GAN: Improving the Generator Architecture in Image Generation from Noise

Julius Laitala

Helsinki Thursday 2nd April, 2020

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Julius Laitala

Työn nimi — Arbetets titel — Title

GAN: Improving the Generator Architecture in Image Generation from Noise

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Seminar report	Thursday 2nd April, 2020	21 pages

Tiivistelmä — Referat — Abstract

This seminar report takes a look at architectural modifications that can improve the performance of a generative adversarial network's generator. We will focus on image generation from noise input.

First, we'll have a very quick refresher on generative adversarial networks and briefly review some problems that some of the more traditional GAN generators have.

Then, we will take a close look at two potential, generator-architecture-based solutions: self-attention generative adversarial networks and architecture changes inspired by style transfer literature We'll review the architectural changes introduced by each network.

In the end we'll put the three proposed generator improvements to the test. We'll build somewhat simplified versions of the inspected generator architectures, train them with a dataset of fictional creatures, and compare the results to results gained with a DCGAN-based generator.

Avainsanat — Nyckelord — Keywords
Generative adversarial network, Image generation, Generator

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

Since their introduction in 2014 [2], *generative adversarial networks* (*GANs*) have become the state-of-the-art of image generation. A generative adversarial network consists of two neural networks: a *generator*, which generates data mimicking the target distribution, and a *discriminator*, the task of which is to distinguish between real data from the target dataset and the generator's results. In this report, we will look into the generator part of this network.

We'll start the report with a very brief review of GANs and a look into DCGAN, a GAN architecture that we'll consider our baseline when inspecting some more advanced architectures, in section 2. We'll also limit ourselves to inspecting generators that take noise vectors as their input.

Regardless of their already very impressive performance, GANs still have a long way to go. As we will discuss in section 3, the generators are plagued by a multitude of problems that hamper the quality of the generated data. We will take a brief look to a few of these problems – our aim is not, however, to produce a comprehensive review of GAN issues.

Solutions to these problems are rapidly surfacing. We will take a look at two solutions that can be applied to just the generator architecture, without modifying the discriminator or the training process. In section 4, we will look at Zhang et al.'s approach of adding a *self-attention mechanism* to the generator. In section 5, we will review Karras et al.'s *style-based* generator architecture.

After we are familiar with these possible solutions, we will put them to the test. In section 6, we'll use a Pokémon dataset to train a simple version of each of the three generator architectures (DCGAN baseline, self-attention- and style-based generator), using the same discriminator architecture and training process in each of the three cases. We will compare the results and see which generator architecture best fits our strange, niche problem.

2 Generative adversarial networks

As mentioned in the introduction, a generative adversarial network consists of two neural networks: a generator and a discriminator. The way a GAN learns is via a two player game: the discriminator is first trained with some real- and generator generated pictures. Then, the generator is trained on the current state of the discriminator.

In this report, our focus is on generators that generate images based on noise vectors, as this is the simplest and most pure type of GAN generator input. These noise vectors are called *latent codes*. They are thought to be drawn from the *latent space* – a distribution that we can consider the generator to be mapping from to images. We can think that the learning task of our generator is learning a mapping from the latent space into our data distribution [1].

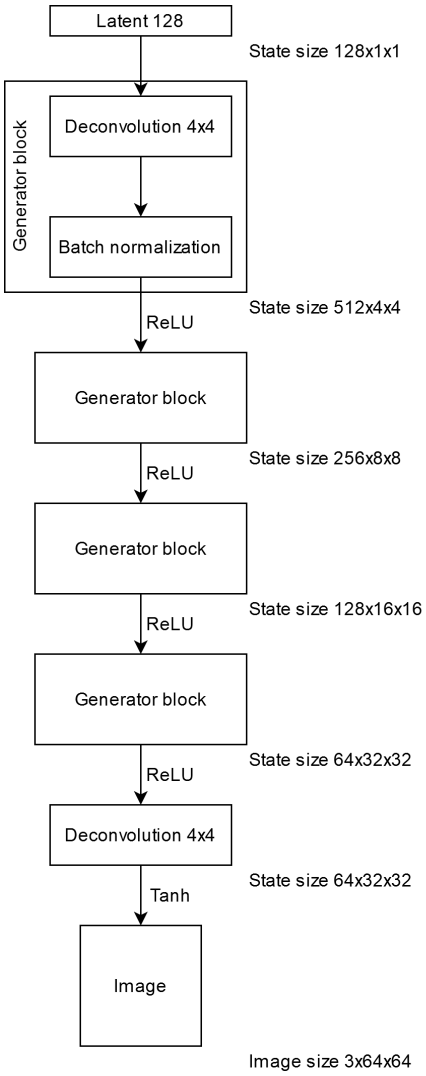

Figure 1: The baseline DCGAN architecture used for Pokémon generation in section 6.

