TANSZÉKVEZETŐ

SZAKDOLGOZAT FELADAT

Élő Zsombor
Mérnökinformatikus hallgató részére

Multiple Input Transfer Learning Generative Adversarial Network (MITLGAN) alkalmazása valósághoz hű, mindennapi képek generálásához

A ma ismert GAN megvalósításoknak gyakran visszatérő hibája, hogy a generált adat egy, vagy akár több részlete is hibás, nem valósághoz hű. Például egy portrén egybe mosódik a haj a háttérrel vagy a szemüvegben rossz tükröződés jelenik meg. A hallgató feladata egy általa kigondolt, új típusú GAN (MITLGAN) készítése, amelynek generátora előre megedzett, egyszerű generátorokat köt össze, hogy egy olyan kimenő adatot adjon eredményül, amely az előedzett hálók sajátosságait kombinálja a transfer learning és egyéb Deep Learning módszerek segítségével.

A dolgozatban többféle adathalmazt kell használnia a hallgatónak, hogy alá tudja támasztani a modelljének működését, felhasználhatóságát. A modell eredményeit össze kell hasonlítani már meglévő GAN modellekéivel, és összehasonlítani, hogy a MITLGAN hogyan teljesít hozzájuk képest.

A hallgató feladatának a következőkre kell kiterjednie:

- Mutassa be a GAN modellek alapvető működését és hogy miben más a javasolt modell!
- Készítse el a MITLGAN alapú megoldást!
- Ismertesse az adathalmazokat és hogy mely adathalmazokat hogyan használta fel!
- Prezentálja a modell működését különböző grafikonok, mérőszámok segítségével!
- Elemezze a kapott eredményeket, azokat hasonlítsa össze már meglévő megoldásokkal!

Tanszéki konzulens: Pomázi Krisztián, doktorandusz

Budapest, 2021. október 4.

Dr. Charaf Hassan
egyetemi tanár
tanszékvezető

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék
H-1117 Budapest, Magyar tudósok krt 2. Q.B.207
Telefon: 463-2870. Fax: 463-2871
www.aut.bme.hu

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics

Multiple Input Transfer Learning Generative Adversarial Network (MITLGAN) based realistic image generation

Bachelor's Thesis

Author
Zsombor Élő

Advisor
Krisztián Dániel Pomázi

December 10, 2021

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Élő Zsombor*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2021. december 10.

Élő Zsombor
hallgató

Kivonat

A Generative Adversarial Networks (GAN) egy state-of-the-art neurális hálózat modell, ami 2 dologra képes: realisztikus adatot gyártani, illetve megkülönböztetni valódi és hamis (vagy kigenerált) adatot. Ebben a szakdolgozatban a generálás részre koncentrálunk és magának a modellnek az edzési fázisára.

A mi esetünkben az adat képeket jelent, és a cél az, hogy javísunk az eredmény megfelelő aspektusain, valamint ennek az elérési útján. Manapság is létezik már sok nagyszerű változata a GAN-nak, amik ténylegesen megtévesztően hitelesen tudnak generálni valósághoz hű adatokat, de ezen modelleknek is megvannak a maguk apróbb hibái. Egy ilyen apróbb hiba az, hogy egy újonnan készített képen összemosódik az előtér és a háttér nagyon vékony dolgok miatt (például hajszál), vagy más indokok miatt (a színek nagyon hasonlóak az előtéren és a háttéren). Ez a tézis kipróbál egy olyan megközelítést, ami ezt a hibát célozza meg orvosolni. Mivel a számító kapacitás véges volt (GPU idő és kapacitás különösen) és néhány esetben nem volt elég, már rendelkezésre álló adat, ezért ez a megoldás a jövőbeni potenciálját szeretné megmutatni, korai edzési eredmények és meglévő modellekhez lévő különbségek segítségével, három különböző esetben.

Mivel ez a megoldási javaslat eltér már meglévő modellektől, ezért egy új modellt építünk, inspirációt nyerve más modellektől (WGAN és ResNet). A lényeg az, hogy a modellünk generálásért felelő részében több (jelen esetben 2) neurális hálót (WGAN) kötünk össze, amik az előtér és a háttér megfelelő részeit már képesek kigenerálni. Ezután ezeket a köztes eredményeket azzá transzformáljuk, amit ténylegesen szeretnénk elérni (ResNet alkalmazásával). A kísérletnek voltak ígéretes eredményei, amik későbbi projektekben felhasználásra kerülhetnek.

Abstract

Generative Adversarial Networks (GAN) is a state of the art neural network model, which aims at two objective: generating realistic data and differentiating between real and fake (or generated) data. This Bachelor's thesis concentrates on the generating part of the model and overall the training process of the model.

In our case the data are images, and the goal is to improve on certain qualities of the generated pictures and the training process of acquiring these better products. These days there are many great variants of GAN, that actually can produce very real-like data, but in many cases some minor issues come up in mind, when looking at the newly got data. These little details include a case, where the background of an image gets mixed with the foreground. This can be really seen, when on the photo there are some really thin objects (like hair) or other disturbing things (the colors of the foreground and background are very similar), and those tend to ruin the result in a way, that gives away whether the data is real or fake. This thesis is experimenting an approach, that is trying to fix this issue. As there were not enough computing power (GPU time and power especially) and in some situations enough data, this solution is trying to explain its potential to work showing early training results and the difference to already existing solutions, using three different cases.

As the idea of this solution is different from already existing models, we are making a brand new model using already existing models (WGAN and ResNet) as inspirations. The main thing is, that in the generator part of the model we connect multiple (in our case two) neural nets (in our case WGANs), that produce some features of the foreground and background separately and we transform the joined results of them into what we really want (with ResNet). The experiment shows some very promising results, that can be used in further projects.

Chapter 1

Introduction

As lately in humanity's history, every objective, that seemingly can be made by machines without someone's interruption, was attempted to automatized by certain technologies. These include everyday chores like washing clothes, baking a pie etc., but also very simple things, like recognizing objects, moving objects.

Of course these tasks are relatively easy for us humans, because the evolution didn't fail us and eventually developed instincts, thought processes and habits to ensure survival on Earth. But machines and computers doesn't have that much time to "learn" to do these exercises. First we gave them exact "instructions", algorithms to accomplish what we want them to. That worked fairly well in many cases, like adding numbers together or showing the right time on our clocks. But this approach could be only successful, when all of the steps are known beforehand and the only thing remaining is to implement some method, some machine and later on some computer code. When we talk about algorithms that we "just know", because we learned it in the process of growing up, like recognizing food, danger or drawing a picture of an animal, the situation changes quickly. If the person making the solution doesn't know exactly, how the process works, won't be able to make something, that is accurate most of the time. Because of this issue, the demand to develop a technology, that can "learn" like us, humans, was growing each day. If this goal is accomplished, many workload can be taken from people, making life easier for everyone.

One of the solutions, that found its way into our everyday life, is machine learning. This collective concept proved through time its worth, showing rapid increase in accomplishing tasks, that seemed to only be doable by humans. These tasks include also recognizing objects, generating data, identifying some qualities of data. There are some objectives, that even a person can found hard at first, like paint a portrait that is similar to Picasso's art. Also in these cases, machine learning can make a better work than most of humanity.

There are many machine learning techniques, but when we talk about generating data, one state-of-the-art concept comes to mind right away, which is Generative Adversarial Networks (GAN) [8]. The main concept of this model is to use 2 neural network: a generator and a discriminator, that compete each other. The generator tries to make realistic data to fool the discriminator, the discriminator tries to distinguish between the generator's work and real data (figure 1.1). GAN is a rapidly changing branch of machine learning, many implementation for many tasks are made every year, such as DCGAN [21], StyleGAN2 [17] or WGAN [32]. As this concept is very new, there is room for improvement. The goal of this thesis is to experiment in a way, that possibly can show a way for further evolving in this area.

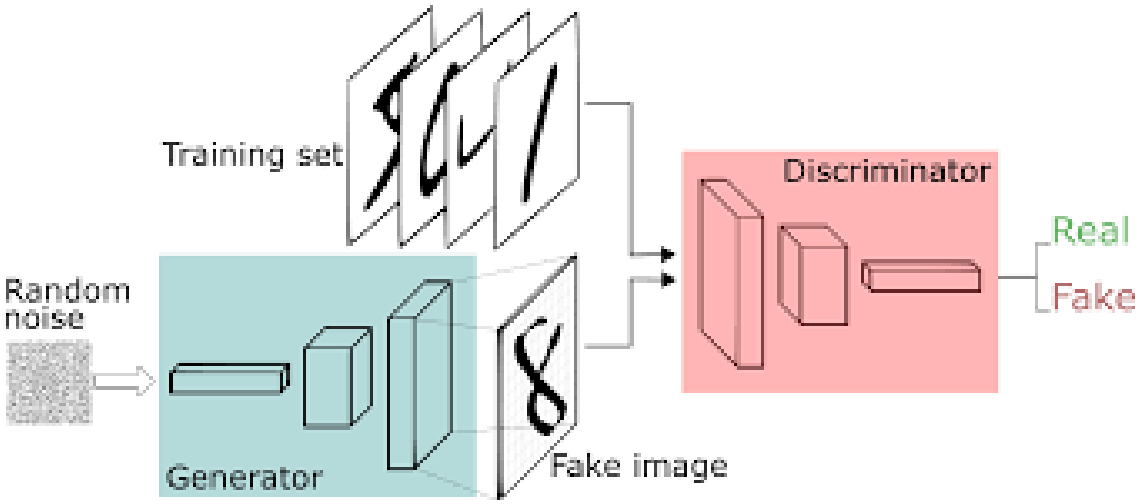

Figure 1.1: An example model for a GAN. Source: [7]

1.1 Problem statement

Generating data was never an easy task, many research and experiment is trying to deal with it. As it is something, that is from a human's perspective a creative exercise, the possibilities to improve are endless. In our mind, many nuances contribute to make an entirely new thing, based on what we already know and the circumstances. There are many solutions, but each and every one of them can be improved in my opinion, therefore I made my attempt to do so.

In my fifth semester did I first get to know machine learning and deep learning, and what they are capable of really. In the first half of that semester I concentrated on understanding the basics of the technology and I eventually arrived at the thing I really wanted to know more about: GAN. I was fascinated, that something that looks like a childish debate between two models is able to create brand new images, that is very similar to original images, but different in some ways. My first venture with GANs was with Wasserstein GAN (WGAN) [32], and the goal was to generate pokemons. I used 2 types of neural networks: Dense Neural Network, and Convolutional Neural Network. From the two approach, I preferred the convolutional one, because it seemed to work better from the generated images and evaluation of the model. At the end of the semester I didn't get great results, although I even used image augmentation to improve on it (figure 1.2). The cause was of course shortage on time, because the training process of a GAN is very time-consuming and not much time was left, when I could even start with WGAN, because I had to learn a bunch of things beforehand. But this mini-project was a great first impression and I got to work more with GANs.

After looking at more tasks, models and solutions, I tried to look for a task, that was not done before, giving myself a chance to use everything I was taught in the field. This thought process lead my way into making a new model, that works a bit differently, but can work in real life as well. I always really liked chess-like games, therefore I wanted to make my own, but instead of making it by hand, I really liked the idea, to make a model to do it instead. I really wanted to generate visuals for that would-be-game and just after that is accomplished, make the rules. Quickly came to me, that maybe if I give multiple networks simpler objectives (in a chess-like game's visuals: making the board and making the pieces) and later on I summarize what they learned, my results would be somewhat better, than just trying to make a single model learn everything. I researched this approach, but found

Figure 1.2: My best result from my fifth semester. The model was a WGAN, and the dataset was an augmented one from pokemons with no background.

not much even similar to what my idea was. Throughout my sixth semester, the goal was to make a new model that works at least. Through the hardships of making my multiple dataset to try to code a model, that gets multiple inputs, I eventually got to make the model, that was never-before-seen. As it didn't exist prior me, I gave it a name, so I can call it simpler. That name is Multiple Input Transfer Learning GAN (MITLGAN). The name implies, that more than one input is given, and that I use already trained models, to make it work. My results were again not so great (figure 1.3), but there were some signs, that this model actually can work.

Before I began my thesis, I talked to my advisor and we agreed, that trying out other datasets and trying to use this model to improve on tasks that already have some solutions, is a way to prove this model's right to exist. The most conspicuous quality of a generated image to improve on, is to make clearly visible foreground and background, that are not mixed together.(figure 1.4) Of course there are many generated examples, that at first look, doesn't mix the two of them, but after looking at the image, something wrong almost always can be seen. As the goal of a GAN is in many occasion to make a neural net, that can produce realistic images, this little detail can ruin its results.

1.2 Methodology

In this thesis I am proposing a new model, MITLGAN, to show its potential to solve the problem of foreground and background mixing in an image and all in all improve on the quality of the image. As I didn't had the computing time and power (GPU time and capacity), to fully train my model, this thesis is trying to explain its proof of existence with early training results and the difference to already existing solutions, using three different cases.

This model has a very easy concept: We leave the discriminator the same, as in a usual GAN. I used a fairly small Convolutional Neural Network, which works exactly the same as in a WGAN with Gradient Penalty [10]. The generator consists of two Convolutional

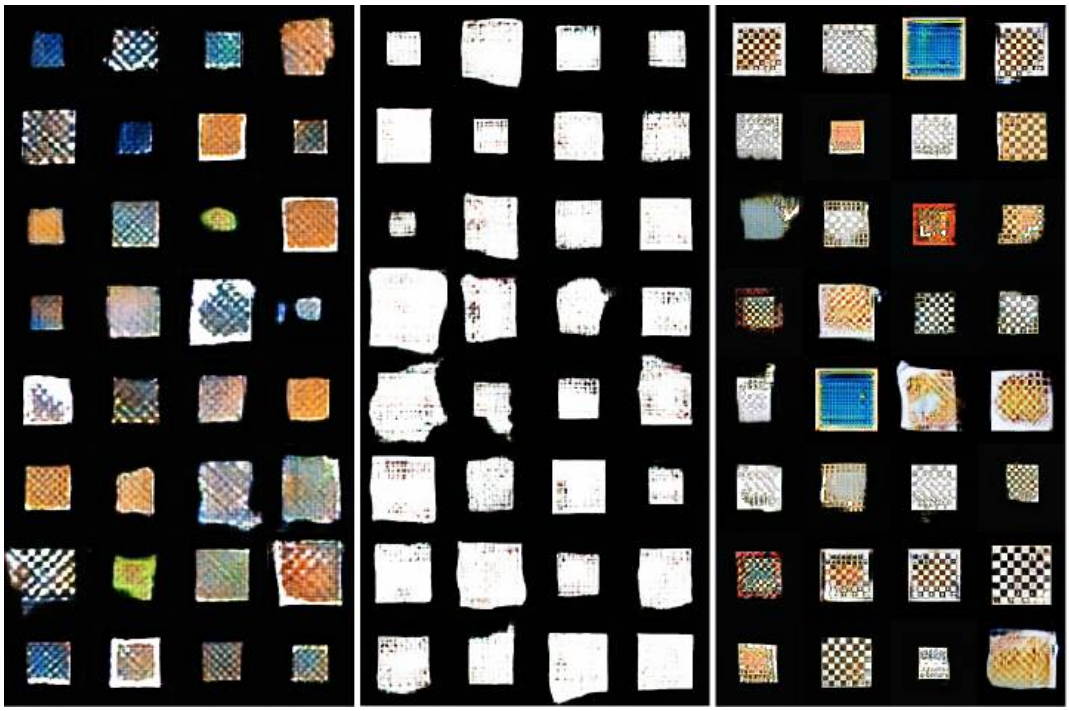

Figure 1.3: My best result from my sixth semester. From the left: the first image contains chess-like boards generated with WGAN, the second pieces generated with WGAN, the third complete starting board generated with MITLGAN.

Figure 1.4: Examples on the issue at hand. In the first image the hair blends in the image, In the second the pullover is blending in. The images were made with StyleGAN2 [17].

Neural Network, that are each pretrained on some data, that has the features we need in the actual result (the foreground and the background). We combine the outputs of these

models, and use a Residual Neural Network [11], to transform the combined data into what we really want (a complete image this time) (figure 1.5).

