Exploring Generative Adversarial Networks

Amund Vedal amund@kth.se
Christos Matsoukas matsou@kth.se
Wojciech Kryściński wkr@kth.se

DD2424 Deep Learning in Data Science Project

Abstract. Generative Adversarial Networks (GANs) were recently introduced in the field of Machine Learning and quickly gained traction within the research community. The ideas behind this technique are intriguing since they propose a completely new approach to training models. In this work we explore their domain by implementing and comparing the results of very recent works in an empirical manner.

1 Introduction

Generative Adversarial Nets (GANs) are a relatively new class of methods for learning generative models that was proposed by Ian Goodfellow et al. in 2014 [14]. In the adversarial training framework, a generative model G is placed against a discriminative model D. The role of D is to distinguish whether the data presented at input comes from the true data distribution p_{data} or was it sampled from the distribution p_{model} modeled by G, while the goal of G is to generate samples that closely resemble those from the training set i.e. learning the underlying true distribution p_{data} of the data. An intuitive way of seeing this framework was proposed by the original author: the model G can be thought of as a counterfeiter trying to produce fake currency. In this situation, model D should be pictured as the police, trying to detect the fake money in circulation [13].

GANs can be analyzed from the game-theoretical perspective, where the training procedure maps to finding a Nash equilibrium of a non-convex game with continuous, high dimensional parameters [13, 15]. GANs are typically trained using gradient descent techniques that are designed to find a low value of a cost function, rather than to find the Nash equilibrium of a game. So far, both D and G were implemented as some type of neural network, but this is not a requirement. The framework is general and should be applicable to different kinds of models and optimization techniques [13, 1].

The trick with the formulation of GANs is that the models are trying to learn the underlying training data distribution by minimizing the difference between the $p_{model}(X;\theta)$ and $p_{data}(X)$. Generally, the loss function that measures this difference has a simple form such as the Euclidean distance, but can also

be more sophisticated [12]. GANs use the discriminator to fabricate this loss function. In other words, D gives us the necessary information to compute the maximum likelihood gradient of the G by discerning whether a given data point x was sampled from the training (true) data or sampled from the generator's distribution (fake data). This allows an automatic learning of a sophisticated loss function [12].

1.1 Theoretical background

In many applications, when trying to fit a probability distribution to a set of observations X that follow a distribution $p_{data}(X)$, a parametric family of probability distributions indexed by a parameter vector θ in which the $p_{model}(X;\theta) := p_z(z)$ can approximate the reference distribution p_g by using a statistical estimator to find the best parameters θ [19]. The most common estimator is the Maximum Likelihood Estimation (MLE) which tries to maximize $p(X \mid \theta)$.

A GAN is a statistical estimator which approaches the subject of the underlying distribution estimation differently by considering a two-player minimax game with a game value function [14].

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

The learning is formulated in two steps: first, a discriminator is trained to minimize is loss function:

$$J^{(D)}(\theta^{(D)}, \theta^{(G)}) = -\frac{1}{2}\mathbb{E}_{X\sim p_{data}(X)}[\log D(X)] - \frac{1}{2}\mathbb{E}_{z\sim p_z(z)}[1 - \log D(G(z))] \quad (2)$$

which is the standard cross-entropy loss that is minimized when training a binary classifier with a sigmoid output [13]. Then, in the simplest case of a zero-sum game, the generator is trained to maximize:

$$J^{(G)} = -J^{(D)} \quad (3)$$

Thus, in this game, the generator minimizes the log-probability of the discriminator being correct [13]. The authors of the original paper [14] show that when G is fixed, the optimal discriminator is:

$$D^*_G(x) = \frac{p_{data}(X)}{p_{data}(X) + p_g(X)} \quad (4)$$

and that $J((D^*_G(x)), g_\theta) = 2JSD(\mathbb{P}_r||\mathbb{P}_g) - log4$, so minimizing eq.2 as a function of θ equals minimizing the Jensen-Shannon divergence when the discriminator is optimal.
Thus, one would expect that we should train iteratively, a discriminator as close as possible to optimality and then do gradient steps on θ. However, this is computationally prohibitive and results in overfitting [14]. The original GAN paper argued that this issue arose from saturation, and suggested a non-saturating game

with cost function, where the generator instead maximizes the log-probability of the discriminator being mistaken [13]:

$$J^{(G)} = -\frac{1}{2}\mathbb{E}_{z\sim p_z(z)}[\log D(G(z))] \tag{5}$$

As mentioned above, the training of GANs means finding the Nash equilibrium, which in this case means finding a global saddle point that maximizes $J^{(G)}$ and minimizes $J^{(D)}$ simultaneously [14, 13]. If these functions are convex, it can be proven that the GAN will converge to that point. Unfortunately, however, the objective functions usually are high dimensional and non-convex thus, the convergence of the p_{model} to p_{data} is not guaranteed, but depends on the way we compute the distance (divergence) between distributions [1]. In GANs, the objective value for the discriminator always has the same form, but different methods have been proposed to compute the difference between the distributions e.g.[1]. The main difference is how the generator is trained or, equivalently, which divergence is minimized.

