TECHNICAL UNIVERSITY OF MUNICH

TUM Data Innovation Lab

"CreativeAI"

Authors	Anastasia Litinetskaya, Felix Altenberger, Jongwon Lee, Vadim Goryainov, Vitalii Mozin
Mentors	Arndt Kirchhoff, Matthias Rebs, Matthias Wissel (Capgemini Deutschland GmbH)
Co-Mentor	M.Sc. Sandro Belz (Department of Mathematics)
Project Lead	Dr. Ricardo Acevedo Cabra (Department of Mathematics)
Supervisor	Prof. Dr. Massimo Fornasier (Department of Mathematics)

July 2019

Abstract

In this final report, we summarize our project "CreativeAI - Using GANs for text-based image creation" of the TUM Data Innovation Lab, in cooperation with Capgemini, in the summer semester 2019. The goal of our project was to build an application to automatically generate realistic images based on text descriptions. We have implemented multiple models based on Generative Adversarial Networks, with which the user interacts via a web-frontend. There the user can simply enter a text description and a corresponding image will be generated.

The models used in our implementation are mostly based on the StackGAN [10] and AttnGAN [20] network architectures. We have trained most of these models ourselves and have implemented various performance-improving extensions. As part of our project, we have also researched a large variety of GAN literature, including multiple works on GAN stabilization and GAN evaluation, which we have partly applied in our implementation.

Furthermore, our implementation could be delivered and deployed in a real-world scenario, as it is based on a modular system-independent architecture. We also employed state-of-the-art software development methods to ensure high code quality, and easy maintainability and extensibility.

Contents

1 Introduction

Generative adversarial networks (GANs) [6] are a recent development for image generation. Compared to other methods, GANs are able to generate significantly more detailed images, which enables a lot of exciting applications. In this project, we have used GANs to generate images from text. For this purpose, we have built a demo that is accessible online at `http://10.195.1.122:8050/`. There, the user simply selects a model, enters an image description, and a corresponding image will automatically be generated. An example of this is shown in Figure 1.

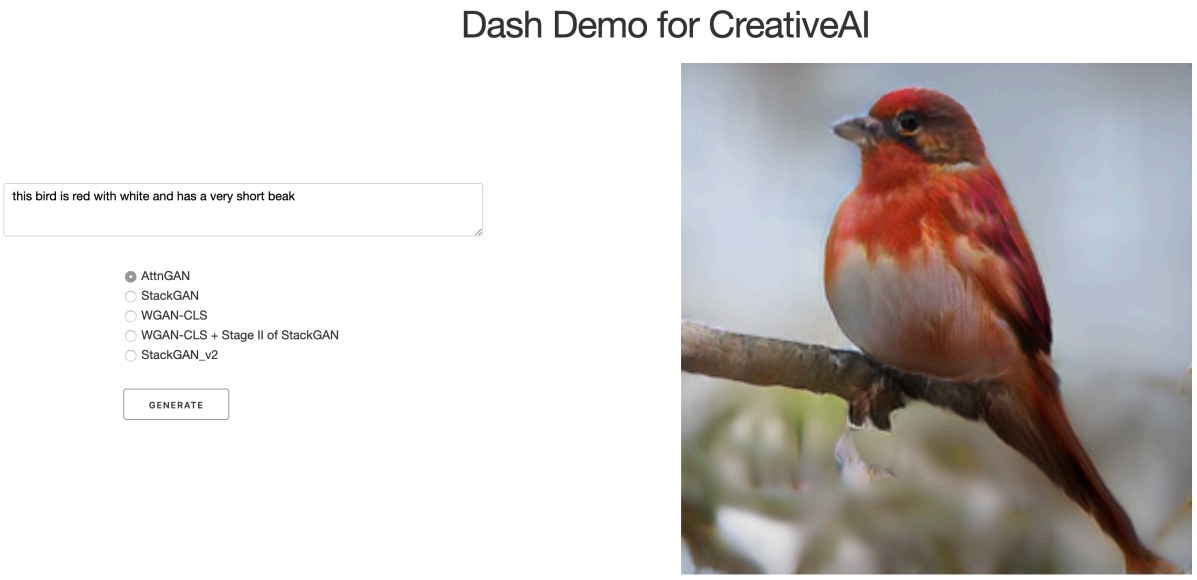


Figure 1: Demo of our CreativeAI implementation. When the user enters a text description, chooses a model, and presses the 'generate' button, an image matching the description will automatically be generated.

In the following, we will explain our system and our GAN models in more detail. First, we will have an introduction to the theoretical background of the project in Section 2. There, we will explain how generative adversarial networks work, how they can be used to generate images based on text, how to address the issue of unstable GAN training, and how to evaluate GAN performance. Then, in Section 3, we will go into more detail on the functionality of our implementation. Furthermore, we will explain the underlying system architecture, under the aspects of both development and deployment. Afterwards, we will show a comparison of our models in Section 4. Specifically, we will show generated images of our models and list the advantages and shortcoming of each model. Finally, we will conclude our report with potential next steps in Section 5. Our code repository is available at `https://gitlab.lrz.de/di57sam/creativeai`.

2 Image Generation with Generative Adversarial Networks

2.1 Foundations

2.1.1 Generative Adversarial Networks

A *generative adversarial network* (GAN) [6] consists of two main components: A *generator* G and a *discriminator* D. The goal of the generator is to learn the distribution of input data to generate fake data that is indistinguishable from the real one. The discriminator's goal is to distinguish the fake data from real data. These two models are trained together, whereby the generator's task is to deceive the discriminator into judging the generated data as real ($D(G(z)) = 1$), and the discriminator's task is to correctly identify real data as real ($D(x) = 1$) and generated data as fake ($D(G(z)) = 0$).

This can be expressed as the following minimax game [6]:

$$\min_G \max_D \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]. \tag{1}$$

It can be shown [6] that this minimax game converges to an optimal generator, which produces fake data with the same distribution as the training data, and an optimal discriminator always predicting 0.5.

In practice, the generator is typically trained on the following, equivalent, more stable objective [6]:

$$\max_G \mathbb{E}_{z \sim p_z(z)} \log D(G(z)). \tag{2}$$

2.1.2 Conditional GANs

It is also possible for GANs to learn conditional generations, simply by providing additional input data to both generator and discriminator, which changes the minimax formulation in equation 1 to:

$$\min_G \max_D \mathcal{L}_{cGAN}(G, D) = \mathbb{E}_{x,y}[\log D(x|y)] + \mathbb{E}_{z,y}[\log(1 - D(G(z|y)|y))]. \tag{3}$$

Such a model is called *conditional GAN* (cGAN) [12]. cGANs have been used for many applications, including image generation from text [16, 10, 20], image generation from discrete labels [12], and image generation from images [4, 14, 18].

2.2 Image Generation from Text

2.2.1 StackGAN

The results of the training of conventional GANs show that the quality of the generated images declines with increasing image resolutions. StackGAN [10] tackles that problem by combining multiple generator/discriminator pairs in order to produce more stable results. It builds upon the conditional GAN model which allows training the model to output

images based on text descriptions, which consequently requires a dataset of images that are labeled with captions.

