Warsaw University of Technology

FACULTY OF
ELECTRONICS AND INFORMATION TECHNOLOGY

Institute of Control and Computation Engineering

Bachelor's diploma thesis

in the field of study Computer science
and specialisation Information and Decision Systems

Generating new types of spiders using Generative Adversarial Network

Valery Burau
student record book number 288128

thesis supervisor
Prof. dr hab. Andrzej Pacut

WARSAW, 2021

Abstract

This thesis is based on the presentation of several ideas. The first of them is the structure of the GAN network and its main characteristics. The second idea is what subtype of the GAN network is used to solve an assigned task: generating new types of spiders. Main characteristics of this GAN subtype and how this network is trained based on the used algorithms and functions. The last idea of this thesis is what results have been achieved, what attempts have been made to get the best results and what conclusions can be drawn from the presented results.

The first chapter presents main goals and assumptions of the thesis.

The second chapter provides an introduction to the GAN.

The third chapter shows the list of used abbreviations.

The fourth chapter gives the information about the selected GAN subtype for the thesis problem and the areas, in which the expected results can be used.

The fourth chapter introduces the algorithms of convolutional and convolutional transposed layers algorithms that are commonly used for the DCGAN (subtype of GAN) structure.

The fifth chapter illustrates the structure of the discriminator and generator in the DCGAN model.

The sixth chapter presents the activation functions used in the model.

The seventh chapter gives the information about used loss function in the model

The eighth chapter presents the optimization functions used in the model.

The ninth chapter gives information about which libraries have been used to build the network and provides information about the dataset preprocessing part.

The tenth chapter gives the information how the training process is going in DCGAN model.

The eleventh chapter shows the achieved results and their analysis.

The twelfth chapter introduces the inception model and presents the achieved results and their analysis for this model.

The thirteenth chapter presents conclusions / summary of the made work and plans for future work.

The fourteenth chapter is a list of used literature.

The fifteenth chapter is a list of used references.

The sixteenth chapter shows the list of used abbreviations.

Key words: GAN, DCGAN, Inception model, Convolutional layers, Convolutional transposed layers.

Tytuł pracy

Generowanie nowych typów pająków za pomocą sieci GAN.

Streszczenie pracy

Niniejsza praca inżynierska skupia się na przedstawieniu kilku różnych koncepcji. Pierwszą z nich to jest struktura sieci GAN i jej główne cechy. Druga koncepcja dotyczy tego, jaki podtyp sieci GAN jest używany do rozwiązania przydzielonego zadania: generowania nowych typów pająków. Główne cechy tego podtypu GAN i sposób uczenia tej sieci na podstawie używanych algorytmów i funkcji. Ostatnim zamysłem tej pracy inżynierskiej jest przedstawienie osiągniętych wyników, jakie podjęto próby dla uzyskania najlepszych wyników oraz jakie wnioski można wyciągnąć z przedstawionych wyników.

W pierwszym rozdziale przedstawiono główne cele i założenia pracy inżynierskiej.

Rozdział drugi zawiera wprowadzenie do sieci GAN.

Rozdział trzeci zawiera informacje o wybranym podtypie GAN dla problemu pracy dyplomowej oraz o obszarach, w których można wykorzystać oczekiwane rezultaty..

Rozdział czwarty przedstawia algorytmy splotowych i splotowych transponowanych warstw, które są powszechnie używane w strukturze DCGAN (podtyp GAN).

Rozdział piąty ilustruje budowę dyskryminatora i generatora w modelu DCGAN.

Rozdział szósty przedstawia funkcje aktywacyjne zastosowane w modelu.

W rozdziale siódmym zawiera się informacja o zastosowanej funkcji straty w modelu.

Rozdział ósmy przedstawia funkcje optymalizacji stosowane w modelu.

W dziewiątym rozdziale przedstawiono informacje o użytych bibliotekach dla zbudowania sieci oraz o części dotyczącej wstępnego przetwarzania zbioru danych.

Rozdział dziesiąty zawiera informacje o przebiegu procesu trenowania w modelu DCGAN.

Rozdział jedenasty przedstawia osiągnięte wyniki i ich analizę.

Rozdział dwunasty przedstawia inception model oraz przedstawia uzyskane wyniki i ich analizę dla tego modelu.

Rozdział trzynasty przedstawia wnioski / podsumowania wykonanych prac oraz plany przyszłych prac.

Rozdział czternasty to spis używanej literatury.

Rozdział piętnasty to lista wykorzystanych odniesień.

Rozdział szesnasty przedstawia listę użytych skrótów.

Słowa kluczowe: GAN, DCGAN, Inception model, Splotowe warstwy, Splotowe transponowane warstwy.

Warszawa, 29.01.2021
miejscowość i data
place and date

Valery Burau
imię i nazwisko studenta
name and surname of the student

288128
numer albumu
student record book number

Informatyka
kierunek studiów
field of study

OŚWIADCZENIE

DECLARATION

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.
Under the penalty of perjury, I hereby certify that I wrote my diploma thesis on my own, under the guidance of the thesis supervisor.

Jednocześnie oświadczam, że:
I also declare that:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- *this diploma thesis does not constitute infringement of copyright following the act of 4 February 1994 on copyright and related rights (Journal of Acts of 2006 no. 90, item 631 with further amendments) or personal rights protected under the civil law,*
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- *the diploma thesis does not contain data or information acquired in an illegal way,*
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- *the diploma thesis has never been the basis of any other official proceedings leading to the award of diplomas or professional degrees,*
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,

- *all information included in the diploma thesis, derived from printed and electronic sources, has been documented with relevant references in the literature section,*
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.
- *I am aware of the regulations at Warsaw University of Technology on management of copyright and related rights, industrial property rights and commercialisation.*

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej
w module APD systemu USOS są identyczne.
I certify that the content of the printed version of the diploma thesis, the content of the electronic version of the diploma thesis (on a CD) and the content of the diploma thesis in the Archive of Diploma Theses (APD module) of the USOS system are identical.

Valery Burau
czytelny podpis studenta
legible signature of the student

Table of contents

1. Aim & main assumptions of the thesis 10

2. GAN 10

2.1 Basic idea behind GANs 10

2.2 Characteristics [2] 11

2.3 Usage 11

2.4 Steps to train a GAN [3] 11

2.5 Types of GANs 12

3. Selected GAN subtype & Usage of expected results 13

4. Main algorithms in convolutional & convolutional transpose layers (DCGAN structure) 14

4.1 Main idea behind convolutional & convolutional transpose layers 14

4.2 Discriminator algorithms in convolutional layers [8] 14

4.2.1 Discriminator idea in convolutional layers 14

4.2.2 Stride in a convolutional layer [7][8] 15

4.2.3 Filters (=Feature detectors; =Kernels) in a convolutional layer 15

4.2.4 Padding in a convolutional layer 16

4.2.5 Pooling in a convolutional layer 17

4.3 Generator algorithms in convolutional transpose layers [8] 17

4.3.1 Generator idea in convolutional transpose layers 17

4.3.2 Stride in a convolutional transpose layer 17

4.3.3 Filters (=Feature detectors; =Kernels) in a convolutional transpose layer ... 18

4.3.4 Padding in a convolutional transpose layer 20

5. Code: DCGAN structure & illustration 21

5.1 Selected GAN model – DCGAN 21

5.1.1 DCGAN introduction 21

5.1.2 Discriminator & Generator main aspects [13] 21

5.2 DCGAN: Discriminator part 21

5.2.1 Discriminator introduction 21

5.2.2 Convolutional layer – main steps in theory 21

5.2.3 Discriminator structure in code 22

5.2.4 Discriminator illustration part – example 23

5.3 DCGAN: Generator part 24

5.3.1 Generator introduction 24

5.3.2 Convolutional transpose layer – main steps in theory 24

5.3.3 Generator structure in code 24

5.3.4 Generator illustration part – example 25

6. Activation functions 26

7. Used loss function 27

7.1 Introduction [15][16] 27

7.2 Implementation 27

8. Training process – optimizers 28

8.1 Introduction [18] 28

8.2 Optimizers [18] 28

8.2.1 Mini-batch Gradient Descent 28

8.2.2 RMSprop 28

8.2.3 Adam (Adaptive Moment Estimation) 29

9. Code: Dataset preprocessing part 30

9.1 Programming language, libraries & environments [19][20][21][22] [23] 30

9.2 Dataset 30

9.2.1 Used datasets & preprocessing part 30

9.2.2 Characteristics of the final dataset [24][25][26][27][28] 31

10. Code: Training process 33

11. Results 34

11.1 New types of spiders 34

11.1.1 Generated spiders – Result №1 (*Influence of optimizers*) 34

11.1.2 Generated spiders – Result №2 (*Influence of hyperparameter*) 37

11.1.3 Generated spiders – Result №3 (*Influence of batch size*) 39

11.1.4 Generated spiders – Result №4 (*batch size=4096*) 43

11.1.5 Generated spiders – Result №5 (different number generator/discriminator updates) 44

11.1.6 Characteristics of new generated spiders 45

11.1.7 Generated spiders – Summary 47

11.2 Loss results during the training process 47

11.2.1 Loss of different optimizers – Result №1 47

11.2.2 Loss for different beta1 parameter - Result №2 49

11.2.3 Loss for different batch size - Result №3 51

11.2.4 Generator & Discriminator output for different batch size - Result №3 55

11.2.5 Loss for batch size=4096 (2300 epochs) - Result №4 57

11.2.6 Loss for different number generator/discriminator updates - Result №5 58

11.3 Info table 59

12. Inception model 60

12.1 Introduction [31] 60

12.1.1 What problems arise in usual structure 60

12.1.2 Some inception ideas & their purpose 60
12.2 Inception model – used structure 62
12.3 Results of inception model 65
12.4 Losses of inception model 65
12.4.1 Generator & Discriminator loss in inception model for both optimizers 65
12.4.2 Generator & Discriminator output in inception model for both optimizers. 68
13. Conclusions & future work 68
14. Additional literature 68
15. References 69
16. Abbreviations 70

1. Aim & main assumptions of the thesis

Aim:

The main goals of this thesis is:

1. To introduce the work of the GAN network, which is the main idea and characteristics behind this network.
2. To determine what subtype of the GAN network could be used to solve these following problem such as generating new types of spiders, a thorough study of the selected subtype and the necessary information of this network. Also to demonstrate how the computational processes of the interested subtype occur from the mathematical background.
3. To analyze the results and make conclusions from the made work.

Main assumptions to achieve:

- The newly created spiders should look as realistic as possible.
- The new generated spiders should look like a new kind of spiders or similar to a training images. Due to about 10 different spider species in the training dataset, there is a chance that neural network can create a new 'mixed' versions (=kind) of spiders.

2. GAN

The task of this chapter is to present a GAN model.

Artificial intelligence gains great popularity and necessity for everyday life. Lots of industries start to incorporate AI into their projects. Currently, neural networks are used for various tasks such as classification, prediction, precision, video and image generation, etc. Generating new images is one of aforementioned problems. GAN is regarded as the best and most popular model for this kind of task. It's the reason why this model has been used in my dissertation.

Generative Adversarial Network (GAN) was invented by Ian Goodfellow and colleagues in 2014.

GANs are a framework for teaching a DL model to capture the training data's distribution. New data can be generated from that same distribution. [1]

2.1 Basic idea behind GANs

There are two most important parts while creating this type of network:

- Generator: replicates real data to produce fake data.
- Discriminator: distinguishes real data from fake data.

The job of the generator is to replicate 'fake' data that looks similar to the training data. The job of the discriminator is to look at input and output whether or not it is a real/fake data from the generator. During training, the generator is constantly trying to outsmart the discriminator by generating better and better fake data's, while the discriminator is working to become a better detective and correctly classify the real and fake data. The equilibrium of this game is when the generator is generating perfect fakes that look as if they came directly from the training data, and the discriminator is left to always guess at 50% confidence that the generator output is real or fake.

The typical representation of GAN is Original Vanilla GAN. (Section 2.5).

2.2 Characteristics [2]

- Class of machine learning systems.
- Given a training set, the technique learns to generate new data with the same statistics as the training set.
- The *generative network* generates candidates while the *discriminative network* evaluates them.
- The generator trains based on whether it succeeds in fooling the discriminator.
- The generator is typically a deconvolutional neural network, and the discriminator is a convolutional neural network.

2.3 Usage

- Creating fake and realistic photos (e.g. *portraits, landscapes and album covers)* and videos.
- Increasing amount of data.
- Precision of some images (e.g. *improvement of astronomical images and simulation gravitational lensing for dark matter research*).
- Reconstruction 3D models of objects from images and model patterns of motion in video.
- Aging face photographs to show how an individual's appearance might change with age.
- Visualization of the effect that climate change will have on specific houses.
- Model called Speech2Face can reconstruct an image of a person's face after listening to their voice.
- Generating new ways to solve various problems (e.g. in construction) that people are even not able to foresee.
- Generating fake voices.

2.4 Steps to train a GAN [3]

1. Define the problem (e.g. synthesize images from a caption, audio synthesis from sentences, so and so).
2. Define GAN architecture (depends on the complexity of the problem, e.g.: multi-layer perceptron, neural network or a simpler model).
3. Train the discriminator to distinguish “real” Vs “fake” data (data labelled as real data, generated data labelled as fake data).
4. Train the generator to synthesize data (parameters of the generative model should be modified to maximize a loss of the discriminator).
5. Repeat steps 3 and 4 for numerous times until the goals will be achieved (N

epochs).

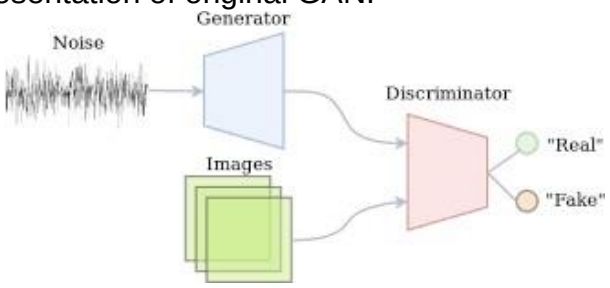
6. Final result: the discriminator will not be able to distinguish the real and the fake samples.
7. Once training is complete, synthesize data from the generator.

2.5 Types of GANs

There are various types of GANs. Each model is usually used to solve a specific problem. The basic idea (Section 2.1) is the same for all GANs, but the structure is different for all models.

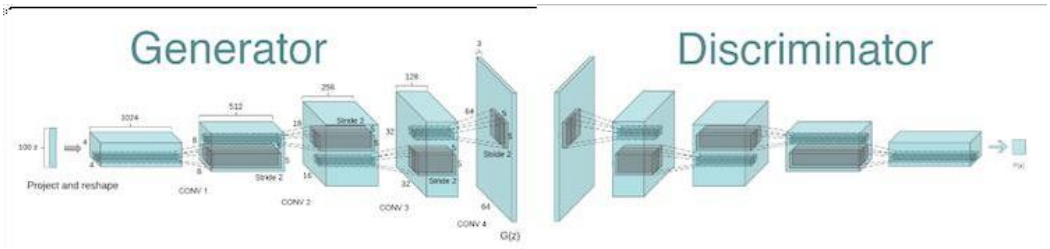
The following popular GAN models are represented below to see the main difference in structures:

1. Original Vanilla GAN
   - A representation of original GAN.



Picture 1: Original Vanilla GAN [4]

2. Deep Convolutional GAN (DCGAN)
   - CNNs used in unsupervised learning.
   - Generators are Deconvolutional Neural Networks.
   - Discriminators are CNNs.

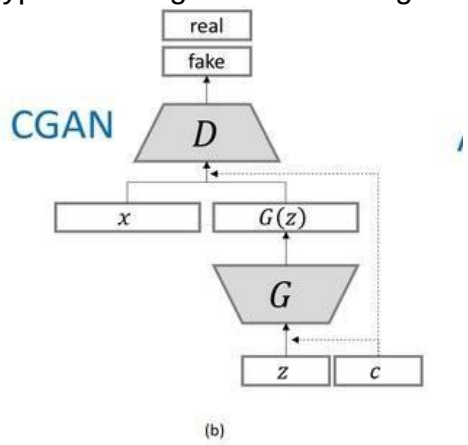


Picture 2: DCGAN [5]

The generator input is a latent vector that is taken from a standard normal distribution and the output is an image.

The discriminator input is an image and the output is a scalar probability that the image is from the real data distribution.

3. Conditional GAN(CGAN)
    - Dictate the type of data generated through a condition.



Picture 3: Conditional GAN (CGAN) [6]

3. Selected GAN subtype & Usage of expected results

This section is about chosen GAN model to solve the thesis task and the areas where the expected results could be used.

To train a neural network to generate new data, we must have a training dataset which this network will learn on. Since the main task of this thesis is to generate images of new species of spiders, the training dataset should consist of images of real spiders.

CNN (Section 15) is usually the best NN (Section 15) for work with images and it's made up of convolutional layers. One of two main tasks is to reduce image size and present it as a vector of values. This part CNN does. The second task is the transformation of the input vector (= noise) into an image. To implement this part, a reverse CNN model consisting of transposed convolutional layers should be used. These two parts are represented in DCGAN model (Section 2.5), so this NN has been used to generate fake spiders.

Necessary assumption is that spider generation refers to a main issue i.e. generating new images for different objects (e.g. vehicles, animals, buildings, etc.).

Taking into account the aforementioned assumption the generation of new images can be used in different tasks such as:

- Creating new videos (different locations, different objects in any position). Moreover, the copyrights will not be violated.
- Creating a new kind of animal, a new type of transport (e.g. a new car model) and furniture, an unusual view of building.
- Creating a large database for a particular object.
- Creating multiple new objects for video games.

4. Main algorithms in convolutional & convolutional transpose layers (DCGAN structure)

The task of the chapter to introduce the DCGAN model and how the convolution and convolutional layers work in model - what algorithms they use.

4.1 Main idea behind convolutional & convolutional transpose layers

The main task of the thesis is to generate new images. To accomplish the task, a NN should be created, which will be trained on the training dataset. Since the data we want to generate are images, the training dataset should also contain images.

The entire NN can be divided into two necessary parts: discriminator & generator work. The first part is about discriminator work where the input image transforms to one node output - the answer whether the image is fake or real. The second part is about generator work where incoming noise convert into an image of the same size as the training images.

Convolutional layers and convolutional transpose layers are the primary building blocks for the discriminator and generator respectively.

4.2 Discriminator algorithms in convolutional layers [8]

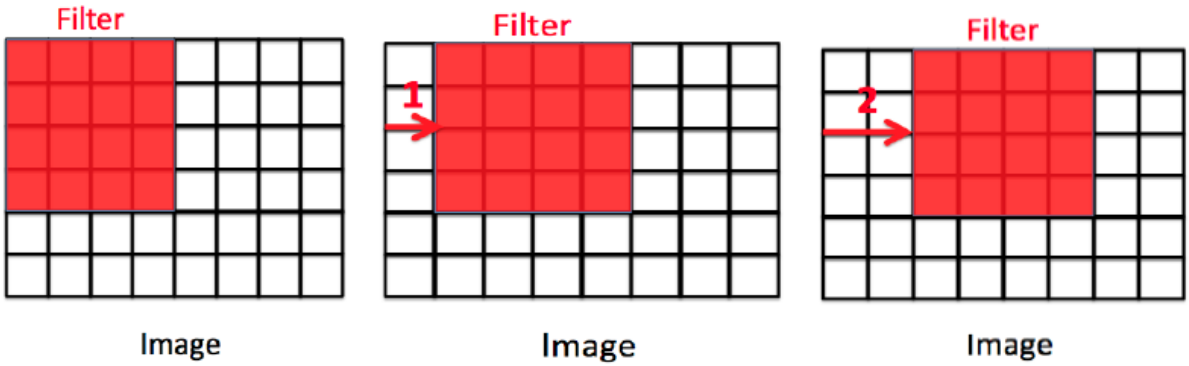
The subsection presents the information about usually used algorithms in convolutional layers.

4.2.1 Discriminator idea in convolutional layers

The idea is that each pixel of the training image is fed to the input of a NN and can be used in various mathematical calculations. But unfortunately, there is such a problem that the current computers do not have enough computing power to process so much data within reasonable time. To avoid this problem, different algorithms were invented to reduce the number of values (pixels) and to save the most significant information of the image at the same time. Most common algorithms are presented below.

4.2.2 Stride in a convolutional layer [7][8]

The strides parameter indicates how fast the kernel moves along the rows and columns on the input layer. If a stride is (1, 1), the kernel moves one row/column for each step; if a stride is (2, 2), the kernel moves two rows/columns for each step. As a result, the larger the strides, the faster the end of the rows/columns is reached, and therefore the smaller the output matrix (if no padding is added). Setting a larger stride can also decrease the repetitive use of the same numbers.



Picture 4: Stride representation in convolutions Equation 1: Calculation of the size of the output image [7]

The image above shows how stride works for a 4x4 filter with steps of 1 and 2. The illustration of the example will be presented in further part of the thesis [5.2].

4.2.3 Filters (=Feature detectors; =Kernels) in a convolutional layer

The convolution is a mathematical operation used to extract features from an input image. The convolution is defined by an image filter. The feature detector is a small matrix. The job of the filter is to find patterns in the image pixels in the form of features. The kernel size affects how many numbers in the input layer is "projected" to form one number in the output layer. The larger the kernel size, the more numbers in use, and each number in the output layer is a broader representation of the input layer and carries more information from the input layer. But at the same time, using a larger kernel will give an output with a smaller size.

Feature map is a result that is obtained after convolution operation of input image and filter ("Output" at the (Picture 4)).

Usually it will use many types of filters at input image to extract different features of the image. As a result, a specified number of feature maps is obtained at the output. Each feature map corresponds to a certain filter.

The size of the output image can be calculated if parameters such as input image size, a kernel size, padding and a stride step in a convolutional layer are known. Below is the formula for calculating the size of the output image. (Formula is the same for the output image for pooling operation [4.2.5]).

