FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Patrik Dokoupil

Generating synthetic data for an assembly of police lineups

Department of Software Engineering

Supervisor of the master thesis:	Mgr. Ladislav Peška, Ph.D.
Study programme:	Computer Science
Study branch:	Artificial Intelligence

Prague 2021

Hereby I would like to thank my supervisor Mgr. Ladislav Peška, Ph.D. for his time, suggestions, and all the help he gave me during my work on this master thesis. Furthermore, I would like to thank RNDr. Jakub Yaghob, Ph.D. for his help and support regarding computational infrastructure. Finally, I would like to thank my family, friends, and everyone else who has supported me during my studies.

Title: Generating synthetic data for an assembly of police lineups

Author: Patrik Dokoupil

Department: Department of Software Engineering

Supervisor: Mgr. Ladislav Peška, Ph.D., Department of Software Engineering

Abstract: Eyewitness identification plays an important role during criminal proceedings and may lead to prosecution and conviction of a suspect. One of the methods of eyewitness identification is a police photo lineup when a collection of photographs is presented to the witness in order to identify the perpetrator of the crime. In the lineup, there is typically at most one photograph (typically exactly one) of the suspect and the remaining photographs are the so-called fillers, i.e. photographs of innocent people. Positive identification of the suspect by the witness may result in charge or conviction of the suspect.

Assembly of the lineup is a challenging and tedious problem, because the wrong selection of the fillers may end up in a biased lineup, where the suspect will stand out from the fillers and would be easily identifiable even by a highly uncertain witness. The reason why it is tedious is due to the fact that this process is still done manually or only semi-automatically.

This thesis tries to solve both issues by proposing a model that will be capable of generating synthetic data, together with an application that will allow users to obtain the fillers for a given suspect's photograph.

Keywords: GAN, generative adversarial network, police lineup, deep learning, image generation

Contents

Introduction

Eyewitnesses often play an important role in identifying possible suspects and uncovering the details about a crime. One of the eyewitness identification procedures is called *police photo lineup*, sometimes also referred to as a photo array, police lineup, or just lineup. An example of a simple police lineup is depicted in Figure 1.

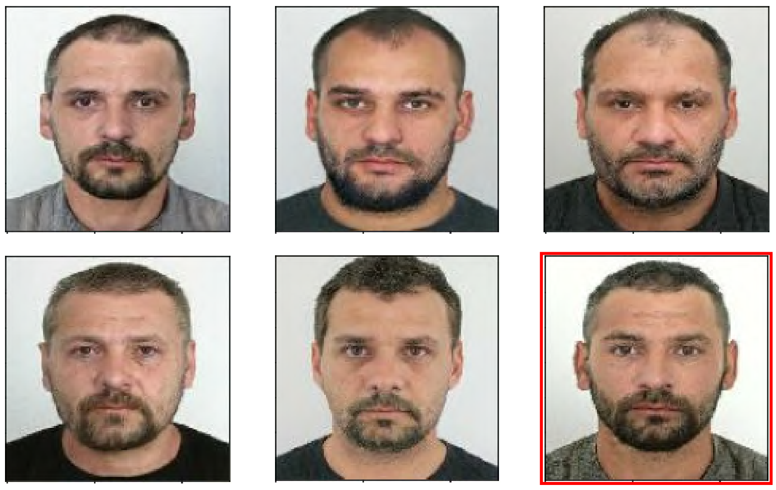


Figure 1: Example of a police photo lineup. These images were artificially generated and do not show real persons.

The evidence provided by this procedure can lead to incrimination and eventually to the conviction of the suspected criminals, therefore it is crucial that the eyewitness evidence is as reliable and as accurate as possible, in order to prevent the possibility of charging and convicting innocent suspects.

In the context of this thesis, police lineups refer to the process of presenting a collection of photographs to a witness in order to determine if the witness can recognize a person involved with the crime. There are two types of photographs that could appear in a police lineup—the suspect, which is a photograph of a person who police believe has committed the crime, and so-called *filler* which is a photo of a person other than the suspect.

The example of a lineup in Figure 1 shows 6 images, where the image with a red border is the suspect and the remaining 5 images are the filler. Using lineups of size 6 is quite common in practice but the actual size could be affected by various factors, one of them is a country. For example, in the United States, the lineup size is typically 6, but in the United Kingdom, lineups of size 9 are more common [1].

The accuracy and reliability of police lineups are highly affected by the way how the fillers are selected [2]. It is very important that the selected fillers are similar in appearance and overall in all the characteristics including gender, age, race, and other extraordinary facial features [2]. The selected photographs should also keep the consistency of color, quality, size, and resolution [3][4].

Typically, the lineup contains only one suspect and the rest are fillers, but there are situations or variations of the lineup where the suspect's photo was replaced by a filler. Before the identification process, the witness should be told that the suspect may or may not be present in the lineup [5]. The witness is then asked whether he or she recognizes anyone from the presented images.

It is generally agreed that using low-similarity fillers increases the risk of mistaken identification of an innocent suspect [6][7]. Lineups consisting of low-similarity fillers are called *biased lineups* [8]. An artificial example of a lineup that could be considered unfair or biased is shown in Figure 2. On the other hand, the lineup from Figure 1 seems to be fair, because all the images show persons with a similar appearance, age, and other characteristics.

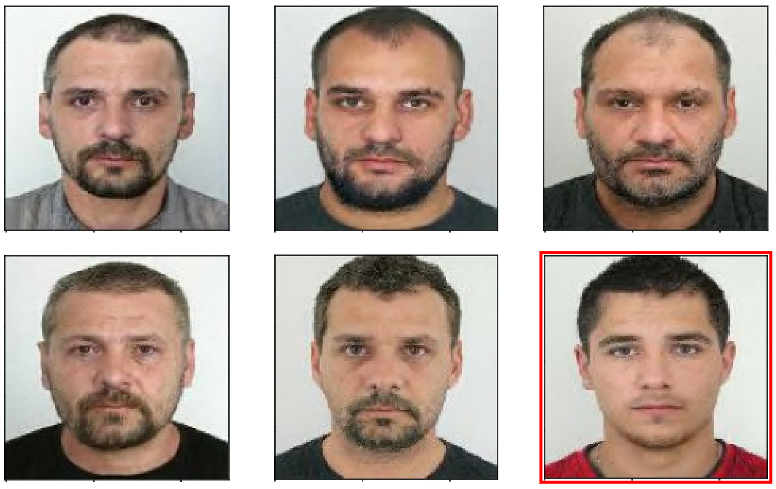


Figure 2: Example of an unfair or biased police photo lineup. These images were artificially generated and do not show real persons.

The reason why the police lineup from Figure 2 might be considered biased is that the suspect is very different from the fillers. Notice that the suspect is the only one who has no beard and that the suspect is considerably younger than the fillers. Some attention could also be attracted by the garish red t-shirt. The unfairness can be even more prominent when there is such a significant difference between the fillers and the suspect as for example different ethnicity.

There are two main strategies for selecting the fillers, namely: *match-to-description* strategy and *resemble-suspect* strategy [9][10]. The former strategy uses a verbal description of the suspect that was provided by the witness, while the latter strategy involves selecting fillers who physically resemble the suspect. One of the difficulties of the resemble-suspect strategy is to determine a threshold of how similar the fillers should be to the suspect [10]. If the fillers are not similar enough, it would lead to a biased lineup [10]. On the other hand, if the fillers are too similar to the suspect, the resulting lineup would be a collection of near-clones, therefore making it too difficult to identify the suspect from such a lineup. The match-to-description strategy does not have this issue, because it is simple enough that the fillers comply with the verbal description given by the witness. Despite this seeming disadvantage of the first strategy, both strategies were shown to be equally effective in reducing innocent suspect identifications. Some studies show that the resemble-suspect strategy might result in a reduction of accurate identifications of the suspect, while the match-to-description strategy does not [10][8]. However, other studies have shown that there is no significant difference between the strategies [11] and their impact on making the suspect stand out [9]. More information about police lineups and recommendations regarding their construction and organization of the whole process could be found in the following article [12].

Assembling unbiased lineups is a challenging and time-consuming task because large databases of faces have to be traversed in order to find appropriate fillers [13]. It is still very common that this process is being done in a manual way which makes the assembling even slower. However, with the increasing advancement of technology, there were several attempts to simplify this task by making it (semi-)automatic. A certain level of automation can be achieved by using tools that are able to search the database for the fillers that are similar to the suspect and then "recommend" [14][13] these fillers to the lineup administrator.

Automatic lineup assembling has the potential to greatly simplify and speed up the whole process, yet still, some problems are preventing the automation from being successful. There are two issues connected to the automatic search of the face databases. First is that it is difficult to define the similarity of two faces so that it matches people's notion of the similarity. Second, even when the similarity measure is available, it may happen—and it actually frequently happens in practice—that the database simply does not contain face photos that are similar enough to the suspect, i.e. there are either not enough appropriate fillers in the database or there are no appropriate fillers at all [14][13].

The lack of appropriate fillers in the photo databases could be solved by extending the database and an elegant way to do this would be to do it in an automatic way using some kind of generative model that would be capable of generating face photographs. We presume that this could be done, thanks to the recent advances in the fields of artificial intelligence, especially machine learning and deep learning.

Goals

The difficulties connected with the process of police lineup assembling, most notably the problem of finding appropriate fillers lead us to the main goal of this thesis which is to propose a new variation of—or to adapt—an existing model to a task of generating synthetic face photographs. The proposed model has to adhere to the following requirements, which are essential for building unbiased lineups:

1. The model should be able to generate images with a reasonable quality (in terms of image resolution).
2. The generated faces should not be easily distinguishable from photos of real people, i.e. it should not be obvious that the image does not depict a real person.
3. The output of the model should be diverse enough.
4. The output of the model should be controllable, meaning that the consumer of the model should have some control over the resulting images.

Apart from the critical requirements mentioned above, there are some features which would be appropriate to achieve, when it would be possible:

1. The model should be capable of generating images of people with extraordinary or rare facial features.

2. The control over the model's output should be achieved by seeding or initializing the model with a photo of the suspect and the generated images then should be similar to the seed image (in terms of facial features).

Afterward, the resulting model should be incorporated into a framework for the assembly of the police photo lineups.

1. Background

This chapter presents a brief introduction to artificial neural networks, followed by a description of the main types of deep neural networks that are needed for understanding the rest of this thesis. This chapter does not attempt to provide a comprehensive and rigorous description of all the principles of artificial neural networks as this is something out of the scope of this thesis. Instead, a special emphasis is taken on a description of the core principles of the two types of deep neural networks, namely: generative adversarial networks and autoencoders as these two will play an essential role throughout the whole thesis. It should be mentioned that sometimes the terminology is not unambiguous and similar ideas are called by different names and conversely different things are given similar or same names. Rather than trying to describe all the possible variations of the presented models and theoretical constructs, this chapter only presents these variants that are assumed throughout this thesis by its author. It should be mentioned that at least a basic knowledge of machine learning terminology and its techniques is assumed, as these topics are used throughout this thesis even though they are not described in this chapter. Machine learning related information can be found, for example, in any of the following books [15], [16], [17].

1.1 Artificial neural networks

Artificial neural networks are a computational model that is inspired by biological neural networks present in animal brains. Although there are many parallels between artificial neural networks and biological neural networks, artificial neural networks are rather only an oversimplification of their biological counterpart.

Based on their biological inspiration, artificial neural networks consist of processing units that are connected by edges. These units are also sometimes called artificial neurons. Connections between the units mimic the purpose of synapses in the biological brain so that they are able to transmit a signal to other units. Each individual connection has associated a real-valued weight that determines the importance of that connection. These weights[1] are adaptive and are adjusted during a process that is called learning or training. A unit can receive a signal, perform computation and then send a signal to its neighboring units. In artificial neural networks, these signals are represented by real numbers, and outputs of the units are calculated by first taking a weighted sum of the neuron's inputs in order to obtain an activation which is then passed through the activation function to yield the final output.

1.1.1 Historical overview

The first computational model of an artificial neuron was introduced by McCulloch [18] in 1943. This model of a neuron expected boolean inputs and produced boolean output. Schematics of McCulloch's neuron is depicted in Figure 1.1.

[1]The weights are also often called parameters.

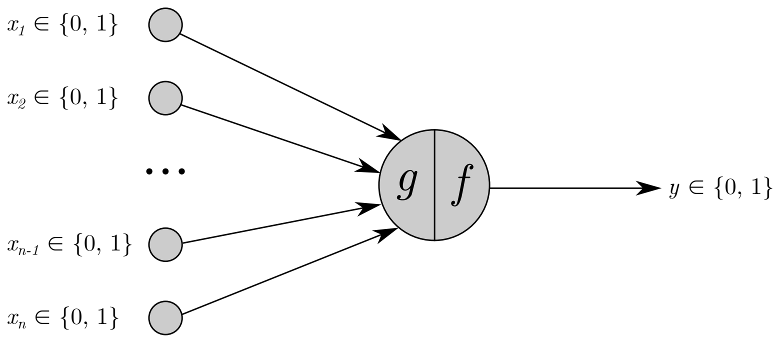


Figure 1.1: Schematics of McCulloch's neuron model.

In Figure 1.1, the links denoted by $x_1, \ldots, x_n$ correspond to the inputs of the neuron, g is called aggregation function and f is a function computing the output y, often called activation[2] function. Frequently it is convenient to use vector notation for the inputs because it allows them to write the inputs in a more concise way:

$$\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$$

The aggregation function g typically performs a summation of the inputs:

$$g(\boldsymbol{x}) = \sum_{i=1}^{n} x_i$$

The inputs to the neuron were either inhibitory or excitatory and based on it, the output of the neuron is computed by using the following formula:

$$y = \begin{cases} 0 & \text{if any } x_i \text{ is inhibitory} \\ y = f(g(\boldsymbol{x})) & \text{otherwise} \end{cases}$$

Definition of the output function f is following:

$$f(g(\boldsymbol{x})) = \begin{cases} 1 & \text{if } g(\boldsymbol{x}) \geq \theta \\ 0 & \text{if } g(\boldsymbol{x}) < \theta \end{cases}$$

Parameter θ is the so-called threshold. When an output of the neuron is equal to 1 it is said the neuron fires.

This model had several limitations, most notably the fact that both inputs and outputs were boolean variables, thresholding parameter had to be set by hand, and that this kind of neuron was only able to represent linearly separable functions (so for example, XOR function could not be represented by this kind of neuron).

In 1958—after McColluch's work—Frank Rosenblatt [19] introduced a new and improved model that was called the classical perceptron model. Rosenblatt's model can be thought of as the simplest model of a contemporary neural network. The classical perceptron model is a generalization of the McColluch neuron model

[2]Using term "activation" for a simple McCulloch's neuron may seem excessive, but actually the function for computing an output of the McCulloch's neuron can be implemented using hard limit function as an activation.

and it brought several improvements which have remained present in neural networks to this day. The main improvements are the introduction of numerical weights for inputs and a mechanism for learning these weights. Also, the inputs are no longer restricted to boolean values. The schematics of the classical perceptron model are shown in Figure 1.2.

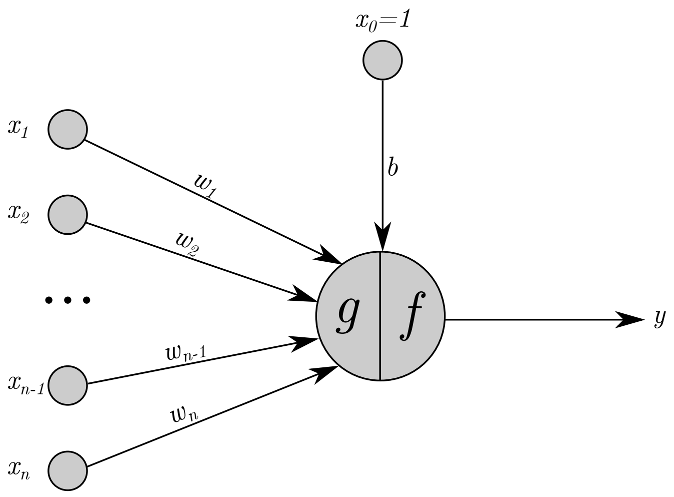


Figure 1.2: Schematics of Rosenblatt's classical perceptron model.

In Figure 1.2, the components of the model (g, f, $\boldsymbol{x}$ and y) have the same meaning as in McCulloch's neuron model. The only differences are the fact that the inputs are no longer binary, weights were added to the connections and there is a new term called bias b. The computation of the classical perceptron model can be obtained as the following generalization of McCulloch's neuron computation: The aggregation function g is adjusted as follows:

$$g(\boldsymbol{x}, \boldsymbol{w}) = \sum_{i=1}^{n} x_i w_i$$

But now, instead of using f in the same way as it was done with McCulloch's perceptron, it is often convenient to get rid of the θ threshold by using the following transformation:

$$f(g(\boldsymbol{x}, \boldsymbol{w})) = \begin{cases} 1 & \text{if } g(\boldsymbol{x}, \boldsymbol{w}) = \sum_{i=1}^{n} x_i w_i \geq \theta \\ 0 & \text{if } g(\boldsymbol{x}, \boldsymbol{w}) = \sum_{i=1}^{n} x_i w_i < \theta \end{cases}$$

could be rewritten into the following equation:

$$f(g(\boldsymbol{x}, \boldsymbol{w})) = \begin{cases} 1 & \text{if } g(\boldsymbol{x}, \boldsymbol{w}) = \sum_{i=1}^{n} x_i w_i - \theta \geq 0 \\ 0 & \text{if } g(\boldsymbol{x}, \boldsymbol{w}) = \sum_{i=1}^{n} x_i w_i - \theta < 0 \end{cases}$$

the $-\theta$ part could get replaced by b together with adding new weight $w_0 = 1$ in order to finally arrive at the following, more concise definition of f

$$f(g(\boldsymbol{x}, \boldsymbol{w})) = \begin{cases} 1 & \text{if } g(\boldsymbol{x}, \boldsymbol{w}) = \sum_{i=0}^{n} x_i w_i \geq 0 \\ 0 & \text{if } g(\boldsymbol{x}, \boldsymbol{w}) = \sum_{i=1}^{n} x_i w_i < 0 \end{cases}$$

Although the classical perceptron model solves some issues of McCulloch's neuron model, others still remain present, for example, the classical perceptron model is only able to implement linearly separable functions. To remove this constraint, two key concepts are needed, namely: the notion of layers and activation functions both of which will be discussed in the subsequent sections of this chapter.

1.1.2 Organization of neurons

The neurons are typically organized into layers where a single layer means a group of neurons. Neurons within a single group do not have any connections within each other, but they are typically connected to neurons in the previous layer and/or to the neurons in the next layer. This idea is illustrated in Figure 1.3. The signal is transmitted from the first layer (input layer) to the last layer (output layer), the layers in between these two layers are called hidden layers.

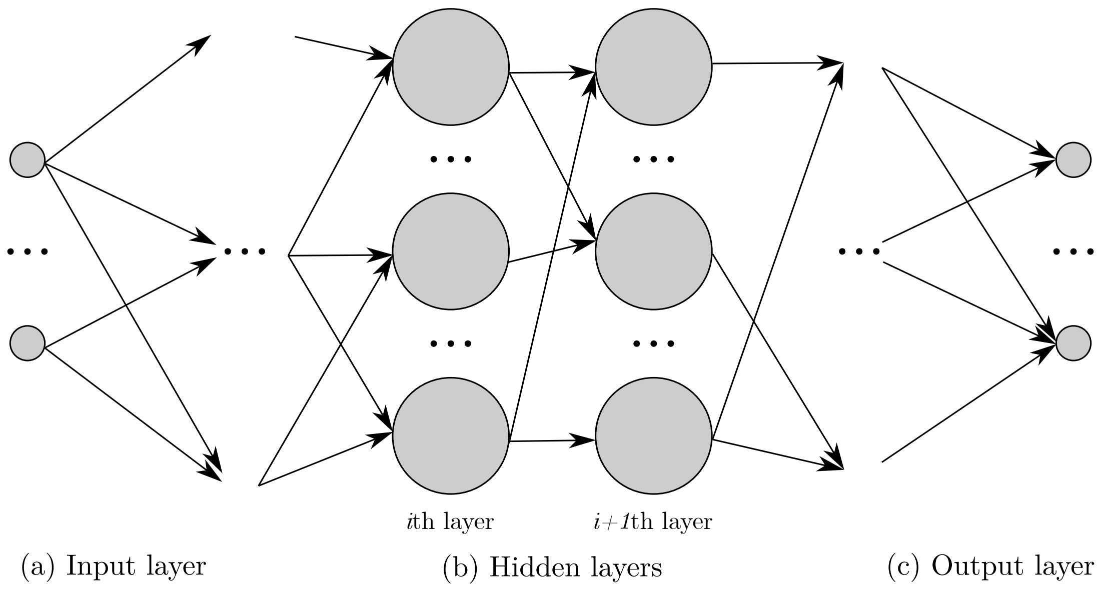


Figure 1.3: Organization of neurons in input, hidden, and output layers.

The left image in Figure 1.3 shows the organization of the input layer. Notice that the neurons in this layer do not have any input connections going from the left, but only output connection that is going to the next layers (to the right).

The middle image in Figure 1.3 shows how the neurons are organized in the hidden layers. In each hidden layer, neurons are connected to some neurons from the previous layer (which might be either the precedent hidden layer or an input layer) and to some neurons from the next layer (which is either the next hidden layer or an output layer).

Lastly, the right image in Figure 1.3 shows an organization of neurons within the output layer, which is symmetrical to the input layer, i.e. there are no output connections going from the neurons of this layer and instead, only input connections are present (from the left).

1.1.3 Feedforward networks

Feedforward networks are a kind of neural network architecture where the units and their connections form a directed acyclic graph. Feedforward networks typically consists of a special kind of a layer called dense layer. As the name “dense” says, the neurons in such a layer are densely connected, which means that a single neuron is connected to all neurons from a previous layer (receiving the inputs) and also to all neurons in the next layer (passing the outputs). This is illustrated in the Figure 1.4.

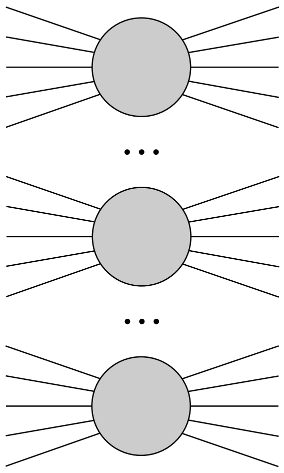


Figure 1.4: Illustration of a dense layer.

Computation of the feedforward networks consisting of dense layers can be written in the following, very concise way:

$$\boldsymbol{y} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

or simply

$$\boldsymbol{y} = f(\boldsymbol{W}\boldsymbol{x}) \text{ if } W_{i,0} = 1 \; \forall \; i \wedge x_i = b_i$$

Single-layer-perceptron

The simplest architecture of a neural network is called a single-layer perceptron. This architecture does not contain any hidden layers, but only the input and output layers. The architecture of a single-layer perceptron is depicted in Figure 1.5.

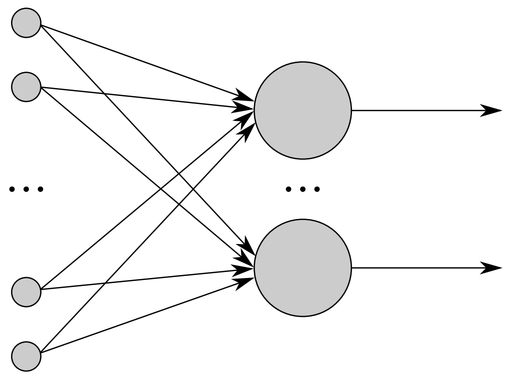


Figure 1.5: The architecture of a single-layer perceptron.

Notice that the architecture depicted in Figure 1.5 basically corresponds to stacking multiple Rosenblatt’s classical perceptrons onto each other.

The output of a single layer perceptron is computed in a way that the computation of output for Rosenblatt's perceptron is performed for each output neuron separately. Therefore the output will be a vector, where each output neuron corresponds to a single scalar component of that vector. The main drawback of a single-layer perceptron is the fact that it can only learn linear functions.

Multi-layer perceptron

The idea of a single-layer perceptron can be improved and further generalized to the so-called multi-layer perceptron, where there are one or more hidden layers in between the input and output layer. Figure 1.6 shows an example of multi layer perceptron.

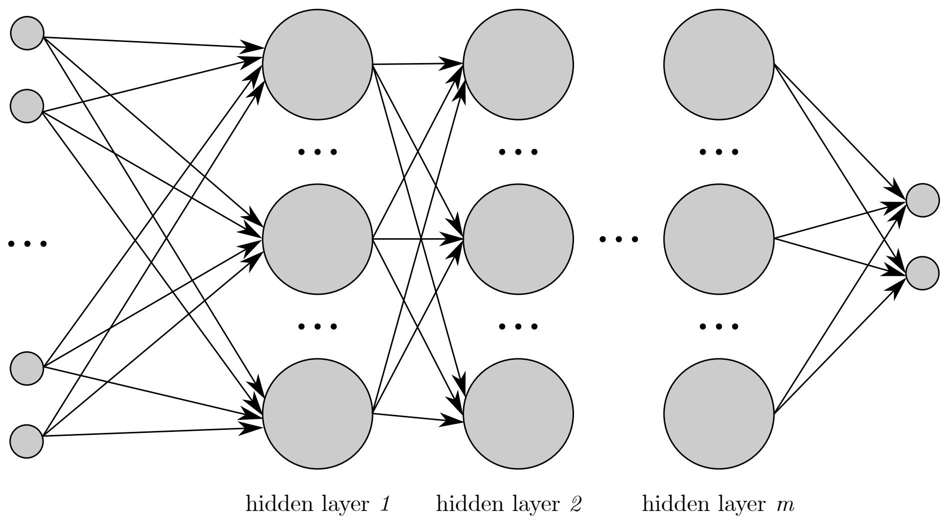


Figure 1.6: The architecture of a multi-layer perceptron.

Multi-layer perceptron is capable of learning non-linear functions (i.e. XOR problem can be solved), but there is one more thing needed for gaining this ability—non-linear activation [20]. There exists various kinds of activation functions, a few most frequently used activation functions are the following: *ReLU*, *tanh*, *sigmoid*. Activation function can also be applied to the output layer, frequently this is done with for example *sigmoid* or $softmax$ functions. The definitions of the aforementioned activation functions are following:

$$\text{ReLU: } max(0, x) \tag{1.1}$$

$$\text{tanh: } tanh(x) \tag{1.2}$$

$$\text{sigmoid: } \sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.3}$$

The *ReLU*, *tanh* and *sigmoid* activation functions are shown in Figure 1.7.

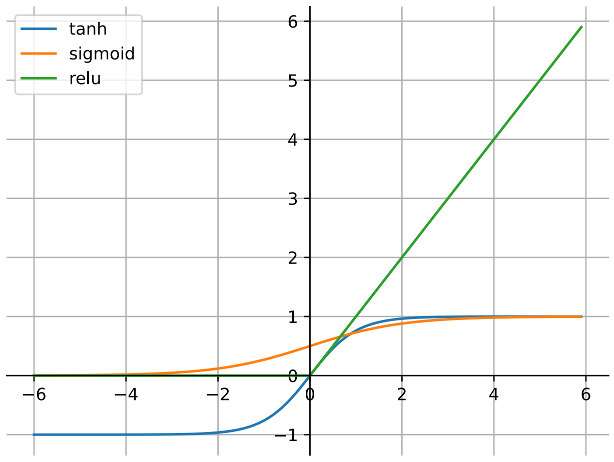


Figure 1.7: Plot of a *ReLU*, *tanh* and *sigmoid* activation functions.

1.1.4 Function approximation

Another way to look at artificial neural networks is from a function approximation point of view. When looking from this perspective, the main goal of a (feedforward) network is to approximate a function $y = f^*(\boldsymbol{x})$ that maps an input $\boldsymbol{x}$ to an output y. The network defines a mapping $\hat{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ where $\boldsymbol{\theta}$ are the parameters that the network tries to learn in such a way that the resulting function is the best function approximation of $f^*(\boldsymbol{x})$. When the network has several layers, each layer describes a function and the resulting function represented by the whole network is a composition of these functions, i.e.:

$$y = f^{(m)}(f^{(m-1)}(\dots f^{(2)}(f^{(1)}(\boldsymbol{x})))) \tag{1.4}$$

with $f^{(1)}$ corresponding to the first layer, $f^{(2)}$ corresponding to the second layer, and so on all the way up to the $f^{(m)}$ which corresponds to the mth layer.

One of the main theoretical results about multi-layer feedforward networks is called Universal Approximation theorem [21] which can be formulated in the following way [22]:

Theorem 1. *Let $f(x)$ be a nonconstant, bounding and nondecreasing contiuous function. Then for any $\epsilon > 0$ and any continuous function f^* on $[0,1]^m$ there exists an $N \in \mathbb{N}, v_i \in \mathbb{R}, b_i \in \mathbb{R}$ and $\boldsymbol{w_i} \in \mathbb{R}^m$ such that if we denote*

$$F(\boldsymbol{x}) = \sum_{i=1}^{N} v_i f(\boldsymbol{w_i} \cdot \boldsymbol{x} + b_i)$$

then for $\forall \boldsymbol{x} \in [0,1]^m$:

$$|F(\boldsymbol{x}) - f^*(\boldsymbol{x})| < \epsilon$$

Roughly speaking, this theorem says that any multi-layer neural network with at least a single hidden layer has enough power to approximate arbitrary continuous function f^* to a desired degree of accuracy (specified by ϵ) assuming that it uses activation function f that satisfied all the abovementioned conditions. More rigorous explanation including formal proofs can be seen in [21].

More generally, this theorem was also proven for some of the unbounded activation functions [23] e.g. for *ReLU*.

1.1.5 Learning and backpropagation

As it was already mentioned above in Section 1.1, learning means the adaptation of the weights of the neural network. During the training of a network, the network weights $\boldsymbol{\theta}$ are adjusted so that the function approximation $f(\boldsymbol{x};\boldsymbol{\theta})$ better matches the function f^*. At the beginning of the training, the weights of the network are typically initialized to random values [24].

In order to be able to adjust the weights so that the function approximation $f(\boldsymbol{x};\boldsymbol{\theta})$ "gets closer" to the function, f^* some function measuring the quality of the current approximation is needed. This function is called loss function, sometimes also called objective function or cost function (although in some special contexts these two terms might have a slightly different meaning). The goal of the learning process then can be formulated as minimizing the loss function on the training data with respect to the input.

The per-sample loss function is denoted as $L(f(\boldsymbol{x},\boldsymbol{\theta}),y)$, but instead of working on the per-sample basis, the loss function gets aggregated (typically averaged) over a training dataset as follows:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\ \hat{p}_{data}} L(f(\boldsymbol{x};\boldsymbol{\theta}),y)$$

Examples of the commonly used loss functions are:

$$\text{Mean squared error } MSE(\boldsymbol{y},\hat{\boldsymbol{y}}) = \frac{1}{m}\sum_{i=1}^{m}(\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i)$$

$$\text{Kullback-Leibler divergence } D_{KL}(p||q) = \sum_{x\in\mathcal{X}} p(x)log(\frac{p(x)}{q(x)})$$

$$\text{Cross-Entropy } H(p,q) = \mathbb{E}_p[log\ q]$$

where p, q are compared distributions (typically data distribution and model distribution in the context of neural networks). The $\mathcal{X}$ in the definition of Kullback-Leibler divergence corresponds to a probability space over which the distributions p and q are defined.

Because the loss function should be minimized, a natural way to do it is to adjust the parameters $\boldsymbol{\theta}$ in the direction opposite to the direction of the gradient. The direction of the gradient corresponds to a direction of the steepest ascent, therefore the opposite direction corresponds to the direction of steepest descent—so-called gradient descent. This leads to the following formula:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{1.5}$$

Notice that there is an additional parameter α which is called learning rate and it specifies the length of the step in the given direction. This update is performed iteratively and it stops once a local minimum is reached. Beware of the word "local", this specifically means that the algorithm is not guaranteed to reach the global minimum of the loss function.

It is a common practice to adaptively modify the learning rate during training (typically decrease it) in order to gain faster convergence and better training stability. There are several schemes for adaptive learning rate decay, e.g. exponential learning rate decay [25].

Generally, there are three variants [26] of gradient descent that differ in a way of computing $J(\boldsymbol{\theta})$:

1. Regular (or vanilla) Gradient Descent—Use all the data to compute it.

2. Stochastic Gradient Descent—Estimate the expectation inside the definition of $J(\boldsymbol{\theta})$ by a single example that is randomly sampled from the training data.

3. Minibatch Stochastic Gradient Descent—which stands between the above two variants and estimates the expectation using a batch of m random independent examples from the training data.

A common way to compute the gradient in an efficient manner is by using a backpropagation algorithm. Roughly speaking, backpropagation works by computing the derivatives (needed for the gradient) by propagating information through a network in a backward direction. A more detailed description will be given in the rest of this section.

From a high-level perspective, the learning procedure can then be described using two phases: *forward pass* and *backward pass*. During forward pass, the output of the network given the input data is computed allowing to compute the loss function (parameters are kept fixed while forward pass continues). Then comes the backward pass which is responsible for computing the gradient of the loss function by going from the last layer (output) back towards the first layer (input). Derivatives with respect to each weight are calculated by the chain rule, but this approach is more effective than a naive chain rule because redundant calculations are skipped thanks to the fact that the algorithm goes backward and derivatives at each level (layer) only depend on the derivatives from previous, already computed layers. To illustrate the whole process, it is useful to introduce the notation depicted in the Figure 1.8

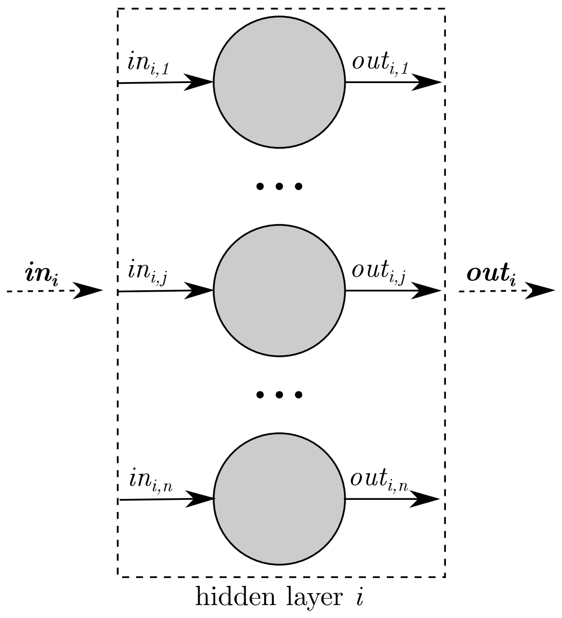


Figure 1.8: Notation for representation of neuron input and output.

