANs with fixed weights. The weights are either chosen completely randomly or semi-randomly, where some previous knowledge is incorporated in their choice. Only a small amount of all the weights is trained. Furthermore, we perform experiments to test the robustness of the generator against noise perturbations of its weights and show that the knowledge of pre-trained GANs can successfully be reused.

The code of all experiments that are performed in this thesis is available at `https://github.com/FlorianKrach/A-Study-of-GANs.git`.

2 Generative Adversarial Networks

In Section 2.1, we review first the GAN as it was introduced by Goodfellow et al. in [30] and then some changes to this setting, which yield the standard GAN framework used in image generation. Then we have a closer look on the problems that arise in GAN training in Section 2.2. Afterwards, we examine a variety of different methods to tackle these problems, which will be split up in two groups: using optimal transport (OT) for the objective function in Section 2.3 and modifying the model architecture in Section 2.4. We conclude with a comparison of some of the methods in Section 2.5.

2.1 Standard GAN

Let $\mathcal{X}$ be some compact subset of $\mathbb{R}^n$ (in case of images often the flattened space $[0,1]^{f\times h\times w}$), with Σ the set of all Borel sets on $\mathcal{X}$ and $\text{Prob}(\mathcal{X})$ the set of all probability measures on $\mathcal{X}$. Further, let $\mathcal{Z}$, denoted *latent space*, be a finite dimensional normed vector space or a subset of such a space (usually $\mathbb{R}^d$ or $[0,1]^d$ and chosen lower dimensional than $\mathcal{X}$) and $\mathbb{P}_Z$ a fixed probability measure on $\mathcal{Z}$ (usually chosen as multivariate standard normal or uniform distribution). Let $\mathbb{X} \subset \mathcal{X}$ be a given set of training data which is distributed according to $\mathbb{P}_r \in \text{Prob}(\mathcal{X})$. We define the following two maps

$$G : \mathbb{R}^{n_G} \times \mathcal{Z} \to \mathcal{X},\ (\theta, z) \mapsto G(\theta, z) =: G_\theta(z)$$
$$D : \mathbb{R}^{n_D} \times \mathcal{X} \to \mathbb{R},\ (\omega, x) \mapsto G(\omega, x) =: D_\omega(x)$$

and denote G_θ as the *generator* and D_ω as the *discriminator*. $\theta \in \mathbb{R}^{n_G}$ and $\omega \in \mathbb{R}^{n_D}$ are interpreted as parameters of the functions, where n_G and n_D are the number of parameters of the generator and discriminator respectively. We assume that for every fixed $z \in \mathcal{Z}$ and every fixed $x \in \mathcal{X}$ the maps $\theta \mapsto G_\theta(z)$ and $\omega \mapsto D_\omega(x)$ are continuous and differentiable almost everywhere. This assumption is needed, since we want to apply stochastic gradient descent (SGD) to learn the parameters θ and ω. Neural networks with standard activation functions satisfy this assumption. With G_θ we can define the pushforward probability measure $\mathbb{P}_G := G_\theta \# \mathbb{P}_Z$ which is the distribution of the generated data.
Goodfellow et al. [30] define G_θ and D_ω as multilayer perceptrons (feed-forward artificial neural networks with fully connected layers and non-linear activation functions) for which θ and ω are their respective trainable weights. Moreover D_ω is constrained to map to $[0,1]$, which we interpret as the probability for some $x \in \mathcal{X}$ to come from the given data $\mathbb{X}$ and not from the generator. Hence, D_ω is trained to correctly assign labels for both, samples from the data and generated samples. Informally, that is to maximize

$\log(D_\omega(x)) + \log(1 - D_\omega(G_\theta(z)))$, -1 times the binary cross entropy loss of the respective probabilities. On the other hand, we want G_θ to produce meaningful samples. Assuming D_ω is a good discriminator, this means for G_θ to be able to fool D_ω. For that reason, we train G_θ with the objective to minimize $\log(1 - D_\omega(G_\theta(z)))$. Put together, we arrive at the following two-player minimax zero-sum game

$$\min_\theta \max_\omega \mathbb{E}_{X\sim\mathbb{P}_r}\left[\log(D_\omega(X))\right] + \mathbb{E}_{Z\sim\mathbb{P}_Z}\left[\log(1 - D_\omega(G_\theta(Z)))\right]. \tag{2.1}$$

A *two-person zero-sum game* [3] is a game in which two players try to maximize their pay-off, whereby the pay-off of one player is always given as -1 times the pay-off of the other player. Hence, the sum of the two pay-offs is zero for every possible combination of playing-strategies of the two opponents. Given the pay-off function $H(a, b)$ of player one, which depends on the playing-strategies a and b of player one and two respectively, a mathematically equivalent description of the game is that player one tries to maximize H, while player two aims at minimize it. A possible playing-strategy of player two is to minimize his possible maximum loss, which can be written as $\min_b \max_a H(a, b)$. This is the so called *minimax* strategy, which leads to an optimal solution if $\min_b \max_a H(a, b) = \max_a \min_b H(a, b)$ holds [2]. John von Neumann and Oskar Morgenstern proved that this equality holds whenever a saddle point of H exists [78, 13.5]. In 2.1, we can interpret the generator is player two and the discriminator is player one with pay-off function H given by the sum of the expectations. In line with this, we will sometimes refer to them as players. It was shown in [30] that the minimax solution for the generator is to perfectly reproduce the distribution $\mathbb{P}_r$.

In practice, the expectations in 2.1 are approximated by means over mini-batches (in the following we simply denote them as batches) of training data and generated samples respectively. Recall that sampling from $\mathbb{P}_G$ is done by sampling from $\mathbb{P}_Z$ and applying G_θ. Thereby, (2.1) implies a tractable objective function for both neural networks, which can be optimized by SGD using backpropagation.

Since training of D_ω to completion in the inner maximization problem would be computationally intractable and might also lead to overfitting, the training is done iteratively, where one step of optimizing G_θ is followed by n_{disc} steps of optimizing D_ω. The number n_{disc} can be chosen bigger 1 to yield a more acceptable approximation of the optimal ω before updating G_θ. Although not common, it is also possible to update the generator multiple times before updating the discriminator.

When D_ω is able to clearly separate generated samples from training data (this happens especially in the beginning of the training where G_θ is still

poor), $\log(1 - D_\omega(G_\theta(z)))$ might saturate and not give sufficient gradients for training. To prevent this, the objective of G_θ is changed to minimize $-\log(D_\omega(G_\theta(z)))$ instead. This is sometimes referred to as "-log D trick" and results in the same minimax solution as (2.1). We did now arrive at the original GAN formulation of [30].

The most important application of GANs are image tasks. It is common practice in deep learning to use convolutional neural networks (CNN) (a review of CNNs for image classification can be found in [64]) for visual tasks [70]. Hence, it seems natural to modify the network architectures of G_θ and D_ω to use convolutional layers instead of fully connected layers, yielding *deep convolutional generative adversarial networks* (DCGAN) introduced in [63]. For the generator only the first layer and for the discriminator only the last layer are kept fully connected. All other layers are chosen as transposed convolutions for G_θ and standard convolutions for D_ω. Upsampling respectively downsampling is done by convolutions with stride 2 instead of using pooling layers. Beside the convolutions, the architecture includes batch normalization [44] and ReLU activation functions for the generator respectively Leaky-ReLU activation functions for the discriminator in all but the last layer (see Definition A.1 in Appendix A). The last layer of the generator has a tanh respectively sigmoid activation function (depending on the target space), while the discriminator always uses a sigmoid function to map to $[0, 1]$. We also mention the standard choice $n_{\text{disc}} = 1$ for DCGAN. In the following we will use DCGAN as our standard GAN framework.

2.2 Problems in training of GANs

There are many known problems that arise during the training of GANs. First of all, the training process is unstable, meaning that one has to keep a careful balance between the training of generator and discriminator. Training the discriminator too well, especially in the beginning of the training, typically results in vanishing gradients if the -log D trick is not used or in very unstable updates of the generator if it is used. This was studied extensively in [8]. Either way, the updates of the generator do not improve or even decrease the sample quality.
Moreover, GANs are not very robust to varying architectures as shown, for example, in [9]. While the DCGAN framework usually is able to produce good results, the training gets much worse, or even ineffective, as soon as one leaves away batch normalization, does not increase the number of feature maps or changes (back) to multilayer perceptrons (instead of convolutions).

When training GANs with the standard objective (2.1), we also face the difficulty that the loss function does not converge, even if the training is

successful. Among other things, this problem was studied in [9]. Of course this is not such a big problem if one only wants to train one generator. But when studying different types of GANs with the goal of comparing them, it often turns out that one has to eyeball lots of generated images to be able to draw a conclusion [9]. There are alternatives, like the *Inception Score* (IS) [66], the *Fréchet Inception Distance* (FID) [37] or for high resolution image generation the recently introduced *multi-scale statistical similarity* [46]. They try to automate human judgement, i.e. evaluate the variety and quality of the images. But they come at the cost of additional computational effort and all have individual downsides.

Another problem that arises especially when the given training dataset is highly diverse, is mode collapse. A mode is a cluster point in the data space, which is more or less well separated from other such points. Assuming that the data distribution $\mathbb{P}_r$ admits a density function, one could equivalently say that the modes are the local maxima of this function. Mode collapse means, that the generative model only learns a few out of many modes that are present in the dataset. Or even worse, it is too confused by very diverse training samples, that it learns some meaningless interpolation of the modes. To illustrate it with an example (taken from [75]), think of a dataset consisting of unlabelled images of 10 different types of animals (like fish, dogs, birds, spiders, bees, etcetera). Then the model might only learn to produce images of dogs. This is not what we want it to do, but still preferable to producing only noise, which might also happen. A natural remedy would be to split up the dataset and train for each animal category another GAN, but since the data usually is unlabelled this is not straight forward. To make matters worse, the diversity of the dataset can be twofold. On the one hand there is inter-class diversity (here between the different types of animals), but there can also be intra-class diversity, e.g. that the images of birds are very diverse [14].
Mode collapse is a very present problem that can already occur on synthetic toy datasets. In [59], the authors sample training data from a 2 dimensional mixture of 8 Gaussian distributions and experimentally show that the standard GAN undergoes mode collapse on this (easy) dataset (see Figure 1).

Although not as severe as for variational auto encoders, GANs also have the problem that the sample quality is often not as good as desired. Especially when it comes down to high resolution image generation, the standard GAN framework often fails. This is due to high resolution making it easier for the discriminator to tell generated images apart from training samples [46], which leads to the same problem as training the discriminator too well.

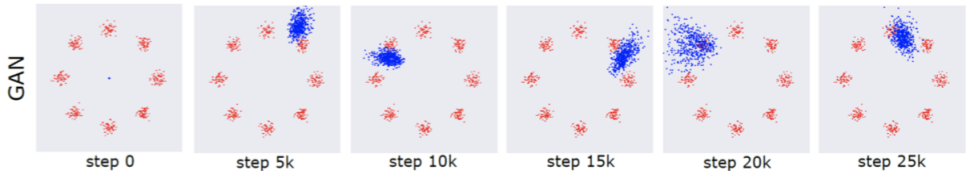


Figure 1: Mode collapse on toy dataset (source [59]). *Red: training dataset, blue: samples generated by GAN after respective training time.*

2.3 Different OT loss functions for GAN training

In this section we focus on modifying the objective function in the GAN formulation, mainly to address the problems of stability of training and non-converging loss functions. First we review some important distances and divergences between probability measures (Section 2.3.1). Then we establish a connection to different objective functions for GANs, which yield different GAN variants (Section 2.3.2).

2.3.1 Distances and divergences between probability measures

Given two probability measures P and Q in $\mathrm{Prob}(\mathcal{X})$ one can define the *Kullback-Leibler divergence* (KL) as

$$\mathrm{KL}(P\|Q) := \int_{\mathcal{X}} \log\left(\frac{dP}{dQ}\right) dP,$$

when P is absolutely continuous w.r.t Q ($P \ll Q$) and as $+\infty$ otherwise. Notice that the Kullback-Leibler divergence is not symmetric. Using it, we can define the symmetric *Jensen-Shannon divergence* (JS)

$$\mathrm{JS}(P,Q) := \tfrac{1}{2}\,\mathrm{KL}\left(P \,\|\, \tfrac{1}{2}(P+Q)\right) + \tfrac{1}{2}\,\mathrm{KL}\left(Q \,\|\, \tfrac{1}{2}(P+Q)\right).$$

Furthermore we define for any given cost function $c : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ the *optimal transport problem*

$$\inf_{\pi \in \Pi(P,Q)} \int_{\mathcal{X}\times\mathcal{X}} c(x,y)\, d\pi(x,y), \tag{2.2}$$

where $\Pi(P,Q)$ is the set of all joint probability measures on $\mathrm{Prob}(\mathcal{X} \times \mathcal{X})$ with marginals P and Q respectively. If the cost function is the p-th power of a distance d on $\mathcal{X}$, the p-th root of (2.2) yields the *Wasserstein-p distance*

$$\mathrm{W}_p(P,Q) := \inf_{\pi \in \Pi(P,Q)} \mathbb{E}_{(X,Y)\sim\pi}\left[\, d(X,Y)^p \,\right]^{1/p}. \tag{2.3}$$

As was shown in [77, Definition 6.1], this definition satisfies the properties of a metric. For the case of $p = 1$ we get by the *Kantorovich-Rubinstein duality* [77, Remark 6.5] the equivalence

$$\alpha \, \mathrm{W}_1(P, Q) = \sup_{\|f\|_L \leq \alpha} \mathbb{E}_{X \sim P}\left[f(X)\right] - \mathbb{E}_{Y \sim Q}\left[f(Y)\right], \tag{2.4}$$

where the supremum is taken over all α-Lipschitz functions f.

The *regularized optimal transport problem* is closely related to the optimal transport problem (2.2) and given by

$$\inf_{\pi \in \Pi(P,Q)} \int c(x, y) \, d\pi(x, y) + \epsilon \, \mathrm{KL}(\pi \| P \times Q), \tag{2.5}$$

where ϵ is a non-negative constant and $P \times Q \in \Pi(P, Q)$ is the product probability measure. For $\epsilon = 0$, the problems 2.5 and 2.2 coincide. However, if $\epsilon > 0$, problem 2.5 is strongly convex, hence, it is easier to solve than 2.2 and the optimal solution π_ϵ is unique [25]. Given the optimal solution π_ϵ of 2.5, we define the *regularized optimal transport divergence* as

$$\mathcal{W}_{c,\epsilon}(P, Q) := \int c(x, y) \, d\pi_\epsilon(x, y). \tag{2.6}$$

With this the *Sinkhorn loss* [26] can be defined as

$$\hat{\mathcal{W}}_{c,\epsilon}(P, Q) := 2 \, \mathcal{W}_{c,\epsilon}(P, Q) - \mathcal{W}_{c,\epsilon}(P, P) - \mathcal{W}_{c,\epsilon}(Q, Q). \tag{2.7}$$

The following distance between probability measures has no direct relation to an optimal transport problem, but will be used in combination with the optimal transport distance. Furthermore, its similarity to the Sinkhorn loss is evident. We define the *generalized energy distance* as

$$\mathrm{GED}(P, Q) := \sqrt{2\mathbb{E}[d(X, Y)] - \mathbb{E}[d(X, X')] - \mathbb{E}[d(Y, Y')]}, \tag{2.8}$$

where X, X' and Y, Y' are independent samples of P and Q respectively. In [49] it was shown that the generalized energy distance is indeed a metric, if d is a metric.

Next we review the *sliced Wasserstein-p distance* (see for example [16, Chapter 5.1]). For any $\lambda \in \mathbb{R}^n$ we consider the projection map

$$\lambda^* : \mathbb{R}^n \to \mathbb{R}, \; x \mapsto \lambda^T x.$$

With the pushforward measures through λ^* we can define the sliced Wasserstein-p distance

$$\mathrm{SW}_p(P, Q) := \left(\int_{\mathbb{S}^{n-1}} W_p(\lambda^* \# P, \lambda^* \# Q)^p d\lambda \right)^{1/p}, \tag{2.9}$$

where we integrate over the unit sphere $\mathbb{S}^{n-1}$, assuming that $\mathcal{X} \subset \mathbb{R}^n$. In [16, Proposition 5.1.2, Theorem 5.1.5] it is shown that the sliced Wasserstein distances are indeed metrics and that they are equivalent to their respective Wasserstein distance on compact sets.
The sliced Wasserstein-2 distance for empirical measures can be computed using the following property of the Wasserstein-2 distance. For empirical measures on finite sets $\{x_1, \dots, x_m\}, \{y_1, \dots, y_m\} \subset \mathcal{X}$, i.e.

$$\hat{P} := \tfrac{1}{m} \sum_{i=1}^{m} \delta_{x_i}, \quad \hat{Q} := \tfrac{1}{m} \sum_{i=1}^{m} \delta_{y_i},$$

the Wasserstein-2 distance is equivalently defined [16, Lemma 5.2.4] by

$$\mathrm{W}_2(\hat{P}, \hat{Q})^2 = \frac{1}{m} \min_{\sigma \in \mathcal{P}_m} \sum_{i=1}^{m} \|x_i - y_{\sigma(i)}\|_2^2, \tag{2.10}$$

where $\mathcal{P}_m$ is the set of permutations on $\{1, \dots, m\}$. In the case that x_i and y_i stem from $\mathbb{R}$, there exists a closed form solution for the minimizing permutation. Define σ_x and σ_y as the permutations such that the resulting orders are increasing, i.e.

$$x_{\sigma_x(1)} \leq \dots \leq x_{\sigma_x(m)} \quad \text{and} \quad y_{\sigma_y(1)} \leq \dots \leq y_{\sigma_y(m)}.$$

Then the optimal permutation is $\sigma^* := \sigma_y \circ \sigma_x^{-1}$, which pairs $x_{\sigma_x(i)}$ with $y_{\sigma_y(i)}$, as was shown for example in [22]. This allows to compute the Wasserstein distance occurring in (2.9) efficiently in time $\mathcal{O}(m \log(m))$ using e.g. the merge sort algorithm. Note that the pushforward measure through λ^* for an empirical measure is again a weighted sum of dirac measures

$$\lambda^* \# \hat{P} = \tfrac{1}{m} \sum_{i=1}^{m} \delta_{\lambda^T x_i}.$$

2.3.2 Application in GANs

In [30], Goodfellow et al. showed that training the generator by minimizing the objective

$$C(G) = \mathbb{E}_{X \sim \mathbb{P}_r} [\log(D(X))] + \mathbb{E}_{Z \sim \mathbb{P}_Z} [\log(1 - D(G(Z)))]$$

for a fixed D, means to minimize a lower bound of $2\,\mathrm{JS}(\mathbb{P}_r, \mathbb{P}_G) - 2\log(2)$, with equality if D is optimal. Hence, the generator is trained by minimizing an approximation of the Jensen-Shannon divergence between the data distribution and the distribution of the generated samples.

Arjovsky et al. [9] argued that using Jensen-Shannon divergence for the generator training is not preferable. Let $\mathbb{P}_\theta := \mathbb{P}_{G_\theta}$ for given parameters θ. Further let $(\theta_t)_t$ be a sequence of parameters for G and assume that $\mathbb{P}_{\theta_\infty} = \mathbb{P}_r$. If the distributions are supported on lower dimensional manifolds of $\mathcal{X}$ (which can be the case for real world data), it can be that $\mathbb{P}_{\theta_t}$ does not converge to $\mathbb{P}_r$ in Jensen-Shannon divergence although $\mathbb{P}_{\theta_\infty} = \mathbb{P}_r$ holds. In particular, $\theta \to \mathrm{JS}(\mathbb{P}_r, \mathbb{P}_\theta)$ might not be continuous. This is a drawback, since we cannot expect to improve G_θ by applying gradient descent.
On the other hand the authors showed that under mild conditions (which are usually met by neural networks), the Wasserstein-1 distance, $\theta \to \mathrm{W}_1(\mathbb{P}_r, \mathbb{P}_\theta)$, is continuous everywhere and differentiable almost everywhere. These are both properties that are desirable for a loss function to which gradient descent is applied.
Arjovsky et al. gave the following example to demonstrated that there exist very simple target distributions supported on lower dimensional manifolds, where learning with the Jensen-Shannon divergence does not work, while learning with the Wasserstein-1 distance yields good results.

Example 2.1. Let $\mathbb{P}_0$ be the distribution of $(0, X) \in \mathbb{R}^2$, where $X \sim U[0,1]$ is a uniform random variable. Furthermore, assume that $G_\theta(z) = (\theta, z) \in \mathbb{R}^2$ and let $\mathbb{P}_\theta$ be the distribution of $G_\theta(Z)$, for another random variable $Z \sim U[0,1]$. Obviously, $\mathbb{P}_\theta \overset{d}{=} \mathbb{P}_0$ for $\theta = 0$.
To compute the Wasserstein-1 distance, we note that $\|(0,X) - (\theta, Z)\|_1 \geq |\theta|$ and that one can choose π as the distribution of $((0,Y),(\theta,Y))$ for a common random variable $Y \sim U[0,1]$. This distribution π is an element of $\Pi(\mathbb{P}_0, \mathbb{P}_\theta)$ and using it for the other inequality, it follows that

$$\mathrm{W}_1(\mathbb{P}_0, \mathbb{P}_\theta) = |\theta|.$$

To compute the Jensen-Shannon divergence, we note that for $\theta \neq 0$, the supports of $\mathbb{P}_0$ and $\mathbb{P}_\theta$ do not intersect. Therefore,

$$\mathrm{KL}\left(\mathbb{P}_0 \| \tfrac{1}{2}(\mathbb{P}_0 + \mathbb{P}_\theta)\right) = \int_{\mathbb{R}^2} \log\left(\frac{d\mathbb{P}_0}{\frac{1}{2}d\mathbb{P}_0}\right) d\mathbb{P}_0 = \log(2)$$

and equivalently $\mathrm{KL}\left(\mathbb{P}_\theta \| \frac{1}{2}(\mathbb{P}_0 + \mathbb{P}_\theta)\right) = \log(2)$. On the other hand, if $\theta = 0$, the two Kullback-Leibler divergences obviously are 0, so that we have

$$\mathrm{JS}(\mathbb{P}_0, \mathbb{P}_\theta) = \begin{cases} 0, & \text{if } \theta = 0 \\ \log(2), & \text{if } \theta \neq 0 \end{cases}.$$

Hence, for a sequence $(\theta_t)_{t\in\mathbb{N}} \subset \mathbb{R}$, not all equal to 0 and satisfying $\theta_t \to 0$, we have that $\mathrm{W}_1(\mathbb{P}_0, \mathbb{P}_{\theta_t}) \xrightarrow{t\to\infty} 0$, while $\mathrm{JS}(\mathbb{P}_0, \mathbb{P}_{\theta_t})$ does not converge. This shows, that $\theta \to \mathrm{JS}(\mathbb{P}_0, \mathbb{P}_\theta)$ is not continuous in general.

Since the infimum in (2.3) is intractable, Arjovsky et al. suggest to use the dual formulation (2.4) of the Wasserstein-1 distance. This naturally leads to an adversarial training setting, when the Lipschitz function f is approximated by a deep neural network D_ω, trained to approach the maximum. In [9], D_ω is called *critic*, but we will stick to the name discriminator. The resulting two-player minimax game is the following:

$$\min_\theta \max_\omega \mathbb{E}_{X\sim\mathbb{P}_r}\left[\, D_\omega(X)\,\right] - \mathbb{E}_{Z\sim\mathbb{P}_Z}\left[\, D_\omega(G_\theta(Z))\,\right]. \tag{2.11}$$

At this point we want to highlight the different approaches with which we arrived at the similar minimax formulations (2.1) and (2.11). In the first case we wanted the discriminator to learn predicting the probability that a sample comes from training data rather than sampled data and applied the cross entropy loss. In the latter case, we saw that the first method yields minimizing the Jenson-Shannon divergence between the distributions of training data and generated data and argued that the Wasserstein distance would be better suited instead. Approximation of this distance led again to a GAN formulation.
To enforce a Lipschitz constraint on D_ω, Arjovsky et al. proposed to use weight clipping. That is to fix a constant $c > 0$ and to replace each weight ω_i by $sign(\omega_i)\min(|\omega_i|, c)$, after each update of the discriminator. This implies that D_ω is α-Lipschitz for some α only depending on the network architecture and c. We now arrived at the *Wasserstein GAN* (WGAN) formulation, where an approximation of $\alpha\, W_1(\mathbb{P}_r, \mathbb{P}_G)$ is minimized by the generator.
Since D_ω is Lipschitz and not constrained to map to $[0, 1]$ any more, the vanishing gradient problem should be resolved. This makes the training process much more stable since it theoretically allows to train the discriminator to completion after each generator update. In fact, the better the discriminator is trained, the better are the gradients for updating the generator. Therefore n_{disc} is typically set to 5, which is a tradeoff between training the discriminator to completion and not increasing the time of one training iteration too much.
Moreover, the authors showed empirically that WGANs can still be trained successfully when changing the architecture of the networks. Another big benefit of the Wasserstein distance is, that it correlates well with the generators convergence and sample quality, i.e. one can read of convergence from the loss function.

It is obvious that weight clipping is not a nice method to enforce Lipschitz constraints. That is why Gulrajani et al. [32] propose to use a soft gradient norm penalty added to the loss function of the discriminator instead. The

reasoning behind is, that a function is 1-Lipschitz if and only if its gradient norm is bounded by 1. In particularly, after a sign switch, the discriminator is trained to *minimize* the following loss function

$$\begin{aligned} L = \mathbb{E}_{Z\sim\mathbb{P}_Z}\left[\, D_\omega(G_\theta(Z))\,\right] - \mathbb{E}_{X\sim\mathbb{P}_r}\left[\, D_\omega(X)\,\right] \\ + \lambda\, \mathbb{E}_{\hat{X}\sim\mathbb{P}_\epsilon}\left[\left(\|\nabla_{\hat{X}} D_\omega(\hat{X})\,\|_2 - 1\right)^2\right], \end{aligned} \tag{2.12}$$

where $\mathbb{P}_\epsilon$ is the distribution of the random variable $\hat{X} := \epsilon\, X + (1-\epsilon)\, G_\theta(Z)$, with ϵ a uniformly distributed random variable on $[0,1]$, $X \sim \mathbb{P}_r$ and $Z \sim \mathbb{P}_Z$. The hyper-parameter λ controls the influence of the penalty term and is typically set to 10. Note that this loss function forces the discriminator to become a function with minimal Lipschitz constant 1, rather than just 1-Lipschitz. In experiments this formulation produced better results. The authors call this the *Wasserstein GAN with gradient penalty* (WGAN-GP) but it is also referred to as *Improved WGAN* (IWGAN) due to the title of the paper. They exploit the stabilized training procedure by using residual blocks [34] leading to a deeper and more expressive network architecture, denoted WGAN-GP ResNet. Overall, WGAN-GP is a true improvement of WGAN and should therefore be favoured.

Although the experimental results of WGAN-GP are promising, there is a theoretical downside. The architecture of D_ω normally involves ReLU or Leaky-ReLU activation functions. In [42, 7] the authors show, that there does not exist a neural network with element-wise activations (ReLU, Leaky-ReLU and all other standard activation functions) that is 1-Lipschitz (w.r.t. L_∞ or L_2 norm) and replicates the absolute value function, an important member of the class of 1-Lipschitz functions. The authors of [7] propose a new activation function named *GroupSort*, with which they show a universal approximation result of 1-Lipschitz (w.r.t L_∞ norm) functions using 1-Lipschitz GroupSort networks.

While the WGAN approach uses the dual formulation of the Wasserstein-1 distance as a loss function, Genevay et al. [26] propose to use the primal formulation of the general optimal transport loss (2.2). To make it tractable, they switch to the regularized version (2.5) and subsequently use the Sinkhorn loss (2.7) between the distributions of the training data and the generated data, $\hat{\mathcal{W}}_{c,\epsilon}(\mathbb{P}_r, \mathbb{P}_G)$, as objective function for the generator G_θ. To be able to compute gradients of (2.7), on each mini-batch of size m the probability measures $\mathbb{P}_r$ and $\mathbb{P}_G$ are approximated by their empirical measures on

the batch

$$\hat{\mathbb{P}}_r := \tfrac{1}{m}\sum_{i=1}^{m}\delta_{x_i}, \quad \hat{\mathbb{P}}_G := \tfrac{1}{m}\sum_{i=1}^{m}\delta_{y_i}, \tag{2.13}$$

where x_i, y_i are sampled from $\mathbb{P}_r$ and $\mathbb{P}_G$ respectively. Then the Sinkhorn algorithm [71] is applied, which yields the gradient for updating G_θ.
Until now, the described approach works with a fixed cost function c and without the need of a discriminator. For the MNIST dataset [53], Genevay et al. use the fixed cost $c(x,y) := \|x-y\|_2^2$ and are able to successfully train the generator in this non-adversarial setting. But it turns out that training on data with higher resolution does not work well without an adversarial counterpart to the generator. Hence, a discriminator network $D_\omega : \mathcal{X} \to \mathbb{R}^l$ reducing the dimensionality of $\mathcal{X}$, is introduces, with which a trainable cost function is defined as $c(x,y) := \|D_\omega(x) - D_\omega(y)\|_2$. The discriminator is then alternately trained to maximize the Sinkhorn loss $\hat{\mathcal{W}}_{c,\epsilon}(\mathbb{P}_r, \mathbb{P}_G)$. The same technique as for the generator is applied to compute the gradients for updating D_ω.