StackGAN introduces the conditioning augmentation method for text captions: before starting the training, a Gaussian distribution is fitted on embeddings of the image captions. Instead of taking a real image caption from the dataset, we sample from the appropriate Gaussian. This is beneficial, as the dimensionality of the text embeddings is rather high and the datasets do not cover a sufficient variety of that space. The conditioning augmentation thus yields more training data for a small number of image captions. The authors of [10] also argue that the conditioning augmentation is justified as any text can have a great variety of possible images associated with it.

The first version of the StackGAN consists of two generator/discriminator pairs. One pair (called Stage I) is responsible for generating and discriminating 64x64 pixel images, which is a standard image size for conventional GANs. The second pair (Stage II) takes images of the size 64x64 as an input and outputs 256x256 pixel images. A graphical representation of the architecture can be seen in Figure 2.

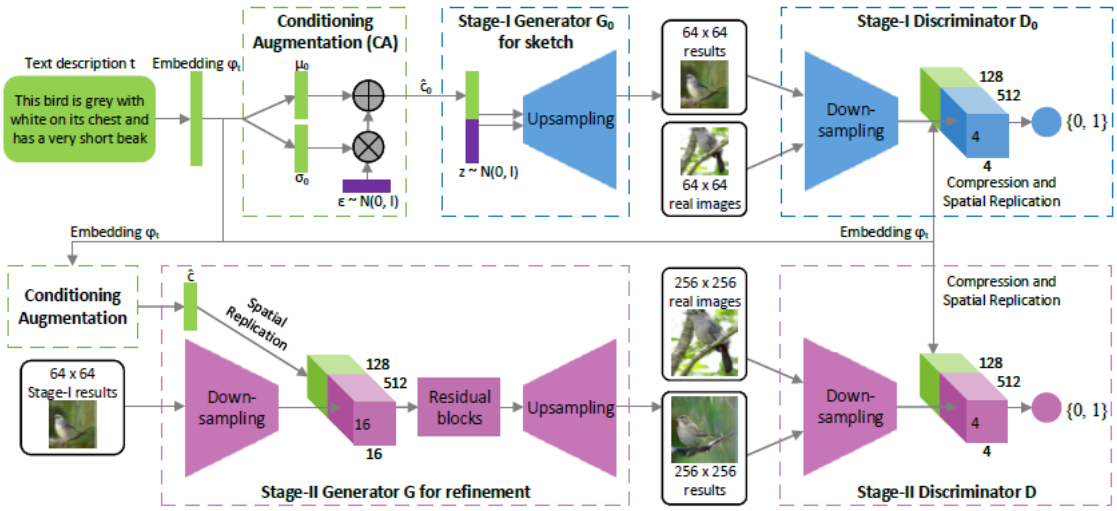


Figure 2: Architecture of StackGAN version 1, taken from the original paper [10].

In the first phase of the training process, a caption is sampled from the training set and then passed through the conditioning augmentation in order to generate a conditioning variable. This variable is concatenated with a sample from the random uniform distribution and passed to the Stage I generators and discriminators. The discriminator additionally receives images from the dataset (resized to the required 64x64 pixels) that fit the caption that has been sampled previously. Stage I can be trained independently from Stage II.

The Stage II pair receives a conditioning variable along with real images from the dataset (resized to 256x256 pixels) and fake images from the Stage I generator. Note that no additional source of randomness is used. The fake images are passed to the generator which has the task to augment the low-resolution images (64x64 pixels) into more realistic looking high-resolution images (256x256). Next, the discriminator trains on discriminating

the real images against the fake ones from the generator. In the implementation, all the discriminators and generators are convolutional neural networks.

2.2.2 StackGAN++

Although the first version of the StackGAN model shows promising results, they still can be greatly improved. One notable problem is *mode collapse* (discussed in Section 4) where many extremely similar images are generated. The StackGAN model is highly susceptible to mode collapse [10]. [9] proposes a new model — StackGAN++ or StackGAN version 2 — that shows a higher variety of generated images. The model uses different layer structures for the generators and discriminators and introduces one additional generator/discriminator pair for 128x128 pixel images in between the 64x64 pixel pair and the 256x256 pixel pair. The architecture is illustrated in Figure 3.

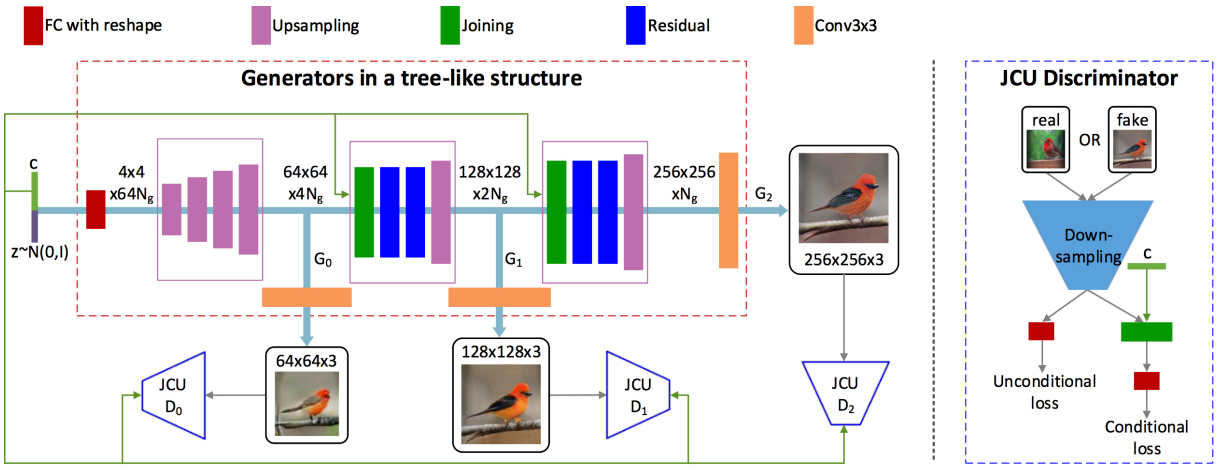


Figure 3: Architecture of the StackGAN++ model, taken from the original paper [9].

Other notable differences from the first version include the joint conditional and unconditional loss and the color-consistency regularization.

The idea of the joint conditional and unconditional loss is to combine the conventional GAN loss function with the conditional GAN loss function. This yields the discriminator objective function

$$\begin{aligned}\mathcal{L}_D = \max_D \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] \\ + \mathbb{E}_{x,y}[\log D(x|y)] + \mathbb{E}_{z,y}[\log(1 - D(G(z|y)|y))],\end{aligned} \tag{4}$$

and the generator objective function

$$\mathcal{L}_G = \min_G \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] + \mathbb{E}_{z,y}[\log(1 - D(G(z|y)|y))]. \tag{5}$$

The intuitive reasoning behind this idea is that we train the generator to output images that fit well to the text description but also include the constraint that they should look unconditionally well. This aims to improve the overall quality of the output images.