In Figure 1.8 the jth neuron in ith layer has its weighted input denoted as $in_{i,j}$ and its output is denoted by $out_{i,j}$. A collection of these inputs and outputs for the whole layer are then denoted by $\boldsymbol{in_i}$ and $\boldsymbol{out_j}$ respectively. The relation between inputs and outputs is following: $\boldsymbol{out_i} = activation(\boldsymbol{in_i})$. The weighted inputs are $\boldsymbol{in_i} = \boldsymbol{W}^{(i)}\boldsymbol{x}$ where the $\boldsymbol{x}$ is either the output of the previous layer $\boldsymbol{out_{i-1}}$ or input to the whole network (this depends on value of i) and $\boldsymbol{W}^{(i)}$ are the weights between layers i and $i-1$ which can be specified by matrix:

$$\boldsymbol{W}^{(i)} = \begin{bmatrix} w^i_{1,1} & \dots & w^i_{1,n} \\ \dots & \dots & \dots \\ w^i_{n,1} & \dots & w^i_{n,n} \end{bmatrix}$$

where w^i_{jk} is a weight between kth node in the $i-1$th layer and jth node in the ith layer as depicted in Figure 1.9.

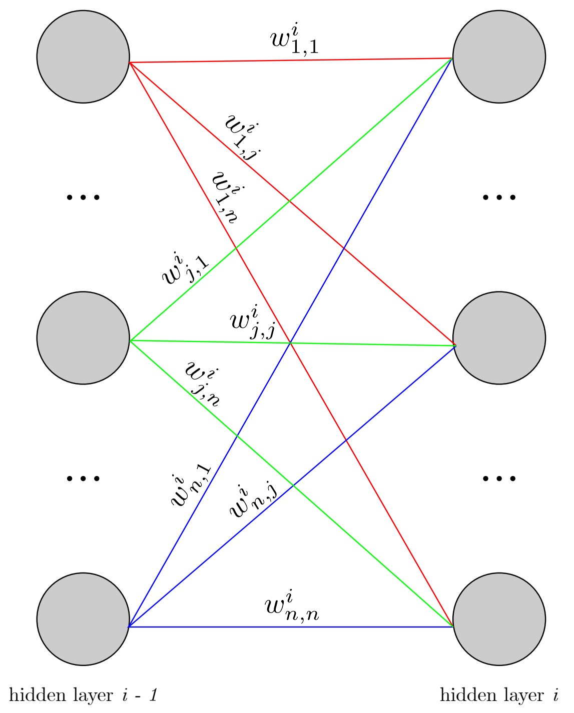


Figure 1.9: Notation for representation of neuron weights.

After introducing the notation for a concise representation of the network computation and giving a high-level overview of the backpropagation, it is now the right place to provide a more in-depth explanation. Keep in mind that the goal of backpropagation is to compute a gradient of the loss function and this can be achieved by computing partial derivatives of the loss with respect to the network's inputs.

First notice, that by using an equation 1.4 the forward pass, i.e. the computation of the output of the network could be written in the following way (assuming that the bias term is hidden inside the weights $\boldsymbol{W}$):

$$\hat{\boldsymbol{y}} = f^{(m)}(\boldsymbol{W}^{(m)} f^{(m-1)}(\boldsymbol{W}^{(m-1)} \dots f^{(2)}\boldsymbol{W}^{(2)}(f^{(1)}(\boldsymbol{W}^{(1)}\boldsymbol{x}))))$$

This calculation of $\hat{\boldsymbol{y}}$ is done by evaluating the network from left to right (i.e. from layer 1 all the way to the last layer m).

Furthermore, assume that the loss function is denoted as $L(\boldsymbol{y}, \hat{\boldsymbol{y}})$, then the derivative of the loss function with respect to the inputs can then be obtained by using the chain rule as follows:

$$\frac{\partial L}{\partial \boldsymbol{x}} = \frac{\partial L}{\partial \boldsymbol{out_m}} \cdot \frac{\partial \boldsymbol{out_m}}{\partial \boldsymbol{in_m}} \cdot \frac{\boldsymbol{in_m}}{\boldsymbol{out_{m-1}}} \cdot \dots \cdot \frac{\boldsymbol{out_1}}{\boldsymbol{in_1}} \cdot \frac{\boldsymbol{in_1}}{\boldsymbol{x}}$$

Rewriting this expression back into the original notation yields the following formula:

$$\frac{\partial L}{\partial \boldsymbol{x}} = \frac{\partial L}{\partial \boldsymbol{out_m}} \cdot (f^{(m)})' \cdot \boldsymbol{W}^{(m)} \cdot (f^{(m-1)})' \cdot \boldsymbol{W}^{(m-1)} \cdot \dots \cdot (f^{(1)})' \cdot \boldsymbol{W}^{(1)}$$

The backward pass then corresponds to incremental evaluation of this expression from the left to right computing the gradient at each layer—actually, there is one more multiplication needed in order to obtain gradient with respect to the weights.

Denoting $\boldsymbol{\delta^i} = \frac{\partial L}{\partial \boldsymbol{out_m}} (f^{(m)})' \cdot \boldsymbol{W^{(m)}} \cdot (f^{(m-1)})' \cdot \boldsymbol{W^{(m-1)}} \cdot \dots \cdot (f^{(i+1)})' \cdot \boldsymbol{W^{(i+1)}} \cdot (f^{(i)})'$ allows the gradient of the ith layer's weights to be computed as:

$$\nabla_{\boldsymbol{W^i}} L = \boldsymbol{\delta^i} (\boldsymbol{out_{i-1}})^T$$

and $\boldsymbol{\delta^i}$ can be calculated recursively as:

$$\boldsymbol{\delta^{i-1}} = (f^{(i-1)})' \cdot (\boldsymbol{W^i})^T \cdot \boldsymbol{\delta^i}$$

Hereby leading towards a more efficient computation that naively computing each $\boldsymbol{\delta^i}$ from scratch without using the knowledge obtained during computation of $\boldsymbol{\delta^{i-1}}$.

To conclude, it should be emphasized that the backpropagation algorithm is not guaranteed to find a global optimum, because it could get stuck in a local optimum as was already mentioned at the beginning of this section. However, for many practical problems, this is not a major issue [27]. There exist several variants and improvements of the simple SGD algorithm (the one using the update from equation 1.5) that were proposed with an intention of obtaining an algorithm with better properties (most importantly, better stability and/or convergence speed), for example, Adam [28], RMSProp [29], etc.[3]

1.1.6 Regularization

As usual for machine learning methods, neural networks are also prone to overfitting and there exist several regularization techniques that try to combat overfitting.

Frequently used regularization techniques include the following:

- L1, L2 regularization[4].
- Dataset augmentation
- Dropout

[3] Other variations of the update step can be found in [30].

[4] These techniques are frequently used in classical machine learning scenarios so their description is omitted and can be found in [17].

L1, L2 regularizations are a sort of classical techniques that are also frequently used in standard machine learning (even when no deep neural networks are used) where they are commonly used in the context of ridge regression and lasso regression. Roughly speaking, these regularizations works by adding a model complexity penalty term to a loss function and the difference between L1 and L2 is in the way what the penalty looks like. An exact definition and more details about these regularizations can be found in [31]. Note that in the context of deep learning, these regularizations are commonly called weight decay.

Dataset augmentation combats overfitting by enlarging the training dataset with slightly modified versions of the original examples. In the domain of image processing, this means performing some processing/modification of the images, for example, horizontal flipping, rotations, translations, etc. Some image augmentations are shown in Figures 1.10 and 1.11.

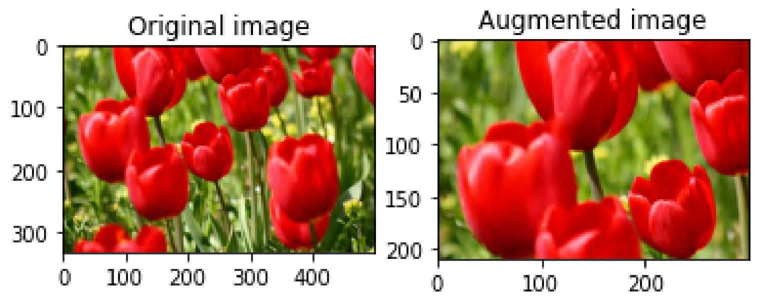


Figure 1.10: Illustration of random cropping augmentation. This image was taken from: [32].

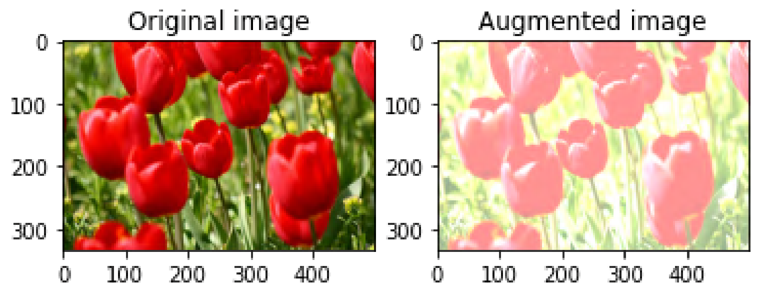


Figure 1.11: Illustration of random brightness augmentation. This image was taken from: [32].

Figure 1.10 shows how an image can be augmented by taking a random crop. Figure 1.11 then shows how an image can be augmented by randomly changing its brightness. In both figures, the image on the left shows the original image and the image on the right corresponds to an output image.

Dropout [33] is another frequently used regularization technique and it can be applied to a layer of neurons. When the dropout is applied, each neuron is dropped independently with a probability of p. It is usually implemented in such a way that for the rest of the network, these dropped neurons seem to have a value of 0.

1.2 Deep neural networks

Deep neural networks take the idea of the "classical" artificial neural networks even further, by stacking several layers onto each other in the hope of a more powerful model. However, this comes at some price, because several new problems arise when multiple layers are being stacked (for example, vanishing and exploding gradients [34] [35]).

Deep learning methods have various real-world applications, like image processing, speech recognition, creation of transcripts, etc. and almost all of the state-of-the-art models in all these applications are using very deep[5] neural networks.

1.2.1 Convolutional networks

Convolutional neural networks (CNNs) are a kind of neural network that is useful for dealing with data that has some underlying structure. As their name says, convolutional neural networks are using an operation called *convolution*.

Assume two functions x and w then the convolution $x * w$ is defined as:

$$(x * w)(t) = \int x(a)w(t-a)da$$

where x is the *input* (a function of a single parameter), t is the so-called time index, w is a weighting function ($t - a$ represents the age of the input) and the output is often referred to as a *feature map*. The weighting function is frequently called *kernel* or filter. Convolution is commutative, therefore it can be written as:

$$(w * x)(t) = \int x(t-a)w(a)da$$

When t is discrete then this operation is called *discrete convolution* and it can be written in the following way:

$$(w * x)(t) = \sum_{a=-\infty}^{\infty} x(t-a)w(a)$$

Discrete convolution can be generalized into two dimensions as follows:

$$(\boldsymbol{K} * \boldsymbol{I})_{i,j} = \sum_m \sum_n \boldsymbol{I}_{i-m,j-n} \boldsymbol{K}_{m,n}$$

where $\boldsymbol{K}$ is the two-dimensional kernel (with size $m \times n$) and $\boldsymbol{I}$ is the two-dimensional input.

However, deep learning libraries often implement a slightly different operation [31]—*cross-correlation*—and call it convolution. The cross-correlation is defined in the following way:

$$(\boldsymbol{K} \star \boldsymbol{I})_{i,j} = \sum_m \sum_n \boldsymbol{I}_{i+m,j+n} \boldsymbol{K}_{m,n} \tag{1.6}$$

In this thesis, cross-correlation is referred to as a convolution as it is often the case in the context of convolutional neural networks.

[5]Meaning they contain a large number of layers.

Convolutions help to achieve the following three properties[6]: *parameter sharing*, *local interactions* and *shift-invariance* which is something that is not achievable by the fully connected layers.

Convolutional neural networks are frequently used for processing images, if that is the case, the input image has C (input) channels and the convolutional layer is parametrized by a kernel $\boldsymbol{K}$ of a total size $H \times W \times C \times F$ where F is the number of output channels. Often it is useful to consider only every Sth pixel of the input[7] which can be achieved by using a convolution with a stride S[8]. Another way of thinking about stride is the number of "cells" or pixels for which the kernel or sliding window is moved over the input. The default value for stride is $S = 1$. Furthermore, the equation (1.6) can be written in an even more general way as:

$$(\boldsymbol{K} \star \boldsymbol{I})_{i,j,o} = \sum_m \sum_n \sum_c \boldsymbol{I}_{i \cdot S+m, j \cdot S+n, c} \boldsymbol{K}_{m,n,c,o}$$

where c is the number of input channels and o is the number of output channels.

When applying a convolution with a filter to a given input image, there is a problem of handling the convolution operation at the borders of the images. These problems are solved by adding padding, and the two, most commonly used padding schemas are:

- *Valid:* does not pad the data, this causes the spatial dimension to reduce after performing the convolution.
- *Same:* pads the input with 0 values in such a way that the spatial resolution after performing convolution is the same as before performing the convolution.

Valid padding and same padding are illustrated in Figures 1.12 and 1.13 respectively.

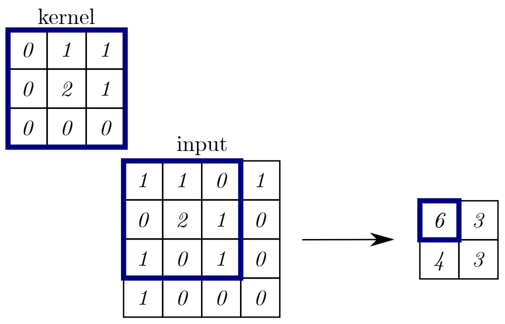


Figure 1.12: *Valid* padding illustration.

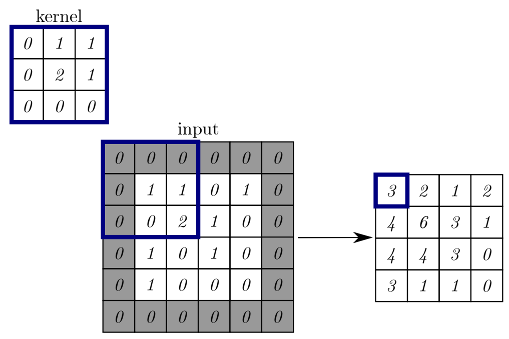


Figure 1.13: *Same* padding illustration.

Figure 1.12 shows usage of a valid padding, with a kernel of size 3×3 over input of size 5×5 with stride $S = 1$. Notice that the spatial resolution of the output is smaller than that of the input which is not a case for the same padding from Figure 1.13 which preserves spatial dimensions of the output.

[6]These properties and the main motivation for convolutional neural networks are nicely described in [31].

[7]There are various reasons why it is useful, e.g. performance/memory improvement.

[8]Using stride $S > 1$ causes the spatial resolution of the output to be smaller than that of the input.

Another operation that is frequently used in convolutional neural networks is called pooling and it replaces the output of a previous layer with summary statistics of the neighboring cells or units. Especially this means, that it performs downsampling of its input to produce an output with smaller spatial dimensions. The two most popular pooling layers/operations are max pooling and average pooling where the cells are replaced by their maximum and average respectively. Max pooling and average pooling with a pool size $m = 2$ and stride $S = 2$ are shown in Figures 1.14 and 1.15 respectively.

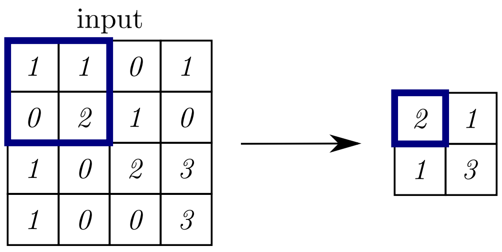


Figure 1.14: Max pooling operation.

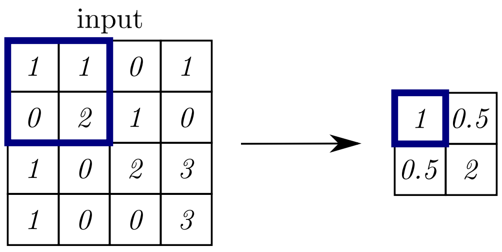


Figure 1.15: Average pooling operation.

Figures 1.14 and 1.15 illustrates a 2D version of a pooling operation, but it should be mentioned that the pooling operation can also be generalized into different dimensionalities than 2D (1D or 3D). There is yet another frequently used operation—a special case of pooling—called global pooling [36] which downsamples the whole input into a single value.

An example of CNN architecture is shown in Figure 1.16.

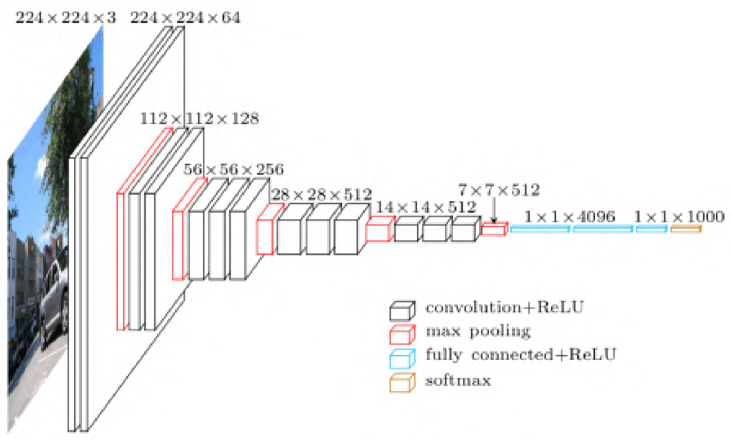


Figure 1.16: The architecture of a typical CNN (in this case VGG 16). This image was taken from [37].

Figure 1.16 illustrates the architecture of VGG 16 network [38] which is commonly used for image recognition or as a feature extractor. On the left of this figure, there is an input image with dimensions 224 × 224 × 3 which is used as the input to the network. The network itself starts with a series of convolutional layers interleaved with max-pooling layers. When the input is passed through these layers, its spatial dimensions are decreased while the number of channels increases. The network ends with a series of fully connected layers with a ReLU activation function followed by a softmax (classification) layer.

1.2.2 Feature pyramid networks

Feature pyramid networks [39] or simply feature pyramids is an architecture that allows extracting features at various levels of detail (i.e. resulting in multi-scale feature maps). Motivation is that using several feature maps with varying levels of detail should result in better network performance (this property will be later used for the final encoder model in Section 3.4). These networks are frequently used in object detection, but a similar principle can be applied for feature extraction in a more general sense. The architecture of a feature pyramid network is shown in Figure 1.17.

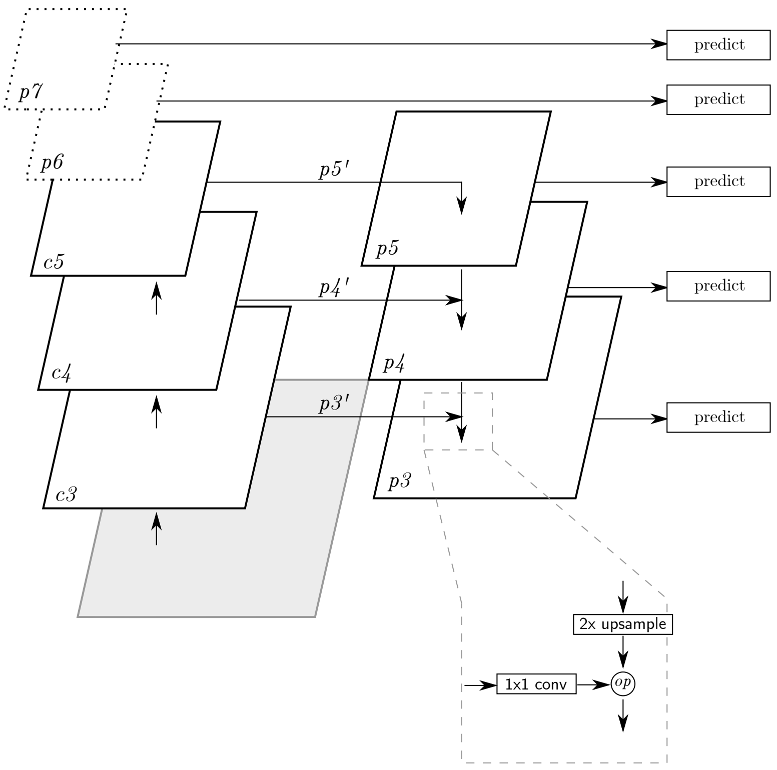


Figure 1.17: The architecture of a typical feature pyramid network.

As illustrated in Figure 1.17, the feature pyramid network take an image on its input, perform several downsampling convolutions (their outputs are denoted as *c3*, *c4*, *c5* in the figure) during a bottom-up pathway. Then the *c3*, c4, and *c5* are passed through a 1×1 convolution (in order to change the number of channels) to obtain *p3'*, *p4'*, *p5'* respectively. During a top-down pathway, *p5* is taken, upsampled, and then "merged" with *p4'* to obtain *p4*. The merge is typically performed using addition or concatenation. This process continues all the way to the bottom where *p3* is produced. *"The top-down pathway hallucinates higher resolution features by performing a 2x upsampling of spatially coarser but otherwise semantically stronger feature maps."* [39]. Later, these features are enhanced with features from the bottom-up pathway via merging (the merge connection is also called lateral connection.

1.3 Generative adversarial networks

There are several widely used generative models that are frequently used for image generation. The two that are most frequently used and that bring the best results are *generative adversarial networks* (GAN) [40] and *variational autoencoders* [41]. This section is concerned with a description of GANs, while variational autoencoders will be described later, in sub-section 1.4.1.

GAN is an architecture or a model that consists of two components — generator G and discriminator D — that are competing (in an adversarial manner) with each other in a zero-sum game. The main purpose of GANs is to use them as a generative model, where for a given training dataset, the GAN generates new data with the same statistics (from the same distribution) as the training dataset. GANs are frequently used to generate images and this is a scenario that is assumed in the rest of this section.

The generator component is given a "seed" (called latent vector $\boldsymbol{z}$) and then it syntheses or generates an image from it, i.e. given $\boldsymbol{z} \sim P(\boldsymbol{z})$, generator $G(\boldsymbol{z}; \boldsymbol{\theta_g})$ generates data $\boldsymbol{x}$. The discriminator $D(\boldsymbol{x}; \boldsymbol{\theta_d})$ takes an image $\boldsymbol{x}$ and predicts the probability of whether this image is real or fake (generated by the generator). The generator is trained indirectly through the discriminator – it tries to fool the discriminator – this fact enables the model to be learned in an unsupervised manner (training the discriminator is easy because it is known which image was generated by the generator and which is taken from the input dataset).

The GAN architecture is depicted in Figure 1.18.

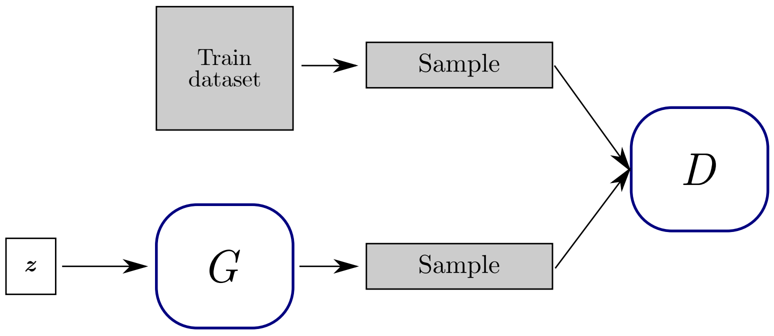


Figure 1.18: Illustration of GAN architecture.

The original loss of the GAN is the following:

$$L(D, G) = \mathbb{E}_{\boldsymbol{x} \sim P_{data}}[logD(\boldsymbol{x}, \boldsymbol{\theta_d})] + \mathbb{E}_{\boldsymbol{z} \sim P(\boldsymbol{z})}[log(1 - D(G(\boldsymbol{z}, \boldsymbol{\theta_g}), \boldsymbol{\theta_d})]$$

And it is optimized by playing the following game:

$$\min_G \max_D L(D, G)$$

Generator and discriminator are being trained alternately, the generator by:

$$\arg\min_{\boldsymbol{\theta_g}} \mathbb{E}_{\boldsymbol{z} \sim P(\boldsymbol{z})}[log(1 - D(G(\boldsymbol{z}))]$$

i.e., trying to fool the discriminator to think that the generated data is real. And the discriminator by:

$$\arg\max_{\boldsymbol{\theta_d}} \mathbb{E}_{\boldsymbol{x} \sim P_{data}}[log(D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim P(\boldsymbol{z})}[log(1 - D(G(\boldsymbol{z})))]$$

These loss functions are the ones used in the original GAN [40] and since then, there were quite a few variants of loss functions for training GANs, for example, the loss used in Wasserstein GAN [42].

Training of the GANs is very fragile and there are several issues with training of the GANs (mode collapse, unstable training, divergence, etc.) all of them are described in Section 1.3.1. Moreover, when the GANs were introduced, there were several limitations to them, for example, initial GAN models were not able to generate images with higher resolutions (they were typically generating images with a very small resolution, something like 16×16 or 32×32 and the overall quality of the generated images was not very impressive).

1.3.1 Problems of GAN models

Training GAN models is not easy and there are various problems (although some of them are either solved or at least mitigated in the state-of-the-art models) that are happening frequently. One of these problems is training instability which stems from the nature of training GAN. For example, when the discriminator is much better than the generator, it prevents the generator from learning anything because it always detects that the image is fake. The training instability is further magnified by the limited GPU memory which typically leads to the usage of very small batch sizes (using batch sizes of 8 or 16 is not uncommon for large GAN models). The training instability frequently causes the training to diverge in such a way that the model generates only noise, black color, or some other kind of a mess. There were several attempts to improve GAN convergence to the date [43] [44].

A big challenge of GAN models is a generation of high-quality images because when the resolution is large, the discriminator can easily spot that the image is fake based on some very detailed flaws in the image.

Another problem related to GAN models is called mode collapse [45] which means that the generator does not generate outputs that are diverse enough. This typically means that the generator output space is very little, often containing only several same or highly similar images. The result of a mode collapsed generator is shown in Figures 1.19 and 1.20.

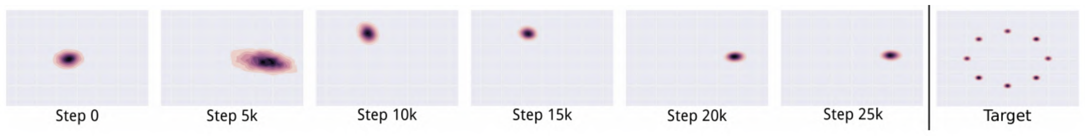


Figure 1.19: Illustration of progressive growing, together with a few samples generated by Progressive GAN. This image was taken from [45].

Figure 1.19 considers a case of a GAN trained on a dataset with a 2D mixture of Gaussians and it shows heatmaps of generator distributions (at different training steps) and the final column shows the data distribution. See that the generator simply rotates over the modes of the data distribution, it never converges to a fixed distribution and moreover, it assigns a significant probability to only a single sample.

Figure 1.20 presents a typical mode collapse scenario where the GAN is able to generate only a single sample.

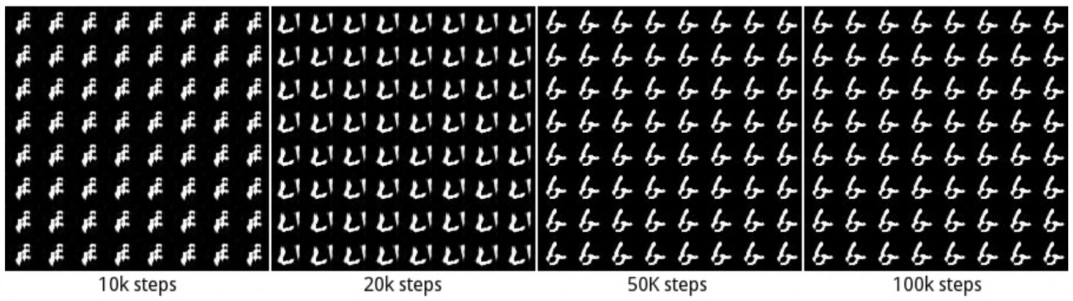


Figure 1.20: Outputs of a GAN trained on MNIST dataset that generates only a single character. This image was taken from [45].

1.3.2 Vector manipulation

An interesting property of GANs is that that they allow—at least to some extent—to perform meaningful linear interpolations and/or vector arithmetics in the latent space. These operations, together with a more general task of latent space exploration will be referred to as *vector manipulations*. Ideally, these manipulations of a latent vector should lead to a meaningful change of the generated image. For example, assume a latent vector $\boldsymbol{z}$ that generates image $\boldsymbol{x}$ which depicts a female with brown hair, and suppose that we would like to find an image $\boldsymbol{x'}$ that would depict the same person but now with blonde hair. This problem could be formulated as a finding of a $\dot{\boldsymbol{z}} = f(\boldsymbol{z})$ such that $G(\dot{\boldsymbol{z}}) = \boldsymbol{x'}$ where f is a function representing latent vector manipulation.

Vector interpolation

Suppose two latent vectors $\boldsymbol{z_1}$, $\boldsymbol{z_2}$ then their *linear interpolation* is defined as $\tilde{\boldsymbol{z}} = (1-t)\boldsymbol{z_1} + t\boldsymbol{z_2}$ where $t \in [0, 1]$. An example of latent interpolation is shown in Figure 1.21

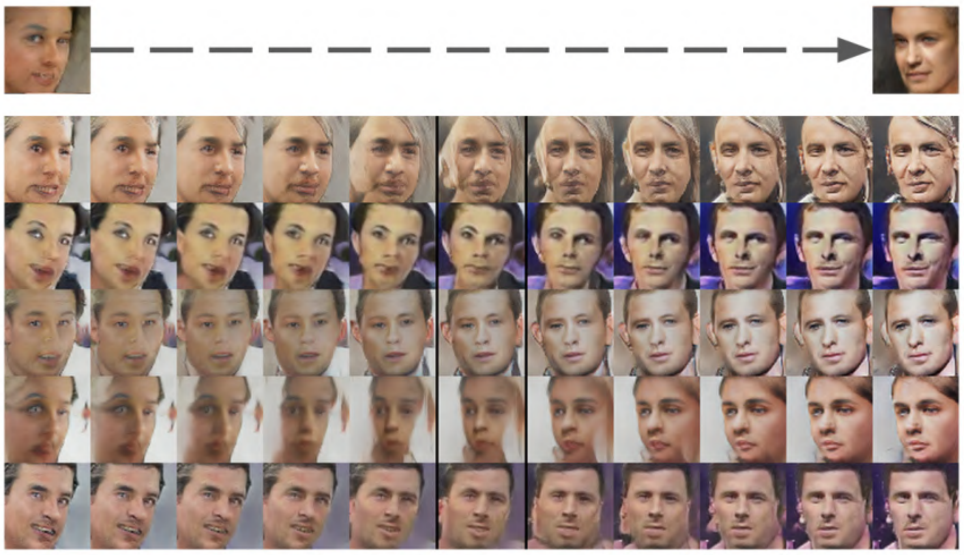


Figure 1.21: Illustration of latent vector interpolations and their effect on the generated images. This image was taken from [46].

Latent space exploration

Sometimes it might be interesting to observe the neighborhood of a latent vector within the latent space. This operation could be called *latent space exploration* and it could be performed by slightly moving a latent vector $\boldsymbol{z}$ in a given direction to produce a new latent vector $\hat{\boldsymbol{z}}$. As an example of usage of this operation, consider an image of a person (e.g., the suspect in the context of police lineup construction), then the goal of this operation might be to find a new latent vector that will generate a similar person (e.g., one filler) but with a slightly different face.

Vector arithmetic

Vector arithmetic then typically refer to queries of type:

$$\textit{smiling woman} - \textit{neutral woman} + \textit{neutral man} = \textit{smiling man}$$

An example of vector arithmetics is shown in Figure 1.22.

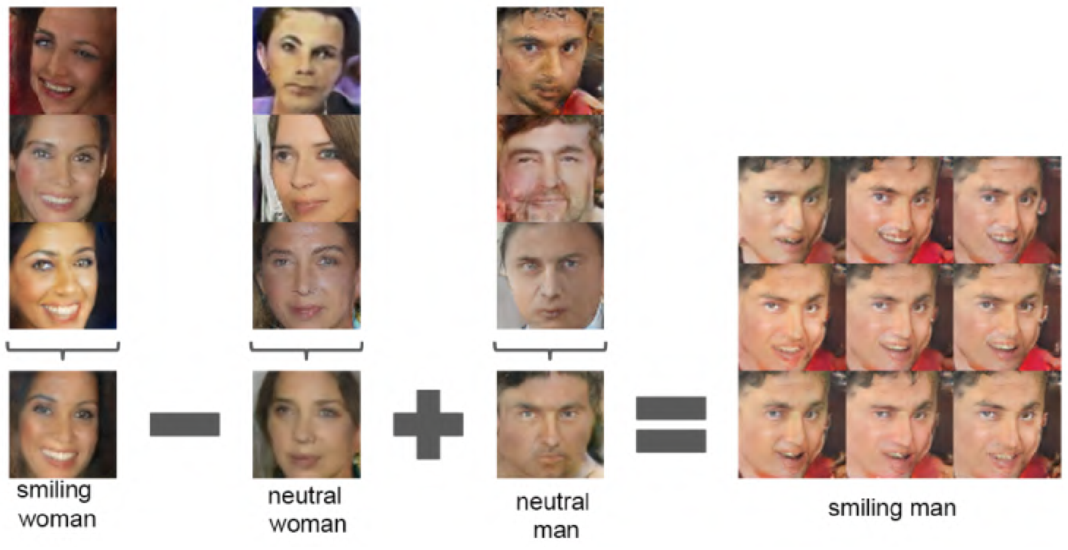


Figure 1.22: Illustration of latent vector arithmetics and their effect on the generated images. This image was taken from [46].

The following example scenario presents the usage of vector arithmetic that is more relevant for the topic of this thesis. Assume that there is an image of suspect x and that the suspect has some rare combination of facial features, e.g. a face tattoo together with green hair. In that case, it would be extremely difficult to find appropriate fillers but the vector arithmetic could help there. Assume, that there are images of people having green hair and images of people having a face tattoo (but not both). Then a query similar to the following could be constructed:

$$\textit{green hair} + \textit{face tattoo} = \textit{green hair and face tattoo}$$

hopefully allowing to generate an appropriate filler image.

Problems of vector manipulation

Although the vector arithmetic queries above may seem intuitive and the results presented in this section so far may seem impressive, it is not always easy to produce results of similar quality. Generally, there two major issues related to vector manipulations: feature entanglement and features that are not general.

Feature entangelement means that when a latent vector is changed in some direction, more than only one attribute in question changes. Consider the artificial example from the first paragraph of this section where to goal was to generate an image of the blonde hair woman that looks otherwise the same as the brown-haired woman. If the features were entangled, then there would be more attributes apart from the hair color that would have changed.

Non-general features refer to a problem that the learned features are not universal enough and they are not generally applicable. For example, someone might find a direction that seems to affect the age of the generated person, but it may easily happen, then given a different latent vector, this direction will have a very different meaning than a change of the age.

1.3.3 Progressive GAN

Progressive GAN [47] can be considered as a milestone in GAN models because was one of the first models that allowed to generate high-resolution images and therefore overcame one of the big issues or weakness of early GAN models which were only capable of generating low-resolution images.

The key concepts and characteristics of the Progressive GAN are following:

- During training, the number of layers is increased incrementally.
- Both generator and discriminator have similar architecture, they start from a resolution of 4×4 and continue all the way up to the 1024×1024.
- During training, new layers are added to both generator and discriminator.
- Incremental increasing of the layers allows the models to first discover and learn the large-scale (coarse) structure of the image distribution and then continually move the attention to lower-scale, finer details.
- When a new layer is added, the "fade-in" phase is performed so that the already trained layers are not "shocked" by the change.