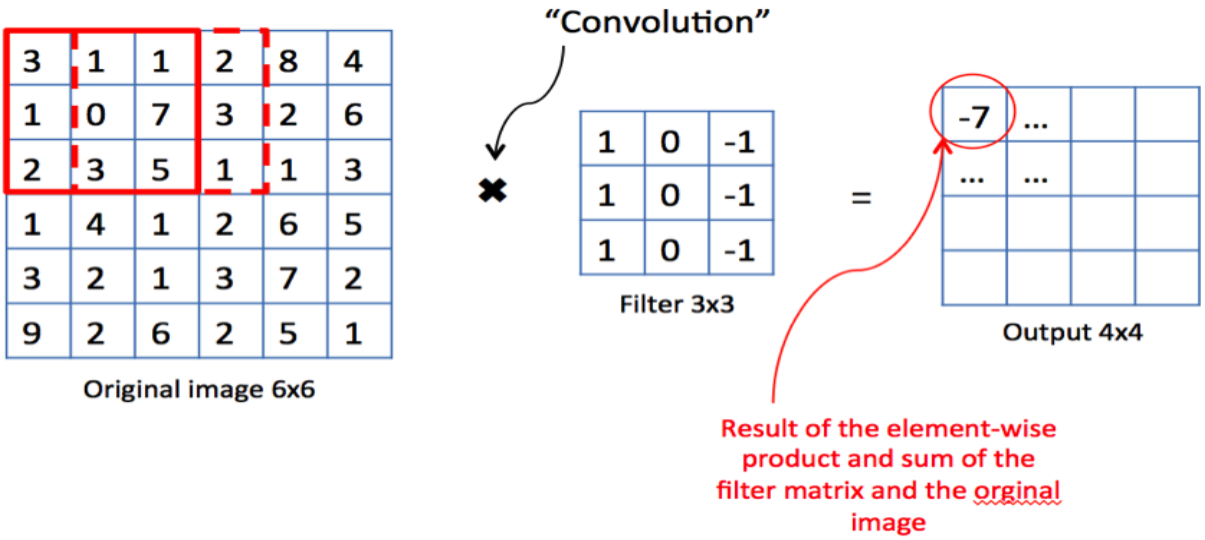
$$W_o = \frac{W_i - F_w + 2P}{S_v} + 1$$

$$H_o = \frac{H_i - F_h + 2P}{S_h} + 1,$$

Equation 1: Calculation of the size of the output image in convolutions [9]

where W_o, H_o are width and height of the output image; W_i, H_i are width and height of the input image; F_w, F_h are width and height of the filter; S_v, S_h are vertical and

horizontal stride steps.



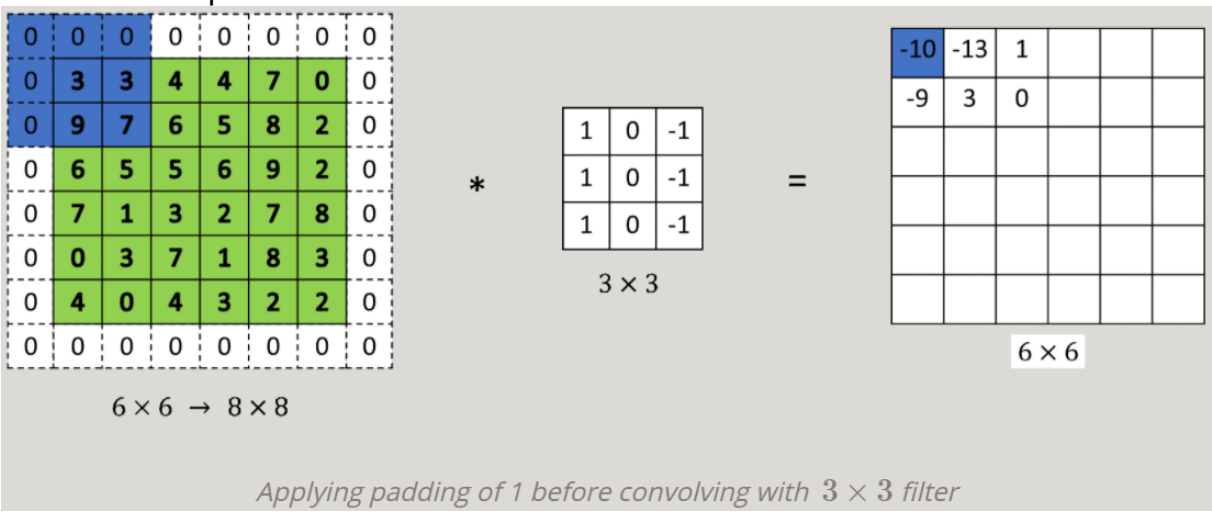
Picture 5: A 6∗6 image is convolved with a 3∗3 kernel in convolutions [10]

The example above shows how convolutional operation works on input image with 3x3 kernel, stride step of 1 and padding equals 0.

4.2.4 Padding in a convolutional layer

Original image size shrinks after convolution operation (Picture 5). NN usually include several convolutional layers. However, sometimes it's necessary to maintain the input size of the image. Moreover, filter touches the edge of the image less comparing to the center area of the image therefore, the edges of the image have less impact on the feature map than other parts of the image. In order to solve these issues, padding has been introduced.

Adding a pixel at each edge of the image helps to eliminate the aforementioned problems. Additional pixels are random numbers of the same value (it's usually 0 or 1). Hence, a padding algorithm preserves the image size (or doesn't shrink the image much) and helps to evenly distribute all the image information on the feature map.



Picture 6: Padding representation in convolutions [11]

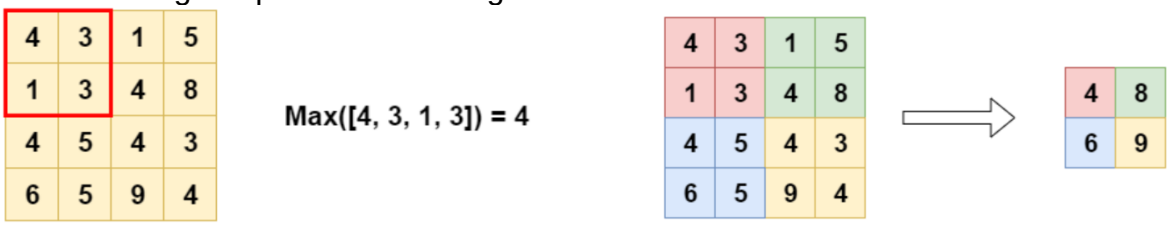
The example above shows how padding operation helps to save the size of input image in the output.

4.2.5 Pooling in a convolutional layer

Pooling is a function that helps to reduce the image size to reduce the network complexity and computational costs.

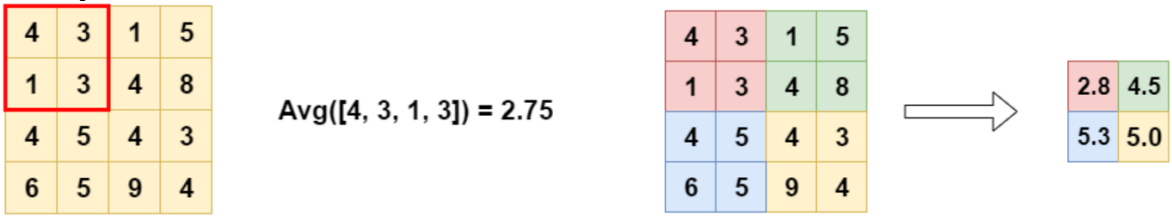
There are different types of pooling. Two of these types are the most popular. These are "Maximum Pooling" and "Average Pooling".

Max pooling takes the maximum of a region and it helps to proceed with the most important features from the image. Max pooling selects the brighter pixels from the image. It is useful when the background of the image is dark and it's necessary to extract the lighter pixels of the image.



Picture 7: Max pooling representation in convolutions [7]

Average Pooling retains much information about the “less important” elements of a block. Whereas, max pooling throws these "less important" elements away by picking the maximum value, average pooling blends them in. This can be useful in a variety of situations, where such information is useful.



Picture 8: Average pooling representation in convolutions [7]

The examples above present how the maximum and average pooling of size 2x2 work for a 4x4 input image with a stride step of 2. The output is reduced to 2x2 matrix. The output image size of this operation can be counted from the (Equation 1).

4.3 Generator algorithms in convolutional transpose layers [8]

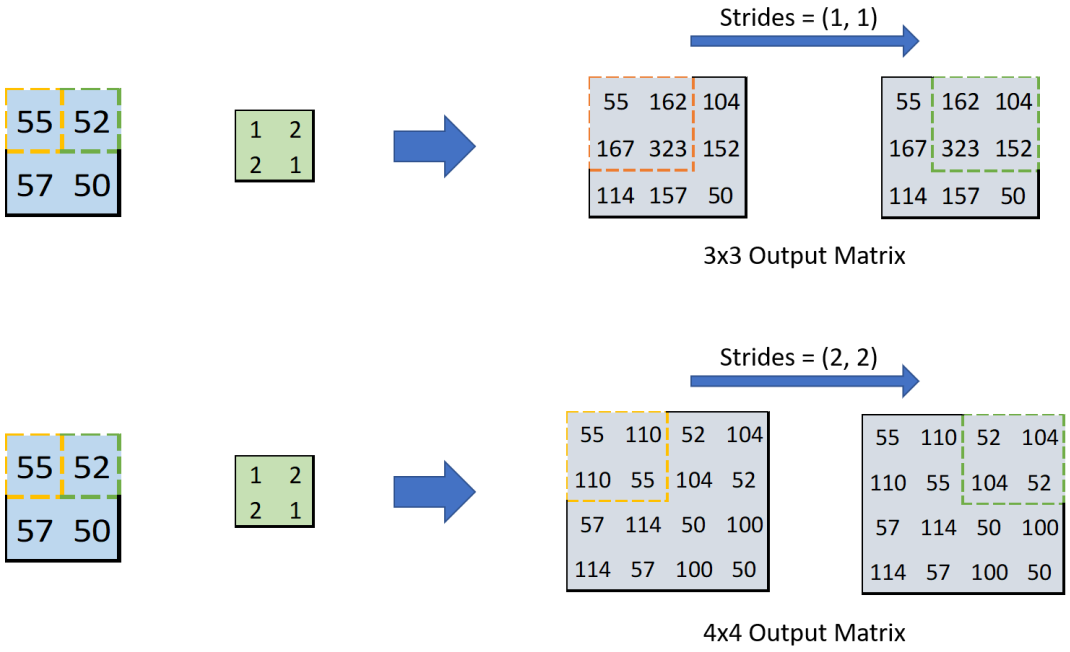
The subsection presents the information about usually used algorithms in convolutional transposed layers.

4.3.1 Generator idea in convolutional transpose layers

The idea is that input noise vector is converted to many small feature matrices. Using the same algorithms in reverse order from the (Section 4.2) helps to build a new image of the same size as the training images.

4.3.2 Stride in a convolutional transpose layer

The strides parameter indicates how fast the kernel moves on the output layer (Picture 9). The kernel always moves only one number at a time on the input layer. Therefore, larger the strides, the larger will be the output matrix. It’s the reverse version of the stride from the (Section 4.2.2).



Picture 9: Stride representation in transposed convolutions [8]

The example above demonstrates how a transposed convolutional operation works for different stride steps (input image of size 2x2; kernel of size 3x3; padding is 0). There is a different output image size which depends on the stride step (Equation 2).

4.3.3 Filters (=Feature detectors; =Kernels) in a convolutional transpose layer

The size of the output image can be calculated if parameters such as input image size, a kernel size, padding and a stride step in a convolutional transpose layer are known. Below is the formula for calculating the size of the output image.

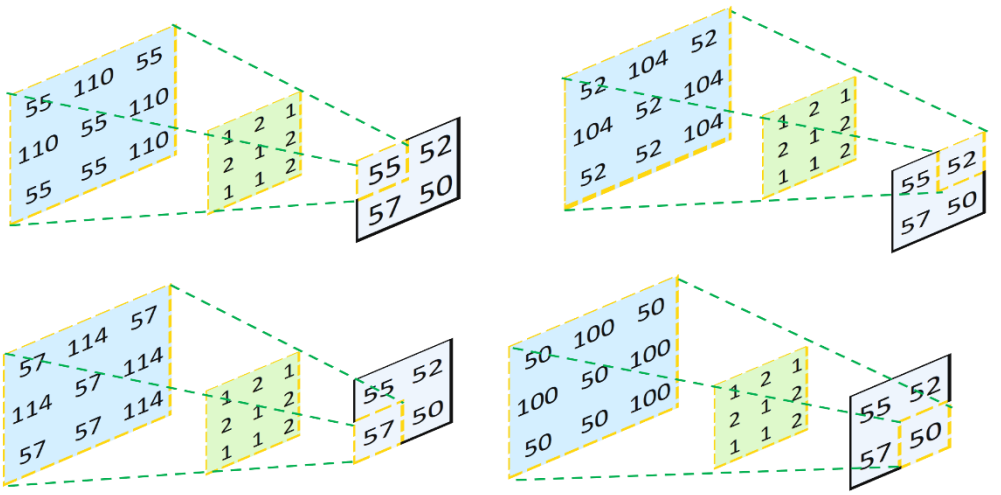
$$W_o = (W_i - 1) * S_v - 2P + (F_w - 1) + 1$$

$$H_o = (H_i - 1) * S_h - 2P + (F_h - 1) + 1$$

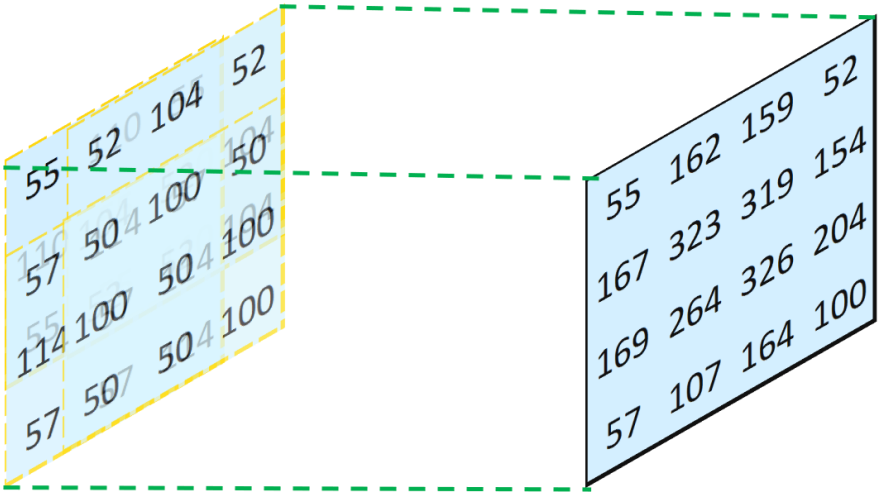
Equation 2: Calculation of the size of the output image in transposed convolutions [12]

where W_o, H_o are width and height of the output image; W_i, H_i are width and height of the input image; F_w, F_h are width and height of the filter; S_v, S_h are vertical and horizontal stride steps.

Each value in the input layer is multiplied with the kernel weights making a new matrix for each kernel value (Picture 10). Then, all new matrices are combined together according to the initial positions in the input layer, and the overlapped values are summed together (Picture 11). Therefore, larger the kernel size, the larger will be the output matrix.



Picture 10: Interaction of the kernel with the input matrix in transposed convolutions [8]

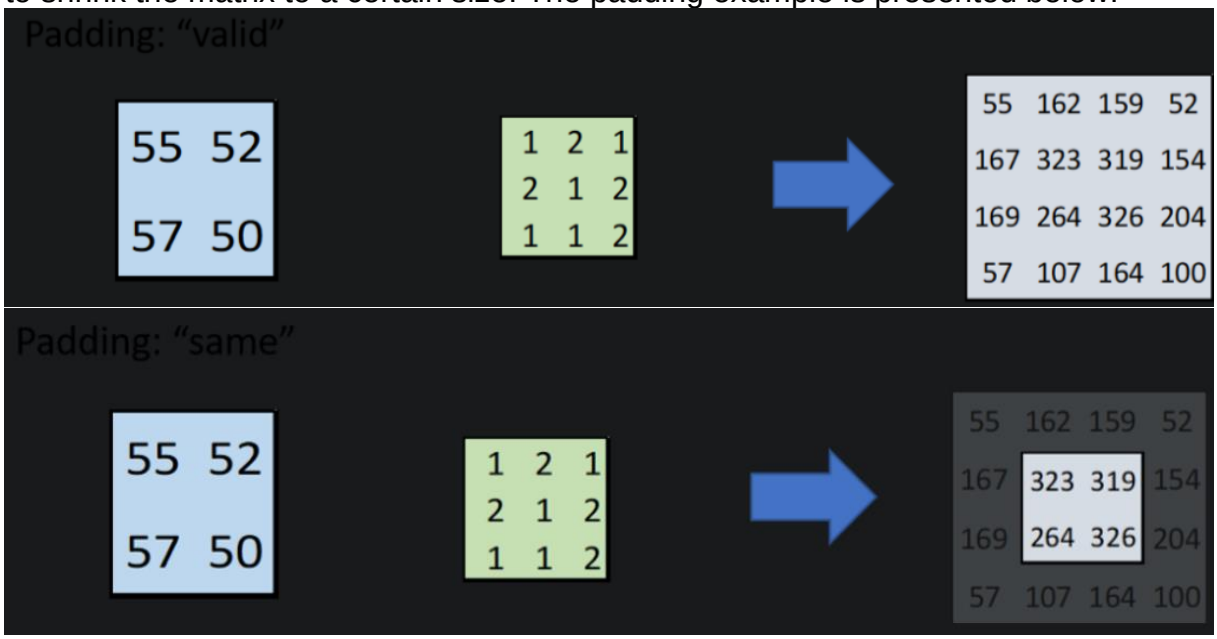


Picture 11: A 2x2 image upsampled to 4x4 size [8]

The examples above shows how transposed convolutional operation works for kernel of size 3x3 (input image of size 2x2; stride step is 1; padding is 0). As a result, the output is an image of 4x4 size (Equation 2).

4.3.4 Padding in a convolutional transpose layer

Padding shrinks the output feature matrix in reverse order, as shown in the (Section 4.2.4). Sometimes it is useful to keep only the most important information or to shrink the matrix to a certain size. The padding example is presented below.



Picture 12: Padding representation in transposed convolutions [8]

The picture above demonstrates how transposed convolutional operation works for different padding [padding is 0 in the 1st example; padding is 1 in the 2nd example] (input image of size 2x2; kernel of size 3x3; stride step is 1). There is a different output image size which depends on the padding (Equation 2).

5. Code: DCGAN structure & illustration

The purpose of this chapter is to show the discriminator and generator used structure in DCGAN model and some illustrations of different algorithms in convolutional and convolutional transposed layers from the code.

5.1 Selected GAN model – DCGAN

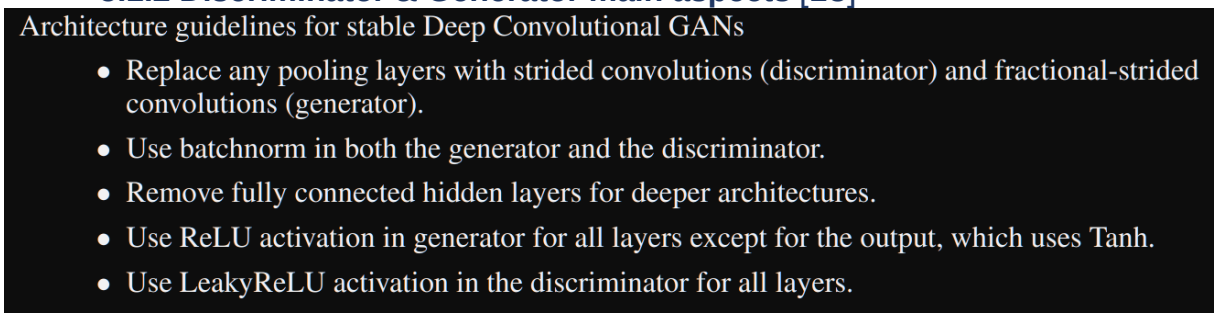
5.1.1 DCGAN introduction

A Deep Convolutional Generative Adversarial Network has been chosen to generate new images of spiders.

A DCGAN is a direct extension of the GAN.

A basic idea of how DCGAN works is shown in GAN (Section 2.5). Main algorithms of DCGAN structure are presented in (Section 4). A simple illustration example of DCGAN work is presented in (Sections 5.2 and 5.3).

5.1.2 Discriminator & Generator main aspects [13]



Picture 13: DCGAN guidelines [13]

The DCGAN paper mentions that it is a good practice to use strided convolution rather than pooling to downsample because it lets the network learn its own pooling function. Also batch norm and leaky relu functions promote healthy gradient flow which is critical for the learning process of generator and discriminator.

5.2 DCGAN: Discriminator part

5.2.1 Discriminator introduction

The discriminator is made up of strided convolution layers, batch norm layers, LeakyReLU and Sigmoid activations.

The input data is the dim-x-y input image, where dim is the color scale of the image (1-gray image; 3-color image), x, y are the width and height of the input image, respectively.

5.2.2 Convolutional layer – main steps in theory

Step 1: Input image (with/without padding) ^{x} Filter -> Feature Map.
Step 2: Feature Map ^{x} Pooling Type -> Pooled Feature Map.
Step 3: Pooled Feature Map ^{x} Filter -> New Feature Map.
Step 4a: New Feature Map = Output scalar.
Step 4b: New Feature Map -> Flattening vector (=input layer of the NN).
Step 5b: Flattening vector & NN structure -> Output scalar.

First three steps are used to decrease the size of input image and extract the most significant information (features) of the image at the same time. Step 2 and 3 may be repeated for several times (Section 5.2.3).

There're two different options after the 3rd step. The first option is when the

feature matrix has decreased to the size of only one value (Step 4a). This approach has been used for the most of NN in this thesis. The second option is when the feature map transformed to a flatten vector and then this vector is transformed into one value (Step 4b and 5b). This approach has been presented for an inception model (Section 12).

The main purpose of convolutional layers is to decrease the size of the input matrix and find different features in the image.

5.2.3 Discriminator structure in code

The discriminator is a binary classification network that takes an image as input and outputs a scalar probability that the input image is real. At the image below, the discriminator takes a 3x64x64 input image, processes it through a series of Conv2d, BatchNorm2d, and LeakyReLU layers, and outputs the final probability through a Sigmoid activation function.

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

Picture 14: Discriminator structure in the used code

In the image above, the number of feature maps is growing with each convolution layer (except the last one). At the same time, the size of feature map shrinks twice with each convolutional layer (except the last one) (It's easy to calculate from the equation 1; an example of first convolutional layer is shown in 7.2.4 section).