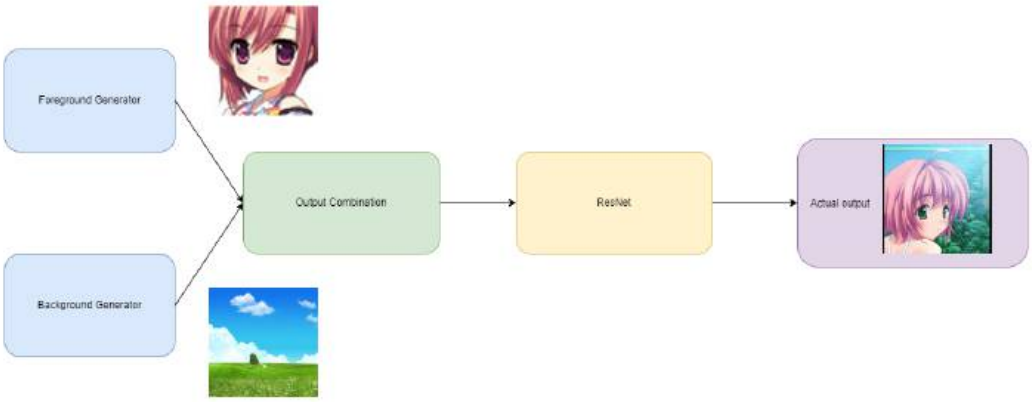

Figure 1.5: The general working of the generator of a MITLGAN.

The experiment can be broken down in the following steps:

1. Train a WGAN with a dataset, that has both of the features (background, foreground).
2. Train a WGAN with a dataset, that only has the features of the background. Keep the generator.
3. Train a different WGAN with a dataset, that only has the features of the foreground. Keep the generator.
4. Train a MITLGAN using the generators from step 2 and 3, with the same dataset as in step 1.
5. See the results of the model from step 1 model and step 4 model.
6. Repeat the steps 1-5 with different datasets, until you feel satisfied.
7. Compare all the results and draw a conclusion.

1.3 Research question

This thesis would like to answer the following question: *"Can MITLGAN be used to improve on the early training process of a GAN and can really combine the already pretrained models' features properly?"*

1.4 Structure

In Chapter 2, after a brief introduction to machine learning, I describe the relevant basics of Deep learning, Convolutional Neural Networks, Neural network layers, Activation functions and Transfer learning, etc... Of course I will only explain the parts, that are relevant to this thesis.

In Chapter 3 I will go deeper into the Generative Adversarial Networks (GAN), explaining the basics. After that I will write about Wasserstein distance and Wasserstein GAN (WGAN).

In Chapter 4 Residual Neural Network (ResNet) will be discussed.

In Chapter 5 I will go deeper into the main point of this thesis, Multiple Input Transfer Learning GAN (MITLGAN).

In Chapter 6 the setup of the experiment will be explained.

In Chapter 7 will be the actual experimentation, the results of the experiment and a discussion section.

The last chapter, chapter 8 will conclude this thesis.

Chapter 2

Deep Learning

In this chapter deep learning will be explained. First machine learning as an introduction to the subject at hand, then the concept of deep learning and the techniques used in the experiment.

2.1 Machine learning

As in chapter 1 it is explained, people had to find an approach to program the tasks, that we humans can learn to do, but can't explain it exactly how. For this purpose came to life machine learning, which is a field of artificial intelligence, that tries to solve this mystery.

Previous programs were wrote in such a way, that if we run the code, the expected result gets produced. As in tasks that machine learning tries to do we don't know how the code should exactly look like, we can't do the same. But there is a catch: we know how the expected result should look like. Therefore machine learning reverses the process.(figure 2.1) A machine learning program learns from the expected results (experience), and improve itself to do a better job (task). It is measured by some method (performance), so we can distinguish, if the system got better or not. [9] This way the program itself is able to create an algorithm, that is up to a standard that is required of it. A machine learning program explores the data, extracts features from it and puts it in the output. This way composes the algorithm using information, that even humans don't know, were there in the data.

Classical machine learning can be done in two main ways: there is supervised and unsupervised learning.

Supervised machine learning means, that the data used is labeled. These labels are about the data itself, therefore are called many times ground truths. When this type of learning is executed, the program uses these labels as well, to learn how the task should be done. Usually these labels contain the right answer to the task at hand. The program tries to extract features from the data, so it can guess, what is the corresponding label to the input. Labeling the data is usually done by people, therefore the program can learn, what a person's solution would be. There are two main tasks, that can be done this way: regression and classification problems.

In case of classification, the set of right solutions are finite and the program tries to create an algorithm, that boils down the data into one of the possibilities. An example: the data are traffic lights, and the labels are, what color were they on the picture. Regression is a bit different, but not much. The labels are real values, therefore there are infinite possible

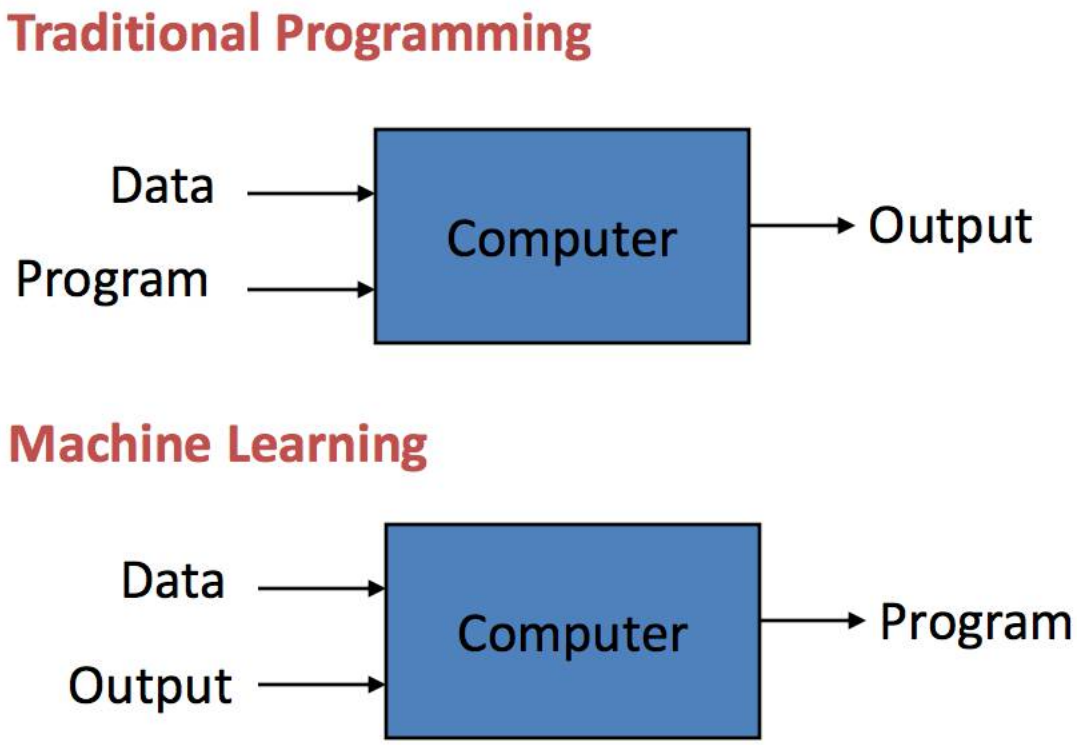

Figure 2.1: The idea of machine learning opposed to traditional programming. Data means the input here.

value. For example: the data are information about toys, and the labels are, how much they cost.

Unfortunately generating data is really hard to do using supervised learning, because of hardships in many cases. When the expected output is a new data, then there isn't a way, to check, if it is exactly correct beforehand. An example would be: we would like to generate a picture of a dog. The generated picture doesn't have a label to see if it really looks like a dog, because it got generated just in that moment. Therefore we can't check the ground truth, because it is not known, what it is exactly. For this type of task we usually use unsupervised learning or use both of them simultaneously.

In contrast to supervised learning, unsupervised learning doesn't have labels for its data. The data only contains initially some sentences or images or some tabular data, but no ground truth. The program tries to extract as many features as possible and tries to recognize some pattern in the data. For example: the input are images and a number of how many groups it should sort them into. The output are the groups that were created. There are many more ways, how an unsupervised learning can be done, but this is the main difference between a supervised and unsupervised learning. The main tasks, that are usually done this way are clustering, dimension reduction and association.

The clustering task means, that the model learns similar and different features between the data, and should group them by those features. Good example is the one, that can be read above by the explanation of unsupervised learning. The dimension reduction task is one, where we generalize the data, and we make conclusions of them. For example: the data are clothes and the output is the best pairing of those. The last mentioned task is association. That is all about trying to find common features of data, and make sets of

them. It is a bit different from clustering, because one data can be in more sets opposed to clustering, where the groups are clearly different. Staying on the cloth line, the example would look like this: the data are clothes, and the output are the pairing of clothes, that are worn most often together.

Many application exist, that use both of them, to explore more possibilities. Sometimes it is called semi-supervised learning, but many times it gets classified in one of the 2 main learning methods.

More about supervised and unsupervised learning can be read in [16] and [20].

2.2 Artificial neural networks

Deep learning is a field inside of machine learning, that uses Artificial neural networks.[9] The idea of deep learning comes from biology, more specifically from how the neurons in the human body works. In the brain there are many neurons connected together, that are signaling each other electrical pulses. The connections are called synapses, and they either block the pulse or let it pass. Artificial Neural Networks try to model this behaviour.

The signals are numbers, that are modified by the neurons, when they pass them. The amplitude of how much the number gets changed are called weights, that are one of the key elements of the model. Through changing these weights can learn the model, how it should work. The synapses' behaviour are modeled by simple functions, called activation functions. When all the inputs are coming into the neuron, the weighted sum of them are calculated and put into an activation function, that either let the value go on or gets it nullified, thus block it.

These neurons are organized into layers. In a layer no neuron gets connected to the other, therefore forming a group, that has 2 purpose: getting the input of the previous layer (or simply getting the input, if it is the first layer) and giving the output to the next layer (or simply give the output of the algorithm, if it is the last layer). The first layer is called the input layer, the last is called the output layer, and the layers in-between are called the hidden layers. The "hidden" word is referencing, that their input and output is not seen from the perspective of an outsider, only the input and the output of the whole model. The "deep" word is referencing, that there are always multiple layers in the model, thus making it "deep". Using more and more layers are useful, to let the model learn more simple features and eventually develop them into more complex ones. More the layers, the training becomes more complex and time consuming, therefore adding more and bigger layers aren't the solution always.

A layer also can have a so-called bias neuron. These neurons are only connected to one layer's input (and not the input layer). They have a value, simply contributing to the weighted sum calculated in their layer's neurons. The value of a bias neuron also gets updated, it serves the purpose of a universal bias based on the whole data.

In our experiment we only talk about feed-forward neural networks. This means, that no layers are giving their output to previous layers, only to a layer, that is coming after it. Therefore the network can be drawn as a Directed Acyclic Graph (DAG). (figure 2.2)

When this type of model learns, we call it training. The training process of a neural network can be easily separated into steps:

1. We feed the network our input data, and get the output.

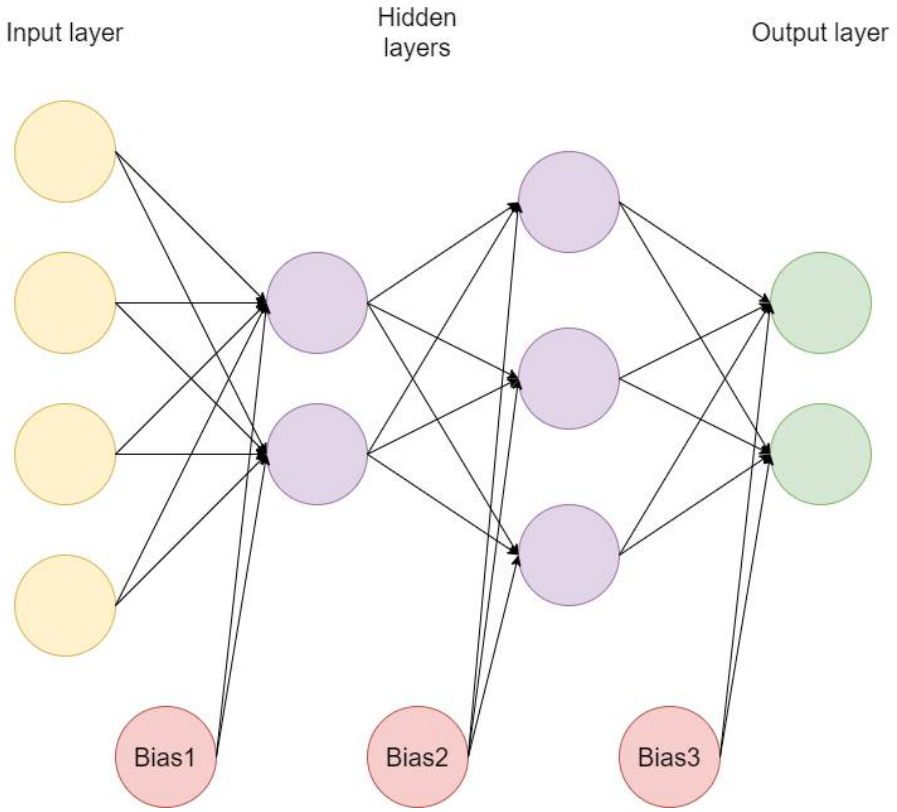

Figure 2.2: A simple example of a neural network. It is a feed-forward neural network, because it is shaped in a DAG.

2. We evaluate the output and calculate the loss value. If the evaluation process' output is stated to be good enough, the training ends.
3. The loss value gets backpropagated through the model. In this process we update the weights, so the network can learn.
4. Repeat from step 1.

In reality the above mentioned steps are repeated an exact number before we check, if the model is good enough and the training can stop. This loop is called an epoch, which is a basic metric for how long we want to train our model. If in fewer epochs than anticipated our outputs are good enough and we stop, we call that early stopping. It really helps in many cases not to ruin our weights accidentally after reaching our minimum goal.

The backpropagation is the key in the training, as it is the only step, where the model can learn. There are many functions, how the weights should be updated. Each and every one of them share one common thing: the weights are updated in terms, how much they were responsible to the result. This can be calculated with a loss (or cost) function. The result of the loss function is the loss, which is used to calculate the "responsibility". That function uses the weights of the neurons and biases, the activation functions, the inputs and the expected outputs. We would like to minimize the loss, which would mean, that our model got better. For this purpose, we use the method called "gradient descent" in most cases. We calculate the gradient vector of the loss function from the point of our actual loss, and we move the weights toward the local minimum. We use a so-called learning

rate, to determine how much we want to change our weights. (figure 2.3) This is used, not to let the neural network "memorize" the dataset. If it wouldn't be part of the optimizer, when new data comes, it would be biased towards the previous data and not its features would be extracted, only the information, if it is exactly the same as a previous one or not. A simple for the backpropagation is:

$$\boldsymbol{W}^{+} = \boldsymbol{W} - \eta \nabla \boldsymbol{C} \tag{2.1}$$

where W^{+} will be the updated vector of weights, W is the old vector of weights, η is the learning rate and ∇C is the gradient of the loss (cost) function. More about calculating gradient descent and backpropagation can be found in [13].