The essence of progressive learning is the fact that the scales are learned incrementally instead of learning all of them simultaneously which seems to bring several benefits. One of these benefits is the fact that the generation of smaller images is significantly more stable because there is less class information and fewer modes [48]. Another benefit is reduced training time because most of the training iterations are done at lower spatial resolutions.

The progressive growing together with several images generated by the Progressive GAN are shown in Figure 1.23.

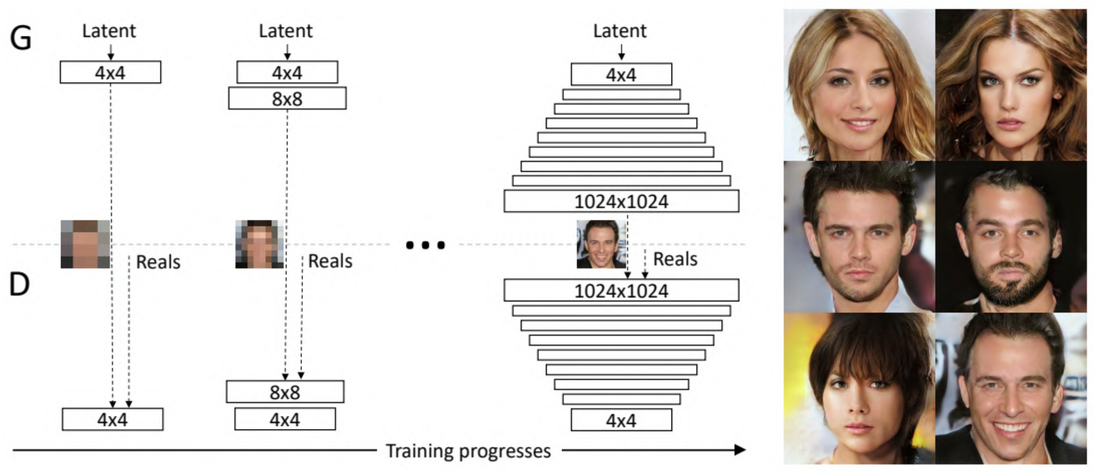


Figure 1.23: Illustration of progressive growing, together with a few samples generated by Progressive GAN. This image was taken from [47].

The left side of Figure 1.23 illustrates the progressive growing of the generator and discriminator networks. The $N \times N$ boxes shown in the picture correspond to convolutional layers operating on images with a spatial resolution of $N \times N$ pixels. Notice that both networks start with a spatial resolution of 4×4 pixels and they keep increasing the spatial resolution (by adding new layers to both networks) as the training process progresses until ending up with a final resolution of 1024×1024 pixels. Note that all the existing layers remain trainable throughout the whole process. The right side of this figure shows several sample images that were generated using Progressive GAN.

Figure 1.24 shows several images that illustrate how the quality of GAN models has evolved.

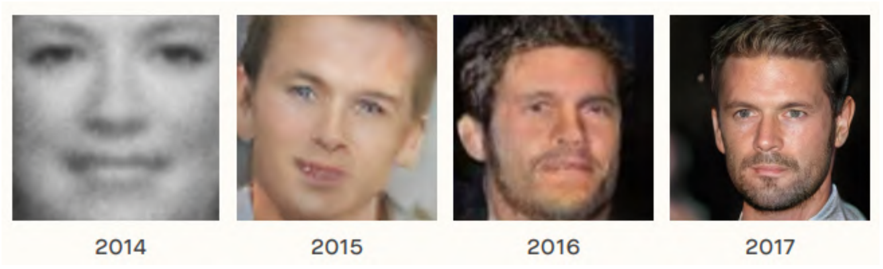


Figure 1.24: Evolution of quality of GAN-generated images in time. This image was taken from [49].

In Figure 1.24 the right-most image was generated by the Progressive GAN and the remaining images were generated by other, older models. Notice the huge difference in the overall quality of the images and also in the resolution of the individual images.

1.3.4 StyleGAN

StyleGAN [50] is built on Progressive GAN and it introduces an alternative generator architecture for GANs which allows for automatically learned, unsupervised separation of high-level attributes (pose, identity) and stochastic variation (minor facial attributes, for example, freckles) in the generated images. Additionally, StyleGAN also enables scale-specific control of the synthesis. Another, very important enhancement introduced by this architecture is that it leads to better interpolation properties and feature disentanglement.

Better feature disentanglement is achieved by introducing the so-called *mapping network* through which the latent vector $\boldsymbol{z}$ is passed before feeding it through the generator itself. The mapping network consists of several dense layers (authors use 8) and it represents a non-linear function $f : \mathbb{Z} \to \mathbb{W}$. Vectors $\boldsymbol{w} \in \mathbb{W}$ are often referred to as the *style vectors*. Other details are not that important for this thesis and if needed, they could be found in the original article [50].

The style vector $\boldsymbol{w}$ is transformed and incorporated into every generator's block using a so-called *adaptive instance normalization* (denoted as *AdaIN*). Adaptive instance normalization performs scaling or normalization of the feature map to a standard normal distribution $\mathbb{N}(0, 1)$ and then adds the style vector as a bias term. It should be mentioned that the addition of the mapping network resulted in a slight change in the naming convention because the actual generator is now frequently called a *synthesis network* and the term generator now refers to a whole generator as such, including mapping network and synthesis network.

AdaIN operation is defined as follows:

$$AdaIN(x_i, y) = y_{s,i}\frac{x_i - \mu x_i}{\sigma x_i} + y_{b,i}$$

where $y_i = (y_{s_i}, y_{b,i}) = f(\boldsymbol{w})$ and f represents a learned affine transformation (dense layer with a linear activation) and y_i are the styles.

StyleGAN architecture is shown in Figure 1.25.

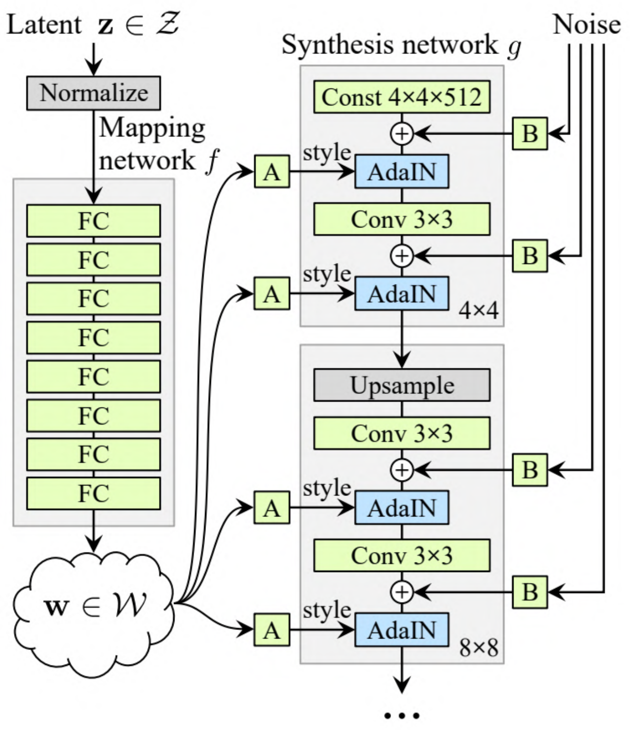


Figure 1.25: The architecture of the StyleGAN network. [50].

Figure 1.25 presents an architecture of the StyleGAN network, more specifically, of the StyleGAN's generator, because the discriminator is fairly similar to the standard discriminator. Notice the mapping network in the left of the figure which takes the latent vector and processes it to produce vector $\boldsymbol{w} \in \mathcal{W}$. This intermediate vector $\boldsymbol{w}$ then controls the generator through the *AdaIN* operation [50] at each convolutional layer. The "A" boxes correspond to learned affine transformation which is just some dense layer with linear activation as it was already described above. Furthermore, the box B applies learned per-channel scaling factors to the noise inputs [50]. The mapping network consists of 8 dense (fully connected) layers and the synthesis network which is shown in the right part of the figure consists of 9 blocks (one block for each of the resolutions $4 \times 4, \ldots, 1024 \times 1024$ and each block consists of 2 layers (so each block gets 2 style vectors).

Issues of the StyleGAN

Even though StyleGAN produces high-resolution images of remarkable quality, after analyzing its outputs thoroughly, the authors of StyleGAN have found several issues in the generated images.

One of the issues with images generated by the StyleGAN models is that they often contain the so-called water droplet-like artifacts. These artifacts are illustrated in Figure 1.26. It was hypothesized that these artifacts are caused by the usage of *AdaIN* operation and it was shown that removal of this operation lets the droplet artifacts disappear.

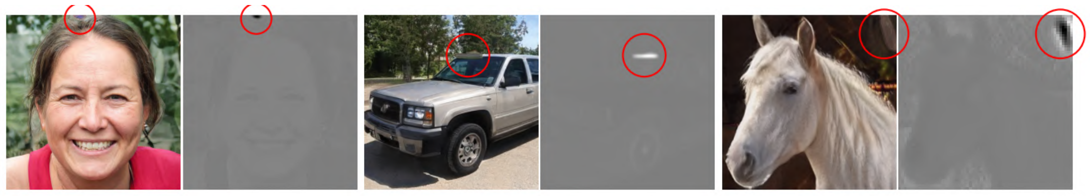


Figure 1.26: Presence of a water-droplet effect in the images generated by StyleGAN. This image was taken from [51].

Another issue is the presence of "phase" artifacts in the generated images as illustrated in Figure 1.27.

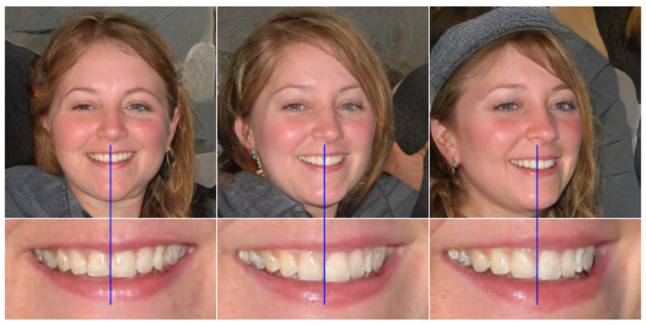


Figure 1.27: Presence of phase artifacts in the images generated by StyleGAN. This image was taken from [51].

In Figure 1.27, notice the position of teeth in the image, which remains the same even though the pose of the face changes.

Authors of StyleGAN believe that these artifacts are caused by progressive growing [51] because it causes the generator to have a strong location preference for details (at which that detail is placed, even when the rest of the image moves).

1.3.5 StyleGAN2

The issues that were described in subsection 1.3.4 has led to a new architecture, called StyleGAN2[9] [51] which is another GAN model that builds on StyleGAN and it tries to solve these StyleGAN issues.

The important difference from a StyleGAN are removal of a progressive growing in a hope of reducing the phase artifacts and removal of the *adaIN* in order to get rid of the water droplet effect. There were a few more improvements and they are all described in the rest of this section.

Overall, the improvements of the StyleGAN2 architecture are the following:

- Weight demodulation
- Lazy regularization
- Path length regularization
- No growing, new generator and discriminator architectures

Weight demodulation

StyleGAN2 uses a slightly different structure of a synthesis network than that which was used in the original StyleGAN. The change consists of replacing the *AdaIN* with a weight demodulation operation. It was shown that the *AdaIN* operation can be divided into two operations: modulation and normalization. Authors later show that the style block consists of modulation convolution and normalization and also that the modulation could be implemented in an alternative way by scaling convolution weights as follows:

$$w^{'}_{ijk} = s_i \cdot w_{ijk}$$

where w and $w^{'}$ are original and modulated weights, respectively, s_i is the scale corresponding to the ith input feature map. Similarly, the normalization could also be written differently by baking scaling into convolutional weights. The resulting equation looks in the following way:

$$w_{ijk}" = \frac{w'_{ijk}}{\sqrt{\sum_{i,j} w'^{2}_{ijk} + \epsilon}}$$

These adjustments result in the simplification of a style block so that the whole style block is baken into a single convolution layer whose weights are modified based on scale s. These changes are illustrated in Figure 1.28.

[9]Starting from the next Chapter 2, StyleGAN2 will be frequently referred to as just StyleGAN.

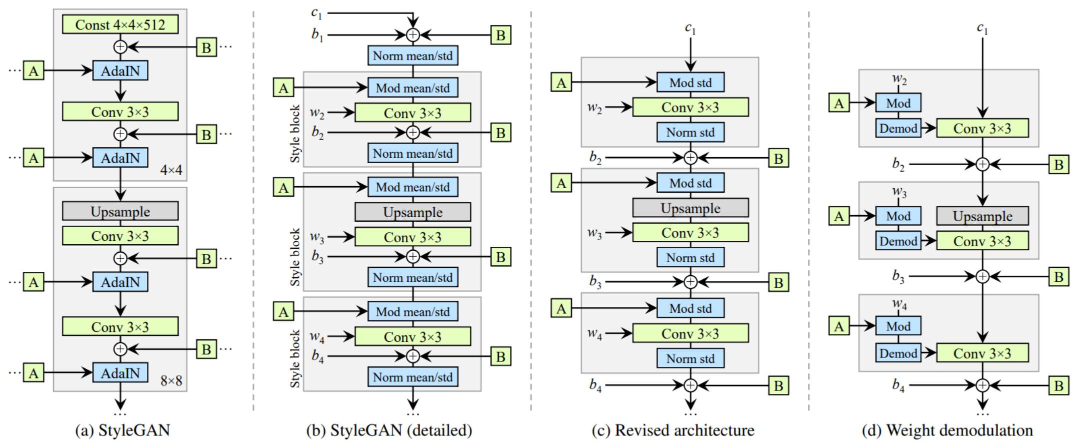


Figure 1.28: Evolution of the synthesis network's structure. This image was taken from [51].

Figure 1.28 shows how did the synthesis network evolve from the original StyleGAN to the new StyleGAN2. Part a) shows the original structure of the StyleGAN synthesis network, part b) shows the same structure, but zoomed in to greater detail (with *AdaIN* operation split into explicit normalization followed by modulation). Part c) removes some redundant operations, moves the addition of B outside the style block, and also modifies only the standard deviation per feature map. Finally, d) shows the new architecture that allows the *AdaIN* operation to be replaced by demodulation which could be applied to weights associated with each convolutional layer [51].

Lazy regularization

StyleGAN and other GAN models typically utilize regularization during training and they do it in such a way that the regularization term is included directly in the computation of the loss function. However, it turns out, that this is not necessary and that the regularization can be computed less frequently (every k iterations) without harming the performance thus significantly reducing the computational cost. In StyleGAN2, it was proposed to perform regularization only once every 16 mini-batches.

Path length regularization

Perceptual path length was introduced with the original StyleGAN [50] and it was originally intended as a metric for quantification of latent space entanglement, i.e. to measure how a small change in latent space affects the generated image. Perceptual length of an arbitrary interpolation path can be computed by dividing this path into n linear segments and then taking a limit (for $n \to \infty$) of a sum of perceptual differences over individual segments. The perceptual difference of a single segment is calculated by using image distance metric d. In practice, the subdivision into $n \to \infty$ segments is not feasible therefore the authors of StyleGAN do an approximation by subdividing the path into very small segments ($\epsilon = 10^{-4}$). The average PPL over in the latent space $\mathcal{Z}$, average over all possible endpoints is then defined as:

$$l_{\mathcal{Z}} = \mathbb{E}[\frac{1}{\epsilon^2} d(G(slerp(\boldsymbol{z_1}, \boldsymbol{z_2}; t)), G(slerp(\boldsymbol{z_1}, \boldsymbol{z_2}; t + \epsilon)))]$$

where G is the generator, $\boldsymbol{z_1}, \boldsymbol{z_2}$ are latent vectors, d is image distance metric and $t \sim U(0, 1)$. Note that *slerp* stands for spherical linear interpolation [52]. PPL for $\mathcal{W}$ space can be obtained similarly, by replacing $\boldsymbol{z_1}$, $\boldsymbol{z_2}$ by $mapping_{network}(z_1)$, $mapping_{network}(z_2)$ and replacing *slerp* with *lerp* [50].

Although it may not be clear at a first glance, the authors observe a correlation between perceived image quality and PPL [50]. According to [51] it seems that images with lower PPL have a better overall quality than those with high PPL. With this observation in mind, the PPL can be considered as one of the metrics that could be used for comparing the overall quality of the generated images[10].

Because of the abovementioned property, it is desired to generate low PPL images, but encouraging the generator to generate minimal PPL is not a good way to achieve this goal, because this would lead the generator to produce degenerated images. Instead, the same goal could be achieved by making the generator smoother, meaning that a small change within a latent space should result in a small change in generated image (actually this corresponds to minimizing PPL). More specifically, there was an intention to modify the generator in such a way that a fixed-size step in $\mathcal{W}$ results in a non-zero, fixed-magnitude change in the generated image [51] and this was achieved by introducing the following path length regularizer[11]:

$$\mathbb{E}_{\boldsymbol{w},\boldsymbol{y} \sim \mathcal{N}(0,\boldsymbol{I})}(\|\boldsymbol{J}_{\boldsymbol{w}}^T \boldsymbol{y}\|_2 - a)^2$$

Where $\boldsymbol{y}$ are random images with pixel intensities distributed according to $\mathcal{N}(0, \boldsymbol{I})$, $\boldsymbol{w} = f(\boldsymbol{z}), \boldsymbol{w} \in \mathcal{W}, \boldsymbol{z} \sim \mathcal{N}$, $\boldsymbol{J}_{\boldsymbol{w}^T} \boldsymbol{y} = \nabla_{\boldsymbol{w}}(g(\boldsymbol{w}) \cdot \boldsymbol{y})$ and a is exponential moving average of the lengths $\|\boldsymbol{J}_{\boldsymbol{w}}^T \boldsymbol{y}\|_2$.

The idea is, roughly speaking, that corresponding $\boldsymbol{w} \in \mathcal{W}$ gradients should have close to an equal length regardless of $\boldsymbol{w}$ or the direction, which indicates that the mapping from latent space to image space is not ill-conditioned. Detailed description could be found in the original paper [51].

No growing, new generator and discriminator architectures

StyleGAN was using progressive growing with a simple feedforward design in the generator and discriminator, but some of the recent studies were dedicated to finding a better architecture. As such, authors of the StyleGAN2 also decided to search for a better architecture that would still be able to produce high-resolution, high-quality images but now without a need of progressive growth.

The two most promising GAN architectures without progressive growing that the authors of StyleGAN2 evaluated were skip connections [55][56] and residual networks [57][58]. These new architectures are shown in Figure 1.29.

[10]There are several other frequently used metrics that are used for the same case, for example, FID [53] or LPIPS [54]

[11]This calculation increased computational time so that was the reason for introducing lazy regularization that was described in sub-section 1.3.5

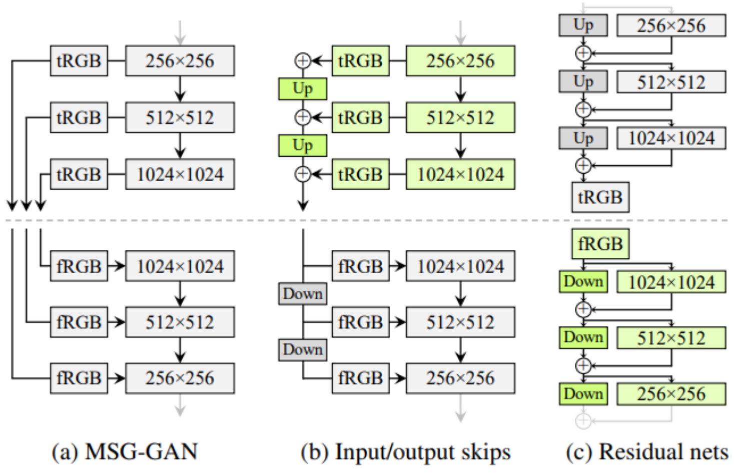


Figure 1.29: New architectures for generator and discriminator networks. This image was taken from [51].

Figure 1.29 shows new architectures for the generator and discriminator networks. The architectures in the first row correspond to the generator while the architectures in the second row correspond to the discriminator. The architectures that were eventually used in the final model are highlighted in green color. *Up* and *Down* correspond to bilinear upsampling and downsampling respectively. *tRGB* and *fRGB* denote conversions from high-dimensional per-pixel data to RGB data and vice versa. The part (a) of Figure 1.29 shows the architecture of MSG-GAN [56] which works so that it allows the discriminator to look not just at the final output (with the highest resolution) but also on the intermediate, lower resolution results. This is achieved by connecting the generator and discriminator by using several skip connections. In part (b), the generator's architecture is simplified by upsampling and summing the RGB contributions corresponding to different resolutions (similarly for the discriminator which performs downsampling instead of upsampling). Part (c) shows a further improved version of (b) by adding skip connections (addition of two paths).

According to the authors [51], "the skip connections in the generator drastically improve PPL and a residual discriminator is clearly beneficial for FID".

1.4 Autoencoders

Autoencoder is a kind of architecture that allows learning compressed representations in an unsupervised manner. Autoencoder consists of two components — encoder and decoder. Given an input, the encoder processes the input and returns output which is a compressed representation of the original input. The compression typically refers to a dimensionality reduction, for example, the input might be an image with a resolution $128 \times 128 \times 3$ pixels and the encoder might "encode" it into a 512-dimensional vector which corresponds to the compressed representation of the image. The encoder's output is frequently referred to as an encoding of the input. The second component is called decoder and it is used for reconstructing the original input from the encoding.

The autoencoder is typically training using a so-called reconstruction loss (or some variation of a loss that incorporates reconstruction loss into its computation). Ideally, the resulting encoding should work in such a way that the reconstruction of the input corresponds (as close as possible) to the original input. The architecture of a simple autoencoder is shown in Figure 1.30.

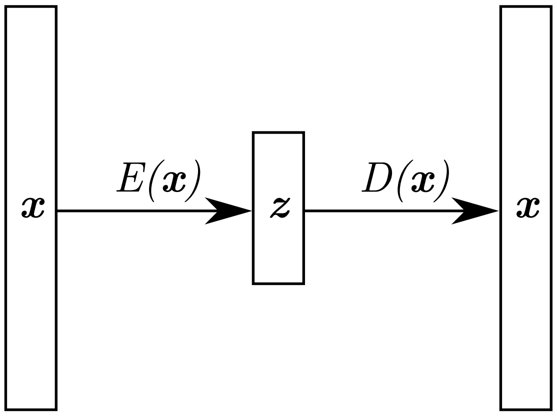


Figure 1.30: Autoencoder's architecture.

Figure 1.30 shows an architecture of an autoencoder. On the left of that figure, there is an input $\boldsymbol{x}$ which could be for example an image. The $\boldsymbol{x}$ is then encoded by the encoder to produce a latent vector $\boldsymbol{z} = E(\boldsymbol{x})$. Finally, the latent vector is passed through a decoder to ideally reconstruct the original input $\boldsymbol{x} = D(\boldsymbol{z}) = D(E(\boldsymbol{x}))$. Notice that the size of $\boldsymbol{z}$ is considerably smaller than that of $\boldsymbol{x}$.

Autoencoders could be trained by using different kinds of loss functions, but they are typically trained by using MSE or binary cross-entropy (the choice of the loss function is also somewhat dependent on the data in this case).

1.4.1 Variational autoencoders

There are several types of autoencoders and one that is frequently used is called variational autoencoder (VAE) [41] which is, similarly to GANs considered as a generative model.

Before introducing variational autoencoders and further delving into them, consider the following, different viewpoint of generative models that will be useful for further definitions. For this viewpoint, assume that the generative model is given a set of observations $\mathcal{X} = \{\boldsymbol{x}^{(i)}\}_{i=1}^{N}$ of a random variable X and its goal is to estimate $P(\boldsymbol{x}^{(i)})$ [22]. One of the approaches for estimating $P(\boldsymbol{x}^{(i)})$ is by assuming that the random variable X in question depends on a latent variable Z. Then the whole generation process consists of sampling $\boldsymbol{z} \sim P(Z)$ followed by generating $\boldsymbol{x}^{(i)}$ from some conditional distribution $P(X|Z)$, i.e.:

$$P(\boldsymbol{x}^{(i)}) = \sum_{\boldsymbol{z}} P(\boldsymbol{z})P(\boldsymbol{x}^{(i)}|\boldsymbol{z}) = \mathbb{E}_{\boldsymbol{z}\sim P(\boldsymbol{z})}P(\boldsymbol{x}^{(i)}|\boldsymbol{z})$$

The conditional probability $P(\boldsymbol{x}^{(i)}|\boldsymbol{z})$ will be estimated using a neural network $P_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}|\boldsymbol{z})$.

Variational autoencoders take a similar approach to this problem and they further assume that $P(Z)$ is fixed and independent on X. The probability $P(\boldsymbol{x}^{(i)}|\boldsymbol{z})$

then can be approximated using $P_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}|\boldsymbol{z})$. Problem is that in order to train the variational autoencoder, the posterior probability distribution $P_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}^{(i)})$ must be known but it is usually intractable. Fortunately, this small issue can be solved by approximating $P_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}^{(i)})$ by a trainable $Q_{\phi}(\boldsymbol{z}|\boldsymbol{x}^{(i)})$.

VAEs are trained by maximizing the following loss [41] (*evidence lower bound (ELBO) or variational lower bound*):

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{x}^{(i)}) = \mathbb{E}_{Q_{\phi}(\boldsymbol{z}|\boldsymbol{x}^{(i)})}[logP_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}|\boldsymbol{z})] - D_{KL}(Q_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}^{(i)})||P_{\boldsymbol{\theta}}(\boldsymbol{z}))$$

Where the expectation $\mathbb{E}_{Q_{\phi}(\boldsymbol{z}|\boldsymbol{x}^{(i)})}$ could be estimated by taking a single sample. Furthermore, $\mathbb{N}(0,1)$ is used as a prior $P(\boldsymbol{z})$. The distribution $Q_{\phi}(\boldsymbol{z}|\boldsymbol{x}^{(i)})$ is parametrized as $\mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}, \boldsymbol{\sigma}^2)$ which brings several benefits, most importantly, it allows for backpropagation, because problem of derivating through $z \sim Q_{\phi}(\boldsymbol{z}|\boldsymbol{x}^{(i)})$ could be solved by using reparametrization trick [41].

More intuitively, this loss consists of the following two components:

- Reconstruction loss ensuring that when starting with $\boldsymbol{x}$ and passing it through Q followed by P should result back in $\boldsymbol{x}$.
- Latent loss forcing the distribution of $Q_{\phi}(\boldsymbol{z}|\boldsymbol{x})$ to be as close to the $P(\boldsymbol{z})$ as possible.

An illustration of images generated by VAE is shown in Figure 1.31.

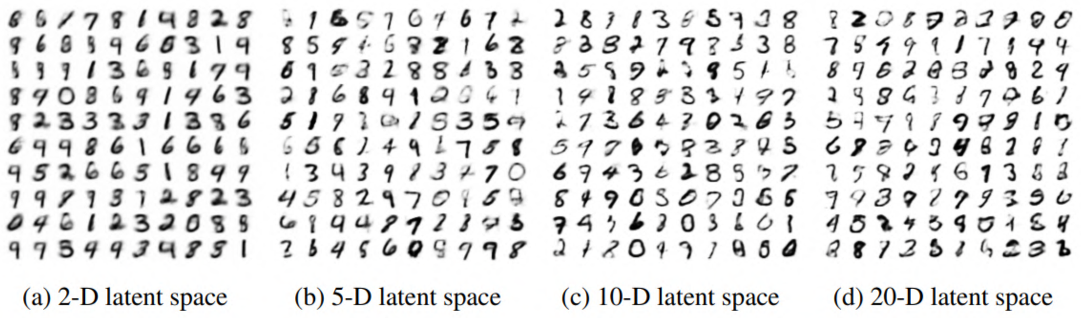


Figure 1.31: Random samples generated by VAE trained on MNIST dataset, compared for different latent space dimensionalities. This image was taken from [41].

Figure 1.31 compares images generated from latent vectors of different dimensionalities. Notice that the latent vector with the least dimensionality (2-D in this case) results in worse results than the images generated from higher dimensionality latent vectors. Especially, see the blurriness of the first image in that figure. The blurriness and/or noisiness is somewhat typical of images generated by VAE and it is often attributed to the dimensionality reduction that happens in autoencoders in general.

2. Our Approach

This chapter briefly describes the analysis, decisions, and overall approach that was used when trying to fulfill the goals defined in Section Goals. A very high-level overview is that the user will somehow specify the suspect's "appearance" and the model will then generate the fillers that have a "similar appearance" to that of the suspect, more detailed description and reasoning over this will be a part of this chapter.

2.1 Analysis

The first big question was to decide and determine the high-level architecture of the model, together with an expected workflow of using the model for police lineup construction. More specifically, the following questions have to be answered before specifying the high-level architecture:

1. What is the ideal user workflow?
2. Choose VAE or GAN?
3. How to control the output of the model?
4. Use a pre-trained model or train from scratch?
5. Which technologies to choose?

All these questions are elaborated upon and eventually answered in the following sub-sections.

Ideal user workflow

Ideally, the user workflow should look in a way that the user simply provides a suspect's photograph together with the number of fillers n and receives a lineup consisting of n photographs of fillers that are very similar[1] to the input (seed) image. Moreover, the user should be able to select some of the generated fillers and let the framework generate more fillers that are similar to the selected filler. This feature should allow the user to continually leverage the generation process towards samples with the desired appearance. This use case is depicted in Figure 2.1.

Even though the abovementioned scenario itself might be sufficient, it can be useful to give the user an opportunity of gaining more control over the generated images. One of the ways to do this could be allowing the user to specify two images and then generate interpolations between these two images. The two images could depict the same person (e.g. two different photographs of a single suspect which might be useful in situations where the suspect has, for example, changed his/her hair color) or they could depict two different persons where each of them has some facial features which are important and which should ideally be present in the output. This leads to the second use case, which is illustrated in Figure 2.2.

[1]According to some similarity measure.

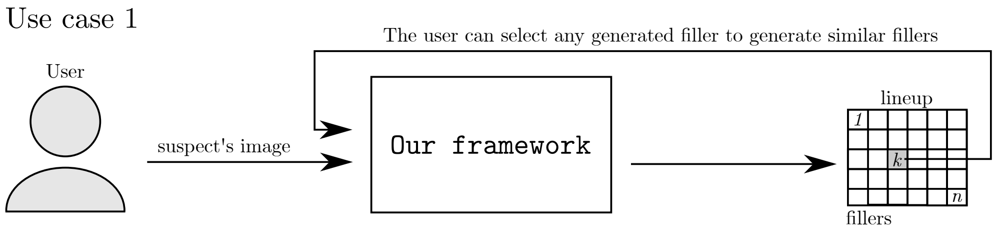

Use case 1
The user can select any generated filler to generate similar fillers
User
suspect's image
Our framework
lineup
1
k
n
fillers

Figure 2.1: Illustration of the ideal user workflow which allows the user to specify the suspect image and obtain the specified number n of fillers with a consequent possibility to recursively generate fillers similar to the selected, already generated filler.

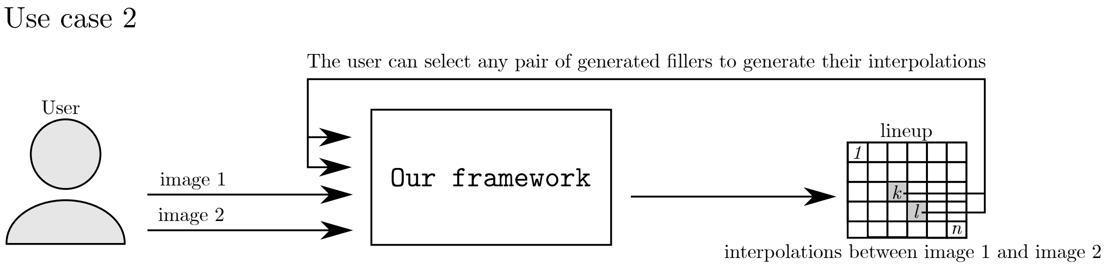

Use case 2
The user can select any pair of generated fillers to generate their interpolations
User
image 1
image 2
Our framework
lineup
1
k
l
n
interpolations between image 1 and image 2

Figure 2.2: Illustration of the second user workflow where the user is allowed to specify two images to the framework which in turn removes n interpolations between these two images. The user is then allowed to recursively select two images from the output and obtain interpolations. It should be mentioned that both use cases 1 and 2 could be combined, so that for example, the user first generates interpolations (use case 2) and then could generate images similar to any of the generated fillers (use case 1).

VAE versus GAN

Both GAN and VAE models have been successfully used for high-resolution, high quality face image generation in the past [59][47][50][51]. However, despite the both's model ability or potential to generate quality face images, it still seems that state-of-the-art GAN models like for example StyleGAN2 [51] outperforms most of the currently available VAE models in terms of image quality. Furthermore, the research in the GAN area also seems to be more active when compared to VAE related research. Because of these reasons, we have decided to choose GAN—more specifically, StyleGAN2—as a model for image generation that will be used.

Controlling the output of the model

The next question is about choosing a way in which the user of the model will be able to control the model's output, i.e. how the "appearance" of the suspect will be described. With GANs, this is usually achieved by somehow passing attributes (e.g. facial attributes of the suspect, hair color, etc.) into the model. This approach will be referred to as an *attribute approach* and examples of architectures where it is used include Controllable GAN [60], Conditional GAN [61]. However, there is a minor issue with the attribute approach because these models need labels as a part of their input which seems to be less convenient for the user. To relieve the user from having to specify these labels manually, it is preferred to choose another approach allowing to describe the suspect's "appearance" directly by the suspect's photograph, i.e. the model's output is controlled by passing the suspect's photo onto its input. Ideally, this approach should simplify the lineup construction even further because the user (in this case, the lineup administrator) does not need to explicitly think about the most important and characteristic features or attributes of the suspect. The aforementioned approach will be referred to as a *"similarity" approach* or *"image seeded" approach* because it works with a similarity between the input and output image and the input image can be considered as a form of seed to the model. The attribute approach versus similarity approach could be thought of as some analogy between match-to-description and resemble-suspect strategies that were previously described in Introduction.

Pre-trained model versus training from scratch

Even though there exist pre-trained StyleGAN2 models, the problem with them is that they were trained on very different data than is needed for purposes of police lineup construction. Either, they were trained on something different than people's faces or they were trained on people's faces but the people came from a very different social group (e.g. actors, celebrities, or other kinds of famous persons) than a typical convict or suspect. Because of this reason, we have decided to train the model from scratch by using a custom dataset that will be more appropriate for this task.

Choice of technologies

There are various deep learning frameworks that are suitable for building deep neural networks and that should be able to cover all the needs of this thesis, therefore it makes perfect sense to stick to some framework instead of implementing everything from scratch. Two, probably the most well-known and most used frameworks are TensorFlow[2] by Google and PyTorch[3] by Facebook. Both of these frameworks could be used from multiple languages, most notably from Python and C++. It was decided to use the Python version of TensorFlow instead of PyTorch because the author of this thesis has more experience in it and Python makes it easier and faster to prototype than C++.

2.2 Dataset

As it was already mentioned, there is a need for a dataset that will be more closely related to the topic of police lineups. An ideal candidate for such a task is the databases of either wanted or missing persons (these typically contain quite "raw" and descriptive images, without any filters and effects that are often present in photographs on social networks and similar sources). Images in such a database are often canonical passport-style photos, which is useful because for lineups it is important to have images that are similar in terms of background, resolution, and other factors.