2.1 DCGAN

Early generative adversarial networks were built out of fully connected layers. They were applied to relatively simple image data sets, with results much less impressive than today's state-of-the-art [1].

Then came the DCGAN, the *deep convolutional generative adversarial network* [6]. DCGAN is a GAN that utilizes multiple layers of convolutions and deconvolutions, which allows it to learn spatial down- and upsampling operations during training. This seems to contribute a lot to the quality in image synthesis [1].

As both of the generator architectures that we will take a closer look at build on convolutions, DCGAN can be thought as a good, tried-and-true baseline to compare our newer models against. An example DCGAN generator architecture is shown in

figure 1.

3 Problems with generators

There are a multitude of open issues with GAN generators, exploring all of which is out of the scope of this report. We will instead drill deeper into perhaps the most fundamental problem, the difficulty of training GANs, and after that into the two issues that the generator architectures we will investigate in sections 4 and 5 promise to solve: the difficulty of capturing global relations with convolutions and the poor understanding of the generator.

3.1 Training problems

One problem with generative adversarial networks is that they are notoriously difficult to train [1]. There are multiple problems one might encounter when training a GAN, including that the generator and discriminator might not converge or that the discriminator loss converges quickly to zero leaving no gradient update paths to the generator. Out of these training problems, the most generator specific one is *mode collapse* – the generator starting to generate very similar samples for different inputs.

While neither of the generator architectures we will inspect later is built around solving this issue by the merit of the network structure, both of the papers introducing them contain suggested solutions. The paper for the style-based generator architecture offers a bunch of regularization tricks [3], while the paper that introduces the self-attention mechanism implements *spectral normalization* [8]. Both of these techniques, however, are out of the scope of this report, though we do use spectral normalization in both the style-based generator and the shared discriminator due to it being easy to apply.

3.2 Difficulty with capturing global relations

Traditional convolutional GANs excel at creating images with only a few geometrical constraints: images of, for example, landscapes and bedrooms, with large, textured areas and simple, more-or-less straight boundaries. However, they have much difficulty capturing more complex, long-range geometrical relations: a DCGAN might generate the fur of an animal perfectly, but draw that animals without any legs. This limitation might well be due to the local nature of the convolutions – to capture long-range relations using only convolutions, you'd need an ever increasing number of convolution layers, which might then cause increased fragility [8].

3.3 Poor understanding of the generator

As usual with deep neural networks, the generator in a GAN is often a black box, and the process of turning the random, latent code into an image is extremely hard to decipher. Even though latent space interpolations – varying the latent code along a vector – give some insight into what each element of the latent code does, the functions of the latent code elements are assigned randomly during the training process making it impossible to compare different generators based on them [3].

4 Self-attention generative adversarial network

The *self-attention generative adversarial network* (SAGAN) is designed to solve the problem of global relations. Proposed by Zhang et al. [8], it employs a self-attention mechanism plugged into a standard DCNN.

An example SAGAN architecture can be seen in figure 2.

4.1 Self-attention mechanism

The self-attention GAN employs a non-local mechanism, first proposed by Wang et al. for video classification, object detection and segmentation and pose estimation [7]. In a GAN, this non-local mechanism is termed as an "self-attention mechanism". In a nutshell, it allows a feature to use information from other features that are further away from it than a feasible-sized convolution kernel's width.

The self-attention mechanism is implemented by first mapping the feature maps of the previous layer into feature maps f, g and h using 1 x 1 convolutions. The transpose of the map f and the map g are then multiplied together, and the result is softmaxed. This gives us the attention map, which in turn is applied to the feature maps h via multiplying. Then, finally, the result is passed through another 1 x 1 convolution layer, out of which we get self-attention feature maps – feature maps with attention applied. These operations are illustrated in figure 3.

All these matrix operations end up calculating response at a pixel's position as the weighted sum of features at all positions. This helps the generator to produce related, far-apart features such as the front- and hindlegs of an animal. The weights for these weighted sums – the attention maps – are illustrated for our Pokémon generation case in figure 12.

In order to not have the self attention overwhelm the network at early stages of training, the self attention feature maps are further scaled by a learnable parameter that's initialized as 0 before they are applied by summing to the original input.

The self attention mechanism is quite "plug-and-play"; it can be applied after an activation layer in a DCGAN without any further changes to the architecture.