The color-consistency regularization furthermore adds an additional term to the objective function of the generators that rises if the colors from the outputs of two generators diverge too much. This should encourage the second and the third generators to refine the input images rather than to generate a completely new image. However, [10] points out that in the experiments, the color-consistency regularization had a positive effect on unconditionally generated images but did not seem to have a large impact on the conditional image generation. That is why it is not present in our implementation.

2.2.3 Attentional Generative Adversarial Network

Attentional Generative Adversarial Network [20] is an extension of StackGAN++'s architecture (Figure 3). There are three additional models: Text and image encoders(1), Deep Attentional Multimodal Similarity Model (DAMSM)(2), and Attention models(3). By using these three models, AttnGAN can refine each sub-region of the image based on the most related words in the input description. The full architecture of AttnGAN is illustrated in Figure 4.

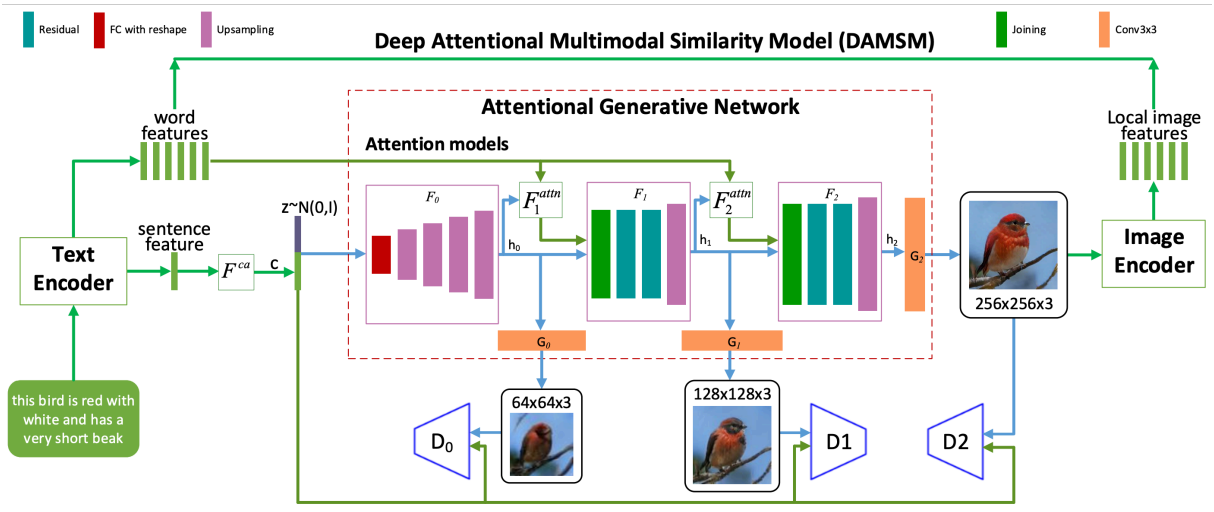


Figure 4: Architecture of the AttnGAN model, taken from the original paper [20].

Text Encoder While the StackGAN model used a pre-trained text encoder model to extract feature from text descriptions, the AttnGAN contains a text encoder as a part of the architecture to train. The proposed text encoder extracts not only the features of a whole sentence but also the features of each word. Text encoder is a bidirectional LSTM (Long short-term memory), which means that it is an LSTM that passes information in both directions (to previous words, as well as to next words). As a result, the LSTM has two hidden states for each word (one per direction). The word features are then simply a concatenation of the two corresponding hidden states. The sentence feature is a concatenation of the two last hidden states in each direction.

Image Encoder The image encoder takes as input an image (256x256), feeds it into the Inception-v3 network and outputs a point in a latent space of similar size to the text encoder's one. This output result is a combination of values from activation functions at multiple layers. Each of these layers contains information of the image sub-regions which can be seen as extraction of local features. This encoded local image feature is then transformed to the word features space via a simple perceptron layer.

DAMSM DAMSM is a model that computes how relevant the generated image is to the given sentence by comparing the local image features to the word features. It is also called the attention model, because when the model calculates the similarity between the features, the relevance of the word features to the features of every sub-region is computed.

Attention models Before the two last feature extractors (h0 and h1 in Figure 4), an attention layer is added, which inputs are the word features and the previous hidden state. Because each column of the hidden state's output contains each sub-regions' image feature, the attention model can learn which word feature is more relevant with which sub-region of the image by computing word context vector for each sub-region. This is how the model learns to generate sub-regions based on the most related words.

AttnGAN is trained by similar loss function as StackGAN++, except for additional DAMSM loss. The weighted similarity score of DAMSM is used as part of the loss. In this project, because we concentrate on refining the StackGAN models with various methods, the AttnGAN model is not implemented.

2.3 GAN Training Stabilization

2.3.1 Wasserstein GAN

As shown in [6], the objective function of the discriminator in the GAN model is linearly proportional to the Jensen-Shannon divergence D_{JS}. This divergence measure has the inherent problem that it gives no valuable feedback when two probability distributions do not intersect. When training a conventional GAN model, D_{JS} often tends to be close to its maximal value $\log 2$. Thus, when trying to approximate the true data distribution p_{data} by training the distribution p_g w.r.t. minimizing $D_{JS}(p_{data}, p_g)$, p_g might not converge to p_{data} [1]. In GAN training, this often results either in mode collapse (discussed in Section 4) or, in the worst case, the generation of the same nonsensical image over and over again.

To avoid this problem, [1] proposes a different loss function, namely the Wasserstein-1 distance:

$$W(p, q) = \inf_{\gamma \sim \Pi(p,q)} \mathbb{E}_{(x,y)\sim\gamma} \|x - y\|. \tag{6}$$

This distance is highly intractable in most cases, but as shown in [19] is equivalent to the equation

$$W(p, q) = K \sup_{\|f\| \leq K} \mathbb{E}_{x\sim p}(f(x)) - \mathbb{E}_{x\sim q}(f(x)), \tag{7}$$

where $\|f\| \leq K$ is the set of all K-Lipschitz continuous functions. This equivalence is called the *Kantorovich-Rubinstein duality*.

Instead of the discriminator, in WGAN the so-called critic is used. Instead of giving a likelihood for the image being real that lies in the interval $[0, 1]$, the critic estimates the Wasserstein-1 distance by maximizing Equation 7. In order to ensure the required Lipschitz continuity, [7] proposes to add an additional penalty term to the loss function:

$$W(p, q) = K \sup_{\|f\| \leq K} \mathbb{E}_{x \sim p}(f(x)) - \mathbb{E}_{\tilde{x} \sim q}(f(\tilde{x})) + \lambda \mathbb{E}_{\hat{x} \sim r}((\|\nabla_{\theta_c} f(\hat{x})\| - 1)^2) \tag{8}$$

2.3.2 Spectral Normalization

Another, more direct approach to enforce Lipschitz continuity is called *spectral normalization* [17]. As the name suggests, spectral normalization works by normalizing all discriminator weights by their respective spectral norm:

$$W_{SN} = \frac{W}{||W||_\infty} = \frac{W}{\sigma_1(W)}. \tag{9}$$

Since the spectral norm of a matrix is by definition equal to its first singular value, we can apply spectral normalization simply by dividing each weight matrix W by its first singular value, which is defined as:

$$\sigma_1(W) = \max_{||v||_2 = 1} ||Wv||_2. \tag{10}$$

To find the first singular value efficiently, the power iteration method [5] is used. Thus, spectral normalization can be applied with a relatively small computational cost.