The next approach we review combines the regularized optimal transport divergence (2.6) with the energy distance (2.8). The resulting distance is called *minibatch energy distance* [67] and achieved by using $\mathcal{W}_{c,\epsilon}$ as distance d in the definition of (2.8). A priori, the cost function c can again be chosen fixed, so that the generative model could be used without adversarial training of a discriminator. But the authors suggest to define a discriminator D_ω mapping from $\mathcal{X}$ into a lower dimensional space and define the cost as 1 minus the cosine of the enclosed angle

$$c(x,y) := 1 - \frac{D_\omega(x)^T\, D_\omega(y)}{\|D_\omega(x)\|_2\, \|D_\omega(y)\|_2}.$$

Given independent batches $\mathbf{X} = \{x_1, \dots, x_m\}$, $\mathbf{X}' = \{x_1', \dots, x_m'\}$ from $\mathbb{P}_r$ and independent batches $\mathbf{Y} = \{y_1, \dots, y_m\}$, $\mathbf{Y}' = \{y_1', \dots, y_m'\}$ from $\mathbb{P}_G$ the squared minibatch energy distance is defined as

$$\mathrm{MED}^2(\mathbb{P}_r, \mathbb{P}_G) := 2\mathbb{E}[\mathcal{W}_{c,\epsilon}(\mathbf{X},\mathbf{Y})] - \mathbb{E}[\mathcal{W}_{c,\epsilon}(\mathbf{X},\mathbf{X}')] - \mathbb{E}[\mathcal{W}_{c,\epsilon}(\mathbf{Y},\mathbf{Y}')].$$

Here the notation means, that the regularized optimal transport divergence between the empirical measures implied by the mini-batches is computed. To do so, the Sinkhorn algorithm [71] is used. The generator is trained to minimize this distance, while the discriminator tries to maximize it in the typical GAN procedure. When implemented, the slight modification

$$\begin{aligned} L = &\mathcal{W}_{c,\epsilon}(\mathbf{X},\mathbf{Y}) + \mathcal{W}_{c,\epsilon}(\mathbf{X},\mathbf{Y}') + \mathcal{W}_{c,\epsilon}(\mathbf{X}',\mathbf{Y}) + \mathcal{W}_{c,\epsilon}(\mathbf{X}',\mathbf{Y}') \\ &- 2\,\mathcal{W}_{c,\epsilon}(\mathbf{X},\mathbf{X}') - 2\,\mathcal{W}_{c,\epsilon}(\mathbf{Y},\mathbf{Y}') \end{aligned}$$

is used as loss function, which tries to approximate the expectations in reasonable time. The authors Salimans and Zhang et al. [67] call this the *optimal transport GAN* (OT-GAN).
An advantage of using the minibatch energy distance is, that it remains a valid objective, even when the discriminator is stopped being trained. This is incorporated by updating the generator more often than the discriminator (e.g. WGAN and WGAN-GP do the opposite). The biggest disadvantage of this method is the very high computational cost, which is induced by the need to use very large batch sizes (the authors use 2048 and 8000 in their experiments).

Deshpande et al. [22] use the squared sliced Wasserstein-2 distance (2.9) as loss function for the generator. The integral is approximated by the mean over a finite set Λ of randomly drawn directions in $\mathbb{S}^{n-1}$. Therefore the generator is trained to minimize

$$\frac{1}{|\Lambda|}\sum_{\lambda\in\Lambda} \mathrm{W}_2(\lambda^*\#\mathbb{P}_r, \lambda^*\#\mathbb{P}_G)^2.$$

Again, the measures $\mathbb{P}_r$ and $\mathbb{P}_G$ are approximated on each batch by their empirical measures (2.13). Hence, the efficient method described in Section 2.3.1 can be used to compute the exact Wasserstein-2 distance between them. The authors call their generative model the *sliced Wasserstein generator* (SWG) and suggest to use up to 10'000 randomly drawn directions.
We notice that similar to [26], the SWG approach does not necessarily need a discriminator. In particular, Deshpande et al. show that on the MNIST dataset this approach works to successfully train the generator. However, on higher dimensional datasets, too many directions θ do not extract useful information from the distributions. There are two natural adversarial options to deal with this problem. Either, to use another network which is trained to produce good directions. Or to use another network which reduces the dimension of the two distributions before comparing them, so that already the important information is extracted. Here the authors choose the latter one (without mentioning the first). They define the usual discriminator $D_\omega : \mathcal{X} \to \mathbb{R}$ and define D'_ω to be some intermediate lower dimensional (w.r.t $\mathcal{X}$) layer of it (in their implementation they set it to be the last hidden layer). Then the generator is trained to minimize

$$\frac{1}{|\Lambda|}\sum_{\lambda\in\Lambda} \mathrm{W}_2\left((\lambda^*\circ D'_\omega)\#\hat{\mathbb{P}}_r, (\lambda^*\circ D'_\omega)\#\hat{\mathbb{P}}_G\right)^2$$

while the discriminator is trained with the standard GAN objective to max-

imize

$$\mathbb{E}_{X\sim\hat{\mathbb{P}}_r}\left[\log(D_\omega(X))\right] + \mathbb{E}_{Z\sim\hat{\mathbb{P}}_Z}\left[\log(1 - D_\omega(G_\theta(Z)))\right].$$

Note the different usage of D'_ω and D_ω for the generator respectively discriminator. In particular, for the loss function of the generator only some of the layers of the discriminator are applied, which project from $\mathcal{X}$ to a lower dimensional space. Of course, also Λ is then chosen as a set of directions in this lower dimensional space.
Beside some theoretical results, the authors show empirically that the sliced Wasserstein distance correlates with the generated sample quality, like the Wasserstein distance does. This was to be expected, since the two distances are equivalent.

Yet another approach to use the sliced Wasserstein distance is suggested by Wu and Huang et al. [81]. Here, the sliced Wasserstein-1 distance is used since it naturally leads, via Kantorovich-Rubinstein duality (2.4), to an adversarial training procedure. In particular, the Wasserstein-1 distance in (2.9) is computed using its dual formulation. The authors claim that enforcing the 1-Lipschitz constraint via gradient penalty works well in 1 dimension, while it might be delicate in higher dimensions. Hence, the use of the sliced Wasserstein distance should have an advantage over the normal WGAN-GP approach [32]. Also here, the integral is approximated by a sum over different directions, but instead of using thousands of random ones, only a few, which are constrained to be orthogonal, are used and trained.
To make it easier to train the directions, the discriminator is split up into two parts. The first part, labelled E (for encoder), maps from $\mathcal{X}$ to a lower dimensional space. This is the space on which the sliced Wasserstein distance is computed by the second part, S. It already incorporates the approximation of the integral by a sum and also includes the approximation of the optimal Lipschitz function f in (2.4). It is given by

$$S(y) := \frac{1}{K}\sum_{i=1}^{K}\phi(\alpha_i\lambda_i^T y + \beta_i),$$

where $\alpha_i, \beta_i \in \mathbb{R}$ are trainable parameters and ϕ is an activation function. The complete discriminator then is $D_\omega := S \circ E$. Instead of penalizing the gradient of D_ω, which is the usual WGAN-GP procedure, the gradients of E and S are penalized separately. The resulting minimax game is

$$\min_{G_\theta}\max_{D_\omega = S\circ E}\mathbb{E}_{X\sim\mathbb{P}_r}\left[D_\omega(X)\right] - \mathbb{E}_{Z\sim\mathbb{P}_Z}\left[D_\omega(G_\theta(Z))\right]$$
$$- \gamma_1\,\mathbb{E}_{\hat{X}\sim\mathbb{P}_{\hat{X}}}\left[\|\nabla_{\hat{X}}E(\hat{X})\|_2^2\right] - \gamma_2\,\mathbb{E}_{\hat{Y}\sim\mathbb{P}_{\hat{Y}}}\left[\left(\|\nabla_{\hat{Y}}S(\hat{Y})\|_2 - 1\right)^2\right],$$

where $\hat{X}$ and $\hat{Y}$ are sampled uniformly and independent of each other from the line between X and $G_\theta(Z)$ respectively the line between $E(X)$ and $E(G_\theta(Z))$ (confer the description of WGAN-GP). The nonnegative hyper parameters γ_1 and γ_2 are used to balance the penalties and the expectations are approximatied by means over batches as usual. Wu and Huang et al. call this the *sliced Wasserstein GAN* (SWGAN). We remark that a special optimization method for updating the directions λ_i is needed, since they are constrained to be orthogonal.

Of course, there are also other approaches using an objective function that is not related to optimal transport. We list some examples, without going into detail.
The *Cramér GAN* proposed in [13] uses the energy distance (2.8) with $d(x, y) = \|x - y\|_2$ and is developed further to the OT-GAN [67] (described above).
The *maximum mean discrepancy GAN* (MMD GAN) [54] combines the ideas of GANs and *generative moment matching networks* (GMMN) [55], which are trained similar to GANs but using a fixed kernel maximum mean discrepancy distance. Although GMMNs have nice theoretical properties, their empirical performance is rather poor compared to GANs. This is improved by GMM GANs.
The *dual discriminator GAN* (D2GAN) [59] aims at minimizing a combination of the Kullback-Leibler divergence and its reversed form and will be reviewed in Section 2.4.2.

2.4 Different architectures for GANs

In this section we review realizations of GANs that have an altered model architecture. The main focus does not lie on fine architectural choices for the single networks like the kernel-size, stride and padding of convolutional layers or the number of layers and whether residual blocks are used or not. These details, which are often set by hyper-parameter optimization rather than being implied by theoretical considerations, can be looked up in the respective papers or implementations. Instead we turn our attention to more fundamental modifications of the GAN design, as introducing additional generators and discriminators or auxiliary classifiers and encoders. We illustrate the different methods such that their fundamental ideas become visible. At the same time we emphasize how these ideas can be implemented, especially, how the training procedures of these GAN variants work.
We focus on three classes of architectures. Firstly, architectures that are used to create more structure on the latent space $\mathcal{Z}$, mainly to tackle the mode

collapse problem (Section 2.4.1). Secondly, other approaches to avoid mode collapse (Section 2.4.2). And finally, GANs where the generator (or multiple generators) is trained in a stacked manner leading to a generator function which is the composition of multiple functions, i.e. $G_\theta = G_1 \circ \cdots \circ G_k$ (Section 2.4.3). This is done to improve image quality when dealing with higher resolutions. We will also see supervised GAN variants (they make use of class labels for training and generation) which naturally belong to the groups mentioned above.

2.4.1 Creating more structure on the latent space

Let us begin with some comments about the choice of the latent space, following the discussion in [5]. A lot of implementations of GANs use 100 as dimension for the latent space. There does not seem to be a special reason for this number. It is rather likely that it was simply reused after being used in the first GAN implementation. On the one hand, for simplicity (otherwise there is one more hyper parameter one has to choose) and on the other hand for comparability. There are also GAN implementations that use higher dimensions. For example in [46], 512 is used with the idea to account for the much higher resolution of images the generator should produce there.
Since most GAN implementations rely on GPUs, the number of dimensions should be taken as a power of 2, because this helps the GPU to increase throughput and efficiency. That is why newer GAN implementations often use 128 instead of 100 as latent space dimension.
Intuitively, the dimension should be big enough, such that there is the possibility for enough variability in the latent space to enable the generator to cover the whole support of the target data distribution. Or to say it a bit differently, the dimension of the latent space should be big enough, that the geometry of the target data set can approximately be embedded in the latent space. This is the main idea used in [82] which will be explained in more detail later.
To the best of our knowledge, there has not yet been done more research about the latent space of GANs and how the choice of its dimension influences the training success respectively model quality.

The first GAN architecture which deals with the mode collapse problem is called *adaptive GAN* (AdaGAN) and was introduced by Tolstikhin et al. [75]. It should be understood as meta-algorithm which can also be applied with other generative models as base algorithm instead of the standard GAN. As the name suggests, this meta-algorithm is somehow related to AdaBoost [24].

We describe the idea by considering the situation described in Section 2.2, where the generative model only learns one or a few (here dogs) of all the modes (10 different animals) of the data distribution. In this situation we expect that the discriminator is quite certain about all the data samples coming from modes missed by the generator (here all others than dogs), since no similar generated samples exist which could confuse the discriminator. By *certain* we mean that the discriminator in the standard GAN setting, which maps to the probability that the sample comes from the true data, returns a value near 1. On the other hand, data samples of modes covered by the generator (here dogs) will confuse the discriminator more, optimally leading to values near 0.5. This can be used to compute weights w_i for all the data samples x_i, putting more weight on samples which the generator has not learned yet. Then a new generator can be trained on the weighted data and afterwards be combined with the previous generator to a mixture model.
This is done repeatedly for a fixed number of times T. In particular, weights $\beta_t \in (0, 1)$ for $2 \leq t \leq T$ are chosen and having the previous mixture model G_{t-1} and the new generator G_t^c trained on the weighted data samples, the new generative mixture model is defined as

$$G_t := (1 - \beta_t)\, G_{t-1} + \beta_t\, G_t^c.$$

This finally leads to the mixture model

$$G_T = \sum_{t=1}^{T} \alpha_t\, G_t^c,$$

where $\alpha_t := \beta_t\,(1 - \beta_{t+1}) \cdots (1 - \beta_T)$ with $\beta_1 := 1$. Tolstikhin et al. propose several heuristics to choose β_t. One is to set $\beta_t := \frac{1}{t}$ leading to a mixture, where every model is weighted equally. Sampling from such a mixture model is done in two steps. First one of the models G_t^c is chosen randomly (according to the distribution $\sum_{t=1}^{T} \alpha_t \delta_t$, where δ_t is the Dirac measure) and then this model is used to generate a sample.

The AdaGAN meta-algorithm is theoretically appealing because it does not build on one specific baseline method. Furthermore the authors showed some convergence results for this procedure, in particular, they specified conditions under which the mixture distribution converges to the true distribution. But it also has some severe practical downsides pointed out in [38]. First of all, it is not clear whether one generator alone can really be trained to generate very good samples of one or a few modes. Especially when the given data set is very diverse (e.g. ImageNet dataset [20]), a single-generator model tends to generate unrecognisable samples, that might be meaningless interpolations of the different modes. Furthermore, the boosting-like approach

is suited for baseline methods which do not take very long to be computed, which generally is not the case for GANs. In particular, the training time for a normal GAN on real world data lies between hours and days, depending on the data, the model specifications and the used hardware (in general, using GPUs drastically reduces the needed time for GAN training). Therefore, applying a boosting-like method (which can not be parallelized as for example the bagging procedure) results in a delicate increase of computation time, which might be prohibitive sometimes.

Hoang et al. [38] adopt the idea from AdaGAN to use multiple generators to cover different modes of the data. But to circumvent the outlined downsides, they propose to train a fixed number of K generators, G_i for $1 \leq i \leq K$, simultaneously. Additionally to the generators and the discriminator, a classifier C is introduced, which is trained to recognise which generator was used to produce a given sample. Using this classifier, the generators can be trained to specialize in different modes of the data. To do so, an additional objective term is introduced, encouraging the generators to maximize the distance among them. Besides, the generator G_θ and discriminator D_ω are trained with the standard GAN objective. Here, the generator G_θ refers to the mixture of the K generators G_i for a given mixture coefficient $\alpha = (\alpha_1, \ldots, \alpha_K)$ (confer the AdaGAN paragraph for the construction of the mixture). In practice, this coefficient is chosen equally weighted, i.e. $\alpha_i = \frac{1}{K}$ for all i. The resulting multi-player minimax game is

$$\begin{aligned} \min_{G_\theta, C} \max_{D_\omega} & \mathbb{E}_{X \sim \mathbb{P}_r} \left[\log(D_\omega(X)) \right] + \mathbb{E}_{Z \sim \mathbb{P}_Z} \left[\log(1 - D_\omega(G_\theta(Z))) \right] \\ & - \beta \sum_{i=1}^{K} \alpha_i \mathbb{E}_{Z \sim \mathbb{P}_Z} \left[\log(C_i(G_i(Z))) \right], \end{aligned} \tag{2.14}$$

where $C_i(x)$ is the classification probability that x was produced by the i-th generator G_i. The positive hyper-parameter β controls the diversity of the generators.
To make the method efficiently trainable, the generators G_i share their weights in all except for the first layer, while classifier C and discriminator D_ω share all weights but the ones of the last layer. Furthermore, the "-log D trick" is applied for the training procedure. The authors call their model *Mixture Generative Adversarial Net* (MGAN) and show that at an equilibrium point of (2.14) the Jensen-Shannon divergence between the data distribution $\mathbb{P}_r$ and the generator distribution $\mathbb{P}_G$ is minimized, while the multivariate extension of the Jensen-Shannon divergence between the single generator distributions $\mathbb{P}_{G_i}$ is maximized.

Moreover, the authors show empirically that MGAN successfully tackles the mode collapse problem on several example datasets. For the synthetic dataset described in Section 2.2, the superiority of MGAN over standard GAN can be seen by comparing Figure 1 to Figure 2.

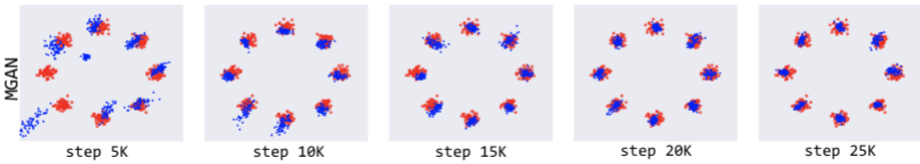


Figure 2: No mode collapse on toy dataset (source [38]). *Red: training dataset, blue: samples Generated by MGAN after respective training. time*

One might ask, why the MGAN approach can be interpreted as creating more structure on the latent space. The answer is the following. The first layer of the generators G_i is usually a fully connected one. Let us assume, that no activation function is applied here. Then it is given by

$$\varphi_i : \mathbb{R}^d \supset \mathcal{Z} \to \mathbb{R}^l,\ z \mapsto A_i\, z + b_i,$$

where $A_i \in \mathbb{R}^{l \times d}$ is a weight matrix and $b_i \in \mathbb{R}^l$ a bias. Let $Z \in \mathcal{Z}$ be a standard multivariate Gaussian random variable (the standard choice for the latent variable), then it holds that $\varphi_i(Z)$ is distributed like $N(b_i, A_i A_i^T)$. Hence, we can think of the mixture of the generators G_i equivalently as one generator (all their shared layers) which is applied to a mixture of Gaussians on the latent space. This Gaussian mixture has the same mixture coefficient α as the generator G_θ. Furthermore, its means and covariance-matrices are trainable. Now it is evident that MGAN creates more structure on $\mathcal{Z}$. A similar reasoning can be used for AdaGAN, with the difference that every mixture component also has its own generator.

With the interpretation of MGANs given above, we see that *dynamic Gaussian Mixture GANs* (GM-GAN) introduced in [14] by Ben-Yosef et al. are a special case of MGANs which are trained without the classifier C. The *static* version is similar, but here the means and covariance-matrices of the Gaussian mixture (respectively the first layers of all the G_i) are fixed.
Additionally, Ben-Yosef et al. introduce a *supervised* GM-GAN version, which can be applied, when the training data consists of N different classes and the class labels are known. To this end, the discriminator is modified, such that it maps $x \in \mathcal{X}$ to the standard simplex Δ^{N-1}, which represent the probabilities that x belongs to the respective classes. The modified loss

functions for generator G and discriminator D, which are both minimized, are define as

$$
\begin{aligned}
L(G) = &- \mathbb{E}_{Z\sim\mathbb{P}_Z}\left[\log(D(G(Z))_{f(y(Z))}) + \sum_{\substack{m=1\\ m\neq f(y(Z))}}^{N}\log(1-D(G(Z))_m)\right]\\
L(D) = &- \mathbb{E}_{Z\sim\mathbb{P}_Z}\left[\sum_{m=1}^{N}\log(1-D(G(Z))_m)\right]\\
&- \mathbb{E}_{X\sim\mathbb{P}_r}\left[\log(D(X)_{y(X)}) + \sum_{\substack{m=1\\ m\neq y(X)}}^{N}\log(1-D(X)_m)\right].
\end{aligned}
$$

Here $f : \{1,\dots,K\} \mapsto \{1,\dots,N\}$ is a fixed map specifying which mixture components should be mapped to which classes. Moreover, $y(X)$ denotes the true class label of a sample $X \in \mathbb{X}$ and $y(Z)$ denotes the index of the Gaussian used in the mixture. The advantage of this supervised GM-GAN over training a single generator for each class is obviously the smaller size of the model. Furthermore, if there is only limited training data for each class, weight sharing of the generators (in the MGAN formulation) might enable them to successfully learn the distributions, while the training of single GANs might fail due to too little training data. Considering the training time and quality of the results (assuming enough training data is given), it might be preferable to use a single GAN for each class, when one has the resources to train them parallelly.

DeLiGAN proposed by Gurumurthy and Sarvadevabhatla et al. [33] is the same as dynamic GM-GAN with the covariance-matrices of the Gaussians restricted to be diagonal and an added loss term penalizing shrinking or exploding entries of these diagonal matrices.

In [27] Gosh and Kulharia et al. introduce the *Multi-Agent Diverse GAN* (MAD-GAN). This is more or less the same as MGAN, with slight modifications of the objective functions. Furthermore, the weights of the first layer might be shared, while the weights of the last layer(s) are not shared. The authors propose that the amount of shared weights should depend on the diversity in the dataset. If a dataset is very diverse then no weight sharing at all might be appropriate. Of course, this fairly enlarges the model.

The next method we review also uses a Gaussian mixture on the latent space to tackle mode collapse, but combines it with the actual construction

of this space. In particular, a combination of the Bourgain Theorem and the Johnson-Lindenstrauss Lemma is used to embed the geometry of the given data $\mathbb{X} \subset \mathcal{X}$ in a (much) lower dimensional space, which is then used as latent space $\mathcal{Z}$. Due to the use of Bourgain Theorem the authors Xiao et al. call their method *BourGAN* [82]. The dimension of the resulting space $\mathcal{Z}$ depends only on the maximal pairwise distance distortion factor that is allowed for the embedding (i.e. the maximal factor by which the pairwise distance between any two points of the dataset is changed by the embedding) and on the number of points that are embedded. But it does not depend on the dimension of the space where these points live (here $\mathcal{X}$). Therefore a number m of data samples are chosen at random, where m is much smaller than $|\mathbb{X}|$, but still big enough that these m points reflect the mode structure of $\mathbb{X}$ well. m, and therefore also the dimension of $\mathcal{Z}$, is chosen automatically depending on $\mathbb{X}$, but the authors highlight that in all their experiments m was not bigger than 4096.
Once the m points are embedded in the space $\mathcal{Z}$, they are used as means for a Gaussian mixture $\mathbb{P}_Z$ on $\mathcal{Z}$ (the covariance matrix is set to identity matrix times a factor $\sigma = 0.1$). To transfer the geometric structure of $\mathbb{P}_Z$ (which is assumed to reflect the mode structure of $\mathbb{X}$) to the distribution of generated samples $\mathbb{P}_G$, an additional penalty term is added to the standard GAN objective. This penalty encourages the generator to match the pairwise distance of latent vectors with the pairwise distance of their generated samples. The resulting minimax game is

$$\begin{aligned}\min_{G_\theta} \max_{D_\omega} \; & \mathbb{E}_{X\sim\mathbb{P}_r}\left[\log(D_\omega(X))\right] + \mathbb{E}_{Z\sim\mathbb{P}_Z}\left[\log(1 - D_\omega(G_\theta(Z)))\right] \\ & + \beta\, \mathbb{E}_{Z_1,Z_2 \overset{\text{iid}}{\sim} \mathbb{P}_Z}\left[\left(\log(d(G_\theta(Z_1), G_\theta(Z_2))) - \log(\|Z_1 - Z_2\|_2)\right)^2\right],\end{aligned}$$

where β is a hyper-parameter and d is a metric on $\mathcal{X}$. Beside some theoretical results, the authors empirically show that BourGAN resolves mode collapse, while not producing unwanted samples lying outside the support of $\mathbb{P}_r$.
The BourGAN method is nice, because it is the first method which *constructs* the latent space and does not depend on the choice of its dimensionality which sometimes seems rather arbitrary. But this comes at the price of computing the embedding map, which is in $\mathcal{O}(m^2)$.

The last generative model we review in this section is inspired by both, GANs and VAEs, but can not be numbered among the one or the other. It rather defines its own class. Still it fits very well in this section, as will become clear in a moment. It is called *generative latent optimization model* (GLO) and was introduced by Bojanowski et al. [15]. The main idea is to pair each data sample $x_i \in \mathbb{X}$ with a latent vector $z_i \in \mathcal{Z} := \{z \in \mathbb{R}^d : \|z\|_2 \leq 1\}$. The

latent vectors are initialized by drawing them from a d-dimensional standard Gaussian distribution and rescaling them by the factor $1/\sqrt{d}$. Afterwards they are projected to $\mathcal{Z}$. This leads to the dataset $\{(x_1, z_1), \ldots, (x_N, z_N)\}$, where $N := |\mathbb{X}|$. The generator $G_\theta : \mathcal{Z} \to \mathcal{X}$ is defined as usual, but is trained to minimize a reconstruction loss. Furthermore, also the latent vectors are trainable such that they can be picked optimally for the reconstruction task. The complete training procedure is the following minimization problem

$$\min_{G_\theta} \tfrac{1}{N} \sum_{i=1}^{N} \min_{z_i \in \mathcal{Z}} L(G_\theta(z_i), x_i).$$

Here $L : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a loss function measuring the reconstruction error. For best image quality, the authors suggest to use the *Laplacian pyramid* Lap_1 *loss* defined as

$$Lap_1(x, y) := \sum_j 2^{-2j} \|L_j(x) - L_j(y)\|_1,$$

where $L_j(x)$ is the j-th level of the Laplacian pyramid representation of x (confer LAPGAN in Section 2.4.3). The optimization is done, as usual, with SGD, whereby the updated latent vectors are afterwards projected back to $\mathcal{Z}$ if needed.
To sample from the GLO model, a Gaussian distribution with full covariance is fitted to the latent vectors. Then, noise is drawn from this distribution and passed through the generator. Bojanowski et al. highlight that GLO can be interpreted as GAN without discriminator or as (variational) auto-encoder without encoder. In particular, GLO is a generalization of (variational) auto-encoders, since it can recover all of their solutions (z_i's can be chosen freely instead of following a parametric model). We also note that the training procedure is much easier compared to GANs (since it is a pure minimization task compared to a minimax game) and at the same time is forced to learn all modes of the given data. Until now this comes at the price of reduced sample quality compared to state-of-the art GANs dealing with mode collapse (e.g. MGAN).

2.4.2 Other architectural modifications to avoid mode collapse

Srivastava et al. [73] propose to use a reconstruction network $F : \mathcal{X} \to \mathcal{Z}$ in addition to the generator G_θ and discriminator D_ω. This network F, also a DNN, is trained with 2 goals. One is to approximately invert the generator function and the other is to map the true data distribution $\mathbb{P}_r$ to a Gaussian distribution on $\mathcal{Z}$, particularly to $\mathbb{P}_Z$. The authors call this

method *variational encoder enhancement of GANs* (VEEGAN), since the training procedure can be interpreted as an implicit variational principle. Here the discriminator is defined as mapping $D_\omega : \mathcal{Z} \times \mathcal{X} \to \mathbb{R}$ and trained to minimize

$$\begin{aligned} L_{D_\omega} := &- \mathbb{E}_{Z\sim\mathbb{P}_Z}\left[\log(\sigma(D_\omega(Z, G_\theta(Z))))\right] \\ &- \mathbb{E}_{X\sim\mathbb{P}_r}\left[\log(1-\sigma(D_\omega(F(X), X)))\right], \end{aligned}$$

where σ is the sigmoid function. Note the change of the interpretation of the discriminator output. In particular, $\sigma(D_\omega(\cdot,\cdot))$ gives the probability that a sample was generated by G_θ. Furthermore, note that the discriminator is trained to distinguish $\mathbb{P}_r$ from $\mathbb{P}_G$ as well as $F\#\mathbb{P}_r$ from $\mathbb{P}_Z$, by comparing the joint distributions. Simultaneously, the generator and the reconstruction network are trained to minimize

$$L_{G_\theta,F} := \mathbb{E}_{Z\sim\mathbb{P}_Z}\left[\, D_\omega(Z, G_\theta(Z)) + \|Z - F(G_\theta(Z))\|_2^2 \,\right].$$

The authors argue that the reconstruction network helps avoiding mode collapse, because as long as G_θ misses some modes, it is unlikely that both training goals of F are satisfied. Especially, if F inverts G_θ sufficiently well, then the distribution $(F \circ G_\theta)\#\mathbb{P}_Z$ is very similar to $\mathbb{P}_Z$. Since G_θ misses some modes, Srivastava et al. claim it is unlikely that $F\#\mathbb{P}_r$ is similar to $\mathbb{P}_Z$ as well. On the other hand, if $F\#\mathbb{P}_r$ is similar to $\mathbb{P}_Z$, then it is unlikely that the same holds for $(F \circ G_\theta)\#\mathbb{P}_Z$, since G_θ does not cover the whole support of $\mathbb{P}_r$.
Here we have to remark that there are easy examples contradicting this reasoning. Assume $\mathbb{P}_Z = N(0,1)$ and $\mathbb{P}_r$ is an arbitrary mixture of $N(m,1)$ and $N(-m,1)$ with m large (e.g. 10). Furthermore assume that G_θ undergoes mode collapse and only covers one mode such that $\mathbb{P}_G = N(m,1)$. Then setting F to be the relatively easy function $F(x) := |x| - m$, satisfies both objectives of F nearly perfectly. Nevertheless, Srivastava et al. showed in their experiments that VEEGAN can prevent mode collapse.

