Comparing Normalization Functions for Generative Adversarial Networks

Jan Ebert

May 20, 2019

Abstract

Generative Adversarial Networks have gone through a lot of research to improve stability during training and to refine the quality and size of generated material. This work compares normalization functions for Generative Adversarial Networks in a controlled and deliberately simple fashion using both an acknowledged objective evaluation function and subjective human evaluation.

We confirm literature statements and find combinations of normalization functions that work well together after conducting experiments in both a low-dimensional and high-dimensional case. We also find disadvantages of using the Fréchet Inception Distance as on objective measurement for generative networks.

The source code is available at `https://github.com/janEbert/GANerator` and `https://github.com/janEbert/pytorch-fid`.

1 Introduction

Generative Adversarial Networks (GANs) at present have achieved remarkable success in generating all kinds of data, for example images [1], audio [15] or video [16]. They can be used to transfer styles [17] and even to improve non-generated data (meaning for example noisy data found in the real world or created by hand), for example by enhancing a given image's resolution while generating previously non-existant details [18].

In this work, we present an objective evaluation regarding the effects of different normalization methods on the performance of GANs. The normalization functions we are concerned with are the following:

- Batch normalization [4]

- Instance normalization [5]
- Affine instance normalization [5]
- Virtual batch normalization [6]
- Spectral normalization [7]
- No normalization

As the objective evaluation method, we chose the Fréchet Inception Distance [8] (FID) to allow the use of unlabeled data in the comparisons (which is not possible with the classic Inception Score [6]). Although the results are not 100 % compatible to the FID used in the original paper (because a PyTorch implementation [9, 10] was used), it provides an objective measurement that should not deviate too much from the original implementation.

The source code is largely based on the PyTorch DCGAN Tutorial [2].

2 Background on the Methods

Batch normalization [4] was suggested to address the distribution of a layer's inputs changing after each training step. It normalizes each mini-batch during training and applies a transformation to scale and shift the output using using learnable parameters.

With x_i as the values in a mini-batch, μ as the mean and σ as the standard deviation of a mini-batch, and γ and β as learnable parameters, we obtain the individual elements of each output y_i of a batch normalization layer as

$$y_{i_k} = \gamma \frac{x_{i_k} - \mu}{\sqrt{\sigma + \epsilon}} + \beta, \tag{1}$$

where $0 < \epsilon \ll 1$ is a small constant for numerical stability.

Where batch normalization computes the mean and variation over each mini-batch, *instance normalization* [5] instead computes the normalization values for each value in the batch separately. The effect is that instance-specific mean and covariance shifts are also prevented. With the mean μ_i and standard deviation σ_i per value (over its dimensions) and otherwise the same parameters as in Equation 1, this gives us

$$y_{i_k} = \gamma \frac{x_{i_k} - \mu_i}{\sqrt{\sigma_i + \epsilon}} + \beta. \tag{2}$$

In the source code and for the rest of this text, the above equation equates with affine instance normalization while "instance normalization" instead means that the learnable parameters γ and β do not exist ($\gamma = 1$, $\beta = 0$).

Batch normalization's weakness is that a neural network's output is highly dependent on the other values in a mini-batch, meaning results per value can be skewed due to variance per mini-batch (as all values are normalized with the same statistics). This can be problematic with data that has a high variance. To address this, *virtual batch normalization* [6] computes the normalization statistics not only on the current mini-batch but also on a reference batch that is also propagated through the network.

While the reference batch r of length n (the amount of samples in it) is normalized based only on its own statistics (like in standard batch normalization), the current mini-batch b uses the following formulae for normalization, where $c_b = (n+1)^{-1}$ and $c_r = 1 - c_b$:

$$\begin{aligned} \mu &= c_b * \mu_b + c_r * \mu_r \\ \sigma &= c_b * \sigma_b + c_r * \sigma_r \\ \\ y_{b_{i_k}} &= \gamma \frac{x_{i_k} - \mu}{\sqrt{\sigma + \epsilon}} + \beta \end{aligned} \tag{3}$$

In the above equation, μ_b and σ_b are the mean and standard deviation of the current mini-batch, and μ_r and σ_r are analogous for the reference batch. y_{b_i} is the output of the virtual batch normalization layer for the current mini-batch; the other parameters are the same as in Equation 1.

Even though the reference batch is normalized according to standard batch normalization, for its normalization, the same γ and β are used as for the current mini-batch. Therefore the behaviour of the reference batch during propagation is not the same as if standard batch normalization was applied to it. Due to two values being propagated through the network, virtual batch normalization is also much more computationally costly than any of the other normalization functions discussed here.

Because the function space from which the discriminators are selected largely affects the performance of GANs, Lipschitz continuity has been advocated as a very important characteristic of the discriminator function in order to assure that its derivative is bounded [7]. *Spectral normalization* [7] tries to address this by guaranteeing local Lipschitz continuity for each layer's linear function (the multiplication of the weights with the inputs) with Lipschitz constant $K = 1$, making

a large amount of features per layer possible due to the regularization. To achieve this guarantee, each layer's weights W are normalized according to the spectral norm, given by their largest singular values $\sigma_{max}(W)$ (which is also the L^2 matrix norm of W, $\|W\|_2$). The normalization function for each W is then

$$\mathrm{SN}(W) = \frac{W}{\sigma_{max}(W)}. \tag{4}$$

The objective evaluation method is the *Fréchet Inception Distance* [8]. It uses statistics obtained by applying an Inception model [11] on the data. The statistics considered are the inner coding layers of the Inception machine which contain vision-relevant features. We measure the difference of the multivariate Gaussian distributions of the features of both the real-world dataset (FFHQ in our case) and a generated dataset. This difference is measured by the Fréchet distance. Given the two Gaussians $\mathcal{N}(\mu_r, \Sigma_r)$ and $\mathcal{N}(\mu_g, \Sigma_g)$ for the real and generated data respectively, their Fréchet distance is given by

$$d^2(\mathcal{N}(\mu_r, \Sigma_r), \mathcal{N}(\mu_g, \Sigma_g)) = \|\mu_r - \mu_g\|_2^2 + \mathrm{tr}(\Sigma_r + \Sigma_g - 2(\Sigma_r\Sigma_g)^{\frac{1}{2}}), \tag{5}$$

where $\mathrm{tr}(A)$ denotes the trace of matrix A.

Which features from the Inception machine are used for the calculation of the statistics is arbitrary (though obviously it should not change). To obtain the most abstract, high-level features, we use features from one of the last layers in the Inception model (details in the next section). As we measure the distance – therefore the difference – between the two datasets, a lower FID is better.

3 Experimental Setup

We re-use the same network architecture (explained in detail on page 14), only swapping out the aforementioned normalization functions. The parameters were chosen based on [1], the corresponding PyTorch tutorial [2] and [3]. After two evaluation experiments (see details on experiments below; batch normalization was used for both networks, meaning `normalization` was `Norm.BATCH`) per parameters, the parameters with the better FID were chosen. The small, heuristic search favored the PyTorch DCGAN Tutorial parameters, so all following experiments use those parameters as well (see page 12, Table 4 for details).

Even though the GAN Hacks parameters outperformed the PyTorch DCGAN Tutorial parameters on larger images, results on smaller images were so much better on the PyTorch DCGAN Tutorial parameters that we decided that those were the better starting point for the experiments.

	64	128
DCGAN	65.234	553.479
GAN Hacks	415.984	415.476

Table 1: FID scores in the evaluation of base parameters. Top row is the value of `img_shape` while the left column indicates the type of parameters used (see tables 5 and 6 on page 13). Values rounded to the third decimal place.

All experiments were only conducted once, so they are not statistically significant and we cannot safely assume independence with regards to the initialization (which depends on the random seed). Because GANs consist of two separate networks (discriminator and generator), two normalization functions per experiment must be chosen. The Cartesian product of all six activation functions with themselves yields $6 \cdot 6 = 36$ different combinations.