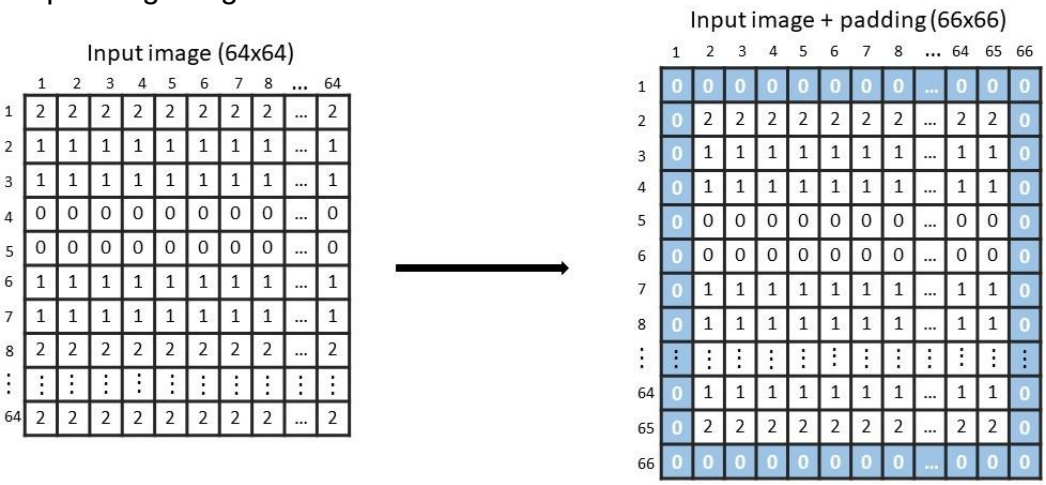
5.2.4 Discriminator illustration part – example

Input image size (W_i, H_i) = 64x64.
Padding = (1, 1).
Filter size (F_w, F_h) = 4x4.
Stride step (S_v, S_h) = 2.

Output image size (W_o, H_o) can be counted from the equation 1. The width and the height of the output image will be the same because the width and the height are the same for other parameters in the (Equation 1).

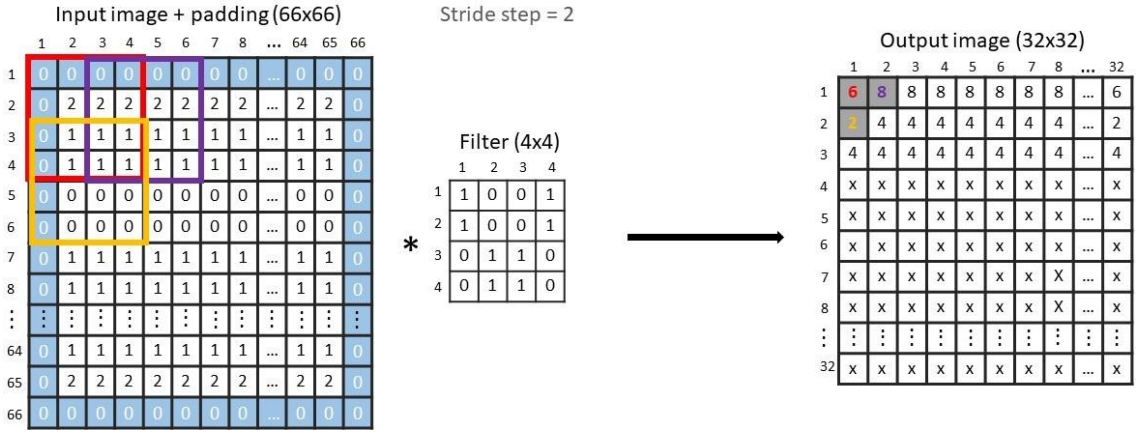
$$W_o = H_o = \frac{64 - 4 + 2 * 1}{2} + 1 = 32$$

Step 1 – padding usage:



Picture 15: Padding illustration

Step 2 – filter usage:



Picture 16: Filter illustration

5.3 DCGAN: Generator part

5.3.1 Generator introduction

The generator is comprised of convolutional-transpose layers, batch norm layers, ReLU and Tanh activations.

The input is a latent vector (=noise) that is drawn from a standard normal distribution and the output is a 3x64x64 RGB image.

5.3.2 Convolutional transpose layer – main steps in theory

Step 1: Latent vector ^{x} Filter -> Feature Map.
Step 2: Feature Map ^{x} Padding -> Padded Feature Map.
Step 3: Padded Feature Map ^{x} Filter -> New Feature Map.
Step 4: New Feature Map = Output image.

The first step is used to transform the latent vector into many matrices, where each matrix Is unique and correspond to a specific feature of the future image (Section 5.3.3).

The second and third steps are used for several times to increase the size of the feature maps (Section 5.3.3). The output image is obtained when the size of the feature map is the same size as the size of the images from the training dataset.

The main purpose of convolutional transposed layers is to create the image from the latent vector. At the same time these layers try to recreate the similar features that are in a real image.