In the last section we have seen some methods introducing multiple generators acting in a parallel manner (AdaGAN, MGAN, MAD-GAN). A natural question is, whether we can also profit from multiple parallel discriminators. Nguyen et al. [59] show with their *dual discriminator GAN* (D2GAN) that this is the case. Their idea is the following. Instead of using an approximation of the Jensen-Shannon divergence as objective function, they propose to use and approximation of the Kullback-Leibler divergence $KL(\mathbb{P}_r\|\mathbb{P}_G)$ and its reversed form $KL(\mathbb{P}_G\|\mathbb{P}_r)$. This naturally leads to a two discriminator formulation. In particular, the discriminators D_1 and D_2 map $\mathcal{X}$ to $\mathbb{R}_{\geq 0}$ and

do not share any weights. The minimax game they are playing with the generator G_θ is

$$\min_{G_\theta} \max_{D_1,D_2} \mathbb{E}_{X\sim\mathbb{P}_r}\left[\,\gamma_1 \log(D_1(X)) - D_2(X)\,\right] \\ + \mathbb{E}_{Z\sim\mathbb{P}_Z}\left[\,-D_1(G_\theta(Z)) + \gamma_2 \log(D_2(G_\theta(Z)))\,\right],$$

where the hyper-parameters $\gamma_1, \gamma_2 \in (0,1]$ stabilize the training and control the effect of the KL and reverse KL divergence. In particular, by relatively increasing γ_1 the focus is shifted towards minimizing the KL divergence $KL(\mathbb{P}_r \| \mathbb{P}_G)$ and vice versa.
Instead of going more into detail here, we continue with another method which also incorporates two discriminators, but at two different levels.

Manifold guided GANs (MGGANs), proposed by Bang et al. [11], start by training an auto-encoder (AE). Like VAEs, an auto-encoder consists of an encoder $E : \mathcal{X} \to \mathcal{M}$ and a decoder $D : \mathcal{M} \to \mathcal{X}$, which are trained to minimize the (L_1 or L_2) reconstruction error. The main difference is that E is not trained to match a given distribution on $\mathcal{M}$. Hence the auto-encoder can not be used as generative model. But for a fixed lower dimensional space $\mathcal{M}$ (the so called *manifold* space) it aims to learn (using SGD) the best approximation of the target data $\mathbb{X} \subset \mathcal{X}$ in $\mathcal{M}$. If the dimension of $\mathcal{M}$ is big enough we therefore assume that the encoding map E can approximately embed the geometry of $\mathbb{X}$ in $\mathcal{M}$.
The MGGAN formulation involves the usual generator G_θ and discriminator D_ω. Beside them, the pre-trained encoding map E together with another discriminator, which we notate D_E, are used. D_E is trained to distinguish the encoded distributions of the given data $E\#\mathbb{P}_r$ and of the generated data $E\#\mathbb{P}_G$ on the latent space. Simultaneously, G_θ is trained to fool not only D_ω but also D_E. This way, the generator is *guided* to cover the whole support of $\mathbb{P}_r$, since otherwise it is easy for D_E to distinguish $E\#\mathbb{P}_r$ from $E\#\mathbb{P}_G$ (under our assumption that the geometry of the given data is well approximated in $\mathcal{M}$ by E). At the same time the quality of the generated data should not diminish too much, since also the standard GAN objective is used. The minimax game formulation is

$$\min_{G_\theta} \max_{D_\omega,D_E} \mathbb{E}_{X\sim\mathbb{P}_r}\left[\log(D_\omega(X)) + \log(D_E(E(X)))\right] \\ + \mathbb{E}_{Z\sim\mathbb{P}_Z}\left[\log(1 - D_\omega(G_\theta(Z))) + \log(1 - D_E(E(G_\theta(Z))))\right]$$

When implemented the "-log D trick" is used for the generator (applied to both terms). The authors call the added parts, E and D_E, *guidance network*

and highlight that it can also be added to baseline methods other than standard GAN.
Instead of creating more structure on the latent space (by replacing the standard Gaussian by a Gaussian mixture) this approach aims at matching the implied structures of given data and generated data on the manifold space $\mathcal{M}$. However, this approach does not solve the problem that the generator has to map the unimodal distribution $\mathbb{P}_Z$ to a multimodal distribution which should be close to $\mathbb{P}_r$. As outlined in [82, Figure 1], this leads to 2 problems. Assume that the generator covers all modes of $\mathbb{P}_r$ and that these modes are well separated. Then there exist points in $\mathcal{Z}$ which are mapped to interpolations of modes of the true data, where $\mathbb{P}_r$ is not supported. Furthermore, there exist points in $\mathcal{Z}$ with arbitrarily large absolute gradient values, which make the network unstable to train.
Still, their experiments show that MGGAN counteracts mode collapse, while preserving good image quality.

The last method we review here is called *class-splitting GAN* (SplitGAN) and was introduced by Grinblat et al. [31]. Previous works show that class label information can improve a GAN. Therefore the authors suggest to use k-means clustering repeatedly to generate artificial class labels. In particular, they either start with one class (unsupervised version) or with classes implied by given class labels (supervised version). Always after a specific number of training iterations, 2-means clustering is applied separately to each of the current classes which still has more samples than a certain threshold. Instead of applying the clustering algorithm to the samples directly, their outputs of the last hidden layer of the discriminator are used. This yields an advantage, because the discriminator extracts useful, abstract, lower dimensional information which is easier to compare than the image samples themselves.
To make use of the class labels, Grinblat et al. use the *auxiliary classifier GAN* (AC-GAN) [60] architecture in combination with the WGAN-GP loss function. We briefly describe this GAN variant. The class label information is injected into the generator G_θ via the scale and bias term in the batch normalization layers. Particularly, each class trains its own scale and bias. Whenever a parent class is split up into 2 new classes, they are both initialized with the scale and bias of the parent class but then trained on their own. The discriminator is defined as usual in the WGAN-GP setting and shares all but the last layers with the *auxiliary classifier* C. The last layer of C maps to the standard simplex Δ^{K-1}, and its output is interpreted as the classification probabilities for the K current classes. When splitting up a parent class, the last layer of C is extended (by 1 dimension) and the newly added weights for the child class are initialized with the weights of the parent

class. The training procedure works similar to the one of WGAN-GP, but in addition the generator is trained to produce samples of the correct class and the classifier is trained to correctly predict class labels, using the cross entropy loss. Assume there are currently K classes and let $\mathbb{P}_K := \sum_{k=1}^{K} \frac{1}{K}\delta_k$ be the equally weighted distribution on $\{1, \ldots, K\}$. Let $y(x)$ be the artificial label assigned to $x \in \mathbb{X}$. Furthermore, let $C_k(x)$ be the classification probability that $x \in \mathcal{X}$ belongs to class k. Then we can define the loss for the generator as

$$L_{G_\theta} := \mathbb{E}_{Z\sim\mathbb{P}_Z, Y\sim\mathbb{P}_K}\left[-D_\omega(G_\theta(Z,Y)) - \gamma \log(C_Y(G_\theta(Z,Y))) \right]$$

and the loss for the discriminator and the classifier as

$$\begin{aligned} L_{D_\omega, C} := {} & \mathbb{E}_{Z\sim\mathbb{P}_Z, Y\sim\mathbb{P}_K}\left[D_\omega(G_\theta(Z,Y)) - \gamma \log(C_Y(G_\theta(Z,Y))) \right] \\ & + \mathbb{E}_{X\sim\mathbb{P}_r}\left[-D_\omega(X) - \gamma \log(C_{y(X)}(X)) \right] \\ & + \lambda \mathbb{E}_{\hat{X}\sim\mathbb{P}_\epsilon}\left[\left(\|\nabla_{\hat{X}} D_\omega(\hat{X})\|_2 - 1 \right)^2 \right]. \end{aligned}$$

Here γ and λ are positive hyper-parameters and $\mathbb{P}_\epsilon$ is as in (2.12). All 3 players are trained to minimize their respective loss function.

2.4.3 Stacked generator functions

In this section we look at some GAN variants which all have a stacked architecture. The main goal of them is to improve the quality of the generated images, especially when it comes to higher resolutions. We present the models in their chronological order. In line with this, the achieved results improve from one model to the next.

The first method we review uses a Laplacian pyramid of the images. Hence, Denton and Chintala et al. call it *Laplacian GAN* (LAPGAN) [21]. We briefly explain the Laplacian pyramid. Let $d(\cdot)$ and $u(\cdot)$ be down- and upsampling operators which half, respectively double, the width and height of an image. Downsampling typically works by pooling, while upsampling can be done by replacing each pixel by a 2×2 square with copies of this pixel. Let I be an image and define its lower dimensional representations $I_k := d^k(I)$, the output after applying d for k times. Then the Gaussian pyramid of I is $\mathcal{G}(I) := [I_0, \ldots, I_K]$. Here K is chosen such that I_K has width and height less or equal 8 pixels. Furthermore let $L_K := I_K$ and recursively define $L_k := I_k - u(I_{k+1})$. Then the Laplacian pyramid is $\mathcal{L}(I) := [L_0, \ldots, L_K]$. Note that the image I can be restored from its Laplacian pyramid $\mathcal{L}(I)$.
In the LAPGAN approach $K+1$ GAN generators G_k and discriminators

D_K, for $0 \leq k \leq K$, are trained. G_k learns the distributions of the k-th level of the Laplacian pyramid applied to the given data. This is the distribution of $\mathcal{L}_k(\mathbb{X})$, where the meaning of the notation should be clear. For all $k < K$, conditional GANs are used, which were introduced in [58]. In this GAN version, the generator and discriminator get, beside their standard noise input, extra information on which they are conditioned. Usually this would be class labels, but in the case of LAPGANs the next lower dimensional version of the image is upsampled and passed as additional input. In particular, G_K and D_K are trained as usual with the target $\mathcal{G}_K(\mathbb{X})$. However, for $k < K$, the generator gets noise $Z_k \in \mathcal{Z}_k$ and $u(I_{k+1})$ as input, for some $I \in \mathbb{X}$. The conditionally generated image version is then $\tilde{I}_k := G_k(Z_k, u(I_{k+1})) + u(I_{k+1})$. The input for the discriminator D_k is either the lower dimensional version I_k of a true image $I \in \mathbb{X}$ or a generated version $\tilde{I}_k$. So for the discriminator the standard (not conditional) version is used. The resulting minimax game to obtain G_k and D_k is

$$\min_G \max_D \mathbb{E}_{I \sim \mathbb{P}_r} \left[\log(D(I_k)) \right] \\ + \mathbb{E}_{Z_k \sim \mathbb{P}_{Z_k}, I \sim \mathbb{P}_r} \left[\log(1 - D(G(Z_k, u(I_{k+1})) + u(I_{k+1}))) \right].$$

The authors also suggest a version, where class labels are passed as additional conditional variable to generator and discriminator. When it comes to generating new samples, the restoring procedure of the Laplacian pyramid is applied. In particular, at first $\tilde{I}_K := G_K(Z_K)$ is generated and then $\tilde{I}_k := G_k(Z_k, u(\tilde{I}_{k+1})) + u(\tilde{I}_{k+1})$ is produced recursively. The final output is $\tilde{I}_0$.
The LAPGAN approach splits up the learning of a hard task into several easier steps. Since each of these tasks is independent of the others, they can be parallelized. Compared to state-of-the-art GAN variants like WGAN-GP, OT-GAN or MGAN the quality of the generated samples of LAPGAN is much worse. This might partly be a consequence of not training the whole generation procedure together. In the next reviewed model this will be done, beside significant changes of the architecture.

Huang et al. propose the *stacked GAN* (SGAN) [41], which is similar to LAPGAN in the sense that it also splits up the hard training task into a sequence of easier sub-tasks. But instead of producing images in a coarse-to-fine manner, SGAN aims at modelling multi-level representations from abstract to specific, as the authors claim. Furthermore, SGAN needs labelled training data to be applicable, since it starts by training a deep neural classification network, which Huang et al. call *encoder* E. For some fixed number N, this encoder is written as $E = E_{N-1} \circ \cdots \circ E_0$, where each E_i consists of one

or more layers of the DNN E (the exact choice can be made by the user). This sequence of functions is used to define for a training sample $x \in \mathbb{X}$ its specific-to-abstract representations $h_0 := x$ and recursively $h_{i+1} := E_i(h_i)$, yielding $h_N = E(x) = \hat{y}$, the predicted class label (a vector in Δ^{K-1} where K is the number of classes). For each E_i a conditional GAN (in fact only the generator is conditional) is trained, where the generators goal is to invert the mapping E_i. In the first step, the GANs are trained independently, i.e. generator G_i gets a noise input $Z_i \in \mathcal{Z}_i$ and some h_{i+1} coming from a true data sample and produces $\hat{h}_i := G_i(Z_i, h_{i+1})$. The discriminator D_i gets either a true representation, h_i, or a generated one, $\hat{h}_i$, and tries to tell them apart. In the second step, all GANs are jointly trained, meaning that the conditioning is done on the output of the previous generator, i.e. $\hat{h}_i := G_i(Z_i, \hat{h}_{i+1})$. Only the first generator G_{N-1} is still conditioned on the predicted class labels. We now explain the optimization procedure in detail, which is also sketched in Figure 3. The discriminator D_i is trained to minimize the standard loss function (for a conditional generator)

$$\begin{aligned} L_{D_i} := &- \mathbb{E}_{h_i \sim \mathbb{P}_{r,i}} \left[\log(D_i(h_i)) \right] \\ &- \mathbb{E}_{Z_i \sim \mathbb{P}_{Z_i}, h_{i+1} \sim \mathbb{P}_{r,i+1}} \left[\log(1 - D_i(G_i(Z_i, h_{i+1}))) \right], \end{aligned}$$

where $\mathbb{P}_{r,i} := (E_{i-1} \circ \cdots \circ E_0)\#\mathbb{P}_r$ is the distribution of the h_i's. Note that this is the loss for independent training, which can easily be adapted for joint training by replacing h_{i+1} by $\hat{h}_{i+1}$. The same holds for the generator loss, which consists of the usual adversarial GAN loss (using the "-log D trick") as well as as two extra terms, a conditional loss and an entropy loss. In particular, the generator is trained to minimize

$$L_{G_i} := \lambda_1 L_{G_i}^{adv} + \lambda_2 L_{G_i}^{cond} + \lambda_3 L_{G_i}^{ent},$$

where $\lambda_i > 0$ are balancing parameters and

$$L_{G_i}^{adv} := -\mathbb{E}_{Z_i \sim \mathbb{P}_{Z_i}, h_{i+1} \sim \mathbb{P}_{r,i+1}} \left[\log(D_i(G_i(Z_i, h_{i+1}))) \right].$$

The conditional loss is added to encourage the generator to actually use the extra input h_{i+1}, by penalizing the reconstruction error of h_{i+1}

$$L_{G_i}^{cond} := \mathbb{E}_{Z_i \sim \mathbb{P}_{Z_i}, h_{i+1} \sim \mathbb{P}_{r,i+1}} \left[f(E_i(G_i(Z_i, h_{i+1})), h_{i+1}) \right].$$

Here f is the cross-entropy loss when labels are compared ($i = N - 1$) and otherwise the Euclidean distance. Only adding the conditional loss might lead to the problem that the generator G_i overdoes in using h_{i+1} and does not use its noise input Z_i any more. To counteract this behaviour, the entropy loss encourages G_i to use Z_i, by penalizing the reconstruction error of Z_i

$$L_{G_i}^{ent} := \mathbb{E}_{Z_i \sim \mathbb{P}_{Z_i}, h_{i+1} \sim \mathbb{P}_{r,i+1}} \left[\|(Q_i(G_i(Z_i, h_{i+1})) - Z_i\|_2 \right].$$

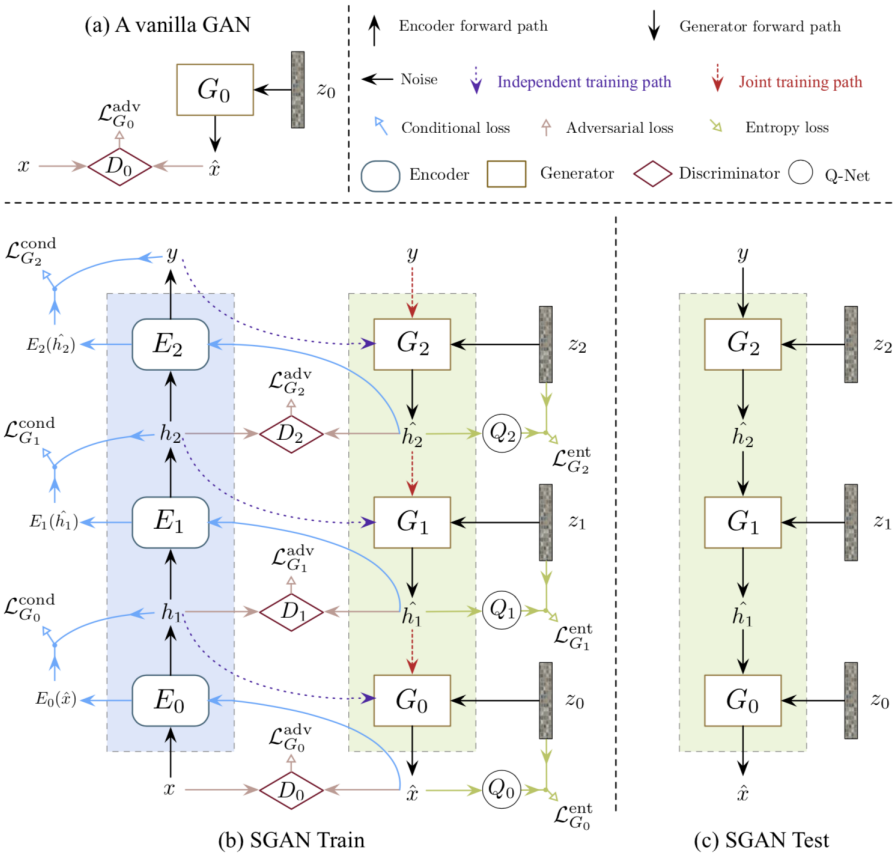


Figure 3: SGAN overview (source [41]). *(a) standard GAN. (b) SGAN training. (c) sampling from SGAN.*

Q_i is an additional DNN, that shares most of its weights (all but the last layer) with D_i and is trained to reconstruct Z_i from the output of the generator G_i. The training of Q_i is done together with the training of D_i. This loss term can be interpreted as maximizing a variational lower bound of the conditional entropy $\mathrm{H}(\hat{h}_i|h_{i+1})$, justifying the name entropy loss.
To generate new samples with class label y, noise vectors $Z_0, \ldots, Z_{N-1}$ are randomly drawn from their respective latent spaces and then the single generators are applied in a stacked manner, starting with $\hat{h}_{N-1} = G_{N-1}(Z_{N-1}, y)$ and going on recursively by $\hat{h}_i = G_i(Z_i, \hat{h}_{i+1})$. The final output is $\hat{x} := \hat{h}_0$.
The authors conduct several experiments to test their model, showing for example that joint training improves the results (confer Table 1). This is the reason for our hypothesis that LAPGAN might be improved by an additional joint training step.

Progressively growing GANs (PGGAN) introduced by Karras et al. [46] is the last method we review in this section. It builds on the same idea as LAPGAN and SGAN that learning to generate high-resolution images is easier when it is done in smaller steps. But instead of using multiple GANs which are stacked up, it comprises only one GAN that progressively grows during the training process. In particular, the generator and discriminator network are initialized having only a few layers such that images of size 4×4 can be processed. Then the following training-and-growing step is applied multiple times. Assume that the network currently deals with images of size $j \times j$. After training the GAN with a specific number of training samples (to get samples of the current size, a downsampling operator $d(\cdot)$ is applied to the given data as in LAPGAN), the generator and discriminator networks are both enlarged to be able to process images of size $2j \times 2j$. Note that before adding new layers to the generator the last layer, which always drastically reduces the number of features (to 3 for colour images or 1 for gray scale images) has to be removed. A similar statement holds for the first layer of the discriminator. Further note that except for the first layer in the generator and the last layer of the discriminator, all other layers of the discriminator mirror the layers of the generator. This means that when a transposed convolutional layer is added in the end of the generator then a similar convolutional layer is added at the beginning of the discriminator. Similar upsampling is mirrored by downsampling.

After adding new layers, there is a transition training phase in which the new layers are faded in smoothly, so that the already well trained smaller-resolution layers are not suddenly shocked. In the generator the output of the previous last layer is upsampled, weighted with $1 - \alpha$ and added up with the output of the new last layer which is weighted with α. The same technique but mirrored is used for the discriminator. During the transition phase the parameter α is linearly changed from 0 to 1.

For training samples of size $2^{N+2} \times 2^{N+2}$ this training-and-growing step is executed N times. It is important to note that all layers stay trainable throughout the whole training process. Still we can think of the generator as a composition of the layers that were added in each training step, $G_\theta = G_N \circ \cdots \circ G_1$. Beside stabilizing the training process, the progressive growing also has the advantage of reducing the training time, since a big part of the training is done with lower resolutions.

Karras et al. suggest to use the WGAN-GP training procedure but with $n_{\text{disc}} = 1$ and an extra term added to the discriminator loss function (2.12). This loss term prevents the discriminator output from drifting to far away

from 0 and is given by

$$\lambda_{\text{drift}}\, \mathbb{E}_{X\sim\mathbb{P}_r}\left[\, D(X)^2 \,\right],$$

where the parameter $\lambda_{\text{drift}} > 0$ is set to the small value 0.001. Furthermore, instead of using batch normalization a pixel-wise normalization to unit length over the parallel feature maps is applied after each convolutional layer. This is only done in the generator, while the discriminator does not include any normalizations. But it includes a mini-batch standard deviation layer, which deterministically produces one additional feature map out of the others. This layer is added towards the end of the discriminator with the purpose to counteract mode collapse.
The PGGAN method was able to generate high-resolution images (up to 1024×1024 pixels on the CelebA-HQ dataset [46]) of a quality that outperformed all previous methods.

2.5 Comparison of different GAN variants

In this section we compare some of the GANs that we have seen in the previous sections. As we outlined in Section 2.2 this is not a straight forward task. Here we choose the *Inception Score* (IS) [66] on the CIFAR-10 dataset [51] as main benchmark, since this is the score that is provided most often.
The CIFAR-10 dataset consists of 60'000 RGB images (often only the 50'000 training images are used) in 10 equally sized categories: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. Each image is of size $33 \times 32 \times 3$.
Salimans et al. invented the IS to mirror human judgement for a set of images that is distributed according to $\mathbb{P}_I$. In particular, the IS aims to judge the quality of samples as well as their diversity. It makes use of the pre-trained *Inception model* [1] (a NN classifier) which returns, for an input image x, the predicted conditional probability that x belongs to class y, denoted $P(y|x)$. With the marginal $P(y) = \int P(y|x)d\mathbb{P}_I(x)$, the Inception Score is defined as

$$\mathrm{IS}(\mathbb{P}_I) := \exp(\mathbb{E}_{X\sim\mathbb{P}_I}\left[\,\mathrm{KL}\left(P(\cdot|X)\,\|\,P(\cdot)\right)\,\right]).$$

For a generative model G_θ producing images distributed like $\mathbb{P}_G$ we write $IS(G_\theta)$. Beside talking about the issues of IS, Barratt and Sharma [12, Chapter 3.3] showed that the IS can be written with the (conditional) entropies as

$$\mathrm{IS}(\mathbb{P}_I) = \exp(\mathrm{H}(Y) - \mathrm{H}(Y|X)).$$

Here (X, Y) is a pair of random variables distributed according to the joint measure $P(x, y) = P(y|x)\mathbb{P}_I(x)$. If the diversity of the set of images is high,

the entropy $\mathrm{H}(Y)$ is large since all classes are likely to appear. Similar, if the image quality of each sample is good, the classifier should always be quite confident about its prediction yielding a small conditional entropy $\mathrm{H}(Y|X)$. Hence, high diversity and good image quality lead to a high IS. In practice, $\mathbb{P}_I$ is approximated by the empirical measure associated with a given set of images. This set is typically chosen of size 50'000.
We list all available Inception Scores in Table 1 in the order the models were presented. All scores were taken from the respective papers (unless mentioned otherwise) and are given in the format mean $\pm$ standard deviation (the Inception Score calculator provides mean and standard deviation by splitting up the input dataset into 10 subsets). Usually, only one model is trained to get the IS which can lead to lucky configurations overestimating the score. Or the best score out of many training runs might be picked and displayed. Since most papers do not clarify how exactly the IS was computed, one should have these things in mind when comparing the methods. For PGGAN the authors provided two Inception Scores, the overall highest one and an average of the highest scores seen in 10 different runs starting with randomly initialized weights. For the latter, the standard deviation was computed from the 10 means. As one can see in Table 1, the difference is not irrelevant.
Before comparing different models, we emphasize that there does not exist the single best GAN variant. If at all, there exist methods that are best suited for specific tasks. One always has to consider the training time and available resources on the one side and sample diversity and quality of the generator on the other side.

We start with a discussion about the right loss function to use (confer Section 2.3). We have already argued that the Wasserstein distance, especially in the WGAN-GP formulation, has many advantages over the standard Jensen-Shannon divergence. In particular, when the architecture of the generator and discriminator is changed, for example by using deeper networks, residual blocks or by going without batch normalization, the Wasserstein distance can stabilize the training significantly. For that reason the WGAN-GP loss is used in SplitGAN as well as PGGAN. But already when combining this loss with a residual network structure, mode collapse issues can be reduced and image quality improved. This is reflected by the score of WGAN-GP ResNet. However, this comes at the price of higher training time, also because n_{critic} is set to 5 instead of 1. Hence, the standard DCGAN might be preferable, especially when other measures are taken to improve the model. If training time does not matter, the most appealing loss function up to now is the minibatch energy distance used by OT-GAN. To reach the high score a batch size of 8000 was used and the model was trained on 8 GPUs for several

Method (unsupervised)	**Inception Score**
DCGAN [63] (score from [41])	6.16 ± 0.07
WGAN [9] (score from [38])	3.82 ± 0.06
WGAN-GP ResNet [32]	7.86 ± 0.07
Sinkhorn GAN [26]	4.81 ± 0.05
OT-GAN [67]	8.47 ± 0.12
MMD GAN [54]	6.17 ± 0.07
GMMN [55] (in [54])	3.94 ± 0.04
MGAN [38]	8.33 ± 0.10
D2GAN [59]	7.15 ± 0.07
SplitGAN [31]	7.90 ± 0.09
PGGAN (averaged) [46]	8.56 ± 0.06
PGGAN (best) [46]	8.80 ± 0.05
Method (supervised)	
DCGAN with labels (score from [79])	6.58
WGAN-GP ResNet with labels [32]	8.42 ± 0.10
AC-GAN [60]	8.25 ± 0.07
SplitGAN with labels [31]	8.87 ± 0.09
SGAN-no-joint [41]	8.37 ± 0.08
SGAN [41]	8.59 ± 0.12
true data (score from [38])	11.24 ± 0.12

Table 1: Inception Scores on CIFAR-10 in format mean $\pm$ standard deviation. Higher is better.

days. We see that this loss function alone can improve GANs as much as different GAN architectures designed to reduce mode collapse and improve image quality.

When comparing different GAN designs (confer Section 2.4) it is intuitively clear that supervised methods (those using class labels) should achieve better scores than unsupervised ones, since they have additional information. This can also be seen in Table 1. But note that the best unsupervised versions outperform many of the supervised ones.

We use MGAN as representative for all GAN variants that implement a Gaussian mixture on the latent space or parallel generators. We see that these methods are successful in reducing mode collapse and concurrently can improve image quality, since each generator can specialize on different modes. Compared to the D2GAN which uses two discriminators and the unsupervised SplitGAN which artificially creates labels, we see that MGAN achieves the best score. We explain this by the better suited approach to use a multi-

modal distribution on the latent space when the data distribution seems to be multi-modal.
Considering the methods with stacked generators, we see that SGAN achieves a very good IS that is higher than the ones of OT-GAN and MGAN. However, note that SGAN uses class labels in contrast to the other two methods. This label information that is passed to the generator as one-hot vector together with the random latent vector accomplishes a similar job as a mixture on the latent space. But it has the advantage that this information is already given and correct. Furthermore, it seems like the stacked GANs help to produce samples of better quality. We already outlined that PGGAN was able to generate high-resolution images of a quality that was not reached by any other method so far. Moreover, we see that the progressive training procedure also helps when the target resolution is quite small. In particular, PGGAN achieved the highest IS without using class labels up to now. We remark that the averaged score is more meaningful than the other one. PGGAN is definitely the state-of-the-art model considering image quality. But as all high-resolution experiments were conducted on datasets consisting only of one category of images, we do not know how well PGGAN can prevent mode collapse there. Hence, it might be interesting to combine it with the idea of e.g. MGAN.