2 Related work

2.1 Other generative models

The promising results have made GANs the exemplary model of generative modeling, but it is also worth mentioning other generative models. As shown in [2] if we only look at the models that work according to the Maximum Likelihood principle, GANs have several competitors. These generative models were grouped based on their characteristics. The main feature separating generative models is how they represent the probability density function, which yields two groups: One consists of models that explicity define the underlying density function, and one that relies on an implicit definition. Explicitly defined densities can be tractable or non-tractable. Examples of models with tractable density functions are Fully Visible Belief Networks (responsible for the success of WaveNet [9]), non-linear ICAs and PixelRNNs. Compared to GANs, optimizing the model parameters is straightforward here, but sampling is computationally demanding (FVBN), or it is hard to create a tractable model under narrow restrictions, as with non-linear ICA. Models with non-tractable density functions, such as Variational Auto-Encoders (based on Variational Inference techniques) and Boltzmann Machines (utilizing Markov Chain sampling) rely on approximation. Deterministic approximation enables a variety of possible priors and posteriors to be defined to fit existing data, with the drawback of producing (empirically) less pleasing results and being harder to optimize. Stochastic approximation also has its strengths and weaknesses. On the positive side, it produces samples very quickly. On the negative side, it imposes a trade-off between good scalability (to higher dimensions) and computational cost. It is also difficult to determine whether a stochastic approximation model converged or not.
We divide models with implicit densities between the ones using Markov Chains

to generate samples (computationally relatively slow), such as Generative Stochastic Networks, and the other where one samples directly from the implicit density. GANs fall into the last group. They were designed to improve on the shortcomings of other generative models. Unfortunately they also introduce some new problems of their own. They can also be combined with other generative models, as has been shown to yield promising results [6].

2.2 Extensions of GANs

Even though the adversarial training framework was introduced less than three years ago, a large number of models and algorithms extending the original idea have been proposed. A notable work was done by Radford et al. [11], who replaced the generators MLP-based architecture with a deconvolutional network - a change bringing a great improvement to the quality of generated samples. The authors also observed that it is possible to do latent-space vector arithmetic in a similar fashion to that known from word embeddings.

Further attempts of improving the empirical quality of output have also been made through changing the cost function based on divergence between p_{model} and p_{data}. According to [2], maximizing $p_{model}(X)$, and thus minimizing the KL divergence $KL(p_{data}||p_{model})$ is not necessarily the best metric for assessing the quality of generated samples. For this reason, other divergences, such as the JS divergence (mentioned earlier) and very recently the Wasserstein divergence have been suggested [1][3] to increase the (empirical) quality of outputs.

More notable extensions were contributed in [15]. The authors propose various ways of stabilizing the learning process of GAN models, with techniques such as feature matching, minibatch discrimination, one-sided label smoothing, etc. They also noticed that using GANs for semi-supervised learning generated results close to the state-of-the-art, using only a small percentage of the data required by fully supervised models. The original paper and previously mentioned extensions used GANs to generate visual images, but this is not a requirement. Research shows that GANs can also be applied to Multi-Agent systems [4], Speech Enhancement [10] and many other problems. The popularity of the GAN-approach has lead to many other successful attempts of applying it to new problems, often resulting in curiously-named hybrid systems [16].

3 Approach

To explore the GAN landscape, we started from the basic GAN architecture proposed by [14], and then progressed to newer and more advanced models. We trained all models from scratch until we obtained satisfying results, which usually took between 10 and 15 epochs. Instead of striving for excellent image quality and training for longer times, we prioritized many experiments with different parameters and architectures, to learn about and analyze their effect

on the stability of training and the quality of generated images. For the sake of learning, we also implemented our own models for the basic GAN and DCGAN architecture using the Keras library. Experiments on Wasserstein GAN, Unrolled GAN and Conditional GAN were conducted using publicly available code.

Datasets For our experiments we chose two different datasets: MNIST and Labeled Faces in the Wild. MNIST, consisting of 70.000 grey-scale images that depict hand-written digits. Labeled Faces in the Wild (LFW) containing around 13.000 colored images of faces of almost 5800 different people. The LFW dataset comes in two variations, *cropped*, with images cropped around the head of the person, and *full* where the person is in the center of the image, with the head and neck visible. We did experiments on both.

Preprocessing We chose not to attempt any particularly advanced feature scaling or manipulations, but stayed with simple standardization, which minimizes the influence of different lighting conditions [18].