[2]Visit the project's website [62] for more information.

[3]For more information about PyTorch visit the following website [63].

A lot of countries have their own missing and/or wanted person databases that are typically available through a local police website. Sometimes, the databases are too small, difficult to access or they contain corrupted or somehow broken data. The first idea was to use the Czech database of missing persons[4] but it turns out that this database is too small (containing between 4000 and 5000 records at the time of writing this thesis). After investigating databases from various other European countries, it seems that the database of Polish[5] and Slovak[6] police provide a sufficient number of images of reasonable quality, therefore, images from these two databases will be used for training the generative model in this thesis. When combined together, these two databases contained around 130 000 raw (including corrupted) images[7].

The idea is to download these images, filter them, preprocess them and produce a dataset that contains only images that are appropriate for training the StyleGAN2 model. The word appropriate encompasses that the images should be of sufficient quality (resolution, color, etc.) and they should only display a face of a single person. It makes perfect sense to do this in advance so that the amount of preprocessing steps during model training is minimized in order to maximize the training performance. The preprocessing and filtering parts are really crucial and they deserve a thorough explanation which will be given later, in Section 3.1.

2.3 High-level architecture

The analysis from Section 2.1 showed some of the needed components and most importantly, the expected, high-level overview of the two workflows or use cases from the user's point of view. To remind, the first use case is a similarity search seeded by the suspect's photo, and the second use case is an interpolation between two photos (presumably one of them is the suspect). In both cases, the user should be able to recursively repeat this procedure by either selecting a single filler for which new similar images will be generated (in the first use case) or to select two fillers and generate another set of interpolations of them (in the second use case).

The user will be able to follow both of these use cases and work with the framework through a frontend of some software application. At the same time, the backend part of the software application will be responsible for preparing the output images that will later be served to the user. In order to do this, the application needs to access two models: generator and encoder, both of them pre-trained on a custom dataset that was briefly described above in Section 2.2. The generator will be needed for image generation itself, while the encoder will be needed for obtaining a latent vector $\boldsymbol{z}$ from the suspect's image and therefore to allow filler generation by using vector arithmetics and interpolations of the latent vector $\boldsymbol{z}$. The training of generator and encoder works in two stages, first, the whole StyleGAN is trained, and second, the encoder is trained by using a generator from the first stage as the decoder. Overall high-level architecture is shown in Figure 2.3.

[4]The Czech database is accessible from the following url [64].

[5]The Polish database is accessible from the following url [65].

[6]The Slovak database is accessible from the following url [66].

[7]Note that this number will change because these databases are "live".

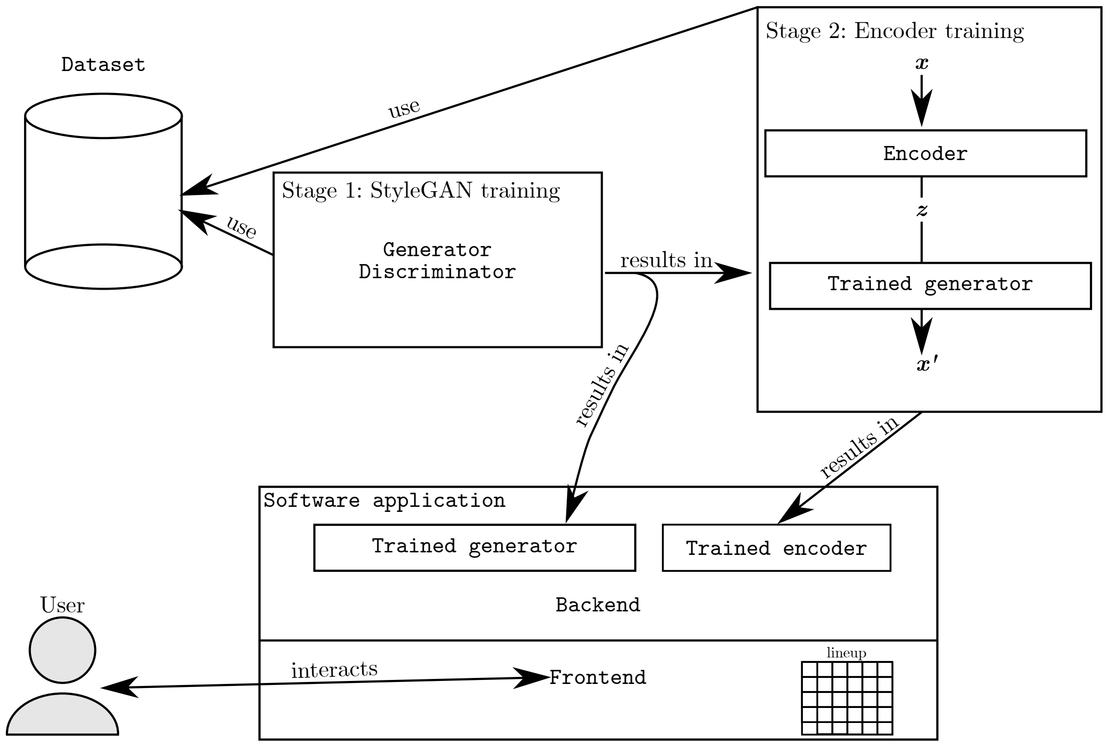


Figure 2.3: An overview of high-level architecture. Notice that the StyleGAN is trained separately and when it finishes, the trained generator is taken as a decoder for an encoder training which is again independent of the previous StyleGAN training. Once both training stages finish, there is a software application that uses the trained generator and encoder to serve the lineups desired by the user.

3. Design

This chapter provides a more detailed explanation of the concepts that were briefly described in Chapter 2. To be more specific, this chapter provides an in-depth description and design choices regarding dataset, generative model (StyleGAN2), encoder and other, feature-related models that play an important role throughout the whole thesis.

3.1 Dataset preprocessing

Even though the Polish and Slovak databases provide images of reasonable quality, the problem is that these databases contain a lot of records that either do not contain any image—making it more difficult to parse—or they contain corrupted, low quality, low resolution, or otherwise inappropriate images. A typical example of an inappropriate image that could not easily be used for model training is an image where there is a photograph of a whole person instead of just its face, or a photo where there is several persons, no person, or when the photo is monochrome/grayscale which is sometimes the case with images in these databases. Examples of inappropriate images are shown in Figures 3.1, 3.2.

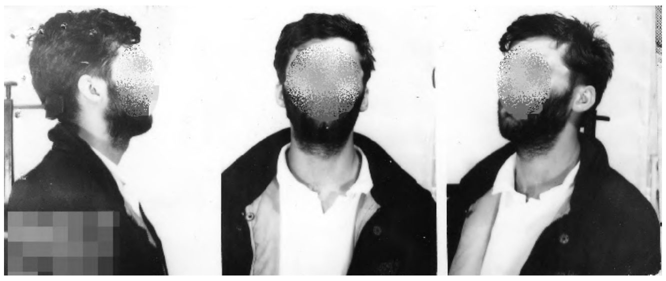


Figure 3.1: An example of inappropriate, multi-view face image. The noise was added intentionally to the images for the sake of anonymization.

Figure 3.1 shows an image that is not suitable for training the model because it contains several sub-images showing a person from different views. Another problem with this image is that it does not show only the face, but also a certain part of the upper body.

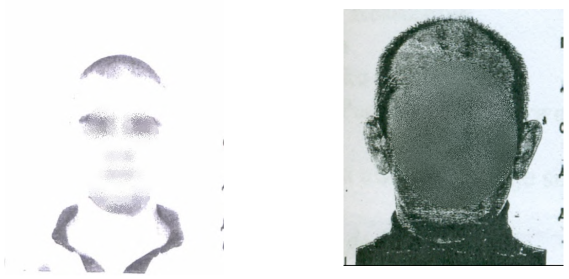


Figure 3.2: Examples of two inappropriate, low-quality images. The noise was added intentionally to the images for the sake of anonymization.

The left image in Figure 3.2 shows a photo that has very poor quality because it is grayscale or maybe almost a monochromatic image and there is not even a clear border between the face and background because the image is too bright. The right image in Figure 3.2 shows another grayscale photo with poor quality.

Another problem relates to the fact that the images in these databases have various resolutions which makes it impractical to train a model. This problem is further supported by a limitation that the chosen StyleGAN2 expects the images to have a squared resolution $n \times n$ where $n = 2^k$ (i.e. powers of two). These two observations suggest that the images should be resized to an adequate power of two. The exact choice of the resolution depends on two factors: the size of the images in the dataset and expected training time. The size of the images within the dataset should be considered, because choosing a resolution that would be larger than most of the images of the data from the database would require most of the images to be upscaled, therefore decreasing the quality of these images. On the other hand, choosing a resolution that is too small would waste the potential of images, i.e. if all the images were of size 512×512 and larger, it would not make much sense to downscale them to 32×32. Finally, the resolution is also subject to performance limitations, because training such a large model as StyleGAN2 might be very demanding and time-consuming when high-resolution images are used. The histogram illustrating the number of images above a certain resolution threshold is shown in Figure 3.3.

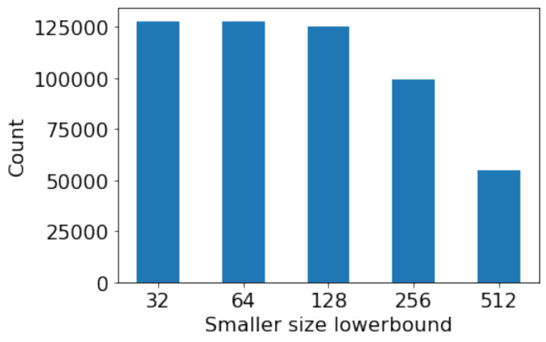


Figure 3.3: The number of images satisfying a constraint that smaller of their sides is not smaller than the specified threshold.

With these observations and limitations in mind, it was decided that all the images will be downscaled to 256×256 so that the potential of the data is maximized and the resulting dataset is as good as possible (given the source images), even when this slightly contracts with the second expectation that the model should take a reasonable time to train (this will be described in more detail later in this chapter.

The problem that the database contains both colored (RGB) and grayscale images can be solved in one of the following two ways: either convert everything to grayscale and train the model on grayscale images or use only the colored images and omit the grayscale images. Choosing between these two options involves some tradeoff because if everything is converted to grayscale, there is a benefit of a larger number of images but at the same time, there is a drawback of generating only grayscale images. On the other hand, omitting all the grayscale images and using only the colored images decreases the dataset size, but allows to generate colored images that are more suitable for police lineups. Based on the relatively low proportion of greyscale images (see Table 3.1) it was decided to keep only colored images in the dataset.

All these quality-related issues are the reason why the downloaded images should first be filtered and preprocessed. The goals of the filtering and preprocessing steps are the following:

1. Discard all invalid images.
2. Discard all images with very low quality (either with a very small resolution, without a person's face, or with a small color spectrum, etc.).
3. Ensure that all the images have the same size.
4. Discard all grayscale images.

These goals has led to a design of a "dataset preprocessing pipeline". The dataset preprocessing pipeline works in a way that it takes the downloaded images and then passes every downloaded image through a series of filtering and preprocessing (mapping) steps. Finally, the images at the output of this pipeline are then used to build the resulting dataset. This approach is illustrated by the pseudocode in Listing 1.

```
dataset = (dataset
    .filter(filter_1)
    .apply(mapping_1)
    .apply(mapping_2)
    .filter(filter_2)
    .filter(filter_3))
```

Listing 1: Illustration of dataset preprocessing pipeline

In order to get rid of low-quality images and to obtain the images of the same size depicting faces of persons (i.e. crop to the face) the designed pipeline from Listing 1 was instantiated with particular steps leading to the following pipeline:

```
FacesDataset \
    .load([folder_path_1, ..., folder_path_n]) \
    .filter(size_filter, min_size=256) \
    .filter(color_filter) \
    .apply(extract_face, min_face_width=128,
        min_face_height=128, confidence_threshold=0.9,
        output_face_width=256, output_face_height=256) \
    .filter(quality_filter, white_threshold=240,
        black_threshold=15, white_ratio_threshold=0.55,
        black_ratio_threshold=0.2) \
    .filter_window(remove_too_similar, window_size=2,
        similarity_threshold=0.9) \
    .apply(convert_to_rgb) \
    .create()
```

Listing 2: Dataset preprocessing pipeline

Listing 2 shows how exactly is the data filtered and preprocessed when creating the dataset that will be used for training. The description of the individual steps from this pipeline follows:

- `.load([folder_path_1, ..., folder_path_n])` takes paths of directories that contain the images and it basically creates an iterator over that images.
- `.filter(size_filter, min_size=256)` adds a new filter operation calling `size_filter` filter function with a parameter `min_size=256` which removes all the images whose size is smaller than 256 pixels.
- `.filter(color_filter)` adds a filter operation calling `color_filter`. This removes all the grayscale images.
- `.apply(extract_face, min_face_width=128, min_face_height=128, confidence_threshold=0.9)` adds a preprocessing step that takes an incoming image and extracts a rectangular sub-image containing the person's head. This works by finding bounding boxes of all faces in the image and returning the largest one that at the same time, satisfies additional constraints. These constraints are that the bounding box should have dimensions of at least 128×128 and the reported confidence of this detection should be ≥ 0.9. If none of the bounding boxes satisfy these conditions, the image is filtered out[1]. The face detection is done using OpenCV and a simple, pre-trained, deep neural network[2] based on ResNet [57]. From this step on, all the upcoming steps will work with the extracted face. More details about the implementation of `extract_face` will be given later in sub-section 3.1.1.
- `.filter(quality_filter, white_threshold=240, black_threshold=15, white_ratio_threshold=0.55, black_ratio_threshold=0.2)` adds a filtering operation that attempts to filter out all images with poor quality. The two police databases contained quite a number of whitish, (not exactly) grayscale or monochromatic images and some of these were not captured by the basic `color_filter` therefore this step serves as an additional filter that attempts to filter them out. This filter takes arguments that specify what is a threshold for a pixel value to be considered a black or white color and then filter images whose percentage of white or black color is above `white_ratio_threshold` or `black_ratio_threshold` respectively.
- `.filter_window(remove_too_similar, window_size=2, similarity_threshold=0.9)` adds a filter window operation, which has a purpose similar to filter operation, but instead of taking a single image, it takes multiple images (a window or range of images). In this case, the `remove_too_similar` takes two consecutive images and it makes sure, that the second image is included in the result only if it is not too similar to

[1]This might seem like a design flaw, but this is done due to the performance reasons so that it is enough to detect the faces only once, instead of first detecting and ensuring that there is big enough face and then detecting again and cropping that face out.

[2]More information about the model can be found in OpenCV respository [67].

the first image (i.e. `similarity_threshold` should be < 0.9). This filter is really specific to the data and it is because the database sometimes contained several images of the same person (either completely identical images or very similar images) in a row and it is not desired to have all of these included in the resulting dataset. This filter exploits the feature of the dataset that if the same person is depicted multiple times, his/her IDs always represent an uninterrupted sequence. More details about filtering similar images of the same person including an example will be given later in sub-section 3.1.2.

- `.apply(convert_to_rgb)` converts the images from BRG format used by OpenCV back to RGB format.

- `.create()` Finally, the create method creates an iterator that allows iterating over the new, filtered and processed dataset. In particular, this means that the pipeline is lazy i.e. it loads at most `window_size`[3] images at a time instead of having to load all of them at once.

3.1.1 Face extraction

The `extract_face` step is actually slightly more complicated than as described above, and that is because there is a fundamental problem with how the face detection model detects the bounding boxes around the face. The problem is that the bounding box is frequently too small, containing only the face without the neck, ears, and hair. This is something that is considered a problem because if the model is trained only on such images, it would probably generate images that have the same flaws (i.e. no ears). To solve this issue, the bounding box that was detected by the face detection model should be slightly extended. Simply speaking, the bounding box should be extended as much as possible, but at the same time, this extension should be the same for both vertical directions and for both horizontal directions (so that the face itself is kept centered plus it should be ensured that this extension does not make the bounding box to grow beyond the desired image size of 256×256 pixels). When the bounding box was extended then if its size is still below the desired size, upsampling should be performed. This procedure works that for each image a goal size is set to the following quantity:

$$goal_{size} = 2max(W_{face}, H_{face})$$

where W_{face} and H_{face} corresponds to the width and height of the largest (in terms of area) face detected in the given image. The idea behind $goal_{size}$ is that there should be enough free space around the detected face itself so that ears, hair, and neck fit into the image. After that, compute the padding for both horizontal directions as:

$$padding_{left} = \lfloor \frac{goal_{size} - W_{face}}{2} \rfloor$$
$$padding_{right} = goal_{size} - W_{face} - padding_{left}$$

[3]Maximum of window sizes in the case that multiple of them were specified and 1 if no window filter was specified.

Then ensure that the padding fits into the image:

$$padding_{left} = min(padding_{left}, left_{face})$$

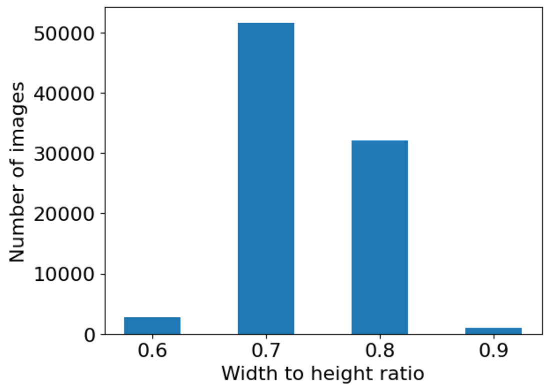
where $left_{face}$ is the x coordinate of the face's bounding box top left corner and similarly:

$$padding_{right} = min(padding_{right}, W_{image} - right_{face})$$

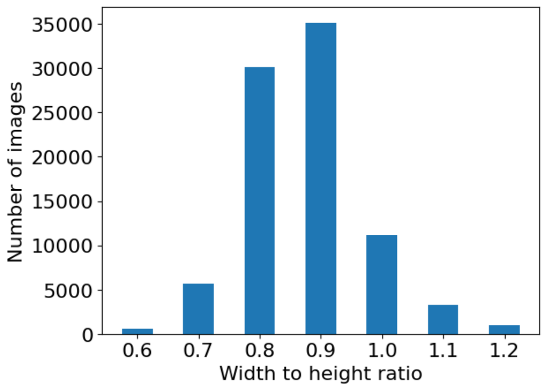
where $right_{face}$ is the x coordinate of the face's bounding box bottom right corner. Finally, in order to keep the face horizontally centered, the padding is set to:

$$padding_{left} = padding_{right} = min(padding_{left}, padding_{right})$$

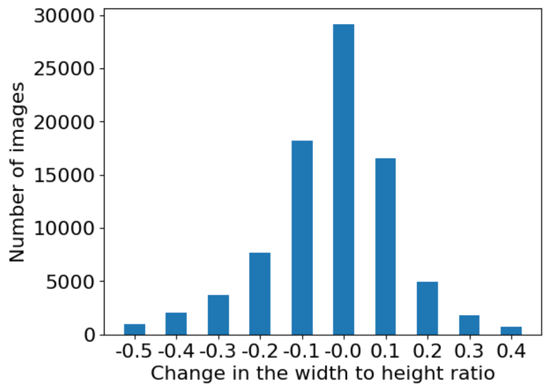
Symmetrical procedure is then performed for the vertical padding and after that, the calculated paddings $padding_{left}, padding_{right}, padding_{top}, padding_{bottom}$ are added to the corresponding side around the face and this padded face is cropped and then resized to the desired dimensions of 256×256 pixels. The only problem with this procedure is that it may change a ratio between face width and face height, however, it turns out that on average, these changes are not very significant, as can be read from the Figures 3.4a, 3.4b, 3.4c. Note that in these three figures, outliers (ratios occurring in less than 500 images were omitted).



(a) Ratios of the original data.



(b) Ratios of the new, processed data.



(c) Change in ratio between new and old data.

Figure 3.4: The degree of preservation of face image proportions.

Figure 3.4a shows the W_{face}/H_{face} ratios, that is the rations of the face's width to face's height in the original data, while Figure 3.4b shows the same quantity for new data, i.e. data after adding padding and resizing the image to the resolution of 256×256 pixels. Figure 3.4c shows how did the ratio change. Notice that even though the ratio has indeed changed, the change is quite small on average meaning that in most of the images the ratio has changed by no more $\approx 10\,\%$ in absolute value.

An example of cropping and padding face from an image is shown in Figure 3.5.

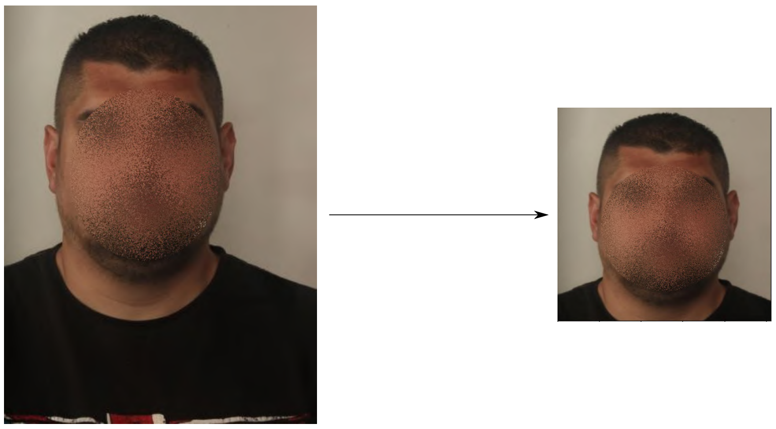


Figure 3.5: The result of cropping and then padding a face image where the left image represents the original image from which the face was cropped then padded and resized to 256×256 pixels. Notice that in the original image, the face is in the top left corner of the image, while in the output image, the face is properly centered, both vertically and horizontally. The noise was added intentionally to the images for the sake of anonymization.

3.1.2 Filtering photos of a single person

An example of two similar images of the same person is shown below, in Figure 3.6. In this case, the two images are not exactly the same, because the image on the left is slightly shifted, but nevertheless, they still show the same person. Intentionally, there is a red border around the images so that their boundary is highlighted to emphasize the fact that the first image is shifted.

Because the images are not exactly the same, it is not enough to compare them pixel by pixel, but this issue can be overcome by comparing their embeddings (which can be obtained using for example a pre-trained OpenFace[4] [69] model) instead of comparing their pixels. More specifically, by calculating an L2 distance between their embeddings. This allows, to some extent, to capture whether two different images show the same person based on a result of comparing this distance to some threshold. However, the threshold should be set in such a way that persons that are only similar, but not same, are not filtered out. Note that this image would be filtered out anyway, because it is in grayscale, so it serves only as an illustration of the purpose of similarity filtering.

[4]More information about the OpenFace model can be found on the following website [68].

Figure 3.6: Two consecutive images of a single person. The noise was added intentionally to the images for the sake of anonymization.

3.1.3 Preprocessing results

After passing the images downloaded from the databases and passing them through the filtering and preprocessing pipeline, the number of images has decreased from ≈ 130000 to ≈ 90000—as can be seen from Table 3.1—which is still comparable to FFHQ dataset [50] that was used by authors of StyleGAN(2). The resulting images are then divided into 10 batches each of them having a size of 10000 images (except for the last one which is smaller) and these batches are then serialized into TensorFlow's `.tfrecord`[5] files which are easily consumable by TensorFlow dataset API[6].

Total images	Valid images	Grayscale/RGB	Only RGB
≈ 130 000	≈ 128 000	≈ 100 000	≈ 91 000

Table 3.1: Number of images in the dataset.

Table 3.1 shows the number of images that remain after filtering the original database with 130 000 images. Some of the images in the original database were not valid (those that could not be read by OpenCV) and after removing them about 128 000 images remained. When the valid images were filtered using all the filters except for filter removing grayscale images (`color_filter`) the number of images decreased to 100 000. When the `color_filter` was also included, the number of images further decreased to approximately 91 000. The impact of individual filters on the number of filtered images is shown in Figure 3.7.

[5]For more information about TFRecord format, visit the following website [70].

[6]More information about TensorFlow dataset API can be found at the following website [71].

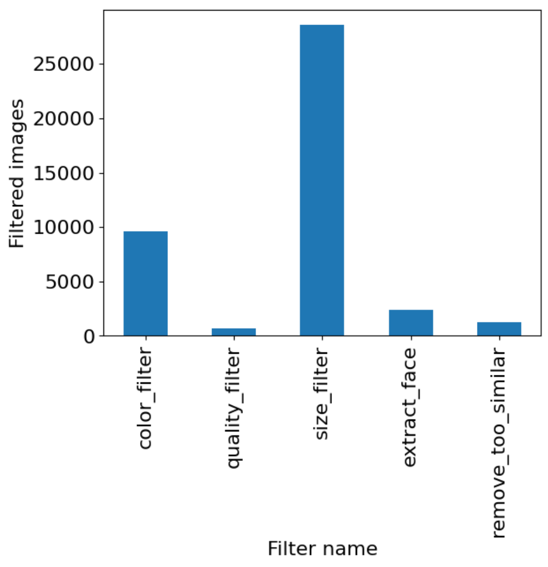


Figure 3.7: The number of images filtered by each of the filters.

3.2 StyleGAN2

The StyleGAN2 model that is being used has the original architecture [51] as was described in Chapter 1 without any modifications.

The model has been trained on the *Faces* dataset which was resized from 256×256 pixels down to 128×128 pixels in order to gain better performance in terms of training speed. All the images were in channels first format and their pixel values were rescaled to $[-1, 1]$ range. The training was done in a distributed manner on 4 Nvidia Tesla V100 GPUs with a batch size 8 per GPU and it was done for 400 000 training steps.

The hyperparameters (learning rates, frequency of regularization steps, etc.) that were used are the same as those used in the original work, see the Github repository [72][7].

3.3 Feature extraction

Before starting work on the encoder, it is important to realize that there will be a need of measuring similarity between two images. The most essential reason for this need is the fact that the encoder basically tries to perform a reconstruction so there will be a need to compare the reconstructed image to the original. However, this is not the only reason for needing the similarity measure, because the requirement of the similarity measure actually pops up everywhere throughout this thesis. Another example is that it will be important to evaluate similarity when deciding whether the generated filler is similar to the input suspect.

There exist various approaches for measuring similarity between two images, the most basic one is probably taking either L2 or L1 distance of their pixels.

[7]Even though the source code of the repository may change and so the hyperparameters, it is always possible to look up the hyperparameters in the source code of this work. See attachment A.

Another option would be to use binary cross-entropy for the same task. The general problem with these approaches is that they do not always express the similarity of two images in the same way as people perceive it. As an example, consider two face images showing almost the same person but each time with different eye color. In this example, the L2 or L1 distance of these two pictures will be very small, although the two pictures are quite different, especially in the context of this thesis and generation of fillers for police lineups where it may be quite important that eye color between the filler and suspect matches.

A common approach for solving the aforementioned problem would be to first extract some features from the pictures and only then calculate their distance or similarity using a standard L2 or L1 distance. The intuition behind this approach is that the features better represent the meaning and attributes or characteristics of the images than pixels do. Ideally, the low distance between two feature vectors should correspond to similar images. Moreover, the distance between images img_i, img_j should be lower than distance between images img_k, img_l $\iff$ img_i, img_j are more similar to each other than img_k, img_l are.

The task of extracting features and calculating their distance can be achieved by using some standard, pre-trained face recognition model, e.g. VGG-Face. However, it would be useful to be able to not only measure a distance between two faces but also to classify faces to some facial attributes, e.g. predict the hair color of a person from an image. This is especially useful for the analysis of the generated data diversity. For these reasons, it was decided to use a custom model that could predict facial attributes and also be used to generate feature vectors by using outputs of a certain hidden layer (typically the one before the classification layer). Before proceeding with training the custom model for attribute classification a suitable training dataset should be found.

3.3.1 Datasets

There exist several datasets of person faces with some facial attribute annotation, but a problem with most of them is that they are either too small, poorly annotated, contain only a few attributes, or that the images have a poor resolution. Some of the datasets are quite rich but they frequently do not contain attributes and instead they provide a link to some open data which is rarely uniform so parsing attributes from such a link is quite demanding. Some of the larger datasets are also often private and therefore not suitable for this thesis. After analyzing several possibilities, the two suitable datasets are CelebA [73] and FairFace [74] datasets.

The CelebA dataset contains more than 200 000 images with a resolution of 178 $\times$ 218 pixels, altogether depicting more than 10 000 identities of celebrities. With its 40 binary attributes per image, this dataset is one of the attribute richest datasets publicly available. Examples of the attributes are, for example, *Eyeglasses, Wavy_Hair, Smiling* which determine whether the person on the image wears eyeglasses has wavy hair and is smiling respectively. However, not all of these attributes are suitable for the purpose of this thesis. For example, it is not that important whether the person is smiling or whether the person wears a hat (this is something that will almost never be the case due to the nature of the images from the police lineup domain) so several attributes were skipped in a favor of gaining better accuracy on the remaining attributes. The list of attributes that were preserved is shown in Table 3.2.

5_o_Clock_Shadow	Bags_Under_Eyes	Bald	Bangs
Big_Lips	Big_Nose	Black_Hair	Blond_Hair
Brown_Hair	Chubby	Double_Chin	Goatee
Gray_Hair	Male	Mustache	Narrow_Eyes
No_Beard	Oval_Face	Pointy_Nose	Receding_Hairline
Sideburns	Straight_Hair	Wavy_Hair	

Table 3.2: The selected subset of attributes from the CelebA dataset.

Fairface dataset consists of more than 100 000 images that are diverse in terms of ethnicity in order to mitigate the bias which is typically present in the face images datasets. There are 7 race categories by which the images are annotated: *White, Black, Indian, East Asian, Southeast Asian, Middle Eastern*, and *Latino Hispanic*. Furthermore, the dataset is also diverse in terms of age as each image is also annotated by one of the following age groups: *0-2, 3-9, 10-19, 20-29, 30-39, 40-49, 50-59, 60-69, 70+*. Apart from the race and the age group, each image is also annotated by gender, but this information is not that important, because it was also present in the CelebA dataset. All the images from the dataset have a resolution of 224×224 pixels.

The fact that there are two datasets that are quite different (in terms of image resolution, attributes, and so on) it seems reasonable to train two separate models one for each dataset. The model trained on CelebA dataset will be referred to as the *CelebA model* while the model trained on FairFace dataset as the *FairFace model*.

3.3.2 CelebA model

The main task of the CelebA model will be to classify an input image into the subset of the attributes listed above in Table 3.2.

Because the images generated by the generative model will have a resolution of 128×128 pixels it makes sense to train this model on the same resolution so that the output images from the generator can be easily passed through this model, therefore the whole CelebA dataset should be resized from 178×218 pixels to 128×128 pixels. The resizing is done so that the image is downscaled while preserving the aspect ratio and then it is padded with zeros to the desired size. This approach ensures that the aspect ratio of the face is preserved.

The problem of classification into the selected subset of attributes essentially corresponds to a multi-label classification, i.e. that a single image can have assigned several attributes. An easy way to do this would be to consider each attribute as a binary attribute (this is actually in correspondence with the way how it is represented in the data) and predict a probability for each of the attributes (by using a sigmoid activation function). This means that each attribute is either present or not present, but what is a problem with this approach is that in some circumstances it is not desired that all attributes from a certain group are not present. For example, there are several attributes representing hair color and attribute stating whether the person is bald or not. Typically it is expected that each person has assigned either a single hair color attribute or a bald attribute which is not always true when using the sigmoid function (consider an *empty prediction* scenario when all these attributes have a probability < 0.5).

After observing the empty prediction issue it may be tempting to divide the selected attributes into several categories of exclusive attributes and classify them by using softmax activation in each of the categories. However, it turns out that even this approach is not ideal and that is for the following reasons. First, there are not that many groups apart of the hair color that could be identified therefore this approach leads to one or some small number of moderately sized groups of attributes, and the rest are small groups of a single attribute that still need to use sigmoid activation. The second and more important issue is that the data annotations are not perfect and often there are either missing annotations (e.g. that the given image has assigned neither hair color nor bald attribute) or multiple attributes which would be considered exclusive when softmax was used (i.e. there are cases, where for example, a single image has assigned multiple hair colors). After all these observations and corner cases, it seems easier to stick to a simple binary classification into several attributes with a possibility to solve some of the corner cases using a kind of postprocessing (e.g. if neither of the hair colors nor bald attribute was classified because they all were under a threshold for the sigmoid activation, then simply choose the one with the largest value).

The architecture of the model is illustrated in Figure 3.8.

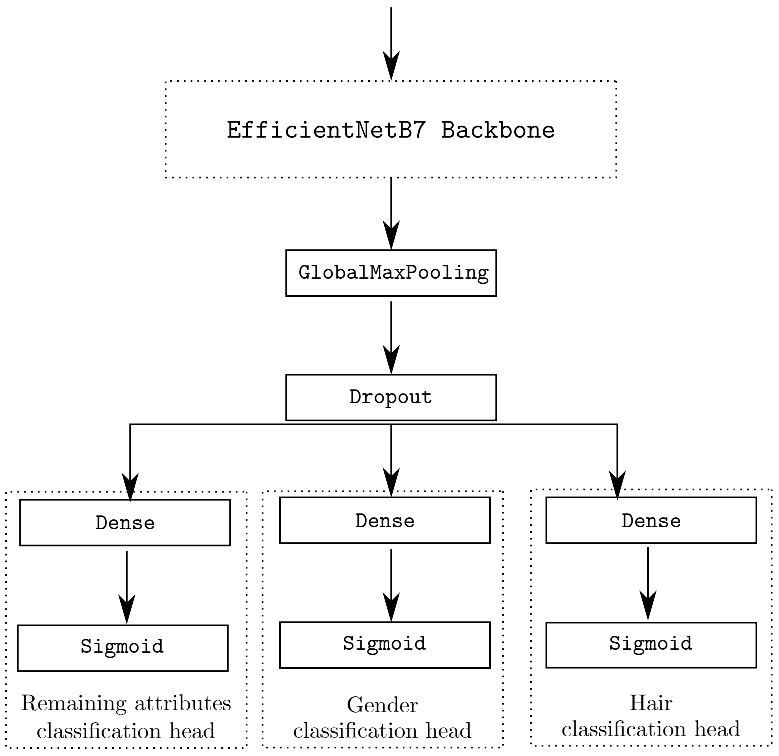


Figure 3.8: The architecture of the CelebA model.

As can be seen from Figure 3.8 the model itself is a fairly standard classification model, using an EfficientNetB7 [75]—without top layers, pre-trained on ImageNet [76]—as a backbone, followed by a global max pooling and a dropout layer with 0.5 rate. At the end of the model, there are three classification heads for hair, gender, and remaining attributes. Having these three classification heads means that instead of producing a single large vector of outputs, the model actually produces several smaller vectors of outputs corresponding to the following three categories: `outputs_hair`, `outputs_gender`, `outputs_remaining`. The reason for this division is mostly because of the historical reasons when experimenting with different activations (i.e. using softmax for hair categories) and also to get extra control over the outputs (i.e. to have a possibility to easily select the hair outputs). There is also an L2 regularization incorporated which together with the dropout layer tries to mitigate overfitting.