3 Combining the Johnson-Lindenstrauss Lemma with the SWG

In this chapter we present our idea to extend the *sliced Wasserstein generator* (SWG) model that was introduced by Deshpande et al. [22]. We propose a generative network that is very similar to the SWG, but makes use of the Johnson-Lindenstrauss Lemma [19, Theorem 2.1] at a certain point. In many settings, this leads to a significant reduction of training time, while resulting in a generative model of similar quality. We first review the Johnson-Lindenstrauss Lemma in Section 3.1 and then go on to describe our model and our experimental results in the subsequent sections.

3.1 Johnson-Lindenstrauss Lemma

The Johnson-Lindenstrauss Lemma was first published in the work of W. Johnson and J. Lindenstrauss on the extension of Lipschitz mappings into a Hilbert space [45]. In the following, we will refer to the more recent paper by Dasgupta et al. [19] in which the authors give an elementary proof of a stronger version of the result. We state this version of the Johnson-Lindenstrauss Lemma as Theorem 3.1 below.

Theorem 3.1. *Let n, k be positive integers. For any $0 < \epsilon < 1$ let m be an integer satisfying*

$$m \geq \frac{4\ln(k)}{\epsilon^2/2 - \epsilon^3/3}.$$

Then for any subset V of $\mathbb{R}^n$ consisting of k elements, there exists a map $f : \mathbb{R}^n \to \mathbb{R}^m$ such that for all $u, v \in V$ it holds that

$$(1-\epsilon)\|u-v\|_2^2 \leq \|f(u)-f(v)\|_2^2 \leq (1+\epsilon)\|u-v\|_2^2. \tag{3.1}$$

This map can be found in randomized polynomial time.

For the proof of Theorem 3.1 we refer the reader to the proof in [19]. We call (3.1) the *approximate distance preserving property* and a map f that satisfies it is denoted *Johnson-Lindenstrauss transformation.*

We briefly summarize the nice property of Johnson-Lindenstrauss transformations in words. Starting with a set of k arbitrary elements in $\mathbb{R}^n$ we can embed these elements into the space $\mathbb{R}^m$, such that their pairwise distances are approximately preserved. This means, that the geometric structure of the set is approximately preserved by the Johnson-Lindenstrauss transformation. Furthermore, the target dimension m is independent of the original

dimension n. It depends only on the number k of elements that have to be embedded as well as the maximal allowed error ϵ. Therefore, it is possible that Johnson-Lindenstrauss transformations lead to huge dimensionality reductions, especially when n is large and $\ln(k)$ is small. On the other hand, when n is small and $\ln(k)$ is large, Johnson-Lindenstrauss transformations are not very useful.

We are especially interested in constructing Johnson-Lindenstrauss transformations. There does not exist a deterministic way to do this. But there are several possibilities to define mappings that will satisfy (3.1) with positive probability. As outlined in [19, Section 3], there are some particularly easy ways to construct such a mapping. The first two maps, which were proposed in [6], are constructed as follows. Let $A \in \mathbb{R}^{m\times n}$ be a matrix with independent entries either distributed like $\frac{1}{2}\delta_1 + \frac{1}{2}\delta_{-1}$ or like $\frac{1}{6}\delta_{\sqrt{3}} + \frac{1}{6}\delta_{-\sqrt{3}} + \frac{2}{3}\delta_0$. In the first case this means, taking 1 or -1 as entry with the same probability. Then

$$f : \mathbb{R}^n \to \mathbb{R}^m, x \mapsto \tfrac{1}{\sqrt{m}}Ax$$

is a Johnson-Lindenstrauss transformation with positive probability. Another possibility suggested in [43] uses i.i.d entries of the standard normal distribution $N(0,1)$ for the matrix $A \in \mathbb{R}^{m\times n}$ and defines the map

$$f : \mathbb{R}^n \to \mathbb{R}^m, x \mapsto \tfrac{1}{\sqrt{n}}Ax.$$

Again this is a Johnson-Lindenstrauss transformation with positive probability. Note the different scaling factors in the two definitions of f. Both constructions need a slightly bigger lower bound for M than the one given in Theorem (3.1). The exact bounds are omitted here, since they will not be important for us, but can be looked up in the respective papers.

3.2 The generative model

As mentioned before, our model is an extension of the *sliced Wasserstein generator* (SWG) [22]. Although we already explained the SWG in Section 2.3 we briefly go through its structure again to clarify and motivate the construction of our model. We use the notation that was introduced in the beginning of Section 2.1. In particular we assume that $\mathcal{X} \subset \mathbb{R}^n$ and we use the integer B as mini-batch size. Moreover, we use the following notation.

Notation 3.2. Let $\mathcal{X}$ and $\mathcal{Y}$ be two spaces and I a set of indices. For a set $\mathbf{X} = \{x_i\}_{i\in I} \subset \mathcal{X}$ and a map $f : \mathcal{X} \to \mathcal{Y}$ we denote $f(\mathbf{X}) := \{f(x_i)\}_{i\in I}$. In particular, for a linear map A we write $A\mathbf{X}$.

Notation 3.3. For two sets $\mathbf{X} = \{x_1, \dots, x_k\}$ and $\mathbf{Y} = \{y_1, \dots, y_k\}$ and their respective empirical distributions $\hat{\mathbb{P}}_{\mathbf{X}}$ and $\hat{\mathbb{P}}_{\mathbf{Y}}$ we denote the Wasserstein-p distance

$$\mathrm{W}_p(\mathbf{X}, \mathbf{Y}) := \mathrm{W}_p(\hat{\mathbb{P}}_{\mathbf{X}}, \hat{\mathbb{P}}_{\mathbf{Y}}).$$

3.2.1 JL-SWG

The SWG uses a standard generator G_θ but does not necessarily need a discriminator. In particular, the generator is trained to minimize an approximation of the squared sliced Wasserstein-2 distance (2.9) between $\mathbb{P}_G$ and $\mathbb{P}_r$. In contrast to the objective functions for the standard GAN generator and the Wasserstein GAN generator, this distance can be approximated without the need of a discriminator that has to be trained concurrently. First, the probability measures $\mathbb{P}_r$ and $\mathbb{P}_G$ are approximated by their empirical distributions $\hat{\mathbb{P}}_r$ and $\hat{\mathbb{P}}_G$ on the batches $\mathbf{X} = \{x_1, \dots, x_B\}$ and $\mathbf{Y} = \{y_1, \dots, y_B\}$. Here, the elements in $\mathbf{X}$ are randomly sampled from $\mathbb{X}$ (in fact not entirely at random, since in each epoch the dataset is shuffled and then split up in mini-batches) and the elements in $\mathbf{Y}$ are randomly sampled using the generator. Secondly, the integral in (2.9) over $\mathbb{S}^{n-1}$ is approximated with the Monte Carlo method, i.e. a large number L of elements $\{\lambda_1, \dots, \lambda_L\} \subset \mathbb{S}^{n-1}$ is randomly sampled from a uniform distribution on $\mathbb{S}^{n-1}$ and the integral is approximated by the mean using these elements. Putting this all together, we arrive at the loss function (when no discriminator is used) that G_θ is trained to minimize

$$\frac{1}{L} \sum_{i=1}^{L} \mathrm{W}_2(\lambda_i^T \mathbf{X}, \lambda_i^T \mathbf{Y})^2. \tag{3.2}$$

L is typically chosen as 10'000 and the Wasserstein-2 distances occurring here can be computed efficiently with the closed form described in Section 2.3.1.

The resulting distance (3.2) is an approximation of the squared Wasserstein-2 distance $\mathrm{W}_2(\mathbf{X}, \mathbf{Y})^2$. Equality (2.10) implies that the Wasserstein-2 distance between two finite sets depends only on the pairwise distances of their elements. As discussed in Section 3.1, Johnson-Lindenstrauss transformations can embed a finite number of points in a probably much lower dimensional space, while approximately preserving their pairwise distances. Therefore, it should intuitively be clear that the squared Wasserstein-2 distance $\mathrm{W}_2(\mathbf{X}, \mathbf{Y})^2$ is also approximately preserved by a Johnson-Lindenstrauss transformation, that embeds the $2B$ elements $\mathbf{X} \cup \mathbf{Y}$ in $\mathbb{R}^m$. We will discuss how to choose m later. This intuition is rigorously proven in Proposition 3.4 below. So instead of approximating $\mathrm{W}_2(\mathbf{X}, \mathbf{Y})^2$ with the sliced

Wasserstein-2 distance, one can first apply a Johnson-Lindenstrauss transformation $f : \mathbb{R}^n \to \mathbb{R}^m$ to the batches, yielding $f(\mathbf{X})$ and $f(\mathbf{Y})$. Afterwards the squared Wasserstein-2 distance $\mathrm{W}_2\left(f(\mathbf{X}), f(\mathbf{Y})\right)^2$ can be approximated by the squared sliced Wasserstein-2 distance.

Proposition 3.4. *Let n, m, B be integers and $0 < \epsilon < 1$. Furthermore, let $\mathbf{X} = \{x_1, \ldots, x_B\} \subset \mathcal{X}$ and $\mathbf{Y} = \{y_1, \ldots, y_B\} \subset \mathcal{X}$, where $\mathcal{X}$ is a subset of $\mathbb{R}^n$. Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a Johnson-Lindenstrauss transformation for $\mathbf{X} \cup \mathbf{Y}$ with error ϵ, i.e. the approximate distance preserving property* (3.1) *is satisfied. Then also the squared Wasserstein-2 distance is approximately preserved by f, i.e.*

$$(1-\epsilon)\ \mathrm{W}_2(\mathbf{X}, \mathbf{Y})^2 \leq \mathrm{W}_2(f(\mathbf{X}), f(\mathbf{Y}))^2 \leq (1+\epsilon)\ \mathrm{W}_2(\mathbf{X}, \mathbf{Y})^2.$$

Furthermore, if $\mathcal{X}$ is compact with diameter D the following holds

$$\mathrm{W}_2(\mathbf{X}, \mathbf{Y})^2 - \epsilon D^2 \leq \mathrm{W}_2(f(\mathbf{X}), f(\mathbf{Y}))^2 \leq \mathrm{W}_2(\mathbf{X}, \mathbf{Y})^2 + \epsilon D^2.$$

Proof. Equality (2.10) tells us that we can write

$$\mathrm{W}_2(\mathbf{X}, \mathbf{Y})^2 = \frac{1}{B} \min_{\sigma \in \mathcal{P}_B} \sum_{i=1}^{B} \|x_i - y_{\sigma(i)}\|_2^2,$$

where $\mathcal{P}_B$ is the set of permutations of B elements. Let σ^* be the optimal permutation achieving the minimum. Then

$$\begin{aligned}
\mathrm{W}_2(f(\mathbf{X}), f(\mathbf{Y}))^2 &= \frac{1}{B} \min_{\sigma \in \mathcal{P}_B} \sum_{i=1}^{B} \|f(x_i) - f(y_{\sigma(i)})\|_2^2 \\
&\leq \frac{1}{B} \sum_{i=1}^{B} \|f(x_i) - f(y_{\sigma^*(i)})\|_2^2 \\
&\leq (1+\epsilon) \frac{1}{B} \sum_{i=1}^{B} \|x_i - y_{\sigma^*(i)}\|_2^2 \\
&\leq \frac{1}{B} \sum_{i=1}^{B} \|x_i - y_{\sigma^*(i)}\|_2^2 + \epsilon D^2,
\end{aligned}$$

proves the claimed upper bounds. For the second inequality we used (3.1) and for the third we assumed that $\mathcal{X}$ is compact with diameter D. The claimed lower bounds follow in a similar way. Let σ^{**} be the optimal permutation for

$\mathrm{W}_2(f(\mathbf{X}), f(\mathbf{Y}))^2$, then we have

$$\begin{aligned}
\mathrm{W}_2(f(\mathbf{X}), f(\mathbf{Y}))^2 &= \frac{1}{B}\sum_{i=1}^{B}\|f(x_i) - f(y_{\sigma^{**}(i)})\|_2^2 \\
&\geq (1-\epsilon)\frac{1}{B}\sum_{i=1}^{B}\|x_i - y_{\sigma^{**}(i)}\|_2^2 \\
&\geq (1-\epsilon)\frac{1}{B}\min_{\sigma\in\mathcal{P}_B}\sum_{i=1}^{B}\|x_i - y_{\sigma(i)}\|_2^2 \\
&\geq \frac{1}{B}\min_{\sigma\in\mathcal{P}_B}\sum_{i=1}^{B}\|x_i - y_{\sigma(i)}\|_2^2 - \epsilon D^2.
\end{aligned}$$

Here we used (3.1) for the first inequality and assumed that $\mathcal{X}$ is compact with diameter D for the third inequality. □

As we have discussed in Section 3.1, it is not straight forward to construct a Johnson-Lindenstrauss transformation, although there are easy ways to get linear mappings which satisfy (3.1) with positive probability. One could invest effort into finding a map that really satisfies this property, by checking the pairwise distances (which is expensive) and choosing a new map if necessary. Instead of doing so, we simply use a map that has positive probability to be a Johnson-Lindenstrauss transformation without further verification. In the following we will denote such a map *Johnson-Lindenstrauss map* (JL map). However, we make use of the fact that the loss has to be recomputed in every iteration of the training process, which means, that in every iteration we can choose a new JL map. Let p be the probability that a JL map is a Johnson-Lindenstraus transformation. Then we expect to get a meaningful loss in $100p$ percent of all iterations. But even if the JL map does not satisfy (3.1), it might still lead to a useful generator update.
So instead of sampling random projection directions in $\mathbb{S}^{n-1}$, we have to sample random projection directions in $\mathbb{S}^{m-1}$ and a random JL map in every iteration of the training. For choosing the JL map, we suggest to use one of the methods outlined in Section 3.1. In our experiments we chose the JL map to be a matrix $A \in \mathbb{R}^{m\times n}$ with i.i.d. entries coming from $N(0,1)$. We note at this point that any scaling factor γ for A would only rescale the entire loss function by γ^2, so that it can also be omitted. Defining $f_A : x \mapsto Ax$, this leads to the new loss function

$$\frac{1}{L}\sum_{i=1}^{L}\mathrm{W}_2\left(\lambda_i^T f_A(\mathbf{X}), \lambda_i^T f_A(\mathbf{Y})\right)^2. \tag{3.3}$$

We call a model that is trained to minimize this loss function *Johnson-Lindenstrauss sliced Wasserstein generator* (JL-SWG). The entire training procedure is summarized in Algorithm 1.

Algorithm 1: JL-SWG training procedure

Given: B, L, n, m, latent space dimension d, randomly initialized generator weights θ, number iterations I, training dataset $\mathbb{X}$, learning rate α

for $i \leftarrow 1$ **to** I **do**
  sample latent vectors $\{z_j\}_{j=1}^B \sim \mathbb{P}_Z$;
  sample training data $\{x_j\}_{j=1}^B \subset \mathbb{X}$;
  sample JL map $A \in \mathbb{R}^{m \times n}$;
  sample random projection directions $\{\lambda_j\}_{j=1}^L \subset \mathbb{S}^{m-1}$;
  generate samples $\{y_j\}_{j=1}^B$, with $y_j \leftarrow G_\theta(z_j)$;
  `// compute generator loss`;
  $L_{G_\theta} \leftarrow 0$;
  for $k \leftarrow 1$ **to** L **do**
    $\{\tilde{x}_j\}_{j=1}^B \leftarrow \mathrm{sort}(\{\lambda_k^T(Ax_j)\}_{j=1}^B)$;
    $\{\tilde{y}_j\}_{j=1}^B \leftarrow \mathrm{sort}(\{\lambda_k^T(Ay_j)\}_{j=1}^B)$;
    $L_{G_\theta} \leftarrow L_{G_\theta} + \frac{1}{BL}\sum_{j=1}^B \|\tilde{x}_j - \tilde{y}_j\|_2^2$
  end
  update weights $\theta \leftarrow \theta - \alpha \nabla_\theta L_{G_\theta}$
end

Let us compare the JL-SWG with the SWG. To this end, we fix a batch size B and a number of projection directions L, which are used for both models. Our intuitive explanation together with Proposition 3.4 should already have made it clear, why we expect a trained JL-SWG to behave similar to a trained SWG, if m is chosen reasonably. To see why the JL-SWG training can be faster than the one of SWG we compare the number of needed random samples, as well as the number of elementary operations (EOps) which are needed for the transformation and projection. It is important to note, that the projection directions $\lambda \in \mathbb{S}^{n-1}$ respectively $\mathbb{S}^{m-1}$ are sampled by taking i.i.d. $N(0,1)$ entries and normalizing the vector. It was shown in [62] that this indeed yields elements of a uniform distribution on the sphere and is very efficient. Computing the sliced Wasserstein-2 distance, is equally expensive for both models.
The number of random $N(0,1)$ samples needed in one iteration of the JL-SWG training is mn for the JL map and mL for the projection directions.

Compared to the nL samples in for the SWG, we see that this number is smaller for JL-SWG if $m < \frac{nL}{n+L}$. In particular, this is the case whenever $m < \frac{1}{2}\min(n, L)$ holds true.
In every iteration of the JL-SWG approach, $2B$ samples are projected from $\mathbb{R}^n$ to $\mathbb{R}^m$ via the linear JL map f_A. This needs $2B(2n-1)m$ EOps. Then, these $2B$ samples are projected from $\mathbb{R}^m$ to $\mathbb{R}$ with L different directions λ_i, needing $2B(2m-1)L$ EOps. On the other hand, in the iterations of the SWG training we directly project the $2B$ samples from $\mathbb{R}^n$ to $\mathbb{R}$ with L different directions, which needs $2B(2n-1)L$ EOps. Therefore, the JL-SWG needs less EOps whenever $m < \frac{2nL}{2n+2L-1}$. This holds in particular if $m < \frac{nL}{n+L}$. We also remark that multiplying the two projections first in the JL-SWG would always lead to a bigger number of EOps than used in the SWG.
Finally, we count the EOps to normalize the projection directions. For the JL-SWG these are $3mL$ compared to $3nL$ for the SWG. Hence, this is always smaller for JL-SWG, since we assume that m is chosen smaller than n.
In sum, we see that choosing $m < \frac{nL}{n+L}$ leads to fewer computations and fewer random samples. The resulting time reduction per iteration depends also on the parameters n and L. The bigger they are and the smaller m is in relation to them, the more time will be saved. We remark that it is preferable to choose L large, since then the Monte Carlo approximation of the integral in (2.9) is more accurate. Using the JL-SWG approach, the increase in training time with growing L can be heavily reduced.

Having this in mind, we now discuss how m can be chosen. Out of the reasons discussed before, we use $\frac{nL}{n+L}$ as upper bound for m. Theorem 3.1 yields the lower bound $\frac{4\ln(2B)}{\epsilon^2/2-\epsilon^3/3}$ for m, but here ϵ needs to be specified. Very small values for ϵ would often lead to a lower bound bigger than $\frac{nL}{n+L}$. Since it is not clear how big ϵ can be chosen to still lead to a meaningful loss function, searching for a suitable ϵ would only be a substitute for searching for m. Therefore, we propose to treat m as another hyper-parameter of the model, with $\frac{nL}{n+L}$ as upper bound. In our experiments, values between 200 and 600 were used.

3.2.2 JL-SWGAN

Until now we have only focused on the case where no discriminator is used. Although this is theoretically appealing, the training results on more complex datasets are not satisfying. When Deshpande et al. introduced the SWG [22], they also proposed a way to change the design of the SWG, such that it also includes a discriminator. Although this extension is somehow artificial, it improves the training results. Furthermore, the Johnson-Lindenstrauss

Lemma can be applied similar as before, with the same theoretical justification. Again we briefly explain the structure of the SWG with discriminator and our extension of it. In addition to the generator G_θ, also a discriminator $D_\omega : \mathcal{X} \to [0,1]$ is used, which maps elements of the target space $\mathcal{X}$ to their probability to come from the true training data $\mathbb{X}$. This discriminator is trained with the same objective as the discriminator of the standard GAN, i.e. to minimize the binary cross entropy

$$\min_\omega \mathbb{E}_{X\sim\mathbb{P}_r}\left[-\log(D_\omega(X))\right] + \mathbb{E}_{Z\sim\mathbb{P}_Z}\left[-\log(1-D_\omega(G_\theta(Z)))\right].$$

Let D'_ω be an flattened intermediate layer of the discriminator (normally chosen to be the output of last hidden layer) and let k be its dimension. Then the generator of the SWG model is trained to minimize

$$\frac{1}{L}\sum_{i=1}^{L} \mathrm{W}_2\left(\lambda_i^T D'_\omega(\mathbf{X}), \lambda_i^T D'_\omega(\mathbf{Y})\right)^2,$$

where λ_i are randomly sampled from a uniform distribution on $\mathbb{S}^{k-1}$. Deshpande et al. motivate this by the assumption that a good discriminator always tries to extract features with which the generated samples and the training data samples are easier to distinguish. This in turn helps the generator to improve, since the discriminator, in a way, tells it where it currently performs worst.

Here, the dimension reduction using a JL map can be applied to the discriminator output. In each iteration, we sample a JL map $A \in \mathbb{R}^{m\times k}$ and define f_A as before in addition to the random projections sampled from $\mathbb{S}^{m-1}$. Then, the generator G_θ is trained to minimize

$$\frac{1}{L}\sum_{i=1}^{L} \mathrm{W}_2\left(\lambda_i^T f_A(D'_\omega(\mathbf{X})), \lambda_i^T f_A(D'_\omega(\mathbf{Y}))\right)^2,$$

while the discriminator is trained as before. The training procedure is summarized in Algorithm 2, for the case that the discriminator is updated once per generator update. We call a model that is trained with this procedure a *Johnson-Lindenstrauss sliced Wasserstein generative adversarial network* (JL-SWGAN).

Comparing the training time of JL-SWGAN to JL-SWG, we see an increase at two points. First, the forward and backward passes through the network take approximately twice as long, since the added discriminator usually is of similar size as the generator. Moreover, updating the discriminator once per generator update means that in each iteration double work is done.

However, Deshpande et al. empirically showed that the SWG is also stable when updating the discriminator only once per a few generator updates [22, Appendix C]. Assuming that the same is true for JL-SWGAN, this would be a way to reduce the amount of additional training time.
The increased training time usually comes with the benefit that the iteration-wise improvement is higher. This is due to the discriminator which guides the generator in the right direction. Furthermore, the final quality of the generator can be much better when it is trained with a discriminator, as will be shown in Section 3.3.

Algorithm 2: JL-SWGAN training procedure

Given: B, L, k, m, latent space dimension d, randomly initialized generator weights θ, randomly initialized discriminator weights ω, number iterations I, learning rate α, training dataset $\mathbb{X}$

for $i \leftarrow 1$ **to** I **do**
  sample latent vectors $\{z_j\}_{j=1}^{B} \sim \mathbb{P}_Z$;
  sample training data $\{x_j\}_{j=1}^{B} \subset \mathbb{X}$;
  sample JL map $A \in \mathbb{R}^{m \times k}$;
  sample projection directions $\{\lambda_j\}_{j=1}^{L} \subset \mathbb{S}^{k-1}$;
  $\{y_j\}_{j=1}^{B} \leftarrow \{G_\theta(z_j)\}_{j=1}^{B}$;
  `// compute generator loss & update generator`;
  $L_{G_\theta} \leftarrow 0$;
  for $l \leftarrow 1$ **to** L **do**
    $\{\tilde{x}_j\}_{j=1}^{B} \leftarrow \text{sort}\left(\{\lambda_l^T (A\, D'_\omega(x_j))\}_{j=1}^{B}\right)$;
    $\{\tilde{y}_j\}_{j=1}^{B} \leftarrow \text{sort}\left(\{\lambda_l^T (A\, D'_\omega(y_j))\}_{j=1}^{B}\right)$;
    $L_{G_\theta} \leftarrow L_{G_\theta} + \frac{1}{BL} \sum_{j=1}^{B} \|\tilde{x}_j - \tilde{y}_j\|_2^2$;
  end
  $\theta \leftarrow \theta - \alpha \nabla_\theta L_{G_\theta}$;
  sample new training data $\{x_j\}_{j=1}^{B} \subset \mathbb{X}$;
  sample new latent vectors $\{z_j\}_{j=1}^{B} \sim \mathbb{P}_Z$;
  `// compute discriminator loss & update discriminator`;
  $\{y_j\}_{j=1}^{B} \leftarrow \{G_\theta(z_j)\}_{j=1}^{B}$;
  $L_{D_\omega} \leftarrow -\frac{1}{B} \sum_{j=1}^{B} \log(D_\omega(x_j)) + \log(1 - D_\omega(y_j))$;
  $\omega \leftarrow \omega - \alpha \nabla_\omega L_{D_\omega}$;
end

3.3 Experimental Results

We conduct a series of experiments to show the usefulness of our extention of the SWG[1]. We extensively compare our JL-SWG and JL-SWGAN models to their SWG counterparts. Due to the similar properties after the generative models have been trained, we do not compare our approach to other GAN versions. This was already done by Deshpande et al. [22], whereby their conclusions also hold true for our extension of the SWG.
The main difference between the SWG without discriminator and our extension of it lies in the needed training time, which was one of the concerns of Deshpande et al. [22, Section 4.5]. When using a discriminator, the two models behave very similarly.
To be able to compare the models, other then by eyeballing generated samples, we use the sliced Wasserstein-2 distance (SWD) computed between generated samples and samples of the test set (this is a set of target samples that is not shown to the model during training). We approximate the SWD using L random projection directions and batch size B, where L and B are the same as used for training the models. Whenever computing the SWD for evaluation, the entire test set (usually of size 10'000) is used, by splitting it up into batches of size B and averaging the distances computed on the single batches. We remark that for the SWG without discriminator, this is exactly the distance which it is trained to minimize on the training set. Furthermore, we note that the SWD decreases with increasing batch size B (confer [22, Section 4.1]). This should be clear from the way in which the Wasserstein-2 distance is computed, namely by sorting the elements of the sets such that the minimal distances are achieved. Therefore, the SWD can't be used to compare models that were trained with different batch sizes. This could be bypassed by fixing a batch size $\tilde{B}$ that is always used to compute the SWD for evaluation, independent of the batch size B used for training. However, this was not done in our experiments.
We also remark that the SWD alone is not enough to assess and compare the quality of models with different architectures. This will be discussed in Section 3.3.5.

For our experiments we use a toy dataset as well as the MNIST [53], CelebA [56] and CIFAR-10 [51] datasets. The latter 3 are image datasets. Images are usually stored as unsigned 8-bit integer arrays. This means, every pixel is represented by an integer value in $[0, 255]$ for grey-scale images, respectively a 3-tuple in $[0, 255]^3$ specifying the RGB values for color images. To make the generator training possible, a linear respectively affine transformation is used to rescale the pixel values to lie in $[0, 1]$ respectively $[-1, 1]$.

[1]Code available at `https://github.com/FlorianKrach/A-Study-of-GANs.git`.

Depending on this choice, either a sigmoid or a tanh activation function is used in the last layer of the generator. The generated arrays are transformed back to have integer values in $[0, 255]$, so that they can be displayed as images.

If not mentioned otherwise, we use 128-dimensional latent variables drawn from a multivariate standard normal distribution. Moreover, we train the networks with stochastic gradient descent using the Adam optimizer with $\beta_1 = 0.5$ and default values for β_2 and ϵ.

JL-SWGAN and SWG with discriminator are always trained with one discriminator update per generator update. Hence, in each iteration 2 data batches are used. This is taken into account when computing the number of epochs (in each epoch the entire training data set is used once).
For the discriminator we test batch normalization as well as layer normalization, which was used by Deshpande et al. [22]. We remark that in all our experiments, layer normalization was computationally more expensive than batch normalization. We do not always test both variants. If not mentioned otherwise, batch normalization is used.

All experiments were done using CPUs only. Whenever comparing training times, all involved models were trained on a similar CPU (using 1 CPU per model). Although the state of the art for neural network training is to use GPUs, we claim that the qualitative insights we made using CPUs also apply when using GPUs. Especially the ratios between training times should stay valid.