In contrast to gradient penalty, spectral normalization directly enforces Lipschitz continuity, since the Lipschitz norm for a linear function $g(x) = Wx + b$ is defined as:

$$||g||_{Lip} = sup_x\{||\Delta g(x)||_\infty\} = sup_x\{||W||_\infty\} = ||W||_\infty. \tag{11}$$

As shown by Zhang et al. [8], GAN training with spectral normalization can still be unstable if the generator and discriminator updates are not balanced well. Zhang et al. demonstrated that applying spectral normalization to the generator as well greatly mitigated this problem, and that when used together with TTUR [11] (separate learning rates for generator and discriminator) the quality of generated images increases monotonically throughout the entire training.

2.3.3 Improved Techniques for Training GANs

These techniques are proposed to make the model converge with higher probability. These techniques are motivated by a heuristic understanding of the non-convergence problem rather than by proved mathematical theory [15].

Feature matching Feature matching introduces a new objective for the generator that prevents it from overtraining on the current discriminator. This new cost function is designed to match the statistics of the real data and generated data by matching the expected value of the features on an intermediate layer f(x) of the discriminator

$$||\mathbb{E}_{x\sim p_{data}} f(x) - \mathbb{E}_{z\sim p_z(z)} f(G(z))||. \tag{12}$$

There is no guarantee to reach a fixed point in practice. But, empirical results indicate that the proposed technique is indeed effective.

Mini-batch discrimination This technique allows the discriminator to check multiple data examples in the same batch. It aims to identify the generator's samples that are particularly close together (mode collapse) and penalize this situation. The closeness between samples in a batch is calculated in a mini-batch layer

$$\begin{aligned} &f(x_i) \in \mathbb{R}^A,\ T \in \mathbb{R}^{A\times B\times C},\ M_i = f(x_i)T \in \mathbb{R}^{B\times C}, \\ &o(x_i)_b = \sum_{j=1}^{n} c_b(x_i, x_j) = \sum_{j=1}^{n} \exp(-||M_{i,b} - M_{j,b}||_1) \in \mathbb{R}, \\ &o(x_i) = [o(x_i)_1, o(x_i)_2, ..., o(x_i)_B] \in \mathbb{R}^B. \end{aligned} \tag{13}$$

Then, concatenation of the mini-batch layer's output ($o(x)$) and the intermediate layer's output ($f(x)$) in discriminator is used as a new intermediate layer in discriminator.

Historical averaging Historical averaging adds an L2 term to the cost function:

$$||\theta - \frac{1}{t}\sum_{i=1}^{t} \theta[i]||_2 \tag{14}$$

It penalizes the difference in the model's parameters from the historical average. It may prevent the model from circling around the equilibrium point and act as a damping force to help the model to converge.

One-sided label smoothing Instead of 0 and 1 targets for the discriminator, it uses smoothed values like .9 and .1. It also can reduce the vulnerability of neural networks to adversarial examples. But for this technique, only positive labels are smoothed as .9, while negative labels are left as 0. That is why it is called one-sided smoothing.

Virtual batch normalization Because of batch normalization, the output of a neural network given an input example is highly dependent on several other inputs in the same batch as the input example. Virtual batch normalization is proposed to tackle this problem. We sample a reference batch before the training, and it cannot be changed after sampling. In the forward pass, the reference batch will be used for computing the normalization parameters (μ and σ) instead of the current batch. However, this method can make the model overfit with this reference batch since the same batch is used during the

whole training. To mitigate that the combination of the reference batch and the current batch can be used to compute the normalization parameters.

In Figure 5, all the above techniques except for feature matching are applied to 'Our methods' model. All techniques improve the model, but label smoothing and mini-batch discrimination show significant improvement. That is the reason that we applied these two techniques to our text-image GAN model.

Model	Real data	Our methods	-VBN+BN	-L+HA	-LS	-L	-MBF
Score ± std.	11.24 ± .12	8.09 ± .07	7.54 ± .07	6.86 ± .06	6.83 ± .06	4.36 ± .04	3.87 ± .03

Figure 5: Samples and inception scores with proposed techniques. VBN: Virtual batch normalization, BN: Batch normalization, L: used Label, HA: Historical averaging, LS: Label smoothing, and MBF: Mini-batch discrimination. This figure was taken from the original paper [15].

2.4 GAN Evaluation

To evaluate how well different models of GANs perform, two main scores are used in the literature: Inception Score and Fréchet Inception Distance. We discuss both concepts in this section. Due to time constraints, we did not implement these scores.

2.4.1 Inception Score

The main idea behind Inception score (IS) is that generated images should have two qualities:

1. images are easy to classify;
2. images are diverse.

In [15] where IS was introduced, authors suggested using a pre-trained inception network (e.g. Inception-v3) to classify generated images. The procedure is the following: split the dataset into train and test sets, then generate images from text descriptions from the test set so the generator has not seen the text descriptions before, run a classifier on the generated images and finally calculate IS using output labels.

More precisely, we define (x, y) as pairs of generated images with class labels, i.e. outputs of an inception network. Then $p(y)$ denotes the distribution of class labels, and the conditional distribution $p(y|x)$ can be interpreted as the likelihood that generated image x belongs to class y. We want $p(y|x)$ to have low entropy, i.e. be highly predictable, and $p(y)$ to be close to uniform distribution. It corresponds to Kullback–Leibler divergence between the two distribution $D_{KL}(p(y|x) \,\|\, p(x))$ being as high as possible. Thus, IS can be calculated as

$$\mathrm{IS(G)} = \exp\left(\mathbb{E}_{x \sim p_G} D_{KL}(p(y|x) \,\|\, p(x))\right), \tag{15}$$

where G denotes the generator and p_G is the distribution of generated images.

There are several issues with IS as Shane Barratt et al. pointed out in [2]:

1. IS does not take into account real images;
2. an inception network, which is used to classify generated images, is trained on a dataset different from the dataset on which GAN is trained (ImageNET in the case of Inception-v3);
3. higher score does not necessarily correlate with better results.

2.4.2 Fréchet Inception Distance

Fréchet Inception Distance (FID) was introduced in [11], and since then it has been the second standard score used to evaluate the performance of GANs. FID computes Fréchet distance between distributions of real and generated images. The procedure is similar to IS: first, run an inception network on real images from the test set and on generated images from the test text descriptions, then calculate distributions of both output labels (we assume Gaussian distributions for both) and calculate the Fréchet distance between the two distributions:

$$\text{FID}(y, r) = \|\mu_r - \mu_y\|_2^2 + \text{Tr}(\Sigma_r + \Sigma_y - 2(\Sigma_r\Sigma_y)^{\frac{1}{2}}), \tag{16}$$

where $\mathcal{N}(\mu_r, \Sigma_r)$ and $\mathcal{N}(\mu_y, \Sigma_y)$ are Gaussian distributions of labels of real and generated images, respectively.