Optimizers As updating the neural network parameters require an optimization method, we chose to rely on two well-known methods: stochastic gradient descent (SGD) with momentum and ADAptive Moment estimation (Adam). SGD w/momentum, the standard for neural networks, optimizes the parameters of the model by following the gradient downward. It has some hyperparameters such as learning rate and momentum. Adam [5] is an extension with exponentially decaying momentum-like terms, one based on the gradient and one on the square gradient. Some authors [2, 15] have suggested using Adam for GAN-training specifically.

3.1 Models

Basic GAN The basic GAN introduced in [14] was built using two feed-forward, fully-connected neural networks. In our implementation, the networks were 3 layers deep, and in some experiments the D had an additional layer to make it more powerful. The Generator had an architectures where the number of hidden units in consecutive layers was growing, the Discriminator had the opposite. The hidden layers used leaky ReLU activation, additionally the discriminator used Dropout to prevent it from overfitting.

DCGAN Since recent papers [1, 3] use it as a baseline, a natural step further were to replace the fully-connected layers of the basic GAN with convolutional layers. We based our experiments on a simplified version of the model structure proposed in [11], with two convolutional layers per network (not deep, as the D in DCGAN suggests, but we chose to stick with the naming convention of [11]). In G, the spatial dimensions (x,y) increase with the depth of the network, while the amount of feature maps (resulting from fewer kernels) decrease. For

D, the evolution of output data after each layer is opposite, with smaller (x, y)-dimensions and larger z-dimensionality towards the end. Again, the hidden layers of the network used leaky ReLU activations, and some general neural network "tricks" that stabilized the models.

4 Experiments & Results

4.1 Basic GAN

The basic GAN architecture is known to be extremely unstable during training. The first experiments conducted on this architecture aimed at finding parameters that would ensure relatively stable learning. All experiments described in this section were conducted on the entire preprocessed MNIST dataset. The models were trained on batches of size 128, which yielded 546 batches per epoch.

In this step we explored general techniques used for training neural networks, such as testing different optimizers and network hyperparameters. We also examined two network topologies to verify how they influence the learning process. One architecture was "wider" with a maximum of 512 units in the hidden layer, the other was "narrower" with 128 units in the hidden layer and an extra layer of depth. We followed the suggestion given in [2] and trained D and G in an equal manner, but made the D a bit more powerful than G, by using Dropout layers.

As our first optimizer we chose SGD with Momentum. We performed a coarse-grained grid search for hyperparameters of the model such as learning rate η, momentum rate α, "leakiness" of ReLU α, and the probability of dropout p. We also tested different sizes of the latent space $\mathcal{Z}$ that the generator samples and different activation functions of the Generator, *tanh* and *linear*. The latent space was modeled by a n-dimensional Gaussian with 0 mean and unit variance.

The effects of the grid search were not satisfactory. For most parameters settings the losses of the Generator and Discriminator quickly diverged, yielding a wide gap of loss between a weak G and a highly specialized D. The generated images were very noisy. The generator returned nearly the same results from different noise samples – an indicator of the so-called "mode collapse" problem of GANs. In some cases the mode was a number that could be distinguished, in other, it was a combination-"blob" of different digits, as in the "cherry-picked" result in figure 1.

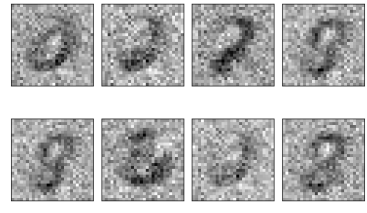
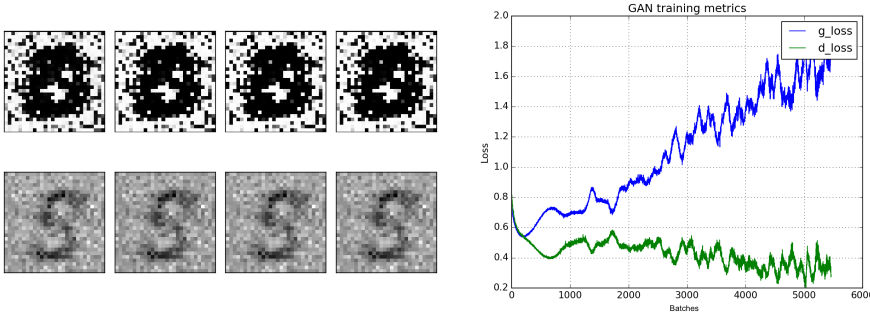


Fig. 1: Best generated image after training with SGD

These initial experiments allowed us to conclude the following. Learning rates of magnitude `1e-3` and lower seemed to yield a more stable training process, but it also took a longer time for G to start outputting any recognizable images (as expected). Momentum seemed to have very little effect on the performance of the GAN. The loss fluctuations between consecutive training steps seemed to be smaller, as expected, but had no overall stabilizing effect. Adding Dropout to the D seems to have a stabilizing effect on the training process. The magnitude of the losses of both G and D is much lower than without Dropout. Using *tanh* activation function as output of the Generator yields sharper images sharper when compared to the *linear* output, but also yielded more "blob" modes. The choice of the function has no effect on the stability of the training. The "leakiness" of the ReLU didn't bring any changes to the stability of the training, nor the quality of generated images. The size of the latent space proved to have little significance, although it seems that a lower dimensional $\mathcal{Z}$ space (in the magnitude of `1e2`) made it easier for G to produce reasonable images. The choice of network didn't seem to matter, and the conclusions described above are valid for both architectures.