3.3.1 Toy dataset

In our theoretical analysis, we already claimed that applying a Johnson-Lindenstrauss map is the more effective the larger the target dimension n and the number of projection directions L are. To test this, we construct a toy dataset consisting of random Gaussian samples in an arbitrarily high dimensional space. Fixing the dimension n, we first choose a mean vector μ of a uniform distribution on $[0.2, 0.8]^n$, fix $\sigma := 0.01$ and then sample 11'000 random points of $N(\mu, \sigma^2 I_n)$, where I_n is the n-dimensional unit matrix. Afterwards we clip these samples to $[0, 1]^n$, by replacing each coordinate by 0 respectively 1 if it is smaller than 0 respectively bigger than 1. The way the mean was chosen together with the small standard deviation σ, the clipping should not be needed at all. Finally, we split up the set into a train set of 10'000 samples and a test set of 1'000 samples.
This dataset, respectively its distribution, should be easy to learn (much easier than any image dataset). We assume that already a simple generator architecture and a relatively small number of iterations are sufficient for successful training. This is desirable, since typically the training of GANs is

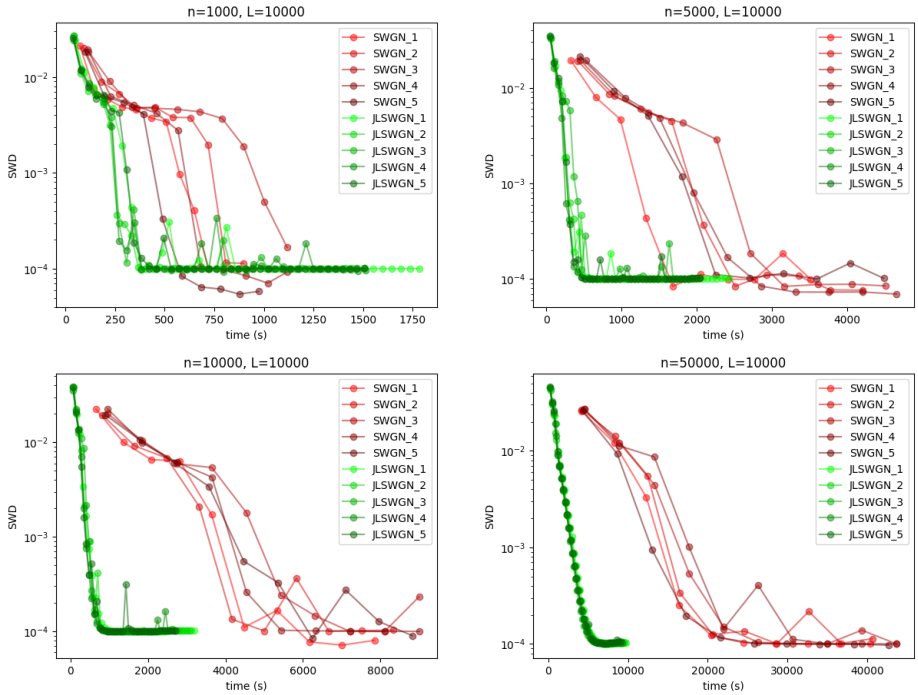


Figure 4: Comparison of JL-SWG and SWG on toy datasets.

very expensive, especially in high dimensional spaces.
We use the following architecture for the SWG and JL-SWG. The 128-dimensional latent variable is mapped with 3 fully-connected layers to $[0, 1]^n$. The two hidden layers use ReLU activation functions and their output dimension is 256. The last layer uses a sigmoid activation function. All layers use batch normalization. The batch size B is 50 and the number of projection directions L is 10'000 (this is the suggested number in [22]). We choose the target dimension n in $\{1000, 5000, 10'000, 50'000\}$. For the JL-SWG we always use a JL mapping that reduces the dimension to $m = 360$. This mapping is chosen as explained in Section 3.2. We train all models with learning rate 0.01.
The SWG is trained for 1000 iterations (with batch size 50 and 10'000 training samples this equals 5 epochs) while the JL-SWG is trained for 4000 iterations (20 epochs). The experiment was run 5 times, in each run constructing new datasets and initializing all weights with random seeds. The SWD measured between generated samples and the test set is plotted against time in Figure 4. Each dot in the plot corresponds to 100 iterations.

For all choices of n the JL-SWG improves faster in time than the SWG. This effect gets stronger with increasing target dimension n. For $n = 50'000$

the JL-SWG converges approximately 5 times faster than the SWG. When using smaller values for L this effect is also smaller (not shown in the plots here). This is in line with our theoretical analysis of the JL-SWG. We also note that the iteration-wise improvement is smaller for the JL-SWG. We assume that this is a side-effect of the error that is introduced by the JL map. However, this behaviour is not bad. Instead of making big steps that take very long, the JL-SWG makes small but fast steps. This leads to showing the model all the training samples more often. At the same time, the JL-SWG approach is much stabler, with all runs being very similar, compared to much higher variance for the SWG approach.
The SWD of both models finally converges to similar values. For the SWG this value tends to be a bit smaller than for the JL-SWG. However, the SWG is trained to minimize exactly the SWD corresponding to the Wasserstein distance $\mathrm{W}_2(\mathbf{X}, \mathbf{Y})^2$ (this SWD is also measured for evaluation), while the JL-SWG minimizes the SWD corresponding to $\mathrm{W}_2(f(\mathbf{X}), f(\mathbf{Y}))^2$. Proposition 3.4 implies that this introduces an error of the factor $(1 \pm \epsilon)$, which explains why the SWG converges to slightly better values.
We also remark that the faster convergence of JL-SWG is not an effect of the different scaling of the loss function, due to omitting the scaling factor of the JL map A. The adaptive learning rate of the Adam optimizer compensates this very quickly. Experiments without omitting the scaling factor did confirm this.
We finally note, that we did not experiment with different generator architectures, or values for m. Maybe smaller values for m would lead to even faster convergence. However, on these toy datasets, we only wanted to test whether the JL-SWG really behaves as claimed in the theoretical analysis. This was demonstrated. For further comparisons we use other datasets, where also the generated samples can be compared.

3.3.2 JL-SWG on MNIST

The MNIST training set consists of 60'000 grey-scale images of handwritten digits. We divide it into a train set and a test set in the ratio 5:1. All images are of size 28×28 which corresponds to the dimension $n = 784$.

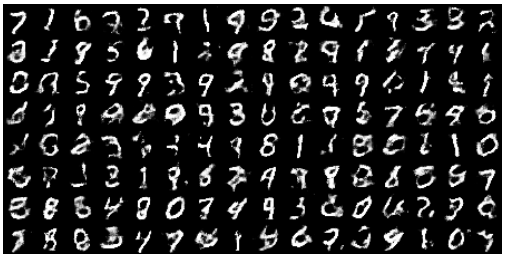
In the first experiment on MNIST we test different values for m, the dimensions after the JL map of the JL-SWG and compare them to the SWG. In particular we choose m in $\{n/4, n/2, 3n/4\}$, while fixing $L = 10'000$. At the same time we test different batch sizes, learning rates and training with or without batch normalization applied after all but the last layer. The best of the tested configurations was the same as in [22] given by batch size 250, learning rate $5 \cdot 10^{-4}$ and no batch normalization. The generator architecture

Layer	Operation	Act. Fn.	Output shape
Input	latent variable $Z \sim N(0, I_d)$	-	128
1a	fully connected	ReLU	4096
1b	reshape	-	$4 \times 4 \times 256$
2	convolution 2D transpose, 5×5 kernel, stride 2	ReLU	$8 \times 8 \times 128$
3	convolution 2D transpose, 5×5 kernel, stride 2	ReLU	$16 \times 16 \times 64$
4a	convolution 2D transpose, 5×5 kernel, stride 2	sigmoid	$32 \times 32 \times 1$
4b	cut off 2 pixels at borders	-	$28 \times 28 \times 1$

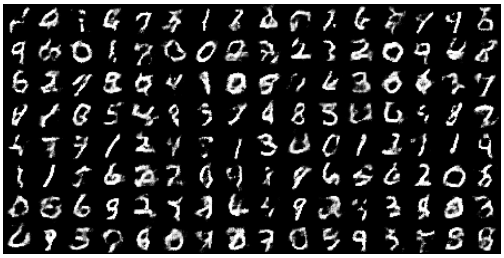
Table 2: generator architecture for MNIST, for layers 1b to 4b the shape is in the format *height* $\times$ *width* $\times$ *features*

is displayed in Table 2.
Randomly generated samples after training for 40'000 iterations (200 epochs) with these configurations are displayed in Figure 5. All 4 models produce samples of similar quality. Since the dimension n is not very big, the training of the SWG takes only 24% longer than the training of the JL-SWG with $m = n/4$.

This experiment showed that the JL-SWG can generate samples of similar quality as the SWG in less time. On the toy dataset, we have seen that this time advantage grows with the data dimension n. We want to test whether this is also the case on the more complex MNIST dataset, or whether this was just an effect of the easy distribution of the toy dataset. Therefore, we artificially enlarge the MNIST images from size 28×28 to 32×32, 64×64 and 128×128, by zooming with spline interpolation of order 1. The new datasets have dimensions 1024, 4096 respectively 16'384. For each dimension we train a SWG as well as two JL-SWG with m either 256 or 512. The generator architecture is given by the natural extension of the architecture in Table 2. One (for image size 64×64) respectively two (for image size 128×128) replicas of layer 3, all having 64 feature maps, are added to reach the wanted image size. Moreover layer 4b is dropped. As before we us the learning rate $5 \cdot 10^{-4}$, batch size $B = 250$ and number of projections $L = 10'000$. With this configuration, we train all models for 10'000 iterations (50 epochs) without batch normalization.
Plots of the SWD versus training time for 2000 and all iterations are shown in Figure 6. Moreover, randomly generated samples at the end of the training are displayed in Figure 7. As on the toy dataset, the difference between



(a) JL-SWG, $m = n/4$, 2.55s



(b) JL-SWG, $m = n/2$, 2.75s

(c) JL-SWG, $m = 3n/4$, 3.01s

(d) SWG, 3.17s

Figure 5: JL-SWG & SWG samples on MNIST with mean time per iteration

the convergence times of the SWD for JL-SWG and SWG grows with the dimension of the dataset. While for dimension 1024 the convergence behaviour for all 3 models is very similar over time, the JL-SWG models outperform the SWG in dimensions 4096 and 16'384. All generated samples are of similar quality. The ratio of average training times per iteration between SWG and JL-SWG with $m = 512$ are 1.16 ($n = 1024$), 1.81 ($n = 4096$) and 2.29 ($n = 16'385$).

3.3.3 JL-SWG on CIFAR-10 and CelebA

We also test the JL-SWG and SWG approach on more complex datasets. On the one hand we use CIFAR-10, which consists of 10 equally sized categories (airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck) of RGB images with size $32 \times 32 \times 3$. In total these are 50'000 train and 10'000 test samples of dimension 3072. On the other hand we use a subset of the RGB face image dataset CelebA, after preprocessing it. In particular, we randomly select 50'000 train and 10'000 test samples (out of the approximately 200'000 images), cut out the faces and rescale them to size $64 \times 64 \times 3$, which is equal to dimension 12'288.
For both datasets we use generator architectures that are the same as those used on the enlarged MNIST data, except that the last layer has 3 instead of one feature map.

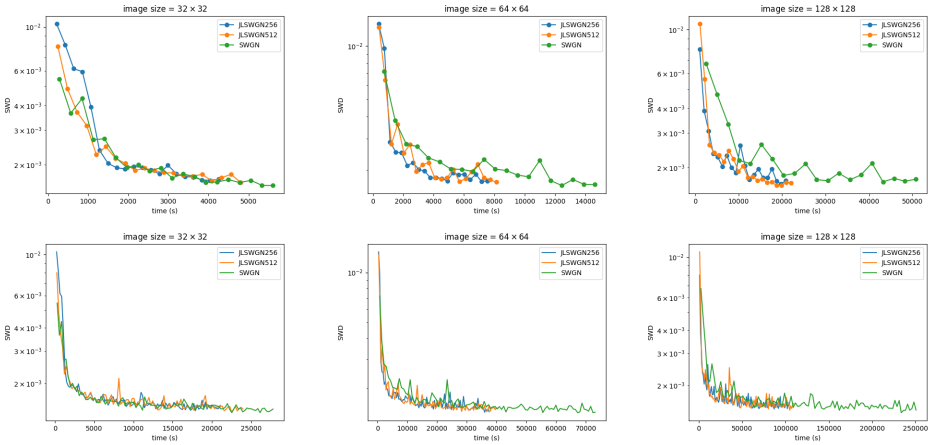


Figure 6: SWD for JL-SWG and SWG on enlarged MNIST, 1st row: 2k iterations (100 iterations per dot), 2nd row: 10k iterations

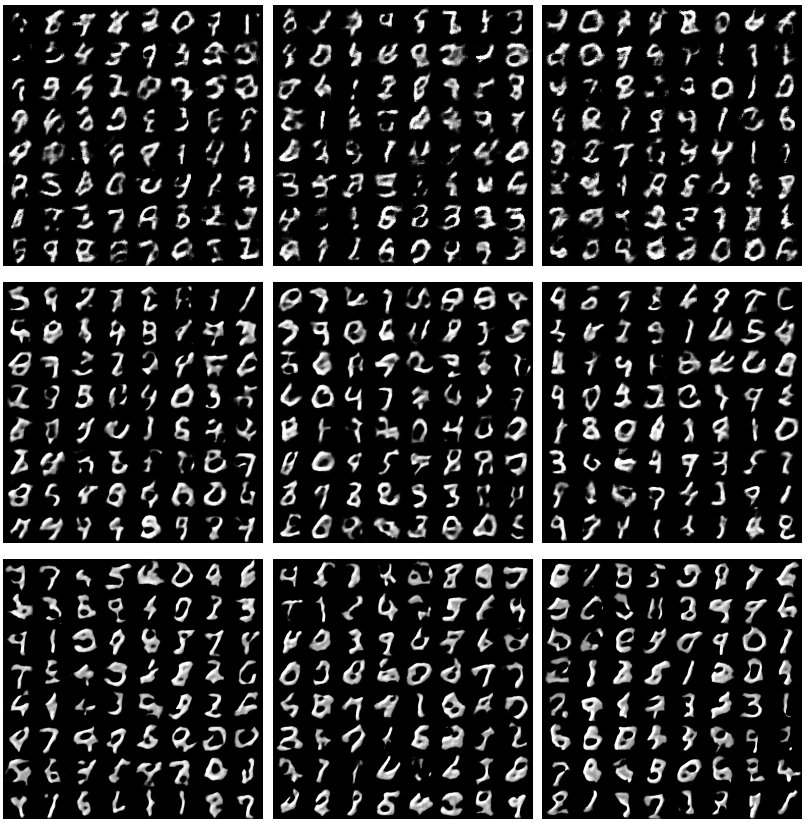


Figure 7: randomly generated samples on enlarged MNIST after 10k iterations, 1st row: 32×32, 2nd row: 64×64, 3rd row: 128×128, 1st column: JL-SWG $m = 256$, 2nd column: JL-SWG $m = 512$, 3rd column: SWG

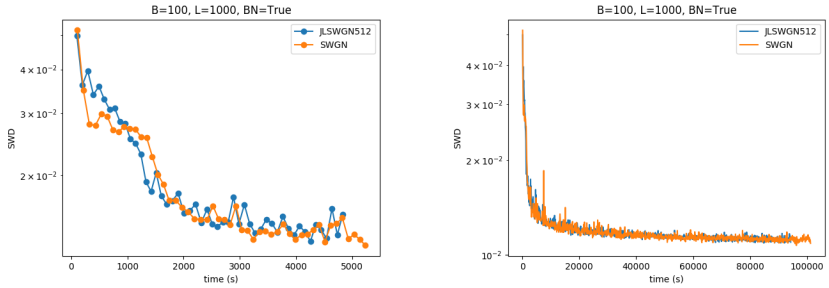


Figure 8: SWD for JL-SWG and SWG on CIFAR-10, left: 5k iterations (100 iterations per dot), right: 100k iterations

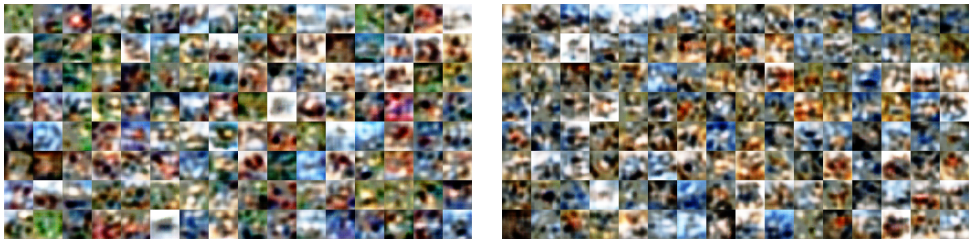


Figure 9: JL-SWG $m = 512$ (left) & SWG (right) samples on CIFAR-10 after 100k iterations

First we train a JL-SWG with $m = 512$ as well as a SWG on CIFAR-10 using batch size $B = 100$ and learning rate $5 \cdot 10^{-4}$. We use a smaller number of projections, $L = 1000$, to test whether the performance of the JL-SWG is also comparable to the one of SWG, if m is near its upper bound $\frac{nL}{n+L}$ due to the small value of L (until now this was only the case due to small values of n). We train the models once with and once without batch normalization and see, that only the models with batch normalization are able to generate meaningful samples. The SWD of the models is plotted against time in Figure 8 and randomly generated samples after 100'000 iterations (200 epochs) are shown in Figure 9. For the small value of L the SWD for both models behaves nearly identically. Also after training for many iterations we do not see an effect as on the toy dataset, where the SWD of the SWG tended to converges to smaller values. Both models produce very blurry images, but of similar quality. With this configuration the training of the JL-SWG is a bit faster. We conclude that the JL-SWG approach can also be used if its configuration is chosen such that its training time is similar to the one of the corresponding SWG. Although the JL-SWG approach does not outperform the SWG in that case, there is also no risk of worse performance.

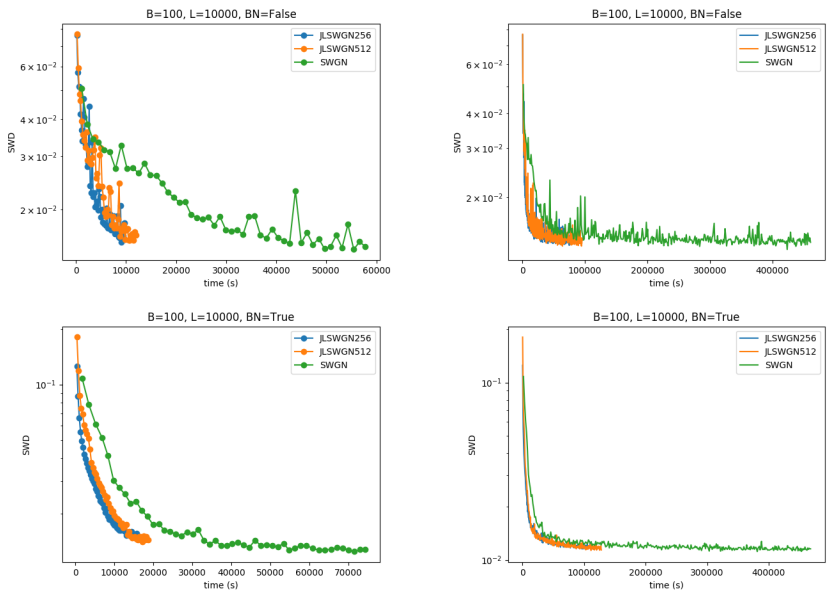


Figure 10: SWD for JL-SWG and SWG on CelebA, left: 5k iterations (100 iterations per dot), right: 41k (top) and 35k (bottom) iterations

In the next experiment we train JL-SWG with $m \in [256, 512]$ and SWG on CelebA. We us batch size $B = 100$, learning rate 10^{-4} and $L \in \{1000, 10'000\}$. All models are trained with and without batch normalization. As on CIFAR-10, the JL-SWG and SWG models that are trained with $L = 1000$ are very similarly. Their SWD converges with similar speed and also their generated samples are of the same quality. For the models trained with $L = 10'000$ the SWD is plotted in Figure 10. As on the other datasets before, the SWD improves faster when using the JL-SWG models. Without batch normalization this effect is stronger. Comparing the average time per iteration of SWG and JL-SWG with $m = 512$, we have a ratio of 4.83 without batch normalization, respectively 3.65 with batch normalization. This is rather big. The training with batch normalization takes longer per iteration, due to the more expensive computation of the layer outputs. This extra time is equally needed in JL-SWG and SWG, leading to the smaller ratio.
Furthermore, we compare the average time per iteration of the models for the different values of L. For JL-SWG with $m = 512$, switching from $L = 1000$ to 10'000, increases the training time only by a factor 1.27 without batch normalization and 1.08 when using batch normalization. On the contrary, for SWG the factors are 4.80 respectively 3.76. This empirically demonstrates the claim we have made in Section 3.2, that the JL-SWG can heavily reduce the increase of training time with growing L.

Figure 11: JL-SWG $m = 256$ (left) & SWG (right) samples on Celeb after training for 50 hours

In Figure 11 we only show samples generated with batch normalization, since they have better quality. However, we remark that it is not the case that training without batch normalization does not work as on CIFAR-10. Furthermore, we only show samples of JL-SWG with $m = 256$, due to the similar quality when $m = 512$. The generated samples of SWG and JL-SWG after the same number of iterations are of similar quality. Therefore, we show samples generated after the same training time (approximately 50 hours). We see that the quality of the JL-SWG samples is superior. As on CIFAR-10 the generated samples are blurry, but on CelebA most of them can be identified as faces, while on CIFAR-10 it is not possible to identify the objects in the generated images.

3.3.4 JL-SWGAN on MNIST

The generated samples of JL-SWG and SWG without discriminator are not of a quality that can be compared to other GAN models. Already on the relatively easy MNIST dataset, the results of the standard DCGAN or WGAN-GP are visibly sharper.
Therefore, we test the JL-SWGAN and compare it to its SWG counterpart, starting on the MNIST dataset with image size 32×32. For JL-SWGAN we test $m \in \{64, 128, 256\}$. All models are trained with $L = 10'000$, batch size $B = 250$ and learning rate 10^{-3}. For comparison we also train JL-SWGAN with $m = 256$ and SWG with $l = 1000$. The models are trained for 20'000 iterations (200 epochs). The generator and discriminator architecture are given in Table 3 and 4 in Appendix A, where c_g and c_d are set to 256 and

layer 4* of the generator is omitted.

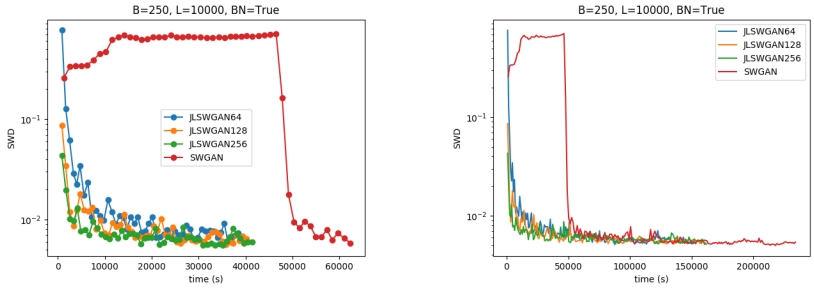


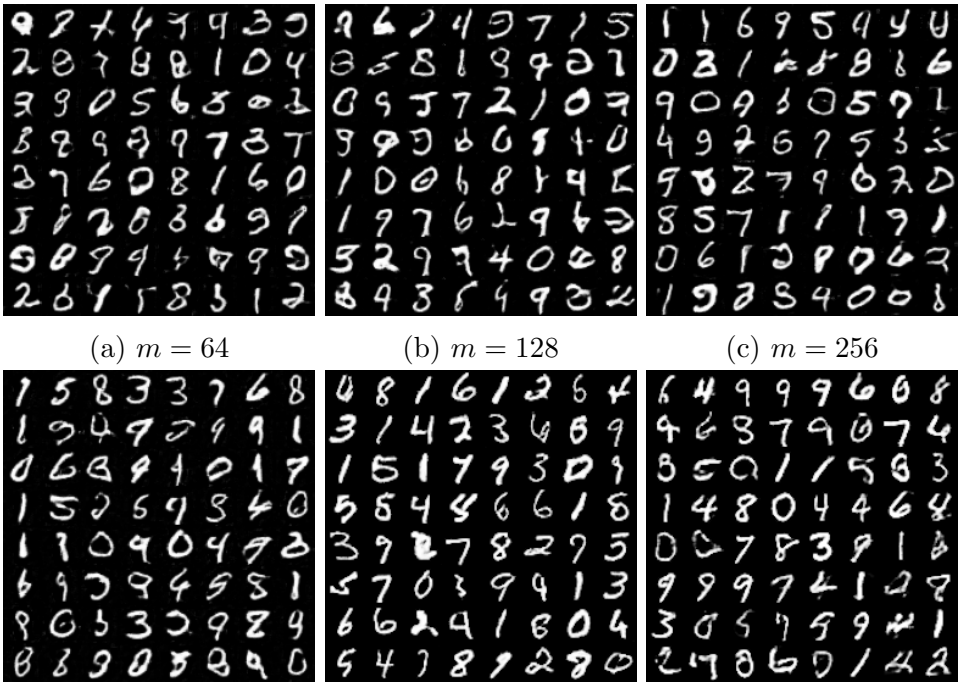
Figure 12: SWD for JL-SWGAN and SWG on MNIST, left: 5k iterations (100 iterations per dot), right: 20k iterations

The SWD is plotted against time in Figure 12. While the SWD of all JL-SWGAN models directly starts to converge, the SWD of the SWG does not improve for the first 3700 iterations. This is an unexpected behaviour and might not show up when rerunning the experiment. However, we see that the JL-SWGAN approach is more stable. The JL-SWGAN with $m = 256$ has the best SWD evolution. One iteration of SWG needs 44.71% more time to be trained than for JL-SWGAN with $m = 256$. Comparing the training times of JL-SWGAN ($m = 256$) with $L = 1000$ and 10'000 we have a ratio of 1.08, while the ratio for SWG is 1.54.
In Figure 13 we display generated samples of all models trained with $L =$ 10'000 as well as samples generated by the standard DCGAN and WGAN-GP models (they were both trained with batch size 50). All generated samples seem to be of similar quality.

3.3.5 JL-SWGAN on CIFAR-10

For JL-SWGAN and SWGAN on CIFAR-10 we use the generator and discriminator architecture given in Appendix A in Table 3 and 4 with c_g and c_d both set to 512. We test the generator with and without the extra layer 4* and the discriminator with batch normalization and layer normalization. For the generator we always use batch normalization. As before we train the models with $L \in \{1000, 10'000\}$. For JL-SWGAN we use $m = 512$. For all models the batch size is $B = 100$ and the learning rate is 10^{-3}.

We show the plots of the SWD against time for the first 33'000 iterations in Figure 14. In all settings with $L = 1000$ the SWD of JL-SWGAN and SWG evolves very similarly. For $L = 10'000$ the JL-SWGAN models tend to converge a bit faster. Smaller or bigger values of m might lead to a better performance (confer the different performances shown in Figure 12).



(a) $m = 64$ (b) $m = 128$ (c) $m = 256$

(d) SWG (e) DCGAN (f) WGAN-GP

Figure 13: randomly generated samples after 20k iterations, 1st row: JL-SWGAN, 2nd row: other models

However, due to the big computational cost, we do not test them here. For $L = 10'000$, one iteration of the SWG models need about 62.39% more time than the iteration of JL-SWGAN. Furthermore, the ratio between the iteration-wise training time of the JL-SWGAN models with $L = 1000$ and 10'000 is approximately 1.07, while the ratio for the SWG models is 1.68.
In Figure 15 we display samples generated by JL-SWGAN and SWG trained with extra layer 4^*, $L = 1000$ and batch normalization for the discriminator. The samples were produced after training for 58'000 iterations.

At this point we remark that the SWD (as we compute it here) alone, can not be used as a measure of the quality of a generative model. Comparing the samples generated by JL-SWG (Figure 9) with those of JL-SWGAN here, it is obvious that JL-SWGAN has much better quality. However, the SWD of JL-SWGAN is slightly bigger (note that the same batch size was used, hence the SWDs can be compared). This might not be surprising, since JL-SWG is trained to minimize directly this distance (up to the Johnson-Lindenstrauss map that is applied in between), while JL-SWGAN tries to minimize the SWD computed after applying a discriminator. Nevertheless, this shows the limitations of the SWD as evaluation metric. For a fixed model the distance

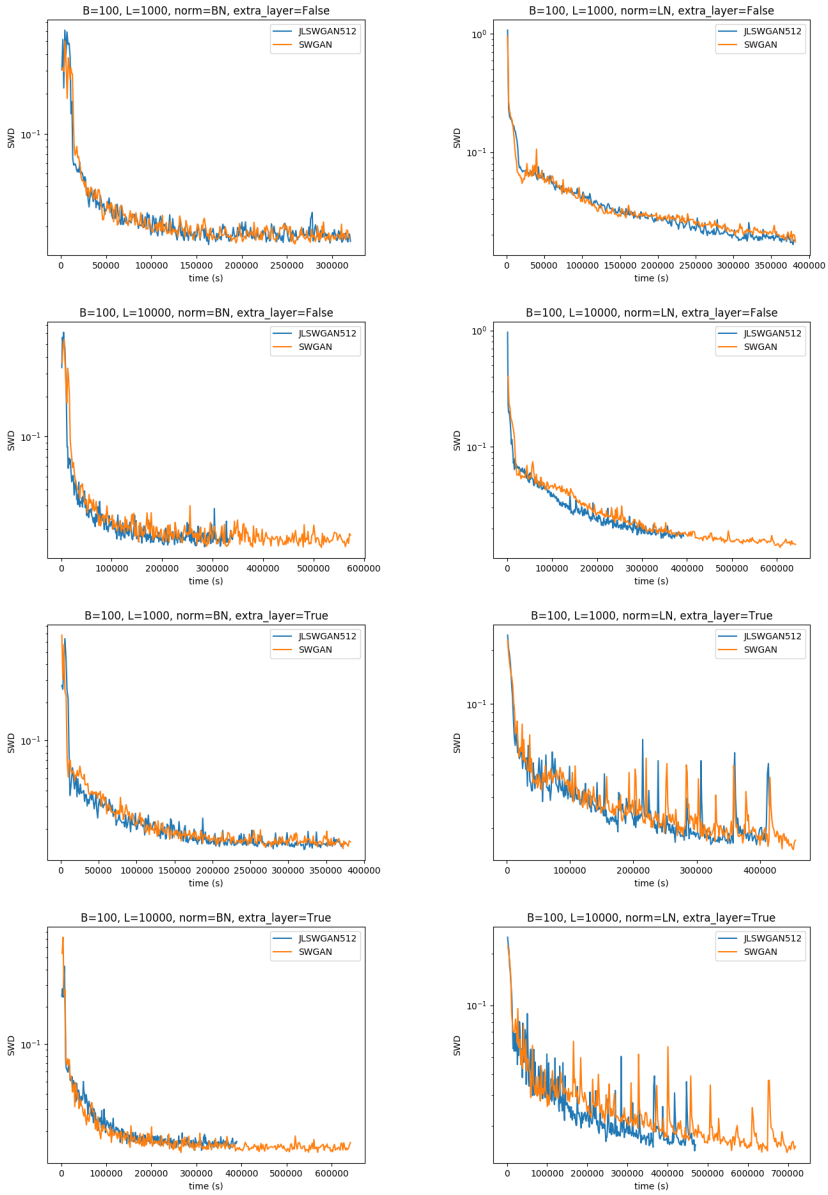


Figure 14: SWD for JL-SWGAN $m = 512$ and SWG on CIFAR-10

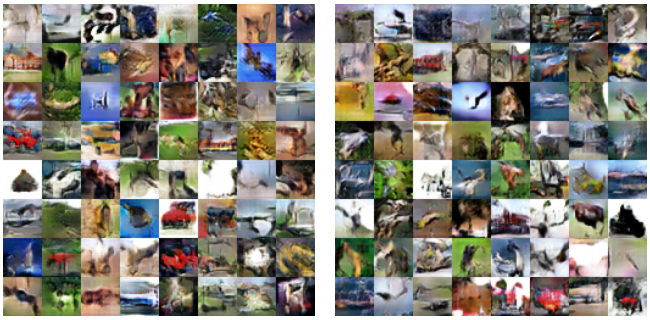


Figure 15: randomly generated samples on CIFAR-10 after 58k iterations, left: JL-SWGAN $m = 512$, right: SWG

correlates quite well with the sample quality, especially in the beginning of the training. However, one has to be careful when using the SWD to compare different models. The way we have used it to compare JL-SWG and SWG models in the previous sections is justified, since there the SWG is trained to minimize this distance. Hence, a JL-SWG model that performs better than the respective SWG with respect to the SWD has an advantage over it. It is not clear, whether the SWD is still as expressive when comparing JL-SWGAN and SWG with discriminator.