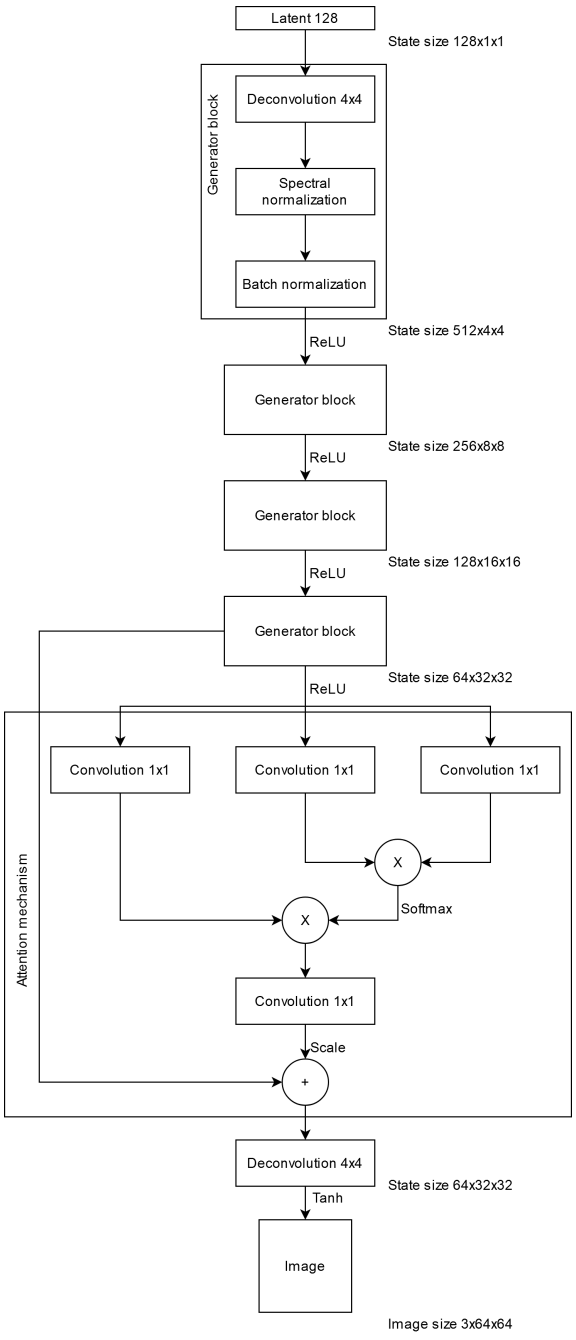

Figure 2: The SAGAN generator architecture used for Pokémon generation in section 6.

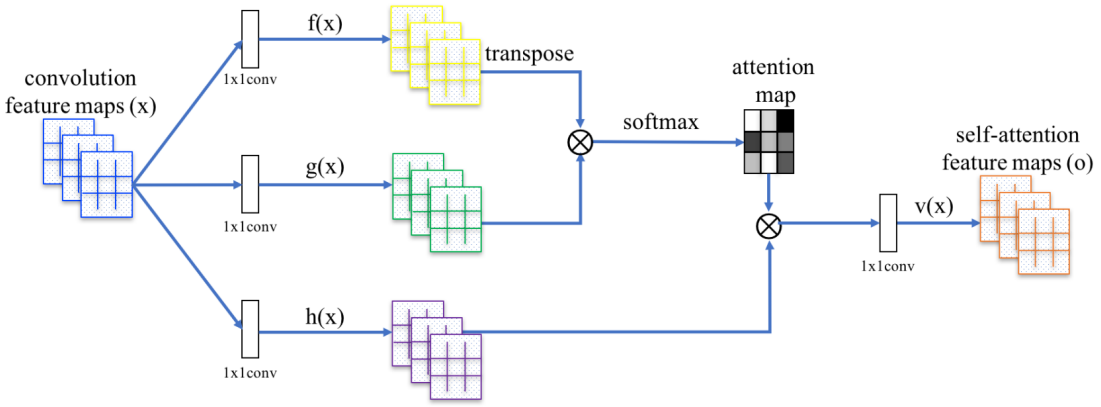

Figure 3: The self-attention mechanism used in SAGAN. The coloured grids denote feature maps, the white boxes are convolution layers, the blue arrows denote the flow of data and $\otimes$ denotes matrix multiplication. Image from Zhang et al. [8].

5 Style-based generator architecture

Style-based generator architecture (StyleGAN), proposed by Karras et al. [3], attempts to make the generator more transparent and to increase control of the generation process. It has taken inspiration from style transfer experiments in other neural network types, and automatically learns a separation between high-level attributes of the data – such as the orientation of a face – and the details, such as freckles. Each level of detail can then be controlled separately via a mechanism of injecting mapped latent information to the network layer by layer.

The style based generator completely revises the classic "stacked" generator architecture, opting instead for a two-network generator consisting of a *mapping network*, that *disentangles* the latent space, and a *synthesis network* that starts from a learned constant, applies both independent noise and disentangled latent information during the middle of the generation, and produces the image. This is elaborated in the simplified example architecture in figure 4.

The original paper promises to be discriminator- and loss function agnostic. However, we can deduce from our training problems in section 6.5 that this might not be the case, and indeed the revised version, StyleGAN2 [4], includes it's own discriminator design with jump connections.