(a) Generated images - upper row G with *tanh* out., lower row G with *linear* out.

(b) Loss function of D and G. Number of batches roughly equal to 10 epochs

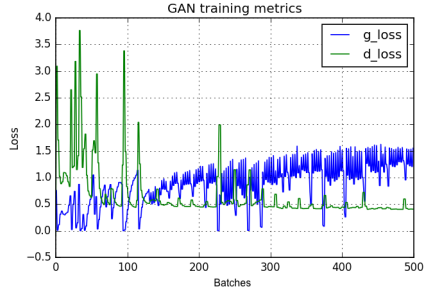
Fig. 2: Example results obtained with the basic GAN trained using SGD

The next step was to implement some of the improvements listed in the work published by OpenAI [15] and Chintala et al. [17]. We attempted one-sided label smoothing, where only the `true` labels were swapped with a random number in the range $[0.8, 1.1]$, and two-sided smoothing where also the `false` labels were swapped with a number from the range $[0.0, 0.3]$. Different shapes of the Gaussian and Uniform distributions were tested to model the latent space. The effects of pre-training either the Discriminator or the Generator before the main training phase were explored. Most off the time was dedicated to experimenting with training D and G in an unequal manner in hope to find a way to balance-out the model.

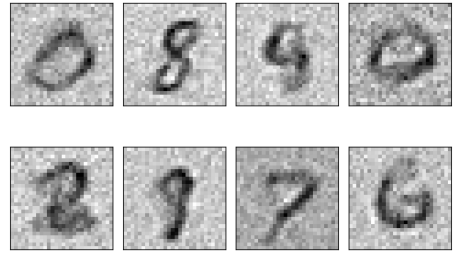
Unfortunately, most of the described experiments brought little to no improvements of stabilizing the model. The exception was sampling the input noise

from a Gaussian distribution. Training one model more than the other, even for very short intervals, made it hard to balance-out the strengths of models at the main learning phase. Even though we were able to find ways of closing the gap between the loss functions by training models, this did not bring any visible improvement to the generated images, which was quite unexpected. However, comparing the plots of the loss-functions gave us further insights into where "mode changes" happened. During training, the generator loss-function seemed to "collapse" periodically, which we believe were the mode changes we observed, so the loss of G goes down and D goes up. Since the G switched the mode for a few iterations it was able to "trick" the D with fake images. Usually the images generated at this peak have the highest quality. This effect is presented in figure 3a.

We also tried changing optimizer to Adam. Again, we performed a coarse-grained grid search, but incorporated some of the insights from previous experiments in the choice of parameters to tune. The parameters tested were learning rate η, decay parameters β_1 and β_2, "leakiness" of ReLU α, and the probability of dropout p. The size of the latent space $\mathcal{Z}$ was kept constant at 100 dimensions. Again *tanh* and *linear* were tested as the output of the Generator, but the architecture was kept constant. We chose the architecture from the SGD experiments with "wider" hidden layers and extended the Discriminator with one additional layer, to make it more powerful.



(a) Complementary behavior of D and G losses, peaks indicate mode switches.



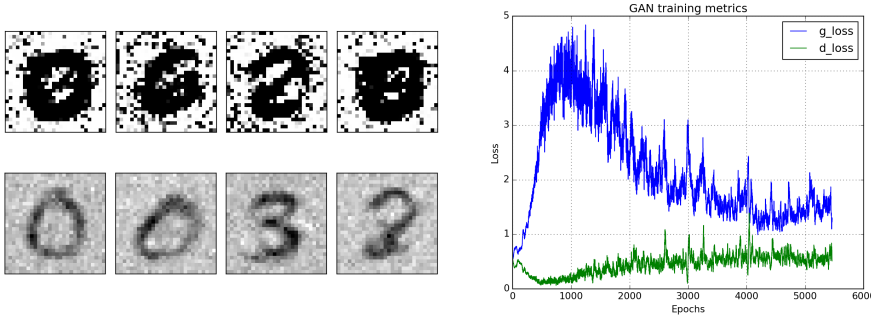
(b) Best generated image after training with Adam

Fig. 3: Experiments with basic GAN

The effects of changing the optimizer were visible instantly. For most of the parameter values, the network was able to converge without the "mode collapse", which was nearly impossible in the case of SGD. The training seemed to be more stable with both D and G models balanced. The gap between the losses was smaller and in general we observed a "converging" trend. Generated images had a noisy background but the digits were clearly visible. "Cherry-picked" generated images are shown in Figure 3b. The empirical quality of generated images seemed to improve, but the biggest difference between using Adam and SGD was the

aforementioned stability of learning.