Each combination has been tested on two different image sizes influencing not only the difficulty of training but also the networks' sizes (amount of layers and therefore parameters). The images trained on and generated were squares of side length 64 and 128. The amount of pixels is $64 \cdot 64 = 4096$ and $128 \cdot 128 = 16\,384$, yielding very different problems in difficulty and – because of that – need for regularization.

After training the different networks, 30 000 random images (random number generator starting seed 0) are generated from the generator. For each experiment, the FID to the dataset is then calculated using the Inception-v3 machine's last average pooling layer's output (the fourth to last layer, see Figure 1). The dataset used is the Flickr-Faces-HQ [13] (FFHQ) dataset, though only the thumbnail version of it (containing images of size 128×128) was used as no larger images were required. We train for 15 epochs to approximate the size of the CelebA [14] dataset which was used in the PyTorch DCGAN Tutorial [2] which was the basis for most parameters.

The FFHQ dataset [13] contains high-quality images of human faces, specifically created as a benchmark for GANs. The images contain faces and backgrounds with considerable variation and also cover accessories such as artificial hair, sunglasses, headwear and more. As the pictures are all obtained from the same source, certain biases are inheritant in the data. The images were preprocessed in the following way, as described in [13]: "The photos were automatically aligned and cropped using dlib. [...] Various automatic filters were used to prune the set, and finally Amazon Mechanical Turk was used to remove the occasional statues, paintings, or photos of photos."

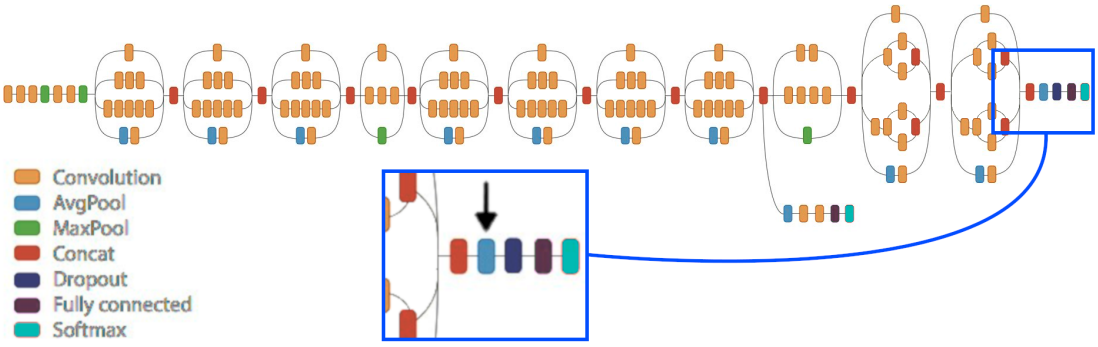

Figure 1: Architecture of the Inception-v3 model. The blue rectangular region represents a zoomed in view. The black arrow indicates the layer from whose output the features were collected. Figure modified from [12].

4 Experimental Results

A box plot of the combined FID scores for the normalization functions is shown in Figure 2 on page 15, giving a visual overview. For the smaller images, affine instance normalization in the discriminator yielded the best results. The best results for the generator were obtained using batch normalization. Together, these two techniques also achieved the minimum FID for the case of smaller images.

Only virtual batch normalization in the discriminator and spectral normalization stood out with much worse results than the others. However, the total FID of virtual batch normalization in the discriminator is heavily influenced by its bad performance with affine instance normalization. All in all, the results for the smaller images are not that conclusive due to the low amount of empirical data. However, batch normalization and instance normalization both achieved solid results in general. The results for the low-dimensional case are collected in Table 2.

Interestingly, spectral normalization and virtual batch normalization were not strong in the low-dimensional case but shone when the problem complexity rose. These two normalization functions were also recommended to be used for only one of the two GAN networks – spectral normalization in the discriminator and virtual batch normalization in the generator (though in the original paper, this recommendation was only based on "computational expensiveness" [6, p. 4]). Although both techniques achieved much better average scores than the other methods in their respective recommended network, when combined, they did not work as well together. Spectral normalization in the discriminator and standard batch normalization in the generator achieved the minimum FID of 91.285 for the high-dimensional case..

G \ D	BN	VBN	IN	AIN	SN	None	**Sum**
BN	71.604	79.816	72.157	71.199	134.378	93.770	522.924
VBN	116.091	164.428	112.274	114.300	174.528	139.162	820.782
IN	94.425	89.495	91.988	93.179	136.420	126.231	631.738
AIN	75.924	658.666	83.101	85.797	148.949	133.996	1186.432
SN	391.383	322.520	422.443	307.532	278.253	264.021	1986.152
None	89.569	378.918	81.310	83.982	136.070	107.864	877.713
Sum	838.995	1693.843	863.273	755.989	1008.598	865.043	6025.741

Table 2: FID scores for the different combinations of normalization functions with `img_shape` set to 64. Each column contains the results for a fixed normalization method in the discriminator, where the normalization in the generator is varied; each row contains the results for a fixed normalization method in the generator, where the normalization in the discriminator is varied. To give an overall estimate on how well each fixed normalization performed in each network, a sum is also given for each row and column. Values rounded to the third decimal place.

These results are much more conclusive than the ones for the low-dimensional case as spectral normalization in the discriminator and virtual batch normalization in the generator emerge as the clear normalization methods of choice for images of size 128. Again, though, the combination of spectral normalization and batch normalization achieved an FID almost 40 points or 43 % better (lower) than the follow-up minimum (spectral and instance normalization in the discriminator and generator respectively). A summary of all the results is given in Table 3.

5 Human Evaluation of Images

To give a less objective but more interpretable, meaningful evaluation method, a short overview on the image quality obtained with the different methods used in the high-dimensional case is given now. This evaluation is based on one human's perceived quality of 64 generated images over training time and will only look at significant examples (no normalization in the discriminator, spectral normalization in the discriminator and virtual batch normalization in the generator, and two special cases). For the high-dimensional case, it is immediately obvious that a lot of methods end early in a collapsed generator. This collapsation has never been escaped in the experiments. Relevant plots and example pictures start on page 16.

Interestingly, no normalization for the discriminator usually yielded better re-

G \ D	BN	VBN	IN	AIN	SN	None	**Sum**
BN	401.034	416.619	484.726	422.747	91.285	262.703	2079.115
VBN	162.829	157.819	139.848	156.811	182.243	174.043	973.594
IN	425.123	435.686	434.324	442.932	131.268	274.168	2143.501
AIN	445.512	454.881	397.076	417.952	159.346	399.155	2273.922
SN	397.497	412.414	391.359	387.741	238.604	399.348	2226.964
None	406.971	447.781	419.423	407.562	214.311	385.283	2281.330
Sum	2238.966	2325.201	2266.757	2235.746	1017.058	1894.699	11978.427

Table 3: FID scores for the different combinations of normalization functions with `img_shape` set to 128. Each column contains the results for a fixed normalization method in the discriminator, where the normalization in the generator is varied; each row contains the results for a fixed normalization method in the generator, where the normalization in the discriminator is varied. To give an overall estimate on how well each fixed normalization performed in each network, a sum is also given for each row and column. Values rounded to the third decimal place.

sults than if normalization was used. This once again emphasizes the importance of the discriminator's function. If it converges too rapidly, the generator will not catch up. When no normalization is used, the discriminator's function is much more unstable giving the generator more freedom in catching up. See also figures 3 and 4. However, the generated pictures were often limited to certain color scapes (for example, as if viewed through rose-tinted glasses) and for some generator normalizations, the training run ended in a huge loss spike near the end with a collapse of the generator, once again emphasizing that regularization – especially early stopping – is important in the training of GANs. Before the collapse, the pictures were always getting more real and more different, meaning the generator noticeably approached a function that varied more in both the size of its output space and its complexity. All in all, while progress to a good generator was visible, the pictures were always very obviously not real and the generator usually collapsed near the end of training.

Even though spectral normalization in the discriminator managed to achieve loss values similar to the other methods (when virtual batch normalization was used in the generator), its generated pictures were usually too dark with only few examples remaining light. However, although dark, the pictures were of good quality. This means that while the discriminator's function space was reduced, so was the generator's due to the evaluation – which did have positive effects on the image quality. The same problem was still suffered with virtual batch

normalization in the generator, although less succinctly. Loss plots and pictures on pages 18 and 19.