5.1 Mapping network

The first major part of a style-based generator is the mapping network. It is implemented as a multi-layer perceptron, and it maps the latent code into another vector that is further used by the synthesis network to produce "styles" in the synthesis network. The inclusion of the mapping network is viewed as a way to learn a distribution of styles which the synthesis network can then sample from to generate new

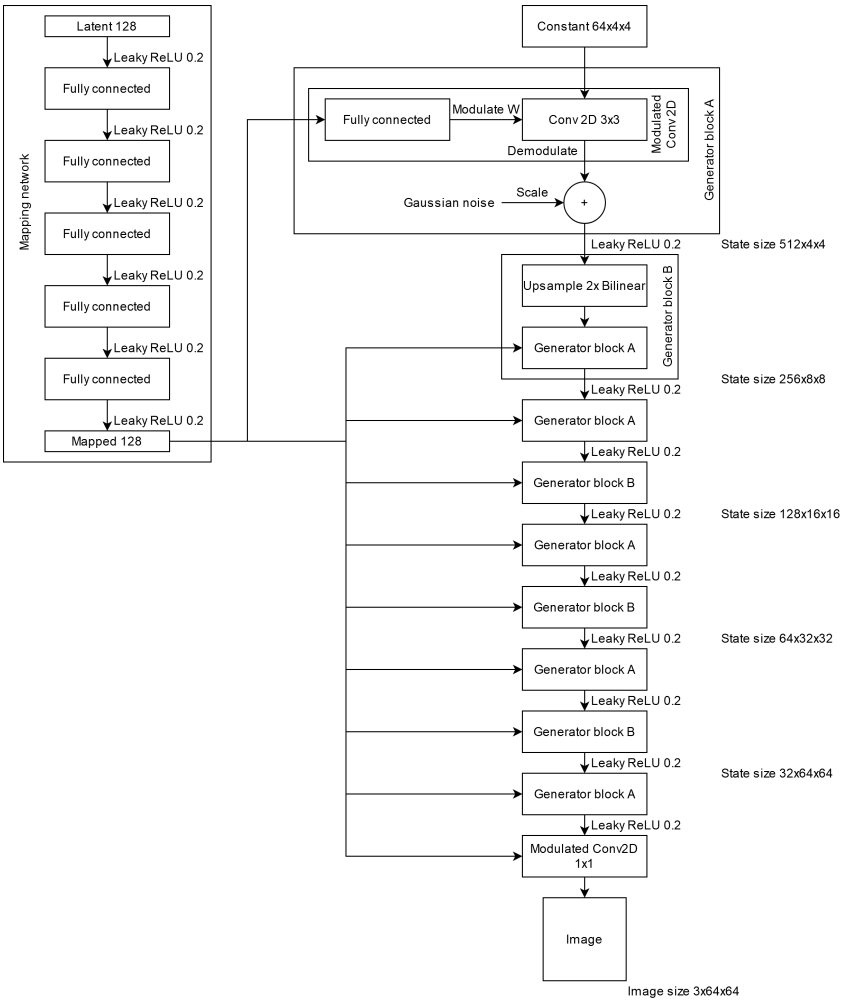

Figure 4: The simplified StyleGAN generator architecture used for Pokémon generation in section 6.

images.

A major advantage of this mapping network is that it "disentangles" the latent space: if the sampling probability of each combination of factors in the latent space doesn't match the densities of variance in the training data (which is often the case with normal-distributed latent codes), the training warps the network weights to compensate for these imbalances. The mapping network essentially learns to warp the latent space into a distribution that matches the real data's distribution.

5.2 Synthesis network

The other major part of a style based generator is the synthesis network. This is the network that produces the actual images.

Unlike the DCGAN introduced in section 2 and the SAGAN introduced in section 4, the synthesis network of StyleGAN doesn't start with a randomly drawn latent vector. Instead, the image synthesis begins with a constant layer, that's learned during the training just as any other parameters. The image is then synthesised by periodically adding Gaussian noise with a learned multiplier, mixing in styles and passing it through 3x3 convolutions. Finally, an 1x1 convolution (again with mixed in styles) transforms the feature maps into an RGB image.

The synthesis network doesn't just use the results of the mapping network as they are. Instead, the mapped latent vector is passed through a fully connected layer – a learned, per-convolution transformation that produces a style based on the sample from the learned distribution.

In the original paper [3], the mixing in of styles happened using an *adaptive instance normalization* operation where each feature map was normalized separately and then scaled and biased using components from the style. However, in the subsequent paper [4] it was found that this operation caused artefacts resembling water droplets. This was hypothesized to happen because of the generator learning to create a strong, localized, statistic-dominating spike, allowing it then to scale the signal as it likes elsewhere. Because of this, the subsequent paper proposed directly scaling and normalizing the convolution weights.

5.3 Style mixing

The fact that the mapped latent code is applied to the synthesis network in multiple stages gives us an interesting new way to control the generator. We can *mix* different styles: use some mapped latent code for earlier layers, while using another mapped latent code for later layers. The good behaviour of the network when doing this is encouraged by constantly mixing styles at various points during training.