Training of the model proceeds as follows: first, there is a fine-tuning for 15 epochs when the backbone weights are kept frozen and then there are another 15 epochs with backbone weights unfrozen, so in total, the model is trained for 30 epochs. During training, an Adam optimizer is used with a learning rate that for the fine-tuning part starts at 1e−4 for the first 10 epochs and then gets decreased to 1e−5 for the next 5 epochs. After the fine-tuning, the first 5 training epochs continue with the learning rate 1e−5, and then it is decreased to 1e−6 for the last 10 epochs. The exact details could be found in the accompanying source code. Moreover, two callbacks are used, one for reducing the learning rate on the plateau and another for early stopping. The batch size was set to 64.

There is a problem which is related to the CelebA dataset, that it is highly imbalanced and several attributes are very rare in the data. This causes the model to have a poor recall thus being unable to classify these rare attributes even when they are present in the input image. There exist different strategies for mitigating the problem of the imbalanced dataset and the two which are used in the CelebA model are a bias initialization trick [77] together with a focal loss [78].

Bias initialization trick is used to manually initialize bias to a value that reflects the class imbalance. This was done by implementing a custom initializer called `ConstantTensorInitializer`.

Focal loss is a loss function that can be helpful when training in cases when the data classes are highly imbalanced. Generally, the idea is that the well-classified examples are down-weighted (so they are treated as a kind of easy example) to allow the model to focus on the hard examples. Originally this loss function was used in the object detection where there was an extreme imbalance between background class and other classes (in the case of CelebA there might be an extreme imbalance between, for example, positive *Bald* examples and negative *Bald* examples (only around 2 % of the images have the *Bald* attribute). Focal loss deals with two parameters [78]: α and γ and in this case their values were set to $\alpha = 0.5,\ \gamma = 2.0$.

3.3.3 FairFace model

The FairFace model predicts age and race for each input image. In contrast to the CelebA model, the FairFace model does not perform multi-label classification into several binary labels, but instead, classifies always exactly two labels (age and race) each behaving as a multiclass classification. Said in other words, the classification is performed using softmax activation instead of sigmoid activation. The main reason why it is done in this way and not the same way as for the CelebA model is that the individual attributes are much more exclusive than for CelebA attribute. This decision is further supported by the fact that in the training data, each image has exactly one age group and exactly one race assigned, i.e., multi-class annotations are used.

The architecture of the model is again, fairly standard and similar to that of the CelebA model as illustrated in Figure 3.9.

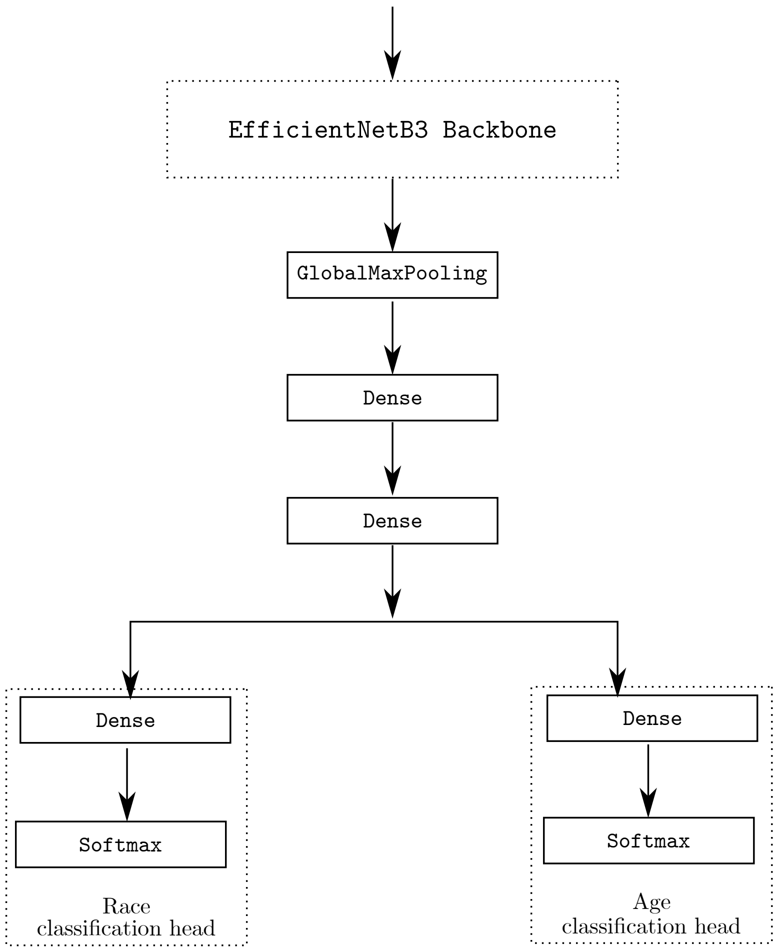


Figure 3.9: The architecture of the FairFace model.

As with the CelebA model, also the FairFace model uses a pre-trained EfficientNet [75] backbone, but EfficientNetB3 is used instead of the EfficientNetB7, and Noisy Student [79] weights are used instead of using the ImageNet weights as was the case with CelebA model. The reason for choosing different backbone and weights was that they have empirically shown slightly better performance in this case. The backbone is followed by a global max pooling but in contrast to CelebA model, there are two more hidden layers with 2048 units and ReLU activations because they empirically showed to bring good performance. Similar to the CelebA model, the FairFace model also has its output divided into multiple parts, namely: into age and race providing more readable and intuitive access to the individual attributes and giving an extra control over these parts (e.g. getting embedding representing only the age).

The model uses neither L2 regularization nor dropout—because the B3 backbone is substantially smaller than B7 so the overfitting is not that visible—but still, in order to mitigate possible overfitting and to make the model more robust a data augmentation is used (saturation, brightness, etc.).

The model is trained using Adam optimizer against categorical cross-entropy loss for 5 epochs with backbone's weights frozen and learning rate set to 1e−4 followed by another 25 epochs with unfrozen backbone and learning rate decreased to 1e−5. Same as with the CelebA, there is a callback for reducing the learning rate on plateau combined with an early stopping callback, but otherwise, the learning rate is not changed during training. The batch size during training is set to 64 as was the case with the CelebA model.

3.3.4 Combining the models

To simplify working with predictions of the CelebA and FairFace models, it makes sense to create a pre-trained CelebA model and pre-trained FairFace model and then make a new model which will call these two models and concatenate their outputs into a single output vector. This is useful because instead of having to call two models every time during prediction, it is enough to call only a single model.

It is also useful to create a wrapper for using the CelebA model and FairFace mode to get the embedding of an image. This is done by using outputs of a layer (last dropout in the case of CelebA model and last dense in the case of FairFace model) before the classification head of each of the models and then concatenating these outputs into a single large feature vector/embedding with dimensionality equal to 4608. Although the number 4608 may seem weird it is because CelebA uses EfficientNetB7 as a backbone which has layers with 2560 at the end while FairFace model uses EfficientNetB3 followed by some hidden layers with 2048 units.

3.4 Encoder

As it was already briefly mentioned in Chapter 2 an encoder will be needed for controlling the generator's output and subsequently to allow generation of similar images by "slightly adjusting" the latent vector corresponding to the suspect's image. This section describes the architecture of the encoder and also clarifies the phrase "slightly adjust the latent vector to generate similar images".

3.4.1 Initial approaches

Remember from Chapter 1 that the generator takes a latent vector $\boldsymbol{z}$ and then produces an image from it. Therefore the simplest approach for the encoder is to attempt to encode the input image as a latent vector $\boldsymbol{z'}$ and then feed it to the generator which should, ideally, generate (reconstruct) an image similar to the input img. However, the problem with this approach is that the input image reconstructions have very low quality and they are often not very similar to the input image as can be seen in Figure 3.10.

After failing with the first approach, another attempt was to use the encoder to encode the image into $\boldsymbol{w}$ instead of $\boldsymbol{z}$ where $\boldsymbol{w}$[8] is the output of the map-

[8]The term $\boldsymbol{w}$ will also sometimes be used when referring to the whole space ($\mathcal{W}$) of all $\boldsymbol{w}$'s and similarly for $\boldsymbol{z}$ ($\mathcal{Z}$) and $\boldsymbol{w}+$ ($\mathcal{W}+$, which will be described later). Nevertheless, the meaning should be clear from the context.

ping network, i.e. $\boldsymbol{w} = \mathit{mapping\ network}(\boldsymbol{z})$ as it was described in Chapter 1. However, this approach also was not successful because even though the generated images were more similar to the input image, they were blurred as shown in Figure 3.10. Generating from $\boldsymbol{w}$ also required a slight change in the way how the generator is called, i.e., the mapping network should be skipped during these calls (calling synthesis network directly). Through the rest of this thesis, these two approaches will be frequently referred to as $\boldsymbol{z}$ *approach* and $\boldsymbol{w}$ *approach*.

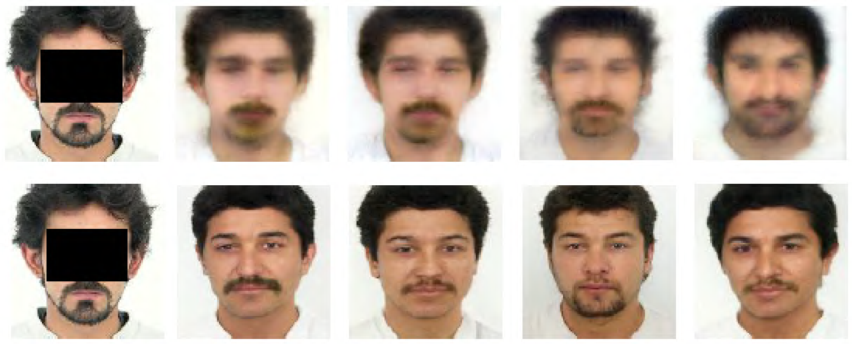

[redacted]
[redacted]

Figure 3.10: Comparison of $\boldsymbol{w}$ approach (first row) and $\boldsymbol{z}$ approach (second row).

The first row of the Figure 3.10 shows several reconstructions (for different architectures and/or hyperparameters) using $\boldsymbol{w}$ approach, while the second row shows reconstruction using $\boldsymbol{z}$ approach. In each row, the left-most image is the (censored) input image and the rest of the images in each row are the reconstructions. Although the input image is censored it can be easily seen that the reconstructions from $\boldsymbol{z}$ are quite dissimilar from the input. On the other hand, reconstructions from $\boldsymbol{w}$ seem to be more accurate (notice for example the beard) but are very blurry.

3.4.2 Final approach

One of the possible reasons why the images were so blurry in the $\boldsymbol{w}$ approach could be the large compression that the encoder has to make when encoding in the image (a tensor with $128 \times 128 \times 3$ dimensions is reduced to only 512 dimensions). Intuitively, this problem could be mitigated by increasing the output dimensionality of the encoder hereby giving the encoder more power. The problem is, that the output dimension of the encoder cannot be just arbitrarily changed because it needs to match the dimensionality the generator is expecting.

Note that in StyleGAN the synthesis network consists of 9 blocks where each of the blocks receives 2 style vectors where each of them originates from $\boldsymbol{w}$. This means that at some point, there are a total of $9 \cdot 2$ copies of $\boldsymbol{w}$ that will be referred to as $\boldsymbol{w}+$ (i.e., $\boldsymbol{w}$ gets broadcasted to $\boldsymbol{w}+$). Therefore it is possible to let the encoder encode images into $w+$ instead of w which further means that the encoder's output dimensionality can be effectively increased from 512 to $18 \cdot 512$. This approach is also used by other authors [80]. In the rest of this thesis, the whole, aforementioned approach will be referred to as $\boldsymbol{w}+$ *approach*. The results for the $\boldsymbol{w}+$ approach will be shown later, in Section 5.3.

The $\boldsymbol{w}+$ approach outperforms both previous approaches ($\boldsymbol{z}$ and $\boldsymbol{w}$) and therefore it was decided to use this approach in the final model of the encoder.

Note that the dimensionality of $\boldsymbol{w}+$ is $18 \cdot 512$ in the case of a generator working with images having a resolution of 1024×1024. In the case of working with 128×128 pixels images, the number of blocks is 6 thus the dimensionality of $\boldsymbol{w}+$ is $12 \cdot 512$ which is still quite an improvement when compared to 512 in the case of $\boldsymbol{w}$ or $\boldsymbol{z}$.

3.4.3 Encoder model

The initial experiments were using an encoder model with a simple convolutional network with a standard ResNet-like architecture. Hover, after plenty of experiments with different hyperparameters and adjustments of the architecture (i.e., changing number of channels, layers, etc.), the results still were not satisfying so the ResNet-like architecture was abandoned and a different architecture—which will be described below—was used for the final version of the encoder model.

The architecture of the encoder model is based on feature pyramid network architecture similarly to [80]. An EfficientNetB7 model pre-trained on ImageNet is taken as a backbone from which the features for the pyramid are extracted. From the backbone, the last layers of any block which are also the last layers before the spatial resolution changes are taken as the feature layers, and their outputs are the features themselves. For the EfficientNetB7 operating on input with a resolution of 128×128 this approach results in five layers and only the last three of them are taken and used as *c3, c4, c5* in the feature pyramid (see sub-section 1.2.2 for more details and illustration of feature pyramid network architecture).

Once *p3, p4, p5* are computed from *c3, c4, c5* in the way it was described in sub-section 1.2.2, they are passed through the so-called *map2style* [80] block which reduces their spatial dimension all the way down to 1×1 ($1 \times 1 \times 512$) producing a single $\boldsymbol{w}$ vector. Because there would be only 3 vectors (one for each of *p3, p4, p5*) and totally 12 are needed, there are actually 12 map2style blocks $mp_1, \ldots, mp_{12}$ where $mp_1, \ldots, mp_3$ take *p5* on their inputs, $mp_4, \ldots, mp_6$ take *p4* on their input and the rest $mp_7, \ldots, mp_{12}$ take *p3* on their input. This distribution of *p3, p4, p5* into the map2style blocks showed empirically the best performence from the tested combinations. Note that this distribution is also slightly different to that used in [80] mainly because working with different spatial resolution. There was also an attempt to incorporate *p6, p7* into the process but it did not seem to have any beneficial effect on performance.

Finally, the outputs of all the map2style blocks $mp_1, \ldots, mp_{12}$ are stacked onto each other to produce the $\boldsymbol{w}+$. After that, the $\boldsymbol{w}+$ is passed through a layer that performs normalization similar to the normalization which is used in the mapping network of the StyleGAN. This normalization seems to improve the performance, empirically, because without it the results were sometimes diverging.

The encoder was trained using Adam optimizer with a learning rate 1e−3 for 5 epochs. The loss function that was used during training is the following:

$$L(\boldsymbol{y}, \boldsymbol{w}+) = L_{pixel} + \lambda L_{feature}, \ \lambda = 0.5$$

Where the $\boldsymbol{y}$ were input images for the encoder and $\boldsymbol{w}+$ were latent vectors predicted by the encoder for the given input images. L_{pixel} was calculated by reconstructing the input images from $\boldsymbol{w}+$ as $\hat{\boldsymbol{y}} = G(\boldsymbol{w}+)$ followed by calculating pixel-based $MSE(\boldsymbol{y}, \hat{\boldsymbol{y}})$. $L_{feature}$ was calculated as $MSE(\boldsymbol{y_{features}}, \hat{\boldsymbol{y}}_{\boldsymbol{features}})$. Both losses were also normalized into $[0, 1]$ range.

3.4.4 Fine-tuning approach

After running several experiments it started to seem that the problem is the generator not being able to generate some of the images, i.e. that the problem was no longer in the encoder. To verify this expectation—at least partially—it is possible to try to fine-tune the encoder by using only a small subset of the training dataset and running it for many iterations. Ideally, the encoder should overfit on this small dataset and should be able to provide an almost perfect reconstruction of these few images, albeit performing worse on unseen data. However, it turned out that even these efforts have failed and the encoder is not able to provide a perfect reconstruction even in this simplified scenario. This clearly supports the previous belief that there is a problem with the generator and not with the encoder.

There are various possible reasons why the generator is not able to generate some kinds of images, but the most probable one is that it is caused by the dataset which was used for StyleGAN training. It does not seem that there is a problem with the dataset size, because the *Faces* dataset is actually larger than FFHQ dataset that was originally used by StyleGAN authors. Instead, it seems that the *Faces* dataset does not provide enough diversity in the images, at least, when compared to FFHQ. For example, the *Faces* dataset contains mostly male photographs and there is only a small percentage of females in it as it will be shown later, in Section 5.2. The problem of a weak generator could be solved, to a certain extent, by extending the training dataset if more data is available. However, this is not always an easy task to do, because getting more images with a reasonable quality is quite demanding.

The fact that the encoder cannot be fine-tuned on the small sample until overfitted is rather unfortunate because if it was possible then the problem of an (almost) perfect reconstruction would be nearly solved. Although it may seem counterintuitive that fine-tuning until overfitting would help somehow (because generally, overfitting is something undesirable that should be avoided) it is actually true, because in an extreme case, the subset of the data to which the encoder could be overfitted is just the single image that should be reconstructed. Provided that the fine-tuning can be done in a reasonable time (at most a few minutes), the encoder can get fine-tuned to the suspect image provided by the user. When the user provides a new suspect image, the encoder should be discarded and a new encoder could get fine-tuned until again overfitted to that new image and so on and so forth. The two problems that are connected to this idea are the fact that the fine-tuning must be fast enough and also that the encoder itself is not able to do this. The first problem could be solved by using a pre-trained encoder and then perform fine-tuning to a dataset consisting single image provided by the user and repeated reasonably many times. The second problem can be resolved by realizing that even though the encoder still cannot be fine-tuned until eventually overfitting and starting producing perfect reconstructions, another component should be able to do so, namely, the generator.

The abovementioned idea could be summarized as *intetionally overfitting a* *generator by fine-tuning on a single image—the user-provided suspect's image—so that the image could be reconstructed in a better way* and it will be referred to as *fine-tuning approach* or *overfitting approach* throughout this thesis. Indeed, this approach will just work, provided that the generator is being trained for a sufficient number of steps it will eventually learn the weights so that it can perfectly reconstruct the suspect's image provided by the user. However, there is yet another problem and that is the fact the generator could overfit so much that it eventually stops generating anything meaningful beyond that single image. A possible approach to this problem could be decreasing the number of steps for which the generator is being fine-tuned, therefore giving a trade-off between reconstruction quality and generator's degeneration with a hope that after finding a proper threshold the reconstruction will be better than without fine-tuning the generator at all, while at the same time, the generator will still be able to generate meaningful results ("around" the suspect's image).

The trade-off between reconstruction quality and generator degeneration could be alleviated by realizing that the encoder can get involved in this process so that the encoder and generator weights are updated alternately. This is done in a hope that even though the encoder itself does not have a capacity to overfit onto the given suspect's image and provide a perfect reconstruction of it, in cooperation with the generator's weights updates it will eventually be possible. Furthermore, this reduces the generator degeneration because it is updated only half the steps as before (the rest half are the encoder updates). To further support the decrease in generator degeneration, the generator updates can be done even less frequently than every second step, i.e. every kth step. The goal of this approach is then to find proper hyperparameters, namely: number of steps (or epochs and steps per epoch) together with a number k determining the frequency of the generator's weights updates so that the generator does not degenerate and the reconstruction is good enough. Moreover, it should be ensured that this process does not break the semantics of the generator's latent space, namely, that vector arithmetics and vector interpolations are not completely broken and lost.

The generator training step is quite different from a standard generator training step because it uses a different loss function. The loss that is used is MSE between pixels of the generated image and input image. **In conclusion**, it is important to mention that this is rather an experimental approach that may help (and also helps as will be seen later in Section 5.3) but it is not intended as an always working, generally applicable approach.

3.4.5 Generating similar images

This section will clarify the phrases *somehow manipulate/adjust/modify the latent vectors to generate similar images* that were used in the previous sections.

The basic idea of generating similar images is following: take an input, encode it as $\boldsymbol{v} \in \mathcal{W}+$ latent vector, shift this vector in certain direction(s) to produce a set of latent vectors $\boldsymbol{v_1}, \ldots, \boldsymbol{v_k} \in \mathcal{W}+$ and then generate images from these vectors as $img_i = G(\boldsymbol{v_i})\ i \in \{1, \ldots, k\}$. Ideally, the resulting images should be similar (in terms of appearance and/or facial attributes) to the input image.

The problem with this idea is that it is quite difficult to decide in which directions the latent vector should be shifted and also what should be the magnitude of the shift. Shifting in the wrong direction or shifting the latent vector too far from the original latent vector could lead to very poor results.

To avoid the necessity of deciding on the direction and magnitude of the shift, a simpler approach would be to take latent vector $\boldsymbol{v}$ of the input image and think of it as a point in a n dimensional space, where n is the dimensionality of the latent vectors from $\mathcal{W}+$, i.e., $12 \cdot 512$. When looking at the problem from this perspective, the manipulations with the $\boldsymbol{v}$ could be replaced by sampling points that are close to $\boldsymbol{v}$. Ideally, the latent space should behave nicely so that the close latent vectors correspond to similar images. The close points are exactly those that lie in a neighborhood, or in other words, inside a n dimensional ball with a center at $\boldsymbol{v}$ and yet unknown radius r. Getting the latent vectors $\boldsymbol{v_1}, \ldots, \boldsymbol{v_k}$ then corresponds to sampling random points from this ball.

Sampling latent vectors from an n dimensional ball simplify generating similar images because it is no longer needed to think about directions and magnitudes of the shifts. Instead, these parameters were reduced to a single parameter and that is the radius r of the n dimensional ball. Results when using this method will be shown later, in Section 5.3.

After experimenting with the aforementioned approach for generating similar images a difficulty with using latent vectors from $\mathcal{W}+$ arose. It seemed that for a significant portion of the latent space, the generated images are corrupted or visibly defected. These issues are most visible when sampling random vectors from $\mathcal{W}+$ itself (just random, normally distributed vectors from the whole space, not from the n dimensional ball around some point) which led to really bad results as can be seen in Figure 3.11.

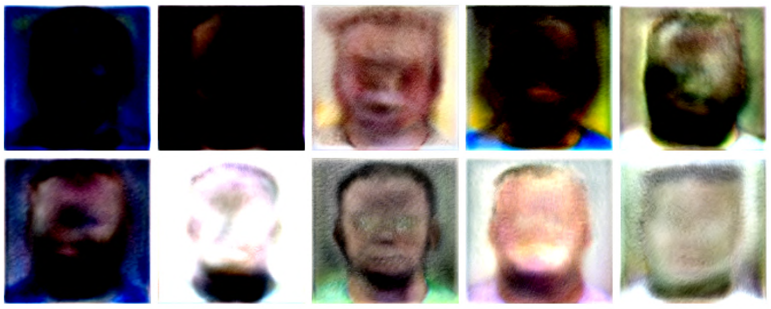


Figure 3.11: Very poor results when sampling random latent vectors from $\mathcal{W}+$.

Figure 3.11 shows 10 images generated from latent vectors that were randomly sampled from a normal distribution. The quality of these images is so low that it is not even recognized they are showing persons. The hypothesis why this is happening is that the $\mathcal{W}+$ space with its $12 \cdot 512$ is simply too large that it contains several areas that are not properly covered by the generator and that leads to such poor results as those from Figure 3.11.

Under the assumption of the aforementioned hypothesis, it makes sense to attempt limiting the $\mathcal{W}+$ to its subspace that contains only (or mostly) latent vectors corresponding to meaningful images. An idea of a simple approach to do this is to use principal component analysis (PCA [81]). The intuition behind using PCA for this scenario is to reduce $\mathcal{W}+$ to keep only the directions that significantly affect the change in images and skip the rest in a hope that the directions being kept are those that allow adjusting a latent vector to another yet still having a meaningful corresponding image.

Next, the number of dimensions (components) to which the PCA should reduce the latent vectors from $\mathcal{W}+$ should be decided. Empirically, after several experiments, it seemed that the results are reasonable with dimension being equal to 48 for easily reconstructable images, but a larger value was needed for the difficult[9] and eventually, it was set to 256.

The PCA implementation from Sklearn[10] requires to first fit the PCA on some data. For this task, a sample of 10 000 images generated by the generator was used. The reason for using images generated by the generator instead of the real images is the fact that the encoder performs better on them but this decision was not verified by comparing it with fitting on real images. Once PCA is fitted, it can be used to first transform the latent vector $\boldsymbol{v}$ to $\boldsymbol{v'}$ in the subspace of $\mathcal{W}+$, then sample vectors $\boldsymbol{v_i}$ from the ball around the resulting point $\boldsymbol{v'}$ (in the subspace) and perform inverse transform of all the sampled vectors. Obviously, the inverse transform is not "loss-less" and therefore it will not lead back to the $\boldsymbol{v_i}$. Ideally, the "loss" of information should be connected to those components of $\boldsymbol{v_i}$ whose change would break the corresponding image. The results of sampling random vectors in the subspace of $\mathcal{W}+$, then inverse transforming these vectors and generating images from them are shown in Figure 3.12.

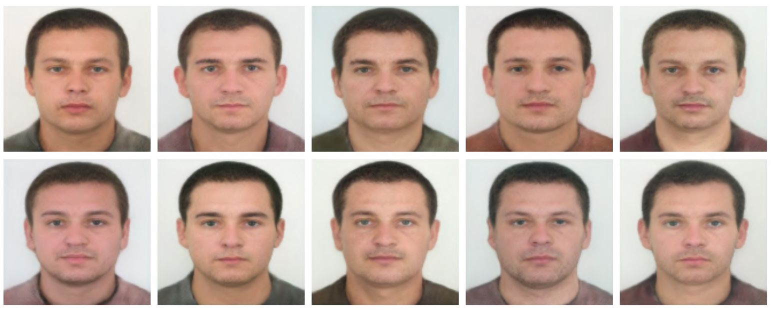


Figure 3.12: Better results when sampling random latent vectors from the subspace of $w+$.

As can be seen from Figure 3.12 the results are much better (although not very diverse in this case) than those from Figure 3.11 because now all the generated images are meaningful and show persons face.

It should be mentioned, that when the images are generated from latent vectors randomly sampled from a $n$ball (instead of the whole space $\mathcal{W}+$) around latent vector $\boldsymbol{v}$ that corresponds to a meaningful image then the quality drop does not appear (or at least it is much less significant). However, even in that case, it is quite difficult to decide a correct value for radius because it seems, empirically, to be more sensitive than the PCA version (i.e. the difference between radius producing images which are almost the same versus those that are corrupted is quite small).

Finally, it should be mentioned that the way how similar images are generated partially invalidates the statement from 3.4.4 about fine-tuning of the encoder until overfitting being safe. Actually, it is no longer safe because using an overfitted encoder for obtaining latent vectors on which PCA is fitted may lead to poor

[9]For an explanation of easy and difficult to reconstruct images, see Section 5.3.

[10]For more information about Sklearn, visit the project website [82].

results. Furthermore, notice that after fine-tuning the encoder, the PCA should be re-fitted because the vectors on which it was previously fitted no longer correspond to the encoder. After realizing that the overfitted encoder is not safe and also after experimenting with generating similar images for various input images, the final frequencies for training steps of the encoder, generator (MSE on pixels of generated and input image) was set to 75 and 1 respectively and the results will be shown later, in Section 5.3.

4. Software Architecture

This chapter is concerned with an architecture of a software application that will allow the user to use the models from previous sections for a task of filler image generation.

The high-level architecture that was shown in Section 2.3 already revealed that the user will be controlling the generation of the fillers through a software application and it remains to show the architecture of this application in a greater detail. A very high-level look at the software architecture is shown in Figure 4.1.

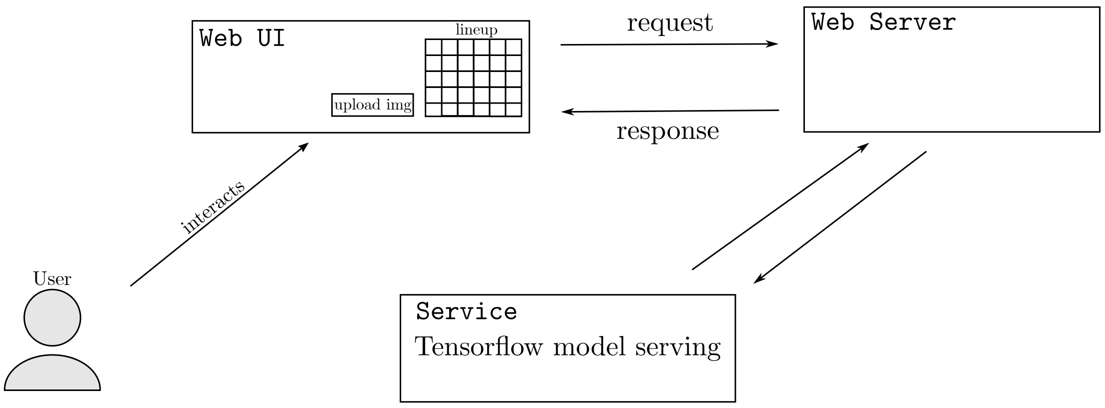


Figure 4.1: An overview of high-level architecture.

Figure 4.1 presents a high-level architecture of the proposed software application. The idea is that the application will consist of two main parts: web UI (frontend) and web server (backend). The frontend part will be handling all the user interactions transforming them to requests that are then going to be passed to the server, handling the incoming responses, and subsequently transforming these responses to results that are then presented to the user. The server is responsible for handling the requests from the frontend and for sending responses. When building the responses it will sometimes be needed that the server uses (consumes) the pre-trained models from Chapter 3.

It is important to mention, that the machine which will consume the pre-trained models needs to have a CUDA-available GPU because the StyleGAN model is quite large and it works in channels first mode which requires a hardware accelerator—in this case, the GPU.

The easiest way how the server can consume the pre-trained TensorFlow models is that it could load the models directly. As mentioned before, the GPU capability is crucial, but often CUDA-available GPUs are not part of standard server installations. This problem could be solved by moving the image generation using TensorFlow models outside of the server into some separate service (denoted as TensorFlow model serving in Figure 4.1). Such an approach is more flexible because it allows it to keep only a single instance of this service running on some GPU-equipped machine and then there can be a separate server(s) running on less powerful hardware and communicating with the service through some APIs.

TensorFlow already supports the scenario of serving the models in a separate container and there already exists a serving system called TensorFlow Serving [83] but there is a problem with this system—the models in TensorFlow Serving are expected to be used for prediction/inference and this is something that is not sufficient for the purpose of this thesis. One of the reasons why it is insufficient is that for example, the fine-tuning approach from Section 3.4.4 needs to modify the generator's weights and therefore a simple inference is not enough. This reason makes TensorFlow Serving inapplicable for our scenario and a custom way of serving the models will be needed.

4.1 Implementation analysis

This section briefly describes all the decisions that were made during the development of the application, together with reasoning over other possibilities and some of the issues encountered and tackled during the development.

4.1.1 GPU cluster

The need for a GPU for running the StyleGAN model has led to a decision to host the whole application in a computational cluster. Assume a generic cluster with the following properties:

- The cluster consists of nodes divided into partitions; the user of the cluster can run jobs on these nodes. Each job is identified by some unique job ID.
- Regardless of the partitions, the nodes are of two categories: frontend nodes which are visible from the outside of the network (and could be accessed via e.g. SSH), and worker nodes that eventually run the jobs.
- The worker nodes inside the cluster are managed by an arbitrary resource manager with an additional feature of running jobs as containers where the actual job is running.
- Worker nodes are of two types: *gpu* and *cpu* where the former are equipped only with CPU while the latter also has a CUDA-available GPU(s).
- Jobs are queued to partitions which, among others, define a priority of the job which could be one of *high priority, low priority, extra-low priority* having runtime time limits equal to, e.g., 1 hour, 24 hours, and 7 days respectively.
- Cluster nodes are accessible through cluster-local connection and host OS can connect to the container.

Note that hosting the application in the computational cluster is by no means necessary, any machine with a suitable, sufficiently powerful GPU card can substitute for it, but on the other hand, it should be emphasized that the intention to run the application in the cluster highly affected the overall architecture of the application (for some specific reasons described in the rest of this section).

Distributing the components among cluster nodes

Because the StyleGAN needs a GPU for both the inference (channels first format) and training, it must be served by one of the *gpu* nodes, while the rest of the application, especially the frontend part may run on an arbitrary node, preferably on some *cpu* node in order to prevent blocking of *gpu* nodes[1]. This kind of distribution is illustrated in Figure 4.2.

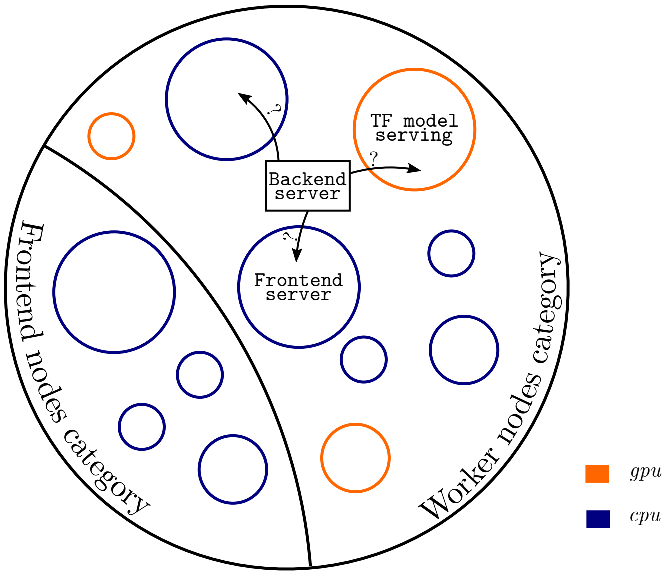


Figure 4.2: Illustrative distribution of application components among the cluster nodes.

In Figure 4.2, notice that the backend part of the application is not tightly bound with any node yet and there are three possible scenarios where it could run: on a separate *cpu* node, on a *cpu* node hosting the frontend server or on the *gpu* node where TensorFlow models will be served. Running backend on a separate *gpu* does not seem to make any sense. In the case that the backend server runs on the same *cpu* node as the frontend server does, these two servers could actually be merged into a single server that corresponds to a frontend server as used in common terminology. However, if they are not running on the same *cpu* node then there is a strict distinction between the frontend server and the backend server as described below.

Notice that there is a slight discrepancy between the terminology that is used in this thesis and commonly used terminology when it comes to the distinction between frontend and backend. The clarification of the terms and their meaning in the context of this thesis follows:

- The term *web UI* corresponds to a web user interface, i.e. it is more or less the same as the commonly used term frontend.
- The term *frontend server*[2] is more or less the same as the commonly used term backend. This means, that the frontend server is a server that is responsible for handling requests that are coming from the web UI.

[1]This decision assumes that there are more *cpu* nodes than *gpu* nodes or that demand for *gpu* nodes is higher.

[2]In this thesis also sometimes referred to as just a *frontend*.

- The term *backend server* corresponds to another server that is responsible for handling requests that are coming from a different source than directly from web UI (e.g. from the frontend server).

Cluster-specific difficulties

Running the application in the cluster brings some specific difficulties and it is a goal of this sub-section to briefly describe them and their possible solutions.

Because the cluster is managed by a resource manager, it is neither possible nor desirable to run any program on a worker node in another way than by queueing it as a job by using the resource manager (**D1**). This goes a hand in hand with another difficulty and that is, the fact that runtime of jobs is limited (see Section 4.1.1)(**D2**).

Another difficulty is that only the nodes from the frontend category are accessible from the internet, i.e., it is not possible to directly connect to the container running as a job on some worker node (**D3**).