Therefore, we also compute the Inception Scores (IS, confer Section 2.5) of the models with $L = 1000$, where the SWDs evolve almost identically. Preliminary computations of the IS showed that models with extra layer 4^* perform better and that there is no significant difference between using batch normalization or layer normalization for the discriminator. Therefore, we only train models with the extra layer 4^* in this experiment. Since models with layer normalization need more time for the training, we only use batch normalization here. However, we also test higher batch sizes, $B \in \{100, 250, 500\}$, and other values for the dimension after the JL map, $m \in \{256, 512, 1024\}$. The Inception Scores are computed during the entire training with 10'000 randomly generated samples. We use the standard setting to split up this set into 10 subsets to compute the standard deviation. The IS is evaluated after each 1000 iterations (for some models it was 2500 iterations for the first half of their training). The IS are plotted against training time in Figure 16.

We do not observe a significant difference between JL-SWGAN and SWG. However, we see that the JL-SWGAN models do not perform worse than SWG. Similar experiments with $L = 10'000$ also did not show a significant difference between JL-SWGAN and SWG. Without using a discriminator we did observe better performance of JL-SWG compared to SWG for a large

number of projections. Using a discriminator, this is not the case any more. We explain this by the much smaller time improvement in relation to the whole training time, when using a JL map. This is due to the more expensive other parts of the training, in particular the additional costs of the discriminator.
We also remark that the models with batch size 500 were not trained long enough to reach good results. Since the purpose of this experiment was to compare JL-SWGAN with SWG and not to achieve the highest possible IS, we did not continue their training.
However, we report the best Inception Scores observed during our experiment. The overall best measured IS was 5.95 ± 0.11 and was achieved by JL-SWGAN with $B = 250$ and $m = 512$. For SWG the best measured score was 5.87 ± 0.13, observed for the same batch size and during training both models for the same time (longer than shown in the plot). Although training JL-SWGAN with $B = 250$ and $m = 1024$ only for 2/3 of this time (the previously mentioned scores were both achieved in the last third of training), it also achieved an IS of 5.88 ± 0.14. With $m = 256$, the model did not reach an IS higher than 5.55 ± 0.20.

The Inception Scores of the JL-SWGAN model is a bit worse than the one of DCGAN, but better than the ones of WGAN and Sinkhorn GAN (confer Table 1). However, there are several possibilities that might further improve the IS of JL-SWGAN. Using very large batch sizes leads to an enourmous improvement for OT-GAN [67]. In our experiment we have seen that already the increase from 100 to 250 leads to a small improvement. Therefore, we assume that increasing the batch size further (in OT-GAN batch sizes up to 8000 are used) could yield significantly higher Inception Scores. Furthermore, a bigger and more complex generator and discriminator architecture like the ResNet design, producing very good results for WGAN-GP [32], could be used. Due to the better performance of JL-SWGAN when using the extra layer in the generator, this is very likely to improve the IS. Another option would be to create more structure on the latent space as it is done for example in MGAN [38]. Or one could progressively grow the generator and discriminator as in [46]. Although we believe that these methods could improve the IS results drastically, we do not test them here due to the high computational costs. We leave this for future work.

3.3.6 JL-SWGAN on CelebA

Here we use the same version of CelebA as in Section 3.3.3 before. We train a JL-SWGAN with $m = 512$ and a SWG with the architecture given in Appendix A for image size 64×64. The maximal number of features are set to

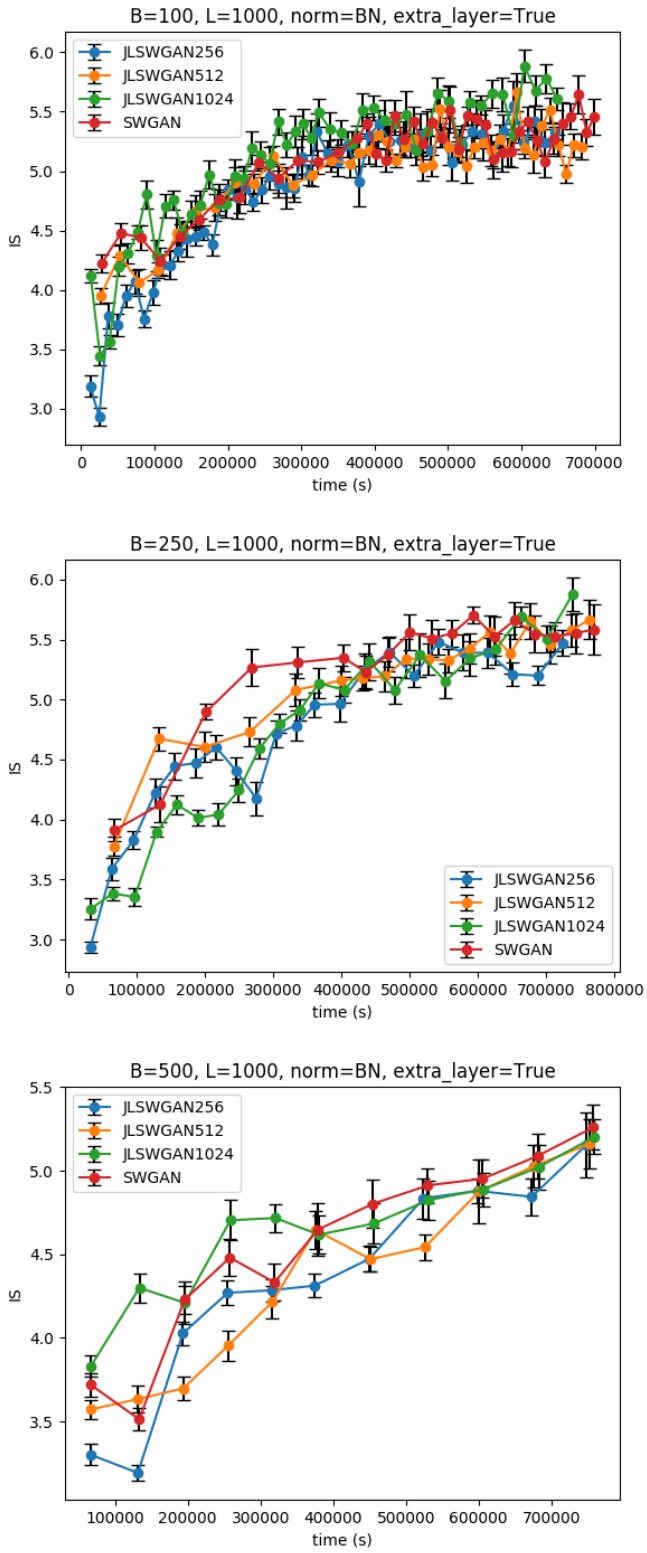


Figure 16: Inception Scores for JL-SWGAN and SWG on CIFAR-10 computed every 1k (or 2.5k) iterations with 10k samples. Error bars are the standard deviation that are computed for the IS.

$c_g = 1024$ and $c_d = 512$. We test configurations with and without the extra layer 4^* for the generator. Furthermore, we test batch normalization and layer normalization for the discriminator. The batch size is set to $B = 100$ and the learning rate to 10^{-4}. Due to the slow training process we only use $L = 1000$ projections.

As we have discussed in the previous section, it is not clear how expressive the SWD is for these models. Nevertheless, we show the plots of it for all models in Figure 17. This is the first time that we encounter slower convergence of the SWD for our model compared to SWG. Again we emphasize that other values for m might lead to better performance.
The models using batch normalization in the discriminator produce better samples, which are shown in Figure 18. Although the JL-SWGAN models performed a little worse with respect to the SWD, the generated samples are of similar quality. Moreover, it seems like the diversity is a bit higher for the JL-SWGAN models.

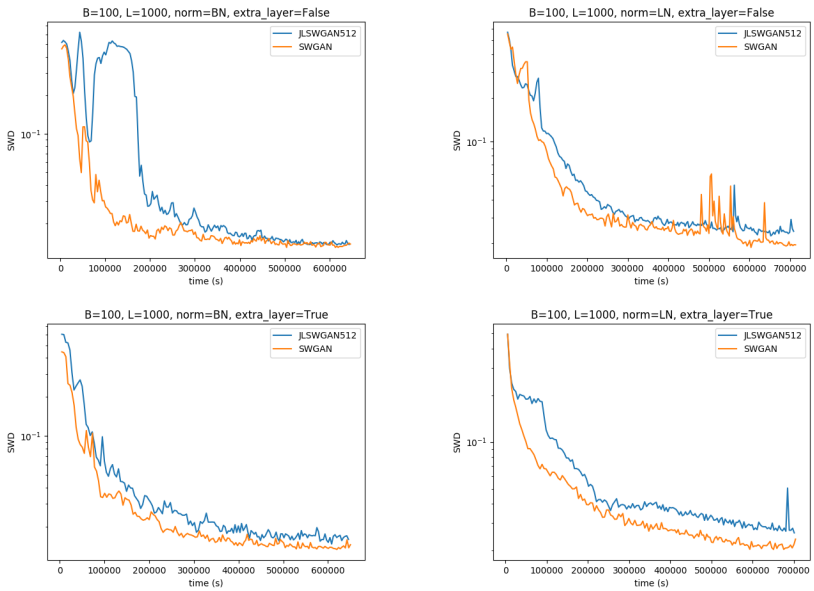


Figure 17: SWD for JL-SWGAN and SWG on CelebA, 1st row: 20k iterations, 2nd row: 16k iterations

Figure 18: randomly generated samples on CelebA, left: JL-SWGAN $m = 512$, right: SWG, 1st row: without extra layer after 22'500 iterations, 2nd row: with extra layer after 17'700 iterations

3.4 Applying the Johnson-Lindenstrauss Lemma somewhere else

In this section we briefly discuss whether the Johnson-Lindenstrauss Lemma could be applied in the SWG (respectively GANs in general) training at a different point.

One possibility is to apply a Johnson-Lindenstrauss transformation f directly to the data, to reduce its dimensionality. A GAN would then be trained to learn the distribution of the transformed data. If one can find a map g approximating the inverse f^{-1}, the generated samples could be mapped back to the target space. The maps f and g can be interpreted as encoder and decoder of a simple autoencoder. The critical point in this approach is to find the inverse map g. Given a Johnson-Lindenstrauss transformation $f : \mathbb{R}^n \to \mathbb{R}^m, x \mapsto \frac{1}{\sqrt{n}}Ax$, where $A \in \mathbb{R}^{m\times n}$ has i.i.d. entries of $N(0,1)$, a typical way to choose the approximate inverse is $g : \mathbb{R}^m \to \mathbb{R}^n, x \mapsto \frac{1}{\sqrt{n}}A^T x$. The reasoning behind is, that the n columns of $\frac{1}{\sqrt{n}}A$ can approximately be thought of orthogonal unit vectors. Of course, for $m < n$ they can not all be orthogonal. However, assuming $m = n$ and the vectors being orthogonal unit vectors, this would yield $\frac{1}{n}A^T A = I_n$, the identity matrix. Hence, $g \circ f = id$ would hold.
The minimal requirement for this approach to work is, that the reconstruction of data samples works well. We test this with the MNIST dataset (enlarged to size 32×32) as well as a version of the CelebA dataset, where the images are converted to grey-scale images of size 32×32. Hence, both datasets have dimension $n = 1024$. For the Johnson-Lindenstrauss map, we use the target dimension $m = n/2$ and the extreme case of $m = n$ and show a subset of images as well as their reconstruction with $g \circ f$ in Figure 19. For MNIST, generated samples of JL-SWGAN (using much smaller values for m) are of better quality than the reconstructions of the true data samples. For CelebA, even the generated samples of JL-SWG (also with smaller m) are of better quality. Using different JL maps does not improve the quality. Therefore we conclude that this approach can not be used as an alternative to JL-SWG and JL-SWGAN.

Another possible way to incorporate a Johnson-Lindenstrauss transformation that could be applied to every GAN approach including a discriminator is the following. Instead of applying the JL map after the discriminator as it is done in the JL-SWGAN, it could be applied directly to the generator output, respectively the training samples. The output of the JL map could then be passed to the discriminator. Like this, the JL map can be interpreted as a fully connected layer of the discriminator with random weights (changing in

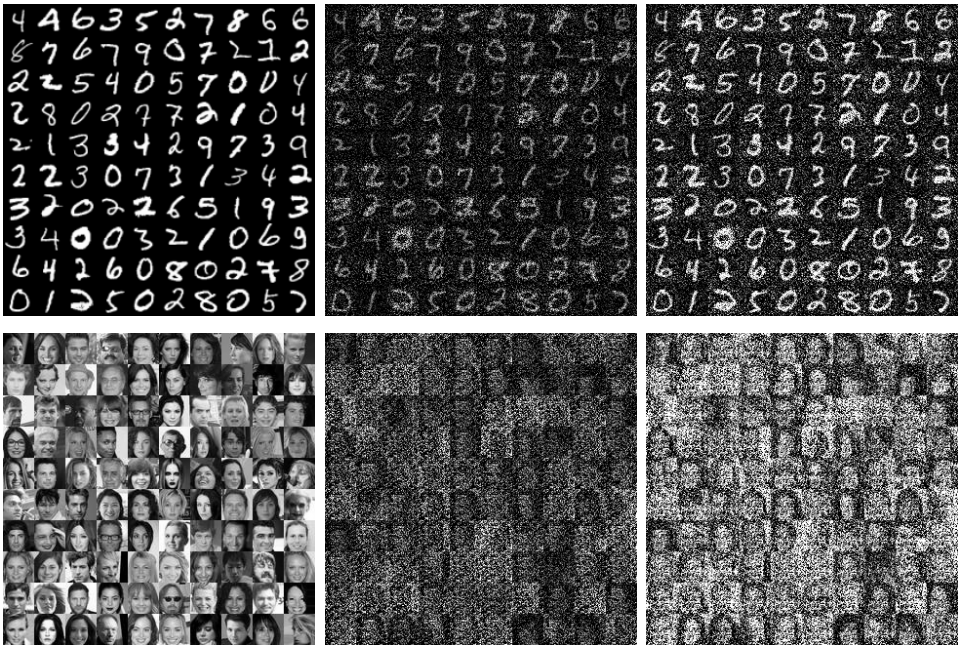


Figure 19: data samples (left) and JL reconstruction with $m = n/2$ (middle) and $m = n$ (right), 1st row: MNIST, 2nd row: CelebA

each iteration or being fixed after choosing them once) and without activation function. Fully connected layers in GANs usually lead to worse performance, as was outlined in [63]. Therefore we claim that also this approach would not lead to the desired result of reducing the dimension to speed up computations, while keeping the same quality of the model. However, we did not explicitly test this approach.

We conclude that the point at which the Johnson-Lindenstrauss Lemma is applied in the JL-SWG and JL-SWGAN approach is the only one where it can effectively reduce the dimension and therefore the training time without degrading the quality of the generator.

3.5 Conclusion

In this chapter we have introduced an extension of the sliced Wasserstein generator (SWG), which we called the Johnson-Lindenstrauss sliced Wasserstein generator (JL-SWG) respectively the Johnson-Lindenstrauss sliced Wasserstein GAN (JL-SWGAN). We have shown in theory, why this extension should yield similar training results as SWG, while it can lead to a faster training procedure due to a smaller number of involved computations. Extensive tests of our models and their SWG counterparts have shown that

JL-SWG, that means when no discriminator is used, converges faster and therefore needs less time to be trained then SWG. However, when using a discriminator the time improvement of JL-SWGAN over SWG is very small and no significant advantage of our method was observed. Still, our method did not show worse results in our tests. We finished the chapter with a discussion about other possibilities to apply the Johnson-Lindenstrauss Lemma in the context of GANs, but could not identify promising ideas.

4 GANs with fixed weights

In this chapter we study generative adversarial networks with random weights and similar approaches in which some weights of a GAN are fixed during training. In Section 4.1, we give a short introduction to neural networks with random weights. Then we discuss why it is interesting to consider GANs with random weights in Section 4.2 and present our experimental results in Section 4.3. When considering GANs with fixed layers, it is also of interest what the layers of well trained GANs with the same architecture look like. Therefore, we discuss the distribution and some statistics of the weights of a trained standard DCGAN in Appendix D. The discussed GAN has the same architecture as the GANs used in our experiments.

4.1 Overview of NN with random weights

There is a wide range of literature on *neural networks with random weights* (NNRW). A good and recent overview paper on this topic is the one by Cao et al. [18]. There, the authors define NNRW as models with neural network architecture, for which all parameters of the hidden layers are fixed after randomly choosing them from a certain distribution. The training of these models, which boils down to the training of the output layer, is done analytically and usually non-iteratively. This means that an objective function is used, for which an analytical solution exists, for example by making use of a pseudo-inverse matrix. Hence, the training is much faster than the usual training of neural networks, which depends on (small) stepwise improvements using some variant of stochastic gradient descent (SGD). This speedup usually comes at the price of worse performance compared to training all weights. However, if the quality is not reduced too much, NNRW can be a good alternative to standard neural networks.
Some of the methods listed in [18] that belong to NNRW are *random vector functional link networks* (RVFL) [61], *extreme learning machines* (ELM) (which are, in their standard form, a special version of *single-hidden-layer feedforward networks* (SLFN) with random weights) [40] and *deep neural networks with random weights* (DNNRW). The last category includes convolutional neural networks (CNN), auto-encoders (AE) and restricted Boltzmann machines (RBM), when they have random weights.
It was shown in [39] that SLFNs with random weights are universal approximators when using a specific incremental constructive method. This result is interpreted as universal approximation theorem for ELMs.

Saxe et al. [68] dig deeper into why one-layer convolutional pooling architectures with random weights can perform so well in classification tasks and

conclude that the architecture itself is much more important than the weights. They also show that the results of architectures with random weights correlate well with the results of the same architectures after training. In their experiments, training always improved the results, but some architectures with random weights performed better than others after training.

Giryes et al. [28] also study CNNs with random weights used for classification. They make the following discovery what training of the hidden layers does. For each data point (of the CIFAR-10 dataset) the euclidean distance to the farthest point in the same class (intra-class) and to the nearest point of a different class (inter-class) are computed before and after applying the hidden layers. Computing the ratios between these distances, they find that applying the trained hidden layers shrinks the intra-class distances and increases the inter-class distances. This is not the case for hidden layers with random weights.

To the best of our knowledge, GANs with random weights have not been studied so far.

4.2 Why considering GANs with random weights

The main reason to consider NNRW is the reduced training time due to the analytical solution of the optimization problem. We first discuss why this is *not* the motivation to consider GANs with random weights. Most GAN versions depend on the adversarial training procedure of the generator and the discriminator. Their successive updates make it possible that the generator learns to produce samples of high quality. Unless omitting the adversarial training, this implies that GANs can not be trained in a non-iterative manner, since updating one player changes also the objective function for the other player.

In Section 2.3 we have seen some GAN variants that do not necessarily need a discriminator. These are Sinkhorn GAN [26], OT-GAN [67] and SWG [22]. The objective functions of the first two depend both on the Sinkhorn Algorithm [71] (which iteratively computes the loss), while the one of SWG depends on sorting of the outputted values. Therefore, there is no straight forward analytical approach to determine the weights of the output layer of the generator (when all other weights are fixed) such that the loss is minimized. Once again, an iterative procedure would be needed for optimizing the weights, since the loss function itself should not be changed (due to its careful choice to approximate some kind of optimal transport distance).
Hence, the training time could not be reduced as drastically as in other NNRW approaches. Furthermore, the quality of generators trained without discriminator is always worse than the ones trained with discriminative

opponent [26, 67, 22].

Therefore, we focus on GAN methods using a discriminator. Let us discuss the maximal time reduction that is possible when fixing some of the weights to randomly chosen values. Since an iterative training procedure can not be prevented, it is natural to use some version of SGD also when fixing some of the weights. SGD has a forward run through the neural network to compute the loss function and then performs a backward run to compute the gradients and to update the weights. Updating only the last layer of the gradient (as it is done in NNRW) the forward run has to be executed entirely but the backward run can be stopped after the first step. Although the backward run is computationally more expensive, its computational cost can be bounded by the one of the forward run times a small constant. This implies that also the relative time improvement can only be by a small factor. An experiment on the MNIST dataset (confer Appendix C for details) yielded a maximal speedup by the factor 2.2, which is equivalent to a training time that is 0.45 times the training time when all weights are updated. We conclude that this small factor is not an incentive to use random weights, which, at the same time, yield worse training results.

Our motivation to consider GANs with random weights is of theoretical and not of practical nature. It is still not well understood why neural networks can produce such incredibly good results in many applications. Typical state-of-the-art neural networks have millions of weights (already the relatively small generator and discriminator architecture given in Appendix B have 1.5 respectively 1 million trainable parameters) which are tried to be optimized at once using some variant of SGD. The loss functions that are used to train neural networks are usually non-convex (as functions of the trainable weights) and often have infinitely many local minima [83]. Since stochastic gradient descent is a local optimization scheme, one has to expect that the weights of a neural network converge to one of the local minima during training (if they converge at all).
It was shown by Kawaguchi [47] that for linear neural networks (linear means that all activation functions are linear which is equivalent to no activation functions) trained to optimize the mean squared error (MSE), in supervised learning tasks, all local minima are also global minima. However, this is not the case for networks with non-linear activation functions. Although many attempts have been made to provide conditions under which local minima of non-linear neural networks are in fact global minima (see e.g. [72]), Yun et al. [83] showed that this search is rather hopeless. In particular, they proved that there exist infinitely many non-optimal local minima of neural networks with ReLU-like activation functions (ReLU, Leaky ReLU and

more) when applied to datasets for which linear models can not perfectly fit the data. This result is established for supervised learning with MSE as objective function.

With this in mind, the undeniable achievements of neural networks in many machine learning problems suggest that also non-optimal local minima can yield high quality results. But more seems to be true, as suggested by neural networks with random weights, which also achieved good results, for example, in classification tasks [68]. If chosen accurately, the structure of the architecture on its own can already solve big parts of a classification problem without training most of the involved weights. This also gives some insights into why the training of millions of weights can work. Instead of starting from scratch and achieving all expressive power only by training of the weights, one already starts off at a good point when choosing a decent architecture with random weights.

It is well known that the choice of the architecture is crucial for the success of training a GAN [63, 9, 32, 22]. This is especially true if the standard GAN objective is used. But also for GAN variants that make the training more stable (WGAN, WGAN-GP, SWG), the quality of the trained GAN highly depends on the architecture [9, 32, 22]. In Section 4.3.1 we empirically show that the architecture alone can solve a great part of the problem to learn a complex distribution and generate new samples of it. In particular, when training less than 0.41% of the weights of a standard GAN on MNIST [53], the network already learns more, for example, than when training all weights but without using batch normalization. Increasing the number of trainable weights to 8.4%, the GAN seems to learn the entire distribution and only suffers of generating pictures that are not very sharp and therefore visually not so appealing.

Due to the visual degradation of the samples when using completely random weights, we conduct further experiments where the weights are not chosen totally at random but still fixed during training.

4.3 Experimental Results

In the following experiments[2] we always use the architecture given in Appendix B, with $c_g = c_d = 256$ and a 128-dimensional standard normal distribution on the latent space. For training we use the standard GAN objective as described in Section 2.1. If not mentioned otherwise, batch normalization is applied to generator and discriminator. The batch size B is set to 50. All models are trained with the Adam optimizer, a variant of SGD, with $\beta_1 = 0.5$

[2]Code available at `https://github.com/FlorianKrach/A-Study-of-GANs.git`.

and default values for β_2 and ϵ. The learning rate is chosen as $2 \cdot 10^{-4}$ if not specified differently.

All experiments are conducted on the training set of the MNIST dataset [53] which consists of 60'000 images of the digits 0 to 9. We split this set into a training set of 50'000 and a test set of 10'000 images.

We always specify the number of iterations for which the GANs are trained. In each iteration, both, the generator and the discriminator, are updated once. Since only the discriminator updates need samples of the training set, $B = 50$ samples are shown to the network in each iteration. The samples are not drawn entirely at random, but instead the training set is shuffled at the beginning of each epoch and the samples are used in this order. This is common usage in neural network training. Hence, after 1000 iterations all samples have been shown to the network once, completing one training epoch.

In the experiments in Section 4.3.1 weights of some layers of the generator and discriminator are fixed after random initialization. Therefore, the initialization method can have a big influence on the outcome of the training. Usually the standard choice is the "He" initialization method [35], which was constructed for deep neural networks with ReLU-like activation functions (ReLU, Leaky ReLU and more). Its derivation is based on the "Xavier" initialization method [29]. The fundamental idea of both methods is to choose the random weights, such that (after initialization) the variance after each layer is approximately the same and the variance of the gradient of each layer is approximately the same. In particular, this should prevent vanishing or exploding gradients. In line with their motivation, these methods are also called *variance scaling initializers*. Beside the "He" method we also tested the "Xavier" initialization method in two forms. Once with weights drawn from a uniform distribution (this is how it was proposed in the paper [29]) and once where they are drawn from a normal distribution that has the same standard deviation as the uniform distribution. All of these initialization methods led to very similar results in our experiments. Therefore, we do not further distinguish between them. On the other hand, using weights drawn, for example, from a standard normal distribution leads to much worse results in our tests. Therefore, the discussed models in the following experiments are always initialized with one of the variance scaling methods.

All experiments are run at least twice with similar results, unless stated otherwise.

4.3.1 GANs with random weights

In the first experiment some layers of the generator and discriminator are fixed after initialization. We display randomly generated images of different configurations after training for 20'000 iterations in Figure 20.

Our first choice is to fix all but the last layer of both networks as it is usually done in NNRW. Like this the GANs in our experiments are not able to generate any meaningful images (Figure 20a). However, as soon as the bias terms (of the batch normalizations) of all layers are trainable as well, we see that the generated images are meaningful (Figure 20b). Although they are not of high quality, close inspection reveals a digit for many of the samples. Furthermore, the generator does not suffer from mode collapse, with the samples having high diversity. For every class (digit) a sample can be found that seems to belong to it. In this configuration, only 5889 weights of the generator and 4545 of the discriminator are trainable. This amounts to 0.4% of all weights of the GAN.

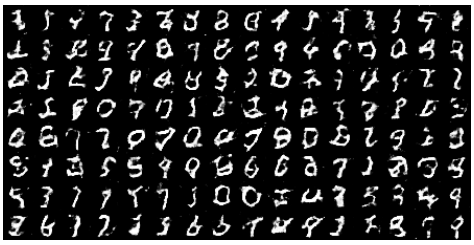
Since the quality of the samples is very low, we also try out configurations where more layers are trainable. We use the following heuristic to choose the trainable layers, whereby it should be clear that fixing the last layer can not lead to good results. The discriminator gets information (the true or generated image samples) as input. Hence, its first layer is very important, since here parts of the information can easily get lost. Therefore, we choose to train the first layer of the discriminator as well. For the generator, the situation is different. It does not get information, but only noise as input. Therefore, no information can get lost and the first layer is not so important. Instead, we can make the second last (3rd) layer trainable, which should give it the power to produce samples of better quality. In contrast to the first experiment, here the GAN training also works when the biases of the remaining layers are not trainable (Figure 20c). The sample quality is also better. However, making the biases trainable further improves the quality (Figure 20d). Now one easily recognises a digit in most of the samples. Furthermore, samples of all digits are generated. Although the sample quality is still worse than the one of the standard GAN (Figure 20i), we conclude that in this configuration the generator successfully learns big parts of the target distribution. We note that in this setting, 210'689 weights of the generator and only 6145 weights of the discriminator are trainable. In total this is equivalent to 8.4% of all weights.