Using style mixing, we can identify which parts of the network are responsible for the coarser features, and which parts generate the finer details. This grants us the promised increase in visibility and control over the network – instead of just

interpolating a random element of our latent vector, giving us unpredictable results, we can vary the whole latent code at a level that we know will modify, say, the details or the rough forms of our generated image. This is illustrated in figure 16 for the Pokémon generation experiments.

6 Experiments in generating Pokémon

To test the effectiveness of these alternative generator architectures, we trained three networks to generate Pokémon. This is quite a challenging task: there are less than a thousand unique types of Pokémon, and yet each Pokémon is very different from the others. The sprites we use are clean, flatly coloured and noiseless, which should add a challenge of it's own.

The Pokémon dataset should bring adequate challenges to both generator architectures we've investigated: There are a lot of regularities (wings, legs, eyes) that SAGAN could pick up on, and some clear boundaries of variation (the general shape, details such as spikes and fangs, colours) that the StyleGAN might be able to specialize in.

We'll start by describing our dataset and training process, and then go through the experimented generator architectures one by one, comparing them based on the training process and visual evaluation of the results.

The code used for these experiments can be found at https://github.com/laitalaj/nopemon.

6.1 Data

As our dataset we used renders of all Pokémon found in the Pokémon Sun and Moon -games. We randomly selected 10 frames from the idle animation of each of the 2258 Pokémon (we count the female- and shiny versions as separate Pokémon), bringing the total input images to 22 526 ($< 2258 \cdot 10$, as some Pokémon have less than 10 frames in their idle animation).

We further augmented the dataset by applying random rotations, skews, horizontal flips and color jitter to the images at load time. We also scaled the images to our desired dimensions of 64 x 64 using bilinear resampling before passing them on to the discriminator.

Some examples of the training dataset can be found in figure 5.

6.2 Discriminator and training process

In order to focus only on the advantages brought forth by changes in generator architecture, the discriminator and the training process were kept constant when training all of the three networks.

Figure 5: 64 examples of the input data with random transformations added.

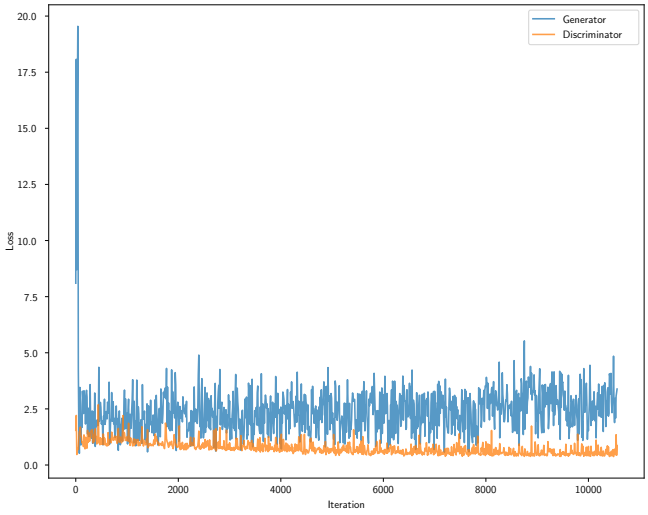

Figure 6: The losses of the DCGAN (baseline) generator and the discriminator as a function of training steps.

We used a DCGAN discriminator with 2 765 568 trainable parameters. We included spectral normalization after each discriminator layer.

For our loss function, we used binary cross entropy loss. We set our optimizer to an Adam optimizer with learning rate 0.0003, $\beta_1 = 0.5$ and $\beta_2 = 0.9$ [5] for both the generator and the discriminator. We trained the networks for 30 epochs, using batch size of 64.

In order to further stabilize the training, we used label smoothing; that is, we set the real label to 0.9 instead of 1.

6.3 Baseline: A "simple" DCGAN

For a baseline, we trained a DCGAN model, the architecture of which is shown in figure 1. This model had 3 806 080 trainable parameters.

There weren't any major problems when training the DCGAN. Towards the end of the training the generator- and discriminator losses started to diverge (figure 6), but this doesn't seem to have affected the generated image quality too much (figure 7).

Our baseline results are not extremely good, but not bad either: a person that doesn't know that the blobs in figure 8 are supposed to be Pokémon probably wouldn't guess that without some help, but with a bit of imagination one can see some Pokémonness in the forms and colours. Still, the shapes are quite amobealike, and the results lack clear, defined detail.

Figure 7: 64 results from the DCGAN generator at various points in training. From left to right: 500 steps, 3000 steps, 5500 steps, 8000 steps.