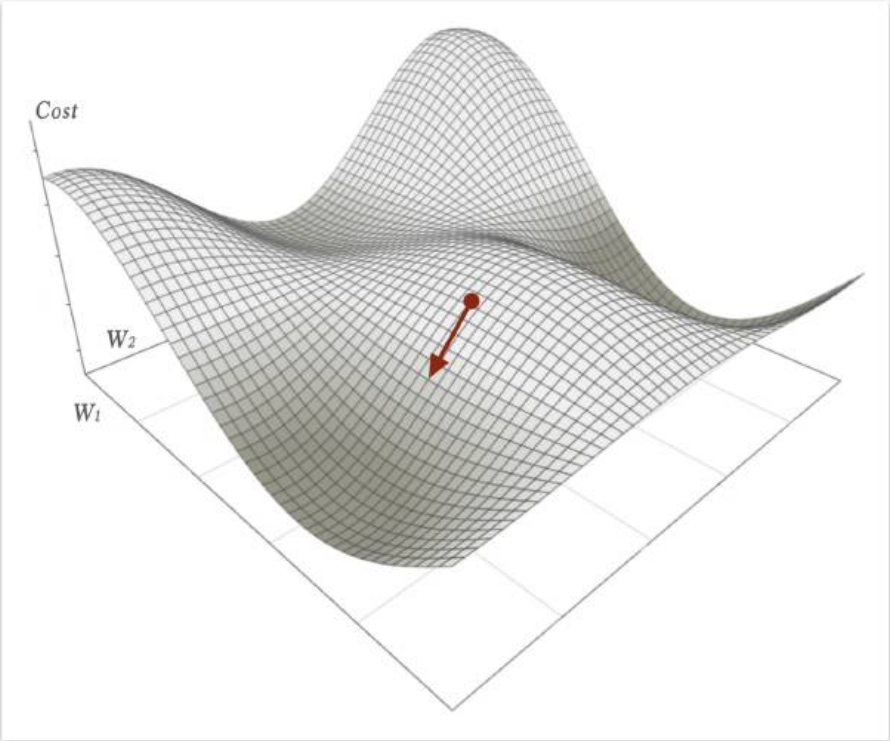

Figure 2.3: A visualization of a simple cost function with only 2 weights. The gradient descent is the red vector. Sorce: [13]

The goal of the training process could be summarized easily: Finding the loss function, that gives the minimum loss for the input, it is trained on. As it can be seen, the loss function uses the expected outcome as well, which is not there in unsupervised learning, because we don't have labels originally. There are two solutions for this problem: Simply we use a different approach to the loss function or the algorithm itself gives label to the data. We will later see, that in this thesis, the latter will be happening.

2.3 Convolutional neural networks

Deep learning is more and more often used these days, to extract features from images. One great example would be object detection. We try to extract from the image, where

and what can be found on an image. Convolutional neural networks [26] are great for this task, and usually on tasks, that include feature extraction from images.

It got its name of the mathematical expression convolution. Convolution is an operation, that combines two functions into one. There are many types of convolution, but a person most often gets to know first the linear convolution, which is noted by the star symbol (*). A good example is the following:

$$x(t) * f(t) = y(t) \tag{2.2}$$

where x(t) is the first function, f(t) is the second function and y(t) is the function we get from the convolution operation.

Images can be seen as functions, therefore convolution can be used using images. Many image processing algorithm takes advantage of it, for example using kernel functions to extract certain features from pictures. Edge detection is one of the tasks, that works like that (figure 2.4), but there are many more uses of it.

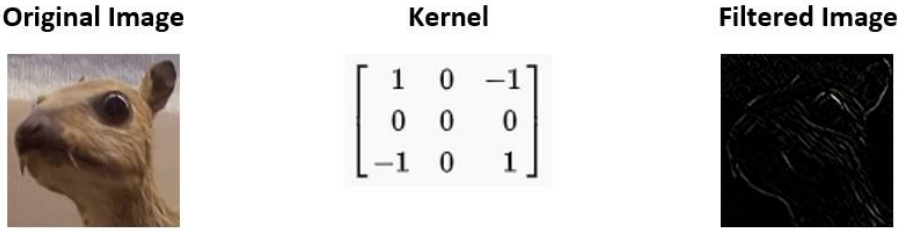

Figure 2.4: A simple convolution using an input image as the first function and a kernel as the second function. The output is a third function, in this case an image. Source: [6]

Convolutional neural networks use so-called convolutional layers, that are using this idea, to try to extract features from images.

2.4 Layers

A deep learning model is using layers as one of the smallest building block. There are many type of them and each one of them are used for various reasons such as extracting one specific type of feature or manipulating the values in a way, that makes the latter layers' job easier. It is possible of making an artificial neural network using only one of the possible layers, but most often combining them in a clever way makes the training more smooth and eventually the results are better in most cases.

2.4.1 Dense layer

This type of layer is one, that is often used in classical artificial neural networks. They work the same way, as it was descripted in the Artificial neural network section. In this layer, the neurons make the weighted sum of their inputs and then apply the specified activation function on them. A dense layer is called a fully connected layer as well, because all of the previous layer's neurons are connected to all of the neurons in the dense layer. This layer

typically exist to extract feature(s) from all of the input at once. The equation looks like this:

$$y_{i'} = \sum_{i} w_{ii'} x_i + b_{i'} \tag{2.3}$$

where y is the output, w is the weight matrix, x is the input and b is the bias neuron's value.[18]

2.4.2 Convolutional layer

A big problem of a model having many fully connected layers, that there are way too many trainable parameter (weights). This slows down the training very much, because more the parameters, more complex the tuning. That is one of the reasons, why bigger models needed a solution, that works with much less parameter, but still trains well. The convolutional layer is one of the products of those thoughts, that can build up a convolutional neural network.

A convolutional layer works a little bit differently, than a classic artificial neural network layer. The weights applied here are in a kernel (filter) matrix. One neuron in the layer has as many inputs, as the size of the kernel is, not as many neurons the previous layer has. All of the neurons in this type of layer share the same kernel, using it to apply the weights to each of their inputs. This way we can extract the same feature(s) from various parts of the input. The size of the kernel impacts the size of the next layer, as it tells, how many inputs, will be converted into one output.

In case of an image input, the convolutional layer works like using a convolution in classical image processing. There is a virtual window in the size of the kernel matrix, that represents the inputs of one neuron. The sum of the window's and the kernel's element-wise product will be the weighted sum, on which the activation function gets applied. For the next neuron's input, we slide the window by a certain amount, and repeat the step. The layer's output can be arranged in a way, that can represent an image, the result of the convolutional step. (figure 2.5)

Using multiple convolutional layer in our model, we can get more and more complex features from the input.

There are three more key hyperparameters, other than the kernel: filter number, stride and padding. The higher the filter number, the more feature maps we would like to extract. The stride tells us, how much we have to slide the window, before getting the input neurons for the next neuron in the layer. Setting this parameter, we can control, how much overlapping we would like between the inputs of our neurons. The last one is padding. That works exactly like padding an image by pixels, but in our case the feature map extracted gets that padding. The bigger the number, the more padding the map gets.

2.4.3 Upsampling layer

This layer simply gets the input tensor, and pads all of its elements by the given amount. It is a key element of a network, that wants to extract the features from a tensor, transforming that into a bigger output than the original input was. When we generate images from a noise vector, this layer is often used.

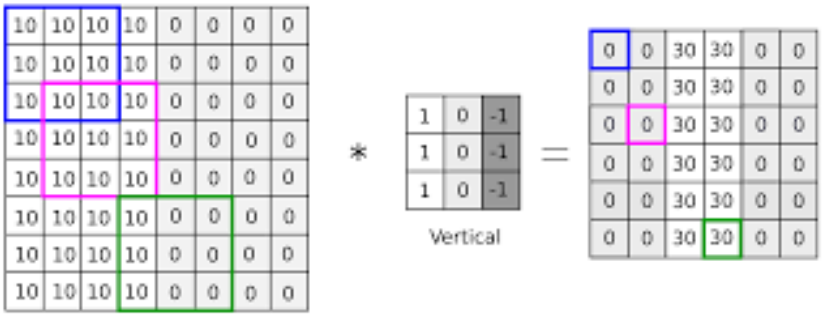

Figure 2.5: On the left we see all the values from the previous layer, arranged as an image would be. On the right we see our convolutional layer's values, before applying the activation function. In the middle is our kernel, which looks exactly like a Prewitt filter, but in our case that is only a coincidence. The values in the kernel are the weights. The same coloured windows are representing, where are connections between the layers. Source: [22]

2.4.4 Transposed convolutional layer

As the name tells us, the transposed convolutional layer works similar to a simple convolutional layer. Visualizing the input as a matrix, each of the elements gets padded first. This way we get a bigger matrix than before. After this padding, we apply the convolutional step as we have seen before. This way the output of this layer can be bigger, than the input was. (figure 2.6) This is a great way, to construct an image from a vector input, as using more transposed convolutional layers in a row, it can reconstruct the coded features from the input into a tensor, that can be converted into an actual image.

This layer actually can be implemented using an upsampling layer and a convolutional layer in a row.

2.4.5 Normalization layers

Normalization layer is trying to standardize the input, so the next layer can learn more efficiently. Different types of normalization layers help in different ways, but one thing is common: how the math works behind them.

$$\hat{x} = \frac{x - \mu}{\sigma} \tag{2.4}$$

First all of them are calculating a mean (μ) and a variance (σ) value. Then the mean is subtracted from each input that is considered in the normalization and divided by the variance. After this step, the input's distribution will be standard normal distribution. Of course that distribution is not always good for us, it depends on the input and what we want to achieve. Therefore there is two learnable parameters in this layers: the scaling and the shifting of the distribution. This way we still have a normal distribution, but it can take into account the actual input incoming.

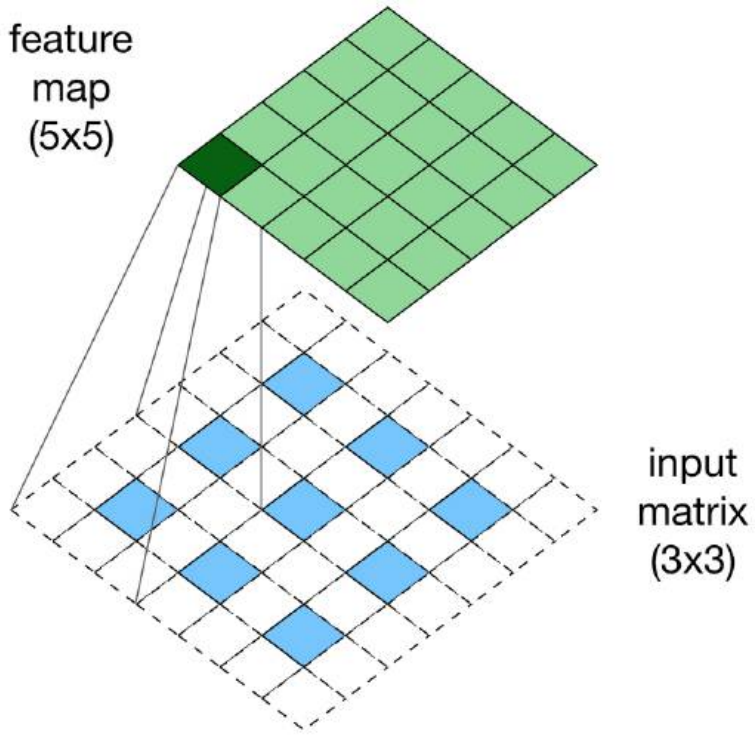

Figure 2.6: A transposed convolutional layer pads each value of the input matrix (blue) as shown in this figure. Doing so it is possible to increase the size of the resulting feature map (green) compared to the input matrix. Source: [30]

There are four dimensions of our tensors, when we use them in our neural network: the dimensions of the input (height, width, channels) and the batch size. The difference between the normalization techniques are, on which axis(es) the mean and the variance are calculated.

In this experiment we use two types of normalization layers: instance and batch normalization. (figure 2.7)

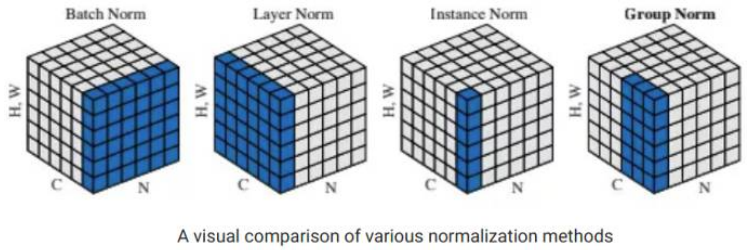

Figure 2.7: A visual representation of normalization techniques. It shows, on which axises gets the mean and variance calculated. Source: [33]

In case of instance normalization, we calculate the statistics on 2 axis: the height and the width. So in our case, all of the features maps are getting normalized one by one. If we imagine an RGB image, the features' number is 3, therefore the statistics are calculated on the red, green and blue part of the image individually. This type of normalization is

helpful to eliminate the contrasts of an image, therefore the next layer can work more independently.

Batch normalization calculates the mean and the variance on 3 axis: height, width and the batch size. For example on a batch of 32 RGB images, it would calculate 3 mean and variance for each colour channels. It is helpful for eliminating the too big of differences in the batch's elements, so the training becomes more smooth.

There is one big difference in usage of the above mentioned normalization techniques: when the contrast matters, batch normalization is a much better option, because it can learn, how much contrast there is between the individual features map in a channel. In the case of instance normalization, the contrast gets minimized, therefore that feature won't be useful in the latter layers. Of course that is not a problem in some cases. There have been some experiments, where researchers tried to switch the two normalization techniques, as they are very similar. More can be read in [2] and [33]

2.4.6 Dropout layer

This layer can disable some percent of the connections randomly between the previous layer and the next layer. It isn't actually a layer, just a method to let the next layer's neuron learn a bit independently of each other, preventing the complex co-adaptations to the input, therefore preventing overfitting, which will be in a section later on.

This type of layer is only active, when we train the network, as its goal is a smoother training process. When not in training, this layer is inactive, and none of the connections get disabled in the model.

2.4.7 Layers, that are only shaping their input

In this section I will write about layers, that are only shaping or reshaping the data and (usually) not applying any weights or activation function. Upsampling layer, that I already mentioned can be seen as one of them, although the layers mentioned here doesn't change the amount of the input data. The three layers mentioned here are: reshaping layer, flatten layer and concatenation layer.

Reshape layer does exactly what its name refers to: it transforms the inputs shape into the given form. It is very important, that the given shape is one, that can fit the same amount of values as the input. It is often used before and after convolutional layers, because there the shape of the data really matters.

Flatten layer is a special reshape layer, only thing it does, that it reduces the inputs in the batch into only one dimension.

Concatenation layer gets multiple inputs, and concatenates them along an axis. It only gives one output tensor, therefore a great way to join networks or part of a network together.

2.5 Activation functions

One of the key element in a deep learning model, what activation functions we are using. These are useful to manipulate further the loss (cost) function. In this experiment I used

3 types of them: rectified linear unit (ReLU), leaky rectified linear unit (leaky ReLU), and hyperbolic tangent (tanh) activation functions.

2.5.1 Rectified linear unit (ReLU)

This is a really simple activation function, that is giving 0, if its input is less or equal than 0, otherwise gives its input as output. It simply eliminates the values, that are not necessary in the extracted feature.(figure 2.8) There are really complex mathematical reasons, why it works, but that won't be discussed in this thesis. More of it can be read in [1], as this is the paper that introduced this.

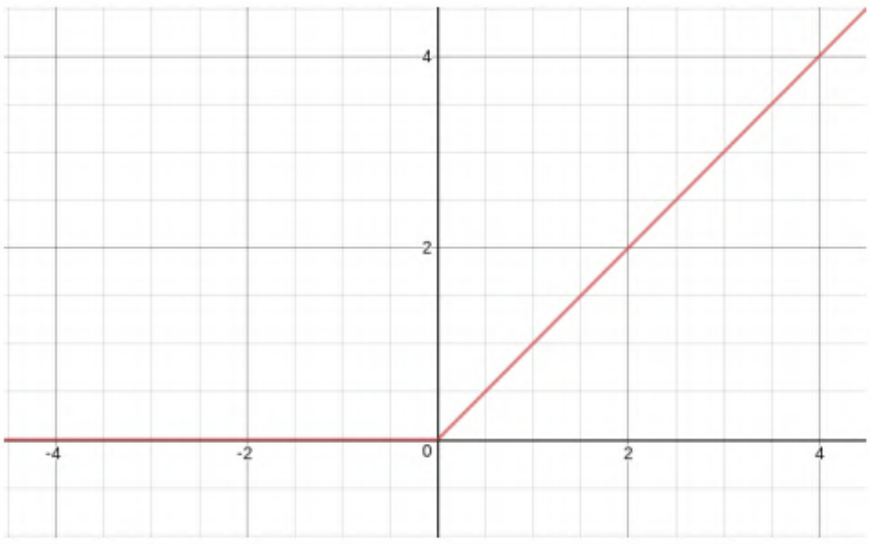

Figure 2.8: The Rectified Linear Unit (ReLU) activation function produces 0 as an output when $x < 0$, and then produces a linear with slope of 1 when $x > 0$. Source: [1]

2.5.2 Leaky rectified linear unit (leaky ReLU)

This function is really similar to ReLU, and in case of positive inputs, it produces the same output. But in case of negative inputs, it doesn't simply nullify them. Instead the functions first derivative in the negative half is changed from 0 to a very minimal value (for example 0.01), letting the negative values through, although only with minimized inpact. (figure 2.9) It is useful for cases, where we want to eliminate the problem of a dead neuron. A dead neuron is one, that really doesn't learn any part of a feature, often because its inputs are nullified in it. More of it can be read in [4].