We also test the choices of the second trainable weight to be similar in generator and discriminator, either being for both the 1st (Figure 20e) or the 3rd layer (Figure 20f). In both cases this leads to an increased number of trainable weights (20.8% respectively 40.1%) without big improvements in

(a) TW: last layer

(b) TW: last layer & all biases

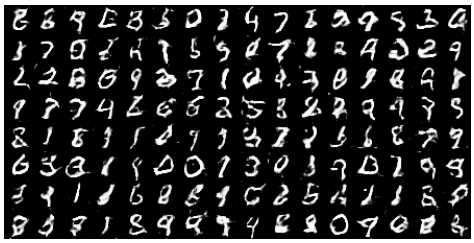
(c) TW: layer 3, 4 of gen., layer 1, 4 of disc.



(d) TW: layer 3, 4 of gen., layer 1, 4 of disc. & all biases

(e) TW: layer 1, 4 of gen. and disc. & all biases

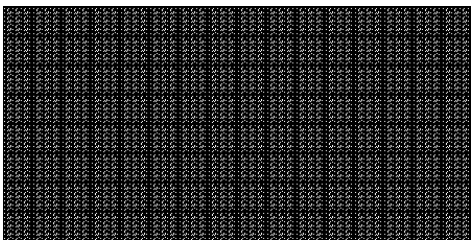
(f) TW: layer 3, 4 of gen. and disc. & all biases



(g) TW: layers 3, 4 of gen., all layers of disc. & all biases

(h) TW: all layers of gen., layer 1, 4 of disc. & all biases, 5 disc. updates

(i) TW: all



(j) TW: all, no BN used

Figure 20: randomly generated samples after 20k iterations with different choices of trainable weights (TW), last row: baseline

the results.

Finally we test to train all layers of one of the players while fixing some layers of the other. We observe that a certain balance between generator and discriminator has to be kept, since the training of DCGAN quickly gets unstable. On the one side, if the discriminator has many more trainable weights than the generator, it is likely that similar problems occur as if the discriminator is updated too often, since in both cases it is trained too well (confer Section 2.2). In line with this, training only the last layer and the biases of the generator but all layers of the discriminator, does not produce meaningful results in our experiments. However, the training works when the last 2 layers of the generator are trainable (Figure 20g).
On the other side, if the generator has a lot more trainable weights than the discriminator, it can happen that the generator does not need to produce good samples to trick the discriminator. Or from another point of view, the discriminator is not able to extract useful information with which the generator can improve. Even when training the first and last layer of the discriminator and all biases, the generator does not learn the distribution. However, this is not very surprising, since like this only 6145 weights of the discriminator are trainable, which is less than 0.6% of all its weights. On the other hand, more than 1.5 million weights of the generator are trainable. To restore the balance between the two players, we update the discriminator 5 times per generator update (Figure 20h). The resulting sample quality is nearly as high as for the standard DCGAN, where all weights are trainable (Figure 20i).

Although it is well known that training a GAN is a delicate task (also on the relatively easy MNIST dataset), looking at the samples one might think that the results with random weights are nothing special. One might even be tempted to think that the results are relatively bad (since the quality of the samples is not so high) and that nearly any algorithm could yield results like that. However, when training all weights of the same GAN but without using batch normalization, the generator is likely not to learn the target distribution at all and to generate only noise. Out of 5 runs, one produced samples similar to using batch normalization but all others ended up with results as shown in Figure 20j. This shows that with a lucky initialization the training can also work without batch normalization, but it does not work reliably. Hence, we conclude that this small change to the architecture immediately makes the training unstable. This shows again the high importance of the right choice of the architecture. A GAN with a good architecture with mostly random weights can outperform a GAN with a bad architecture where all weights are trainable. Still, training the weights of a GAN with good architecture improves the results.

4.3.2 GANs with perturbed weights

In Section 4.3.1 we have seen that training the last layer together with all biases of generator and discriminator is enough to learn a substantial amount of the distribution of the MNIST dataset. However, the sample quality was not appealing. We conclude that choosing 99.6% of the weights totally at random is too drastic to achieve good sample quality. On the other hand, it became clear that those 99.6% of the weights are not so important for the success of the training. In particular, we claim that choosing these weights not totally at random, but with an educated guess and training the other 0.4% percent of the weights afterwards, one can achieve much better sample quality.

To test this claim, we first train a standard DCGAN for 20'000 iterations, such that its generator is able to produce samples very similar to the true MNIST data. We denote the weights of the trained generator by $\theta_0 \in \mathbb{R}^{n_G}$ and the ones of the discriminator by $\omega_0 \in \mathbb{R}^{n_D}$. We assume that initial choices for the weights that are not too far from θ_0 and ω_0 should yield better results than totally random weights. Therefore, we perturb these vectors as follows. For every layer we compute the standard deviation of all weights in the weight matrix and in the bias vector separately. Then we sample i.i.d. noise for each weight, where the noise is drawn from a centered normal distribution with standard deviation given by some scalar factor, $\lambda > 0$, times the standard deviation computed for the respective region the weight belongs to. Denoting the resulting noise vectors as ϵ_G and ϵ_D, we perturb the original weights by adding the noise, yielding the initialization $\theta_0 + \epsilon_G$ and $\omega_0 + \epsilon_D$. Starting from these weights, we now train the last layer of the generator and discriminator together with all bias terms for 10'000 iterations. In Figure 21 we display the results for different values of λ. For all samples the same random input noise vectors (i.e. the input for the generator) were used to generate them. However, this does not mean that after retraining the generated samples belong to the same class as before, since a completely unsupervised learning procedure is used.

For the small factor $\lambda = 0.5$ we see that the sample quality is only reduced by a little, with the edges of the digits getting less sharp. The previous quality can be restored by retraining the chosen 0.4% of the weights.

Doubling the standard deviation factor to $\lambda = 1$, the quality drops much more when perturbing the weights. Still, the retraining of the last layer and all biases yields samples of nearly the same quality as before perturbation. In particular, it is higher than the quality of most samples generated with random weights in Figure 20.

After another doubling of the factor to $\lambda = 2$, the generator only produces

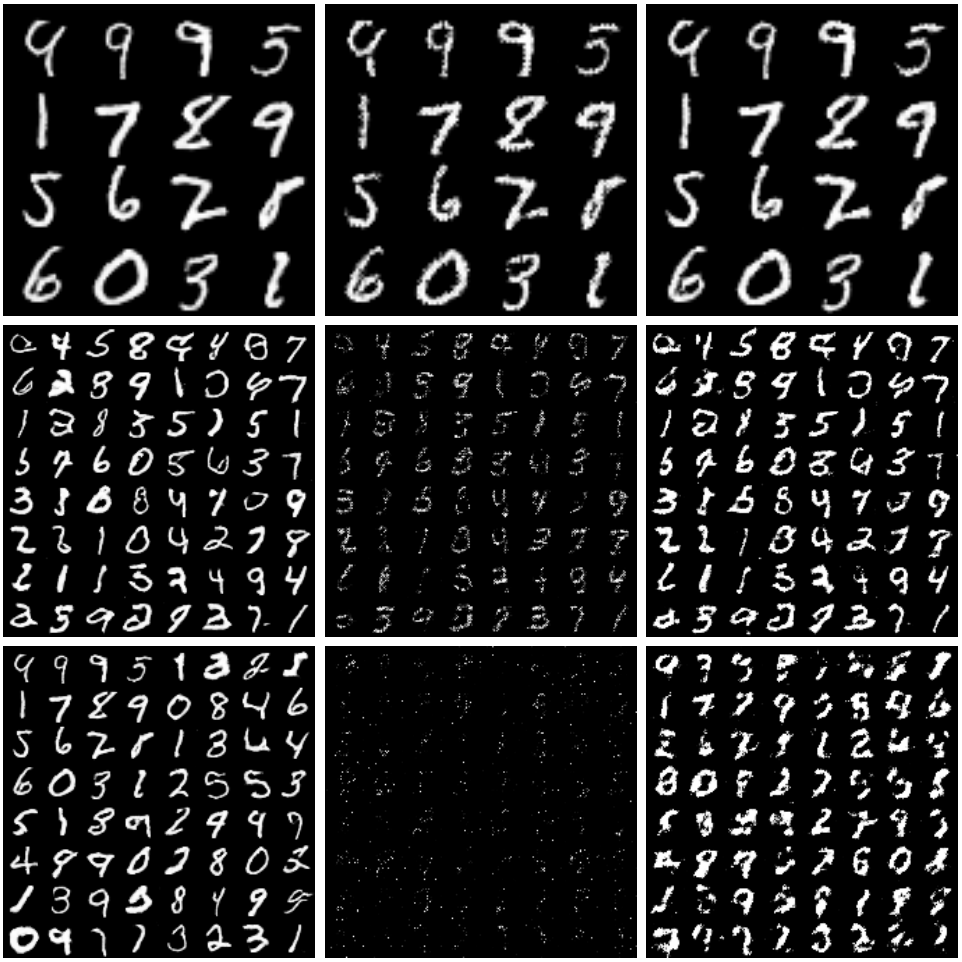


Figure 21: Samples generated before (left) and after (middle) perturbing the weights and after training the last layer and all biases for 10k iterations (right). 1st row: $\lambda = 0.5$ (less samples shown to see the small differences), 2nd row: $\lambda = 1$, 3rd row: $\lambda = 2$.

Figure 22: Zoomed version of samples generated before (left) and after (right) perturbing the weights with uniform noise and $\lambda = 0.5$.

noise. Now retraining of the selected 0.4% of the weights does not yield so high quality any more. However, the results are still much better than when training the same weights but starting from totally random initializations.

These results empirically demonstrate that the effect of perturbing all layers with noise can mostly be compensated by training only the last layer and the biases. One can also interpret the results as an empirical proof that training as few as 0.4% of the weights of a GAN can be enough to reach considerable sample quality, when starting from a point that is not too bad (case $\lambda = 1$).

We conduct another experiments to test the sensitivity of the generator to perturbations of its weights. In particular, we want to see the influence of small and bounded noise terms added to the trained weights. Therefore, we sample i.i.d. noise from a uniform distribution, $\epsilon \sim U(-a, a)$. The limit a is given by a scalar factor, $\lambda > 0$, times the standard deviation of the respective weight matrix or bias vector (as above). We compare samples generated before and after perturbation of the weights with these noise terms by the *mean squared pixel-wise distance* between them. Let $U := \{u_1, \ldots, u_m\} \subset [0, 1]^n$ be flattened image samples generated before perturbation and $V := \{v_1, \ldots, v_m\} \subset [0, 1]^n$ the samples generated from the same input noise vectors but after perturbation. We assume here that the pixel values lie in the interval $[0, 1]$, rather than being integers in $[0, 255]$. Then, the mean squared pixel-wise distance between them is computed as

$$d_{\text{pixel}}(U, V) := \frac{1}{|U|} \sum_{i=1}^{|U|} \|u_i - v_i\|_2^2.$$

This distance reflects the difference of two sets of image samples well. Note that the order of the images plays a crucial role.

We choose $\lambda = 0.5$ and repeat the perturbation experiment 10'000 times. Each time 128 samples are generated and compared. To make the results of the different runs comparable, always the same random input noise vectors are used to generate the samples. In Figure 22 we show some of the samples of the run where the highest pixel-wise error was measured. We see that the perturbation of the weights only has a small effect on the output.

This shows that well trained GANs are rather insensitive to noise perturbing their weights, as long as the noise is not too big.

4.3.3 Pre-training GANs on a subset of the data

In Section 4.3.2 we have tested one way to fix most weights of a GAN such that the weights are a mix of prior knowledge and randomness. In this section we test another possibility to do this, which has also more significance for practical applications.

In particular, we choose a random subset of 5 digits out of $\{0, \ldots, 9\}$ and train a standard GAN on the subset of the MNIST dataset containing these digits. This GAN is trained for 20'000 iterations, whereby all weights are trainable. We assume that the trained weights of layer 1 to 3 have learnt information that is not only useful for this subset of MNIST, but for the entire MNIST dataset respectively the complementary MNIST subset. However, it is unlikely that the trained weights are optimal for one of these other tasks. The question is, whether the GAN can reuse the learnt information for a different but still similar task.
To test this, we retrain only the last layer of the generator and discriminator together with all biases (as mentioned before, this is 0.4% of all weights), either on the entire dataset or on the complementary subset, for another 20'000 iterations. Especially when retraining on the entire dataset, it is likely that the GAN mostly ignores the newly added target samples. To make it easier for the GAN to learn the new target samples as well, we test to perturb the weights before starting the second training. The perturbation is done as in Section 4.3.2, with centered random normal noise terms, whose standard deviations depend on a scalar factor $\lambda > 0$. We choose to use $\lambda \in \{0, 0.5, 1\}$ and show the results in Figure 23 and 24.

We first discuss the results when retraining on the entire MNIST dataset (Figure 23). Without perturbation, only very few of the generated samples represent digits of the newly added targets, as it was expected. Perturbing the weights with $\lambda = 0.5$, many more of the generated samples belong to the complementary subset. At the same time, the quality of the samples is nearly as high as for the standard GAN. Increasing the scalar factor to $\lambda = 1$, we see a much bigger drop in sample quality, while the diversity is not

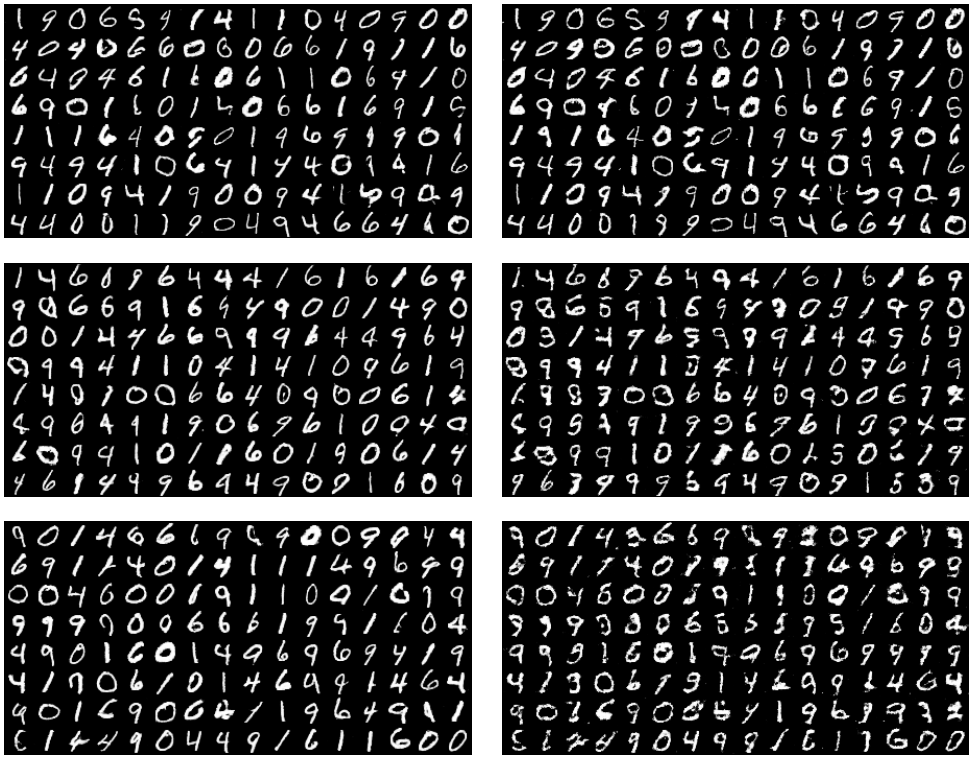


Figure 23: Samples after training a DCGAN on a subset of MNIST (left) and after retraining the last layer and all biases on the entire MNIST dataset (right). The subset of MNIST includes the digits 0, 1, 4, 6, 9 which were chosen at random. 1st row: $\lambda = 0$, 2nd row: $\lambda = 0.5$, 3rd row: $\lambda = 1$.

increased that much. Overall, we do not see any sample of the digit 2, which is only by chance, due to the selected random noise vectors that are used to generate the samples. Generating more samples with the models, also the digit 2 shows up (not shown here).

When retraining happens only on the complementary subset of the MNIST dataset (Figure 24), the GAN generates many samples belonging to the new subset, already without perturbation. However, it does not forget the previously learned distribution entirely and still generates samples of all digits of the first training set. The sample quality is very high. Using perturbations with $\lambda = 0.5$, more generated samples belong to the second training set, but the quality is also lower. For $\lambda = 1$ the sample quality is already quite low, without higher diversity than for $\lambda = 0.5$.

These results show the ability of GANs to use information learnt in the training on one dataset, when retrained on another dataset that is not too different. It also shows that the weights of the first layers are not so important

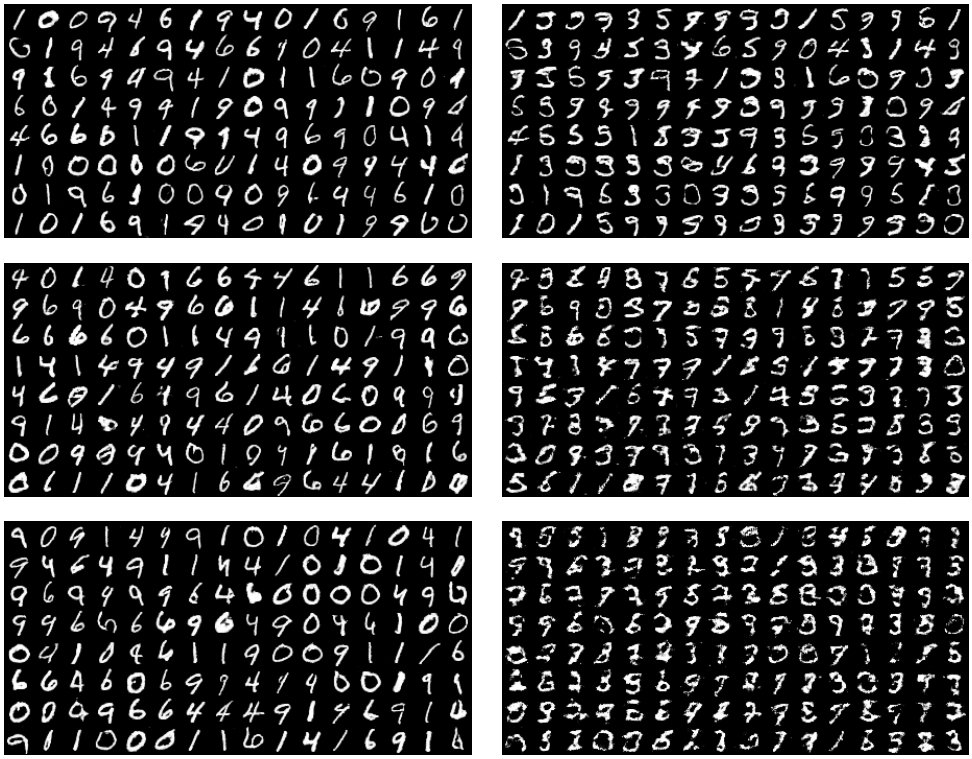


Figure 24: Samples after training a DCGAN on a subset of MNIST (left) and after retraining the last layer and all biases on the complementary subset of MNIST (right). The first subset includes the digits 0, 1, 4, 6, 9 which were chosen at random. 1st row: $\lambda = 0$, 2nd row: $\lambda = 0.5$, 3rd row: $\lambda = 1$.

for the successful training of a GAN. One only needs weights that do well on an easier or similar dataset for the first layers. Then the retraining of the last layer and all biases (0.4% of all weights) can yield very high sample quality on the actual target dataset.

In practice this might have an application, if there is only a few samples of the target distribution, but a large amount of samples of a similar distribution. On the large dataset, the GAN can learn the main structure of the problem, while it learns the specialities during the retraining on the small target dataset. Although a small target dataset is used, it is likely that no overfitting happens in the retraining, since there is also only a small number of trainable weights. Retraining all weights, this might be a more serious problem.

Wang et al. [80] also studied GANs that are retrained on the target dataset after pre-training them on a different dataset. They show that pre-training can improve the speed of convergence and helps when only a small

target dataset is available. However, they do not fix any of the weights during retraining. Doing this might be an interesting topic for future research.

4.4 Conclusion

In this chapter, we have seen different experimental results supporting the hypothesis that the architecture of a GAN is more important than the training of most of its weights. Training only the last layer and all biases of the generator and discriminator, while keeping all other weights fixed after random initialization, a GAN can successfully learn parts of the target distribution. If the initialization of the weights is not done totally at random but makes use of some previous knowledge of the problem, the results get much better and reach nearly the same quality as the results of a standard GAN where all weights are trainable. We have also seen that GANs are quite robust against noise terms perturbing their weights.

Overall, our results support the hypothesis that there are many points (or even open subsets) in the parameter space, which can yield very good generative results, if the architecture is suitable for the problem at hand. This makes it also clearer why the complex training of GANs, and more generally the training of neural networks, can reliably yield state-of-the-art results, although a local optimization scheme is used for a highly non-convex problem. With a good choice of architecture, the sophisticated variance scaling initialization methods populate the weights with values, such that they are close to the next point (or subset) in the parameter space yielding a good solution for the problem.

References

[1] Pre-trained Inception Model. *http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz*, Last-Checked: December 2nd, 2018.

[2] Minimax principle. *Encyclopedia of Mathematics. URL: http://www.encyclopediaofmath.org/index.php?title=Minimax_principle&oldid=34361*, Last checked: March 18, 2019.

[3] Two-person zero-sum game. *Encyclopedia of Mathematics. URL: http://www.encyclopediaofmath.org/index.php?title=Two-person_zero-sum_game&oldid=34414*, Last checked: March 18, 2019.

[4] Forum discussion. *https://stats.stackexchange.com/questions/7758/on-connection-weights-in-an-artificial-neural-network*, Last checked: March 6, 2019.

[5] Forum discussion. *https://www.reddit.com/r/MachineLearning/comments/8fqcfv/d_why_is_zdimension_for_gans_usually_100/*, Last-Checked: November 12, 2018.

[6] D. Achlioptas. Database-friendly random projections: Johnson-Lindenstrauss with binary coins. *Journal of Computer and System Sciences*, 66(4):671 – 687, 2003. Special Issue on PODS 2001.

[7] Anonymous. Universal Lipschitz Functions. In *Submitted to International Conference on Learning Representations*, 2019. under review.

[8] M. Arjovsky and L. Bottou. Towards Principled Methods for Training Generative Adversarial Networks. *CoRR*, abs/1701.04862, 2017.

[9] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein GAN. 01 2017.

[10] D. H. Ballard. Modular Learning in Neural Networks. In *Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 1*, AAAI'87, pages 279–284. AAAI Press, 1987.

[11] D. Bang and H. Shim. MGGAN: Solving Mode Collapse using Manifold Guided Training. *CoRR*, abs/1804.04391, 2018.

[12] S. Barratt and R. Sharma. A Note on the Inception Score. *ArXiv e-prints*, page arXiv:1801.01973, Jan. 2018.

[13] M. G. Bellemare, I. Danihelka, W. Dabney, S. Mohamed, B. Lakshminarayanan, S. Hoyer, and R. Munos. The Cramer Distance as a Solution to Biased Wasserstein Gradients. *CoRR*, abs/1705.10743, 2017.

[14] M. Ben-Yosef and D. Weinshall. Gaussian Mixture Generative Adversarial Networks for Diverse Datasets, and the Unsupervised Clustering of Images. *ArXiv e-prints*, Aug. 2018.

[15] P. Bojanowski, A. Joulin, D. Lopez-Paz, and A. Szlam. Optimizing the Latent Space of Generative Networks. *ArXiv e-prints*, July 2017.

[16] N. Bonnotte. Unidimensional and Evolution Methods for Optimal Transportation. PhD thesis, 2013.

[17] O. Bousquet, S. Gelly, I. Tolstikhin, C. J. Simon-Gabriel, and B. Schölkopf. From Optimal Transport to Generative Modeling: the VEGAN cookbook. Technical report, 2017.

[18] W. Cao, X. Wang, Z. Ming, and J. Gao. A Review on Neural Networks with Random Weights. *Neurocomput.*, 275(C):278–287, Jan. 2018.

[19] S. Dasgupta and A. Gupta. An Elementary Proof of a Theorem of Johnson and Lindenstrauss. *Random Struct. Algorithms*, 22(1):60–65, Jan. 2003.

[20] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[21] E. L. Denton, S. Chintala, A. Szlam, and R. Fergus. Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks. *CoRR*, abs/1506.05751, 2015.

[22] I. Deshpande, Z. Zhang, and A. G. Schwing. Generative Modeling using the Sliced Wasserstein Distance. *CoRR*, abs/1803.11188, 2018.

[23] R. Durrett. *Probability: Theory and Examples.* Cambridge University Press, New York, NY, USA, 4th edition, 2010.

[24] Y. Freund and R. E. Schapire. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *J. Comput. Syst. Sci.*, 55(1):119–139, Aug. 1997.

[25] A. Genevay, M. Cuturi, G. Peyré, and F. Bach. Stochastic Optimization for Large-scale Optimal Transport. In D. D. Lee, M. Sugiyama, U. V.

Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 3440–3448. Curran Associates, Inc., 2016.

[26] A. Genevay, G. Peyre, and M. Cuturi. Learning Generative Models with Sinkhorn Divergences. In A. Storkey and F. Perez-Cruz, editors, *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, volume 84 of *Proceedings of Machine Learning Research*, pages 1608–1617, Playa Blanca, Lanzarote, Canary Islands, 09–11 Apr 2018. PMLR.

[27] A. Ghosh, V. Kulharia, V. P. Namboodiri, P. H. S. Torr, and P. K. Dokania. Multi-Agent Diverse Generative Adversarial Networks. *CoRR*, abs/1704.02906, 2017.

[28] R. Giryes, G. Sapiro, and A. M. Bronstein. Deep Neural Networks with Random Gaussian Weights: A Universal Classification Strategy? *IEEE Transactions on Signal Processing*, 64, 04 2015.

[29] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[30] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[31] G. L. Grinblat, L. C. Uzal, and P. M. Granitto. Class-Splitting Generative Adversarial Networks. *ArXiv e-prints*, Sept. 2017.

[32] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville. Improved Training of Wasserstein GANs. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5767–5777. Curran Associates, Inc., 2017.

[33] S. Gurumurthy, R. K. Sarvadevabhatla, and V. B. Radhakrishnan. DeLiGAN: Generative Adversarial Networks for Diverse and Limited Data. *CoRR*, abs/1706.02071, 2017.

[34] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *ArXiv e-prints*, page arXiv:1512.03385, Dec. 2015.

[35] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[36] A. Heljakka, A. Solin, and J. Kannala. Pioneer Networks: Progressively Growing Generative Autoencoder. *CoRR*, abs/1807.03026, 2018.

[37] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, G. Klambauer, and S. Hochreiter. GANs Trained by a Two Time-Scale Update Rule Converge to a Nash Equilibrium. *CoRR*, abs/1706.08500, 2017.

[38] Q. Hoang, T. D. Nguyen, T. Le, and D. Q. Phung. Multi-Generator Generative Adversarial Nets. *CoRR*, abs/1708.02556, 2017.

[39] G.-B. Huang, L. Chen, and C.-K. Siew. Universal Approximation Using Incremental Constructive Feedforward Networks with Random Hidden Nodes. *Trans. Neur. Netw.*, 17(4):879–892, July 2006.

[40] G.-B. Huang, Q.-Y. Zhu, and C. Siew. Extreme Learning Machine: A New Learning Scheme of Feedforward Neural Networks. In *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)*, volume 2, pages 985–990 vol.2, July 2004.

[41] X. Huang, Y. Li, O. Poursaeed, J. E. Hopcroft, and S. J. Belongie. Stacked Generative Adversarial Networks. *CoRR*, abs/1612.04357, 2016.

[42] T. Huster, C.-Y. J. Chiang, and R. Chadha. Limitations of the Lipschitz constant as a defense against adversarial examples. *CoRR*, abs/1807.09705, 2018.

[43] P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[44] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*, abs/1502.03167, 2015.

[45] W. Johnson and J. Lindenstrauss. Extensions of Lipschitz maps into a Hilbert space. *Contemporary Mathematics*, 26:189–206, 01 1984.

[46] T. Karras, T. Aila, S. Laine, and J. Lehtinen. Progressive Growing of GANs for Improved Quality, Stability, and Variation. *CoRR*, abs/1710.10196, 2017.

[47] K. Kawaguchi. Deep Learning without Poor Local Minima. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 586–594. Curran Associates, Inc., 2016.

[48] D. P. Kingma and M. Welling. Auto-Encoding Variational Bayes. *CoRR*, abs/1312.6114, 2013.

[49] L. Klebanov. *N-distances and their Applications*. 01 2005.

[50] S. Kolouri, C. Martin, and G. K. Rohde. Sliced-Wasserstein Autoencoder: An Embarrassingly Simple Generative Model. 04 2018.