5.3.3 Generator structure in code

The generator is designed to map the latent space vector (=noise) to data-space. Since our data are images, converting noise to data-space means ultimately creating a RGB image with the same size as the training images (3x64x64). This is accomplished through a series of strided two dimensional convolutional transpose layers, each paired with a 2d batch norm layer and a relu activation. The output of the generator is fed through a tanh function to return it to the input data range of [−1,1]. These layers help with the flow of gradients during training.

```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

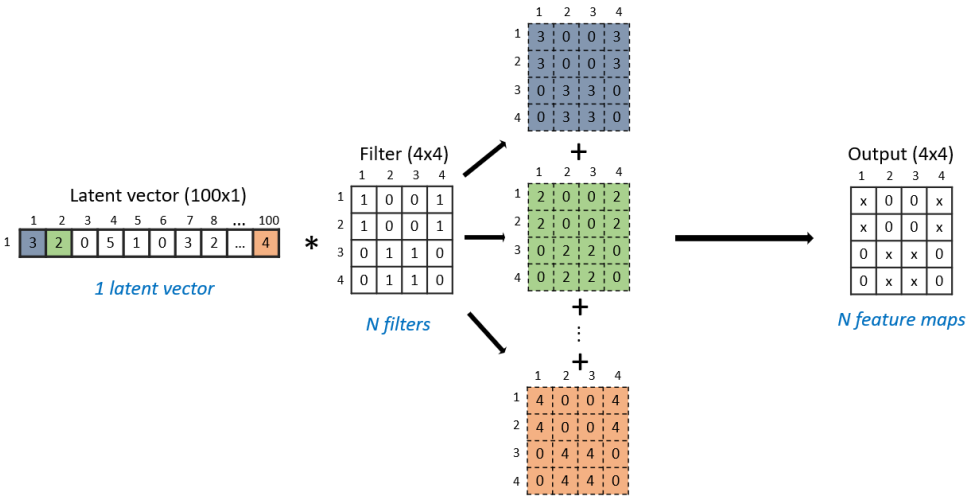
Picture 17: Generator structure in the used code

In the image above, the number of feature maps is decreasing with each convolution layer (except the first one). At the same time, the size of feature map grows twice with each convolutional layer (except the first one) (It's easy to calculate from the (Equation 2); an example of first convolutional transpose layer is shown in

(Section 5.3.4)).

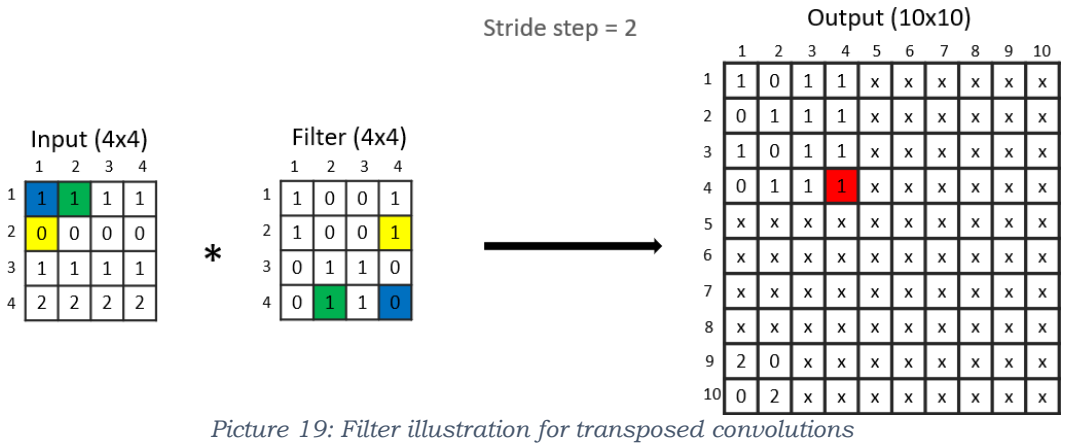
5.3.4 Generator illustration part – example

Step 1 – latent vector transformation (corresponds to the '0' layer in [Picture 17]):
Input image size (W_i, H_i) = 1x100.
Filter size (F_w, F_h) = 4x4.
Stride step (S_v, S_h) = 1.

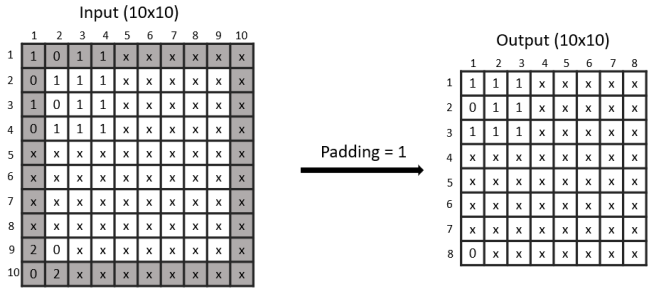


Picture 18: Latent vector transformation

Step 2 – kernel, stride and padding illustration (corresponds to the '3' layer in (Picture 17)):
Input feature map size (W_i, H_i) = 4x4.
Filter size (F_w, F_h) = 4x4.
Padding = (1, 1).
Stride step (S_v, S_h) = 2.



Picture 19: Filter illustration for transposed convolutions

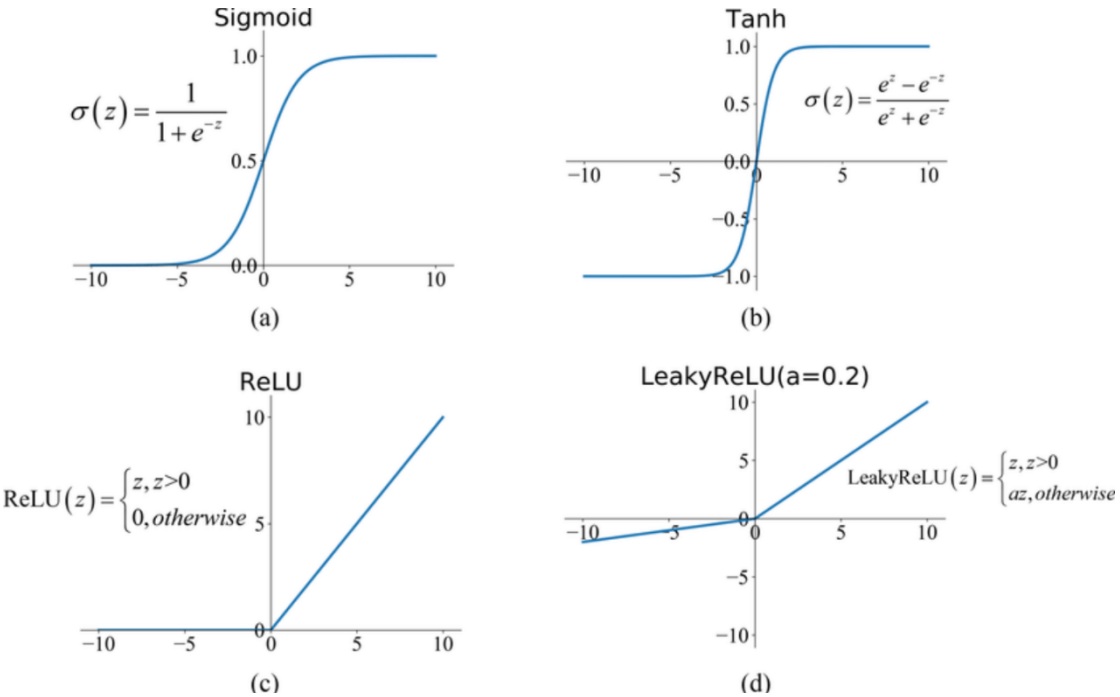


Picture 20: Padding illustration for transposed convolutions

6. Activation functions

The chapter presents activation functions and their equations that have been used in code part.

Activation functions convert large outputs from the units to smaller values and promote non-linearity in the NN.



Picture 21: Used activation functions [14]

ReLU and Tanh functions are used in generator part (Picture 17), whereas LeakyRelu and Sigmoid functions are used in discriminator part (Picture 14).

The tanh function is softer for small input values compared to Sigmoid function (Picture 21). For this reason, it is recommended to use the tanh function for the generator instead of the sigmoid function. ReLU functions are used before Tanh function in the generator part. It means that the input for tanh function are only positive numbers. Hence, only the positive (x, y - positive) section of Tanh function is taken in account.

The discriminator needs to predict whether the input image is real (=1) or fake

(=0). LeakyReLU takes negative inputs into account and is presented before the sigmoid function.

The statements above show how the input values are preprocessed to small numbers and how linearity is removed.

7. Used loss function

The chapter performs value function of GAN and main information about loss function that was used in NNs.

7.1 Introduction [15][16]

In the GAN paper [16], the adversarial modeling framework is most straightforward to apply when the models are both multilayer perceptrons. To learn the generator's distribution p_g over data x, we define a prior on input noise variables $p_z(z)$, then represent a mapping to data space as $G(z; \theta_g)$, where G is a differentiable function represented by a multilayer perceptron with parameters θ_g. We also define a second multilayer perceptron $D(x; \theta_d)$ that outputs a single scalar. D(x) represents the probability that x came from the data rather than p_g. We train D to maximize the probability of assigning the correct label to both training examples and samples from G. We simultaneously train G to minimize log(1 − D(G(z))). In other words, D and G play the following two-player minimax game with value function V (G, D), where the generator tries to minimize the following function while the discriminator tries to maximize it:

$$\min_G \max_D \mathrm{V(D,G)} = E_{X\sim Pdata(x)}[\log(D(X))] + E_{z\sim Pz(z)}\left[\log\left(1 - D\left(G(z)\right)\right)\right],$$

Equation 3: GAN value function [16]

In the function above:

- D(X) is the discriminator's estimate of the probability that real data instance X is real.
- E_x is the expected value over all real data instances.
- Pdata(x) is the distribution which the real data come from.
- G(z) is the generator's output when given noise z.
- D(G(z)) is the discriminator's estimate of the probability that a fake instance is real.
- E_z is the expected value over all random inputs to the generator (in effect, the expected value over all generated fake instances G(z)).
- Pz(z) is the distribution which the fake data come from.
- The formula derives from the cross-entropy between the real and generated distributions.

In theory, the solution to this minimax function is where $p_g = p_{data}$, and the discriminator guesses randomly if the inputs are real or fake.

7.2 Implementation

Binary Cross Entropy loss (BCELoss) function was implemented for discriminator and generator. This function is defined in PyTorch as:

$$l(x,y) = L = \{l_1, \dots, l_N\}^T,\ l_N = -[y_N * \log(x_N)] + [(1 - y_N) * \log(1 - x_N)]]$$

Equation 4: BCELoss function [17]

In the function above:

- N is the batch size.
- x_n is the input after discriminator/generator structure.
- y_n is the expected output.

8. Training process – optimizers

8.1 Introduction [18]

There're several algorithms used for training NN. The main idea behind the training process is to change the NN parameters such as weights and biases. The most common algorithm for training process is a backpropagation algorithm which is based on gradient descent. The backpropagation algorithm has also been chosen for a training process in this thesis. Different optimization functions (=optimizers) are used to find the optimal values for the NN parameters (weights, biases). All of this functions are based on a gradient descent and depends on a chosen loss function.

8.2 Optimizers [18]

The chapter performs the main idea, characteristics and equations about different optimizers that have been tested to generate fake spiders.

8.2.1 Mini-batch Gradient Descent

Gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in R^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ to the parameters. The learning rate η determines the size of the steps to reach a (local or global) minimum. We follow the direction of the slope of the surface created by the objective function downhill until we reach a valley.

Mini-batch gradient descent performs an update for every mini-batch of n training examples:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

Equation 5: Mini-batch Gradient Descent function [18]

In the function above:

- θ is the model parameter.
- η is the learning rate.
- n is the number of examples – batch size.
- $J(\theta)$ is the objective function.
- $\nabla_\theta J(\theta)$ is the gradient of the objective function.

This algorithm computes a mini-batch gradient very efficient and reduces the variance of the parameter updates, which can lead to more stable convergence.

8.2.2 RMSprop

The biggest problem for mini-batch gradient descent is that the learning rate stays the same during the training process. RMSprop algorithm solves this problem by adapting the learning rate to the parameters.

The sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average $E[g^2]_t$ at time step t then depends (as a fraction γ similarly to the Momentum term) only on the previous average and the current gradient

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma){g_t}^2$$

Equation 6: Average of past squared gradients [18]

Equation 4 for RMSprop looks like:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Equation 7: RMSprop function [18]

ϵ is a smoothing term that avoids division by zero.

8.2.3 Adam (Adaptive Moment Estimation)

Adam is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v_t like RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

Equation 8: Decaying average of past gradients [18]

$$v_t = \beta_2 v_{t-1} + (1-\beta_2){g_t}^2$$

Equation 9: Decaying average of past squared gradients [18]

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1). They counteract these biases by computing bias-corrected first and second moment estimates:

$$\widehat{m_t} = \frac{m_t}{1-{\beta_1}^t}$$

Equation 10: Bias-corrected first estimate [18]

$$\widehat{v_t} = \frac{v_t}{1-{\beta_2}^t}$$

Equation 11: Bias-corrected second estimate [18]

Hence, Equation 4 for Adam looks like:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\widehat{v_t}} + \epsilon} \widehat{m_t}$$

Equation 12: Adam function [18]

9. Code: Dataset preprocessing part

The chapter provides information about the working environment and programming language. It also shows how the final dataset has been obtained and what characteristics the images have within this dataset.

9.1 Programming language, libraries & environments [19][20][21][22][23]

This section provides information about the programming language and libraries used to create the fake spiders and what environment has been used for this task.

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

PyTorch is an open source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing.

Matplotlib provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits.

Colaboratory, or “Colab” for short, is a product from Google Research. Colab allows to write and execute arbitrary python code through the browser, and is especially well suited to machine learning, data analysis and education.

Python is an interpreted, high-level and general-purpose programming language. It helps programmers to write clear, logical code for small and large-scale projects.

9.2 Dataset

9.2.1 Used datasets & preprocessing part

The section is about used spider datasets, main problems in a concatenated dataset and how these problems have been eliminated.

The data have been taken from different resources. There’re “Animal Classification” [24], “Arthropod Taxonomy Orders Object Detection Dataset” [25] and “Animals-10” [26] datasets from kaggle. “Spider-recognition_dataset” [27] is from GitHub. Also there’re some spider images from “atlas of living Australia” website [28].

All the datasets have been concatenated into one spider dataset of about 30000 images. But there were such problems with images as different image sizes and trash images. So, the following steps have been taken to avoid these problems:

1. Manual disposal of unnecessary images such as: not spider images, bad spider images (e.g. spider's legs are not visible), images with spider at the very edges of the images.
2. Center cropping (all spiders in the center of the image):
    - Smallest side selection: height or width.
    - The new smallest side size is: new_size = 0.8*old_size
    - The new size is used for both: height and weight.
    - Images have been cropped using the new sizes.
3. Resizing:
    - Images have been resized to 64x64 size. It’s the optional size that is usually used for NN training.

Center crop operation has been taken to remove unnecessary information at the edges of images. And finally, a resizing operation have been made to make all

images of the same size.

As a result, it has been obtained a dataset with the images of the same size and there have been no trash images.

9.2.2 Characteristics of the final dataset [24][25][26][27][28]

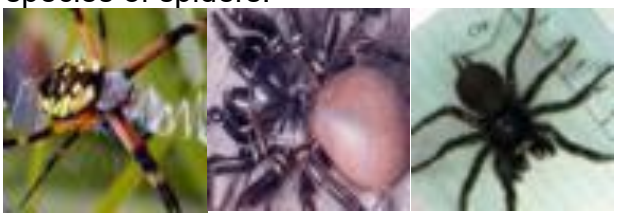
The section provides characteristics for the entire dataset that is used as training data.

1. There're 21809 spider RGB images in a final dataset.
2. All images are of the same 64x64 size.
3. Images before/after preprocessing part:

Picture 22: Dataset examples before preprocessing part

Picture 23: Dataset examples after preprocessing part

4. There're different species of spiders:



Picture 24: Different spider species

5. There're images with different spider sizes:

Picture 25: Different spider sizes

6. There're spiders on different backgrounds. Some spiders are easily distinguishable from the background, some are difficult.

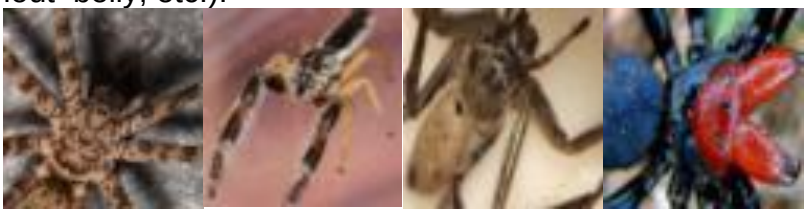
Picture 26: Background with easily distinguishable spiders

Picture 27: Background with difficult distinguishable spiders

7. Images of spiders from different angles:

Picture 28: Spiders from different angles

8. Images of spiders without certain parts of the body (e.g. some legs are no visible, without belly, etc.):



Picture 29: Spiders without certain parts of the body

Due to the wide variety of spider characteristics in the training set, it is expected that newly generated spiders will also have the same characteristics. However, a wide variety of characteristics also increases the difficulty of generating new spiders, since the total number of features increases with the number of characteristics for all spiders in the dataset.

10. Code: Training process

The aim of this section is to describe the process of training step by step for the created NN (Pictures 14, 17).

NN training process:

Step 1: Set the number (N) of epochs for a training process.

Step 2: Take batch of real images that equals to batch size parameter.

Step 3: Set the gradient to zero for discriminator.

Step 4: Put the batch of real images in the discriminator and get the prediction for each image whether it's fake (=0) or real (=1). Prediction is in (0, 1) range.

Step 5: Make a vector full of 1 of the same size as the number of images in the batch.

Step 6: Compare the output from Step 4 with the label from Step 5 using the specified loss function. It means that the loss is calculated for a real batch of images – the 1st part of the discriminator loss.

Step 7: Calculate the gradients for a real batch of images.