2.5.3 Hyperbolic tangent (tanh)

This activation function is really different from the above mentioned ones. This activation function is zero centered, and transforms its output into the range of (-1, 1). (figure 2.10) In this experiment it is favored as the last layer's activation function in a neural network, because it doesn't nullify its input, therefore not leaving out any part of the networks work by accident.

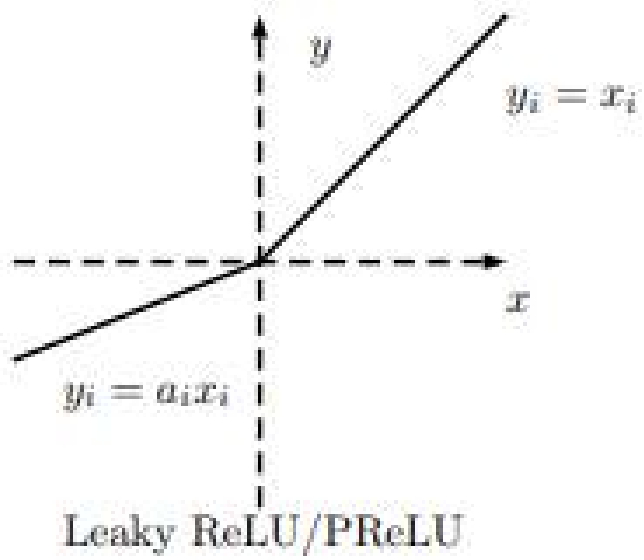

Figure 2.9: The diagram of a leaky ReLU. The mentioned name PReLU references another type of rectified activation functions, that learns the first derivative for the negative half. PReLU is not used in our thesis, therefore it is not further discussed. Source: [4]

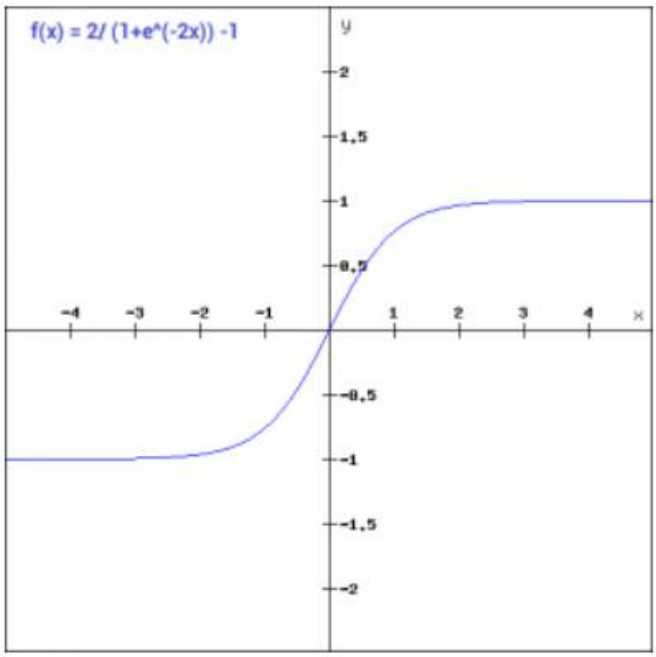

Figure 2.10: The diagram of a tanh function. Source: [24]

2.6 Adaptive moment estimation (Adam) optimizer

Adaptive moment estimation is an algorithm for doing the backpropagation in deep learning. Besides the basic learning rate ν, there are two more learning rates, that are designed to be adaptive to the trainable parameters of the model. [23] The optimizer stores an exponentially decaying average of squared past gradients, as well, as simply an exponentially decaying average of past gradients. The equations look like:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1)\, g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2)\, g_t^2 \end{aligned} \tag{2.5}$$

where m is the exponentially decaying average of past gradients, v is the exponentially decaying squared average of past gradients, β values are the decaying rates and g is the gradient of the last loss function from the point of the loss value.

This helps the optimizer, to get closer to the local minimum of the loss function in respect to past attempts. As researchers found out, that these values are biased towards 0, they corrected them using the β values:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \end{aligned} \tag{2.6}$$

where $\hat{m}$ and $\hat{v}$ are the corrected values.

Using these values the optimization step looks like this:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{2.7}$$

where θ values are the weights, and ν is the learning rate, and ϵ is a relatively small positive number to avoid division by zero.

2.7 Overfitting

Overfitting is a problem, that can occur during training. It refers to the case, when the model almost or even entirely memorizes the training dataset, instead of just extracting its features. An example for overfitting can be seen on figure 2.11.

We used in this experiment many techniques and methods to avoid this. One of them was already mentioned, which is using dropout in our model. Later on we will see more of that.

2.8 Transfer learning

Sometimes we don't want to begin our training from zero, making a new model. In many cases we use an already trained neural network, and expand its knowledge. It is a common method, when a pretrained neural network already learned features, that are useful for our goal. For example: we would like to recognize cats and dogs on our images. We use a

Figure 2.11: A case of overfitting, detected using a testing dataset for continuous evaluation of our training. It is easily observable, that after epoch 7, the testing error becomes significantly bigger, while the training error becomes almost 0. Source: [25]

neural network as a base, that already can recognize dogs. Than we use transfer learning, to teach our expanded network to recognize cats as well.

This method can be broken down into steps:

1. Get a dataset, that includes dogs and cats as well.
2. Load in a neural network, that already recognizes dogs. Be careful, in many cases loading in the output layer can be a bad move, because then the output of the loaded network is a solution for the original task, not extracted features, that can be helpful.
3. Freeze the trainable parameters of the network, making them unable to change. It is a must have step, as we don't want to lose the already extracted features, we want just to expand them.
4. Add some layers to the output of the loaded model, thus getting a new model.
5. Train the new model on the dataset acquired in the first step.

In short, this technique lets the model learn new features based on the features already known. More about this in [27].

Chapter 3

Generative Adversarial Networks (GAN)

Generative Adversarial Networks (from now on: GAN) is an unsupervised deep learning technique for generating new data and discriminating between generated and original data or as in a real life application, fake and real data. The former usage of this technology is the one, we try to applicate most of the time, this is our case now as well. A GAN has (usually) two main parts: a generator and a discriminator. These two parts are playing a minimax game between them. The generator tries to fool the discriminator and the discriminator tries to discriminate between the generated data and the original data. The two parts are trained simultaneously and if our metrics say that the training ended, we usually discard one of them (in our case the discriminator).

3.1 Generator

This is one of the parts of a classic GAN. It is a neural network, that is responsible for producing new data. There are multiple ways to create a model, that does this job.

First of all, making this model is greatly dependent on what our input is. Usually there are 3 types of input: a data, that only needs a transformation (for example: image-to-image translation task); a tensor of data, that contains labeled and encoded some features, that we would like to see in the output (great example is a face generation task, where we construct the input, to tell the generator the hair color and facial expression etc.); or simply a noise vector/tensor, therefore we don't know exactly which part of the input codes which feature. In the last case, the model is ultimately tries to find the patterns in it. In our experiment, the last type will be used as input (figure 3.1), although the first type will make an appearance as well, to better understand the whole process.

All of our inputs are part of a latent space. This latent space is best described as a multi-dimensional space (sometimes even 100-dimensional) where similar inputs are represented as points, relatively close to each other. (figure 3.2) It is latent, because we don't know exactly, how this space looks like (visualizing it is even harder, the higher the dimensionality is). The generator tries to explore this latent space and eventually learns, how to convert a point in this space into the output.

There are many types of model, that can be used as a generator. As the input varies, the used techniques are varying too. I won't discuss the labeled and encoded inputs, as in our experiment it doesn't show up.

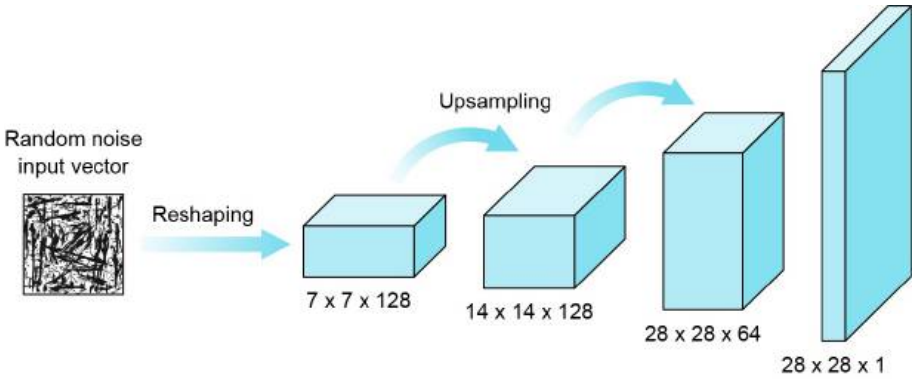

Figure 3.1: An example of a generator, that uses a random noise vector as input, generating a 28X28 gray-scale picture. Source: [5]

A general approach in the case of a noise vector input, to decode the vector into a tensor. Usually after the decoding part, the output will be bigger, than the input was as the decoded data is usually stores the features compressed. In the model the layers can be usually separated into blocks, that has the same layers, but with different parameters. Between the blocks there can be found sometimes other layers that tries to prepare the values for the next block, making the training more smooth (for example normalization layers). Every block tries to extract some features of their input, shaping it into a bit at a time. Building these blocks after each other, slowly we progress feature- and shape-wise closer to the output's shape and eventually getting it. That's where usually the output layer comes. During the training, the latter the block comes in the model, the more complex feature it tries to extract, using the simpler one, got in the former block. This is a very common way of building a generator, it is used in many applications of GANs for example: DCGAN [21], AGE (Adversarial Generator-Encoder Networks) [29], that uses a GAN's generator as a decoder.

The other type of input follows generally a bit different way. As the input is not encoded we can't just simply decode it usually. Two general approaches are there: change the data bit at a time, slowly transforming it into the expected output or encode the features that are useful and then decode it into the expected output.

The first approach is usually implemented by a ResNet, which will get its own chapter later on in this thesis.

The latter can be implemented with a Unet [19]. It got its name from the shape, how it is usually represented. It is many times implemented symmetrical, therefore the encoding and decoding part of it are reversed of each other. In the encoding part, the data gets smaller, but deeper (more feature channels), then in the decoding part, the data gets back the shape, that the input originally had (same shape as the expected output's). This type of network can be used to change the style of the data (usually images) or translate the data into another data (for example image-to-image translation). This approach is used outside of GANs as well. Autoencoders [34] and Variational Autoencoders [31] (that are typically very similar to GANs) use this approach, to try to learn a dataset's features. These models are often used only to understand a dataset's features, not to generate new data.

In our experiment the simple decoding approach, and the ResNet's approach will be very important, as both of them are used in one way or another.

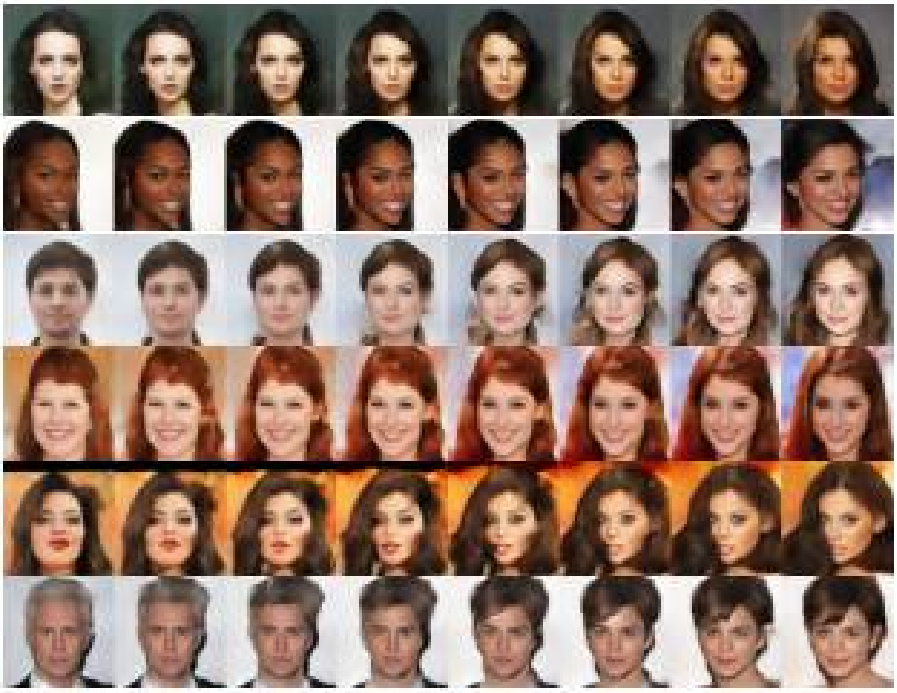

Figure 3.2: Faces in the latent space. From the initial image in a row, we travel in the direction of a vector, generating further images at some points. Source: [3]

3.2 Discriminator

The other main part of a GAN model is the discriminator. It is sometimes called critic as well, because of the objective of this network, which is to judge its input and decide, if it is a real or fake (generated) data. Many GAN implementations use it only, to help train the generator, but there are cases, where this network is kept.

Here are no variances in the shape of the input, because it is defined by the dataset. The network's building is usually reversed to the generator, but not necessary entirely. The general approach is, to decode the data, mapping its features and deciding, if they are fit to be real. This model is also build up similarly block-wise as the generator to achieve its goal. In this case the block try to extract the features as well, but it tries to compress these features, deepening the data's shape in the process. In the end, the data gets transformed into a simple answer: whether the data is real or not. (figure 3.3)

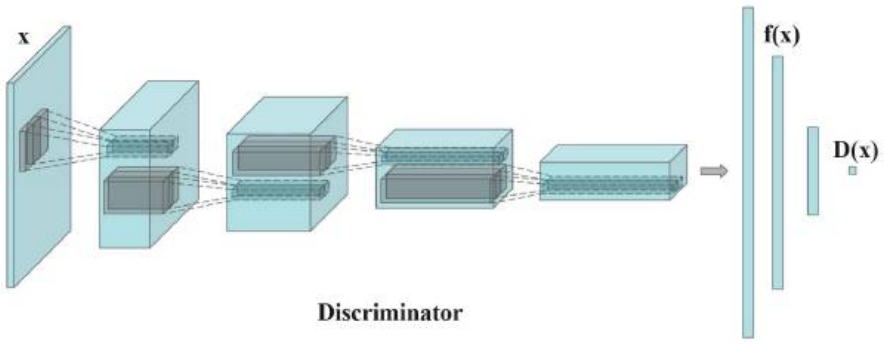

Figure 3.3: An example of a discriminator Source: [15]

The networks vary in the way, how their output is representing. The early implementations' output represented the answer in a binary way: The output varied in the interval of [0, 1]. Closer to 0 meant, that the input is fake. If it was closer to 1, then it was real. Of course, which number represents which answer is implementation dependent. This representation is simple, but it has its negative effects. The main problem is, when the network is trained almost perfectly in terms of loss (every real data produces close to the output 1, and every fake data produces close to the output 0). This is called vanishing gradient and it is very problematic. If the discriminators behaviour is good (the extracted features are good in terms of reality of the data), then the loss will be close to 0 and the training process becomes very slow, or no training happens at all. That is not good, if the generator is not fully trained yet. If the discriminator's behaviour is bad, then the generator gets bad feedback during the training from it (training will be explained in a later section), therefore it doesn't learn the appropriate features, resulting to bad generated data. [32] Although its negative effects, if provided a good setup and an appropriate dataset, the bad training can be avoided.