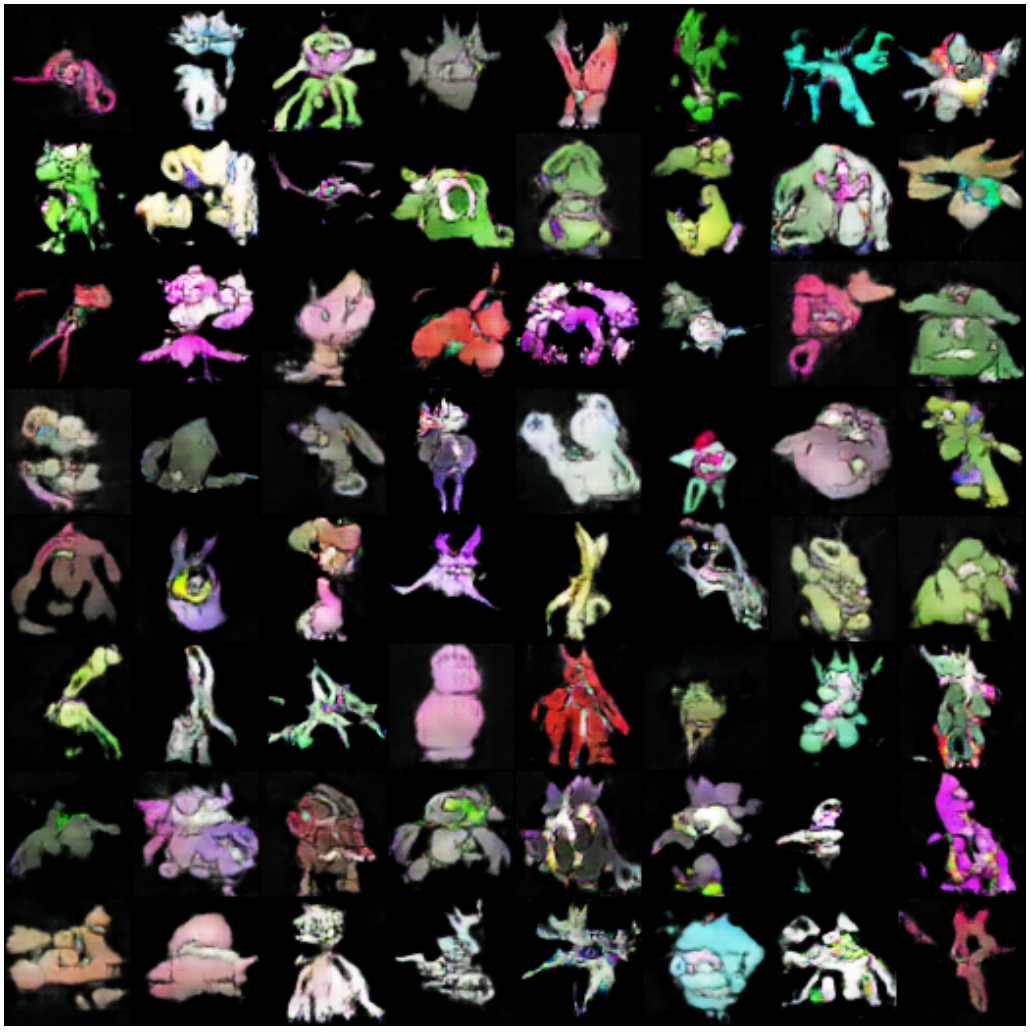

Figure 8: 64 Pokémon generated by the DCGAN generator after 20 training epochs.

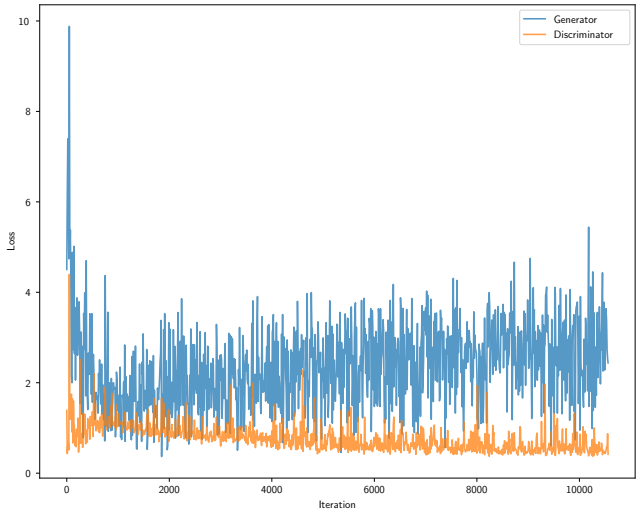

Figure 9: The losses of the SAGAN generator and the discriminator as a function of training steps.

Figure 10: 64 results from the SAGAN generator at various points in training. From left to right: 500 steps, 3000 steps, 5500 steps, 8000 steps.