With all these difficulties in mind, it is obvious that the application's architecture will need to be adjusted so that it matches these restrictions and this is the main purpose of the next Section 4.2.

4.2 Architecture revised

As was already mentioned above, hosting the application within the cluster brings some difficulties which will impact the architecture of the whole application.

To address difficulty **D1** both frontend server and backend server will need to run in containers which will be queued as jobs using the resource manager. This means, that once resources are ready for running the job, the container will start running and so will the server inside it. However, queuing these tasks manually would be rather impractical and this leads to a notion of some kind of *master node* or *master daemon* which will be running on a *cpu* node (with extra-low priority so it can be a long-running job) and will be responsible for interacting with the resource manager in order to queue jobs that will run either frontend server or backend server containers.

The master daemon could be utilized (or rather exploited) to solve the difficulty **D3** because running it on a node from the frontend category makes it accessible from the internet. Therefore the master daemon could tunnel or forward all the traffic to the frontend server running in the container. Furthermore, the master daemon could also help with solving the third difficulty **D2** by repeatedly checking if the appropriate containers are running and starting them if they are not. The communication between these components could be done by using any kind of APIs (gRPC, REST, etc.).

Running the master daemon on a node from the frontend category thus being outside of the worker nodes managed by the resource manager allows it to run for (almost) an arbitrary amount of time. The frontend container running on a *cpu* worker node can run for up to 7 days (under extra-low priority) which is more than sufficient (with automatically re-starting it once master daemon detects that the frontend container stopped running). However, the problem is with the container hosting the StyleGAN node which needs to run on *gpu* node, preferably under high priority. The reason for high priority is a belief that *gpu* nodes are generally highly utilized so running under low priority would cause frequent job suspends and waiting for resource allocation.

Note: It is important to mention that after dealing with all these issues it was decided to merge backend server with TensorFlow model serving and run them on a single node. This is supported by the fact that it turned out that the serving itself also needs some kind of API and that the originally intended backend did not offer that much functionally beyond simply calling serving API's, therefore it is preferred to trade flexibility for greater simplicity in this case (also because this application is just a proof of concept). The TensorFlow serving merged with the backend server will be collectively referred to as just *backend server* or *backend container*. This final architecture is illustrated in Figure 4.3.

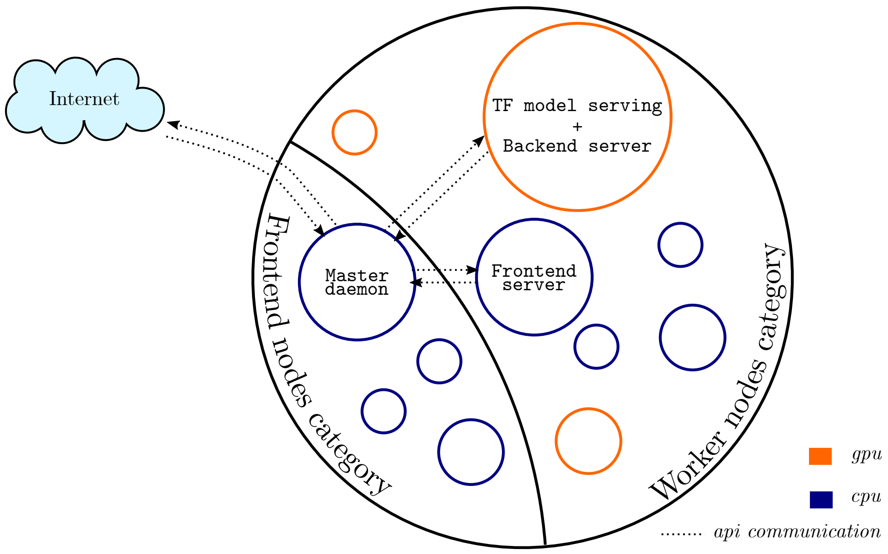


Figure 4.3: The final distribution of application components among the cluster nodes.

The easiest way of managing the runtime and computations of the backend server would be to think in terms of tasks—single units of computation—where each task would run as a single job[3] managed by the cluster resource manager and running the backend server (i.e., having a separate backend server for each task). This would greatly simplify the overall architecture because it would be possible to reuse the queue and scheduling mechanism from the resource manager. However, this simplification comes at a substantial price and there is a major drawback to this approach—the overhead connected with creating the job, waiting until it starts running (it takes a few seconds) and then waiting for container initialization, most notably, waiting until the code within the container loads the pre-trained StyleGAN model and until the TensorFlow itself gets loaded (GPU version takes some time to load, because of loading CUDA libraries). Overall, the overhead takes a vast majority of total runtime, e.g. when the time elapsed between queuing the job and completing the job was approximately 20 seconds,

[3]Note the distinction between *job* and *task*. Starting with this section, the job will refer to the job in the context of the cluster resource manager while the task will be a smaller unit of computation, i.e. single job could possibly perform several tasks.

the time needed for the prediction/inference itself was only about 1 second and the rest of the time was the overhead.

Therefore it seems beneficial to propose a different, more effective approach instead of creating a job for every task. The idea is that some functionality of the resource manager job queue could be reimplemented directly inside the backend container. This would allow the backend container to be running as a single job and multiple tasks would be queued and handled directly within the backend container in the single job, therefore sharing the total time overhead and reducing a per task computation time. The problem with this approach is that the jobs running on *gpu* nodes should not run for a very long time and shorter jobs are preferred to long-running jobs as was already described above.

Reusing a single backend container for multiple tasks brings great flexibility for building priorities and other extra rules on top of the custom task queue inside the backend container. For example, it would be useful to somehow decide a "right" number of concurrently running jobs (i.e. having several backend containers running) and then distribute the tasks between these jobs/containers so that the total balance is optimized. Also note, that each backend job will run on a different node which itself may have several CPUs and sometimes also several GPUs available so it would be possible to utilize the resources as well as possible. The idea is to create some task queue inside each backend container together with a queue of threads and assign the tasks to these threads. For a better intuition, the task could be, for example, *"generate 10 images"*. However, it should be emphasized that this is something that is out of the scope of this thesis and because this application serves only as proof of concept, this feature is not going to be implemented.

After deciding that there should be a single running backend container (job) for multiple tasks, it remains to decide when this container should be started. An extreme approach would be to have a single job running a backend container on a *gpu* node and make the master node ensure that there is always at least a single such running container. Then the tasks would be sent to this container and processed. The main issue with this simple approach is that it is wasteful to force the container to be forever running. Instead, it makes more sense to start the container on-demand. In this case, the demand might arise when the user starts interacting with a frontend. Therefore the idea could be formulated as *"run the backend container once the user starts interacting with the frontend and then keep it running until timeout (in order to make it simple) with a possibility to reuse the same job (running backend container) for future tasks"*. This adjustment brings a performance improvement in a form of reduced resource utilization (because the nodes are idle until the user starts interacting).

What might seem restricting is the 1-hour limit for high-priority jobs running on *gpu* nodes. However, it eventually turns out that the 1-hour limit is not that big problem, because a typical user interaction with the application will probably be shorter than 60 minutes. For the cases when the user interaction lasts longer than 60 minutes, the current runtime of the container should be monitored carefully and when it gets close to 60 minutes, the container should be restarted (cancel the job and start a new one) or do some kind of "double-job" approach by letting the current container running but starting sending all the tasks to the newly started container. The word phrase "close to the end" means that it should be done x seconds before reaching 1 hour where x is such a number that there never happens a situation that the backend job gets canceled during a computation. Because the most time-consuming task is fine-tuning which is limited to ≈ 10 minutes, it makes sense to choose something like $x = 11 \cdot 60$ seconds.

Another problem that should be solved is how to handle multiple users. Basically, there are two solutions to this problem, either give each user a single job with running backend container or try to schedule tasks from multiple users into a single job with running backend container. The sharing of the running backend container from the latter approach could eventually bring extra flexibility but only when assuming that there was some "smart" scheduling mechanism that would distribute the tasks between multiple running containers because using only a single backend container is problematic (consider, for example, the fine-tuning task which could take up to 10 minutes during which other users would not be able to do anything) and this is certainly something that is out of the scope of this thesis. Giving each user a separate running backend container brings a benefit that potentially tasks from several users could run concurrently (although this number is still limited and if there is a large number of users, some of the users would be stuck waiting for a running container job being executed, but still, there would be more parallel ways than with the previous approach where there was just a single, serial, way).

After considering all the observations from previous paragraphs together with their implications, it seems like a good compromise to give each user a single running backend container (within a single job) and then run multiple tasks (sequentially) for the user within this container.

The summary of the architecture of jobs and tasks is illustrated in Figure 4.4.

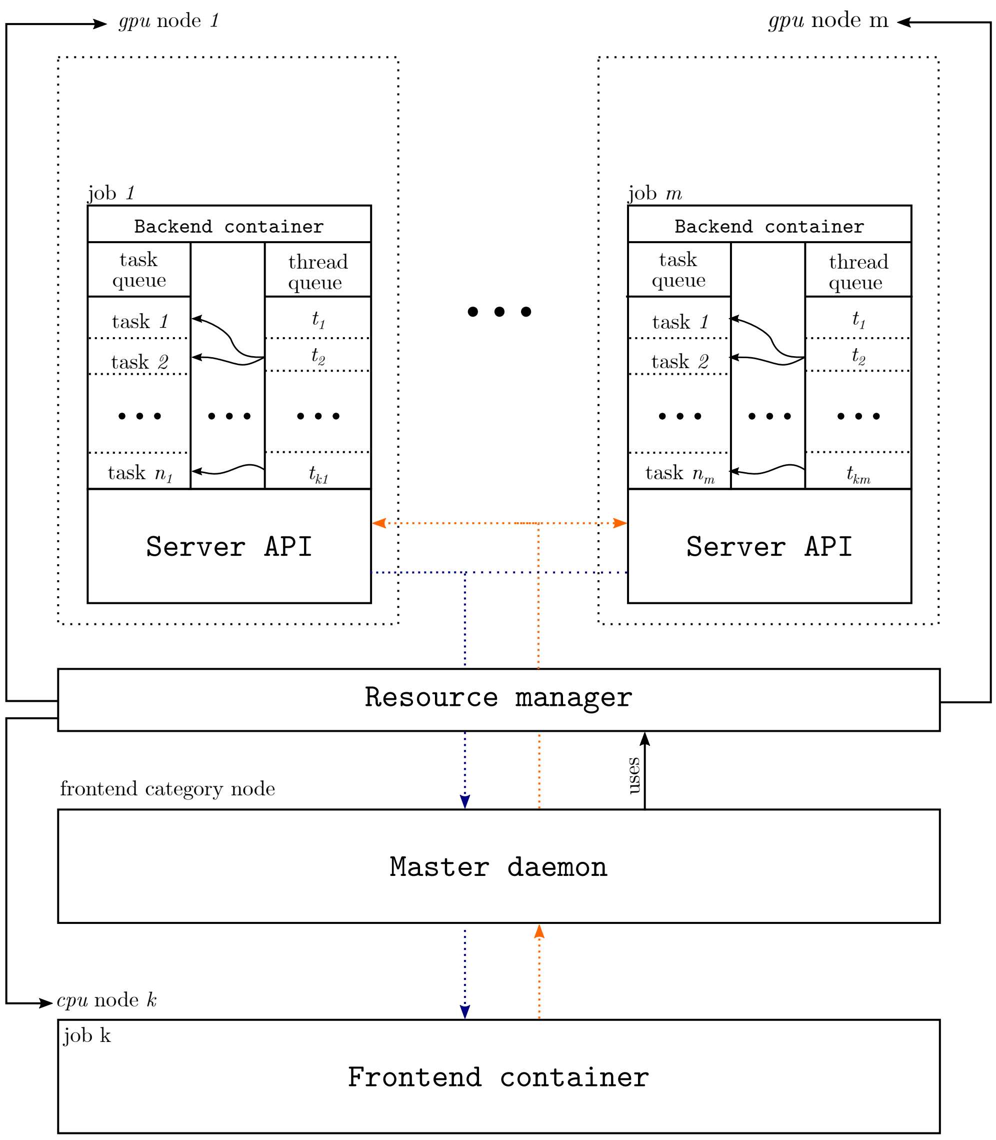


Figure 4.4: Distribution of tasks and jobs among the nodes. The master daemon is running on a frontend category node and it is responsible for running backend and frontend container jobs via interaction with the resource manager (which is directly responsible for their lifetime management). Apart from using the resource manager, the master daemon also tunnels API communication between the frontend server and backend server. When a task is created (upon a request from the frontend server) it is passed to the backend server corresponding to the user initiating the request. The backend server is running in a container on the *gpu* node and the tasks it receives are queued into a task queue. The tasks are popped from the task queue and assigned to available threads[4](one task for a single thread at a time).

Now, it remains to describe how the user can request some results, how to deliver the results back to the user, and overall, how will the communication between components work. Furthermore, the notion of "user" should be clarified. All of this information will be given in the following sub-sections.

[4]The current proof of concept version uses only a single thread for this purpose.

4.2.1 Notion of the user and security concerns

The above paragraphs frequently used a rather vague term of the user. It is vague because the application actually does not work with users in the right sense—like user accounts or something similar—because it would only make things more complicated. Instead, the user is represented by a single session ID which is generated when the frontend page gets loaded (new session every time it is reloaded). Note that the term session ID may have a slightly different meaning in the context of this thesis when compared to common terminology. The term user in the context of this thesis refers either to the user as a person or to this session ID. The session ID is stored in cookies in an encrypted form.

Both frontend and backend containers also have their own IDs assigned. For the backend container, the ID is the same as the session ID of the user to which the backend container is allocated. The frontend ID is generated every time a new frontend container starts running.

The frontend ID also serves as a kind of "protection"[5] from obtaining requests from outside of the local network, because every request that does something sensitive (e.g. queue job using resource manager) requires a correct frontend ID to be passed as arguments/data of the request.

Communication between components

It was decided that all the components will communicate via a network using a simple REST-like API where each component will have a set of defined endpoints and it could send requests to endpoints of other components to obtain some results (as a form of response).

The idea is that the user will request the results (somehow) and then this request will transmit all the way to the backend container where it is processed, results are generated by invoking the TensorFlow model and then should be sent back to the user. The results could be either sent directly to the frontend or by first sending them into a master daemon and then to the frontend. Even though the former approach seems more efficient, the latter approach seems more flexible and therefore it will be preferred. However, there is a problem with this idea, namely that it is not that easy to send the results back in the response, because the computation of the results (task) on the backend side (corresponds to invoking the TensorFlow model for inference/prediction) may take some time (remember the fine-tuning approach which could take up to 10 minutes) therefore making it very probable for requests to timeout. One possible approach to tackle this issue is to return the task ID instead of the results itself and define another set of endpoints that could be used for querying the state of the given task or the result of that task if it is available. Another approach could be—instead of actively querying the backend server for results of the task—to let the backend server notify the frontend once the task is done which seems like a better fit for the purposes of this application. Once the backend server notifies the frontend (with the master daemon in the middle) about task completion, the frontend

[5]However, it is important to state that this application, by no means does not try to solve all the security challenges as it is out of the scope of this thesis and this application serves only as a proof of concept.

could serve this information to the user. The communication of the components illustrated on an example of requesting some results is shown in Figure 4.5.

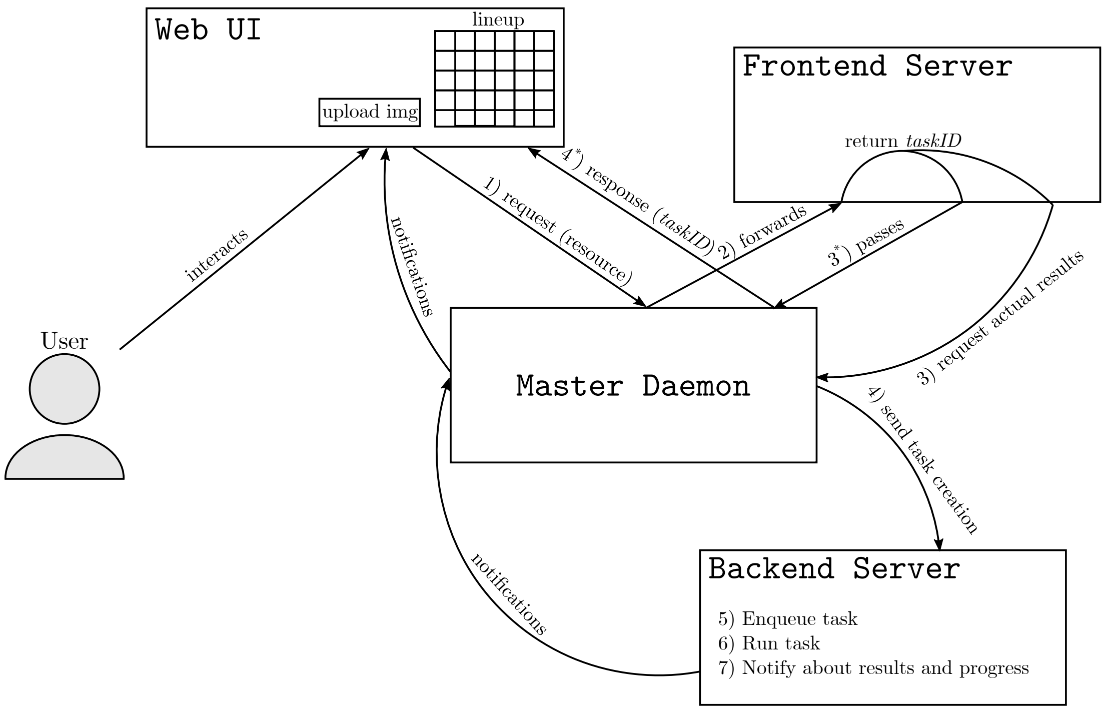


Figure 4.5: Communication between the components.

As can be seen in Figure 4.5 a user's interaction with web UI may lead to sending a request to the master daemon which is then forwarded to the frontend server. Notice, that once the request arrives at the frontend server it is passed through the master daemon (steps 3, 4) down to the backend server where the actual task is created, queued, and eventually executed (steps 5, 6). The ID of this task is then returned to the user (step 3*). While the task is running, the backend server sends notifications about its progress and/or results (step 7).

4.3 Implementation

This section very briefly summarizes some of the implementation details of the application together with used technologies and a sketchy look at the APIs. Note that the implementation is a proof-of-concept only and several additions or improvements will be necessary to make it into a production-ready state.

4.3.1 Used technologies

All the server parts (frontend server, backend server, and master daemon) are running the Flask[6] web framework which was chosen mainly because it is very simple for writing a set of API endpoints and it is built for Python which is ideal due to the fact that the backend server will need to load TensorFlow models (and all TensorFlow code in this thesis is written in Python). Furthermore, the

[6]See Flask website [84] for more details about the framework.

requests library is used for sending requests and the subprocess module is used for executing OS commands (related to the resource manager).

The web UI part is implemented in Javascript using BootstrapVue[7] framework which was chosen mainly because it provides a lot of functionality and built-in responsible components (pagination, popovers, progress bar, and so on), making it quite fast to prototype the UI part of the application without having to use too many additional dependencies (packages or libraries).

4.3.2 Master daemon

Master daemon node defines a set of endpoints which are intended to be invoked from frontend server—e.g. `start-of-interaction` which ensures that the user gets assigned a running backend container—or from the backend container, e.g. `on-ready` that notifies that the backend container is running and ready. It also defines a "catch-all" endpoint that simply forwards all[8] the requests to the frontend server (so that it is accessible from the internet). Master daemon is also responsible for maintaining a single running instance of the frontend container.

Master daemon also defines endpoints[9] that behave as a callback for obtaining notifications (events) from the backend container (e.g., `on-result` and `on-progress` which are responsible for handling result and progress notifications from the current task respectively.). In order to silently forward this information directly to the web UI, Javascript Event Source[10] is used.

As it was already described in sub-section 4.2.1, the user is identified by a session ID which changes every time page is reloaded. Remember that each backend container is paired with the user (via session ID) and that it is started upon the user interacting with the frontend. Specifically, this would mean that every time the user reloads the page, a new backend container would be started which is truly something undesirable. To prevent this behavior, the master daemon tries to behave "smart" by first looking at the "old" session ID (before assigning the new one upon page reload) and if it has a backend container paired then this backend container will be reused for the new session ID (i.e., no new container will be started).

4.3.3 Job management

Job management is a module that provides job management-related functionality, e.g. running jobs and some abstraction over tasks (remember that a task is a single unit of computation while the job is a job used by the cluster resource manager, i.e. some running container in this case). Tasks API is a wrapper above resource manager API (in the context of a particular cluster that was used, the resource manager is SLURM), i.e. it allows to start the backend and to send tasks to the backend. Each task is paired with a task ID and when some task is started, its task ID is returned back to the user.

[7]See BootstrapVue website [85] for more details.

[8]Unless there exists a more specific endpoint matching the request.

[9]It would be better to define them in the frontend server but their forwarding would be quite tricky to implement.

[10]For more information, see the following website. [86]

4.3.4 Backend server

The backend server acts as a TensorFlow model serving with performing some additional duties. It defines a set of endpoints responsible for invoking loaded, pre-trained models in a particular way to produce results. Because the invocation and production of the results may take some time, the backend server also posts "events" to the web UI (via master daemon). The two events are `on-result`[11] and `on-progress` whose callbacks were already mentioned above. It is worth mentioning that the payload of the result event contains the names of the result files (images in this case) and the payload of the progress event is simply a percentage of the current task's progress. These files are accessible because they are located in a mounted folder[12] that is accessible from within the whole cluster. The backend server also notifies the master daemon when it is ready and running (after loading pre-trained models).

4.3.5 Frontend server

The frontend server is tied with the web UI and provides data for the web UI to display by defining a set of endpoints which are called from the web UI (Javascript). An example of requests originating from the web UI which the frontend server handles are file uploads. It also defines various endpoints which forward the request further to the master daemon (which then forwards it to the backend container, e.g. request for image generation).

4.3.6 Web UI

The web UI part is implemented in JavaScript and it uses Fetch API[13] to send requests to the frontend server (via master daemon). It also listens for various events like `on-ready` using Event Source and then updates UI based on these events, for example, the user is only able to generate results when the backend container is running, otherwise, the button for it is disabled.

WebUI also handles file saving (using FileSaver.js[14] library), saving individual filler images as a zip file (using JsZip[15]), and generating an image of the whole lineup (using Canvas).

[11]The names `on-result` and `on-progress` are heavily overloaded because they are used at various levels, namely: as callback names and as event names (at web UI part, backend part and frontend part).

[12]Although it is surely not ideal, it is sufficient as a proof of concept.

[13]See the following website [87] for more details about Fetch API.

[14]Visit the following website [88] for more details.

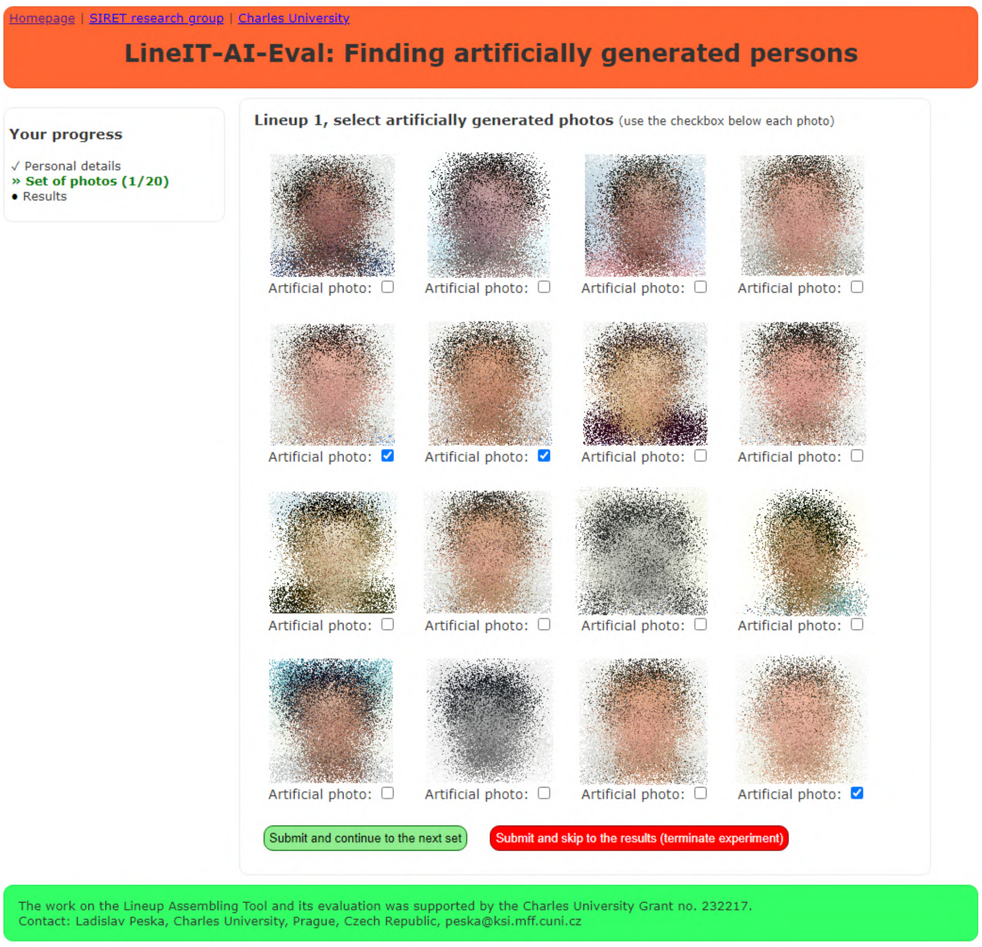
[15]Visit the project website [89] for more details.

5. Evaluation

This chapter summarizes the results from all the individual parts of this thesis. First, the results of the generator are evaluated on a user study presented in Section 5.1. Second, the Section 5.2 briefly summarizes the results of the two attribute prediction models. Finally, the results of the encoder together with the results of the entire architecture—after building the individual components together—are presented in Section 5.3. Only the results of the final model variants are presented.

5.1 Generator's results

For evaluating the quality of images produced by the generator, a user study was constructed to measure to which extent do the users (or participants in this case) consider these images as artificial. The participants of the study were asked to go through several pages of images and select all the photographs they think do not show a real person. The website with the user study is shown in Figure 5.1. A more detailed description of the construction of these pages of images follows.


Homepage | SIRET research group | Charles University

LineIT-AI-Eval: Finding artificially generated persons

Your progress

✓ Personal details
» Set of photos (1/20)
● Results

Lineup 1, select artificially generated photos (use the checkbox below each photo)

The work on the Lineup Assembling Tool and its evaluation was supported by the Charles University Grant no. 232217.
Contact: Ladislav Peska, Charles University, Prague, Czech Republic, peska@ksi.mff.cuni.cz
Artificial photo:
Submit and continue to the next set
Submit and skip to the results (terminate experiment)

Figure 5.1: Website with the user study. The noise was added intentionally to the images for the sake of anonymization.

5.1.1 Evaluation protocol

In this study, there were two databases of images, namely: D_{real} and D_{fake} consisting of real person images and generated images respectively. The D_{real} database contains a sample of 8000 images taken from the *Faces* dataset, while the D_{fake} contains 100000 images generated by the generator. Note that the D_{fake} contains random samples, so there was no additional filtering of the generated images. Especially, this means that there could be images of rather low quality as those that are shown in Figure 5.2.

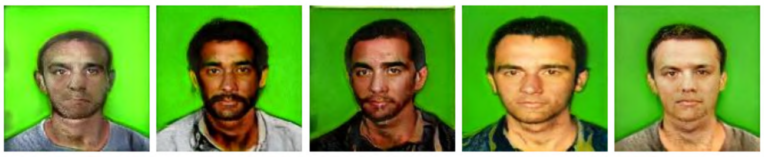


Figure 5.2: Low quality images in the D_{fake} dataset.

Even though images with such a low quality are present in the dataset, their frequency is rather low (at most a few hundred out of 100000 images). Furthermore, these low-quality images could be easily filtered out by considering only images with a discriminator score above some threshold which leads to results shown in Figure 5.3.

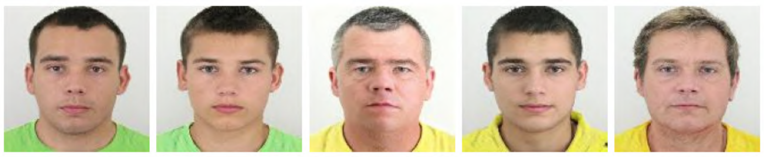


Figure 5.3: High quality images in the D_{fake} dataset.

Figure 5.3 shows the top 5 images from the generated dataset based on the discriminator score.

In the user study, each participant has been shown 20 pages (lineups) of images where each of them was consisting of 16 face images. In each lineup, one of the images was the seed image (corresponding to a suspect in the real scenario) and the remaining 15 images were fillers. The fillers were of two types: generated person fillers taken from D_{fake} and real person fillers taken from D_{real}. The number of generated fillers was chosen randomly as an integer from interval $n_{fake} = [5, 10]$. The rest of fillers ($n_{real} = 15 - n$) were real persons.

Given the seed image, a sparse matrix containing similarities with both generated and real images was generated. For both D_{real} and D_{fake}, 500 most similar images was considered out of which n_{real} and n_{fake} images was sampled respectively. The reason for sampling these n_{real} and n_{fake} images instead of just taking top n_{real} and n_{fake} is to prevent choosing images showing persons who are too similar to each other.

The similarity of the two images was measured by cosine similarity between feature vectors of these images. The feature vectors were outputs of the last but one layer taken from a pre-trained VGG-Face model. Although this worked quite well for the images from D_{real}, this was quite problematic for the images taken from the D_{fake} because the similarity could be small for images of bad quality. The aforementioned issue with simple cosine similarity of feature vectors (will be referred to as *sim*) has lead to adding two more approaches: *simDisc* and *simSimDisc*. The *simDisc* approach deals with similarities multiplied by discriminator score (how much the discriminator thinks that the image is real) with an intent to take image quality into account. Similarly, the *simSimDisc* approach works by using $sim \cdot sim \cdot Disc$ for similarity in order to mitigate an issue of selecting images that are too uniform (this was an issue of *simDisc* approach because very similar images usually had very similar, possibly high, discriminator scores). The aforementioned effect of choosing images with similar discriminator scores is illustrated in Figure 5.4. Finally, each of the lineups was generated in three variants by using all these three similarity approaches, but only one of them (selected at random) was eventually displayed to the user.

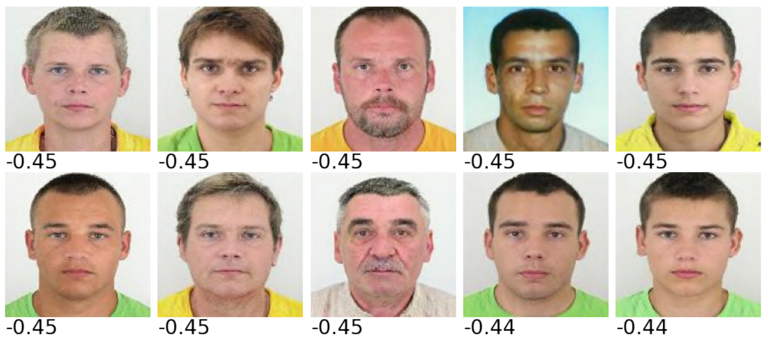


Figure 5.4: Images in the D_{fake} dataset annotated by their discriminator scores. Notice, that all these images have very similar discriminator scores and they tend to depict persons of a very similar appearance, clothes, or image background.

5.1.2 Evaluation results

In total, 81 respondents participated in the survey and completed at least two pages of the survey. The participants were annotated by their gender, age group, education, ethnic group, and a special flag called "ML expert" which symbolizes whether the given user has some experience with machine learning or not. The histograms of these categories are shown below in Figures 5.5, 5.6, 5.7, and 5.8. The results in tables throughout this section were rounded to three decimal places.

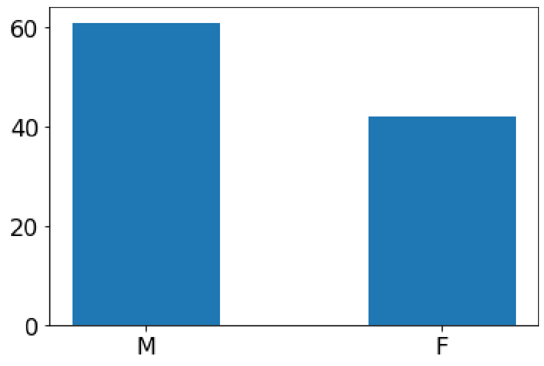


Figure 5.5: Gender

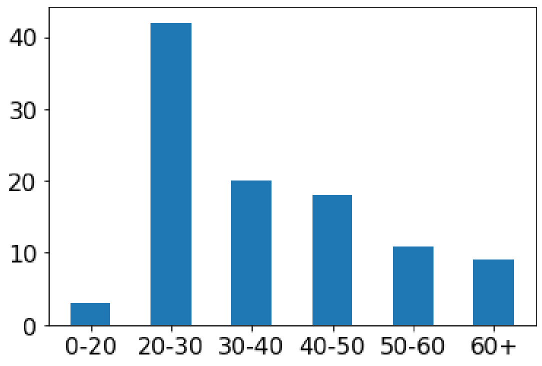


Figure 5.6: Age groups

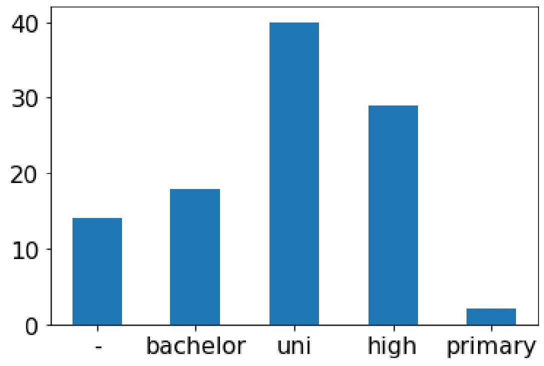


Figure 5.7: Education groups

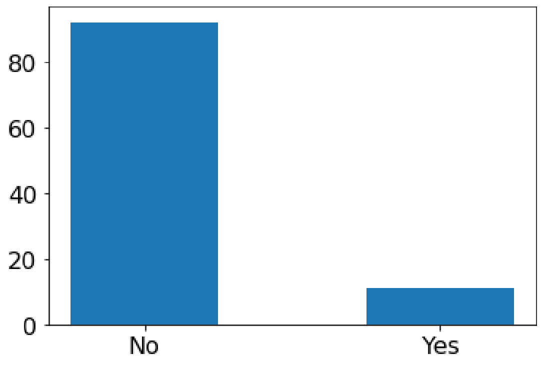


Figure 5.8: ML experience

Summary of the results including accuracy, precision, and recall averaged over all scenarios are shown in Table 5.1.

Users	**Lineups**	**Images**	**Fake images**	**Identified fake images**	**Real images**	**Accuracy**	**Precision**	**Recall**
81	1516	24 256	12 296	4 801	11 960	0.599	0.650	0.389

Table 5.1: Average accuracy, precision, and recall of the user study participants.

From Table 5.1 it can be seen that the accuracy is around 60 %, therefore, showing that the users were slightly better at detecting fake images than guessing it on random. Furthermore notice, that the precision is much larger than the recall (65 % versus 39 %) which means that most of the images users have selected as fake were really fake. On the other hand, the low recall shows that a lot of fake images escaped from participants' attention.