When this type of discriminator output is used, the loss function uses typically the original method, the JS (Jensen-Shannon) Divergence, which is based on the KL (Kullback-Leibler) Divergence.

The KL Divergence:

$$D_{KL}(p\|q) = \int_x p(x) \log \frac{p(x)}{q(x)} dx \tag{3.1}$$

The JS Divergence:

$$D_{JS}(p\|q) = \frac{1}{2} D_{KL}\left(p\|\frac{p+q}{2}\right) + \frac{1}{2} D_{KL}\left(q\|\frac{p+q}{2}\right) \tag{3.2}$$

Both metrics measures how one probability distribution p diverges from a second expected probability distribution q. The JS Divergence is more useful, then the KL Divergence, because JS Divergence is symmetrical and more smooth then the other one. (figure 3.4)

The two probability distribution in our case are the distribution of the discriminator's outputs, in respect to whether the input was fake or real. The more the two overlaps each other, the better the generator is, the more they are separated, the better the discriminator is.

The other approach to the output of the discriminator is one, that tries to eliminate this problem. In this case, the output is a number, that can be any real value. It doesn't actually classify the input, but tries to expand the difference between the outputs by the real and fake data. Usually the higher the number, the network considers the input more realistic, the lower the number, the more unrealistic it is. As the distance can be further and further expanded, the probability of vanishing gradient becomes more and more less. This approach is used with Wasserstein GANs [32], which will be discussed further later on.

3.3 Training process

As there are two networks, the training process needs to be addressed. The discriminator's training is the following:

1. Generate data using the generator.

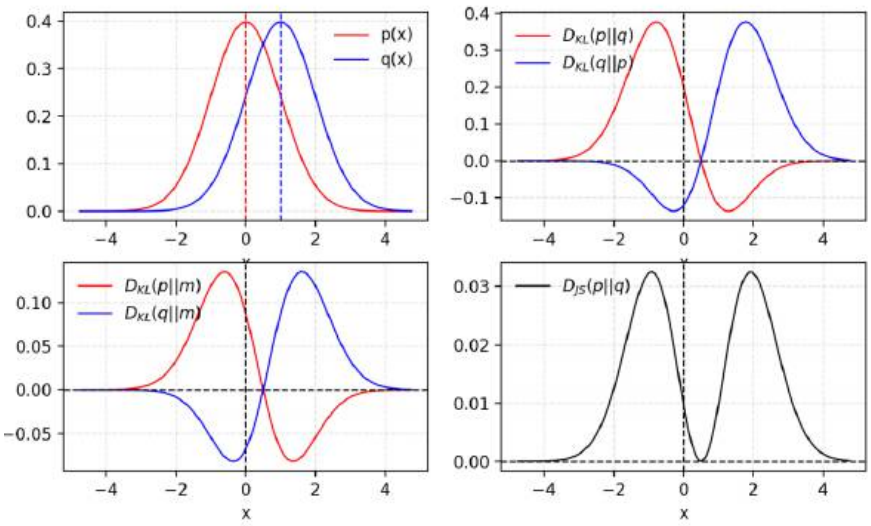

Figure 3.4: Given two Gaussian distribution, p with mean=0 and std=1 and q with mean=1 and std=1. The average of two distributions is labeled as $m = (p + q)/2$. KL divergence D_{KL} is asymmetric but JS divergence D_{JS} is symmetric. Source: [32]

2. Train the discriminator using the real dataset and the generated data. Here the loss function only contains the losses in respect for the discriminator. The generator's loss is not calculated, as if only the discriminator would be present.

The generator's training looks a bit more complicated:

1. Generate data using the generator.
2. Get the discriminators output using the real and fake data.
3. Calculate the loss in respect to the discriminator and generator as well.
4. Use the gradient only to change the generator's weights.

The generator's training is a bit more complex, because it doesn't produce an output, that can be useful for training. Therefore the discriminator helps, providing the necessary feedback. The loss function is created using both of the networks' weights, because the discriminator's output can't be judged without knowing, how good the discriminator is actually. If the discriminator is really bad, the generator shouldn't be changing its weights drastically, only a little bit. But if the discriminator is really good, than the loss is coming mostly from the generator, therefore its weights can be adjusted a bit more drastically.

3.4 Wasserstein GAN (WGAN)

Wasserstein GAN is a type of GANs [32]. With this a new method came into the public consciousness: Wasserstein distance. A WGAN will be a WGAN, because it uses that metric when calculating losses. It is useful for stabilizing the training process.

Wasserstein distance is called Earth Mover's distance as well, short for EM distance, because it can be interpreted as the minimum energy cost of moving and transforming a pile of dirt in the shape of one probability distribution to the shape of the other distribution. The cost is quantified by: the dirt moved multiplied by how far we moved the dirt. (figure 3.5)

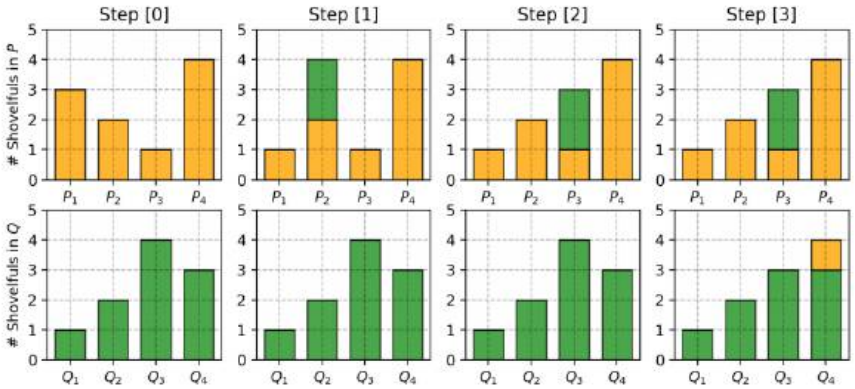

Figure 3.5: A discrete visualization of Wasserstein distance. Source: [32]

The next formula shows, how to mathematically interpret the above:

$$W\left(p_r, p_g\right) = \inf_{\gamma \sim \Pi(p_r, p_g)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|] \tag{3.3}$$

In the formula, $\Pi\left(p_r, p_g\right)$ is the set of all possible joint probability distributions between p_r and p_g, where p_r is the data distribution over a real sample and p_g is the generator's distribution over a data.

Wasserstein Distance is better than JS and KL Divergence, because it doesn't have sudden jumps, and differentiable everywhere, even when the two distributions are fully overlapped. It is very helpful, to stabilize the training process using the gradient descents.

If we want to use the Wasserstein Distance as our loss function, then there is a transformed version of it:

$$W\left(p_r, p_g\right) = \frac{1}{K} \sup_{\|f\|_L \leq K} \mathbb{E}_{x\sim p_r}[f(x)] - \mathbb{E}_{x\sim p_g}[f(x)] \tag{3.4}$$

In the transformed version there is a demand to satisfy. The function f in the new form of Wasserstein metric is demanded to satisfy $\|f\|_L \leq K$, meaning it should be *K-Lipschitz continuous*. In [32], the proposed method was weight clipping, which means that after the gradient descent the weights that are out of a certain interval are clipped, so they will be inside the window again. Later the gradient penalty method [10] was introduced. It basically modifies the gradient in respect to the inputs given in the training, so it becomes 1-Lipschitz continuous, therefore K-Lipschitz continuous. It is producing better results, than weight clipping, therefore I used that.

As it can be easily deducted, Wasserstein distance can be calculated between any two distribution, that have the same amount of samples. It is very useful because in our case the output of the discriminator will only tell us how much the data seems to be realistic on an infinite interval, as it was mentioned in the Discriminator section. More about Wasserstein distance and WGAN in [32] and [10].

3.5 Frechet Inception Distance (FID)

Frechet Inception Distance (FID) is a metric for evaluating a GAN model introduced in the paper [12]. This value tells us, how recognizable the fake images are compared to the real images, using a neural network made for recognizing images.

The calculation begins with acquiring a model, that is pretrained on many images for recognizing objects / pictures. This model usually is Inception V3 or some really similar model, because that is a state-of-the-art solution. We can import it easily through Tensorflow Keras's python packages. It is really important, that we don't load in the head of the network, because we don't want to get an answer, what is on the image, only an embedding of it. The embedding tensor tells us, how precisely can we categorize the image. If in the embedding only has few high value, it is quite recognizable probably, if it has more or less the same values everywhere, the image is not so recognizable. How the embedding looks like really depends on the data, the model was taught on.

After that we only have to generate some images and have equal amount of real data, and we put those values in the recognizing network, getting the embedding. Those embeddings make two probability distribution, just like in the case of Wasserstein distance: one for the real images and one for the fake ones. The Frechet distance, which we have to calculate between the probability distributions, is also called sometimes Wasserstein-2, as both of them work really similar. We calculate the following equation which will give us the FID score:

$$d^2 = ||\mu_1 - \mu_2||_2^2 + Tr(C_1 + C_2 - 2\sqrt{C_1 C_2}) \tag{3.5}$$

, where d^2 is the Frechet distance, the μ values are the means of the 2 distributions, the C values are the covariance matrices of the 2 distributions. This way we get, how close are the real and fake images in terms of recognizability.

Chapter 4

Residual network (ResNet)

Residual network is another type of artificial neural networks. [11] It is based upon a single idea: making really deep neural networks without too much problems during the training. These problems include the vanishing gradient problem, which was discussed in the previous chapter and degradation. The latter means that adding more layers to a neural network after a certain point makes the training and test results much worse. Sometimes we face tasks, that would require many layers, but this problem occurs. Utilizing skip connections and normalization makes it possible to make as deep network, as a plain feed-forward network, but without the aforementioned problems. When a skip connection is coming in a residual network, we call that step a residual learning. Basically we stack some convolutional or dense layers with some type of normalization and then we use residual learning, to stabilize the training.

In a simple feed-forward network we try to transform the input into some desirable output for example some extracted feature. Lets call it $H(x)$, where x is the input. Instead of this, in residual learning we don't want to learn the desired output at once, but the residual to transform the input x into the desired output. Lets call it $R(x)$. $R(x)$ is easily calculated:

$$R(x) = H(x) - x \tag{4.1}$$

, as it is the difference of the input and the output. Then transformed into:

$$H(x) = R(x) + x \tag{4.2}$$

which shows us, that it is enough to learn the residual and then just simply add the input to it. This way the experiments showed, that the above mentioned problems are much less to occur.

This type of learning is really good for a specific task, that inspired this thesis: image-to-image translation. As this network learns by adding residual after residual to the identity (input), it can learn to add the desired style's features as well. Residual blocks were used in CycleGAN [35], where it proved, that ResNet can be used in a generator of a GAN.

4.1 Skip connection

Skip connections are really simple. It means, that the output of a layer gets connected not just to the next layer, but to a layer that comes afterward as well. It is called skip, because

the added extra connection skips the next layer (or some additional layers as well). As this connection usually connects to a layer, that already have an other input, the inputs get added to each other in a way, to simplify themselves into one input, so they will be compatible to the layer they are connected to.

4.2 Residual blocks

A residual network consists of residual blocks. The concrete implementation of the block varies based on what we want it to do, but the basics are the same. We use some convolutional (or dense) layers and normalization techniques to learn the residual for the input of the block. After that we use a skip connection to add the identity (the input of the block) element-wise to the residual, resulting in the output of the block. (figure 4.1) It is very important to use convolutional (or dense) layers in a way, that it doesn't change the dimensionality of the input by the end of the block. If that occurs by any chance, there are solutions to the problem (usually simply reshaping it or adding it in a way, that results in a good output), but it is very questionable, what will the actual model learn then.

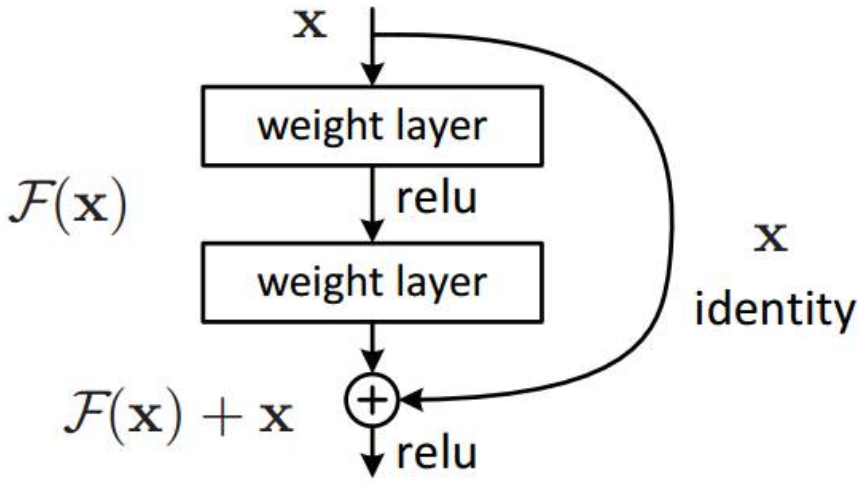

Figure 4.1: A simple example of a residual block. Here the residual is denoted as $F(x)$. Source: [35]

Putting these blocks in sequence, the model can learn slowly the different features' residuals, so we get the desired output. In the early blocks, just like in convolutional neural networks, it learns only simple features / residuals, but in later blocks the learned things get more and more complex.

4.3 Densely connected CNNs (DenseNets)

This is an idea introduced in [14]. It basically uses skip connections differently than the original ResNet, resulting in keeping the simpler / earlier features, and combining them later. It proposes the following advantages: alleviating the vanishing gradient problem, strengthening feature propagation, encouraging feature reuse and substantially reducing the number of parameters.

The vanishing gradient problem occurs with a smaller chance, because the layers have direct connection to the gradient and the original input, therefore the flow of information in the model gets to be much better. The feature propagation and feature reuse is also thanks to the multiple connections to all the layers, because the latter layers have access to all the learned features all the time, to try to find another, more complex feature. The parameters gets reduced, because it uses only limited amount of filters / kernels, in each convolutional layer.

The actual DenseNet is build up by dense blocks, that are looking just like I explained. Putting them after each other separated by some convolutional and pooling layers, proved to be a really good network. (figure 4.2) The idea inspired me, to use something like this in my new model, as keeping many features seemed really advantageous.

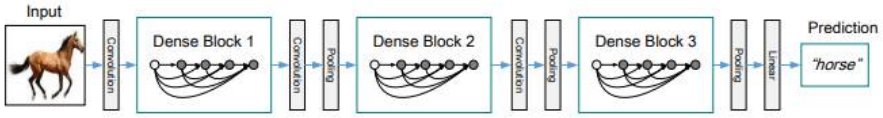

Figure 4.2: The example of a DenseNet. Source: [14]

Chapter 5

Multiple Input Transfer Learning Generative Adversarial Network (MITLGAN)

This is the new model, that this thesis proposes. The goal of this model is to make a generator, that uses pretrained generators' learned features, and combines them. In this experiment we focused on joining the foreground features and background features into a complete image. As its name tells, the generator has multiple inputs, which are appropriate for the pretrained models. After those models extracted the features of their respectful part, the model combines those outputs into one. After that, we use image-to-image translation, to learn to put together the features into the desired image. This model is in early experiments, therefore the proposed solution is a beginning, that if proves to be good enough, can be perfected.

5.1 Connecting two generators

In this section I will write about the considered solutions for combining the two pretrained models' outputs.

The goal is to combine the two tensors without loosing information in the process. Both of the previously mentioned generators' outputs have to have the same shape, to be possible to combine them regularly. If it isn't the case, there should be a solution, but the learned features will lose information or the learned features could be completely useless, as they learned with a different shape in mind.

One of the solutions could be a simple element-wise addition. First I thought, that this is the simple, yet effective way. After further examination of adding two tensors together, I thought of the residual blocks' case. There it is a good solution, because one of the two tensors has only a residual, which is in simpler terms has only minimal values, therefore after the addition the output has its values in more or less the same value range. It means, that the weights can be updated in a similar scale. Adding two similarly large values can result into a training problem, because the model has to later on learn again, to have an output that actually can be an image. A simple normalization could help, but to learning its parameters seems a bit slow of a process, therefore the network later on would have to learn a bit different after the normalization parameters get more or less good. Although

this way could have worked, ultimately I didn't chose it, because a better idea got into my head.