There are some issues with FID as well:

1. again, an inception network trained on a different dataset is used;
2. since we assume Gaussian distributions, a large sample size is needed; 30000 images is the number commonly used.

3 Implementation

3.1 System Architecture

This chapter is dedicated to the architecture of our working process. It was extremely important for us to use as many available computational resources as possible. Simultaneously, the usage should be convenient for developers, everyone should work in the same environment to prevent unnecessary issues, for instance, possible bugs during migrating of the code from one machine to another. Spending some time for creating such a sophisticated system at first helped us to speed up the process of achieving the main goal of the project. Our architecture scheme, which consists of training and frontend phases, is shown in Figure 6. All components will be shortly discussed in the following sections.

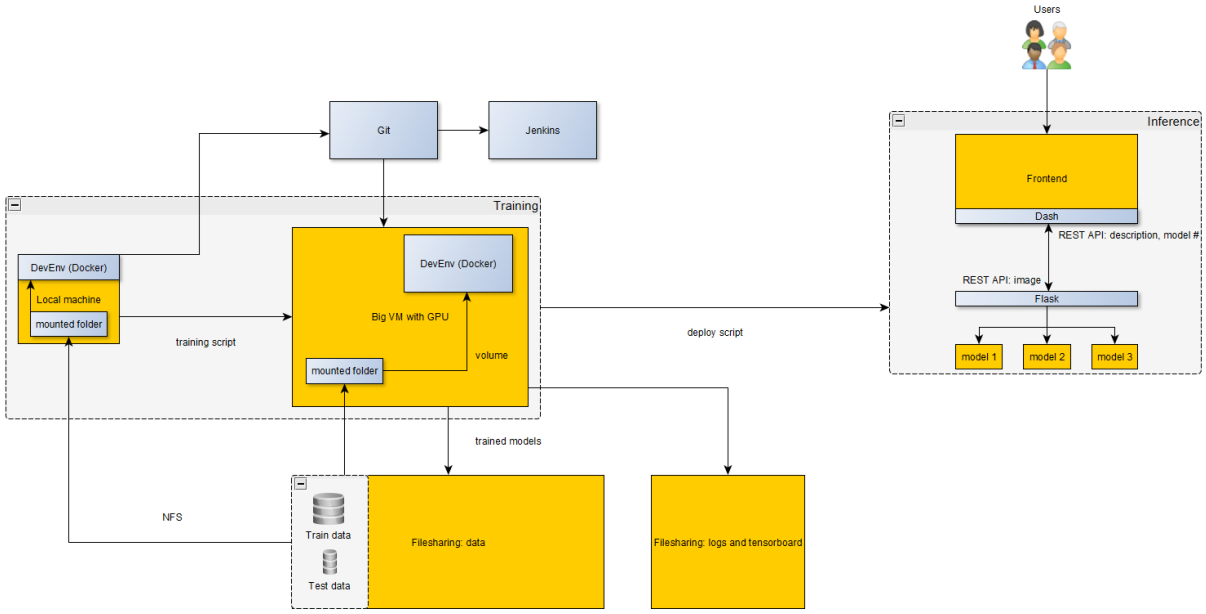


Figure 6: System architecture.

3.2 Application

A demo application is developed to show our models' generated samples with arbitrary text descriptions. In this section, tools and ways used to build and run the demo application are introduced. As we already mentioned in the Introduction, the demo application is running on `http://10.195.1.122:8050/`.

3.2.1 Models

In the demo application, many models are prepared to generate an image. For AttnGAN, pre-trained text encoder and generator models (Pytorch) with 200 epochs are used. This model is a reference model to compare the results to one of our models. For StackGAN and StackGAN++, as mentioned above, pre-trained char-CNN-RNN text encoder model is used. To generate images with arbitrary descriptions, same text encoder model (Torch7)

should be used. Except for these models, all the other models (Tensorflow) are implemented and trained by ourselves. At this moment, all models generate the final image directly, but the StackGAN model shows samples by stage. WGAN model generates only images of size 64x64, but other models can generate images of size 256x256.

3.2.2 Dash

In the beginning, we settled to make a web application in order to show the results of the project. Since everything was going to be coded in Python, it was logical to use Dash by Plotly (`https://plot.ly/dash`) for this purpose. Dash is a framework for building analytical web applications. You do not need to use any Javascript, everything can be done in Python. Nevertheless, there are still plenty of interactive components available such as radio buttons (with them the users can choose which model they would like to use), image and text fields (where the description of the image can be inserted), etc. The example was shown in Figure 1.

3.2.3 Flask

Potentially, we would like to have an opportunity to locate frontend code (Dash part) and trained models, which are used for generating images inside the application, not on a single VM, but even on different physical machines. (However, for the project purposes, we still put it together on the single VM, since it was not so important to follow the differentiation plan till the end. But it was vital to remain the opportunity to split them up among different machines in the future). To achieve this goal we needed a tool for communication between frontend and the models, and we agreed to use Flask (`https://palletsprojects.com/p/flask/`) for it.

Generally, Flask is a framework that describes how a web server communicates with web applications. In our case, the trained models play the role of the server. The full scheme could be described as following: whenever a user writes the description and choose the desired model, Dash sends REST API request to Flask. Flask handles it and has the concrete model to do the task. After it finishes, the result as an array is sent back to the frontend side.

3.2.4 Docker

Docker is a platform for developers to develop, deploy, and run applications with containers. A container is launched by running an image. An image is an executable package that includes everything needed to run an application - the code, libraries, environment variables and configuration files. The full official tutorial of using Docker could be found under `https://docs.docker.com/`. One helpful functionality providing by Docker is volumes. They give the opportunity to share data between Docker container and the host machine, to store and keep it even after the container finishes its job.

During the developing of the demo application, we used Docker containers for both Dash and Flask scripts, i.e. we had separate containers for each of them. It was done to prevent any errors caused by the discrepancy of the dependencies. Actually, we used the containers to create the proper environment, and then shared Flask and Dash scripts with them via

volumes. Also, we used volumes to give the Flask container access to the trained models. Both of these options helped us to make changes into the code or replace one model with the other one without rebuilding everything.

When somebody runs two or more Docker containers on the same machine and they depend on each other (this is our case with the Flask and Dash containers), then it is necessary to carefully tune the communication between them. Hopefully, it could be easily achieved by using Docker compose: it is completely enough to declare which containers are connected with each other in the special configuration file. To read more about Docker compose following link `https://docs.docker.com/compose/` might be helpful.

3.2.5 Deployment

To deploy the application, there are two steps. First, clone the latest repository and build the docker image. Specifically, docker-compose is used for communication between two docker containers (dash and flask). When you build the docker image for the application, all the dependencies to run the application will be installed. Additionally, our source codes are installed as a python library for unified and ordered import path. In the second step, the latest codes and checkpoints of the models will be downloaded inside the docker volume, connected to the docker image already built. Such volume property was applied to avoid rebuilding all the dependencies every time, which could be time-consuming. Two deploy scripts are written for each step, that makes demo application be executed by just running one script in one of the authorized local machines.