Step 8: Generate the latent vectors - noise. The number of these vectors equals to the batch size.

Step 9: Put the batch of latent vectors in the generator and get the batch of fake images.

Step 10: Put the batch of fake images (from Step 9) in the discriminator and get the prediction for each image whether it's fake (=0) or real (=1). Prediction is in (0, 1) range.

Step 11: Make a vector full of 0 of the same size as the number of images in the batch.

Step 12: Compare the output from Step 10 with the label from Step 11 using the specified loss function. It means that the loss is calculated for a fake batch of images – the 2nd part of the discriminator loss.

Step 13: Calculate the gradients for a fake batch of images from Step 9.

Step 14: Update discriminator parameters.

Step 15: Set the gradient to zero for generator.

Step 16: Repeat Step 10. The output should be different from Step 10 because the discriminator has been updated after Step 10.

Step 17: Compare the output from Step 16 with the label from Step 5 using the specified loss function. It means that the loss is calculated for a fake batch of images – the generator loss.

Step 18: Calculate the gradients for a fake batch of images from Step 9.

Step 19: Update discriminator parameters.

Step 20: Take the next batch of real images in the dataset that equals to batch size parameter and repeat Steps 3-19.

Step 21: Repeat Steps 2-20 for the same dataset for N epochs.

The presented steps show how the loss is calculated and when the discriminator and generator parameters are updated.

11. Results

The purpose of this chapter is to present the achieved results and analyze the obtained results.

There're three sections in this chapter. The first and second sections give the information on newly generated spiders and losses in various NNs respectively, while the third section presents the most significant information of used NN models in (Table 1).

ID numbers in image titles correspond to ID numbers in (Table 1).

Need to mention that generative adversarial networks lack an objective function, which makes it difficult to compare performance of different models. [29] Visual examination of samples by human is one of the common and most intuitive ways to evaluate GANs. [30] So, human evaluation has been chosen as the main method for evaluating the generated spiders. In addition, loss results have also been taken into account to evaluate the quality of generated images.

Due to the difficulty of evaluating the results with the above method, multiple spider images have been generated for each NN to see the variability in the results. This approach made it easier and more accurate to determine which of the two networks gives the best results.

11.1 New types of spiders

This section shows a comparison of the obtained results for different NNs and describes the various features of the generated spiders. It also provides brief information on the differences between the different NNs that have been used.

Different batch sizes, optimizers, hyperparameter of the optimizers have been tested. The optimal number of epochs that has been set for a training process is 1000. The generated spiders have been saved every 50th epoch.

Results are mixed. There're bad results for some parameters. At the same time there're generated images with different features that look differently, but it's hard to say which one is better.

Three different optimizers have been used for a training process. These optimizers are: SGD, Adam and RMSprop (Section 8).

Different experiments have been taken that are shown in the following sections. These experiments are:

1) Different optimizers work.
2) Examination the influence of hyperparameter on the final output.
3) Examination the influence of batch size on the final output.
4) Big number of epochs.
5) Different time of generator and discriminator update.

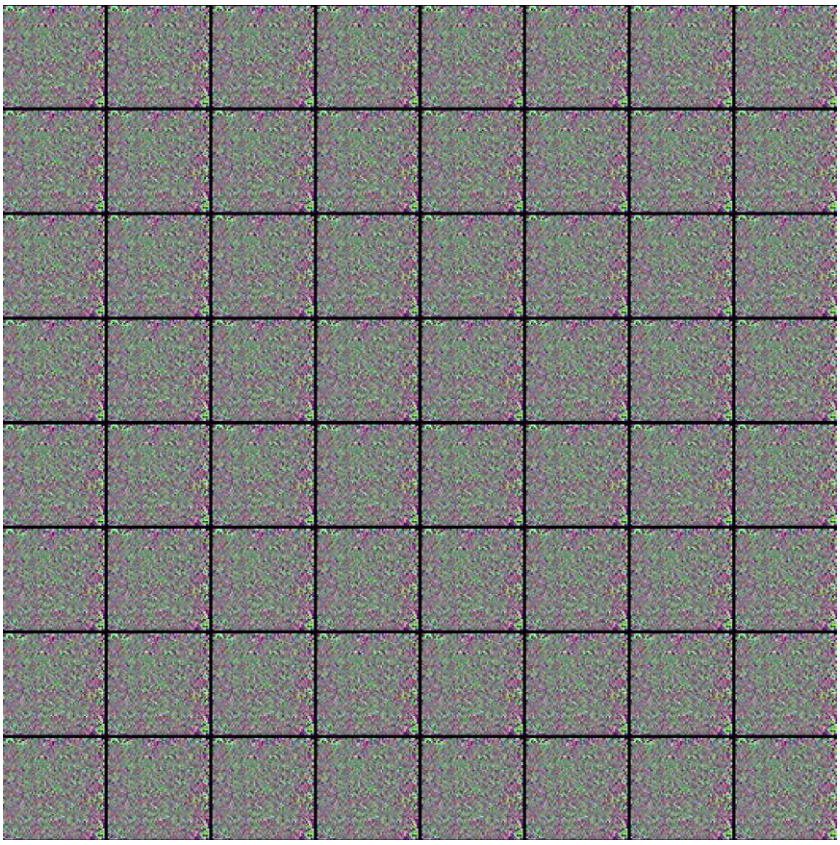
11.1.1 Generated spiders – Result №1 (*Influence of optimizers*)

The main goal of this step was to observe is it possible to generate any images using DCGAN model.

The following examples perform the best generated spiders of the same NN but for different optimizers (Table 1):

Picture 30: Generated spiders, Adam optimizer, 650 epoch [id 2]

Picture 31: Generated spiders, RMSprop optimizer, 450 epoch [id 2]



Picture 32: Generated spiders, SGD optimizer, 1000 epoch [id 2]

In above images, there's a bad result for a SGD optimizer, but Adam and RMSprop optimizers perform results that look similar to spiders. In this subcection the best results have been achieved for Adam optimizer.

Unfortunately, the same situation was for several NNs from (Table 1) using the SGD optimizer. Most probably it was caused by a very little learning rate parameter that was set to 0.0002. Such a small parameter could only slightly change the network parameters, which could lead to the fact that new images will look like noise. But it hasn’t been checked yet therefore, the remaining results will be presented for only Adam and RMSprop optimizers.

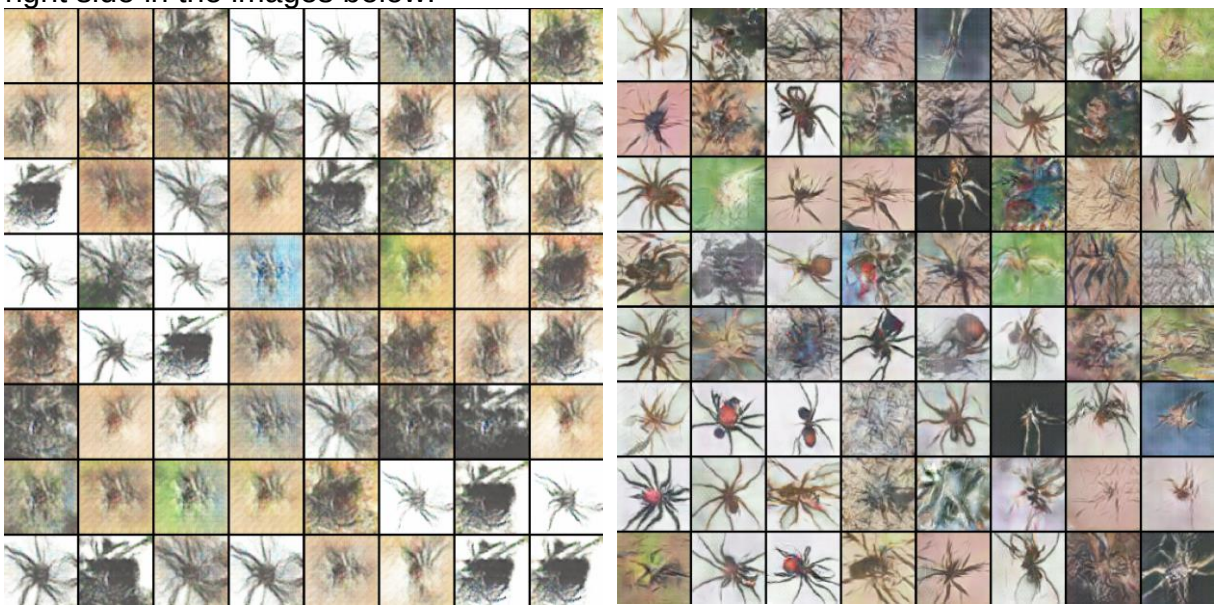
As a result, images that resemble spiders have been obtained for two of the three optimizers. This means It can be possible to get good results for DCGAN model trying to change the parameters or structure of the NN.

More details on these generated spiders will be presented in (Section 11.1.3) (the same images will be shown there). See (Section 11.2) for information about the loss for different optimizers.

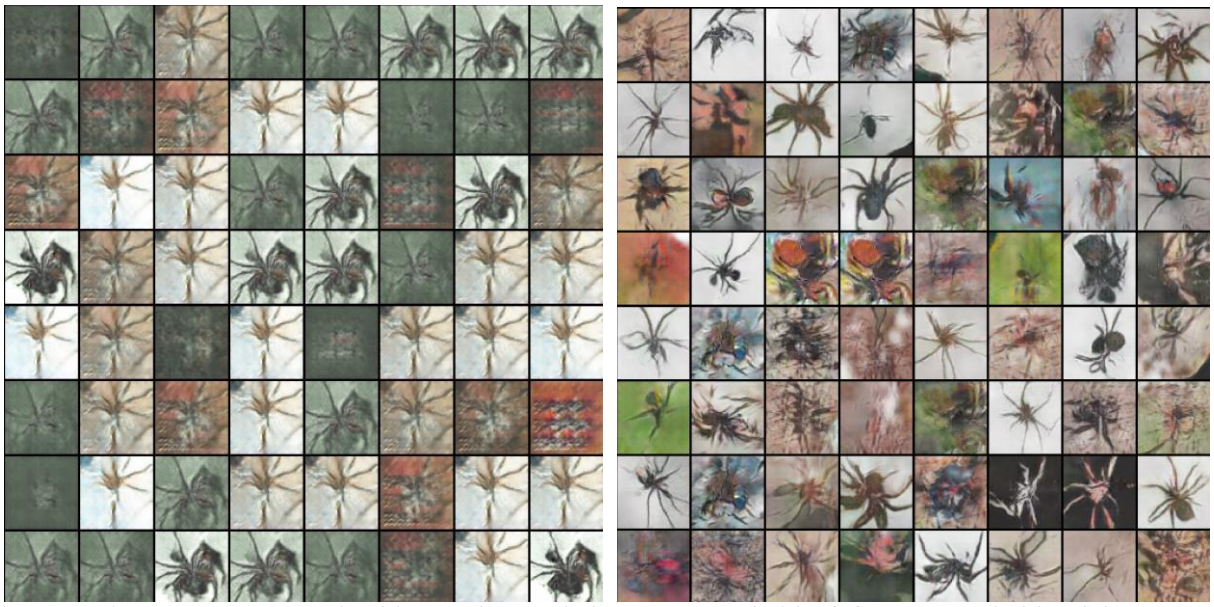
11.1.2 Generated spiders – Result №2 (*Influence of hyperparameter*)

This step has been taken to show the influence of beta1 parameter in a training process for both optimizers. The best generated spiders of training process are taken for both betas for the same NN. The results for two different NNs are presented.

It has been tested for a beta1 equals: 0.9 – on the left side and 0.5 – on the right side in the images below.



Picture 33: Generated spiders, Adam optimizer, 600 epoch [id 9-left], 950 epoch [id 10-right]



Picture 34: Generated spiders, Adam optimizer, 600 epoch [id 3-left], 450 epoch [id 4-right]

Above images demonstrate best generated spiders for NN with Adam optimize. Beta1 equals to 0.9 give bad results comparing with beta1=0.5. The following statement is based on the fact that the generated images for beta 0.9 have a lot of trash images, and there are also many ‘copies’ - images that are almost indistinguishable from each other. From (Equation 8), we can conclude that the less dependence on the previous calculated gradient, the better the result will be.

Picture 35: Generated spiders, RMSprop optimizer, 1000 epoch [id 9-left], 450 epoch [id 10-right]

Picture 36: Generated spiders, RMSprop optimizer 400 epoch [id 3-left], 550 epoch [id 4-right]

Above images demonstrate best generated spiders for NN with RMSprop optimizer. Here the situation is different compared to Adam optimizer. The results are acceptable for both values of beta1 parameter for RMSprop optimizer. However, it was necessary to choose one beta parameter to facilitate the following experiments. In my opinion, RMSprop optimizer with beta1=0.9 presents the best results. The quality and quantity of well-generated spiders with different characteristics and different backgrounds for beta1=0.9 is greater than beta1=0.5 (Also, the results of some other NN's were taken into account, which are not presented here).

From the findings above, the next experiments were made for Adam optimizer with beta1=0.5 and for RMSprop optimizer with beta1=0.9.

11.1.3 Generated spiders – Result №3 (*Influence of batch size*)

This step has been taken to show the influence of batch size in a training process for both optimizers.

The best generated spiders of training process are shown for both optimizers with different batch size parameter.

It is expected that the larger this parameter is, the better the result will be. The larger this number, the more accurately the gradient should be calculated. Therefore, the ideal situation would be when all the images could be taken in one batch. Unfortunately, the computing power of computers does not allow this. The maximum number of images that could be taken into a batch in my computer was equal to 4096.

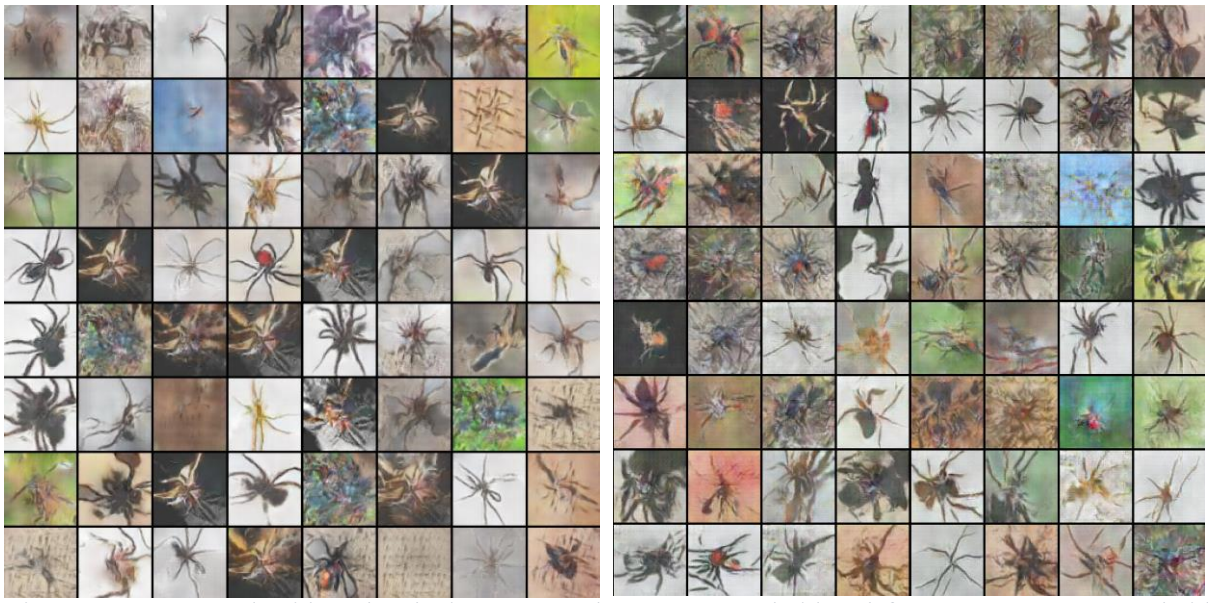
Picture 37: Generated spiders, batch size=32, Adam: 650 epoch [id 2-left], RMSprop: 400 epoch [id 1-right]

Picture 38: Generated spiders, batch size=64, Adam: 450 epoch [id 4-left], RMSprop: 400 epoch [id 3-right]

Picture 39: Generated spiders, batch size=128, Adam=650 epoch [id 6-left], RMSprop: 300 epoch [id 5-right]

Picture 40: Generated spiders, batch size=256, Adam: 650 epoch [id 8-left], RMSprop: 950 epoch [id 7-right]

Picture 41: Generated spiders, batch size=512, Adam: 950 epoch [id 10-left]; RMSprop: 1000 epoch [id 9-right]



Picture 42: Generated spiders, batch size=1024, Adam: 1300 epoch [id 12-left], RMSprop: 1500 epoch [id 11-right]

Picture 43: Generated spiders, batch size=2048, Adam: 1200 epoch [id 14-left], RMSprop: 1300 epoch [id 13-right]

Picture 44: Generated spiders, batch size=4096, Adam: 1300 epoch [id 16-left], RMSprop: 1250 epoch [id 15-right]

It can be concluded from the presented examples that increasing the size of the batch does not guarantee a qualitative and quantitative improvement in the generated spiders. So, in this experiment, images give the best results for batch size=512 for two optimizers (in my opinion). However, this is only a 'superficial' assessment, as it is visually difficult to determine exactly which of the results is the best. Different sizes of batches will be taken for the following experiments.

It's seen that there're many well created fake spiders with different characteristics for all batch sizes for both optimizers. These characteristics are listed in (Section 11.1.6).

Due to the fact that most of the above examples have shown for the optimal number of epochs, the following experiment has been made, which consisted in increasing the number of epochs to 2300. The result of this experiment is presented in (Section 11.1.4).

Some of the images are bad because they demonstrate much noise. It may mean that the gradient descent found a local minimum instead of a global minimum.

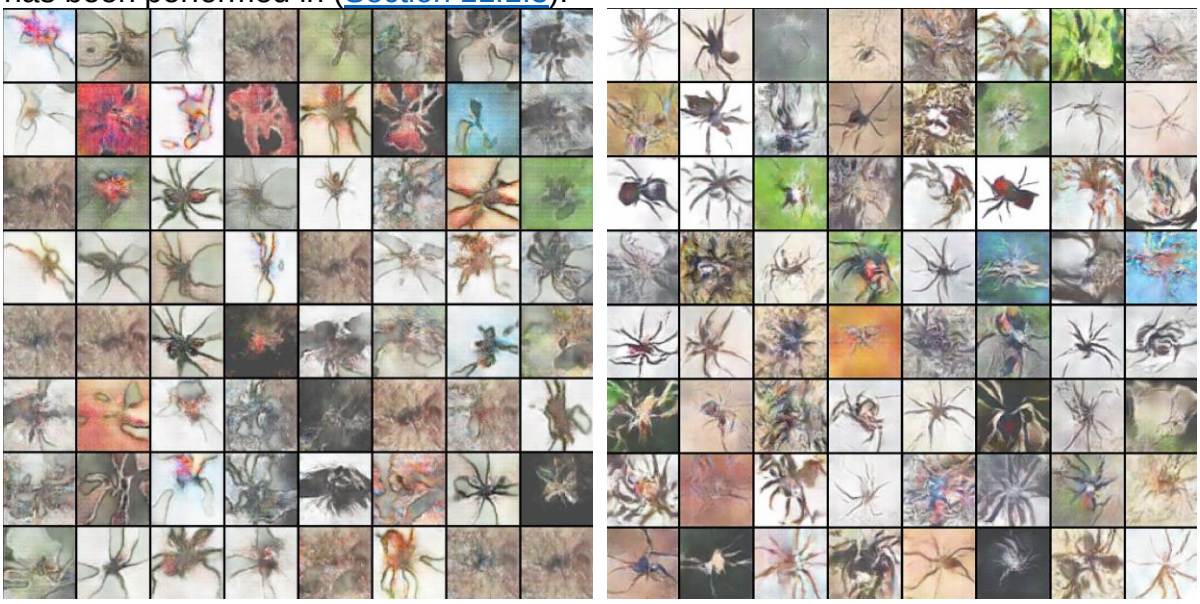
11.1.4 Generated spiders – Result №4 (*batch size=4096*)

The main task of this step is to observe the change in the result depending on the further increase in epochs (for most of the above examples in (Section 11.1.3), the optimal number of epochs was 1000).

The new test included 2300 epochs and has been performed for two optimizers with a sample parameter of 4096. The training was continued for the existing NN - for the same parameters (Picture 44). It was 1300 epochs in a first training and it was continued for 1000 more epochs.

The test was made for this batch size, since the generator loss for Adam optimizer was slightly different compared to losses for fewer batch size and it looked like the NN was not fully trained due to unique oscillations (Picture 60).

The best results for the second training is presented below (the first training has been performed in (Section 11.1.3):



Picture 45: Best generated spiders for batch size equals 4096 (2300 epochs): Adam [id 16-left (1600 epoch)]; RMSprop [id 15-right (1550 epoch)]

In above image, the result doesn't differ much from the result in (Picture 44). Hence, 1000-1500 epochs is a good choice as an optimal number for a training process.

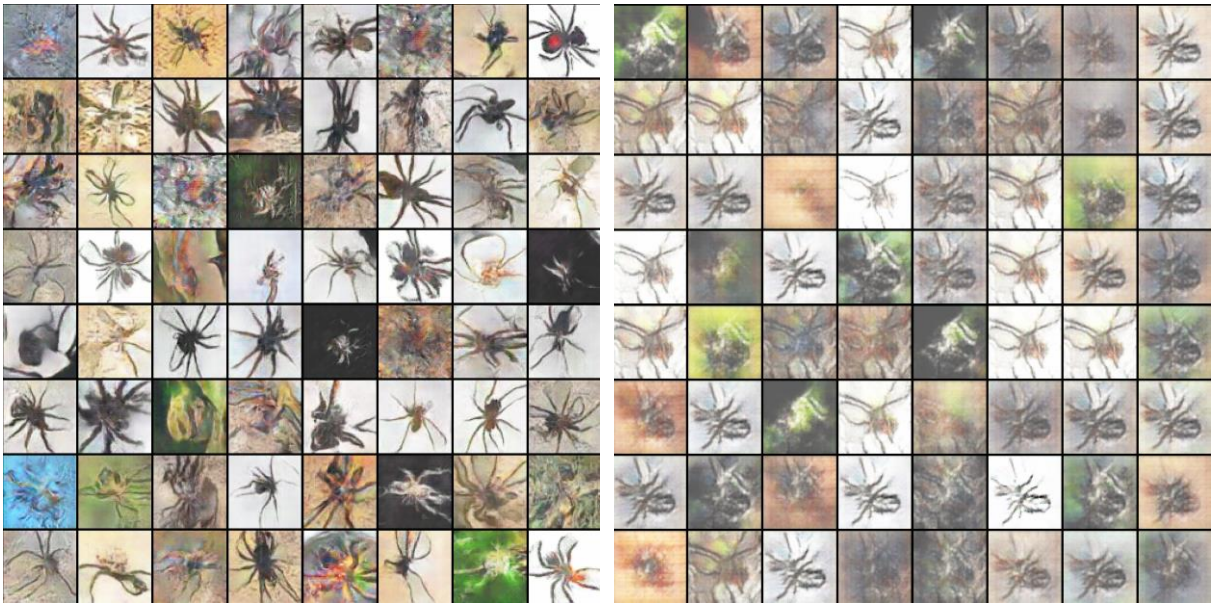
11.1.5 Generated spiders – Result №5 (different number generator/discriminator updates)

The task of this step is to examine how the result will change when:

a) One generator update after five discriminator updates for NN of id 9, 10 [Table 1]. Number of epochs was set for 2500. (On the left side in the examples below).

b) One discriminator update after three generator updates for NN of id 15, 16 [Table 1]. Number of epochs was set for 2500. (On the right side in the examples below).

The main goal of this examples is to observe whether these methods can improve the quality of generated spiders.



Picture 46: Generated spiders, Adam optimizer, (2250 epoch) [id 10-left], (1500 epoch) [id 16-right]

Picture 47: Generated spiders, RMSprop optimizer, (2500 epoch) [id 9-left], (1500 epoch) [id 15-right]

The generated spiders are acceptable (except 'b' option for Adam optimizer). It's difficult to say whether the images are better than in (Section 11.1.3) but some of the generated spiders look pretty well and can be used for different purposes.

11.1.6 Characteristics of new generated spiders

The task of this section is to demonstrate different characteristics of generated spiders (1-5), show the example of a training process and present one of the issue that sometimes appear.

1. Generated images of different spider species:

Picture 48: Different spider species

2. Generated images of spiders from different angles.

Picture 49: Spiders from different angles

3. Generated spiders on different backgrounds. Some spiders are easily distinguishable from the background, some are difficult.

Picture 50: Background with easily distinguishable spiders

Picture 51: Background with difficult distinguishable spiders

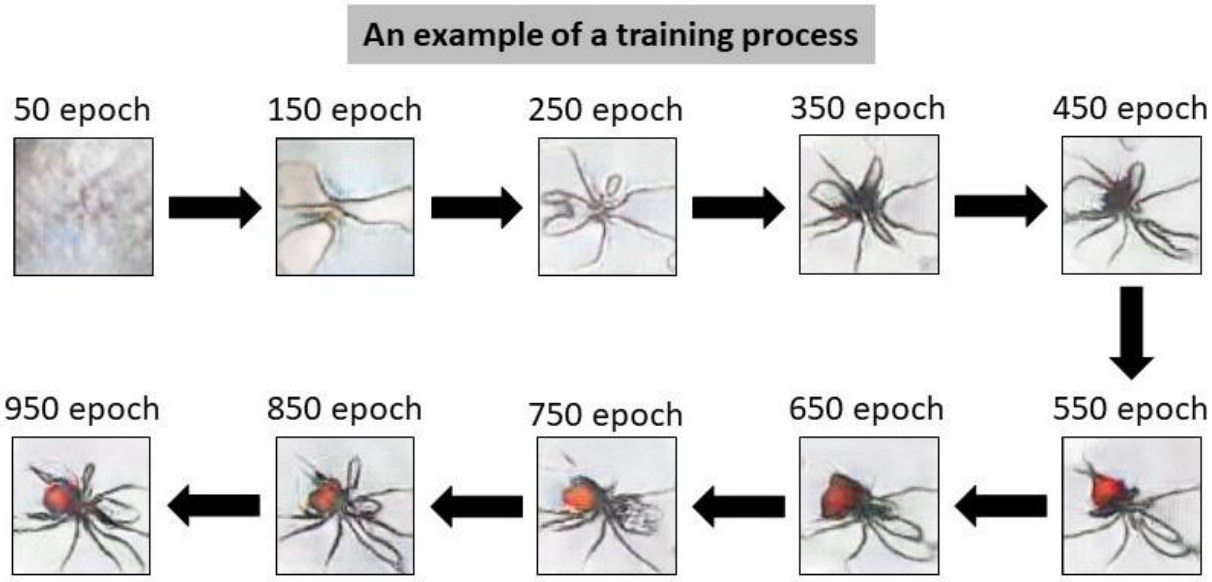
4. Generated images of spiders without certain parts of the body (e.g. some legs are not visible, without belly, etc.):

Picture 52: Spiders without certain parts of the body

5. Generated images of spiders on different backgrounds:

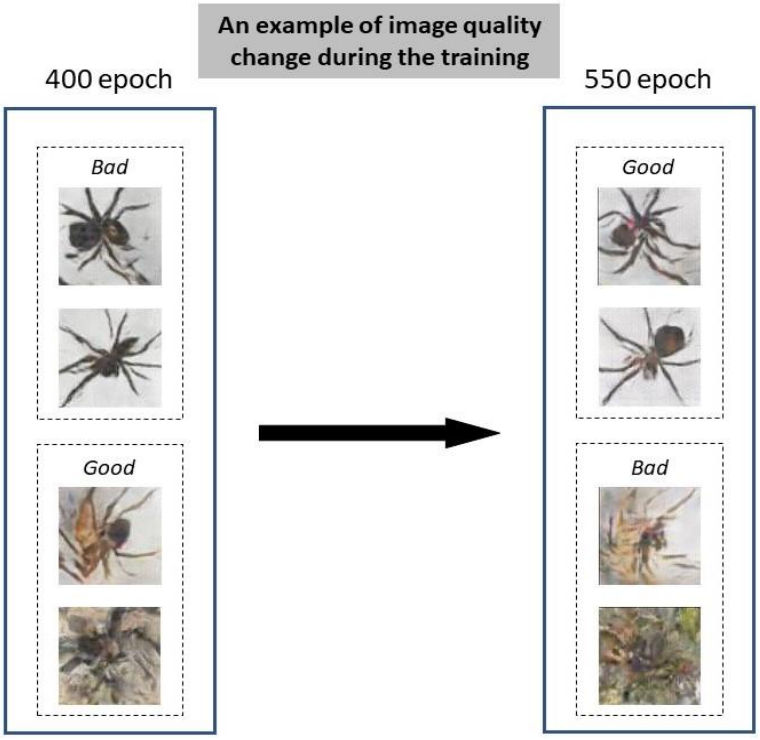
Picture 53: Spiders on different backgrounds

6. An example of a trainining process:



Picture 54: An example of a training process

7. The training process is to achieve the best image taken from different time in different distributions. Overfitting may happen for some distributions if the NN trains too much. At the same time It'll be the best result for other distributions:



Picture 55: An example of image quality change during the training process

11.1.7 Generated spiders – Summary

It's difficult to choose one specific NN that will guarantee good fake spiders all the time. The NN that were used in (Sections 11.1.3 and 11.1.5) may be used to generate new types of spiders with different characteristics as it's shown in (Section 11.1.6).

It has been detected that Adam and RMSprop optimizers can be used to generate new images. Also the number of images in a batch size doesn't influence much on the final result.

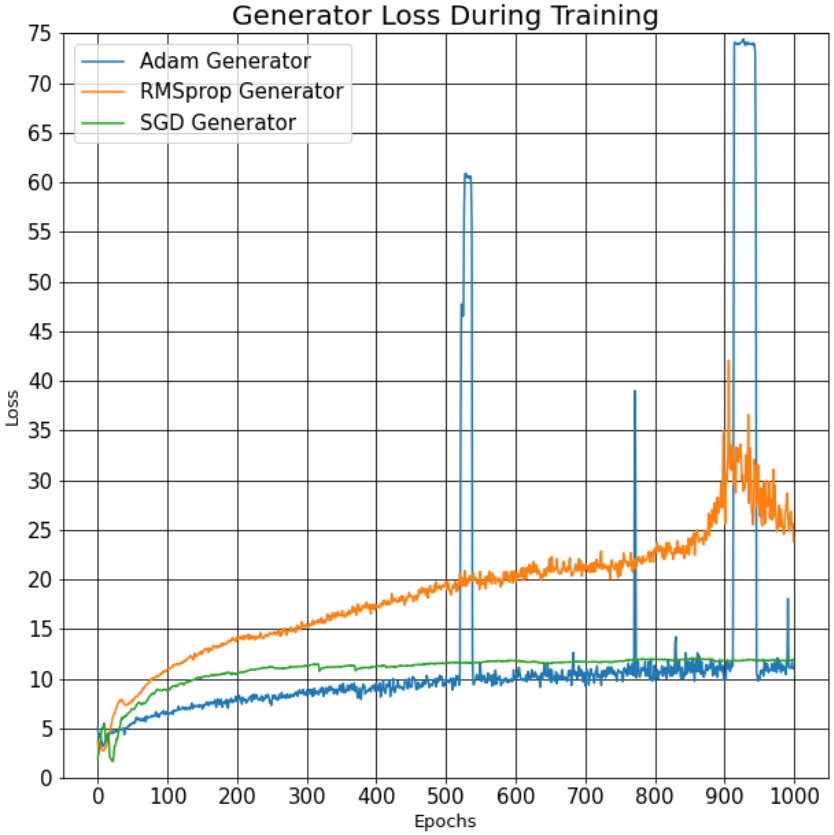
It's necessary to mention that two different losses such as 'L1Loss' and 'Wasserstein' loss were implemented but the results were not acceptable.

11.2 Loss results during the training process

This subsection shows the losses of generator and discriminator parts in DCGAN model that corresponds to achieved images results in (Section 11.1). Also it will be shown how the outputs look for generator and discriminator parts during the training process (Section 11.2.4).

11.2.1 Loss of different optimizers – Result №1

The following losses correspond to achieved results in (Section 11.1.1).

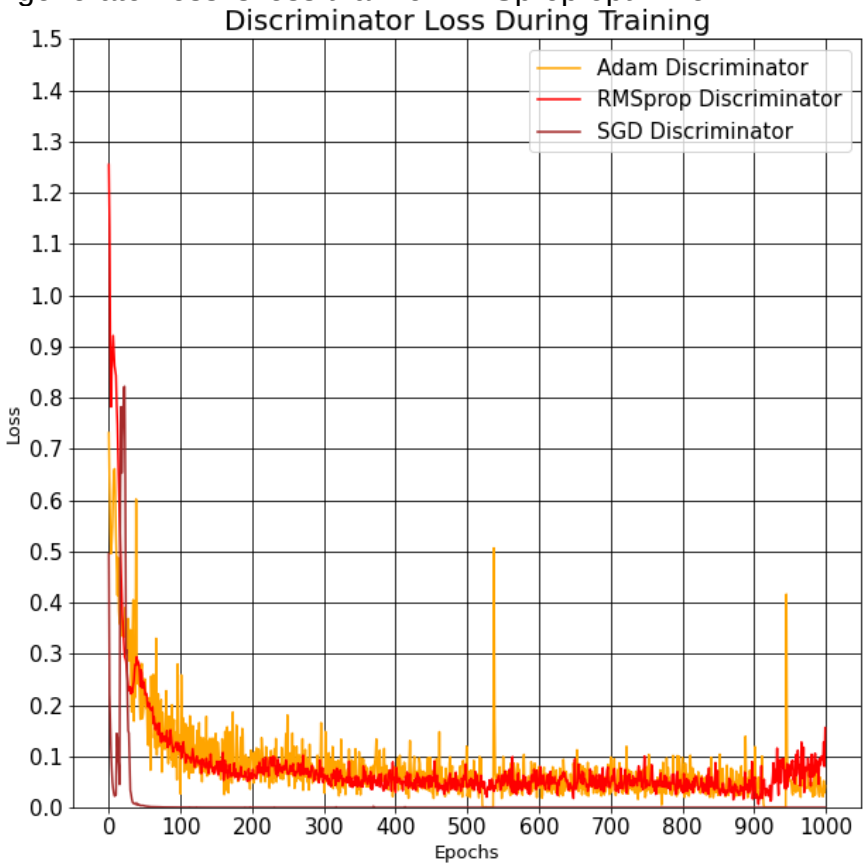


Picture 56: Generator loss, different optimizers, [id 2]

In the generator graph above, it's visible that the green line of SGD optimizer after the 200 epoch stay almost the same and it's smooth. It means that SGD optimizer does something wrong because the generator and discriminator are playing min-max game and little oscillations are expected as it's shown for Adam and RMSprop optimizer.

Sometimes big peaks appear especially for the blue line - Adam generator. In my opinion it can be caused by a little size of batch. In theory, the smaller the batch size, the worse the function is optimized.

Confirmation of this opinion will be demonstrated in (Section 11.2.3). Adam generator loss is less than for RMSprop optimizer.



Picture 57: Discriminator loss, different optimizers, [id 2]

In the above discriminator graph, the discriminator loss for SGD function is almost equal to 0 (and stay at this level) after the 40th epoch. This may be caused by two reasons: it learns too slow or it works wrong. As a result, it's expected a bad result for this optimizer (Picture 32).

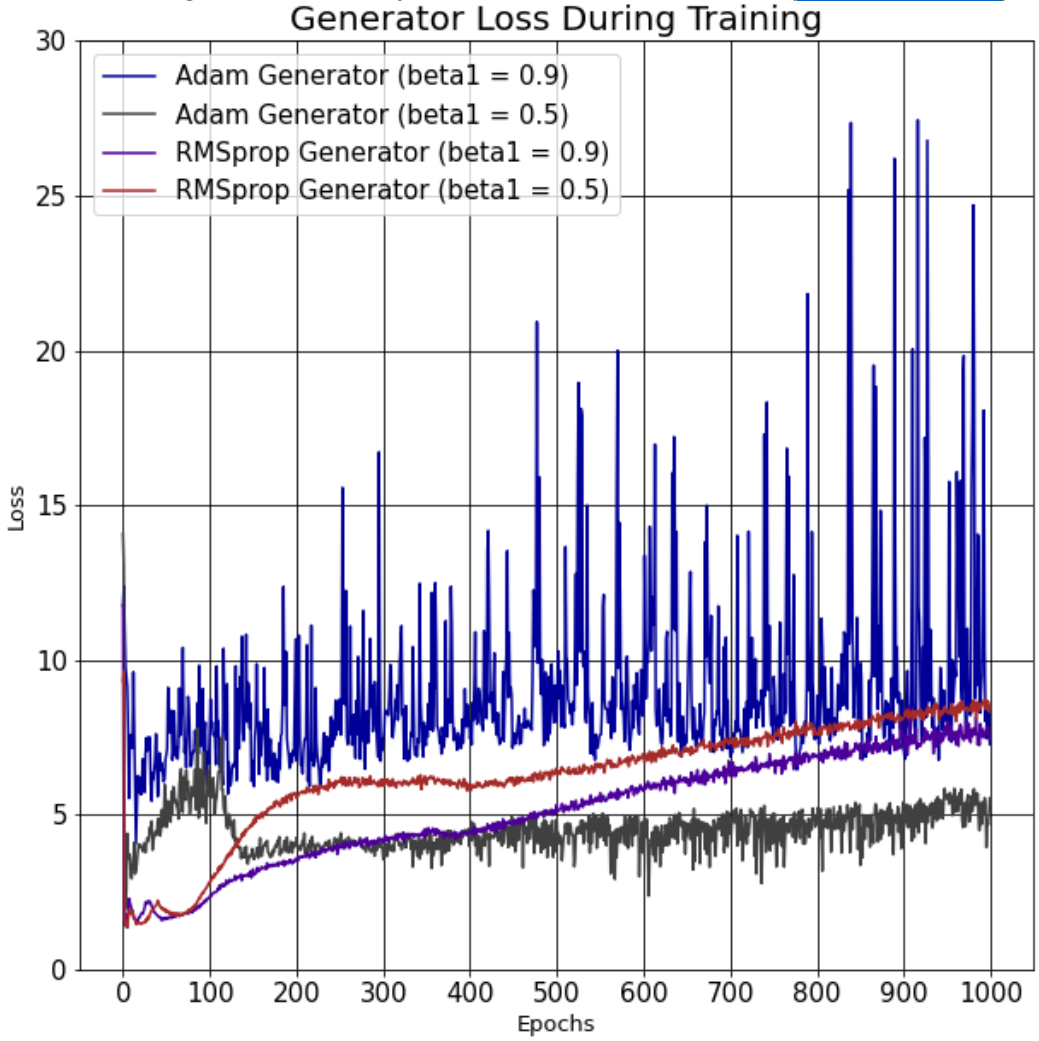
Some peaks in Adam discriminator are not big in reality - the scale is different comparing it to the generator graph.

The discriminator loss for the Adam optimizer is usually slightly larger than for the RMSprop optimizer.

Taking into account all these facts for generator and discriminator parts it points that the Adam optimizer for this NN should give better results than the SGD and the RMSprop optimizers. And this is indeed the case, as shown in (Section 11.1.1).

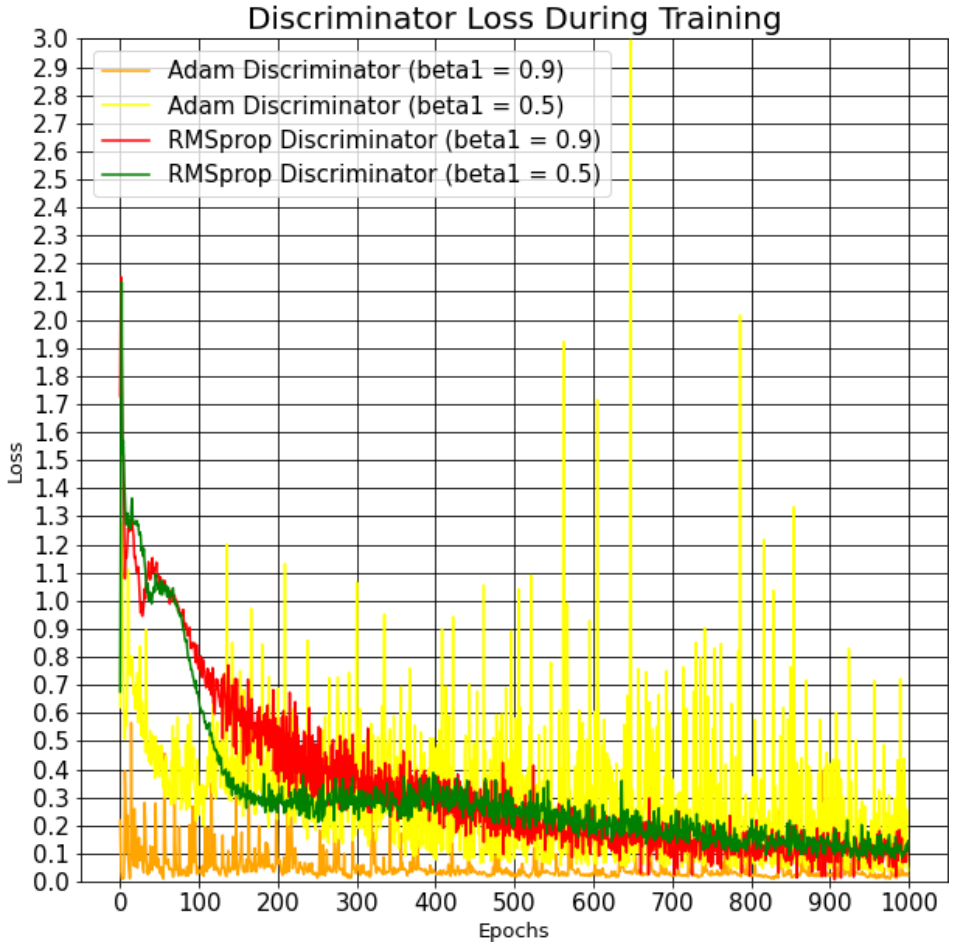
11.2.2 Loss for different beta1 parameter - Result №2

The following losses correspond to achieved results in (Section 11.1.2).



Picture 58: Generator loss, Adam & RMSprop optimizers, [id 9, 10]

In the above discriminator graph, It depicts how the generator is much more stable with beta1=0.5 (black line) than for beta1=0.9 (blue line). At the same time, the loss for both betas in RMSprop optimizer look almost the same but, the loss is less for beta1=0.9 . So, it's expected that Adam optimizer with beta1=0.5 and RMSprop optimizer with beta1=0.9 present better results than for beta1=0.9 and for beta1=0.5 respectively.



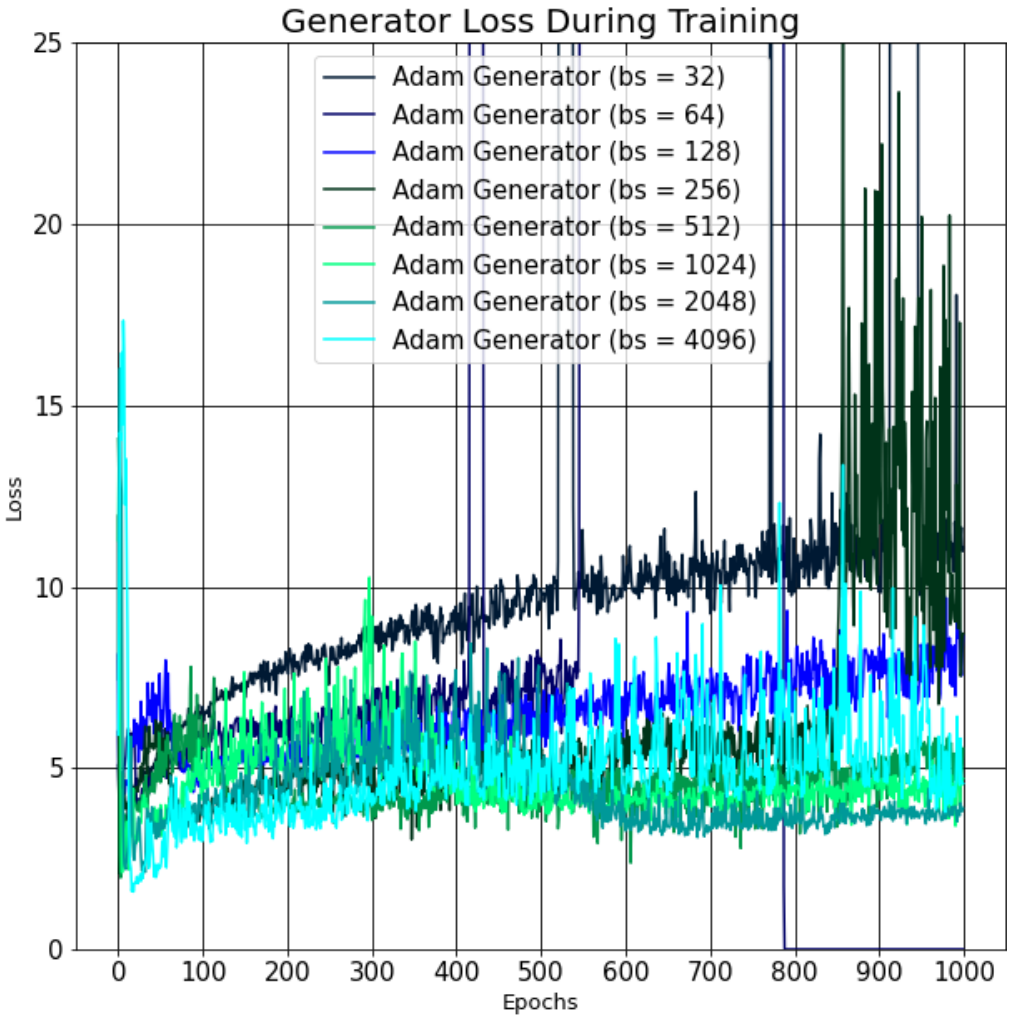
Picture 59: Discriminator loss for different beta1 parameter for Adam and RMSprop optimizers [id 9, 10]

Discriminator loss for beta1=0.9 is too small (close to 0 after the 400th epoch). The discriminator loss for both betas in RMSprop optimizer look similar.

Based on the conclusions from the generator and discriminator part, we can say that results should be acceptable for both betas in RMSprop optimizer and for beta1=0.5 in Adam optimizer. But beta1=0.9 is better in RMSprop optimizer than beta1=0.5. The same conclusion was made in (Section 11.1.2).

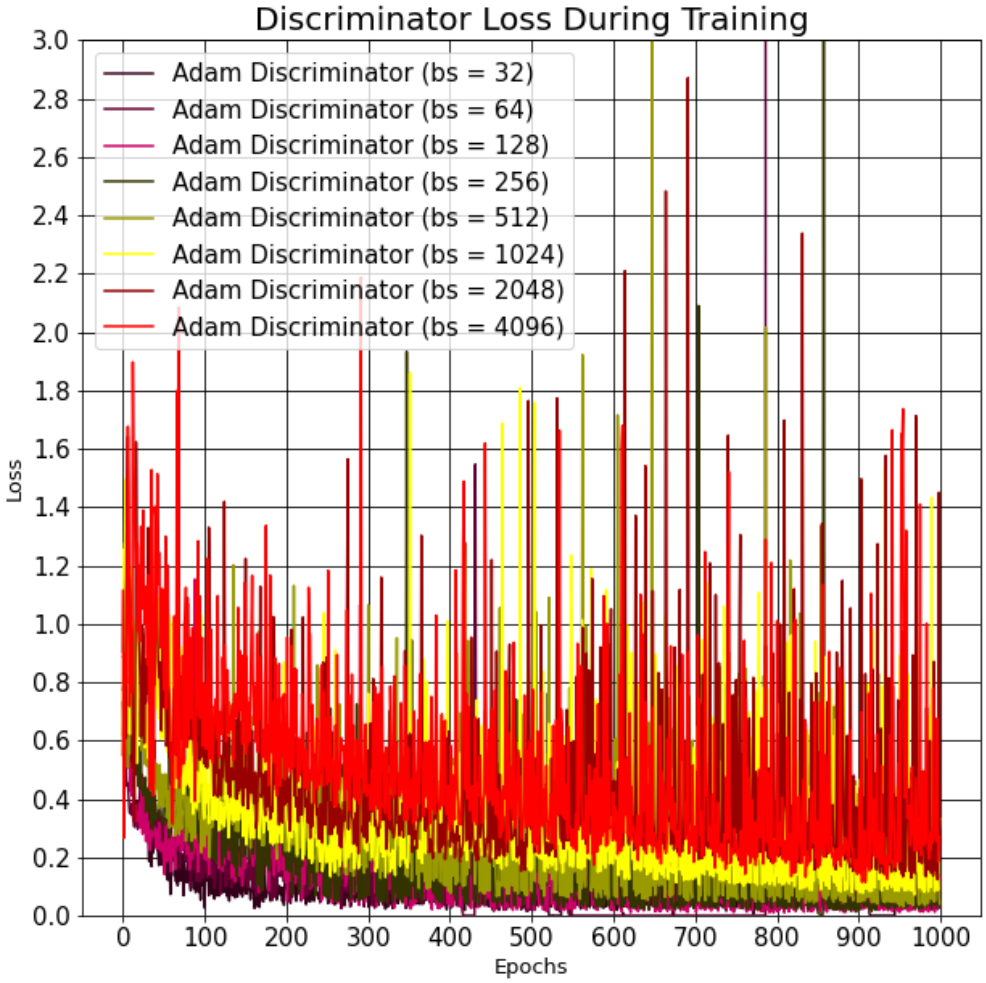
11.2.3 Loss for different batch size - Result №3

The following losses correspond to achieved results in (Section 11.1.3).



Picture 60: Generator loss, Adam optimizer, different batch size, [id 2,4,6,8,10,12,14,16]

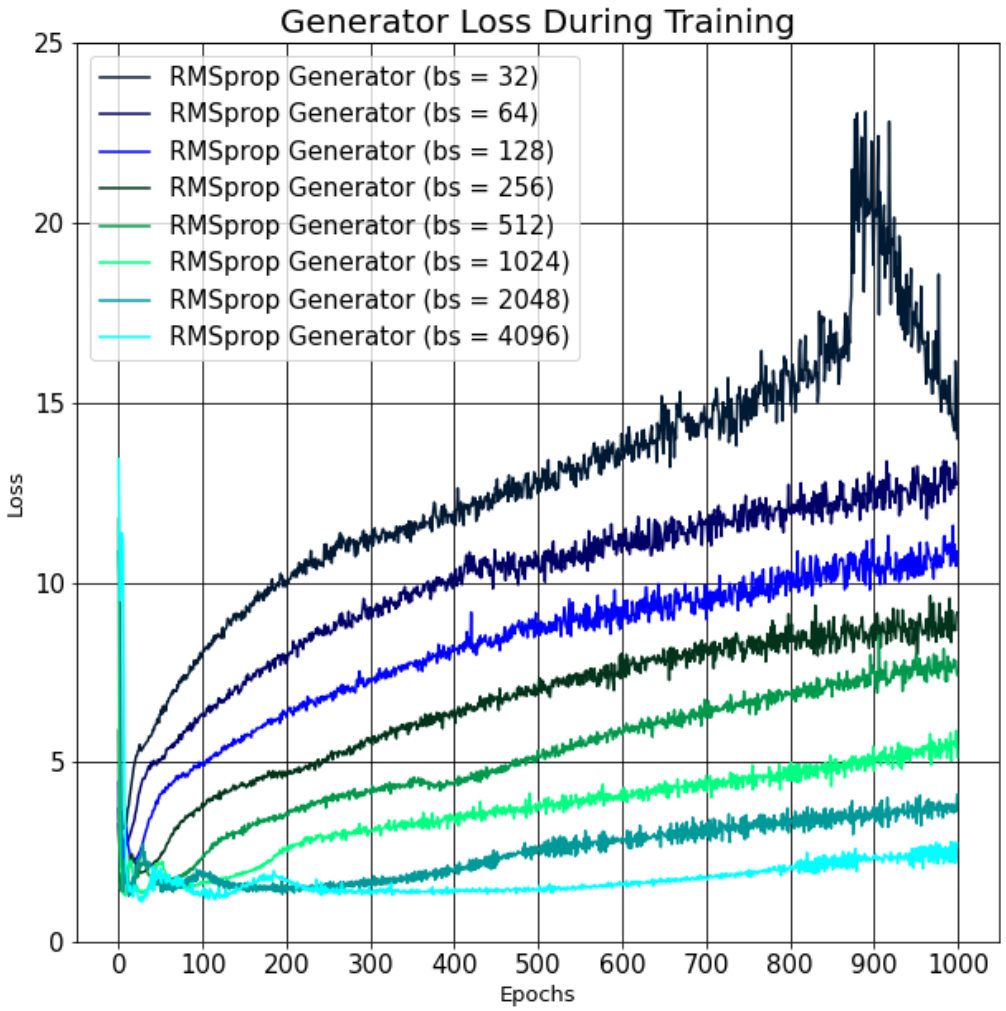
The generator loss of Adam decreases as the size of the batch increases. Larger the batch size, the more stable loss will occur later (for batch size=4096 - (Section 11.2.5)).



Picture 61: Discriminator loss, Adam optimizer, different batch size [id 2,4,6,8,10,12,14,16]

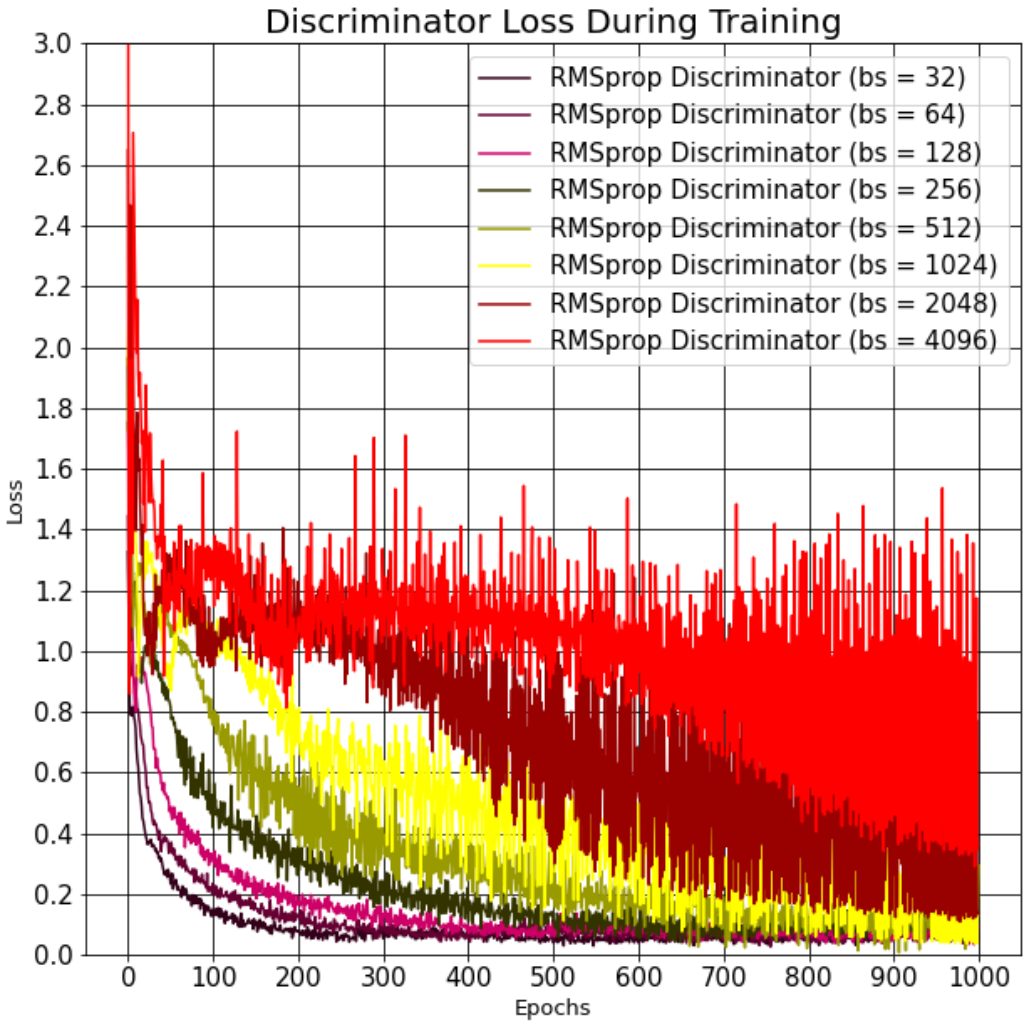
The discriminator loss of Adam optimizer increases as the size of the batch increases.

After the conclusions of the generator and discriminator parts, it can be assumed that, larger the batch size, the better will be the quality of the generated images for Adam optimizer. However, this is not so obvious and, in my opinion, is incorrect (Section 11.1.3).



Picture 62: Generator loss, RMSprop optimizer, different batch size [id 1,3,5,7,9,11,13,15]

The generator loss of RMSprop decreases as the size of the batch increases.



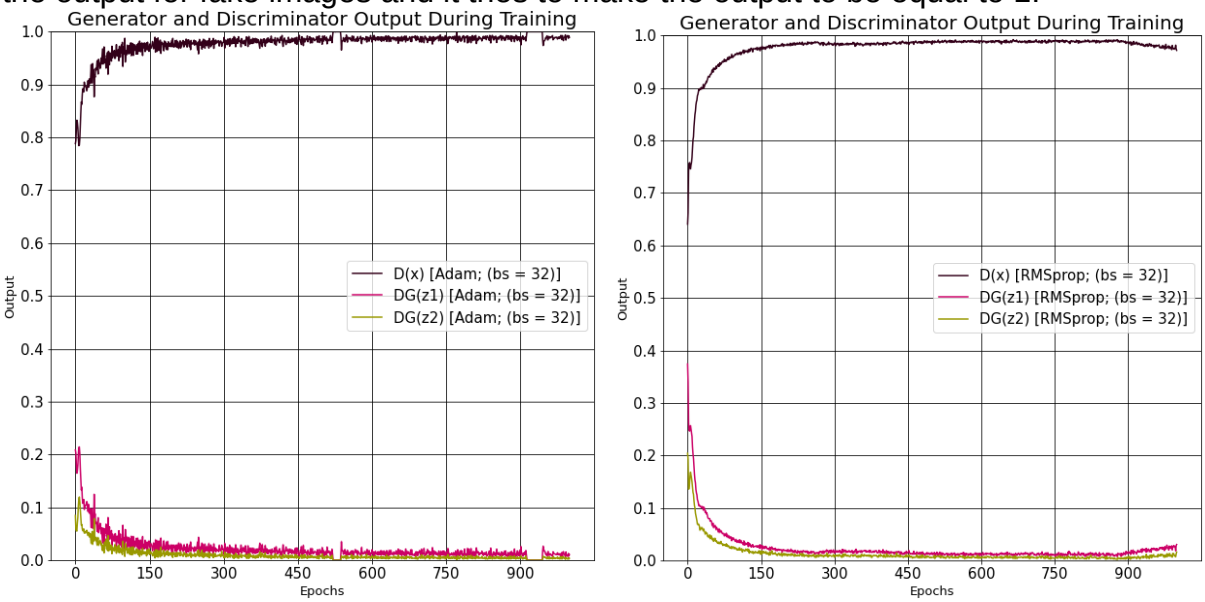
Picture 63: Discriminator loss, RMSprop optimizer, different batch size [id 1,3,5,7,9,11,13,15]

The discriminator loss of RMSprop optimizer increases as the size of the batch increases.

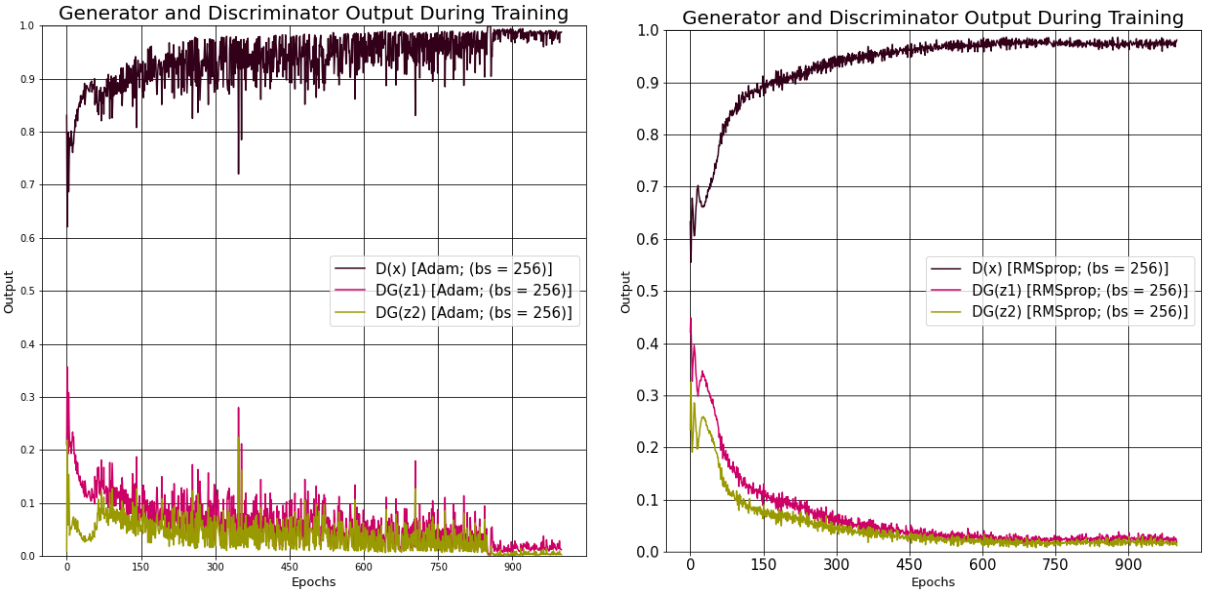
After the conclusions of the generator and discriminator parts, it can be assumed that, larger the batch size, the better will be the quality of the generated images for RMSprop optimizer. However, this is not so obvious and, in my opinion, is incorrect (Section 11.1.3).

11.2.4 Generator & Discriminator output for different batch size - Result №3

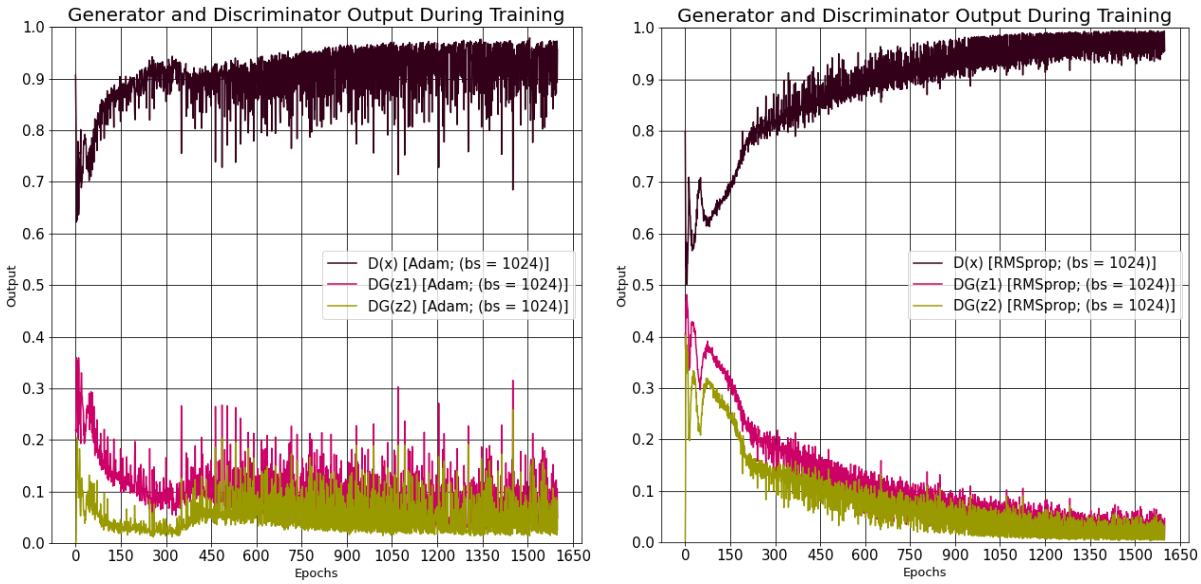
The section demonstrates the outputs of generator and two parts of discriminator according to the batch size. D(x) is the first part of discriminator. It shows the output for real images and it tries to make the output to be equal to 1. D(G(z1)) is the second part of discriminator. It shows the output for fake images and it tries to make the output to be equal to 0. D(G(z2)) is the part of generator. It shows the output for fake images and it tries to make the output to be equal to 1.



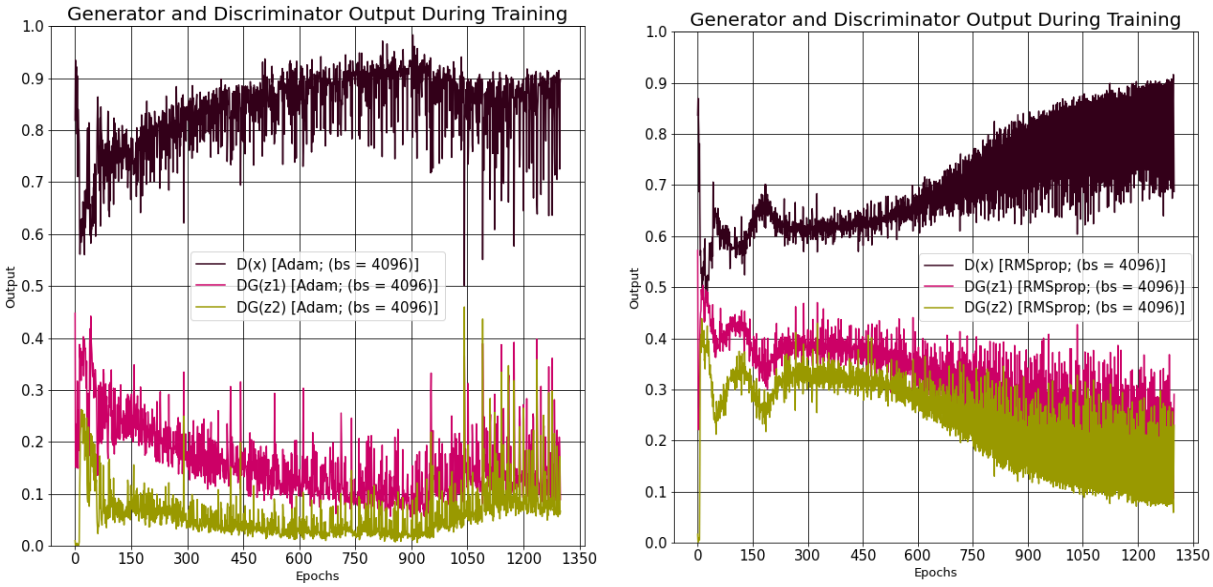
Picture 64: Output, batch size=32, Adam [id 2-left], RMSprop [id 1-right]



Picture 65: Output, batch size=256, Adam [id 8-left], RMSprop [id 7-right]



Picture 66: Output, batch size=1024, Adam [id 12-left], RMSprop [id 11-right]

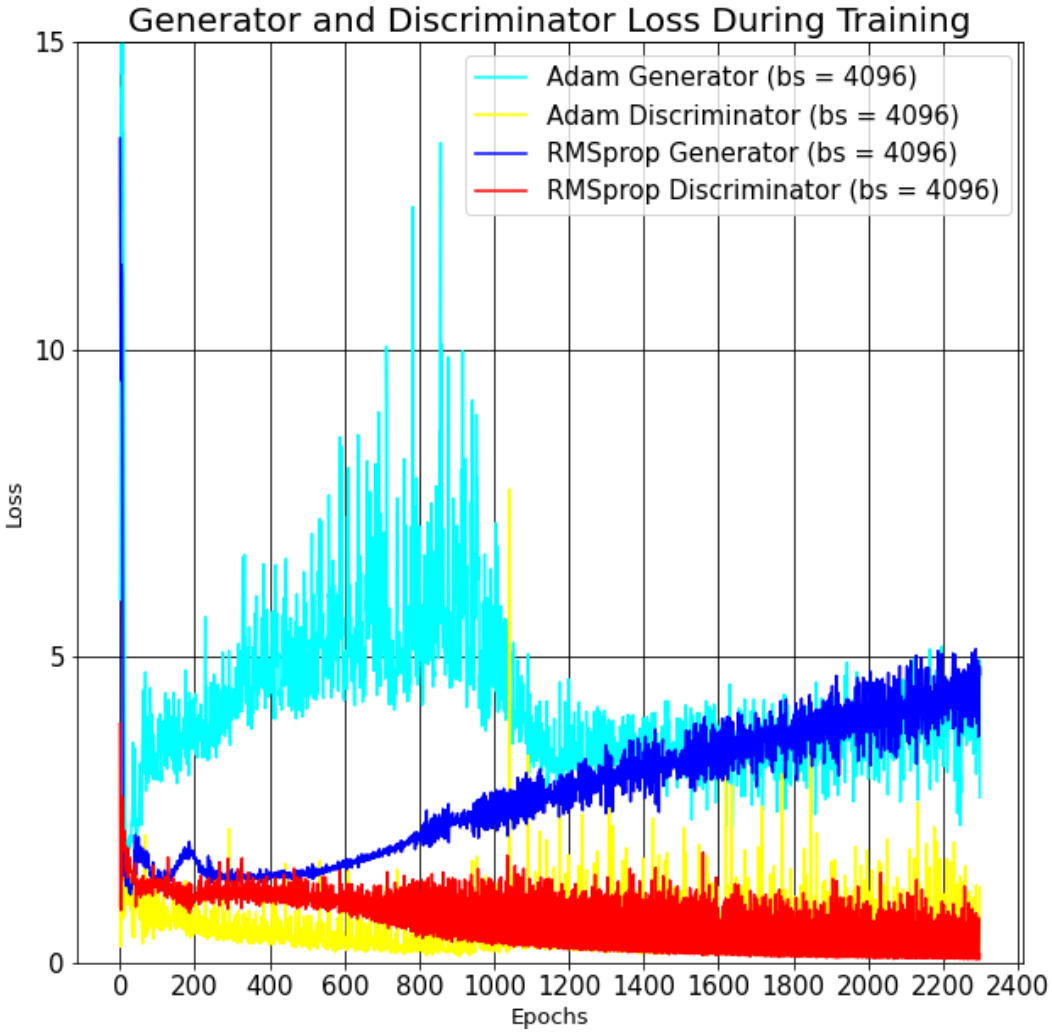


Picture 67: Output, batch size=4096, Adam [id 16-left], RMSprop [id 15-right]

The images above demonstrate how the batch size influences to the outputs. The perfect situation for DCGAN model is when all the outputs are equal to 0.5. Such an output could mean that discriminator tries randomly to guess whether the image is fake or real. Here, the bigger the batch size is, the more errors the discriminator make and the less errors the generator make. In this way, the best outputs are for a batch size=4096. However, the results of generated spiders are not so clear for this batch size (Section 11.1.3). It means that the generator could learn to generate bad images which the discriminator classify as real images.

11.2.5 Loss for batch size=4096 (2300 epochs) - Result №4

The section presents example of losses for a big number of epochs.

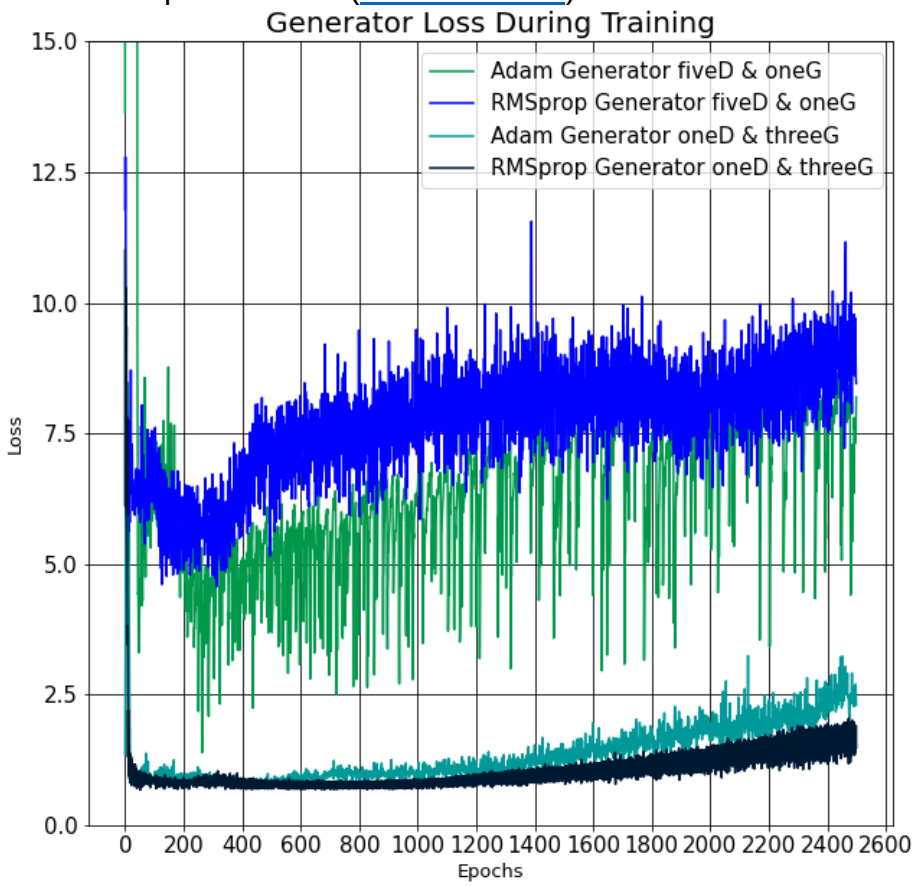


Picture 68: Generator loss, Adam & RMSprop optimizers, batch size=4096, [id 15, 16]

The Adam generator loss looks a little abnormal, but after the 100 epoch the loss stabilizes and begins to grow steadily over time. It means the bigger the batch size the more time is necessary to stabilize the loss. Unfortunately, the increase in epochs leads to the fact that the generator error gradually grows and the quality of the new generated images stop improving (Picture 45). Gradually, this process goes to overfitting.

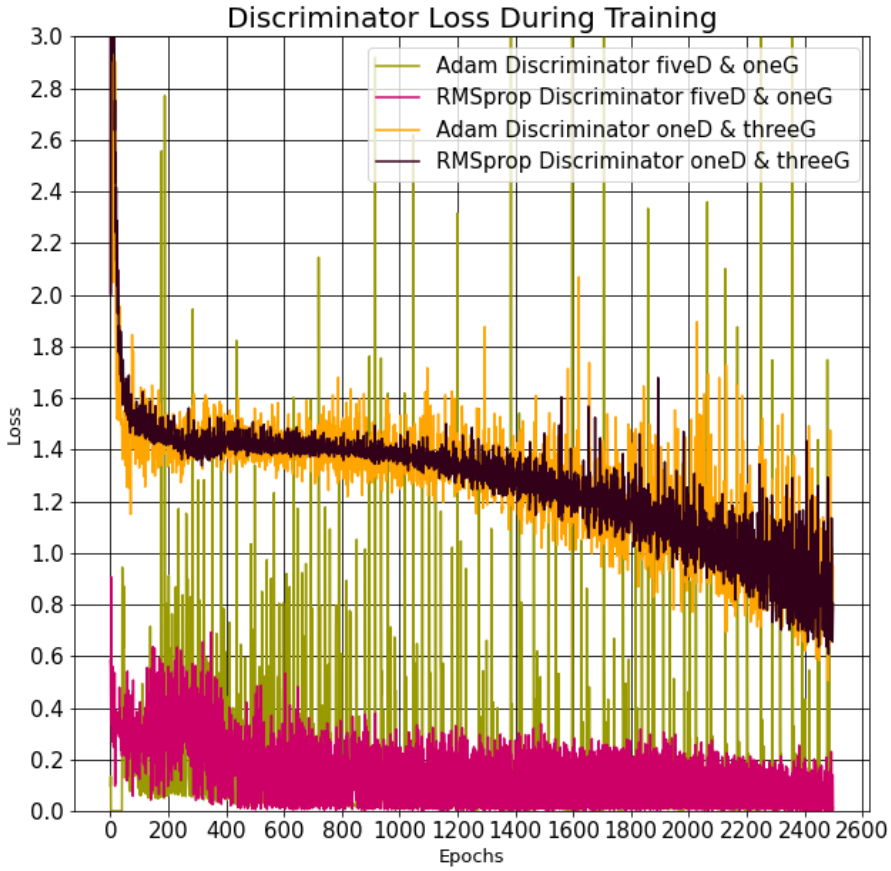
11.2.6 Loss for different number generator/discriminator updates - Result №5

The section presents the loss for two different models. The difference of these models in structure is presented in (Section 11.2.5).



Picture 69: Generator loss, Adam & RMSprop optimizers, batch size [id 10, 15]

As it was expected, the generator loss is much less when it updates several times before the discriminator update. Behavior of the loss is the same as it was shown in (Section 11.2.3).



Picture 70: Discriminator loss, Adam & RMSprop optimizers, batch size [id 10, 15]

As it was expected, the discriminator loss is much bigger when the generator updates several times before the discriminator update. Behavior of the loss is the same as it was shown in (Section 11.2.3).

As results, the quality of generated images should be similar to the generated images from (Section 11.2.3).

The fake spiders are similar for (Sections 11.1.3 and 11.1.5).

11.3 Info table

Table 1: Main information about different NN

ID	LOSS	OPTIMIZERS	BATCH SIZE	LEARNING RATE	BETA1	BETA2
1	BCELoss	Adam, RMSprop, SGD	32	0.0002	0.9	0.999
2	BCELoss	Adam, RMSprop, SGD	32	0.0002	0.5	0.999
3	BCELoss	Adam, RMSprop	64	0.0002	0.9	0.999
4	BCELoss	Adam, RMSprop	64	0.0002	0.5	0.999
5	BCELoss	Adam, RMSprop	128	0.0002	0.9	0.999
6	BCELoss	Adam, RMSprop	128	0.0002	0.5	0.999
7	BCELoss	Adam, RMSprop	256	0.0002	0.9	0.999
8	BCELoss	Adam, RMSprop	256	0.0002	0.5	0.999
9	BCELoss	Adam, RMSprop	512	0.0002	0.9	0.999

10	BCELoss	Adam, RMSprop	512	0.0002	0.5	0.999
11	BCELoss	RMSprop	1024	0.0002	0.9	0.999
12	BCELoss	Adam	1024	0.0002	0.5	0.999
13	BCELoss	RMSprop	2048	0.0002	0.9	0.999
14	BCELoss	Adam	2048	0.0002	0.5	0.999
15	BCELoss	RMSprop	4096	0.0002	0.9	0.999
16	BCELoss	Adam	4096	0.0002	0.5	0.999

12. Inception model

12.1 Introduction [31]

The Inception network was an important milestone in the development of CNN classifiers. It is assumed that this network may develop DCGAN generated images as well as CNN, which is a part of DCGAN structure.

The inception model is a complex (heavily engineered) model that is expected to improve performance in terms of speed, accuracy and quality of the generated images.

12.1.1 What problems arise in usual structure

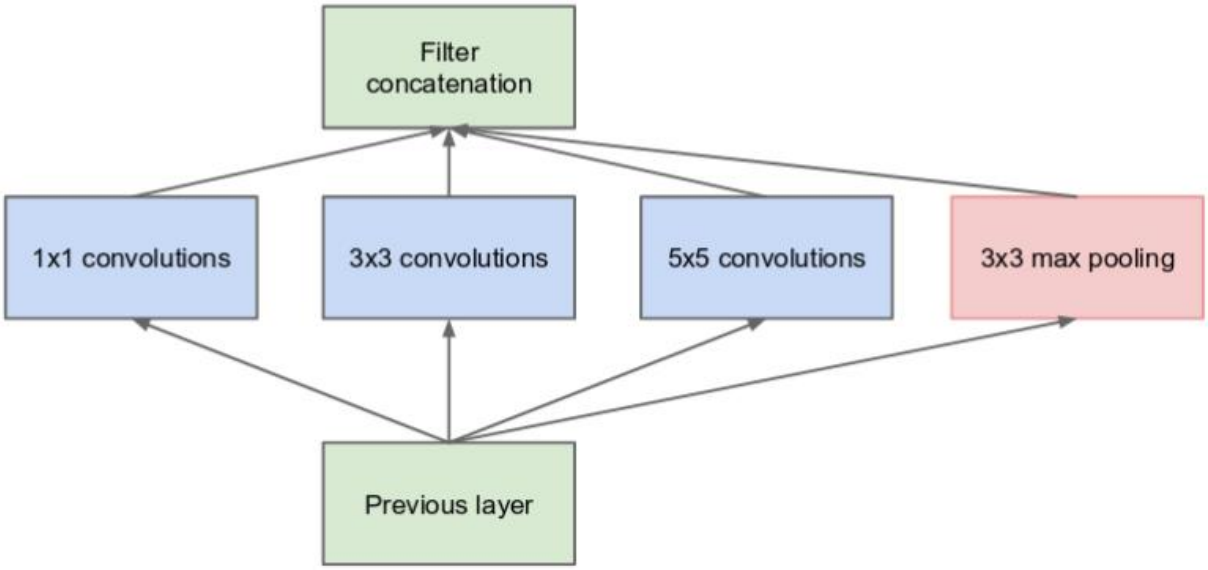
The main object can have extremely large variation in size in different images. The area occupied by the spider is different. It can be a very small or a very big spider comparing with the whole image (Section 9.2.2). Due to this huge variation in the location of the information, choosing the right kernel size for the convolution operation becomes tough. A larger kernel is preferred for information that is distributed more globally, and a smaller kernel is preferred for information that is distributed more locally.

Very deep networks are prone to overfitting. It also hard to pass gradient updates through the entire network.

Naively stacking large convolution operations are computationally expensive.

12.1.2 Some inception ideas & their purpose

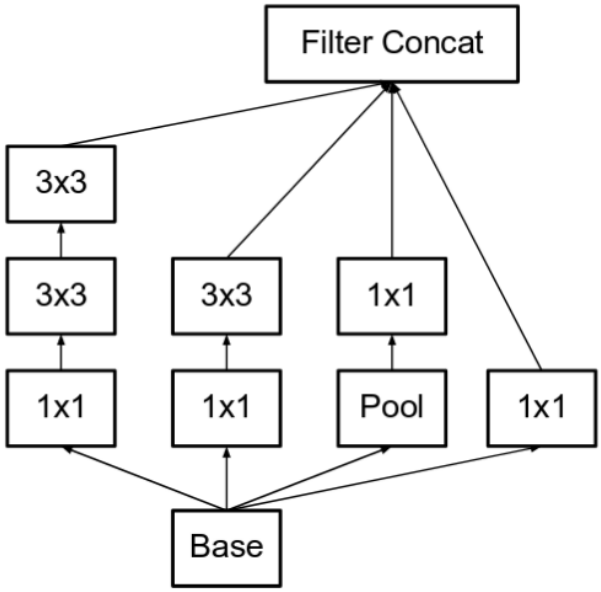
The first idea is to make a NN a bit “wider” rather than “deeper”. Example below performs convolution on an input, with 3 different sizes of filters (1x1, 3x3, 5x5). Additionally, max pooling is also performed. The outputs are concatenated and sent to the next layer.



Picture 71: The naive inception model [31]

The second idea is that NNs perform better when convolutions didn't alter the dimensions of the input drastically. Reducing the dimensions too much may cause loss of information, known as a "representational bottleneck". Hence, this idea assumes to use few small kernels instead of one big kernel. Moreover, such a step usually improves a computational speed.

A convolution of 5x5 size can be factorized on two 3x3 convolution operations. Although this may seem counterintuitive, a 5x5 convolution is 2.78 times more expensive than a 3x3 convolution. So, stacking two 3x3 convolutions leads to a boost in performance. This example is illustrated in the below image.

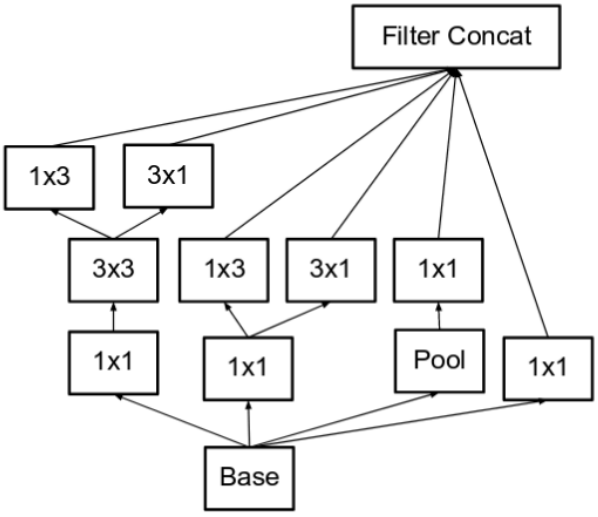


Picture 72: The inception v3 model, module "A" [31]

The third idea is to factorize convolutions of filter size NxN to a combination of 1xN and Nx1 convolutions.

A convolution of 3x3 size is equivalent to first performing a 1x3 convolution and then performing a 3x1 convolution on its output. It has been found that this

method is 33% more cheaper than the single 3x3 convolution. The example image is illustrated below.



Picture 73: The inception v3 model, module "C" [31]

12.2 Inception model – used structure

This model is mostly used for classification problems and for bigger images. But it's also been tried to use it in this thesis and the achieved results will be presented in a (Section 12.4).

Necessary to say that there is a function of inception v3 model in Pytorch framework. But, as it was already mentioned before, this function is intended for a classification problem and the structure of this function is too big (about 100 layers) because of the input image size that should be equal to 299x299. It's no sense to use such a complicated model for the images of size 64x64. Moreover, there is no any functions for a generator part. So, It was manually implemented only one of the blocks of the 'inception v3 model' function for a discriminator part and the reverse model for a generator part.

The discriminator is a binary classification network that takes an image as input and outputs a scalar probability that the input image is real. At the image below, the discriminator takes a 3x64x64 input image, processes it through a series of Conv2d, BatchNorm2d, LeakyReLU, AdaptiveAvgPooling, Dropout and Linear layers.