From these experiments we conclude the following. A learning rate in the magnitude of `1e-3` gives the best quality-time ratio. Values of decay parameters of the Adam optimizer have the biggest influence on the stability of the training process and quality of the generated images. The default settings were β_1 and β_2 are between 0.9 and 0.999 seems to block the model from learning. When the loss functions stayed constant with a large gap between them, the resulting images were blobs of darker pixels in the center of the image. We found that for the mentioned parameters, values around 0.5 gave the best performance and learning stability. This observation is consistent with tips given by Radford in [11]. The results when varying other parameters, such as "leakiness" of ReLU, the type of output from G and dropout probability were consistent with those obtained using the SGD optimizer.



(a) Generated images - upper row G with *tanh* out., lower row G with *linear* out.

(b) Loss function of D and G. Number of batches roughly equal to 10 epochs

Fig. 4: Example results obtained with the basic GAN trained using Adam

4.2 DCGAN

Building and testing our DCGAN-implementation, we realized that our search for good parameters had to be limited compared to the simple GAN due to computational constraints (Convnets take longer to train). For this reason, we mainly followed the advice of [11], using similar architecture, ReLU and leaky ReLU activation functions and the Adam optimizer with similar parameters. The first experiments with this model yielded disappointing blob-images and loss plots with a diverging G-loss, see Figure 5, even when using the Adam optimizer. The results were less noisy than from the basic GAN. We struggled to find a better setup since we assumed that the suggestions from Radford et al. were thoroughly tested.

What eventually lead to a big improvement in empirical quality (subjectively better results) was changing the activation of the generator output layer from

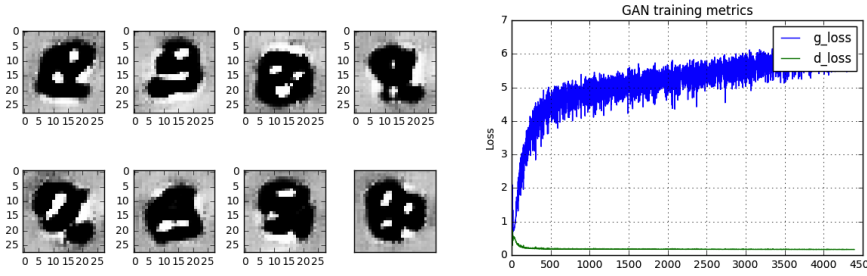


Fig. 5: Shows disappointing results using *tanh* activations in final generator layer.

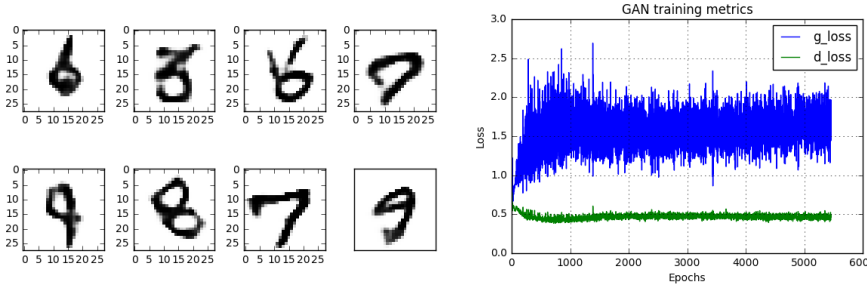


Fig. 6: Shows surprising change after changing generator activation to *sigmoid*

tanh to *sigmoid* or *linear*, see Figure 6. To us, this result seemed counter-intuitive and hard to understand, so we tried adding batch normalization (again, as suggested by Radford et al.) and dropout, but neither of them lead to significant improvement when using *tanh*. This lead us to believe *tanh* wasn't well-suited for the MNIST-dataset, with its comparatively simple features.

To check if the performance of our DCGAN was consistent with more feature-rich datasets, we continued with experiments on the LFW dataset. These were split into stages of growing complexity. The first step was to the train a DCGAN on gray-scale cropped (to contain only faces) images in LFW. We mainly explored the benefits of using Batch Normalization layers, with generated images are shown in the left side of Figure 7. We found that adding Batch Norm significantly improves the quality of generated images. The faces seemed to be sharper, with more distinguishable key-points such as eyes, eyebrows, nose, cheeks and mouth. In contrast, the images were more blurred without Batch Norm. It was interesting to see that the network was able to learn how to generate valid (not-disfigured) human faces, with the majority having key-points in correct relative position to each other - eyebrows above eyes, nose between eyes and mouth below nose. We also conducted similar using color-images, shown in the right side of Figure 7. In this case, the network not only learned to position the face key-points correctly, but also to apply impressive, realistic coloring: redish lips and

Fig. 7: Example results obtained with the DCGAN on cropped gray-scale and color LFW dataset. Upper row without Batch Normalization, lower row with Batch Normalization

darker eyes and eye brows, usually standing out from surroundings.