Virtual batch normalization in the generator achieved the best image results by far no matter which other method it was used with. The best results out of those were, however, achieved with virtual batch normalization in the discriminator as well. That virtual batch normalization was so important in generating good pictures also emphasizes the importance of the generator's function. While methods like spectral normalization try to solve the GAN game via regularizing the discriminator, virtual batch normalization achieved far better "real-world" (humanly evaluated) results, only by regularizing the generator, even though spectral normalization achieved similar results in the objective evaluation. See figures 7 and 8 for loss plots and pictures of qualitative networks.

Two final special cases both had virtual batch normalization in the discriminator. One with normalization in the generator managed to collapse the generator in a way that had not been achieved in any other experiment, having it stuck generating pictures filled with only yellow (and with corner shadows). Maybe with other starting seeds, this method can result in good a good GAN.

The other special case occured with spectral normalization in the generator. While the results were not good, this method managed to not collapse. Both loss plots and pictures are shown in figures 9 and 10 on pages 22 and 23.

6 Conclusion

While the results shown here are to be taken with caution (due to the low amount of experimental data), a certain trend in the higher-dimensional case is observable: Spectral normalization belongs in the discriminator and virtual batch normalization in the generator. Even though the combination of the two was not the strongest, the two functions generally worked well with any other normalization.

However, subjectively evaluated, virtual batch normalization in the generator clearly outperformed spectral normalization in the discriminator, meaning that – through all experiments performed –, virtual batch normalization is the "winner" method for this experimental setup.

While the low-dimensional case has shown that the two normalization functions above do not belong in the other respective network, especially for the discriminator, no normalization function stood out. In the generator case, standard batch normalization worked really well and instance normalization was – as recommended in the GAN hacks [3] – a good alternative, confirming that "GAN hack".

7 References

[1] Alec Radford, et al., *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*, 2015, Retrieved from `https://arxiv.org/abs/1511.06434v2` on 2018-10-28.

[2] Nathan Inkawhich, *DCGAN Tutorial*, 2017, Retrieved from `https://github.com/pytorch/tutorials/tree/51098240d5f9c61c2979ec621fd3f34e7a220dbe` on 2018-11-05.

[3] Soumith Chintala, et al., *How to Train a GAN? Tips and tricks to make GANs work*, 2018, Retrieved from `https://github.com/soumith/ganhacks/tree/d09bbf3983c8b6ab1709f87379c8f969c8a81f93` on 2018-11-12.

[4] Sergey Ioffe & Christian Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, 2015, Retrieved from `https://arxiv.org/abs/1502.03167v3` on 2017-03-05.

[5] Dmitry Ulyanov, et al., *Instance Normalization: The Missing Ingredient for Fast Stylization*, 2016, Retrieved from `https://arxiv.org/abs/1607.08022v3` on 2018-07-11

[6] Tim Salimans, et al., *Improved Techniques for Training GANs*, 2016, Retrieved from `https://arxiv.org/abs/1606.03498v1` on 2018-11-08.

[7] Takeru Miyato, et al., *Spectral Normalization for Generative Adversarial Networks*, 2018, Retrieved from `https://arxiv.org/abs/1802.05957v1` on 2018-11-09.

[8] Martin Heusel, et al., *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*, 2017, Retrieved from `https://arxiv.org/abs/1706.08500v6` on 2019-04-08.

[9] "mseitzer", *Fréchet Inception Distance (FID score) in PyTorch*, 2018, Retrieved from `https://github.com/mseitzer/pytorch-fid/tree/4e366b2fc9fb933bec9f6f24c5e87c3bd9452eda` on 2019-04-20.

[10] "Vermeille", *Dramatically speed up FID computation using DataLoader for asynchronous data loading*, 2019, Retrieved from `https://github.com/mseitzer/pytorch-fid/pull/9` and `https://github.com/Vermeille/pytorch-fid/tree/7974b090c66e6d06507a31974b24b3aa6ac0b3ac` on 2019-05-07.

[11] Christian Szegedy, et al., *Rethinking the Inception Architecture for Computer Vision* 2015, Retrieved from `https://arxiv.org/abs/1512.00567v3` on 2017-07-09.

[12] Christian Szegedy, et al., *inception v3 architecture* 2015, Retrieved from `https://github.com/tensorflow/models/blob/f87a58cd96d45de73c9a8330a06b2ab56749a7fa/research/inception/g3doc/inception_v3_architecture.png` on 2019-05-11.

[13] Tero Karras, et al., *Flickr-Faces-HQ Dataset (FFHQ)*, 2018, Retrieved from `https://github.com/NVlabs/ffhq-dataset/tree/6bd5b2c70639c867867ef691163cb92bbec4130e` on 2019-04-10.

[14] Ziwei Liu, et al., *Large-scale CelebFaces Attributes (CelebA) Dataset*, 2015, Retrieved from `http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html` on 2018-11-08.

[15] Chris Donahue, et al., *Adversarial Audio Synthesis*, 2018, Retrieved from `https://arxiv.org/abs/1802.04208v3` on 2019-05-20.

[16] Justus Thies, et al., *Face2Face: Real-time Face Capture and Reenactment of RGB Videos*, 2016, Retrieved from `http://niessnerlab.org/projects/thies2016face.html` on 2018-12-22.

[17] Yunjey Choi, et al., *StarGAN: Unified Generative Adversarial Networks for Multi-Domain Image-to-Image Translation*, 2017, Retrieved from `https://arxiv.org/abs/1711.09020v3` on 2018-10-28.

[18] Xintao Wang, et al., *ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks*, 2018, Retrieved from `https://arxiv.org/abs/1809.00219v2` on 2019-01-08.

A Parameters

Name	Value
seed	0
ensure_reproducibility	False
flush_denormals	True
epochs	15
batch_size	128
resize	True
data_mean	0.0
data_std	1.0
float_dtype	torch.float32
g_input	128
g_flip_labels	False
d_noisy_labels_prob	0.0
smooth_labels	False
features	64
optimizer	optim.Adam
lr	0.0002
optim_param	((0.5, 0.999),)
optim_kwargs	{}
activation	(nn.LeakyReLU, nn.ReLU)
activation_kwargs	({'negative_slope': 0.2, 'inplace': True}, {'inplace': True})

Table 4: Parameters kept the same over all experiments.

Name	Value
g_flip_labels	False
d_noisy_labels_prob	0.0
smooth_labels	False
optimizer	optim.Adam
optim_param	((0.5, 0.999),)
activation	(nn.LeakyReLU, nn.ReLU)
activation_kwargs	({'negative_slope': 0.2, 'inplace': True}, {'inplace': True})

Table 5: Parameters based on the DCGAN paper [1] and PyTorch DCGAN tutorial [2] which were evaluated as possible base parameters. Only these parameters changed from the ones in Table 4. These were also chosen as base parameters due to the evaluation.

Name	Value
g_flip_labels	True
d_noisy_labels_prob	0.1
smooth_labels	True
optimizer	(optim.SGD, optim.Adam)
optim_param	(0, (0.5, 0.999))
activation	nn.LeakyReLU
activation_kwargs	{'negative_slope': 0.2, 'inplace': True}

Table 6: Parameters based on GAN Hacks [3] which were evaluated as possible base parameters. Only these parameters changed from the ones in Table 4.

B Architecture

The architecture used follows the same architecture as described in [1] and the DCGAN PyTorch tutorial [2]. To be able to work with arbitrarily large images (actually, squares with any side length divisibile by two), the discriminator consists of layers with increasingly many features (doubled in each convolutional layer). The generator mirrors the behaviour, halving the number of features in each transposed convolutional layer. The repeating layers feature the same convolutional kernel size, stride and padding (4×4, 2 and 1 respectively), creating an iteratively shrinking "image" in the discriminator and – once again mirrored – a growing "image" in the generator. Due to the stride and padding, the "image's" side length is halved on each step. "Image" is only a visual description as each layer's output is imaginable as a cuboid with a depth of the amount of features and with a side length and height of the "image".