3.3 Development Environment

To ensure a fast development process, easy collaboration, and good code quality, we set up a sophisticated development workflow, which corresponds to the architecture components on the left side of Figure 6. In the following, we will briefly explain each component and its purpose.

3.3.1 Local Machines

The local machines represent each of the developer's personal computer, where most of the development takes place. Because each developer has a different system, which could lead to code compatibility issues, we considered using dedicated CPU VMs for the development instead. However, we decided against this idea, because developing on a remote machine requires an internet connection and does not support some integrated development environments, thus, slowing down development. Instead, we decided to use Docker on all machines to ensure that our code is machine independent, and to mount training data from an NFS-server so everyone is using the exact same data.

3.3.2 GPU VM

Since generative adversarial networks are some of the most complex and resource-intensive machine learning models, training such models on our local machines is not feasible. Luckily, we had access to a large VM with GPU support, which consisted of 20 virtual CPUs and two Nvidia Tesla V100 GPUs. We used this VM to train all of our models.

Similar to the local machines, we also used Docker and NFS mounting on the big VM, to ensure that the code dependencies and data are the same. The GPU VM was graciously provided by the Leibniz Rechenzentrum (LRZ), which we would like to thank again.

3.3.3 ML Database and NFS Mounting

To have all data in one place we created Machine Learning (ML) Database. It helped us to achieve two main goals. First of all, it was crucial to have access to already trained models even after finishing the project. All additional information like hyperparameters that were applied to the models should also be easily available. So after each training, we copied our results to this machine.

Secondly, we agreed that it would be convenient to have all training data on one VM and mount it to all local machines and GPU VM then. For this purpose, NFS mounting was used. In this case, nobody had to download data by itself and made individual preprocessing - all these preparations were done only once and, afterwards, were accessible for all developers.

3.3.4 Tensorboard VM

Moreover, we decided to have an additional VM to store logs from the training process and run Tensorboard (`https://www.tensorflow.org/tensorboard`) on it. Tensorboard is a convenient tool, enables to nicely visualize all statistical information about the training - behaviour of the loss functions, a graph of the model or how generated images are changing during the time. It helped us a lot in analyzing whether everything goes right during models developing. Furthermore, it was very constructive to use it for the presentations of our progress, since the information about the whole training process could be shown in a very compact way.

The reason why we decided to create a second VM and store everything on ML Database is memory capacity. It was of paramount importance to have at any time enough space to store the checkpoints of trained models in ML Database. However, the size of the Tensorboard logs is quite large, so it could produce some risks of Out Of Memory errors if we put everything on the single VM.

3.3.5 Git

To collaborate effectively, we used Git (LRZ Gitlab) to version control our code. To ensure that the latest version of our code is working, we only developed on feature branches, which we then merged to master via merge requests after a feature is completely implemented and tested. For transparency, we used an agreed naming convention for the branches, which is shown in Table 1.

3.3.6 Jenkins

In addition to manual code checks during merge request review, we also use the continuous integration tool Jenkins to perform automated code checks on our git repository. For this purpose, we have set up Jenkins on a dedicated VM. Whenever code is pushed to

Table 1: Git branch naming convention.

Branch Name	Purpose
feature/backend	development of backend features (models, ...)
feature/frontend	development of frontend features (dash, demo, ...)
bug	bugfixes (non-critical bugs)
fix	hotfixes (critical bugs)

our Gitlab repository, Gitlab notifies our Jenkins server, which is automatically cloning, building and testing the code within Docker. There are four different stages (types of code checks) that Jenkins performs, which are listed in Table 2.

Table 2: Jenkins stages.

Jenkins Stage	Purpose
build	builds the docker container and compiles the project
test	executes all unit tests in the project
lint	performs linting checks (code quality checks)
black	executes python package black to ensure that code is formatted well

Whether all of the Jenkins stages passed or not will then be shown via a green or red indicator next to the corresponding commit in Gitlab, and a summary of the Jenkins results is posted to our dedicated Jenkins channel in Slack. Only when all code checks pass can a branch be merged to master. This ensures that new functionality does not break existing code and does comply with common coding guidelines.

For more information on Jenkins, please refer to `https://jenkins.io/` or to general continuous integration and continuous deployment literature.

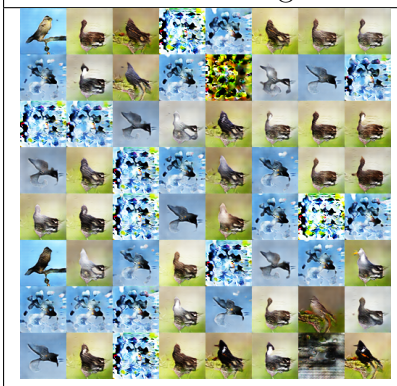
4 Experiments

In our experiments, we decided to use publicly available CUB-200-2011 dataset, which contains 11,788 images of birds from 200 classes (also see Appendix for some results on Oxford-102 flowers dataset). There are 5 descriptions available for each image. We split the dataset into train and test datasets containing 8,855 and 2,933 respectively.

For training StackGAN and StackGAN++, we used the same hyperparameters as in [10] and [9]: learning rate of 0.0002 for both discriminator and generator networks and Adam optimizer with beta decay of 0.5. We trained both Stage I and Stage II, and StackGAN++ for 600 epochs. Moreover, we trained WGAN for 350 epochs.

First, we trained StackGAN Stage 1 and Stage 2, and WGAN. Our results are presented in Table 3:

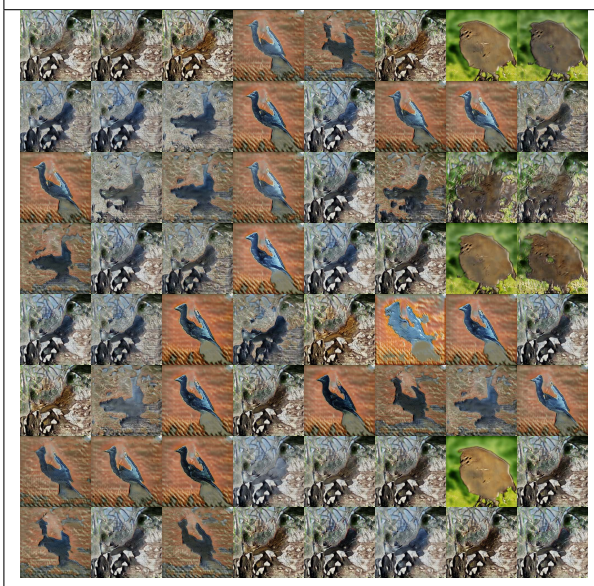
Table 3: StackGAN Stage 1, Stage 2 and WGAN results.