Training the model to generate full images containing not only the faces but also parts of the neck and a background, added another layer of complexity to the task. We show some samples in Figure 8. Even though the faces are smaller than in the previous case, the network was still able to reproduce them with great detail. We also observed how the GAN was struggling to produce a realistic background for the images. In most cases, the background is either very dark or a mosaic of blurred colors. We assume this comes from the high variance of backgrounds in the LFW. We were surprised at how little training was required for the networks to produce such high-quality results.

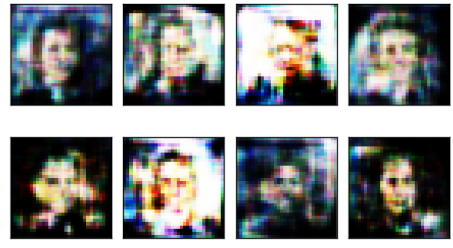


Fig. 8: Example results with the DCGAN trained on full LFW images

Using the models trained on LFW, we also decided to do some additional exploration of the latent space. As in [11], we chose two faces as start- and endpoints in the latent space, and generated 6 equally spaced points lying on the line between them. We fed those "seed" points to the generator and plotted the images in the same order as they were generated. The images are shown in Figure 9 and show a smooth transition between the two endpoints, as if the face of the starting image is gradually transitioning into the other. This can be understood as a sense of position and direction in high-dimensional space, giving a concrete "meaning" to the abstract latent space in which the generated images "live".

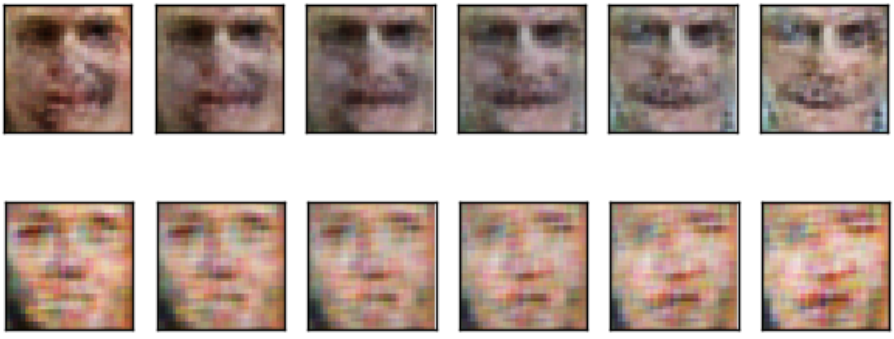


Fig. 9: Example results obtained with the DCGAN by exploring the latent space. The left and right images were the start and end points.

Explorations of more advanced models

In order to explore more advanced models we slightly modify some already implemented algorithms from github* due to limited time. Nevertheless, the point was to witness the differences in the more advanced models as the authors illustrate in their work e.g.[7, 1, 8].

Conditional GAN Our first test was dealing with Conditional GANs [7]. In this modification, Mirza et al. (2014) suggested that both the generator and the discriminator should condition on some extra (prior) information. In this case, the auxiliary information is the labels of the dataset, and therefore the model is conditioning on the labeling which is incorporated as an additional input layer. With this modification of the original GAN, we can explicitly pick the label of sample that we want to create.

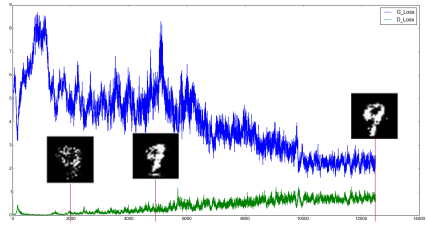


Fig. 10: Loss evolution when training with conditional GANs and a sample of generated numbers on the corresponding iteration

We tried both with MLP† networks and the DC‡ version. In the simple case of the MLP network we found similar results with the our implementation of

* https://github.com/zhangqianhui/Conditional-Gans, https://github.com/carpedm20/DCGAN-tensorflow https://github.com/carpedm20/DCGAN-tensorflow, https://github.com/wiseodd/generative-models https://github.com/musyoku/unrolled-gan, https://github.com/musyoku/wasserstein-gan

† https://github.com/wiseodd/generative-models

‡ https://github.com/zhangqianhui/Conditional-Gans

the simple GAN model (see Figure 10). The Adam optimizer gave better results than the SGD as expected and this is used for the reported results. In general, the only difference is that now we can choose the outcome as we condition on the labels.