When the "image" has been shrunk to a 4×4 square in the discriminator, the stride is set to 1 and no padding is applied, leaving a 1×1 square. The depth of the cuboid (amount of features) is then also set to 1, leaving a $1 \times 1 \times 1$ cube (a scalar) that the output function is executed on (returning the expected probability of the image being real). The generator reverses the process exactly, only starting from the latent vector z of size `g_input`. The first transposed convolutional layer creates a 4×4 "image" that is then iteratively doubled in side length while the cuboid's depth is halved at each step as explained earlier. When the desired image size has been reached (at which point the cuboid's depth has been reduced to 1), the generator's output function is applied.

The discriminator has no normalization after the first convolutional layer. The output functions are `Sigmoid` for the discriminator and `Tanh` for the generator.

C FID Plots

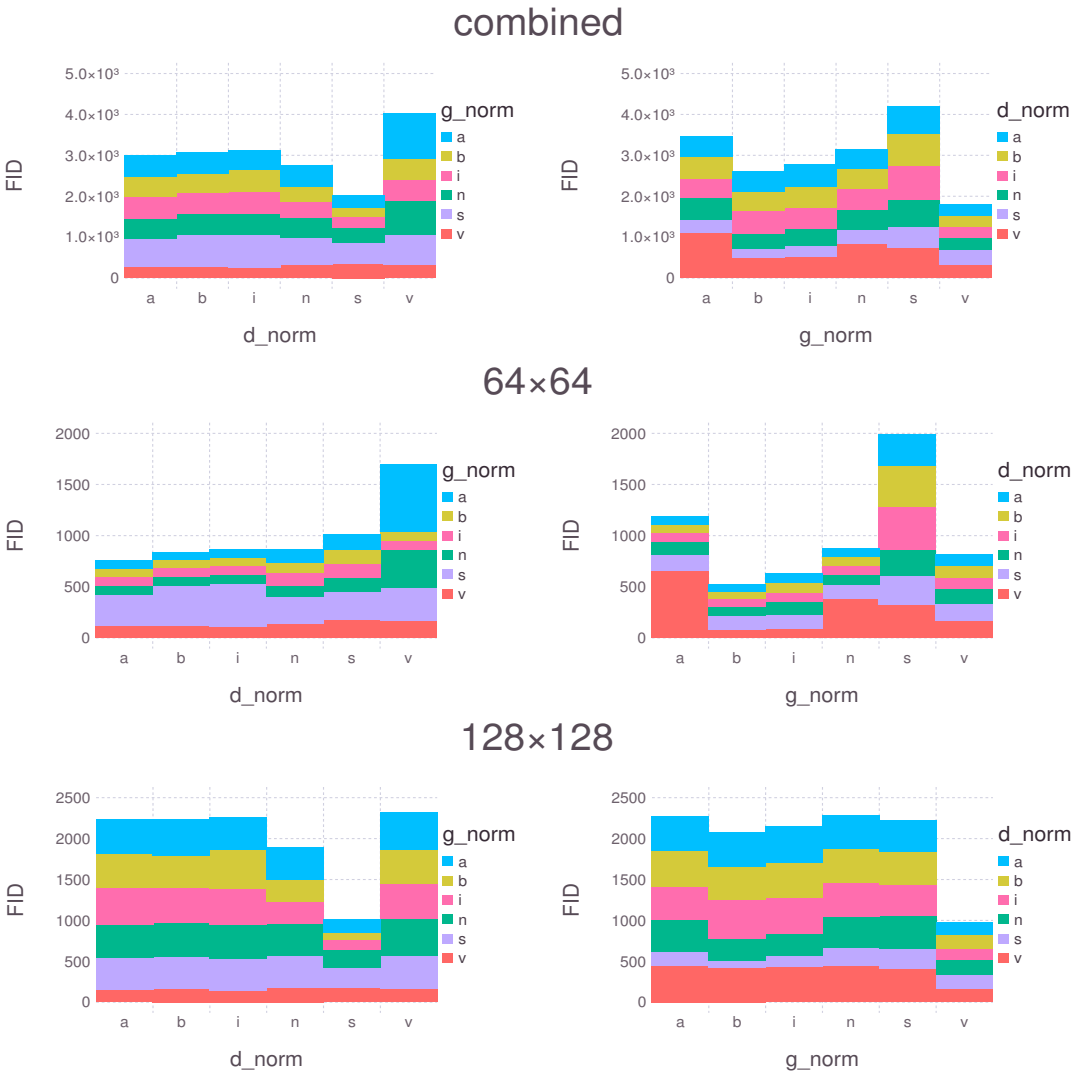

Figure 2: FID scores for the different normalization function combinations (smaller is better). The top row shows the combined results for both image sizes, the middle row the results for experiments with `img_shape` set to 64 and the bottom row the same for `img_shape` as 128. Plots in the left column show the discriminator's normalization on the x axis and indicate the generator's normalization via different colors. The right column shows the same but switches the discriminator and generator. The labels for each color distinctly map to each normalization's starting letter.

D Loss Plots and Pictures

All plots and images in this section share an `img_shape` of 128. Pictures shown are only those generated after training finished (8206 training steps). Discussion in Section 5 on page 7.