The better idea was this: What if I can have some parameters that can learn a way to combine the two tensors? First I put the features next to each other in a single tensor, that resulted in 6 features. After that, I used a convolutional layer, that only has 3 kernels and every kernel has only 1 value. Therefore the kernels would work in a way, that allows to combine the 6 features into 3, while the height and width of the tensor would be the same. The activation function used here is a ReLU, to let the model drop the values, that are not useful in the given position. As it can be seen, it is ultimately an addition as well, but instead of a simple element-wise one, it actually can learn, how to add them together. (figure 5.1) Later I found out, that there are even implementation of ResNet-like networks ([14]), that use a similar method in the residual step, therefore I was convinced, that my idea can really work.

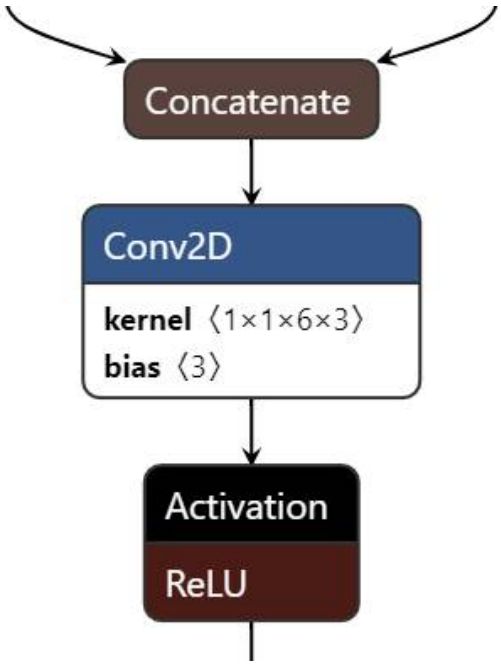

Figure 5.1: Connecting two networks, in a simple way, that can learn, how to add the two inputs together.

5.2 Transfer learning used differently

Transfer learning is used, to expand the capabilities of a model, as it was discussed earlier. This means, that the network learns additional features, therefore it can be used a bit different. For example the different usage could be a pet recognizer, instead of a dog recognizer network. This idea is proven to be useful, therefore many experiments use it, to speed up the process of their work.

In this thesis, transfer learning is used a bit differently. In our case, we want to learn new features, but instead of putting that besides the already existing ones, we want to combine the learned features into a new knowledge. So basically we don't want to use the simple features alone, but make a more complex of them and use only that. In our case, we don't want the model to generate only foregrounds or backgrounds, but combine the two of them into an image, that has both of them.

Although this idea is a new one, it could work based on the already existing usages of transfer learning. In this experiment we try to understand this as well.

5.3 Model

The whole model's simplified semantics can be seen in figure 5.2.

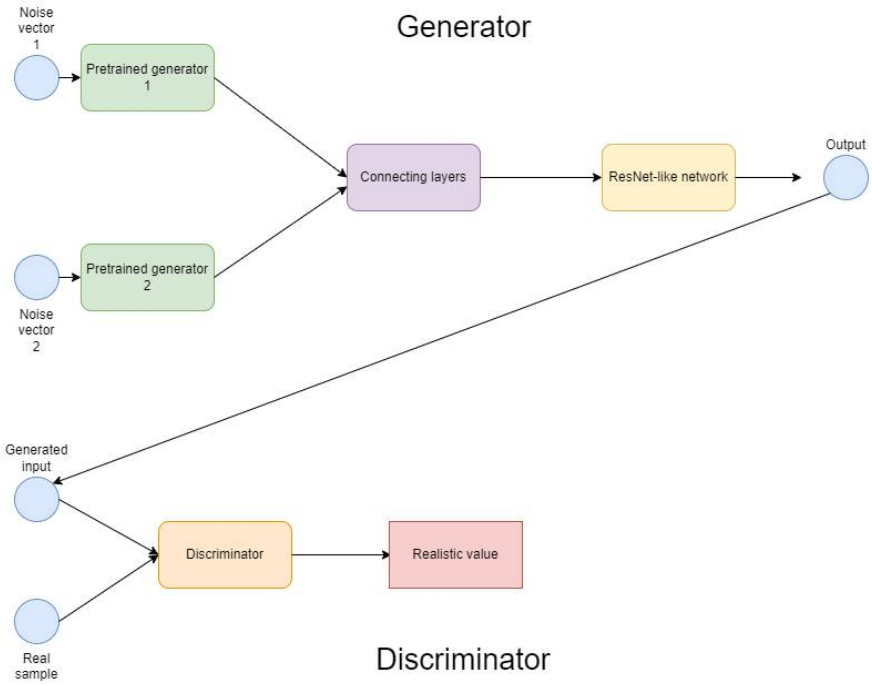

Figure 5.2: An example of a MITLGAN model.

The building of this model is very simple, but it takes many preparation. As we have to use pretrained models, we have to acquire them. If we don't have an already existing one at hand, we have to create it. As most of the existing GANs learn usually complex and many features, instead of a few, simpler feature, finding those is almost impossible. In the end, I had to create GANs, that do what this experiment needs. It is very important to note that those GANs doesn't have to be perfectly trained, because we only need simple features from it, not all of the features in the dataset, that they learned on. For example: in the case of foreground, we don't have to learn the backgrounds of the dataset (the dataset will have some simple background: usually black, or something very dark, that doesn't have big RGB values) only the foreground's basic features. If the foreground is a face, then learning that it has eyes, mouth, hair etc. is enough for us.

After we have those pretrained generators, we freeze the parameters of them and than we have to connect them somehow. The process I already explained in the Connecting two generators section.

After connecting the two, we have a tensor, that resembles an image. The only task left for us is an image-to-image translation. In our case, we added simple features together to get our "new input" and we want to learn, how to make a data, that resembles the new dataset (relative to the pretrained generators'), that has all of those features and possibly

some more. As the difference between the new input and the desired output is relatively small, we connect residual blocks in a way, that resembles ResNet. This way our network can be more deep and we only have to learn the residuals.

After we have our generator, we have to construct a discriminator. An idea would be, to combine the discriminators from the pretrained GANs, but there is two problems. One is, that the discriminator has only one input. One solution would be, to simply give the same input to both of the discriminators, but that wouldn't be right, as the new dataset is more complex, than what the discriminator learned on. It just wouldn't give the correct feedback, no matter how we try to combine the two discriminators' losses. The second is, when we train the generator, we calculate it respected to the discriminator and generator as well. As the discriminators learned with different datasets than the trainable parameters in the generator, the discriminators' part of the loss wouldn't be accurate enough.

Because of the aforementioned reasons, we make a brand new discriminator. I chose a WGAN's discriminator's model, so I can use Wasserstein distance in my experiment.

Chapter 6

Experiment setup

In this chapter I briefly explain, how the preparations were made for the experiment. First I enumerate the technologies used. After that I am introducing all the datasets, including the acquiring and the actual content.

6.1 Technologies

In the beginning phase, I thought, that my laptop has a good enough GPU to do the training on. But after a while I found out, that the GPU in my laptop is specifically made for laptops and those are weaker than the same product for PCs. Therefore I used Google Colab for my research. That is a free Google service, that provides a Python notebook environment in the cloud, really similar to Jupyter notebooks. The provided GPUs varied regularly. The service decides, which type of GPU it connects to. The possibilities are Nvidia K80, T4, P4 and P100. All of them had enough power, to run my codes, although for only a limited time a day. I used two Google accounts for research and that proved to be enough for researching only early training results, therefore I used that for seeing, if my model is good enough for continuation.

The environment already has many python packages installed. For building and training my neural networks, I used Tensorflow and Keras. The most packages were already available, but I had to install some add-ons for Tensorflow, to build my MITLGANs.

As my connection to GPUs were limited, I prepared my datasets beforehand on my laptop. I used Pycharm IDE, to make my datasets. When I didn't found a good dataset for my actual model, I used a javascript code to get the address of the images from a Google Images search and then I downloaded them using a python code. My data is stored in a Github project, so I could easily use them in the cloud.

My results were stored on my Google Drive account, because it can be easily used on Google Colab.

6.2 Datasets

In this section I briefly explain, what datasets I used, how I acquired them and how I used them.

6.2.1 Getting the data

The most data were acquired from Kaggle. Kaggle is a site, that shares many public datasets for data science and machine learning projects, holds competitions and shares many information about machine learning in general. The datasets there are simply downloadable with a click. I tried to find data, that were deliberately used for GANs, but not in all cases was I successful. The size of the sets were quite big, that I really hoped for, but the GPUs on Colab were not as fast, to iterate through all of them in the limited connection time, therefore I had to cut down on them regularly. This is really bad in terms of machine learning, as more the data, better the model can learn the features. For early results the less data are less influential, as simpler features are still learnable from them, but in some cases for the more complex features there were not enough data in the datasets. As I have already mentioned previously, the remainder data, that I couldn't get from Kaggle, I got from Google Images and then transformed into a usable format and shape.

6.2.2 Image augmentation

As I experienced through my studies in the subject, image augmentation helps a lot in training. When I could not get enough data from nowhere, then I used that technique to increase the size and variety of my data. I simply used a python script, that made from every image some augmented versions. The used augmentations were mirroring top-to-bottom and right-to-left, rotating, shearing and distortion. I only used them to transform the data a minimal amount, not to lose the main features of the dataset.

6.2.3 The datasets

I used all in all 8 datasets in this experiment, but they can be separated into 3 groups, as I researched 3 examples, each having 3 datasets: generating anime faces, pokemons with background and peoples faces with background. Only the pokemon and the anime datasets shared 1 dataset, all the others are unique in their way. All of the images were resized into 128X128, RGB images.

6.2.3.1 Anime faces

For generating anime faces with no background, I used a dataset from Kaggle. The final version of the dataset contained 2115 images. Although the images had background, all of them were white, therefore there was not much feature to learn, resulting in focusing only for the actual anime faces. (figure 6.1)

The dataset for the backgrounds were one, that I had to get from different sources, because initially I did not have remotely enough data from only Kaggle. In the end, it contains 2534 images. Most of them were images from anime, where no face can be found, or just in the distance. I focused on having many nature-like images, so the images were more like each other, not entirely different in each case. (figure 6.2)

The combined (foreground and background) dataset was acquired through Kaggle again. The final dataset had 2105 images. In many cases, the background was not as detailed, as in the background dataset, but was almost every one of them had a different background then white, therefore considered as real backgrounds, not like in the foreground dataset. (figure 6.3)

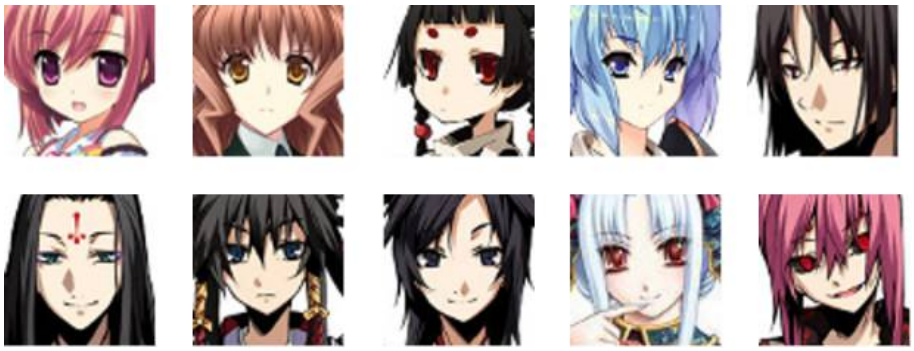

Figure 6.1: The anime faces dataset.

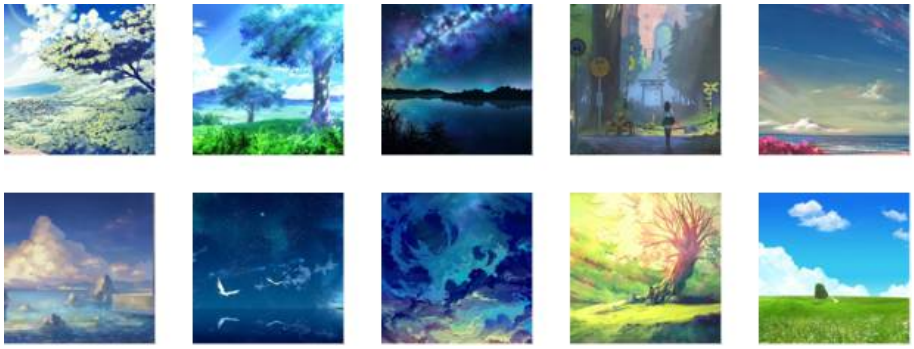

Figure 6.2: The anime and pokemon background dataset.

6.2.3.2 Pokemons

For the pokemons with no background, I used a dataset found on Kaggle. In the end the count of images stopped at 1758 images. As in the case of anime faces, the background "problem" was solved, with a background of only one color, in this case with black. (figure 6.4)

The dataset for the background was the same as anime faces task's. This is because the pokemons are also in animated series usually, therefore it makes sense to use the same dataset as in the case of anime faces' backgrounds. (figure 6.2)

The combined dataset was one, that was cut out from pokemon cards' images, found on Kaggle as well. I mainly cropped the images into a square one, and then resized them, as most of the pokemons were positioned in the middle of the art. After I deleted the images,

Figure 6.3: The anime faces with background dataset.

Figure 6.4: The only pokemons dataset.

where the pokemon was not on the image entirely or the art was too different from others, I got 2094 images. (figure 6.5)

Figure 6.5: The pokemons with background dataset.

6.2.3.3 Real faces

Finding a dataset, where the faces had only backgrounds with one colour, was really difficult, while most of the datasets contained faces from the outside or in a building, not previously set images, like in photo shootings. In the end I found a dataset on Kaggle, where photos with green and brown background were the ones, that I found good enough for this experiment. The count of images were 4884, double of the previous datasets, because many images were of the same person with slightly different emotions, therefore the shape of a face would not be learnable, if I stayed by the ca. 2000 images. (figure 6.6)

Figure 6.6: The real faces only dataset.

For the background generation, I found easily datasets, that had landscapes or buildings on them. As in a real photo, the background is not so sharp, I blurred these images, to resemble actual backgrounds of photos. The number of the pictures were 2520. (figure 6.7)

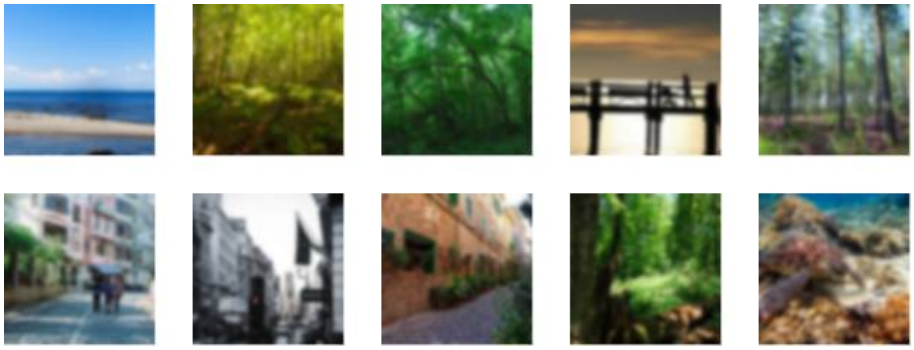

Figure 6.7: The real background dataset.

The combined dataset was easily acquirable, because of the exact reason, why the foreground dataset was hard to get. I got it from a Kaggle dataset and used 2505 images from it. (figure 6.8)

Figure 6.8: The faces with background dataset.

Chapter 7

The implementation of MITLGAN

In this chapter first I will write about the actual implementation of the models used in the experiment, then about the results of the experiment and a discussion part.

7.1 WGANs

WGAN was used for two reasons in this experiment: for the foreground and background pre-training and for evaluating, how fares MITLGAN compared to a simple WGAN. The implemented model is quite simple, there are multiple experiments and public code, that produces the same network.

The tables 7.1 and 7.2 are showing the exact networks used.