6.4 Self-attention generator

We continued by adding an attention mechanism between the last batch normalization and the last generator block of our baseline DCGAN architecture. We also added spectral normalization after all deconvolution layers. With these simple modifications, we turned the DGCGAN generator into a self-attention generator. The generator architecture is visualized in figure 2. The model had 3 808 129 trainable parameters.

The training losses (figure 9) and the mid-training results (figure 10) seem quite similar to the baseline (figures 6 and 7). The achievement of similar stability with slightly added complexity might be due to the spectral normalization added to the generator.

The images generated by the SAGAN generator, shown in figure 11, are also very

Figure 11: 64 Pokémon generated by the SAGAN generator after 20 training epochs.

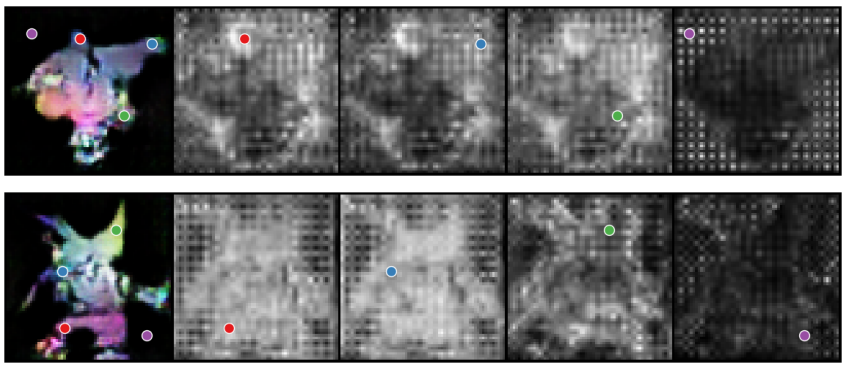

Figure 12: The normalized self-attention maps for two images generated after 20 training epochs. In the leftmost column are the original images with self-attention query locations marked. In the following columns are self attention maps corresponding to the marked query locations. Brighter pixels mean a higher level of attention.

similar in quality to the DCGAN results shown in figure 8. With a bit closer investigation, however, it appears that the SAGAN results are indeed a bit better formed, more symmetrical and more detailed than the DCGAN results. A quick, blind study conducted with some friends in Telegram further advocates this. This might well be due to the added attention mechanism.

Visualizing the self-attention mechanism shows us that this could, indeed, be the case. In figure 12 we can see that the attention seems to focus on relevant areas: in the top image, the attention near the wing-like structures (the red, blue and green dots) focuses on the other wing-like structures (the first three attention maps), and in the bottom image the attention in the left leg-like structure (the red dot) seems to be slightly higher near the feet (the first attention map). In both images the attention in the background focuses mostly in the other background areas, and minimizes focus in the area that the actual Pokémon resides in (the last attention maps).

6.5 Style-based generator

For our style-based generator we scaled down the proposed revised generator in the second StyleGAN paper [4]. The resulting architecture is shown in figure 4. Our style-based generator had 3 830 377 trainable parameters.

As evident from figure 14, the style-based generator suffered a near-immediate mode collapse. The high loss in figure 13 further advocates this theory.

As expected after a mode collapse, the style-based results, shown in figure 15, are very non-varied. However, they are very interesting looking, if not very Pokémon-like.

Despite the mode collapse, the style mechanism seems to work as intended. As

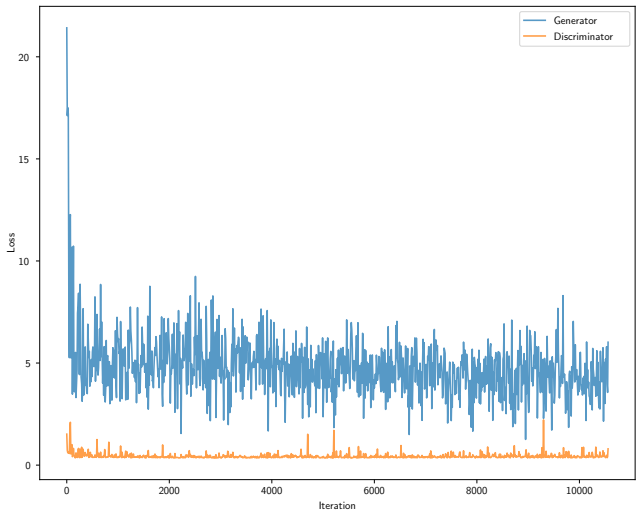

Figure 13: The losses of the StyleGAN generator and the discriminator as a function of training steps.