What is interesting is the fact that there were quite high deviations between the individual users as can be seen from Table 5.2.

User	Images	Fake images	Identified fake images	Real images	Identified real images	Accuracy	Precision	Recall
0	320	161	4	159	117	0.378	0.087	0.025
1	320	167	5	153	126	0.409	0.156	0.030
80	320	171	137	149	136	0.853	0.913	0.801
81	320	172	142	148	113	0.797	0.802	0.826

Table 5.2: Results for two participants with the highest score and two participants with the lowest score.

Table 5.2 shows results for four respondents sorted by recall from lowest to highest. First, two rows are respondents with lower accuracy and the last two are those with the highest. From these results, it seems that the differences between the users are tremendous. All these four users were males, what is interesting is that those two with the highest recall were older (age groups 50, 60+ versus 40) and had university education group whereas those with the lowest recall had high school education group. Nonetheless, the results should not be taken very seriously because the sample is too small.

The results comparing different similarity approaches during filler selection are shown in Table 5.3.

Approach	Images	Fake images	Identified fake images	Real images	Identified real images	Accuracy	Precision	Recall
sim	8048	3794	1277	4254	3389	0.580	0.596	0.337
simDisc	8480	4761	2000	3719	3092	0.600	0.761	0.420
simSimDisc	7728	3741	1524	3987	3256	0.619	0.676	0.407

Table 5.3: Results for the individual similarity approaches.

As can be seen from Table 5.3 the *sim* similarity approach slightly outperforms the other two approaches in all three measures (accuracy, precision, and recall).

The performance was also compared between early and late performed selections. For this reason, the 20 lineups per user were indexed by IDs $1, \ldots, 20$, and then the results over all the users were taken and grouped by this ID. In other words, the lineups were indexed by (page) numbers corresponding to the order in which they were presented to the user. The accuracies for each individual ID are shown in Figure 5.9.

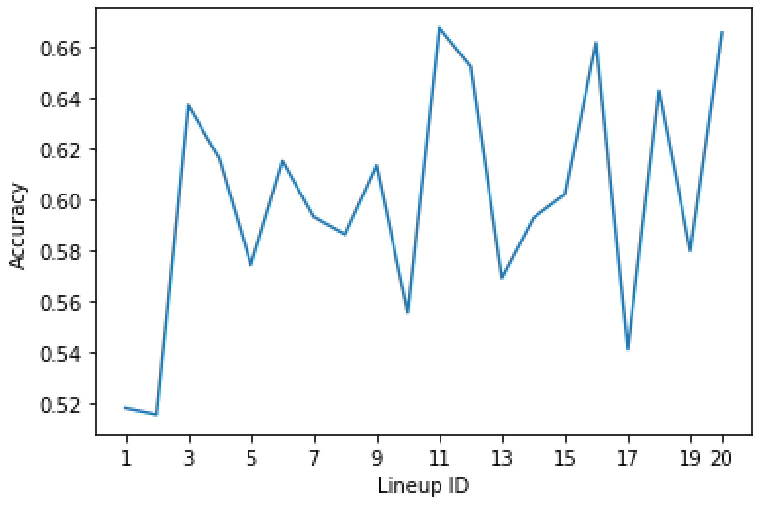


Figure 5.9: Accuracies for each lineup ID.

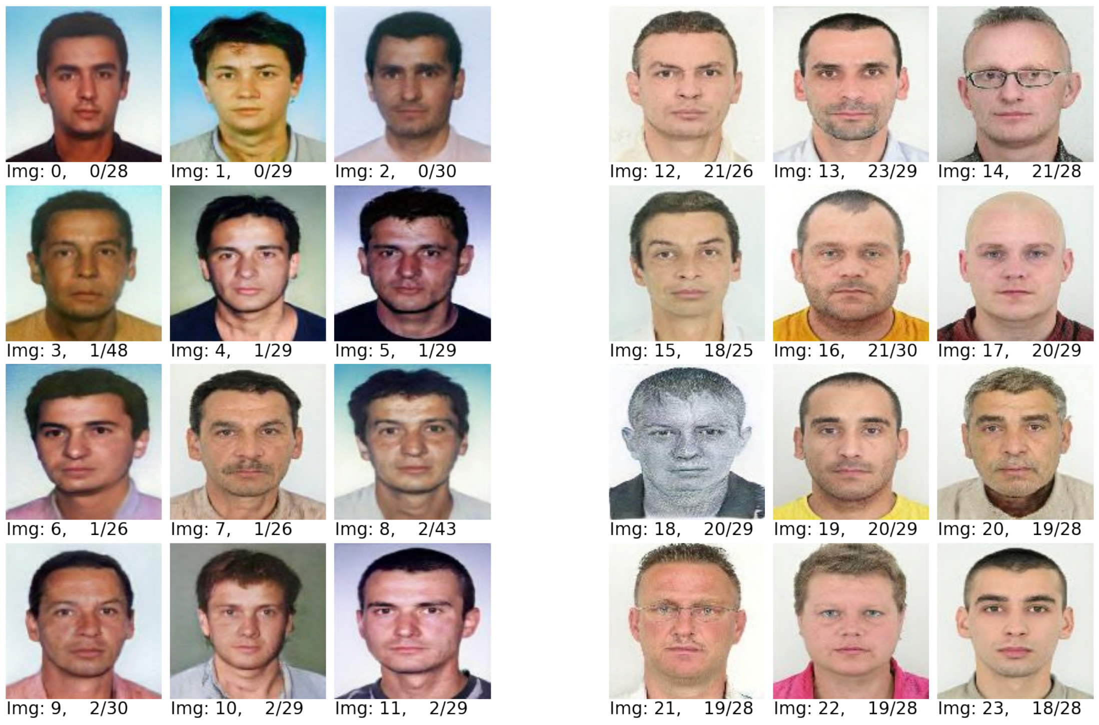
As can be seen from Figure 5.9 the accuracies seem to be changing at random, suggesting that the users did not learn how to identify fake images during the experiment, that is, there is no significant difference between early and later performed selections. Note that the order of lineups was the same for all the users (e.g. lineup with ID 5 always had the same seed image) therefore it seems that the difference in the difficulty of the individual lineups completely hidden any effect of learning in time.

Furthermore, there was no significant difference based on the gender of the participants. Based on participants' age, it seems that users below the age of 30 performed slightly better than those aged above 30.

Comparison of results based on whether the user has or has not some experience with machine learning did not reveal anything special. Although the users with machine learning experience seem to be performing slightly better, the difference between these two groups was not significant. However, this could also be caused by the fact that the number of participants with machine learning experience was rather low.

Results grouped by education group or ethnicity were not analyzed because the amount of data was not high enough for drawing any meaningful conclusions.

Finally, the images were inspected in greater detail with an intention to reveal which images are often correctly identified as fake, which are usually not correctly identified, and also what are the most significant differences between these two groups. Interestingly, some of the images were never identified correctly by the users while other images were almost always identified. Below in Figure, 5.10 are two plots—left and right—showing images that were least frequently and most frequently identified—as artificially generated—by the user. The threshold on how many times each image in the figure was displayed to users (lower bound) was set to 25. Every image in this figure is annotated by image number, the number of times the person was identified, and the total number of times the person was shown, written in the form $n_{identified}/n_{shown}$.



(a) Least frequently identified (b) Most frequently identified

Figure 5.10: Comparison of most frequently and least frequently identified images.

Notice that for the most frequently identified images from the figure above, it seems that they often show some small defects. For example, the glasses shown in images 14 and 21 seem very artificial and unrealistic which may be a hint to the respondents that these images were generated. Some of the persons were having an overall weird appearance, e.g. 18, 22 (which had some minor deformations, generally this was rather rare in the dataset). The last defect were asymmetric eyes where either one eye was looking in a different direction than the other or the eyes were of a slightly different size/color. Some minor eye defects could be observed in images 14 and 21. Another defect that was present in some of the generated images not displayed in Figure 5.10 were weird-looking hair.

The results of the least frequently identified images from left side of Figure 5.10 are rather interesting. What can be seen from these results is that almost all the images had slightly blue or somehow else colored backgrounds, instead of pure white or very bright backgrounds which were prevalent in mostly identified images. The reason that these images were not frequently identified could be the fact that these background effects were often a case with real, physical passports or when such documents were scanned thereby successfully confusing the respondents. It also seems that these images were not as sharp or clear as those from the easily identifiable ones from the right side of Figure 5.10.

To summarize the user study, it seems that the generated images have quite a good quality because the participants were not much better at detecting the fake images than if they would only be guessing them. Moreover, the low recall suggests that if the participants are shown a large enough sample of images then there will be enough fake images that will not get revealed by the participants.

Based on the additional feedback from some of the participants, it seems that they usually mark as fake those images, that are too uniform (e.g. no imperfections), too similar to each other (because real persons are usually not so similar), or show a loss of detail in some parts (e.g. loss of detail in hair, a weird look of ears or some disproportion of the eyes).

5.2 Attribute prediction models' results

The performance of the attribute prediction models was measured by means of their recall, precision, and accuracy. Because both FairFace and CelebA models work in multi-label settings, these metrics were evaluated for each of the labels separately. The performance of the final CelebA model, evaluated on the development dataset is shown in Table 5.4. Note that all the results in this section were rounded to three decimal places.

Attribute	Accuracy	Precision	Recall
5_o_Clock_Shadow	0.932	0.623	0.840
Bags_Under_Eyes	0.843	0.610	0.624
Bald	0.989	0.680	0.851
Bangs	0.956	0.816	0.920
Big_Lips	0.725	0.585	0.355
Big_Nose	0.826	0.587	0.638
Black_Hair	0.898	0.790	0.839
Blond_Hair	0.952	0.776	0.907
Brown_Hair	0.875	0.635	0.780
Chubby	0.944	0.491	0.699
Double_Chin	0.955	0.504	0.660
Goatee	0.965	0.600	0.880
Gray_Hair	0.978	0.633	0.812
Male	0.984	0.981	0.977
Mustache	0.964	0.531	0.670
Narrow_Eyes	0.864	0.514	0.525
No_Beard	0.953	0.986	0.960
Oval_Face	0.742	0.567	0.511
Pointy_Nose	0.753	0.570	0.539
Receding_Hairline	0.928	0.555	0.707
Sideburns	0.972	0.677	0.848
Straight_Hair	0.842	0.621	0.633
Wavy_Hair	0.859	0.847	0.734

Table 5.4: Accuracies, precisions, and recalls of the individual attributes evaluated on 128×128 CelebA dataset.

As it was already mentioned in Chapter 3 this model was designed in such a way that a reasonable recall is achieved. Otherwise, it would be very easy to achieve very high accuracy with very high precision but an extremely low recall. This is probably due to the fact that most of the attributes are present rarely in the data and therefore it is difficult for the model to learn[1]. To better

[1]However, this is not true for all the attributes, see e.g. *Bald* which has quite high recall

quantify this phenomenon, see Figure 5.11 which shows the number of positive examples of each of the individual attributes within the whole CelebA dataset. Another possible problem why some attributes are harder to learn might be the low resolution of the images (128×128 pixels) which simply makes it difficult for some of the attributes to be clearly visible (e.g. *Big Lips*).

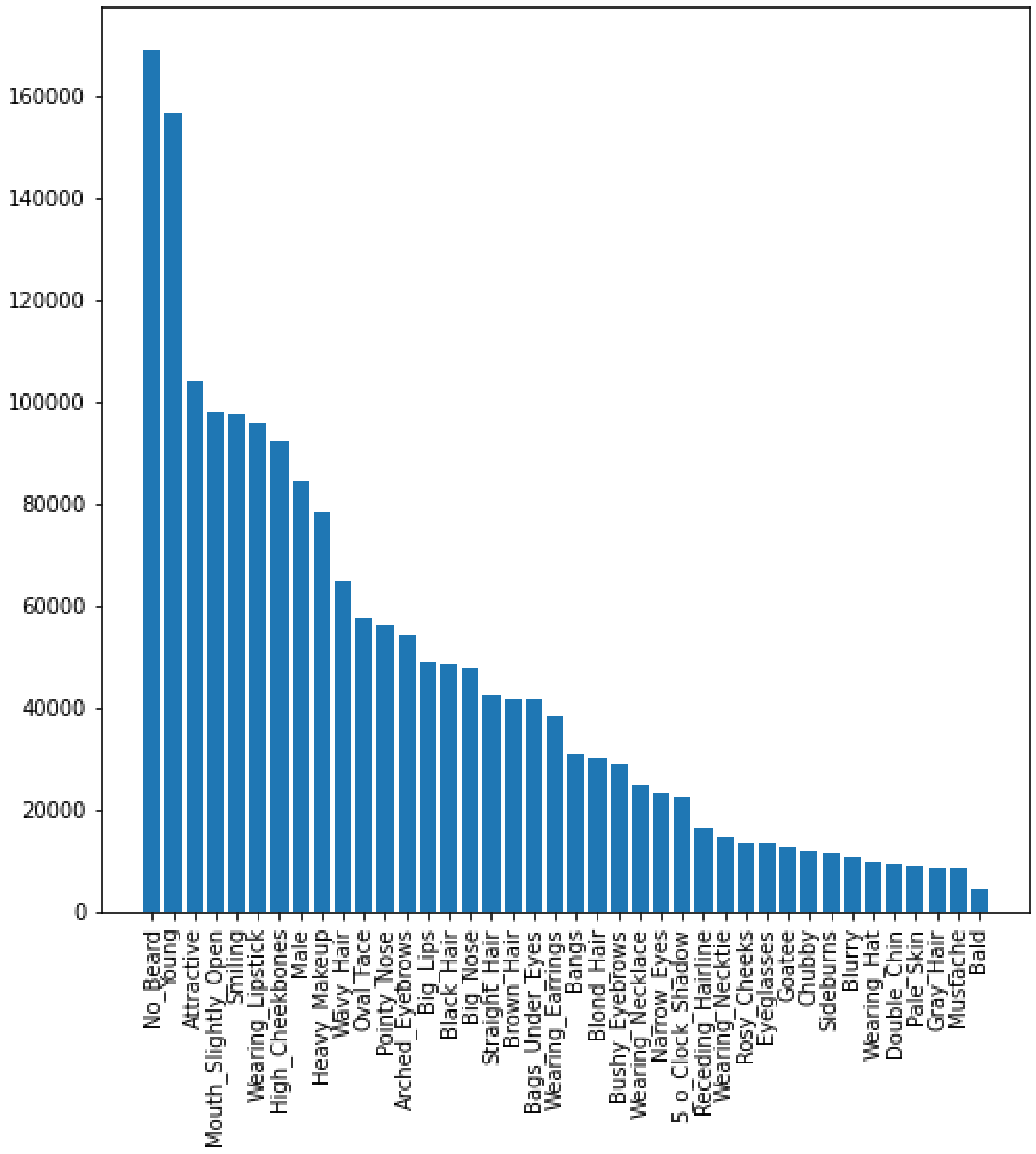


Figure 5.11: Number of positive examples for each of the attributes. Total number of images is around 200000.

albeit being very rare in the data.

The FairFace model was evaluated in a similar way as the CelebA model and the results for the ethnicity attributes can be seen in Table 5.5 while results for the age attributes are shown in Table 5.6.

Ethnicity group	**Accuracy**	**Precision**	**Recall**
Black	0.960	0.856	0.857
East Asian	0.924	0.706	0.799
Indian	0.931	0.770	0.714
Latino Hispanic	0.867	0.556	0.493
Middle Eastern	0.906	0.562	0.679
Southeast Asian	0.907	0.659	0.588
White	0.900	0.738	0.736

Table 5.5: Accuracies, precisions, and recalls of the individual ethnicity groups evaluated on 128×128 FairFace dataset.

Age group	**Accuracy**	**Precision**	**Recall**
$0 - 2$	0.992	0.753	0.814
$3 - 9$	0.952	0.795	0.827
$10 - 19$	0.898	0.531	0.453
$20 - 29$	0.796	0.647	0.712
$30 - 39$	0.786	0.496	0.440
$40 - 49$	0.867	0.461	0.449
$50 - 59$	0.921	0.456	0.447
$60 - 69$	0.964	0.411	0.539
70+	0.990	0.515	0.424

Table 5.6: Accuracies, precisions, and recalls of the individual age groups evaluated on 128×128 FairFace dataset.

To gain a better interpretation of the miss-classifications, Figure 5.12 and Figure 5.13 show confusion matrices for age and ethnicity respectively, both evaluated on the development portion of the FairFace dataset.

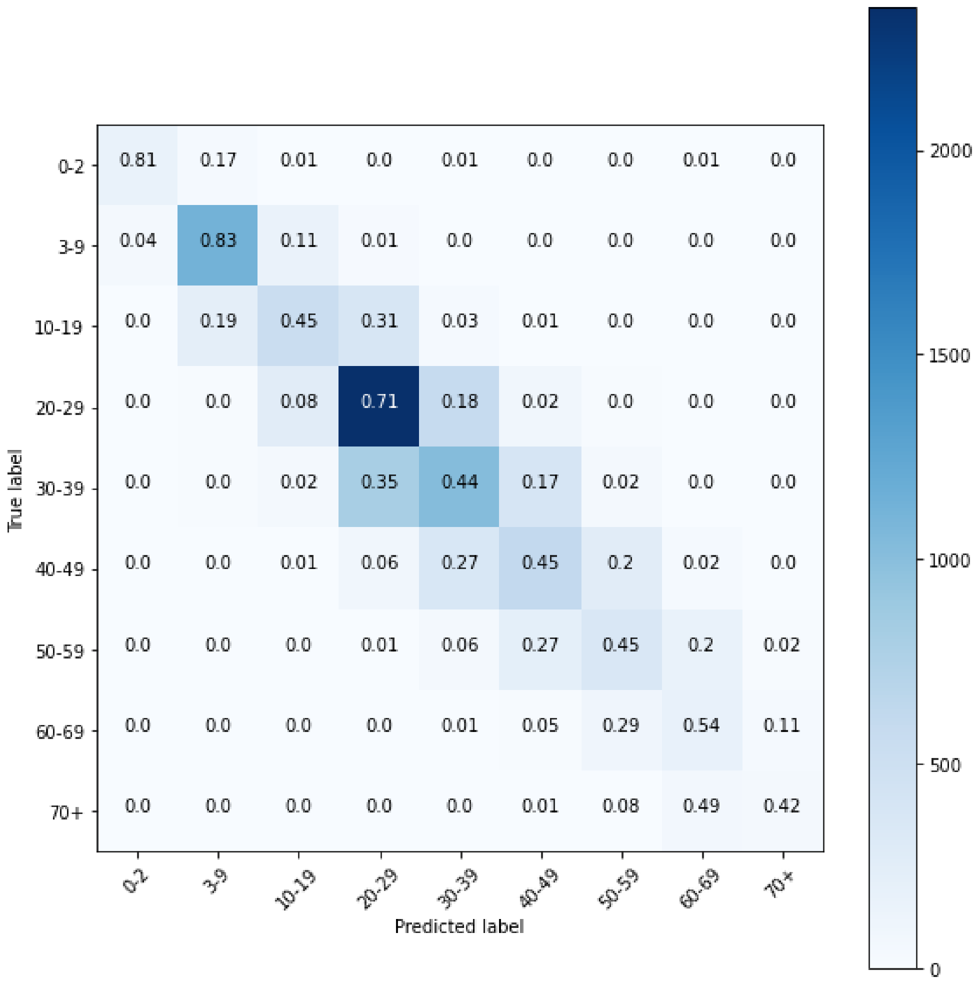


Figure 5.12: Age confusion matrix (on the development dataset).

Figure 5.12 shows that miss-classifications happen almost exclusively in neighboring classes. This observation is positive because it means that the model does not make serious mistakes. Furthermore, these miss-classifications could be caused by the age groups being too coarse making it especially difficult to classify at-border cases (e.g. someone aged 30 might be confused with someone aged 29, etc.). In the same way, as the model makes these little mistakes, people often also have problems classifying the age of an individual with reasonable precision, therefore the model's results seem to be aligned with the way how people perceive age.

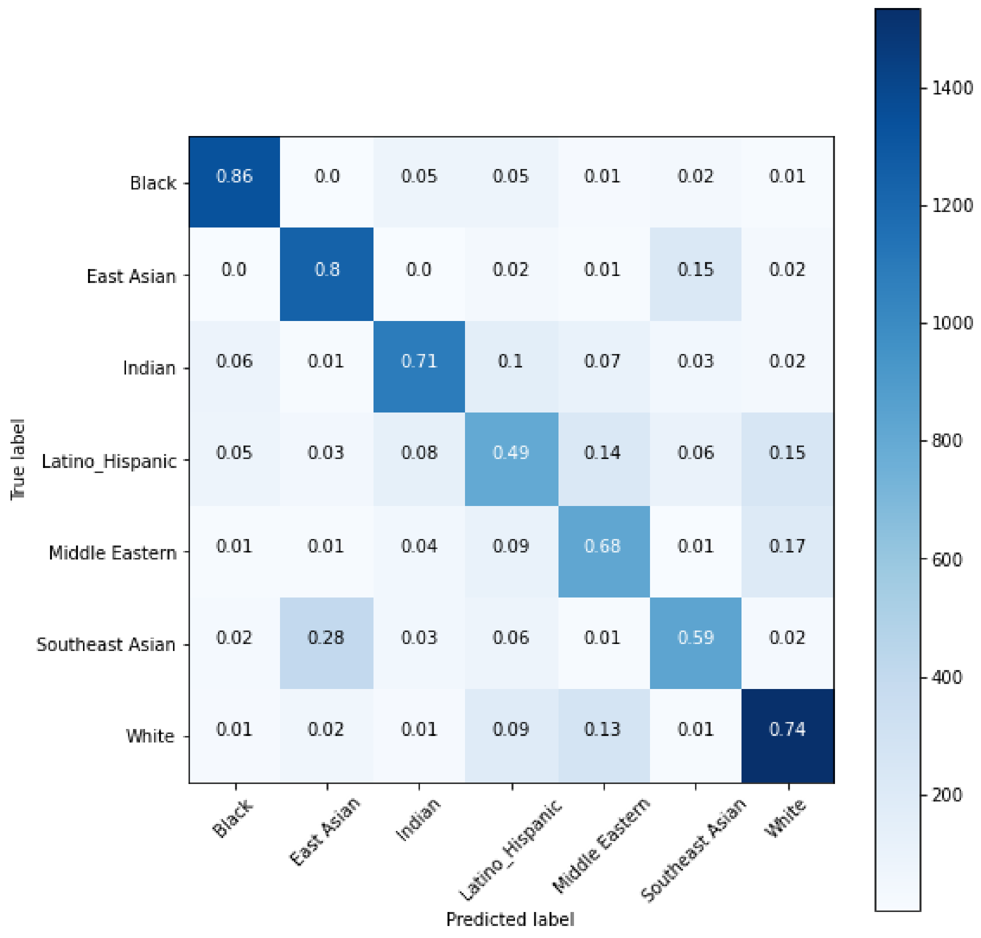


Figure 5.13: Ethnicity confusion matrix (on the development dataset).

From Figure 5.13, the following observations could be seen:

- Model sometimes miss-classifies *White* with *Middle Eastern* and vice versa.
- Model sometimes miss-classifies *Southeast Asian* with *East Asian* and vice versa, but the former misclassification is more frequent than the latter.
- Model sometimes miss-classifies *White* with *Latino Hispanic* and vice versa, the latter more frequently than the former.

It is not that surprising that these miss-classifications happen and similarly to the age miss-classifications, they are not that serious. For example, differentiating between *East Asian* and *Southeast Asian* is quite a challenging task even for people. Similarly differentiating between *White* and *Middle Eastern/Latino Hispanic* is also quite difficult on its own without mentioning that on some photographs from the dataset the assignment of these classes is rather ambiguous. The important observation is, again, that the model does not perform any serious mistakes by miss-classifying e.g. *White* with *South/East Asian* or *Black* with *South/East Asian*, etc. What might seem interesting is that the miss-classification between *East Asian* and *Southeast Asian* is far from being symmetric. One possible reason for this could be the fact that *East Asian* is actually more present in the training data[2] than *Southeast Asian* is, therefore the model might tend to classify into this class more frequently. The distribution of the individual ethnicities is shown in Figure 5.14 while the distribution of the age groups is shown in Figure 5.15.

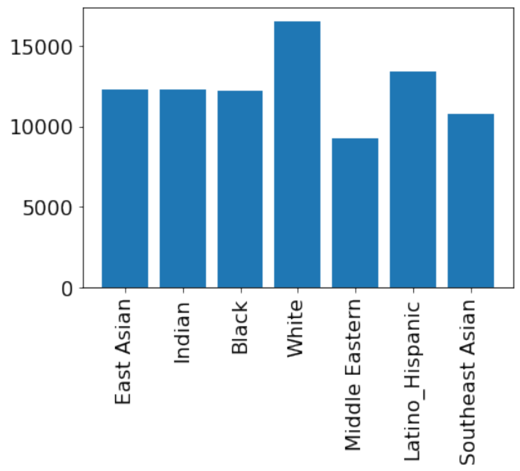


Figure 5.14: Distribution of ethnicity groups.

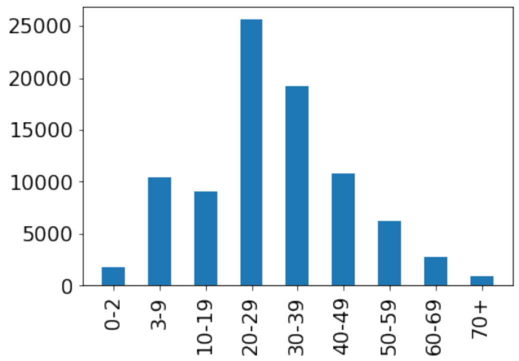


Figure 5.15: Distribution of age groups.

As opposed to the ethnicities, the age groups are not very balanced in the dataset. Nevertheless, the model is still able to perform quite well even on the very rare groups of $0-2$ and $70+$ maybe because these two classes are in some sense outliers (they contain very old and very young people who are very different from anyone else in the dataset).

Finally, both CelebA and FairFace models were used to analyze attribute distribution on the *Faces* dataset and also on the dataset consisting of the generated dataset. For this purpose, a sample of 100 000 generated images was taken and attributes for each image were predicted.

Once these images were annotated by attributes, the dataset could be queried for images having a certain attribute. Consult the Figures 5.16, 5.17, 5.18, 5.19, 5.20 and 5.21 for examples.

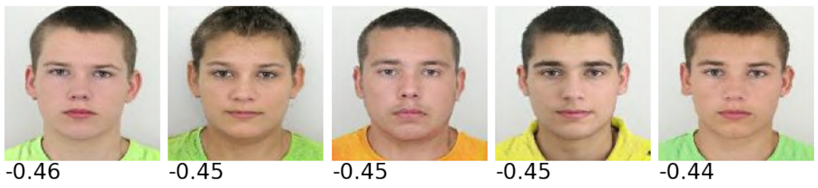


Figure 5.16: Generated images annotated by $10-19$ age group.

Figure 5.16 shows 5 images of persons whose age was predicted to be in $10-19$ image group. These 5 images are the ones with the highest discriminator score out of all images from the same age group. These results seem reasonable because all these people could possibly be aged between 10 and 19 years old.

[2]Similar ratio of these two classes is also present in the development portion of the dataset.

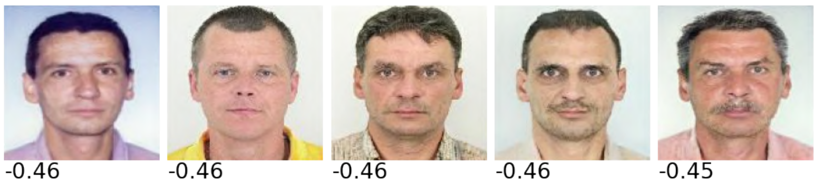


Figure 5.17: Generated images annotated by $50 - 59$ age group.

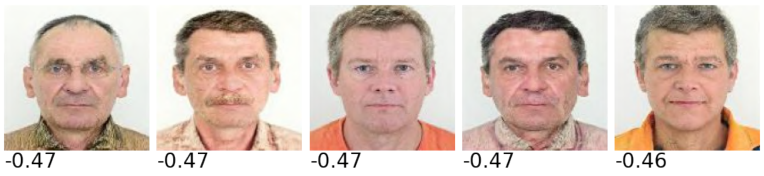


Figure 5.18: Generated images annotated by 70+ age group.

Figures 5.17 and 5.18 show people whose age was predicted to be in the $50 - 59$ and 70+ age group respectively. For some of these images, the age could be subjective or controversial, but it truly seems that the people from 70+ group look older than those who were assigned to the $50 - 59$ group.

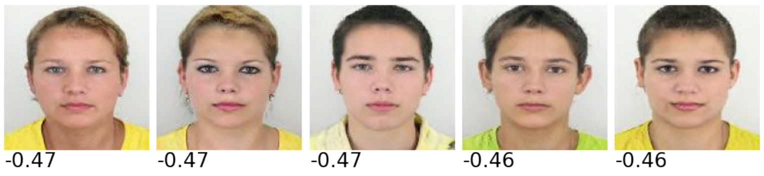


Figure 5.19: Generated images showing females with highest discriminator score.

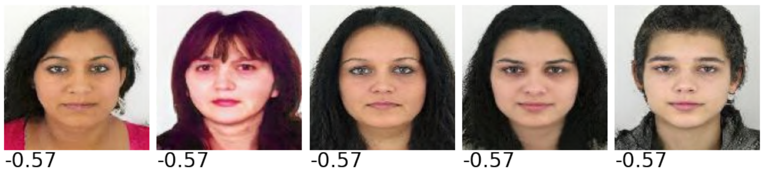


Figure 5.20: Generated images showing females with slightly lower discriminator score.

Figures 5.19 and 5.20 show images of women. Notice that the women from Figure 5.19 look quite boyish (short hair, face). This seems to be caused by the images having a very good discriminator score combined with the fact that most of the images with this discriminator score were men. To produce a more diverse set of women, images with lower discriminator scores should be taken and an example of such images is shown in Figure 5.20. The women in this second figure look more feminine than those from Figure 5.19 (long hair, feminine face).

Notice that all the women images from Figures 5.19 and 5.20 had dark hair. For completeness, the Figure 5.21 shows five images of women with blonde hair.

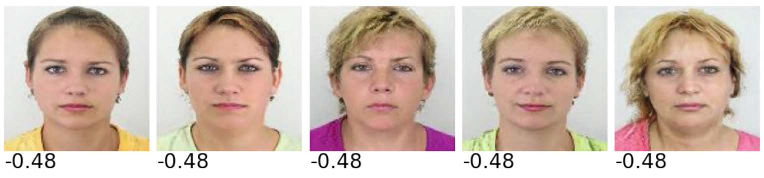


Figure 5.21: Generated images showing blonde females.

Histograms showing the overall results for the *Faces* dataset and generated dataset are shown in Figures 5.22 and 5.23 respectively.

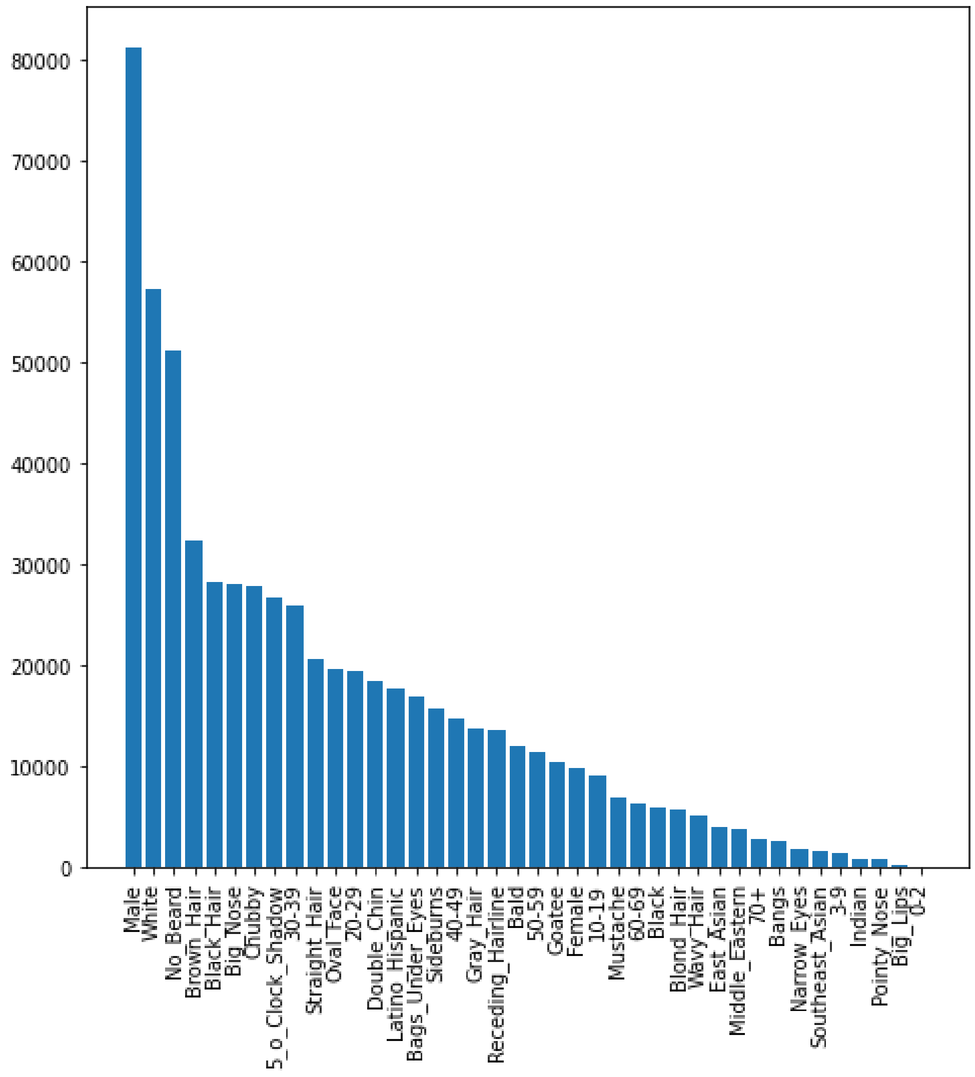


Figure 5.22: Number of positive examples for each of the attributes. Total number of images is around 90k. Taking argmax of hair when all hair predictions are < 0.5. Evaluated on *Faces* dataset.