StackGAN Stage 1	StackGAN Stage 2	WGAN
		

Note that the images generated by StackGAN Stage 1 and WGAN are low resolution images (64x64), while the images generated by StackGAN Stage 2 are of higher resolution (256x256). All generated images correspond to different text descriptions, e.g. the upper left image corresponds to "light tan colored bird with a white head and an orange beak" and the lower right to "the bird has a very tiny black bill as well as a black wingbar". The images generated by StackGAN model match the text descriptions worse than the images generated by the WGAN model. Also, as reported in [10], mode collapse occurred. As mentioned in Section 2.2.2, mode collapse means that the generator produces only a limited variety of images. The goal of the generator is to produce an image out of noise that would fool the discriminator, i.e. to produce an output $x^* = G(z)$ s.t. $x^* = \text{argmax } D(x)$ (for a fixed discriminator D). Thus, we never explicitly force the generator to produce a great variety of images, rather to find a point that maximizes the discriminator loss. [13] points out that it is one of the main reasons why mode collapse happens. It is also the main motivation behind different objective functions, e.g. as in WGAN [1]. Indeed, WGAN showed better results and no mode collapse. Next, we decided to try different approaches to tackle mode collapse discussed in Section 2.3.

Since [9] reported great improvements on mode collapse, we first decided to implement StackGAN++. We trained it with different batch sizes (8 and 32). Results are presented in Table 4. Note that all individual images have resolution 256x256.

Table 4: StackGAN++ results.

batch size 8	batch size 32
	

Strong mode collapse is still present and, as in StackGAN version 1, the images do not necessarily capture features from text descriptions. Since the results with larger batch size are better, next we decided to focus on 1. retraining StackGAN version 1 with bigger batch size; 2. implementing spectral normalization (SN) and mini-batch discrimination (MBD) discussed in Section 2.3. Our results are presented in Table 5 and Table 6.

Table 5: StackGAN Stage 1 results with batch size 8.

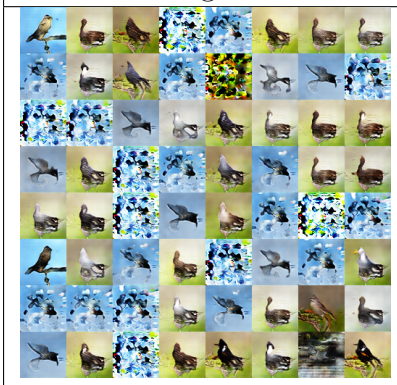
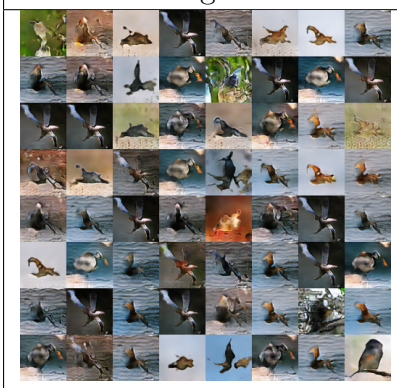
Stage 1	Stage 1 with SN only in D	Stage 1 with SN in G and D
		

Table 6: StackGAN Stage 1 results with batch size 64.

Stage 1	Stage 1 with MBD	Stage 2 with MBD
		

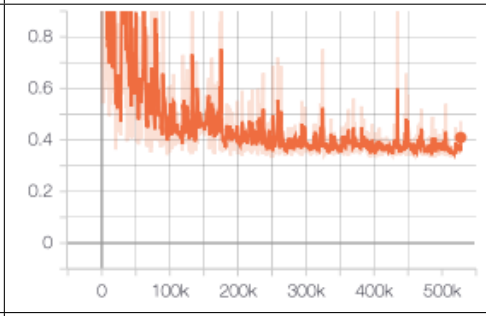
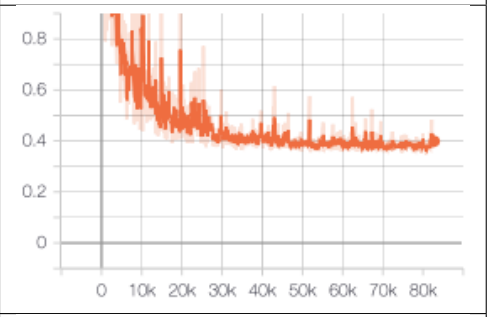
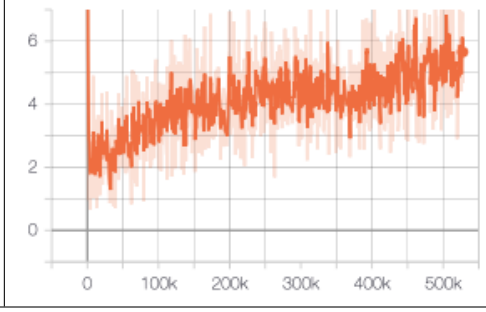
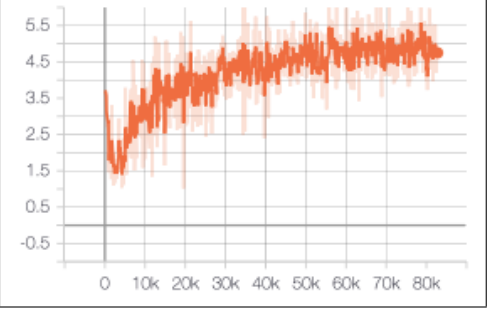
Increased batch size had a significant effect on the results, while the effects of spectral normalization and mini-batch discrimination are not conclusive.

Even though loss functions of GANs are known to be hard to interpret [6], sometimes it is useful to take a look at discriminator and generator losses to notice tendencies. Note that as pointed out in our reference implementation [3], because of problems with saturated gradients, we minimize slightly modified objective functions

$$\min_G -\mathbb{E}_{z\sim p_z(z)} \log D(G(z)),$$
$$\min_D -\mathbb{E}_{x\sim p_{data}(x)} \log D(x) - \mathbb{E}_{z\sim p_z(z)} \log(1 - D(G(z))).$$

Some examples of loss functions are presented in Table 7.

Table 7: Loss functions for Stage 1 with different batch sizes.

	bath size 8	batch size 64
discriminator		
generator		

The discriminator loss seems to converge in both cases, but the generator loss for Stage 1 with batch size 8 increases, which results in bad quality of generated images. On the other hand, loss function with batch size 64 seems to converge, which corresponds to better results.

In Table 8, the time needed to train different models on one GPU (Nvidia Tesla V100) are presented. Because of time constraints, we did not train other models. Next steps would be to implement and train both stages of StackGAN and StackGAN++ with SN and MBD combined with larger batch size.

Table 8: Time to train StackGAN and StackGAN++.

	StackGAN Stage 1	StackGAN Stage 2	StackGAN++
batch size 8	9,5 hours	3 days 3 hours	5 days 6 hours
batch size 64 for Stage 1 and Stage 2, and 32 for StackGAN++	4,5 hours	1 day and 21 hours	3 days 15 hours

5 Conclusion

By the end of our project, we can summarize that the following goals have been achieved. First of all, we conducted extensive research about GANs, how they work, which models already exist for generating realistic images based on text descriptions and how to train them more efficiently.