The input images' values were transformed from integers in range [0, 255] to float numbers in range [-1, 1], so the latent space gets more dense. It is very useful, because the less dense our latent space is, the less effective the training will be, because the distance between them in the latent space would be bigger, which would cause the network the impression, that the images are much more different, than they are in reality. One more good reason for transforming the data is, that the learned weights in the training would not be very big, therefore they can be easier updated. The generator's output is for sure between -1 and 1, because of the tanh activation function, therefore the real and fake data are in the same range value-wise. Of course the generated output that is not used for training gets to be transformed back into the range [0, 255], so we can display them correctly.

The batch size used in the training was 32, because the provided GPU in the cloud could only handle that much at once.

The training was also a very basic one. In one epoch we train the discriminator 5 times on the full available inputs (real + fake images), then we train the generator once. After 5 epoch I saved my models.

Good question would be, why didn't I use early stopping? That is because when training a GAN, we can't really evaluate our models truly, because it is an unsupervised machine learning model. In many cases the real evaluation comes after a person looks at some generated samples, if they are good enough or not. That is not measurable, if we don't know mathematically, what is the ground truth. If we would know it, then we would surely use a supervised machine learning model, so we could stop at the right time. Therefore many GAN training doesn't use early stopping, but of course there are exceptions.

This type of network were trained on all of the datasets in the experiment, meaning this model was used 8 times.

Layer	Output shape	No. param	Normalization	Activation function
Input	(1, 1, 1536)	0	-	-
Dense	(1, 1, 1536)	2365440	Batch	LeakyReLU
Reshape	(4, 4, 96)	0	-	-
Upsampling	(8, 8, 96)	0	-	-
Convolutional	(8, 8, 48)	41664	Batch	LeakyReLU
Upsampling	(16, 16, 48)	0	-	-
Convolutional	(16, 16, 24)	10464	Batch	LeakyReLU
Upsampling	(32, 32, 24)	0	-	-
Convolutional	(32, 32, 12)	2640	Batch	LeakyReLU
Upsampling	(64, 64, 12)	0	-	-
Convolutional	(64, 64, 6)	672	Batch	LeakyReLU
Upsampling	(128, 128, 6)	0	-	-
Convolutional	(128, 128, 3)	174	Batch	Tanh

Table 7.1: Implementation of the WGAN's generator. The LeakyReLU activation function was always implemented, so the negative inputs get a 0.2 multiplier. As it can be seen, transposed convolutional layers were used, but showing the layer as an upsampling and a convolutional layer shows much better, how the generated data shapes during the process. All of the convolutional layers used a (3, 3) kernel size and as many kernels as the output's 3rd channel, the strides were always (1, 1). As this model is training independently of the dataset, no bias neurons were used. The total number of parameters round up to 2421054, but only 2417796 are trainable. The last convolutional layer is the output layer.

7.2 MITLGAN

The main model of our implementation is a very simple one as well, but much more bigger in terms of layers and parameters. The connection of the two pretrained generators were made, just like I explained it in the MITLGAN chapter, using a concatenation and a convolutional layer. After that I implemented a very simple ResNet-like network. As in the case of WGANs, I didn't try to implement a new way of building a ResNet, I used a model that is widely used as a model for beginners and not so big, to keep the training in reasonable time.

In my implementation I used the Wasserstein distance as the loss function, therefore I could use the same discriminator, as in the case of the WGAN model in the previous section. This is good for the experiment, because we can see, how the new generator can learn with an already experimented discriminator model.

The generator is the really important part in this thesis. I initially loaded in the 2 pretrained WGAN's generators, and then connected them. I loaded in the last layer as well, as I wanted the outputs' values to be as minimal as possible, and the last tanh activation function does just that. I could have used a normalization technique for that, but I

Layer	Output shape	No. param	Dropout	Activation function
Input	(128, 128, 3)	0	-	-
Convolutional	(64, 64, 48)	3648	-	LeakyReLU
Convolutional	(32, 32, 96)	115296	-	LeakyReLU
Convolutional	(16, 16, 192)	460992	0.3	LeakyReLU
Convolutional	(8, 8, 384)	1843584	-	LeakyReLU
Flatten	(24576)	0	0.3	-
Dense	(1)	24577	-	-

Table 7.2: Implementation of the WGAN's discriminator. The LeakyReLU activation function was always implemented, so the negative inputs get a 0.2 multiplier. The convolutional layers used (5, 5) kernel size, as many kernels as the output's 3rd channel and (2, 2) strides. The total number of parameters round up to 2448097 and this time all of them were trainable. The last dense layer is the output layer.

thought that keeping the model as simple as it could be, will be now the best scenario and later on, if the model proves its right to exist, I can research that as well.

The ResNet-like model first uses convolutional layers and instance normalizations to create more deep tensors, more feature maps and then comes in the picture the residual steps. I used not just simple element-wise addition here, but the same idea, as I used when connecting the pretrained networks. As in the case of densely connected CNNs [14], all of the previous features get concatenated to each other, simply for the reason to keep the simpler features in the model in this case. It could help making more complex features after the residual blocks. The simpler features could help making the final output as this way they will not be forgotten. I already explained in the ResNet and MITLGAN chapter, that I found some solutions that used that, therefore this is still not a new way of using skip connection. Basically I used a dense block from the DenseNet network, but as I only hava one block, I concatenated many features with each skip connection.

All in all I implemented 6 residual blocks, as it seemed enough and not so big number of parameters that makes the training too long in the cloud (there were time-restrictions for the usage). In the residual blocks I used instance normalization. First, when I browsed the internet for already existing solutions I was sceptical, because of what I read previously about the topic. In the original paper [11] batch normalization were used. After more research, I found [28], where it can be clearly seen, that instance normalization is even a better choice sometimes, when the task comes up of style transfer. As that task can be seen as a type of image-to-image translation task, I thought, that can be used here, and proceeded to use the already found examples to implement the ResNet-like part of the generator.

After the residual blocks I restored the original 128X128X3 shape and combining all of the features learned in the ResNet-like part, using transposed convolutional layers and instance normalization.

Every normalization in this part of the network was instance normalization and the reason is the same in every usage: the task is still the same, no matter what part of this part of the generator I see, therefore I used the same normalization technique all the way. Mixing them together in this part would have been a move, that would have been causing the idea of the model to be more complex. Although there are many ideas in this model, the base

concepts are really simple and from the results hopefully understandable, if something is not a correct approach.

The whole implementation of the generators new part (from the first concatenation layer to the last layer) can be seen in table 7.3.

Layer	Output shape	No. param	Normalization	Activation function
Concatenate	(128, 128, 6)	0	-	-
Convolutional	(128, 128, 3)	21	-	ReLU
Convolutional	(128, 128, 64)	9600	Instance	ReLU
Convolutional	(64, 64, 128)	74112	Instance	ReLU
Convolutional	(32, 32, 256)	295168	Instance	ReLU
Convolutional	(32, 32, 256)	590592	Instance	-
Convolutional	(32, 32, 256)	590592	Instance	ReLU
Concatenate	(32, 32, 512)	0	-	-
Convolutional	(32, 32, 256)	1180416	Instance	ReLU
Convolutional	(32, 32, 256)	590592	Instance	ReLU
Concatenate	(32, 32, 768)	0	-	-
Convolutional	(32, 32, 256)	1770240	Instance	ReLU
Convolutional	(32, 32, 256)	590592	Instance	ReLU
Concatenate	(32, 32, 1024)	0	-	-
Convolutional	(32, 32, 256)	2360064	Instance	ReLU
Convolutional	(32, 32, 256)	590592	Instance	ReLU
Concatenate	(32, 32, 1280)	0	-	-
Convolutional	(32, 32, 256)	2949888	Instance	ReLU
Convolutional	(32, 32, 256)	590592	Instance	ReLU
Concatenate	(32, 32, 1536)	0	-	-
Convolutional	(32, 32, 256)	3539712	Instance	ReLU
Convolutional	(32, 32, 256)	590592	Instance	ReLU
Concatenate	(32, 32, 1792)	0	-	-
Upsampling	(64, 64, 1792)	0	-	-
Convolutional	(64, 64, 128)	2064768	Instance	ReLU
Upsampling	(128, 128, 128)	0	-	-
Convolutional	(128, 128, 64)	73920	Instance	ReLU
Convolutional	(128, 128, 3)	9417	Instance	ReLU

Table 7.3: Implementation of the MITLGAN's generator. As in the output shape column it can be seen, the ResNet idea combined with the DenseNet idea results into many feature map in the middle of the network. The kernel sizes and kernel numbers are varying, so the output shape will be correct each time. The kernel is (3, 3), the stride is (2, 2) everywhere except the convolutional layers, where the output shape (64, 64, 128). In those exceptions the kernel size is (7, 7) and the stride is (1, 1). The parameters count is 23304090, but only 18461982 is trainable, thanks to freezing the 2 pretrained generators' layers.

7.3 Results

First I will introduce the 3 experiment's results separately, then I will write about all of them generally. As I already wrote above. I will use only early results from trainings, therefore I will write about the features of the images most. The results of the pretrained generators can be found in the Appendix (with some images from the MITLGANs results as well), because they are not as important for understanding the results, as the 3 pair comparable models.

7.3.1 Real faces

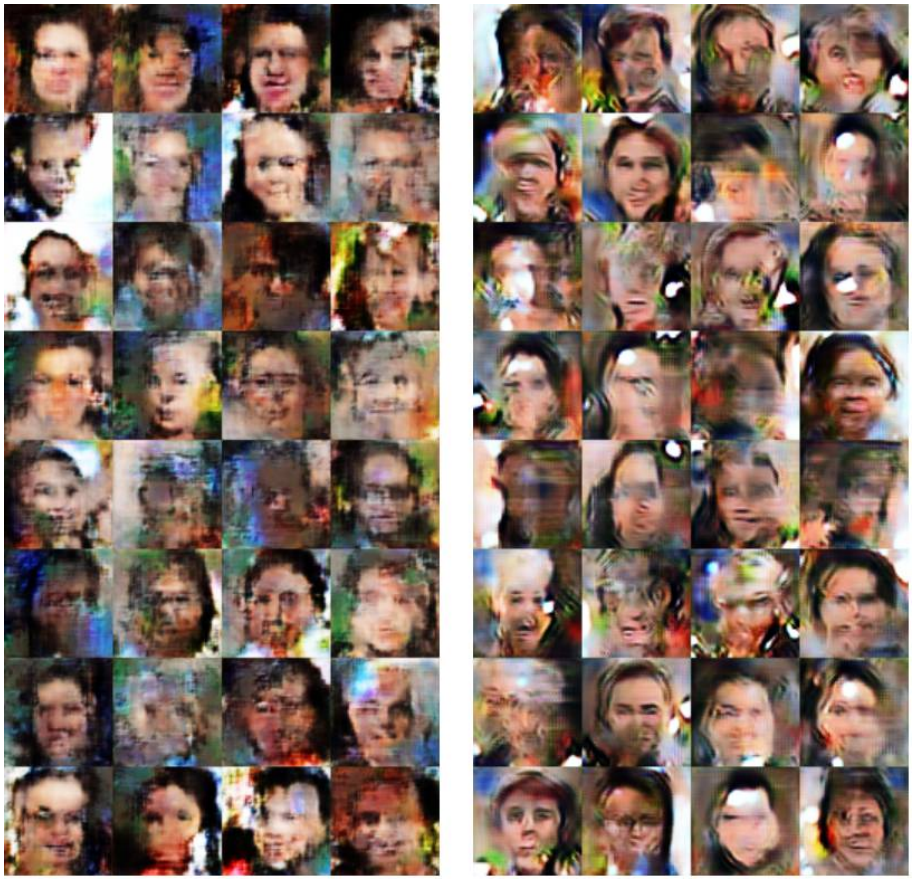

Figure 7.1: The left batch is made by WGAN, the right by MITLGAN.

On figure 7.1 we can see, how the generator fared after ca. 50 epochs in each case (the WGAN ran for 51 epochs, the MITLGAN ran for 46 epochs). From the images easy to see, that the MITLGAN in some cases left some white parts in the pictures, therefore not actually every value in the image got a proper value, that is really imaginable to be like that. Apart from that really conspicuous that the MITLGAN already rounded almost all the faces in some way (not every time optimal). The WGAN images are looking really checkered, but that was expected from the model in the early stages. The WGAN model has less images, that are unrecognizable entirely. The details on the faces are more or less equal, there are good and bad examples for that in both of the batches. The critical point of this experiment is, if the foreground and background are less mixed together or not. In

this case I would say, that among the best results the MITLGAN is better in that part, but its bad results are way worse than the WGANs.

7.3.2 Anime faces

Figure 7.2: The left batch is made by WGAN, the right by MITLGAN.

On figure 7.2 Many black lines can be seen on many of the images' side in both batches. This "feature" comes from the dataset, as there are images, where the actual image originally was not square, therefore it got padded. Aside from that, both of the batches has some really good and really bad qualities. Both of the images were ca. 110 epoch long trained (101 the MITLGAN, 126 the WGAN). Despite the short amount of training time, I think both of the models did quite good.

The WGAN model made some images, that are really close to the real dataset, but with some white lines on them. The WGAN almost in all cases could create anime-like face, I mean the eyes and hairstyle / -colour, that are really taking the anime face from some other faces apart. The background were not as big an issue for the model, although there are some images where the two of them couldn't entirely get apart.

The MITLGAN rounded the faces quite good in many cases, therefore it doesn't get that checkered feeling, as the WGAN's images. The pictures look like water paintings made by some children. The outlines are in many cases get a bit weird (for example the hairs outline gets out of control). The colors look okay here as well, but there are some faces, that got too many or less eyes than it is anticipated. The one great thing is, that there

are some images, where the background and the foreground is easily set apart. Although if the background got a little bit too colorful, then it got mixed up with the hair, which is not good in our experiment.

7.3.3 Pokemons

Figure 7.3: The left batch is made by WGAN, the right by MITL-GAN.

This experiment seems to have the most usable results in my opinion. On the figure 7.3 both of the batches are looking really good in a scientific viewpoint. Both of the models trained exactly for 86 epochs.

The WGAN has the same checkered feeling on it, but apart from that, the colours are great, and most of the pokemons are even recognizable as pokemons. The only bad thing I see is that in some cases the foreground gets blended in the background, so only with a bit harder look can someone locate the pokemon. The colors are beautiful, most of the images are really having unique qualities.

The MITLGAN has some black patches in most of the images and that is really not great. But apart from that, on every image the pokemon are easily locatable, although sometime only the silhouette can be seen. These pokemons also blend in the background sometimes, but I think a bit rarer than the WGAN's pokemons. It can be really seen, that the model put the foreground on the background and then tried to correct the colors accordingly. That colouring sometimes failed, resulting in a gray silhouette, but that is fixable with

some more training I think. The concept of joining the two dataset's features together here is the most recognizable.

7.3.4 General results & discussion

First I would like to write about the similarities of the MITLGAN results in terms of the image qualities. All of them had one big issue: missing parts. There were on all images spots, that didn't look like they belong to the image. The position of these spots were varying, sometimes they looked like plus eyes, sometimes just random little dots were "burnt out" of the images. With more training I think it is correctable, and some bigger dataset seems to be the part of the solution.

Looking at the fore- and backgrounds separately, we can see, that not many details were actually on the images. The outlines were in some cases okay, some cases complete disaster. The problem can be, that the pretrained WGANs didn't train enough, therefore didn't learn enough and enough complex features, to give to the ResNet-like part. But in many cases, the ResNet-like part did a very good job, to transform the image into something, that is close to the goal. As close, that only the sharpness and some simple details were not on the images sometimes and in every experiment there were some images, that looked exactly like that.

The colors were in every case spot on, I think if just looking at the colours, the MITLGAN did very great. Every image had something unique to it, so we can say that, despite the lack of data in all of the cases, the resulting images were promising.