In the case of conditional DCGANS [7, 11] the learning is more stabile in terms of learning evolution fluctuations (see Figure 11) and the mode collapse problem is reduced. In the reported results the Adam optimizer was used. For the discriminator we used three convolutional layers with batchnorm and LeakyReLU activations for all layers. G had approximately the same architecture but now with ReLU activation for all layers except for the output, which uses *tanh* as the authors of DCGAN suggested [11]. This explains the stability of the loss evolution in learning. Note that the performance that we manage to get was not due to the conditional formulation of the problem but in the fine tuning of the network parameters. As we inferred before, the noticeable result is in the picking of the output.

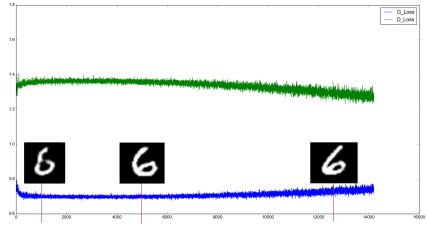


Fig. 11: Loss evolution when training with conditional DCGANs and a sample of generated numbers on the corresponding iteration

Unrolled GAN An other modification of the GANs is proposed by Metz et al. [8] seems to solve the mode collapse issue. In this case, the learning of D is unrolled in K steps, where the problem is equivalent to the basic GAN if $K = 0$ [8, 2]. In our experiments we found that when using 10 unrolling steps the results are really good and without mode collapse (see figure 12). Here we report the results for three-layer MLP with batch norm. We noticed that in our experiments the variation of the loss function was big but the average direction was smoothed enough. This is due to the use of MLP instead of convolutions, in which we expect more stable loss fluctuations. Finally, we infer that when training on numbers of the MNIST dataset, the generated data was starting to get realistic forms in the first epochs. Although, we point out that this may be due to the differences in the network architecture and the tuning of the learning parameters.

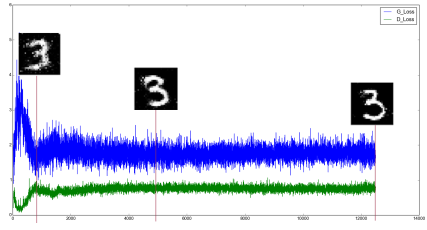
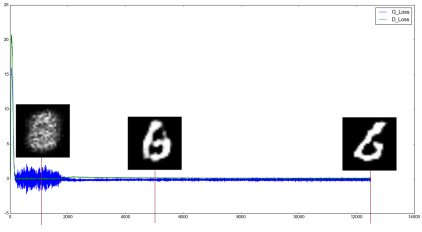


Fig. 12: Loss evolution when training with conditional GANs and a sample of generated numbers on the corresponding iteration

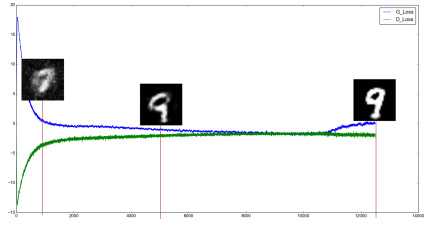
Wasserstein GAN Probably the most promising approach we found to stabilize learning in GANs was made by Arjovsky et al. [1](2017). The authors

suggested that changing the way to measure the difference between two distributions also changes the convergence of the algorithm. They propose minimizing the Wasserstein divergence instead of the aforementined Jenson-Shannon, resulting in better performance mainly because the discriminator is optimized in a better point. They also suggested weight clipping to guarantee Lipschitz continuity. The Wasserstein-GAN requires optimizers that do not use momentum and thus the authors suggested using RMSprop to avoid instabilities [1]. We run tests using this model ⋆ and the updated version [3] in which a weight penalty is present in a weight decay form⋆⋆.

In both WGAN with clipping of weights and gradient penalty (see Figure below) we used MLP architecture with only one hidden layer and an RMSprop optimizer. Here we observed how the algorithm indeed provided a more stable learning, especially in the updated WGAN. The algorithm not only stated to created realistic results from the first iterations but also we noticed an significant improvement on the quality of the results.



Loss evolution when training with Wasserstein GANs and a sample of generated numbers on the corresponding iteration



Loss evolution when training with the improved Wasserstein GANs and a sample of generated numbers on the corresponding iteration

5 Conclusions

From what we've seen in our experiments and explorations, we conclude that implementations should be based on the most recent advances rather than starting experiments with simple GANs, particularly for image generation, since problems such as learning instabilities already have "tailored" solutions. We also strongly recommend studying recent papers before refining GAN implementations by hand, since optimizing model parameters of complex models requires a lot of computational resources.