Subsequently, we decided to start with the implementation of StackGAN and to train it afterwards. Because mode collapse occurred extremely often during the training, we applied a couple of techniques (spectral normalization, mini-batch discrimination, label smoothing) to mitigate it. Additionally, we implemented StackGAN++, which is the second version of the previous model and has a more sophisticated architecture. It was expected to improve the variety and quality of images.

Meanwhile, to present our results, we built the web application, which is ready to be delivered and deployed in a real-world scenario. During the implementation process, we employed state-of-the-art software development methods to ensure high code quality and easy maintainability and extensibility.

In order to improve the generated images and make the analysis of that topic complete, the following steps could be taken in the future. Primarily, it is worth to implement AttnGAN model, which is generally based on already built StackGAN++. In this way, only a few components such as DAMSM and attention models should be added to the existed version. Furthermore, to make the comparison of models stricter and more scientific, evaluation metrics might be applied. We have already made a research about the most common and useful ones, consequently, our suggestions would be to calculate Inception Score and Fréchet Inception Distance of our models.

Contributors

Throughout this lab, we tried to split the tasks in a way that allows each team member to get involved in as many parts of the project as possible. In the following, we provide a short overview of who worked on which tasks, and who wrote which parts of this report.

Project Contributions

- Anastasia Litinetskaya: research (StackGAN, StackGAN++, WGAN, Evaluation Metrics), training (StackGAN, StackGAN++, WGAN), setting up and maintaining GPU VM, Tensorboard VM.
- Felix Altenberger: research (AttnGAN, spectral normalization, diverse GAN applications), implementation (spectral normalization, StackGAN++), setting up and maintaining Jenkins, code refactoring and restructuring.
- Jongwon Lee: research (AttnGAN, Improved Techniques for Training GANs), implementation (mini-batch discrimination, inference), training (mini-batch discrimination), application (adding models to the demo, deployment scripts).
- Vadim Goryainov: research (StackGAN, StackGAN++, WGAN), implementation (StackGAN++), code adjustments (StackGAN), adjustments in data preprocessing.
- Vitalii Mozin: system architecture, Flask, Dash, NFS mounting, setting up and maintaining Fileshare VM, Docker, Docker Compose, implementation (StackGAN++), training (spectral normalization), application (general set up).

Final Report

- Anastasia Litinetskaya: sections 2.4 and 4
- Felix Altenberger: sections 1, 2.1, 2.3.2, 3.3 (apart from 3.3.3, 3.3.4)
- Jongwon Lee: sections 2.2.3, 2.3.3, 3.2.1 and 3.2.5
- Vadim Goryainov: sections 2.2.1, 2.2.2 and 2.3.1
- Vitalii Mozin: sections 3.1, 3.2 (apart from 3.2.1), 3.3.3, 3.3.4 and 5

Bibliography

[1] Martin Arjovski, Soumith Chintala, and Léon Bottou. "Wasserstein Generative Adversarial Networks". In: *International Conference on Machine Learning* (2017).

[2] Shane Barratt and Rishi Sharma. "A Note on the Inception Score". In: *arXiv e-prints* (2018). arXiv:1801.01973.

[3] Cristian Bodnar. "Text to Image Synthesis Using Generative Adversarial Networks". In: *CoRR* (2018).

[4] Qifeng Chen and Vladlen Koltun. "Photographic image synthesis with cascaded refinement networks". In: *Proceedings of the IEEE International Conference on Computer Vision* (2017).

[5] Gene Golub and Henk Van der Vorst. "Eigenvalue computation in the 20th century". In: *Numerical analysis: historical developments in the 20th century* (2001).

[6] Ian Goodfellow et al. "Generative Adversarial Nets". In: *Advances in neural information processing systems* (2014).

[7] Ishaan Gulrajani et al. "Improved Training of Wasserstein GANs". In: *Computing Research Repository* (2017).

[8] Zhang Han et al. "Self-attention generative adversarial networks". In: *arXiv e-prints* (2018). arXiv:1805.08318.

[9] Zhang Han et al. "Stackgan++: Realistic image synthesis with stacked generative adversarial networks". In: *IEEE transactions on pattern analysis and machine intelligence 41, no.8* (2018).

[10] Zhang Han et al. "Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks". In: *Proceedings of the IEEE International Conference on Computer Vision* (2017).

[11] Heusel Martin et al. "GANs trained by a two time-scale update rule converge to a local nash equilibrium". In: *Advances in Neural Information Processing Systems* (2017).

[12] Mirza Mehdi and Simon Osindero. "Conditional generative adversarial nets". In: *arXiv e-prints* (2014). arXiv:1411.1784.

[13] Luke Metz et al. "Unrolled Generative Adversarial Networks". In: *arXiv e-prints* (2016). arXiv:1611.02163.

[14] Isola Phillip et al. "Image-to-image translation with conditional adversarial networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017).

[15] Tim Salimans et al. "Improved Techniques for Training GANs". In: *Advances in neural information processing systems* (2016).

[16] Reed Scott et al. "Generative adversarial text to image synthesis". In: *arXiv e-prints* (2016). arXiv:1605.05396.

[17] Miyato Takeru et al. "Spectral normalization for generative adversarial networks". In: *arXiv e-prints* (2018). arXiv:1802.05957.

[18] Wang Ting-Chun et al. “High-resolution image synthesis and semantic manipulation with conditional GANs”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition* (2018).

[19] Cédric Villani. “Optimal Transport: Old and New”. In: *Grundlehren der mathematischen Wissenschaften* (2009).

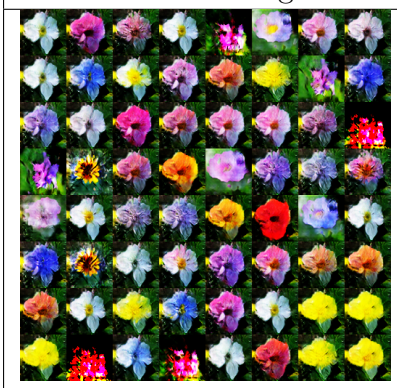
[20] Tao Xu et al. “AttnGAN: Fine-Grained Text to Image Generation with Attentional Generative Adversarial Networks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018).

Appendix

Training on Flowers Dataset

Before we decided to focus on the birds dataset, we run experiments on Oxford-102 flowers dataset. It contains 8,192 images and each has 10 descriptions. The train and test sets contain 7,034 and 1,155 images respectively. The results are presented in 9.

Table 9: StackGAN Stage 1, Stage 2 and WGAN results on flowers.

StackGAN Stage 1	StackGAN Stage 2	WGAN
		

The upper left images correspond to ”a flower with one large white circular petal and three other yellow folded ones” and the lower right to ”this flower is yellow in color, with petals that are wavy and ruffled”.

StackGAN++ Image Sizes

In StackGAN version 1 implementation, Stage II is trained after Stage I finished training. In contrast, in StackGAN++ all three pairs of generators and discriminators are trained at the same time. It allows seeing consecutive improvements in the quality and resolution of generated images.

Table 10: Images generated by StackGAN++ (actual sizes).

G_0 (64x64)	
G_1 (128x128)	
G_1 (256x256)