Now I would like to examine all of the first epochs' training samples, so we can see, how good the concept looked like initially. (figure 7.4)

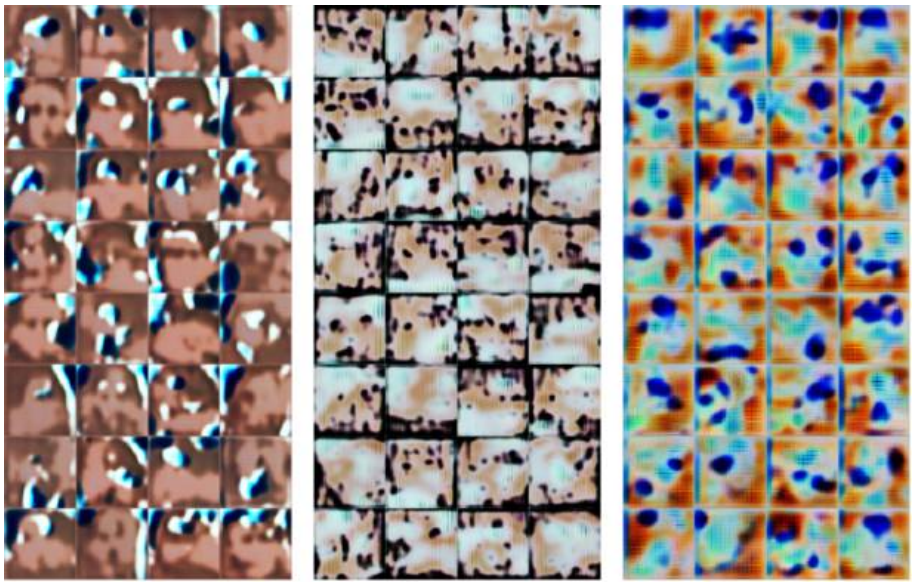

Figure 7.4: The first epochs' training samples. From left ro right: real faces, anime faces, pokemons.

On almost all of the images there were really contrasting colours. I think it happened, because after the first epoch we can see the most clearly the simplest features. These features were really recognizable on all of the inputs, as the most contrasting colours meant the foreground and background respectively in each case. Sometimes though it wasn't entirely the case, for example: on the anime faces, eyes got the colours of the

background, as if they were "holes". These first epoch results prove, that the idea is good, just some issues are, that have to be fixed later on.

One of the issues could be the good training process. Looking back at the training losses, the training looks like it didn't even really began in some cases. (figure 7.5)

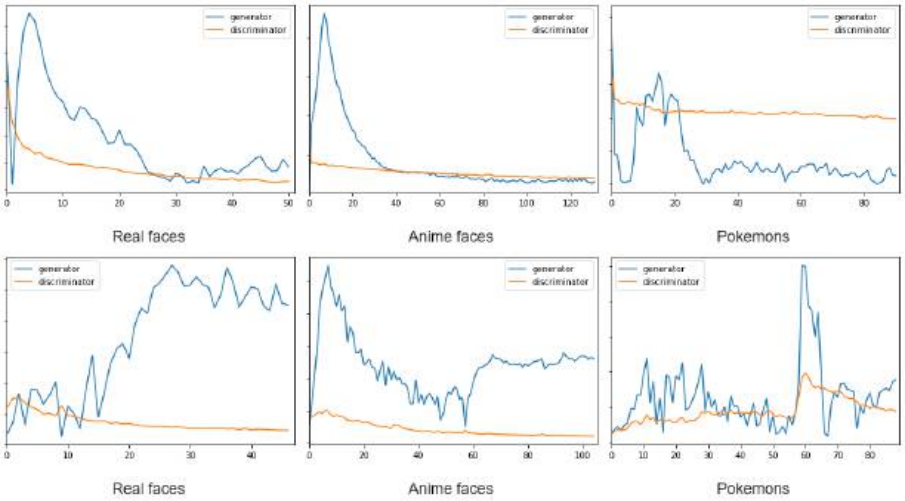

Figure 7.5: The absolute training loss values of the models (as the losses be negative as well in the case of Wasserstein distance). First row is the WGANs' trainings, the second row is the trainings of the MITLGANs.

All of the WGAN models could pull of a kind of good training, but the MITLGANs generators could not lessen their losses for long, the diagrams have big spikes. The discriminators worked more or less okay (in case of the pokemon dataset the MITLGAN's discriminator could not do so great, but much better than the generator). The problem could be the short training time, but maybe the reasons go deeper than that. The generator had really many parameters (although like one-fifth or one-sixth of an actual big network). There is the possibility, that after more training the many parameters could find an optimal line of training, but I think with less parameters the model would do better. As the discriminator got better and better, the generator also got better despite the losses, but not as much as it could have. If the training process doesn't become better, then the model will never have real, full results.

The evaluation of GANs with mathematical and statistical methods is not an easy task, while how accurate an image is, usually every person decides for itself. However there are some methods, that allows us to get a general guess, how good is the image exactly. For that I used Frechet Inception Distance (FID) [12]. The actual numbers are really big, because only early training results were discussed in this thesis. (table 7.4)

The only note I would like to add is, that the scores were really close, so the images we got, already universally resembles the experimented solutions' early generated images. This means, that improving on this model potentially can result in improving on the results we can already get with other models.

Model (dataset)	FID
WGAN (Real faces)	1464.5657
WGAN (Anime faces)	2083.7969
WGAN (Pokemons)	2571.0357
MITLGAN (Real faces)	1494.6489
MITLGAN (Anime faces)	2101.5293
MITLGAN (Pokemons)	2609.1826

Table 7.4: The evaluation of the models. Almost all of the scores are esentially the same, but the WGANs' score always a little bit higher.

Chapter 8

Conclusion

In the end, my model performed better, than I originally thought it would. Despite that, this model is not ready yet, to use for real life problems. The model needs to be reiterated a bit, because the training process was not as stable, as it would be preferred. I think using a regular ResNet or a regular DenseNet or some else network type can improve on the results and only after that, just slowly more techniques could be added to the model. The datasets were a bit small, therefore in the future concentrating on only one of the datasets would be wise (if not getting more powerful GPUs), and only after that trying again other datasets. In this step of development it was a great adventure, to use many datasets, mapping what the model can be in the future capable of.

The idea of connecting two generators in a GAN came as my own idea and after this experiment I can write down: This model can have a future, if it will be fully developed. The only issue with real life applications is, that in many cases some unique, new dataset has to be created in order to work with MITLGAN. But if that obstacle is solved, then it really can help improving on the images. The results are somewhat promising, as there were many images I came across during the training, that needed only a little polish and will look realistic. Future will tell, how well this model can be developed.

The initial question was: *"Can MITLGAN be used to improve on the early training process of a GAN and can really combine the already pretrained models' features properly?"* My answer is: Improving on the early stages may be a difficult task, but not impossible, if the training all in all gets better. The models features can be definitely combined I am certain of it, because most of the prelearned features were in the output in many cases.

If this model will be good enough with 2 pretrained models, maybe a 3rd of 4th could be added to it, using the same concept. This idea could be useful for reinforcing features as well, if the multiple initial datasets have same or similar features. Connecting networks like this in a generator, to use their knowledge has infinite potential, if we can learn, how to do it.

All in all I really liked working with this model, and learning about the whole world of deep learning and GANs. I especially enjoyed, when browsing the publications and papers online, I found so many interesting ideas, that I can use for my own thesis. I think it was a good choice, to develop a new model, because this way I was forced kindly to find answers for my problems and not just getting it, because someone already done it. I think this field is very entertaining to research in and has many potential.

Acknowledgements

Here I would like to express my gratitude towards my advisor, Krisztián Dániel Pomázi, because I was not always the best student, but he persevered and helped me every time I needed it. I am thankful for also everyone, who listened to my ideas related to this experiment and gave me useful advises or helped in any way.

Bibliography

[1] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[2] Aakash Bindal. Normalization techniques in deep neural networks. https://medium.com/techspace-usict/normalization-techniques-in-deep-neural-networks-9121bf100d8.

[3] Piotr Bojanowski, Armand Joulin, David Lopez-Paz, and Arthur Szlam. Optimizing the latent space of generative networks. *arXiv preprint arXiv:1707.05776*, 2017.

[4] Arun Kumar Dubey and Vanita Jain. Comparative study of convolution neural network's relu and leaky-relu activation functions. In *Applications of Computing, Automation and Wireless Systems in Electrical Engineering*, pages 873–880. Springer, 2019.

[5] Mohamed Elgendy. *Deep Learning for Vision Systems*. Simon and Schuster, 2020.

[6] Nuruzzaman Faruqui. What is kernel in image processing? https://www.nzfaruqui.com/what-is-kernel-in-image-processing/.

[7] Rohith Gandhi. Generative adversarial networks — explained. https://towardsdatascience.com/generative-adversarial-networks-explained-34472718707a.

[8] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[10] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of wasserstein gans. *arXiv preprint arXiv:1704.00028*, 2017.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[12] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30, 2017.

[13] Tobias Hill. A neural network from scratch - part 2: Gradient descent and backpropagation. https://towardsdatascience.com/part-2-gradient-descent-and-backpropagation-bf90932c066a.

[14] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[15] Jonathan Hui. Gan — ways to improve gan performance. https://towardsdatascience.com/gan-ways-to-improve-gan-performance-acf37f9f59b.

[16] Tammy Jiang, Jaimie L Gradus, and Anthony J Rosellini. Supervised machine learning: a brief primer. *Behavior Therapy*, 51(5):675–687, 2020.

[17] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of stylegan. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8110–8119, 2020.

[18] Kui Liu, Guixia Kang, Ningbo Zhang, and Beibei Hou. Breast cancer classification based on fully-connected layer first convolutional neural networks. *IEEE Access*, 6: 23722–23732, 2018.

[19] Ange Lou, Shuyue Guan, and Murray H Loew. Dc-unet: rethinking the u-net architecture with dual channel efficient cnn for medical image segmentation. In *Medical Imaging 2021: Image Processing*, volume 11596, page 115962T. International Society for Optics and Photonics, 2021.

[20] Batta Mahesh. Machine learning algorithms-a review. *International Journal of Science and Research (IJSR).[Internet]*, 9:381–386, 2020.

[21] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[22] Anh H. Reynolds. Convolutional neural networks (cnns). https://anhreynolds.com/blogs/cnn.html.

[23] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[24] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *towards data science*, 6(12):310–316, 2017.

[25] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):1–48, 2019.

[26] Yingyi Sun, Wei Zhang, Hao Gu, Chao Liu, Sheng Hong, Wenhua Xu, Jie Yang, and Guan Gui. Convolutional neural network based models for improving super-resolution imaging. *IEEE Access*, 7:43042–43051, 2019. DOI: 10.1109/ACCESS.2019.2908501.

[27] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. A survey on deep transfer learning. In *International conference on artificial neural networks*, pages 270–279. Springer, 2018.

[28] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016.

[29] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Adversarial generator-encoder networks. *arXiv preprint arXiv:1704.02304*, 2, 2017.

[30] Richard Vogl. *Deep Learning Methods for Drum Transcription and Drum Pattern Generation*. PhD thesis, 11 2018.

[31] Nicholas Walker, Ka-Ming Tam, and Mark Jarrell. Deep learning on the 2-dimensional ising model to extract the crossover region with a variational autoencoder. *Scientific reports*, 10(1):1–12, 2020.

[32] Lilian Weng. From gan to wgan. *arXiv preprint arXiv:1904.08994*, 2019.

[33] Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018.

[34] Guijuan Zhang, Yang Liu, and Xiaoning Jin. A survey of autoencoder-based recommender systems. *Frontiers of Computer Science*, 14(2):430–450, 2020.

[35] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232, 2017.

Appendix

A.1 Real faces

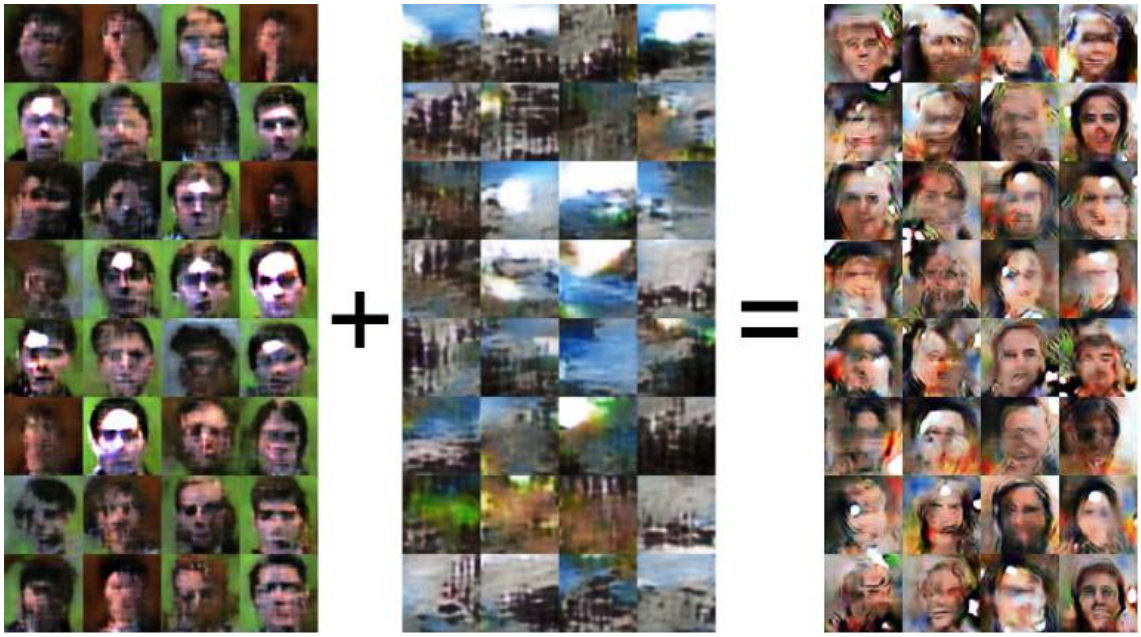

Figure A.1.1: The last epochs' results. From left to right: only face, background, both (MITLGAN).

A.2 Anime faces

Figure A.2.1: The last epochs' results. From left to right: only face, background, both (MITLGAN).

A.3 Pokemons

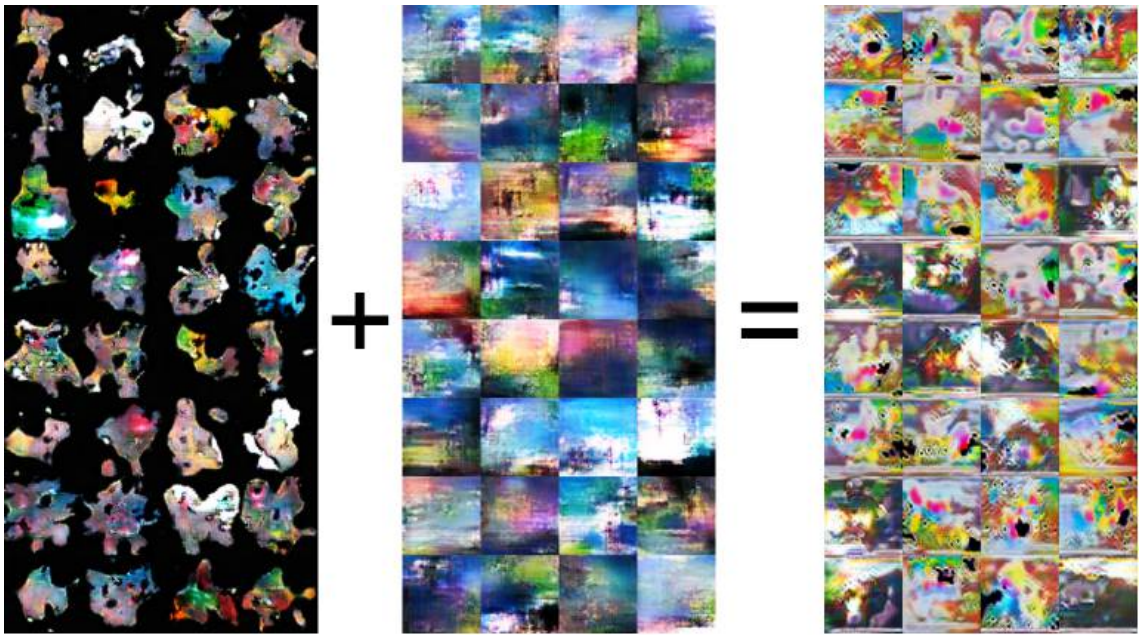

Figure A.3.1: The last epochs' results. From left to right: only pokemon, background, both (MITLGAN).