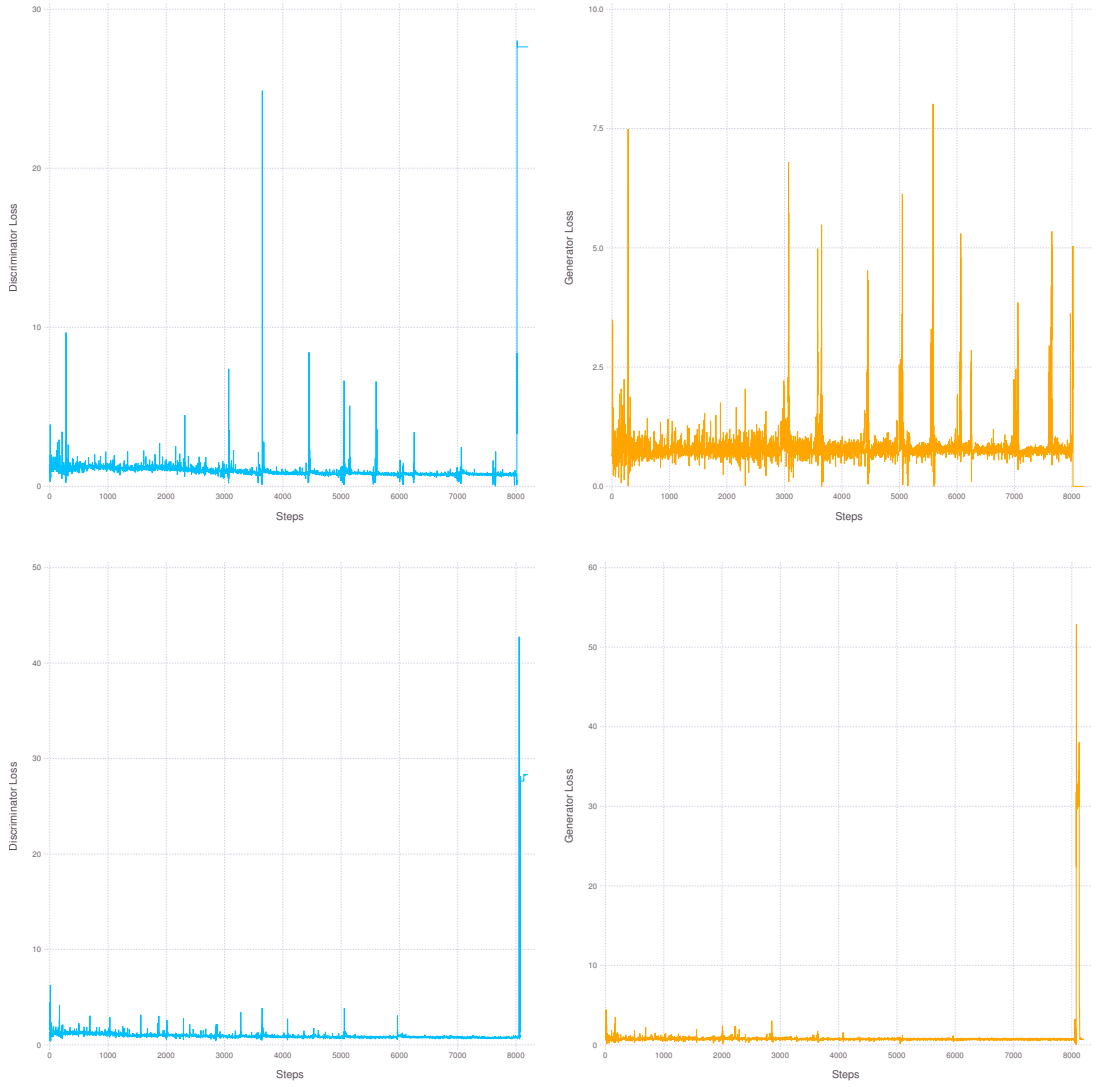

Figure 3: Losses over training steps for the discriminator on the left, the same for the generator on the right. Instance normalization in the generator on top, affine instance normalization in the generator on the bottom. For both, no normalization was used in the discriminator. While the losses varied much more for the networks on top, the resulting generated pictures were similar.

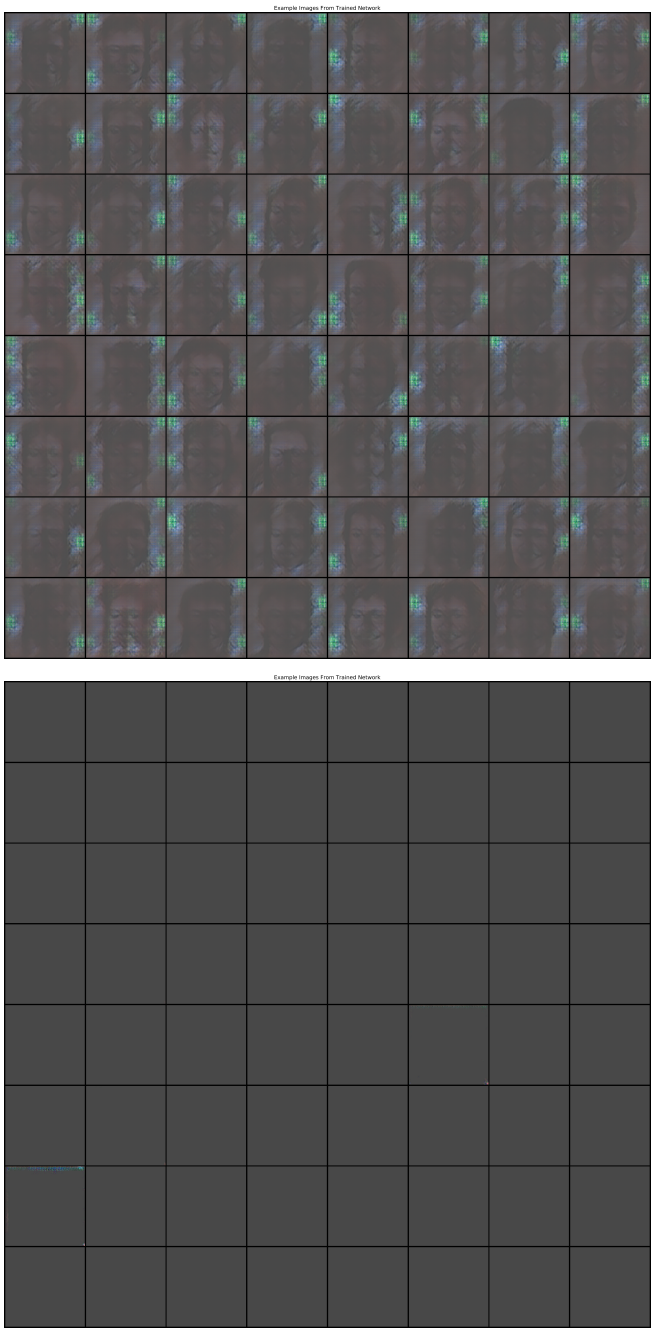

Figure 4: Generated images for the two networks discussed a page prior. Instance normalization in the generator on top, affine instance normalization in the generator on the bottom. For both, no normalization was used in the discriminator. Sadly, these networks collapsed, althouph earlier pictures showed much better quality.

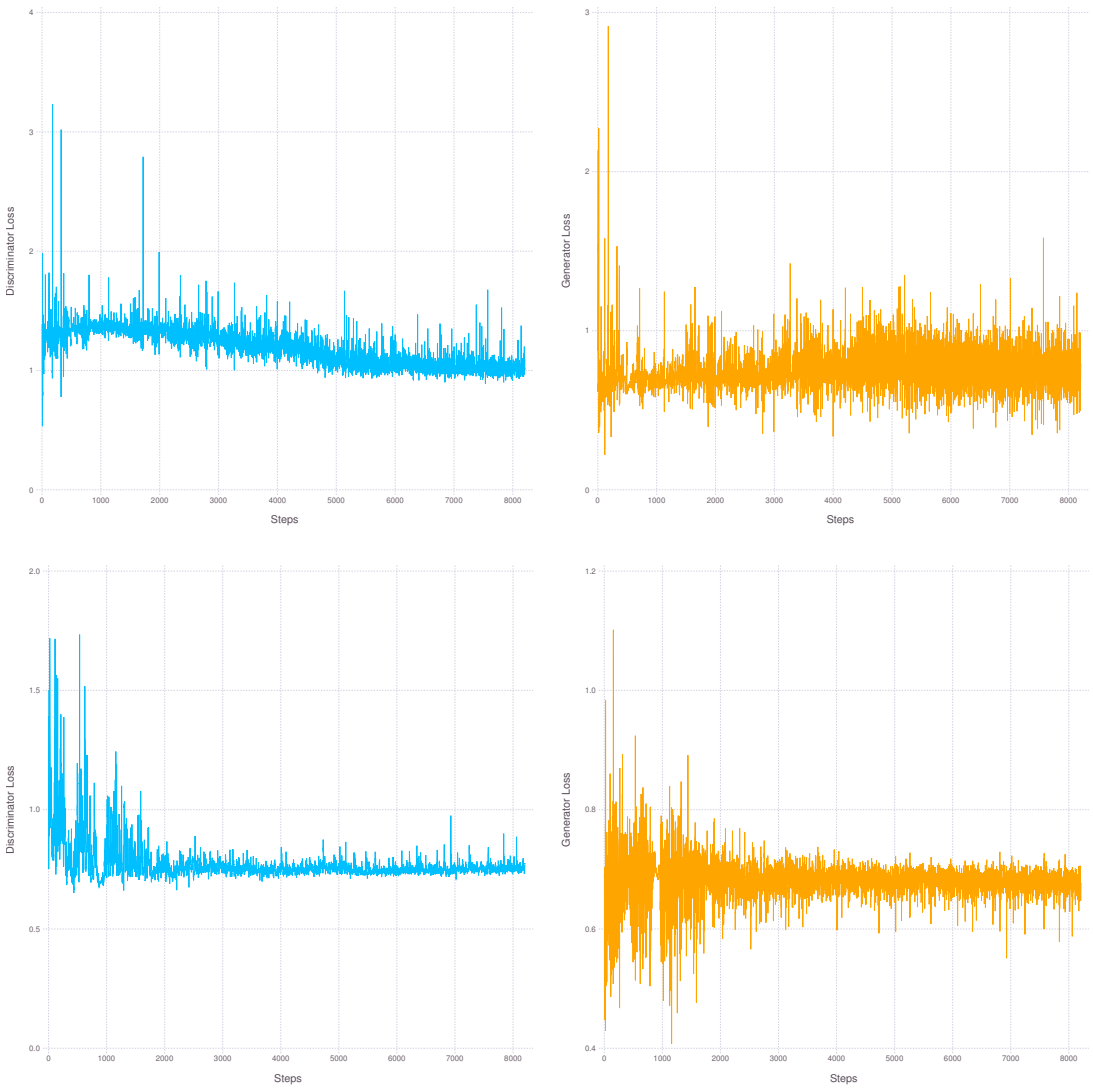

Figure 5: Losses over training steps for the discriminator on the left, the same for the generator on the right. Batch normalization in the generator on top, spectral normalization in the generator on the bottom. For both, spectral normalization was used in the discriminator. Even though the loss plots are similar to those that yielded good results (see for example Figure 7), the generated images are not as qualitative.