```
Discriminator(
  (Conv2d_1a_3x3): BasicConv2d(
    (conv): Conv2d(3, 64, kernel_size=(2, 2), stride=(2, 2), bias=False)
    (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (Conv2d_2a_3x3): BasicConv2d(
    (conv): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (Conv2d_2b_3x3): BasicConv2d(
    (conv): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (Mixed_7b): InceptionE(
    (branch1x1): BasicConv2d(
      (conv): Conv2d(128, 32, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(32, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3_1): BasicConv2d(
      (conv): Conv2d(128, 38, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(38, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3_2a): BasicConv2d(
      (conv): Conv2d(38, 38, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1), bias=False)
      (bn): BatchNorm2d(38, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3_2b): BasicConv2d(
      (conv): Conv2d(38, 38, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0), bias=False)
      (bn): BatchNorm2d(38, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_1): BasicConv2d(
      (conv): Conv2d(128, 44, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(44, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_2): BasicConv2d(
      (conv): Conv2d(44, 38, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(38, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_3a): BasicConv2d(
      (conv): Conv2d(38, 38, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1), bias=False)
      (bn): BatchNorm2d(38, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_3b): BasicConv2d(
      (conv): Conv2d(38, 38, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0), bias=False)
      (bn): BatchNorm2d(38, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch_pool): BasicConv2d(
      (conv): Conv2d(128, 19, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(19, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (dropout): Dropout(p=0.5, inplace=False)
  (fc): Linear(in_features=203, out_features=1, bias=True)
)
```

Picture 74: Discriminator structure in the inception model

In the image above, the number of feature maps is growing with each convolution layer before 'InceptionE' model. At the same time, the size of feature map shrinks twice with each convolutional layer. Then it's a 'InceptionE' structure that corresponds to Inception "C" model (Picture73). After this structure the size of the feature maps stay the same but the number of them grows. Then Pooling, Dropout and Linear layers are used to achieve a final output.

Generator consists of a series of strided two dimensional convolutional transpose layers, each paired with a 2d batch norm layer and a relu activation