[51] A. Krizhevsky. Learning Multiple Layers of Features from Tiny Images. 2009.

[52] A. B. L. Larsen, S. K. Sønderby, and O. Winther. Autoencoding beyond pixels using a learned similarity metric. *CoRR*, abs/1512.09300, 2015.

[53] Y. LeCun and C. Cortes. MNIST Handwritten Digit Database. 2010.

[54] C. Li, W. Chang, Y. Cheng, Y. Yang, and B. Póczos. MMD GAN: Towards Deeper Understanding of Moment Matching Network. *CoRR*, abs/1705.08584, 2017.

[55] Y. Li, K. Swersky, and R. S. Zemel. Generative Moment Matching Networks. *CoRR*, abs/1502.02761, 2015.

[56] Z. Liu, P. Luo, X. Wang, and X. Tang. Deep Learning Face Attributes in the Wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, 2015.

[57] A. Makhzani, J. Shlens, N. Jaitly, and I. J. Goodfellow. Adversarial Autoencoders. *CoRR*, abs/1511.05644, 2015.

[58] M. Mirza and S. Osindero. Conditional Generative Adversarial Nets. *CoRR*, abs/1411.1784, 2014.

[59] T. Nguyen, T. Le, H. Vu, and D. Phung. Dual Discriminator Generative Adversarial Nets. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances*

in Neural Information Processing Systems 30, pages 2670–2680. Curran Associates, Inc., 2017.

[60] A. Odena, C. Olah, and J. Shlens. Conditional Image Synthesis With Auxiliary Classifier GANs. *ArXiv e-prints*, page arXiv:1610.09585, Oct. 2016.

[61] Y.-H. Pao, G.-H. Park, and D. J. Sobajic. Learning and Generalization Characteristics of the Random Vector Functional-Link Net. *Neurocomputing*, 6(2):163 – 180, 1994. Backpropagation, Part IV.

[62] J. Poland. Three Different Algorithms for Generating Uniformly Distributed Random Points on the N-Sphere. 2000.

[63] A. Radford, L. Metz, and S. Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *CoRR*, abs/1511.06434, 2015.

[64] W. Rawat and Z. Wang. Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review. *Neural Comput.*, 29(9):2352–2449, Sept. 2017.

[65] P. K. Rubenstein, B. Schoelkopf, and I. Tolstikhin. On the Latent Space of Wasserstein Auto-Encoders. *arXiv e-prints*, page arXiv:1802.03761, Feb. 2018.

[66] T. Salimans, I. J. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen. Improved Techniques for Training GANs. *CoRR*, abs/1606.03498, 2016.

[67] T. Salimans, H. Zhang, A. Radford, and D. N. Metaxas. Improving GANs Using Optimal Transport. *CoRR*, abs/1803.05573, 2018.

[68] A. M. Saxe, P. W. Koh, Z. Chen, M. Bhand, B. Suresh, and A. Y. Ng. On Random Weights and Unsupervised Feature Learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 1089–1096, USA, 2011. Omnipress.

[69] J. Schmidhuber. Deep Learning in Neural Networks: An Overview. *CoRR*, abs/1404.7828, 2014.

[70] R. K. Sinha, R. Pandey, and R. Pattnaik. Deep Learning For Computer Vision Tasks: A review. *CoRR*, abs/1804.03928, 2018.

[71] R. Sinkhorn. A Relationship Between Arbitrary Positive Matrices and Doubly Stochastic Matrices. *Ann. Math. Statist.*, 35(2):876–879, 06 1964.

[72] D. Soudry and Y. Carmon. No bad local minima: Data independent training error guarantees for multilayer neural networks. *arXiv e-prints*, page arXiv:1605.08361, May 2016.

[73] A. Srivastava, L. Valkov, C. Russell, M. U. Gutmann, and C. Sutton. VEEGAN: Reducing Mode Collapse in GANs using Implicit Variational Learning. *ArXiv e-prints*, May 2017.

[74] I. O. Tolstikhin, O. Bousquet, S. Gelly, and B. Schölkopf. Wasserstein Auto-Encoders. *CoRR*, abs/1711.01558, 2017.

[75] I. O. Tolstikhin, S. Gelly, O. Bousquet, C.-J. Simon-Gabriel, and B. Schölkopf. AdaGAN: Boosting Generative Models. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5424–5433. Curran Associates, Inc., 2017.

[76] D. Ulyanov, A. Vedaldi, and V. S. Lempitsky. It Takes (Only) Two: Adversarial Generator-Encoder Networks. In *AAAI*, 2018.

[77] C. Villani. *Optimal Transport: Old and New*. Springer, 2009.

[78] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 3rd edition, 1944.

[79] D. Wang and Q. Liu. Learning to Draw Samples: With Application to Amortized MLE for Generative Adversarial Learning. *ArXiv e-prints*, page arXiv:1611.01722, Nov. 2016.

[80] Y. Wang, C. Wu, L. Herranz, J. van de Weijer, A. Gonzalez-Garcia, and B. Raducanu. Transferring GANs: generating images from limited data. *CoRR*, abs/1805.01677, 2018.

[81] J. Wu, Z. Huang, W. Li, and L. V. Gool. Sliced Wasserstein Generative Models. *CoRR*, abs/1706.02631, 2018.

[82] C. Xiao, P. Zhong, and C. Zheng. BourGAN: Generative Networks with Metric Embeddings. *arXiv preprint arXiv:1805.07674*, 2018.

[83] C. Yun, S. Sra, and A. Jadbabaie. A Critical View of Global Optimality in Deep Learning. *CoRR*, abs/1802.03487, 2018.

Appendices

A JL-SWGAN architectures

In Table 3 and 4 we show the architecture for the JL-SWGAN and SWG with discriminator, when the target images are of size 32×32. Layer 4^* of the generator is an extra layer that is not always used. For bigger image sizes, copies of layer 2 in the generator respectively copies of layer 1 in the discriminator are added. The number of features of these copies and the following layers of the generator is adjusted in the obvious way, by dividing them by 2 for every added layer.

Layer	**Operation**	**Activation Fn.**	**Output shape**
Input	latent variable $Z \sim N(0, I_d)$	-	128
1a	fully connected	ReLU	$16 \cdot c_g$
1b	reshape	-	$4 \times 4 \times c_g$
2	convolution 2D transpose, 4×4 kernel, stride 2	ReLU	$8 \times 8 \times c_g/2$
3	convolution 2D transpose, 4×4 kernel, stride 2	ReLU	$16 \times 16 \times c_g/4$
4^*	convolution 2D transpose, 4×4 kernel, stride 1	ReLU	$16 \times 16 \times c_g/4$
5	convolution 2D transpose, 4×4 kernel, stride 2	tanh	$32 \times 32 \times$ (1 or 3)

Table 3: generator architecture for JL-SWGAN

Layer	**Operation**	**Activation Fn.**	**Output shape**
Input	image	-	$32 \times 32 \times$ (1 or 3)
1	convolution 2D, 4×4 kernel, stride 2	LeakyReLU	$16 \times 16 \times c_d/4$
2	convolution 2D, 4×4 kernel, stride 2	LeakyReLU	$8 \times 8 \times c_d/2$
3a	convolution 2D, 4×4 kernel, stride 2	LeakyReLU	$4 \times 4 \times c_d$
3b	reshape	-	$16 \cdot c_d$
4	fully connected	sigmoid	1

Table 4: discriminator architecture for JL-SWGAN

The variables c_g and c_d are the numbers of maximal feature maps in the generator and discriminator. They are chosen differently depending on the target data set and wanted number of parameters. Our choices are $c_g = c_d = 256$ for MNIST [53], $c_g = c_d = 512$ for CIFAR-10 [51] and $c_g = 1024$, $c_d = 512$ for CelebA [56]. Whenever batch normalization or layer normalization is used, it is applied after all but the last layer.

For completeness we also give the definitions of the the different activation functions.

Definition A.1. Let

$$f_\alpha : \mathbb{R} \to \mathbb{R}, x \mapsto x\,\mathbf{1}_{x>0} + \alpha\, x\,\mathbf{1}_{x\leq 0}.$$

Then the *rectified linear unit* (ReLU) is defined as f_0. Leaky ReLU is defined as f_α with $\alpha > 0$ and the default value is $\alpha = 0.2$ (used in all our experiments). The sigmoid function is defined as

$$\sigma : \mathbb{R} \to [0,1], x \mapsto \tfrac{\exp(x)}{1+\exp(x)}.$$

B DCGAN architectures for MNIST

The generator and discriminator architecture of DCGAN for the MNIST dataset is given in Table 5 and 6. Again, c_g and c_d are the numbers of maximal feature maps in the generator respectively discriminator. Usually, they are chosen equal, $c_g = c_d$, and in our experiments we mostly use the value 256 for them. If batch normalization is used, it is applied after all but the last layer. In layers with batch normalization, the usual bias term is replaced by a bias of the batch normalization, but no further scaling factor is used. This way, the number of trainable parameters remains the same. In particular, for the default choice $c_g = c_d = 256$, the generator has 1'554'177 and the discriminator 1'030'145 trainable parameters. Although the discriminator architecture mirrors the generator architecture, the difference in the number of weights is quite large. This is mainly due to the fact that the fully connected layer of the generator has 128 inputs, while the fully connected layer of the discriminator has only 1 output.

For a definition of the activation functions confer Appendix A.

C Time improvement of GANs with random weights

We conduct an experiment to test the time improvement of GANs with random weights over standard GANs. We use the MNIST dataset [53] with

Layer	Operation	Activation Fn.	Output shape
Input	latent variable $Z \sim N(0, I_d)$	-	128
1a	fully connected	ReLU	$16 \cdot c_g$
1b	reshape	-	$4 \times 4 \times c_g$
2	convolution 2D transpose, 5×5 kernel, stride 2	ReLU	$8 \times 8 \times c_g/2$
3	convolution 2D transpose, 5×5 kernel, stride 2	ReLU	$16 \times 16 \times c_g/4$
4a	convolution 2D transpose, 5×5 kernel, stride 2	sigmoid	$32 \times 32 \times 1$
4b	cut off 2 pixels at borders	-	$28 \times 28 \times 1$

Table 5: generator architecture for DCGAN

Layer	Operation	Activation Fn.	Output shape
Input	image	-	$28 \times 28 \times 1$
1	convolution 2D, 5×5 kernel, stride 2	LeakyReLU	$14 \times 14 \times c_d/4$
2	convolution 2D, 5×5 kernel, stride 2	LeakyReLU	$7 \times 7 \times c_d/2$
3a	convolution 2D, 5×5 kernel, stride 2	LeakyReLU	$4 \times 4 \times c_d$
3b	reshape	-	$16 \cdot c_d$
4	fully connected	sigmoid	1

Table 6: discriminator architecture for DCGAN

standard image size 28×28 and train standard DCGANs with the architecture given in Appendix B. We consider three different configurations. Either all layers are trainable, or only the last layer or the last layer and the biases of all layers (these are bias terms of the batch normalization for layer 1 to 3). For all configurations we measure t_1: the time of one training iterations, which amounts to computing the loss of generator and discriminator (forward run through network), computing their gradients and updating their weights (backward run) and t_2: the time to compute the loss of the generator and discriminator (forward run). Since the time of the forward run should be the same for all configurations, we call it one time unit and compute the time of one training iteration in time units (t_1/t_2). These are the times we will compare. We do this over a period of 100 iterations and display the average of the observed times in Table 7.

Trainable	t_1	t_2	t_1/t_2
all	1.237	0.392	3.157
last & all biases	0.991	0.386	2.568
last	0.578	0.407	1.418

Table 7: observed training times averaged over 100 iterations

Comparing the training times when only the last layer is trainable and when all layers are trainable we get a ratio of 0.45. On the other hand, when also the biases are trainable this ratio is nearly doubled being 0.81. This is due to the biases being in all layers making it necessary to compute nearly all gradients. However, since there are so little bias terms, most of the weight updates are still omitted.

D Layer statistics and distributions of trained DCGAN

In this section we consider a DCGAN with architecture given in Table 5 and 6 in Appendix B, which was trained for 20'000 iterations. For each layer of this model some standard statistics are computed and displayed in Table 8, whereby the weight matrices and the bias vectors are examined separately. Furthermore, histograms of the weights are plotted in Figure 25 (weight matrices) and 26 (bias vectors).

In Figure 25 we see that the weight matrices of the first 3 layers seem to be more or less normally distributed. This is not the case for the last layer. Although we observe that the distributions of the first 3 layers are very similar to centered normal distributions, this does not mean that the single weights are random. In fact, if they were random this would imply that no training would be needed for them, as soon as they are initialized with the correct distribution. However, our results in Section 4.3.1 did show that the training of GANs with random weights can work, but training all the weights always improved the results. This is a clear evidence that the weights of a trained GAN are not random. Nevertheless, we test training a GAN where the weight matrices of layer 1 to 3 are fixed after initializing them with random samples drawn from the respective normal distributions plotted in Figure 25. This means that only the last layer and all biases are kept trainable. The training with this initialization method does not work at all, i.e. the generator only produces noise after both of our runs of the experiment. On the other hand, when using a variance scaling initializer and training the same weights, we

generator

Layer	size	mean	median	std	min	max	skew	kurtosis
FC w	524288	-0.0001	0.0001	0.0379	-0.1686	0.1705	-0.0199	-0.0453
FC b	4096	-0.0648	-0.0586	0.1567	-0.544	0.4563	-0.0395	-0.5614
Conv1 w	819200	-0.0079	-0.0068	0.0348	-0.1974	0.1542	-0.2272	0.4303
Conv1 b	128	-0.3425	-0.3408	0.0715	-0.5074	-0.1558	0.0426	0.2001
Conv2 w	204800	-0.002	-0.0003	0.037	-0.2667	0.1781	-0.4082	1.1802
Conv2 b	64	0.1573	0.1756	0.0933	-0.235	0.3019	-1.6157	3.9218
Conv3 w	1600	-0.1219	-0.0601	0.2942	-1.3939	0.6486	-0.9758	1.2925
Conv3 b	1	0.1383	0.1383	0.0	0.1383	0.1383	0.0	-3.0

discriminator

Layer	size	mean	median	std	min	max	skew	kurtosis
Conv1 w	1600	-0.0058	-0.0013	0.0443	-0.2343	0.1183	-0.6128	1.0798
Conv1 b	64	-0.0675	-0.0676	0.0541	-0.1703	0.0221	-0.0559	-1.0072
Conv2 w	204800	-0.0003	-0.0001	0.0329	-0.1684	0.2162	-0.0537	0.2536
Conv2 b	128	-0.1462	-0.1411	0.0676	-0.312	0.0045	-0.2778	-0.4462
Conv3 w	819200	-0.0007	-0.0006	0.0288	-0.2145	0.2767	-0.0118	0.6445
Conv3 b	256	-0.0326	-0.0301	0.0278	-0.1282	0.0695	-0.8014	1.6845
FC w	4096	0.0008	-0.0006	0.0749	-0.3414	0.2296	-0.075	-0.5115
FC b	1	0.0097	0.0097	0.0	0.0097	0.0097	0.0	-3.0

Table 8: Statistics of the trained weights of a standard DCGAN. For each layer the weight matrix (w) and bias vector (b) are considered separately. The kurtosis is computed with Fisher's variant.

saw that the training works reliably (Figure 20b in Section 4.3.1). Due to this result, we did not use the distributions of the trained weights in our experiments in Section 4.3.1.

One possible explanation why the trained weights of layer 1 to 3 are almost normally distributed is the following. Assume that the (possibly negative) increments, that are added to the initial value of one weight in the process of training with stochastic gradient descent (SGD), are independent. Then the Central Limit Theorem yields that this weight follows a normal distribution in the limit of infinitely many training steps. This idea was suggested in [4]. We formally proof this statement in Proposition D.3 under 2 assumptions.

Assumption D.1. *A SGD version is used, which updates the weights with their respective gradient of the loss function times the fixed learning rate* $\alpha/\sqrt{N}$*. Here* $\alpha \in \mathbb{R} \setminus \{0\}$ *is a constant and* N *is the total number of training steps that are done.*

Assumption D.2. *The gradients of the loss function that occur during the trainings process with N steps are independent random variables $\delta_{N,m}$, for $1 \leq m \leq N$, such that*

(i) there exists an integrable random variable Y (i.e. $\mathbb{E}[|Y|] < \infty$) with $\delta_{N,m}^2 \leq |Y|$ for all $1 \leq m \leq N$,

(ii) $\mathbb{E}[\delta_{N,m}] = 0$ for all $1 \leq m \leq N$ and

(iii) $\frac{1}{N}\sum_{m=1}^{N} \mathbb{E}[\delta_{N,m}^2] \xrightarrow{N\to\infty} \sigma_1^2 \in (0, \infty)$.

Proposition D.3. *Let w_N be a weight of a neural network after training with SGD for N steps. Assume that the weight w_N was initialized with $w_0 \sim N(0, \sigma_2^2)$ and that Assumptions D.1 and D.2 hold for its training. Furthermore, assume that w_0 is independent of $\delta_{N,m}$ for all $1 \leq m \leq N$ and all N. Then, we can write*

$$w_N = w_0 + \frac{\alpha}{\sqrt{N}} \sum_{m=1}^{N} \delta_{N,m},$$

and it holds that w_N converges in distribution to the normal distribution $N(0, \sigma_1^2\alpha^2 + \sigma_2^2)$, as $N \to \infty$.

Before proving this proposition, we make some remarks. First of all, Assumption D.1 is not too unrealistic. Common versions of SGD have an adaptive learning rate. This means that with progressing training the learning rate gets smaller. The more training steps one does, the smaller the learning rate will be in the end. So it is likely that the steps in the beginning have a bigger learning rate than the one that is assumed here, while the steps in the end have a smaller one. On average, the learning rate might be of the form in Assumption D.1.
Assumption D.2, on the other hand, is very strong. In particular, it seems rather unlikely that the gradients occurring during the training process are truly independent. However, in practice it is enough that they are "pseudo independent", i.e. that a large set of samples of them is hard to be distinguished from a large set of truly independent samples. Note that we only observe the sum of the samples, so also their order and a possible serial correlation does not matter. Furthermore, the assumption that w_0 is independent of all $\delta_{N,m}$ is not realistic. To the contrary, knowing the initial weights, the only randomness that is left in the training algorithm is the order of the target samples in the batches. Hence, it is likely that the gradients are dependent on w_0. However, also here it is enough to have a "pseudo independence"

between w_0 and all $\delta_{N,m}$.
The assumption on the expectation and convergence of the mean of variances seems to be relatively realistic. In particular, if the weights converge to a minimum, it is likely that they oscillate around this minimum, implying expectation 0 for the gradients, and that the gradients have similar variance while oscillating. Moreover, it is reasonable to assume that the gradients are bounded by some large number, especially when the weights converge to a minimum.
Assuming that all these assumptions are satisfied by the weights of a specific layer, there still has to be a large enough number of weights to be able to observe a normal distribution. The difference between a small and a large number of weights is evident when comparing the histograms of the weight matrices and the ones of the bias vectors. Furthermore, training for enough steps is crucial to see the behaviour that is proven only for the limit case of infinitely many training steps.

For the proof of Proposition D.3 we use the Lindeberg-Feller Theorem, a generalization of the Central Limit Theorem, as it is stated and proved in [23, Theorem 3.4.5].

Theorem D.4 (Lindeberg-Feller)**.** *For each n, let $X_{n,m}$, $1 \leq m \leq n$, be independent random variables with $\mathbb{E}[X_{n,m}] = 0$. Suppose*

(i) $\sum_{m=1}^{n} \mathbb{E}[X_{n,m}^2] \xrightarrow{n\to\infty} \sigma^2 > 0$,

(ii) For all $\epsilon > 0$, $\lim_{n\to\infty} \sum_{m=1}^{n} \mathbb{E}[X_{n,m}^2 \mathbf{1}_{\{|X_{n,m}|>\epsilon\}}] = 0$.

Then, $S_n := X_{n,1} + \cdots + X_{n,n}$ converges in distribution to $N(0, \sigma^2)$, as $n \to \infty$.

Proof (Proposition D.3). We define $X_{N,m} := \frac{\alpha}{\sqrt{N}} \delta_{N,m}$ and show that they satisfy the conditions of Lindeberg-Feller's Theorem. Assumption D.2 directly implies that for all N, the random variables $X_{N,m}$, $1 \leq m \leq N$, are independent with $\mathbb{E}[X_{N,m}] = \frac{\alpha}{\sqrt{N}} \mathbb{E}[\delta_{N,m}] = 0$. Furthermore,

$$\sum_{m=1}^{N} \mathbb{E}[X_{N,m}^2] = \sum_{m=1}^{N} \frac{\alpha^2}{N} \mathbb{E}[\delta_{N,m}^2] \xrightarrow{n\to\infty} \alpha^2 \sigma_1^2 > 0,$$

using Assumption D.2 (iii). Finally, fix some $\epsilon > 0$. Then we have

$$\begin{aligned} 0 &\leq \sum_{m=1}^{N} X_{N,m}^2 \mathbf{1}_{\{|X_{N,m}|>\epsilon\}} \\ &= \frac{\alpha^2}{N} \sum_{m=1}^{N} \delta_{N,m}^2 \mathbf{1}_{\{|\alpha\, \delta_{N,m}|>\epsilon\sqrt{N}\}} \\ &\leq \alpha^2 |Y| \mathbf{1}_{\{\alpha^2 |Y|>\epsilon^2 N\}} \xrightarrow[N\to\infty]{P-a.s} 0. \end{aligned}$$

The last term above is bounded by $\alpha^2|Y|$, which is integrable. Hence, the Dominated Convergence Theorem implies that

$$\lim_{N\to\infty} \sum_{m=1}^{N} \mathbb{E}[X_{N,m}^2 \, \mathbf{1}_{\{|X_{N,m}|>\epsilon\}}] = 0,$$

which proofs the last condition of the Lindeberg-Feller Theorem. Therefore,

$$\lim_{N\to\infty} \frac{\alpha}{N} \sum_{m=1}^{N} \delta_{N,m} = \lim_{N\to\infty} \sum_{m=1}^{N} X_{N,m} \sim N(0, \alpha^2\sigma_1^2).$$

Since w_0 is independent of all $\delta_{N,m}$, it is also independent of the limit of their sum as $N \to \infty$, implying that

$$\lim_{N\to\infty} w_N = w_0 + \lim_{N\to\infty} \sum_{m=1}^{N} X_{N,m} \sim N(0, \alpha^2\sigma_1^2 + \sigma_2^2),$$

using the well known formula for sums of independent random normal variables. $\square$

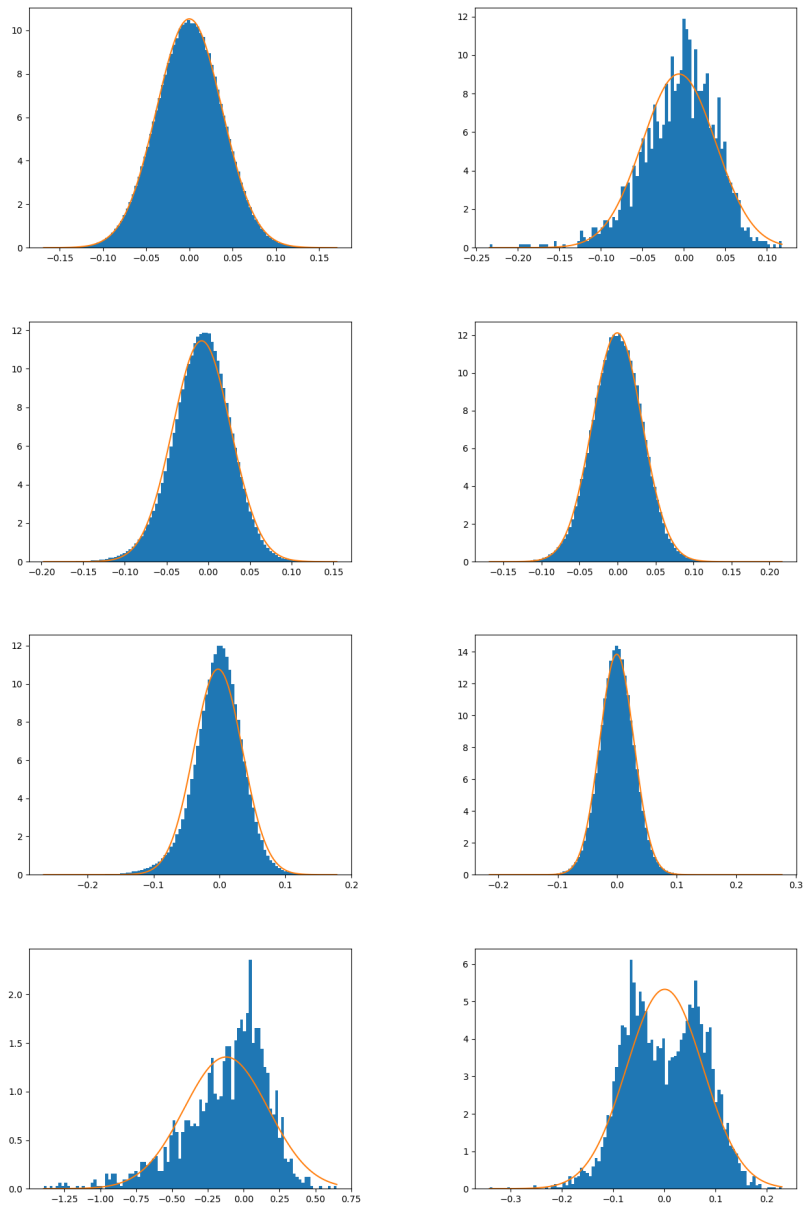


Figure 25: Distributions of the weight matrices of the generator (left) and the discriminator (right). In row i the distribution of layer i is shown. The histograms were generated with 100 bins. In each plot the orange curve is the density of a normal distribution with mean and standard deviation given by the sample mean and sample standard deviation of the respective weights.

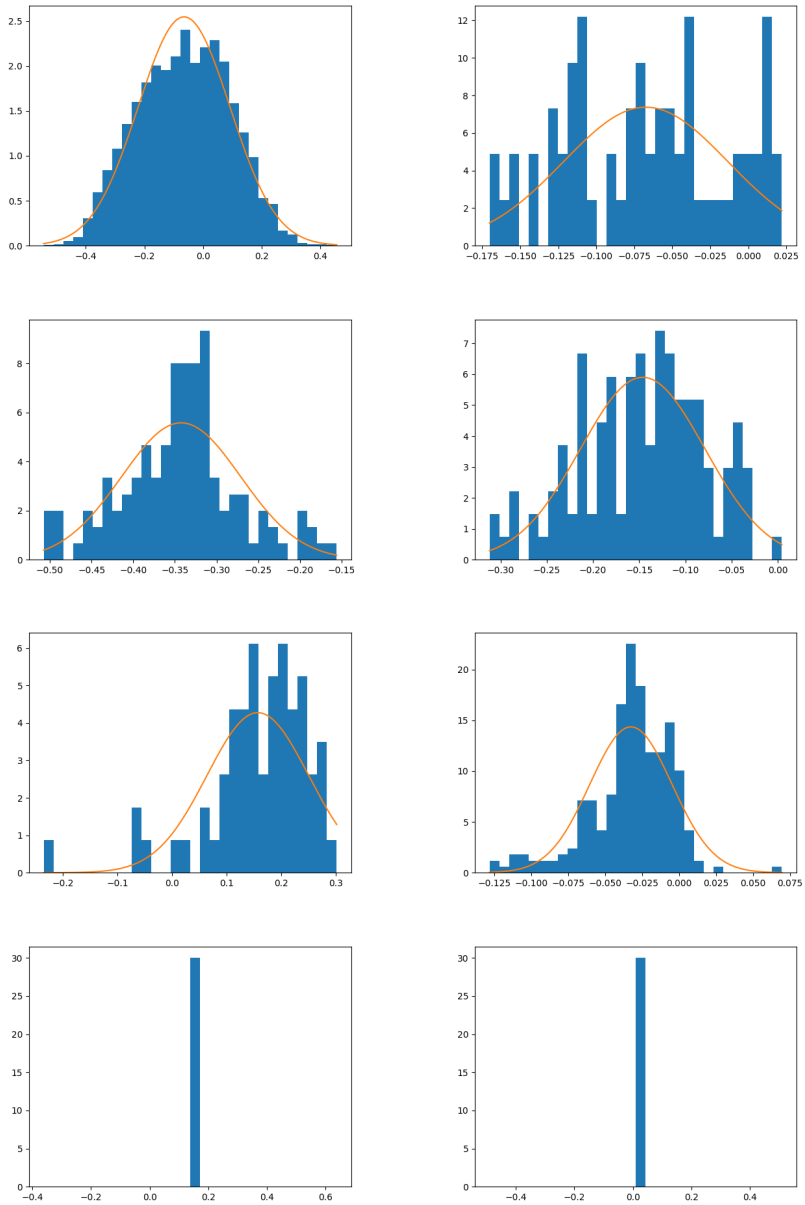


Figure 26: Distributions of the bias vectors of the generator (left) and the discriminator (right). In row i the distribution of layer i is shown. The histograms were generated with 30 bins. In each plot the orange curve is the density of a normal distribution with mean and standard deviation given by the sample mean and sample standard deviation of the respective weights.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

A Study of Generative Adversarial Networks

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Krach	Florian

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 20.03.2019	Krach

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.