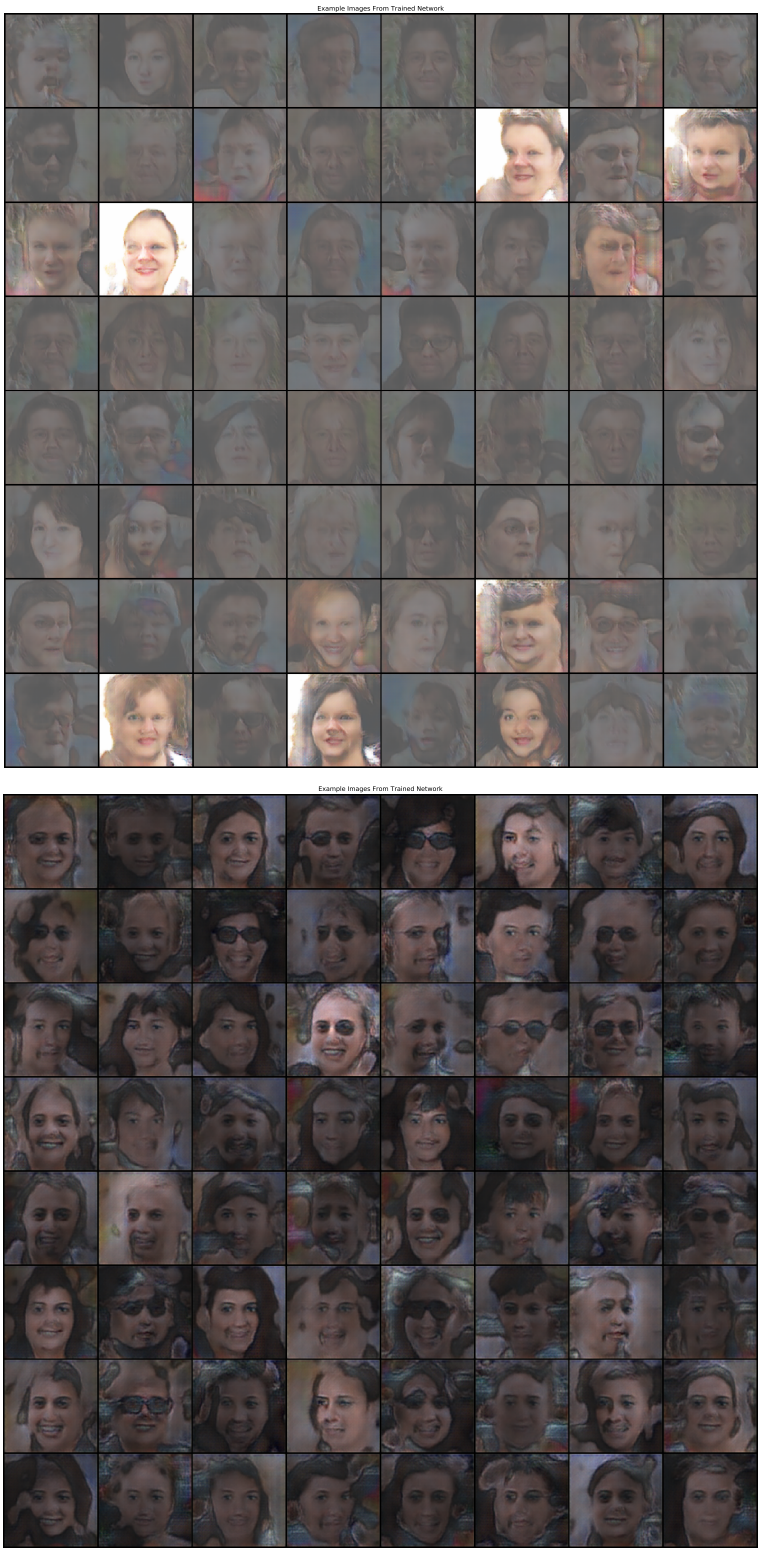

Figure 6: Generated images for the two networks discussed a page prior. Batch normalization in the generator on top, spectral normalization in the generator on the bottom. For both, spectral normalization was used in the discriminator.

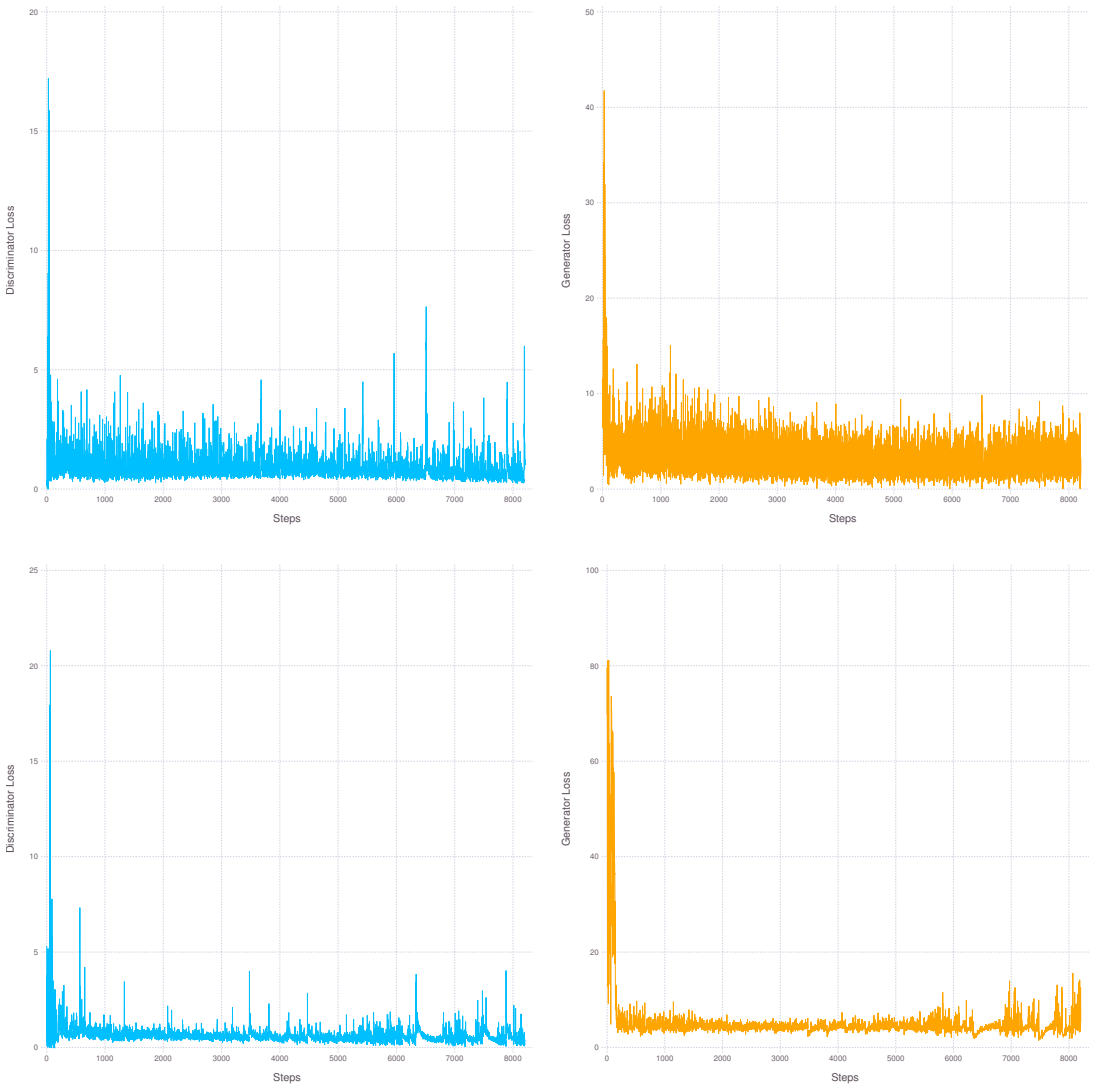

Figure 7: Losses over training steps for the discriminator on the left, the same for the generator on the right. Batch normalization in the discriminator on top, virtual batch normalization in the discriminator on the bottom. For both, virtual batch normalization was used in the generator. These methods both yielded acceptable results, objectively and subjectively evaluated.