```
Generator(
  (ConvTranspose2d_1a_3x3): BasicConvTranspose2d(
    (convTranspose): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(512, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (ConvTranspose2d_2a_3x3): BasicConvTranspose2d(
    (convTranspose): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (bn): BatchNorm2d(256, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (ConvTranspose2d_2b_3x3): BasicConvTranspose2d(
    (convTranspose): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (bn): BatchNorm2d(128, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (MixedTranspose_7b): InceptionTransposeE(
    (branch1x1): BasicConvTranspose2d(
      (convTranspose): ConvTranspose2d(128, 17, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(17, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3_1a): BasicConvTranspose2d(
      (convTranspose): ConvTranspose2d(128, 25, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0), bias=False)
      (bn): BatchNorm2d(25, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3_1b): BasicConvTranspose2d(
      (convTranspose): ConvTranspose2d(128, 25, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1), bias=False)
      (bn): BatchNorm2d(25, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3_2): BasicConvTranspose2d(
      (convTranspose): ConvTranspose2d(50, 50, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(50, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_1a): BasicConvTranspose2d(
      (convTranspose): ConvTranspose2d(128, 25, kernel_size=(1, 3), stride=(1, 1), padding=(0, 1), bias=False)
      (bn): BatchNorm2d(25, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_1b): BasicConvTranspose2d(
      (convTranspose): ConvTranspose2d(128, 25, kernel_size=(3, 1), stride=(1, 1), padding=(1, 0), bias=False)
      (bn): BatchNorm2d(25, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_2): BasicConvTranspose2d(
      (convTranspose): ConvTranspose2d(50, 50, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn): BatchNorm2d(50, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch3x3dbl_3): BasicConvTranspose2d(
      (convTranspose): ConvTranspose2d(50, 50, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(50, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
    (branch_pool): BasicConvTranspose2d(
      (convTranspose): ConvTranspose2d(128, 11, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn): BatchNorm2d(11, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (ConvTranspose2d_new1_3x3): BasicConvTranspose2d(
    (convTranspose): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (ConvTranspose2d_new2_3x3): BasicConvTranspose2d_2(
    (convTranspose): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  )
)
```

Picture 75: Generator structure in the inception model

In the image above, the number of feature maps is decreasing, while the size of feature map is growing.

12.3 Results of inception model

The aim of this section is to present what generated spiders have been achieved using the structure firm the (Section 12.2).

Picture 76: Best generated spiders for inception model (Adam [id 12 – left]-1500 ep; RMSprop [id 11 - right]-450ep)

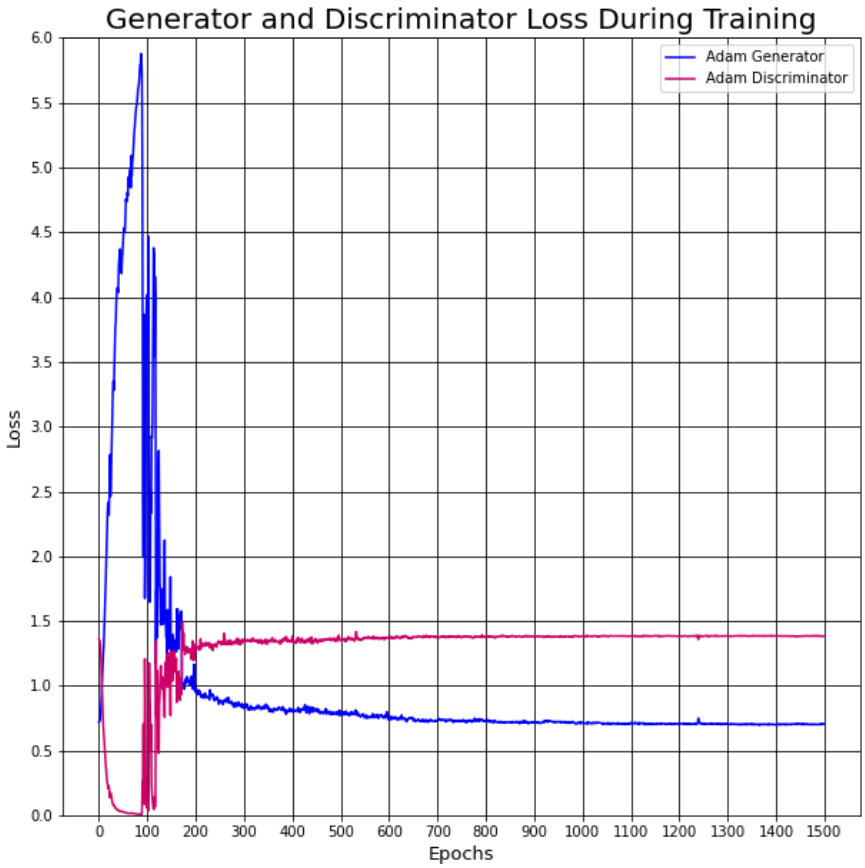
It has been achieved the images that remind spiders. But the quality of this spiders is much worse comparing with the spiders from (Sections 11.1.3 and 11.1.5).

Also, some of the examples have only noise. Most probably it was caused by complexity of the model or by overfitting.

12.4 Losses of inception model

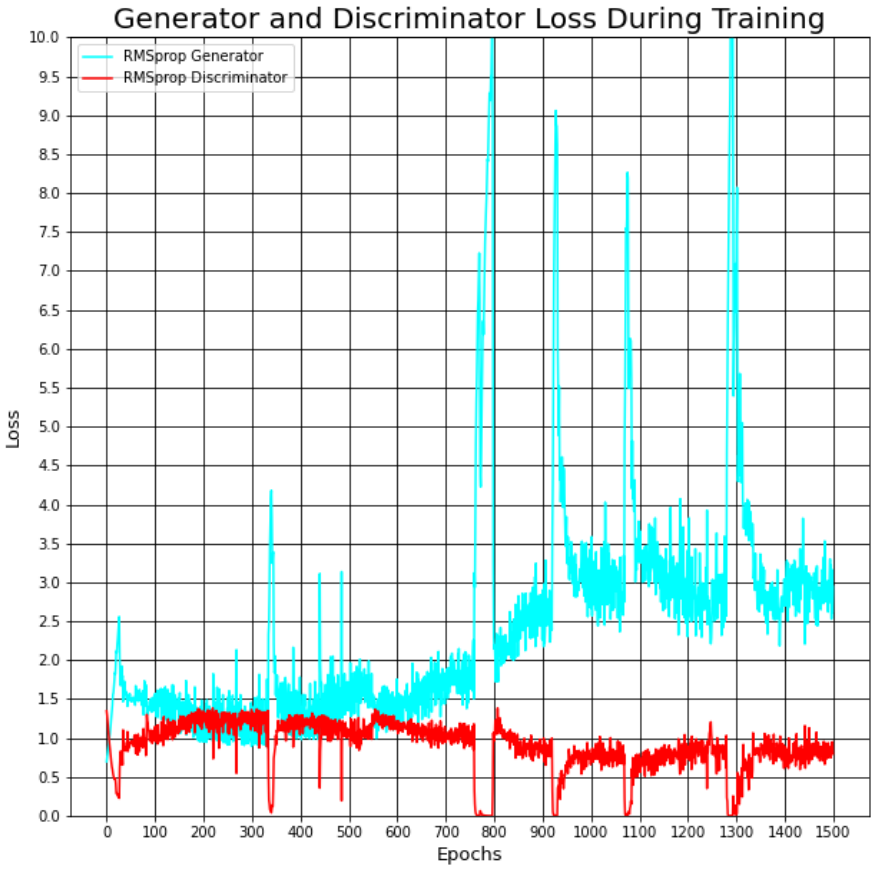
12.4.1 Generator & Discriminator loss in inception model for both optimizers

The section describes the behaviour of generator and discriminator losses in inception model.



Picture 77: Adam loss for inception model [id 12]

The losses for Adam optimizer look different comparing with (Section 11.2). The discriminator loss is higher than a generator loss. Both losses are small enough.



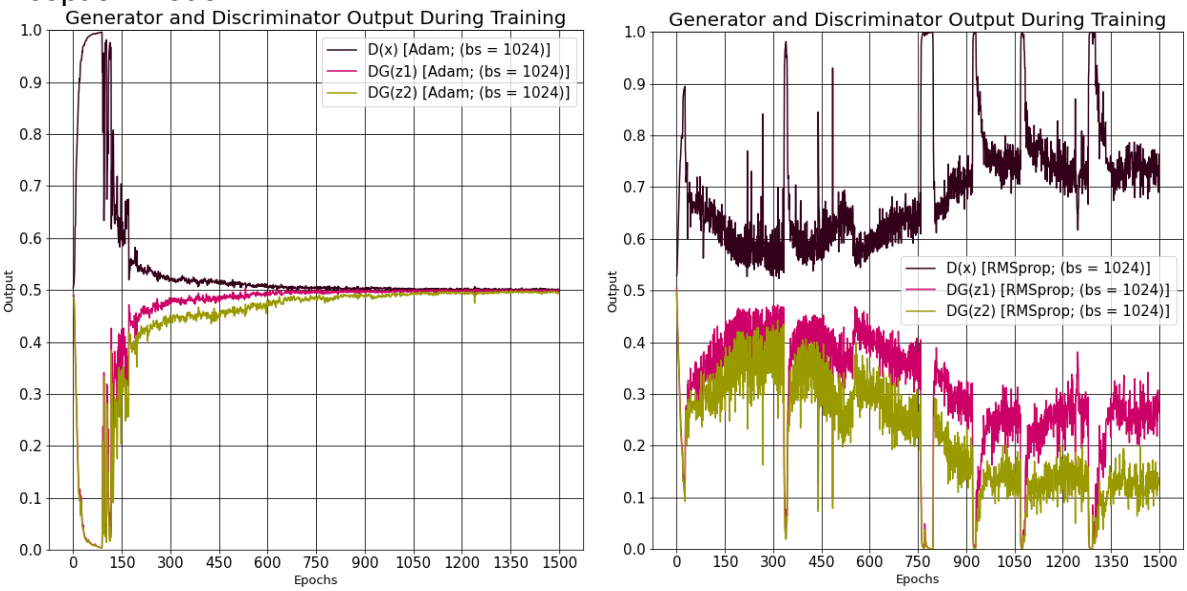
Picture 78: RMSprop loss for inception model [id 11]

The losses for RMSprop optimizer look similar comparing with (Section 11.2). There are some sticks in a generator loss but the similar problem has been described in (Section 11.2.1). The generator loss is small enough.

It could be expected good results for such small losses for both optimizers. Unfortunately, the generated spiders are not promising (presented above). Most probably, it's the problem of overfitting which often appears when the model os complicated.

12.4.2 Generator & Discriminator output in inception model for both optimizers

The section presents the outputs for generator and discriminator parts in inception model.



Picture 79: Output for inception model: Adam [id 12-left]; RMSprop [id 11-right]

Outputs for Adam optimizer are almost ideal but the results are not good enough. This fact also proves the fact of overfitting. RMSprop optimizer outputs look similar to the outputs in (Section 11.2.4).

13. Conclusions & future work

Eventually, different fake spiders have been achieved. There're different spiders with different characteristics (Section 11.1.6) as it's expected in the very beginning. The main disadvantage is that there's no NN that may generate different fake spiders of good quality continuously.

To solve the aforementioned problem other inception models may be implemented. SGD optimizer can be checked for a bigger learning rate parameter. Other quality metrics may be used to determine accurately which result is better.

14. Additional literature

A list of literature is presented, which was used to find a necessary information:

- Alec Radford, Luke Metz, Soumith Chintala (November, 2015). “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks” (PDF).
- Bing Xu, Naiyan Wang, Tianqi Chen, Mu Li (May, 2015). “Empirical Evaluation of Rectified Activations in Convolutional Network” (PDF).
- Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, Zbigniew Wojna (December, 2015). “Rethinking the Inception Architecture for Computer Vision” (PDF).
- Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet,

Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich (September, 2014). "Going Deeper with Convolutions".

- Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, Alex Alemi (August, 2016). "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning" (PDF).
- Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, Anil A Bharath (October, 2017). "Generative Adversarial Networks: An Overview" (PDF).

15. References

1. https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html (web page, visited 2020).
2. https://en.wikipedia.org/wiki/Generative_adversarial_network#cite_note-GANnips-1 (web page, visited 2020).
3. https://www.youtube.com/watch?v=O8LAi6ksC80&amp%3Blist=PLTl9hO2Oobd-1jxZ01__NjibEY6h15Kha&ab_channel=CodeEmporium (web page, visited 2020).
4. https://medium.com/@jos.vandewolfshaar/semi-supervised-learning-with-gans-23255865d0a4 (web page, visited 2020).
5. https://sigmoidal.io/beginners-review-of-gan-architectures/ (web page, visited 2020).
6. https://www.researchgate.net/figure/GAN-conditional-GAN-CGAN-and-auxiliary-classifier-GAN-ACGAN-architectures-where-x_fig1_328494719 (web page, visited 2020).
7. https://medium.com/analytics-vidhya/convolution-padding-stride-and-pooling-in-cnn-13dc1f3ada26 (web page, visited 2020).
8. https://towardsdatascience.com/understand-transposed-convolutions-and-build-your-own-transposed-convolution-layer-from-scratch-4f5d97b2967 (web page, visited 2020).
9. https://www.bigrabbitdata.com/pytorch-gan-2-implement-dcgan-with-mnist-fashion-mnist-deep-convoluted/ (web page, visited 2020).
10. https://laptrinhx.com/convolutional-neural-networks-with-keras-697984035/ (web page, visited 2020).
11. http://datahacker.rs/what-is-padding-cnn/ (web page, visited 2020).
12. http://makeyourownneuralnetwork.blogspot.com/2020/02/calculating-output-size-of-convolutions.html (web page, visited 2020).
13. Alec Radford, Luke Metz (2016). "Unsupervised representation learning with Deep Convolutional Generative Adversarial Networks" (PDF).
14. https://www.researchgate.net/figure/Commonly-used-activation-functions-a-Sigmoid-b-Tanh-c-ReLU-and-d-LReLU_fig3_335845675 (web page, visited

2021).

15. https://developers.google.com/machine-learning/gan/loss (web page, visited 2020).
16. Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio (June, 2014). " Generative Adversarial Nets" (PDF).
17. https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html (web page, visited 2020).
18. Sebastian Ruder (September, 2016). "An overview of gradient descent optimization algorithms" (PDF).
19. https://en.wikipedia.org/wiki/NumPy (web page, visited 2020).
20. https://en.wikipedia.org/wiki/PyTorch (web page, visited 2020).
21. https://en.wikipedia.org/wiki/Matplotlib (web page, visited 2020).
22. https://research.google.com/colaboratory/faq.html (web page, visited 2021).
23. https://en.wikipedia.org/wiki/Python_(programming_language) (web page, visited 2021).
24. https://www.kaggle.com/kdnishanth/animal-classification (web page, visited 2020).
25. https://www.kaggle.com/mistag/arthropod-taxonomy-orders-object-detection-dataset (web page, visited 2020).
26. https://www.kaggle.com/alessiocorrado99/animals10 (web page, visited 2020).
27. https://github.com/RyansQin/spider-recognition_dataset (web page, visited 2020).
28. https://bie.ala.org.au/ (web page, visited 2020).
29. Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, Xi Chen (June,2016). "Improved Techniques for Training GANs" (PDF).
30. Ali Borji (February, 2016). "Pros and Cons of GAN Evaluation Measures" (PDF).
31. https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202 (web page, visited 2021).

16. Abbreviations

AI - Artificial Intelligence.
NN – Neural Network.
GAN – Generative Adversarial Network.
CGAN – Conditional Generative Adversarial Network.
CNN – Convolutional Neural Network.
DCGAN – Deep Convolutional Generative Adversarial Network.
BCELoss – Binary Cross Entropy Loss function

SGD – Stochastic Gradient Descent
RMSprop – Root Mean Squared propagation

Picture 1: Original Vanilla GAN [4] 12
Picture 2: DCGAN [5] 12
Picture 3: Conditional GAN (CGAN) [6] 13
Picture 4: Stride representation in convolutions Equation 1: Calculation of the size of the output image [7] 15
Picture 5: A 6∗6 image is convolved with a 3∗3 kernel in convolutions [10] 16
Picture 6: Padding representation in convolutions [11] 16
Picture 7: Max pooling representation in convolutions [7] 17
Picture 8: Average pooling representation in convolutions [7] 17
Picture 9: Stride representation in transposed convolutions [8] 18
Picture 10: Interaction of the kernel with the input matrix in transposed convolutions [8] 19
Picture 11: A 2x2 image upsampled to 4x4 size [8] 19
Picture 12: Padding representation in transposed convolutions [8] 20
Picture 13: DCGAN guidelines [13] 21
Picture 14: Discriminator structure in the used code 22
Picture 15: Padding illustration 23
Picture 16: Filter illustration 23
Picture 17: Generator structure in the used code 24
Picture 18: Latent vector transformation 25
Picture 19: Filter illustration for transposed convolutions 25
Picture 20: Padding illustration for transposed convolutions 26
Picture 21: Used activation functions [14] 26
Picture 22: Dataset examples before preprocessing part 31
Picture 23: Dataset examples after preprocessing part 31
Picture 24: Different spider species 31
Picture 25: Different spider sizes 31
Picture 26: Background with easily distinguishable spiders 32
Picture 27: Background with difficult distinguishable spiders 32
Picture 28: Spiders from different angles 32
Picture 29: Spiders without certain parts of the body 32
Picture 30: Generated spiders, Adam optimizer, 650 epoch [id 2] 35
Picture 31: Generated spiders, RMSprop optimizer, 450 epoch [id 2] 35
Picture 32: Generated spiders, SGD optimizer, 1000 epoch [id 2] 36
Picture 33: Generated spiders, Adam optimizer, 600 epoch [id 9-left], 950 epoch [id 10-right] 37
Picture 34: Generated spiders, Adam optimizer, 600 epoch [id 3-left], 450 epoch [id 4-right] 37
Picture 35: Generated spiders, RMSprop optimizer, 1000 epoch [id 9-left], 450 epoch [id 10-right] 38
Picture 36: Generated spiders, RMSprop optimizer 400 epoch [id 3-left], 550 epoch [id 4-right] 38
Picture 37: Generated spiders, batch size=32, Adam: 650 epoch [id 2-left], RMSprop: 400 epoch [id 1-right] 39
Picture 38: Generated spiders, batch size=64, Adam: 450 epoch [id 4-left], RMSprop: 400 epoch [id 3-right] 39

Picture 39: Generated spiders, batch size=128, Adam=650 epoch [id 6-left], RMSprop: 300 epoch [id 5-right] 40
Picture 40: Generated spiders, batch size=256, Adam: 650 epoch [id 8-left], RMSprop: 950 epoch [id 7-right] 40
Picture 41: Generated spiders, batch size=512, Adam: 950 epoch [id 10-left]; RMSprop: 1000 epoch [id 9-right] 41
Picture 42: Generated spiders, batch size=1024, Adam: 1300 epoch [id 12-left], RMSprop: 1500 epoch [id 11-right] 41
Picture 43: Generated spiders, batch size=2048, Adam: 1200 epoch [id 14-left], RMSprop: 1300 epoch [id 13-right] 42
Picture 44: Generated spiders, batch size=4096, Adam: 1300 epoch [id 16-left], RMSprop: 1250 epoch [id 15-right] 42
Picture 45: Best generated spiders for batch size equals 4096 (2300 epochs): Adam [id 16-left (1600 epoch)]; RMSprop [id 15-right (1550 epoch)] 43
Picture 46: Generated spiders, Adam optimizer, (2250 epoch) [id 10-left], (1500 epoch) [id 16-right] 44
Picture 47: Generated spiders, RMSprop optimizer, (2500 epoch) [id 9-left], (1500 epoch) [id 15-right] 44
Picture 48: Different spider species 45
Picture 49: Spiders from different angles 45
Picture 50: Background with easily distinguishable spiders 45
Picture 51: Background with difficult distinguishable spiders 45
Picture 52: Spiders without certain parts of the body 45
Picture 53: Spiders on different backgrounds 46
Picture 54: An example of a training process 46
Picture 55: An example of image quality change during the training process 46
Picture 56: Generator loss, different optimizers, [id 2] 47
Picture 57: Discriminator loss, different optimizers, [id 2] 48
Picture 58: Generator loss, Adam & RMSprop optimizers, [id 9, 10] 49
Picture 59: Discriminator loss for different beta1 parameter for Adam and RMSprop optimizers [id 9, 10] 50
Picture 60: Generator loss, Adam optimizer, different batch size, [id 2,4,6,8,10,12,14,16] 51
Picture 61: Discriminator loss, Adam optimizer, different batch size [id 2,4,6,8,10,12,14,16] 52
Picture 62: Generator loss, RMSprop optimizer, different batch size [id 1,3,5,7,9,11,13,15] 53
Picture 63: Discriminator loss, RMSprop optimizer, different batch size [id 1,3,5,7,9,11,13,15] 54
Picture 64: Output, batch size=32, Adam [id 2-left], RMSprop [id 1-right] 55
Picture 65: Output, batch size=256, Adam [id 8-left], RMSprop [id 7-right] 55
Picture 66: Output, batch size=1024, Adam [id 12-left], RMSprop [id 11-right] 56
Picture 67: Output, batch size=4096, Adam [id 16-left], RMSprop [id 15-right] 56
Picture 68: Generator loss, Adam & RMSprop optimizers, batch size=4096, [id 15, 16] 57
Picture 69: Generator loss, Adam & RMSprop optimizers, batch size [id 10, 15] 58
Picture 70: Discriminator loss, Adam & RMSprop optimizers, batch size [id 10, 15] 59
Picture 71: The naive inception model [31] 61
Picture 72: The inception v3 model, module "A" [31] 61
Picture 73: The inception v3 model, module "C" [31] 62
Picture 74: Discriminator structure in the inception model 63
Picture 75: Generator structure in the inception model 64
Picture 76: Best generated spiders for inception model (Adam [id 12 – left]-1500 ep;

RMSprop [id 11 -right]-450ep)..... 65
Picture 77: Adam loss for inception model 66
Picture 78: RMSprop loss for inception model 67
Picture 79: Output for inception model: Adam [id 12-left]; RMSprop [id 11-right].. 68

Table 1: Main information about different NN 59