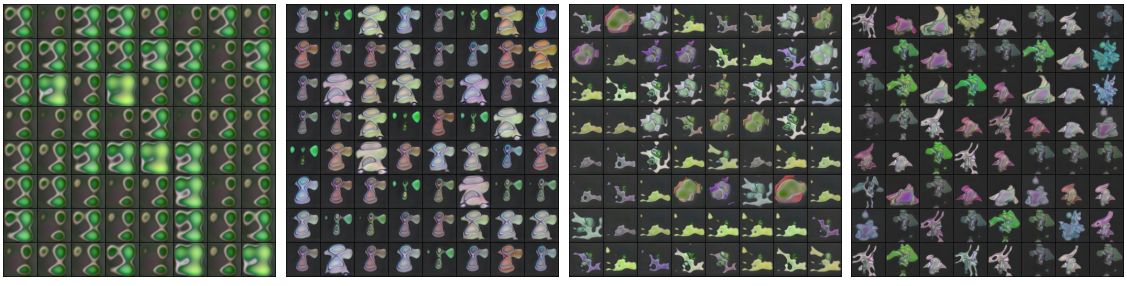

Figure 14: 64 results from the StyleGAN generator at various points in training. From left to right: 500 steps, 3000 steps, 5500 steps, 8000 steps.

Figure 15: 64 Pokémon generated by the StyleGAN generator after 20 training epochs.

Figure 16: The style mechanism in action. The first four styles of each row are the same, and the last six styles of each column are the same.

seen in figure 16, the earlier styles apparently determine the general shape of the generated images: the images on each row are shaped more or less similarly. On the other hand, it looks like the later styles determine details such as colour and "raggedness". It seems like we have indeed achieved some understanding on what different layers do, and obtained new ways to manipulate the image generation.

6.5.1 What went wrong?

As hinted in section 3.1, the StyleGAN papers include a lot of regularization techniques that were suggested to be used during its training. We didn't, however, manage to implement those in a way that didn't lead to an even worse training process. Also, one could argue that having the advantage of a very fine-tuned, customized training environment for one generator would defeat our purpose of comparing different generator architectures.

It's also possible that the style-based architecture is especially bad with very smooth, noiseless images such as Pokémon sprites. This could be inferred from the fact that the style-based generator wasn't able to pick up on the black background.

There's also the possibility of an implementation mistake, as all of the underlying code was patched together from various bits and pieces of code and information from the original paper and existing implementations.

6.6 Comparison of results

The results from StyleGAN (figure 15), while interesting looking, are very unvaried due to the mode collapse problem and not much like our input data. Therefore, our comparison is between the results from SAGAN in figure 11 and from our baseline DCGAN in figure 8.

These sets of results are very similar, but it still seems that the SAGAN has a slight advantage of a bit more symmetrical forms and defined details. Many of the SAGAN results contain, for example, eye-like structures that are basically absent from the DCGAN results. Therefore, according to these experiments the attention mechanism is a worthwhile addition to a generator.

7 Conclusions

We started this report with a quick refresher on GANs and by looking into a couple of problems that GAN generators have. We then investigated two very different GAN generator architectures: the self-attention generator and the style-based generator. We put these architectures to the test with a Pokémon generation problem, and found that the style-based generator is too particular for our crude training scenario, while a self-attention generator produces visually slightly better results than a baseline deeply convolutional generator.

It seems like the self-attention mechanism is a worthwhile but not dramatic improvement to the generator architecture. The attention also seems to be able to take relevant areas of the image into account. On the other hand, even though our results with the style-based generator were not too promising, the style mixing still worked as expected, giving us insight into what each part of the network is doing.

In the end, the new generator architectures probably aren't made to make life easier – with each architectural change training the networks seems to just get more difficult, forcing additional innovations and considerations during the training process. Instead, the true motivation behind them seems to be improved image quality; even we aimed for pretty results instead of an easy, stable and understandable training process when generating Pokémon. The possible problem solving is a welcome side-effect, in which the SAGAN and StyleGAN architectures have excelled.

References

1 Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., and Bharath, A. A. Generative adversarial networks: An overview. *IEEE Signal Processing Magazine 35*, 1 (Jan 2018), 53–65.

2 Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. C., and 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), Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., pp. 2672–2680.

3 Karras, T., Laine, S., and Aila, T. A style-based generator architecture for generative adversarial networks. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019* (2019), Computer Vision Foundation / IEEE, pp. 4401–4410.

4 Karras, T., Laine, S., Aittala, M., Hellsten, J., Lehtinen, J., and Aila, T. Analyzing and improving the image quality of stylegan. *CoRR abs/1912.04958* (2019).

5 Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (2015), Y. Bengio and Y. LeCun, Eds.

6 Radford, A., Metz, L., and Chintala, S. Unsupervised representation learning with deep convolutional generative adversarial networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings* (2016), Y. Bengio and Y. LeCun, Eds.

7 Wang, X., Girshick, R. B., Gupta, A., and He, K. Non-local neural networks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018* (2018), IEEE Computer Society, pp. 7794–7803.

8 Zhang, H., Goodfellow, I. J., Metaxas, D. N., and Odena, A. Self-attention generative adversarial networks. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA* (2019), K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97 of *Proceedings of Machine Learning Research*, PMLR, pp. 7354–7363.