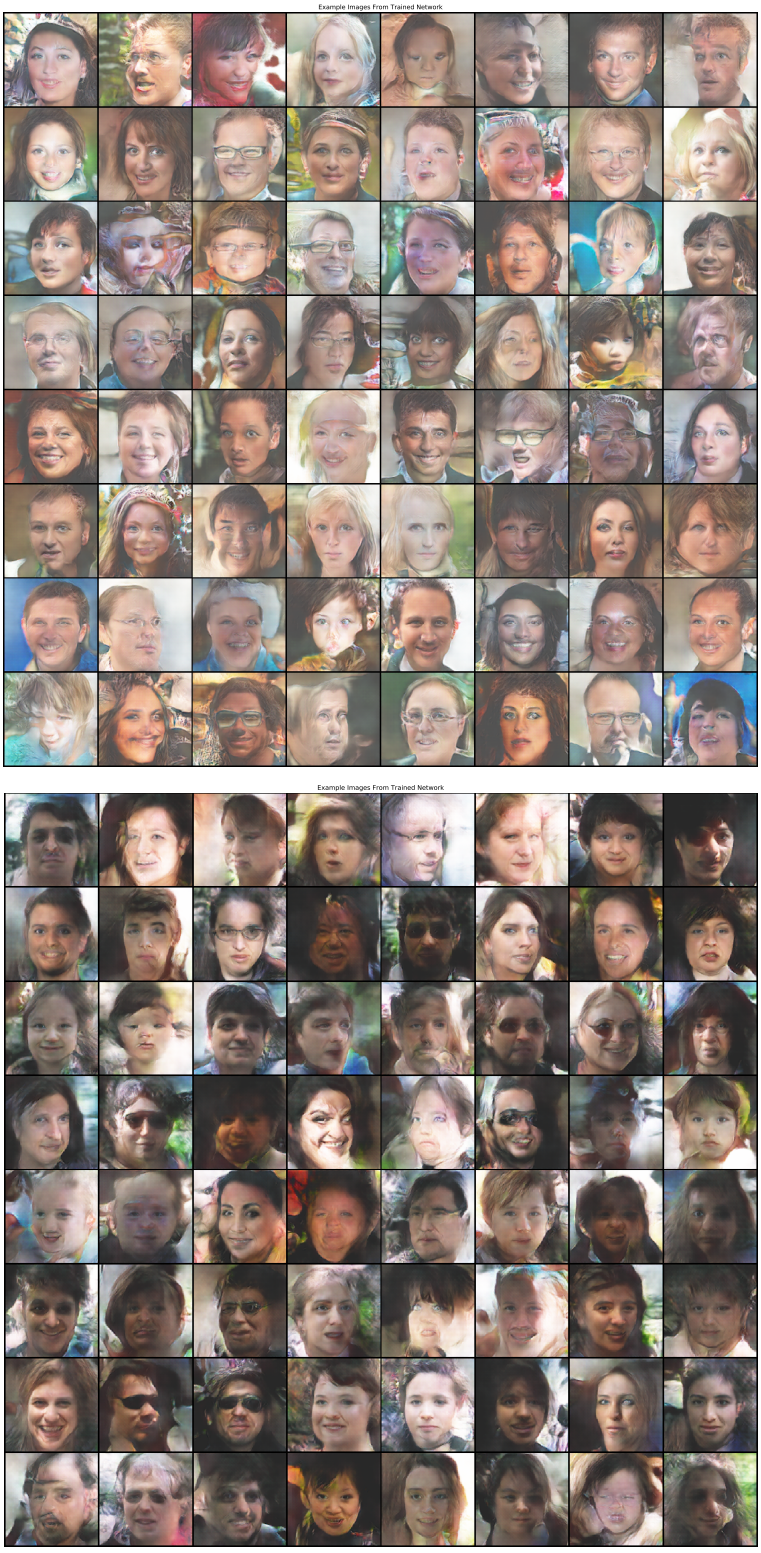

Figure 8: Generated images for the two networks discussed a page prior. Batch normalization in the discriminator on top, virtual batch normalization in the discriminator on the bottom. For both, virtual batch normalization was used in the generator.

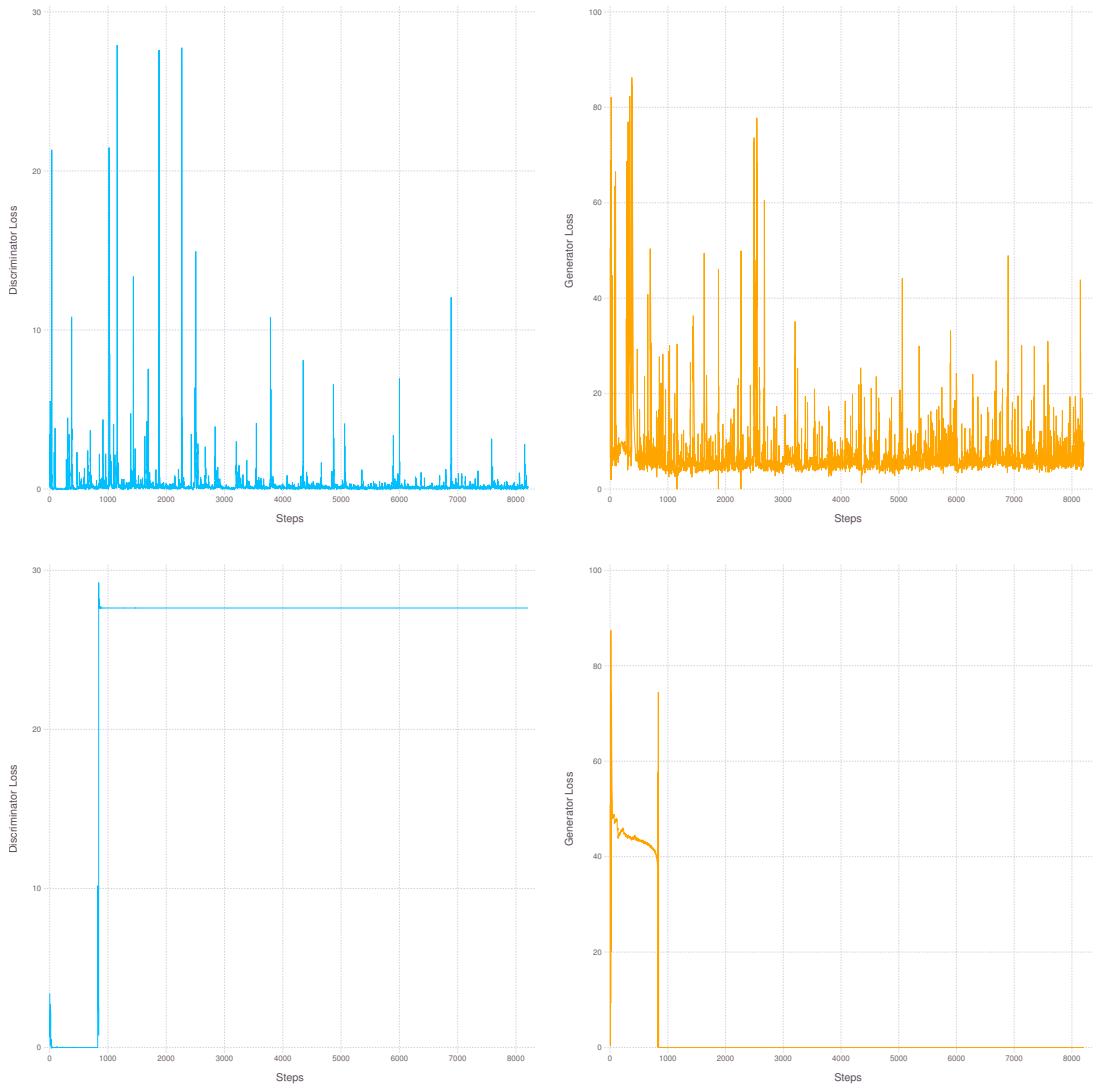

Figure 9: Losses over training steps for the discriminator on the left, the same for the generator on the right. Spectral normalization in the generator on top, no normalization in the generator on the bottom. For both, virtual batch normalization was used in the discriminator. The method on top did not collapse while the one on the bottom collapsed in a unqiue way.