⋆ https://github.com/wiseodd/generative-models, https://github.com/musyoku/wasserstein-gan

⋆⋆ https://github.com/wiseodd/generative-models

References

1. Arjovsky, M., Chintala, S., Bottou, L.: Wasserstein GAN. CoRR **abs/1701.07875** (2017)
2. Goodfellow, I.J.: NIPS 2016 tutorial: Generative adversarial networks. CoRR **abs/1701.00160** (2017)
3. Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., Courville, A.C.: Improved training of wasserstein gans. CoRR **abs/1704.00028** (2017)
4. Ghosh, A., Kulharia, V., Namboodiri, V.P.: Message passing multi-agent gans. CoRR **abs/1612.01294** (2016)
5. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. CoRR **abs/1412.6980** (2014)
6. Larsen, A.B.L., Sønderby, S.K., Winther, O.: Autoencoding beyond pixels using a learned similarity metric. CoRR **abs/1512.09300** (2015)
7. Mirza, M., Osindero, S.: Conditional generative adversarial nets. CoRR **abs/1411.1784** (2014)
8. Metz, L., Poole, B., Pfau, D., Sohl-Dickstein, J.: Unrolled generative adversarial networks. CoRR **abs/1611.02163** (2016)
9. van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., Kavukcuoglu, K.: WaveNet: A Generative Model for Raw Audio. ArXiv e-prints **abs/1609.03499** (2016)
10. Pascual, S., Bonafonte, A., Serrà, J.: SEGAN: speech enhancement generative adversarial network. CoRR **abs/1703.09452** (2017)
11. Radford, A., Metz, L., Chintala, S.: Unsupervised representation learning with deep convolutional generative adversarial networks. CoRR **abs/1511.06434** (2015)
12. Schawinski, K., Zhang, C., Zhang, H., Fowler, L., Santhanam, G.K.: Generative adversarial networks recover features in astrophysical images of galaxies beyond the deconvolution limit. arXiv preprint arXiv:1702.00403 (2017)
13. Goodfellow, I.J., Bengio, Y., Courville, A.C.: Deep Learning. Adaptive computation and machine learning. MIT Press (2016)
14. Goodfellow, I.J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A.C., Bengio, Y.: Generative adversarial nets. In: Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada. (2014) 2672–2680
15. Salimans, T., Goodfellow, I.J., Zaremba, W., Cheung, V., Radford, A., Chen, X.: Improved techniques for training gans. In: Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain. (2016) 2226–2234
16. Hindupur, A.: Gan zoo. https://github.com/hindupuravinash/the-gan-zoo
17. Chintala, S., Denton, E., Arjovsky, M., Mathieu, M.: Gan hacks. https://github.com/soumith/ganhacks
18. Ng, A.: Stanford ufldl tutorial data preprocessing. http://ufldl.stanford.edu/wiki/index.php/Data_Preprocessing
19. Goodfellow, I.J.: On distinguishability criteria for estimating generative models. arXiv preprint arXiv:1412.6515 (2014)

A Appendix

Illustration of mode collapse solution though a simple example Metz et al. (2016) [8] in order to illustrate the mode collapse issue they proposed a toy example. They trained a simple GAN architecture on a 2D mixture of 8 Gaussians arranged in a circle, see figure 14 to show how the probability mass fails to sped in all modes in the mode collapse case. We run these experiment to witness by ourselves the outcomes (see figure 15). Thus we infer, that this toy example summarizes the aforementioned results regarding the mode collapse problem in when using different models in nice illustrative and pedagogical way. The mode collapse in the case of the simple GAN is obvious. Furthermore, in the case of the unrolled GANs we can see that the mode collapse is almost absent but the modes are converging in an area near the true ones. This results the somehow unrealistic results that are reported when using unrolled GANs. Finally, in the case on Wasserstein GAN the mode collapse is absent and the mass probability is equally distributed.

Fig. 14: Plot of input data, 2D Gaussian distributions

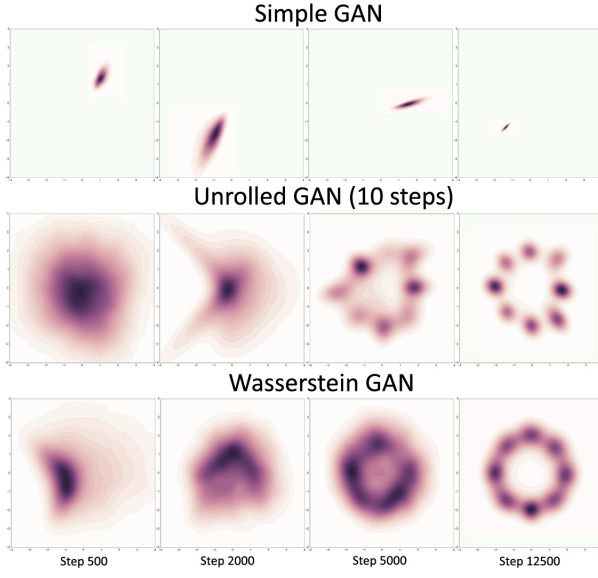


Fig. 15: Toy Example comparing the different advanced GAN's performance on 2D Gaussian distributions