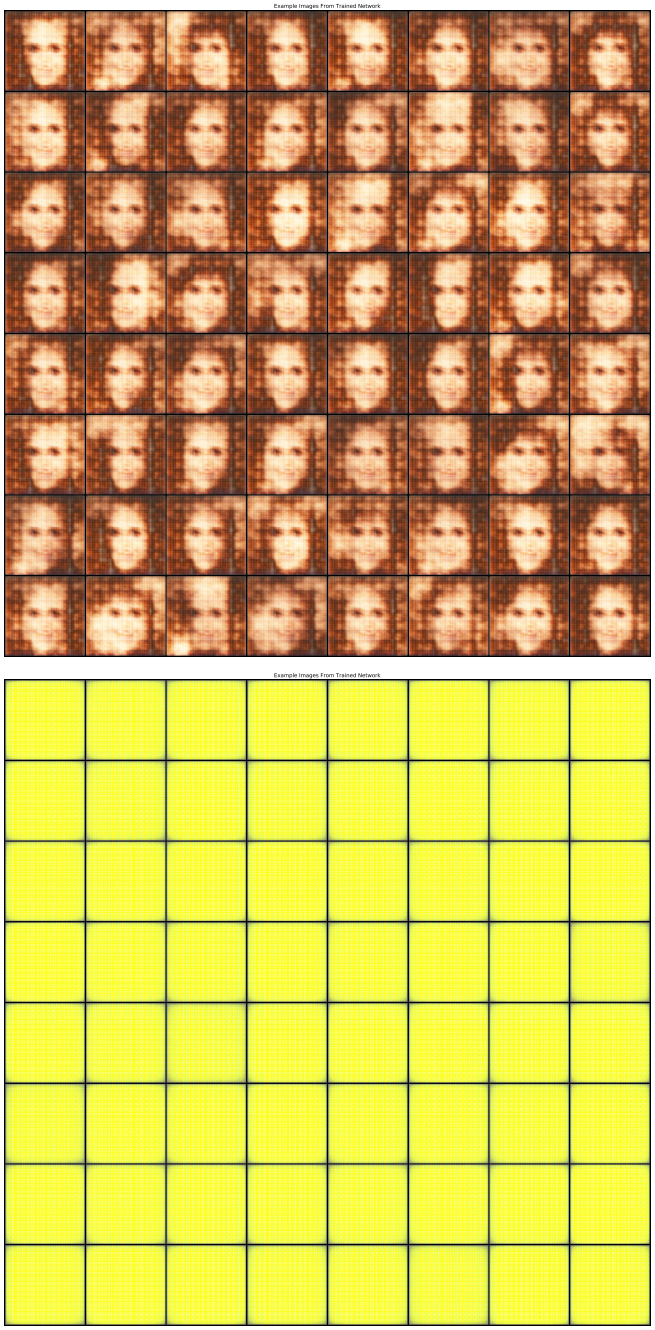

Figure 10: Generated images for the two networks discussed a page prior. Spectral normalization in the generator on top, no normalization in the generator on the bottom. For both, virtual batch normalization was used in the discriminator. The pictures on top also portray images generated early on during training when using spectral normalization in the discriminator well.

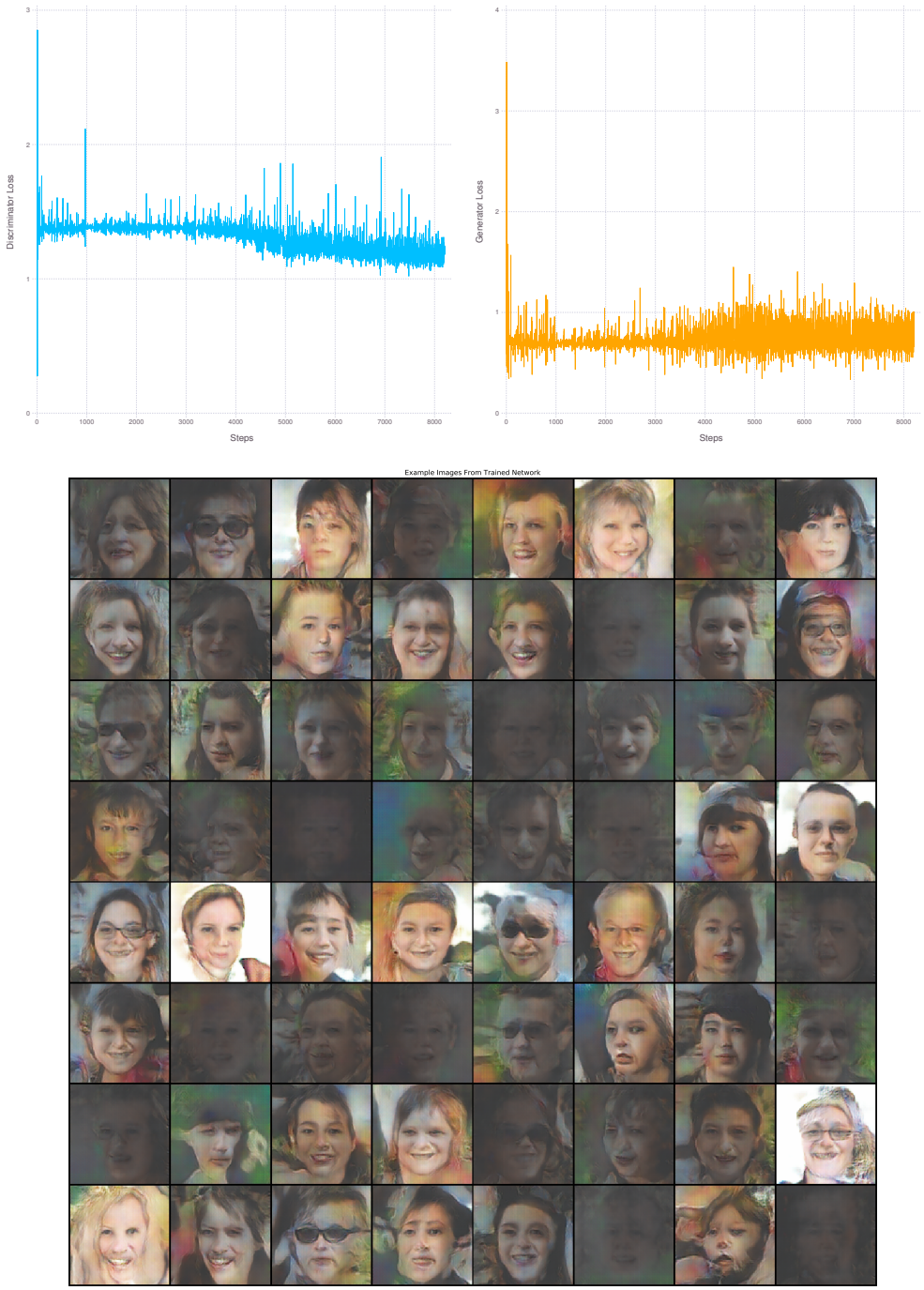

Figure 11: Losses over training steps on the top for the discriminator on the left, the same for the generator on the right. Generated images on the bottom. Spectral normalization in the discriminator, virtual batch normalization in the discriminator. Combining the methods yielding the minimum FIDs in their respective network, we achieve results that are not bad but are lacking in comparison to others methods using virtual batch normalization in the generator.