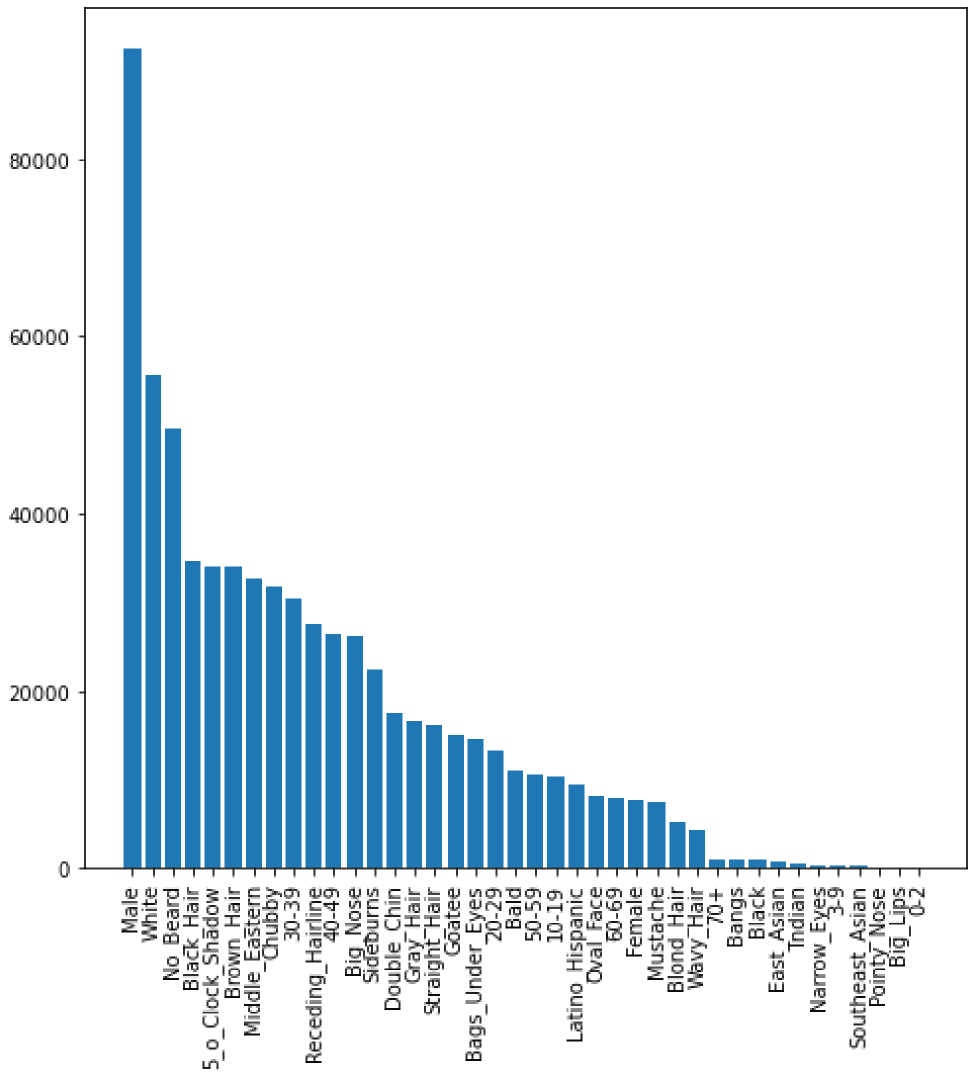


Figure 5.23: Number of positive examples for each of the attributes. Total number of images is around 100k. Taking argmax of hair when all hair predictions are < 0.5. Evaluated on a sample of 100000 randomly generated images.

Figures 5.22 and 5.23 suggest that the attributes in the generated dataset are distributed similarly to the attributes in the *Faces* dataset. In other words, this indicates that the generator reproduces or mimics what it was trained on. Furthermore, this signalizes that the generator might not be able to generate anything above the limited set of images it has seen. However, at the same time, it is believed that the generator is at least able to generate combinations of attributes that were seen in the training data even if they were quite rare, therefore making it able to generate those rare persons assuming that enough images are generated. Nevertheless, it should be mentioned that some of the very rare combinations may actually not be present and the reason they are present in the heatmap might be because of attribute classification errors. In any case, the generator's ability to generate rare cases could be improved by training the generator on a more diverse dataset as is often the case in practice (consider, e.g. FFHQ dataset used in StyleGAN).

5.3 Encoder's results

The main challenge of the encoder is to allow for a good reconstruction of the input image. Several such reconstructions are shown in Figure 5.24. The input images throughout this and the next section were taken from Face Research Lab London Set (FRLLS) [90] dataset.

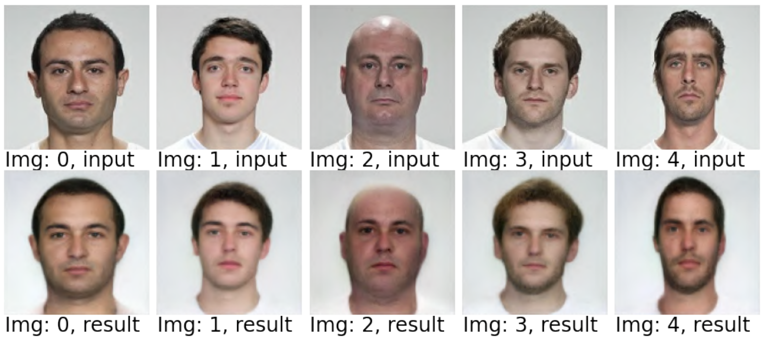


Figure 5.24: Examples of input image reconstructions. Example input images are from FRLLS dataset.

The first row of the 5.24 shows 5 input images $img_1, \ldots, img_5$, where each of these images was taken and passed through the encoder E to obtain five latent vectors $\boldsymbol{w_i} = E(img_i)\ \forall i\ \in 1, \ldots, 5$. Finally, these latent vectors were taken and passed through generator as $img_i^{'} = G(\boldsymbol{w_i})$ to produce the final reconstructions which could be seen in the second row.

Even though all the reconstructions seem to be quite reasonable in terms of image quality, they can be slightly improved by using the fine-tuning approach from Section 3.4.4. This means, that for each input image a pre-trained encoder is taken, fine-tuned to that single image, and then the image reconstruction is performed. The results could be seen in Figure 5.25.

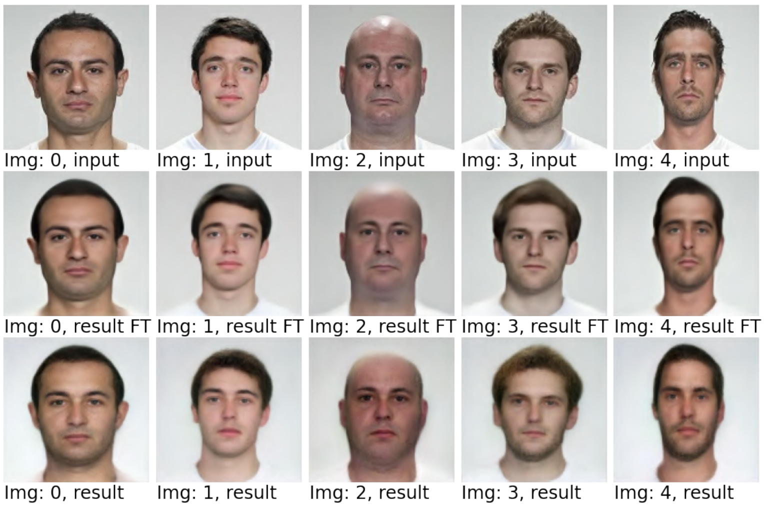


Figure 5.25: Examples of input image reconstructions with fine-tuning. Example input images are from FRLLS dataset.

Figure 5.25 compares the reconstructions for encoder with and without fine-tuning. The first row shows the input images, the second row shows the reconstructions with fine-tuning, and finally, the last row shows ordinary (without fine-tuning) reconstructions. It seems that for all these images (especially images 2 and 4) the fine-tuning approach actually helped with improving the reconstruction quality. Notice that the hair of the fine-tuned results seems way too smooth, but in this case, the hair was weird-looking even before using the fine-tuning approach.

Sometimes it is the case that the input image is quite difficult for the generator to generate and therefore also difficult to reconstruct. In that case, the reconstructions can have quite a low quality as illustrated in Figure 5.26.

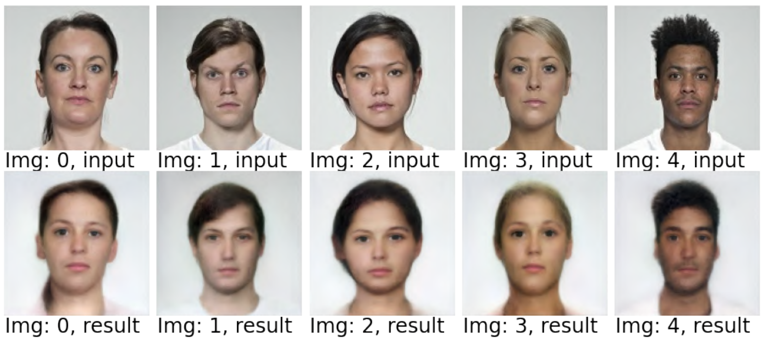


Figure 5.26: Examples of input image reconstructions with low quality. Example input images are from FRLLS dataset.

Figure 5.26 illustrates the issue of reconstructing images that are too different from images from the *Faces* dataset. To some extent, this issue could be solved by using the fine-tuning approach which leads to results that are shown in Figure 5.27.

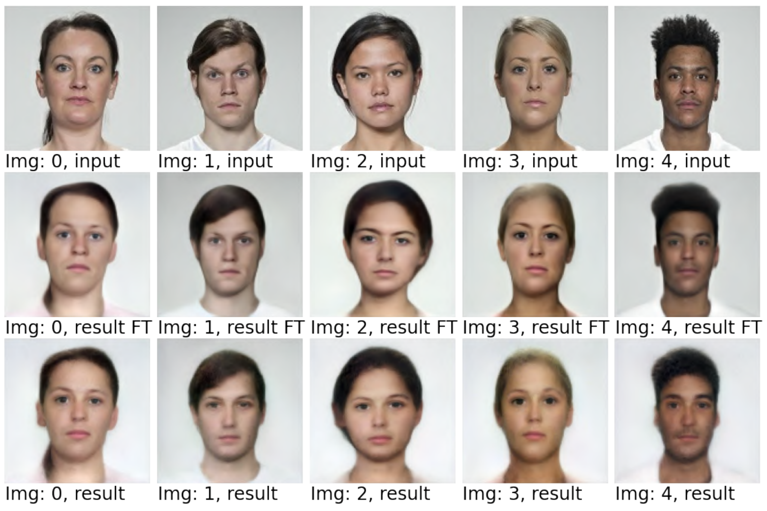


Figure 5.27: Examples of input image reconstructions with low quality with fine-tuning. Example input images are from FRLLS dataset.

Similarly to the Figure 5.25, Figure 5.27 compares results of reconstructions with and without fine-tuning, but now on a more difficult samples which are very different to images from the *Faces* dataset. Images 0 and 3 are difficult because they are showing women which are generally more difficult to reconstruct than men (mostly because the *Faces* dataset contains mostly man photographs). Images 2 and 4 are difficult because the depicted persons are of a different ethnicity than a majority of the persons in the *Faces* dataset. Fine-tuning definitely helped in the reconstruction of images 3 and image 4, and slightly helped in reconstructions of images 1 and 2. For image 0, there is no apparent difference between reconstruction with and without fine-tuning.

It should be emphasized that even though the fine-tuning approach was helpful for some of the above images, it still cannot be considered as a general remedy for all difficult reconstructions because there are still situations where it does not help very much (e.g. image 0 from Figure 5.27). This observation implies that it seems reasonable to have this feature but it should be used with caution and it should be up to the user to decide if it makes sense for a given situation.

The latent vectors $\boldsymbol{w_i}$, $\boldsymbol{w_j}$ could also be used for image interpolations (see sub-section 1.3.2) between two images img_i, img_j by considering n latent vectors $\boldsymbol{w_i} = \boldsymbol{w_1'}, \boldsymbol{w_2'}, \ldots, \boldsymbol{w_n'} = \boldsymbol{w_j}$ and taking their image representations. An example of results for interpolation of two images is shown in Figure 5.28.

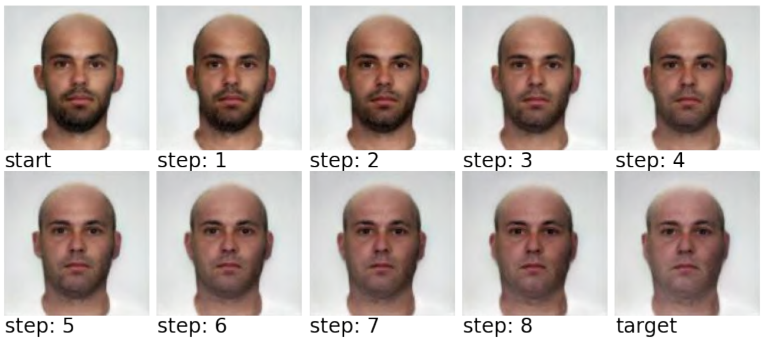


Figure 5.28: Examples of interpolation between two input images. Example input images are from FRLLS dataset.

Figure 5.28 shows an interpolation between two images with $n = 10$. The latent vectors were obtained using an encoder without fine-tuning. Because both persons are bald, the most changing parts are the beard (which is slightly disappearing) together with the face itself. The interpolations work quite well and they are usually also quite smooth (i.e. no extra intermediate persons). Generally, it seems that the most limiting factor in interpolations is the reconstructions of the initial and target images. Interpolations could again be slightly improved by using the fine-tuning approach as shown in Figure 5.29.

Finally, an example of finding similar images (fillers) to the reconstruction of the input image (suspect) are shown in Figures 5.30 and 5.32.

Figure 5.30 shows the input image, its reconstruction together with its reconstruction from a vector passed through PCA, and 22 images showing similar persons (fillers). Similar to all the results above, also these results could be slightly improved by incorporating the fine-tuning as can be seen in Figure 5.31.

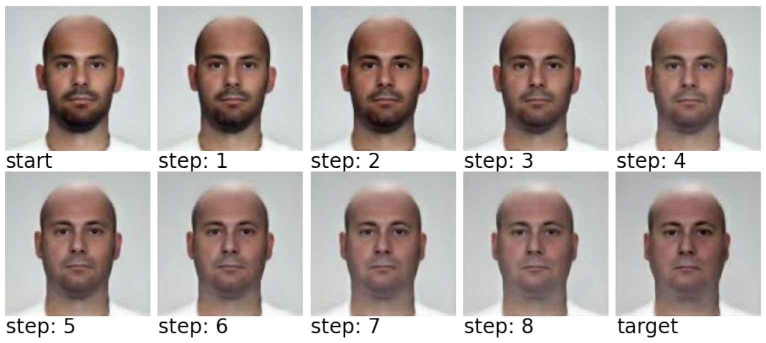


Figure 5.29: An example of interpolation between two images with fine-tuning. Example input images are from FRLLS dataset.

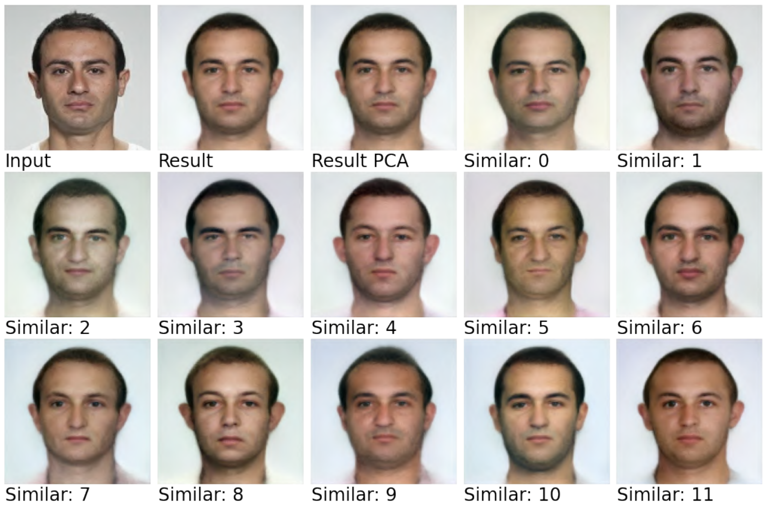


Figure 5.30: Examples of persons similar to the seed person. Example input image is from FRLLS dataset.

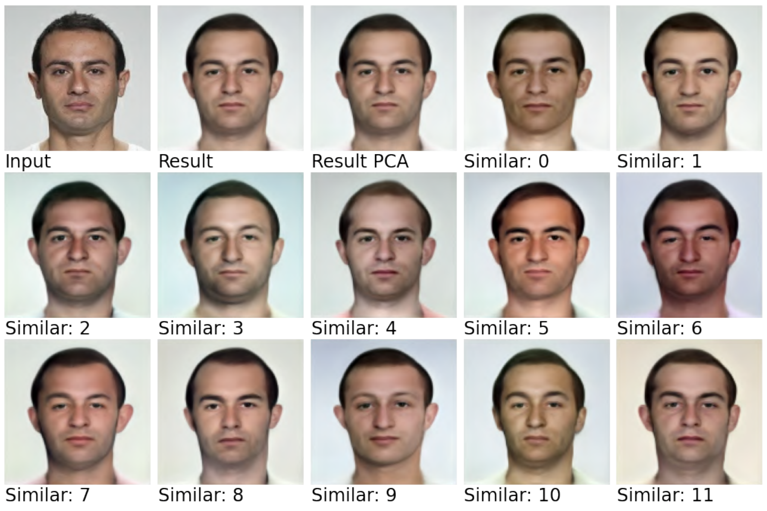


Figure 5.31: Examples of persons similar to the seed person with fine-tuning. Example input image is from FRLLS dataset.

An example of finding fillers for a more difficult suspect image is shown in Figure 5.32.

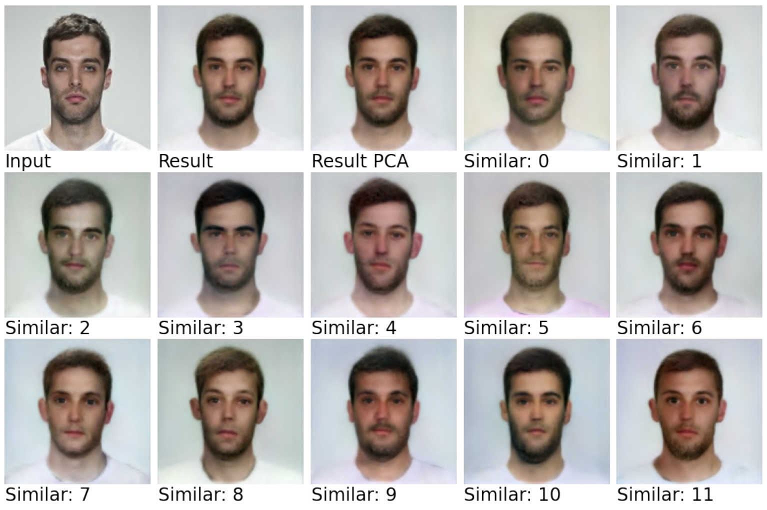


Figure 5.32: Examples of persons similar to the more difficult seed person. Example input image is from FRLLS dataset.

In Figure 5.32 the filler images seem to be slightly noisy, and the overall quality of these images is perceptually worse than the results from Figure 5.30. Using fine-tuning, in this case, helps to de-noise and smooth the results as illustrated in Figure 5.33.

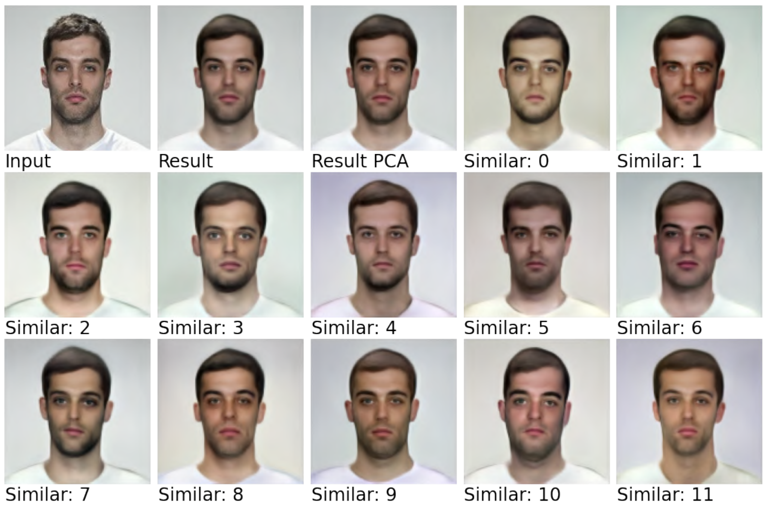
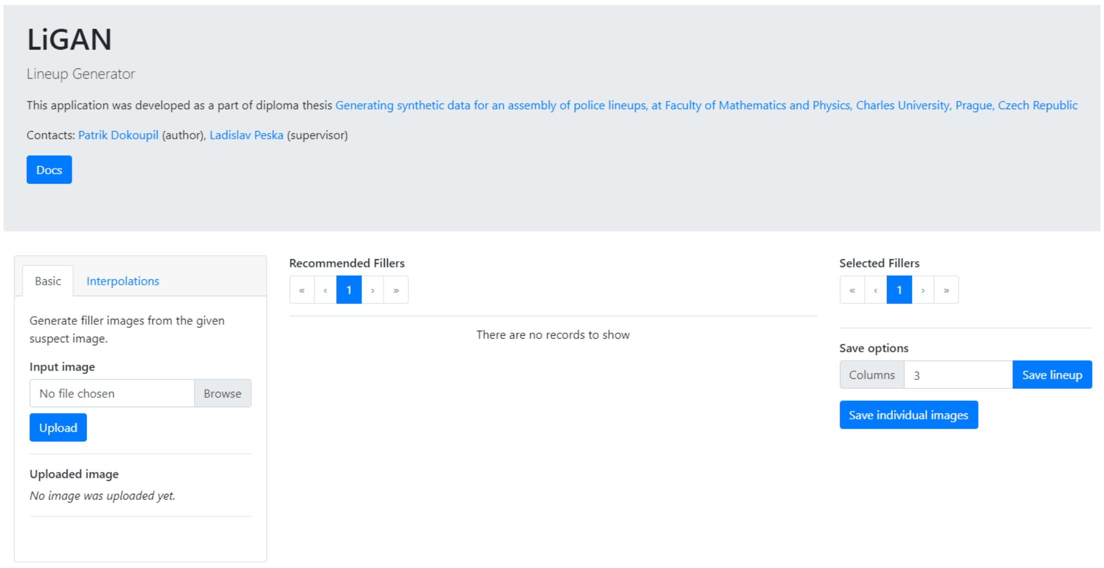


Figure 5.33: Examples of persons similar to the more difficult seed person with fine-tuning. Example input image is from FRLLS dataset.

5.4 LiGAN showcase

The application whose architecture and design were described Chapter 4 was eventually implemented and it was named LiGAN[3]. This section briefly presents what LiGAN looks like and how does it work, both illustrated on a simple work case. It is important to mention that this section is not intended to fully replace user documentation.

For the end-users, LiGAN is simply a website that allows them to generate police lineups and to perform additional tasks which could help them in generating a lineup of their needs. At the very start, the user sees only an empty screen, without any images, as can be seen in Figure 5.34.


LiGAN

Lineup Generator

This application was developed as a part of diploma thesis Generating synthetic data for an assembly of police lineups, at Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

Contacts: Patrik Dokoupil (author), Ladislav Peska (supervisor)

Docs

Basic | Interpolations

Generate filler images from the given suspect image.

Input image

No file chosen | Browse

Upload

Uploaded image

No image was uploaded yet.

Recommended Fillers

« ‹ 1 › »

There are no records to show

Selected Fillers

« ‹ 1 › »

Save options

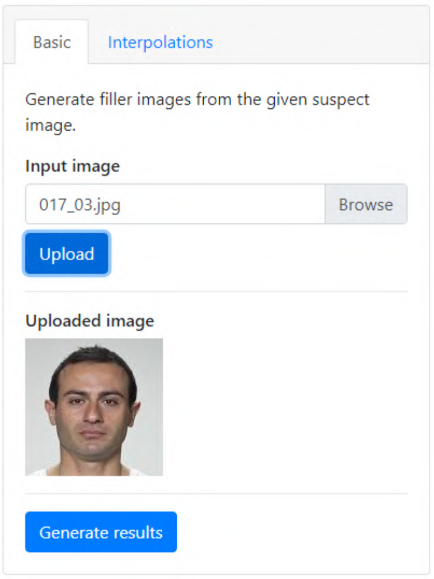
Columns | 3 | Save lineup

Save individual images

Figure 5.34: LiGAN in it's initial state, before starting working on a lineup.

From the empty screen, the user may proceed in two possible workflows, that is, either upload a single image of the suspect (*Basic*) or two images (*Interpolations*) of the suspect and/or a similar person. Examples of setting inputs to each of these scenarios are shown in Figure 5.35.

[3]This name comes from a phrase ***Li****neup* ***Gen****erator* where *Gen* was changed to *GAN* which fits both semantically and phonetically.

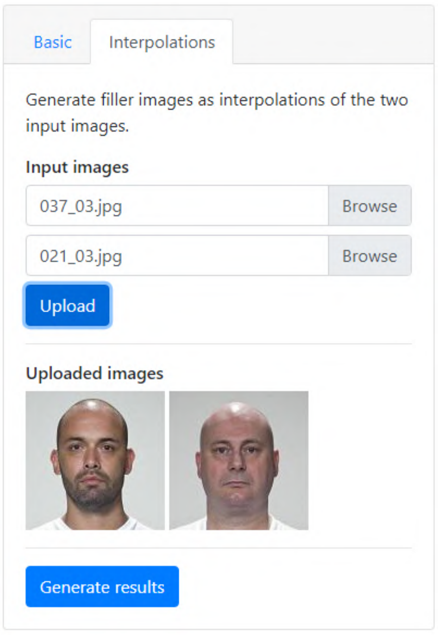

Generate filler images from the given suspect image.

Input image

017_03.jpg

Uploaded image

(a) Basic scenario input.


Generate filler images as interpolations of the two input images.

Input images

037_03.jpg

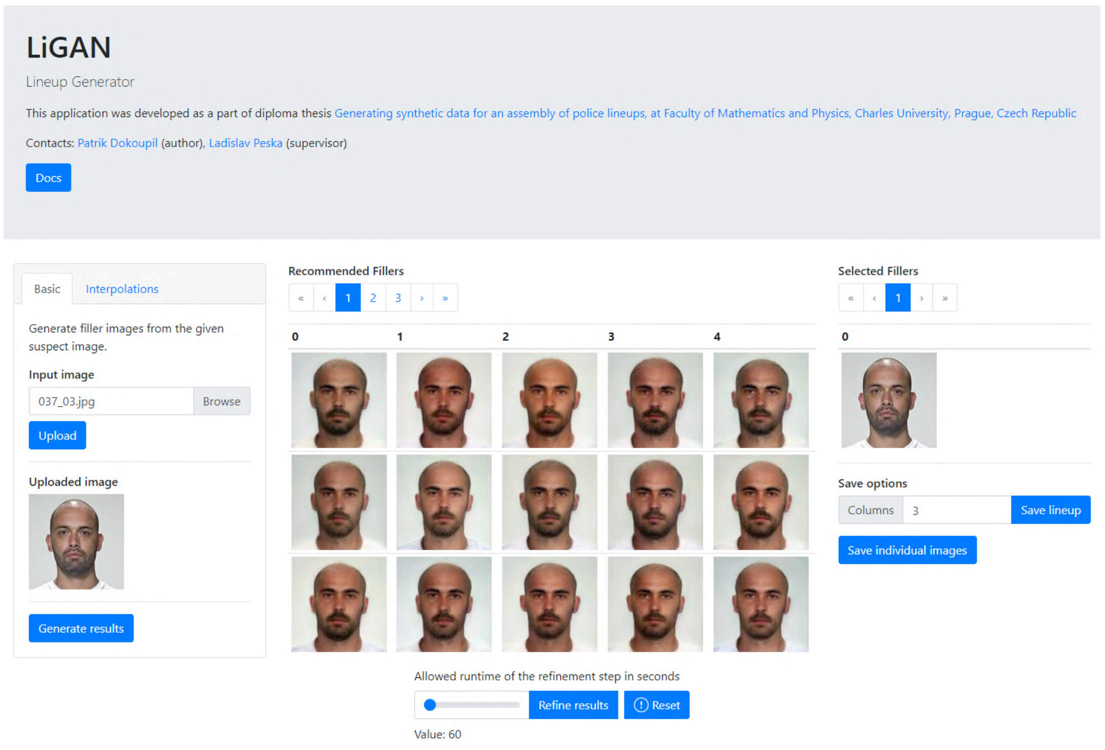
021_03.jpg

Uploaded images

(b) Interpolation scenario input.

Figure 5.35: Example input images are from FRLLS dataset.

For now, assume that the user opted for the *Basic* scenario by uploading a single image as was illustrated on the left of Figure 5.35. By pressing *Generate results* button, the model will generate a single page of results as shown in Figure 5.36.


LiGAN

Lineup Generator

This application was developed as a part of diploma thesis Generating synthetic data for an assembly of police lineups, at Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

Contacts: Patrik Dokoupil (author), Ladislav Peska (supervisor)

Generate filler images from the given suspect image.

Input image

037_03.jpg

Uploaded image

Recommended Fillers

Selected Fillers

Save options

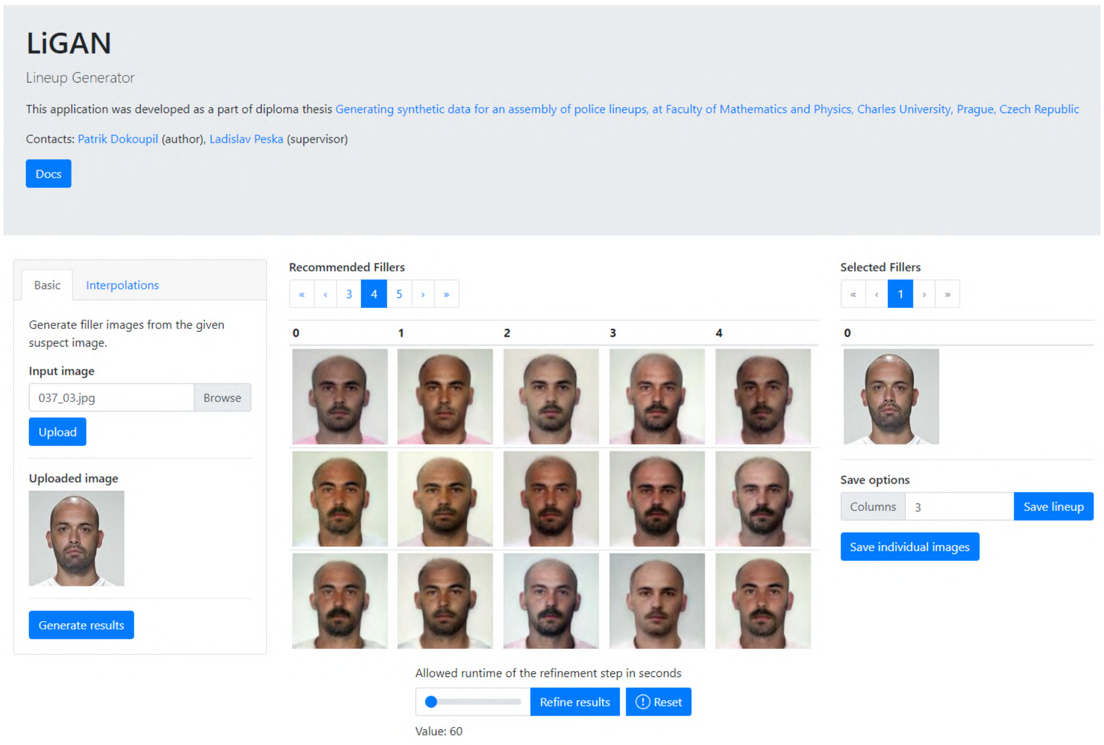
Columns 3

Allowed runtime of the refinement step in seconds

Value: 60

Figure 5.36: First page of results generated for the given suspect image. Example input image is from FRLLS dataset.

As can be seen from Figure 5.36, all the generated images are very similar to each other. This is caused by the fact that the similarity of the images is decreasing with an increasing page number. In other words, the first page contains images that are most similar to the input (and also most similar to each other due to how the fillers are generated) while the latter pages contain less similar images. For example, images on the 4th page will be less similar as can be seen in Figure 5.37.


LiGAN

Lineup Generator

This application was developed as a part of diploma thesis Generating synthetic data for an assembly of police lineups, at Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

Contacts: Patrik Dokoupil (author), Ladislav Peska (supervisor)

Generate filler images from the given suspect image.

Input image

037_03.jpg

Columns 3

Allowed runtime of the refinement step in seconds

Value: 60

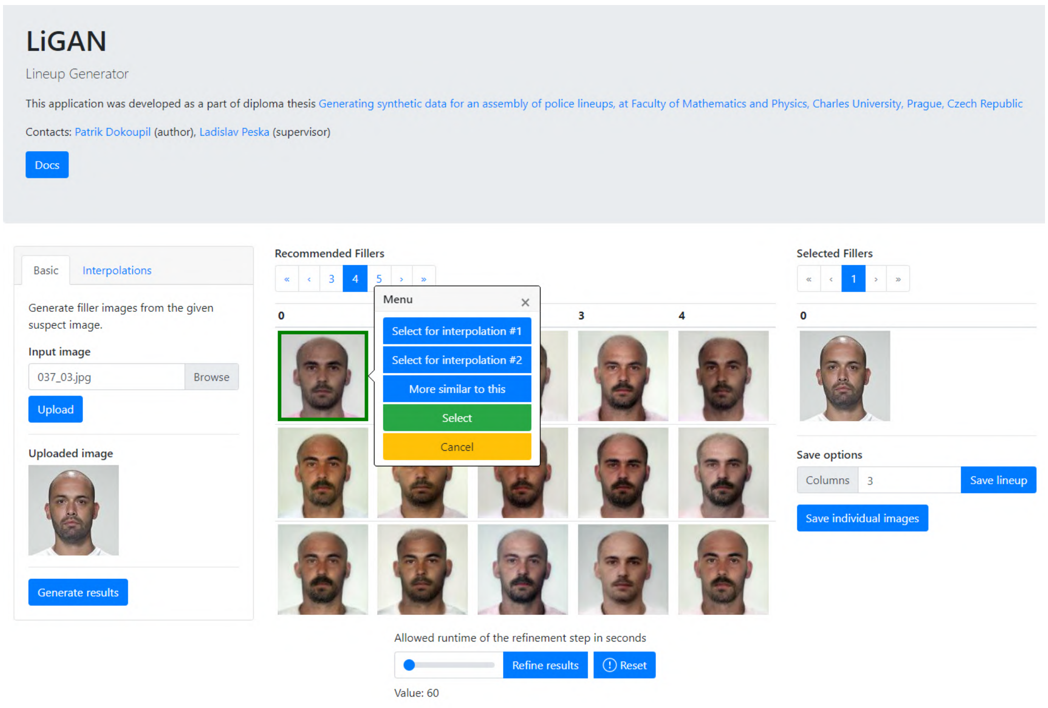
Figure 5.37: Fourth page of results generated for the given suspect image. Example input image is from FRLLS dataset.

When comparing results from the first page (Figure 5.36) with results from the fourth page (Figure 5.37) it is clearly visible that the latter results are slightly more diverse than the former.

Once the results are presented, the user could click any of the generated images to invoke the context menu allowing to do one of the following actions:

- *Select the image as interpolation input #1 or #2* – Simply replaces first or second input image to the interpolation scenario and the user then could proceed by pressing *Generate results* as usual.
- *More similar to this* — Replaces the input image to the basic scenario and the user then could proceed by pressing *Generate results* as usual.
- *Select* — Selects the image as filler.
- *Cancel* — Closes the context menu.

The context menu is shown in Figure 5.38.


LiGAN

Lineup Generator

This application was developed as a part of diploma thesis Generating synthetic data for an assembly of police lineups, at Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

Contacts: Patrik Dokoupil (author), Ladislav Peska (supervisor)

Generate filler images from the given suspect image.

Input image

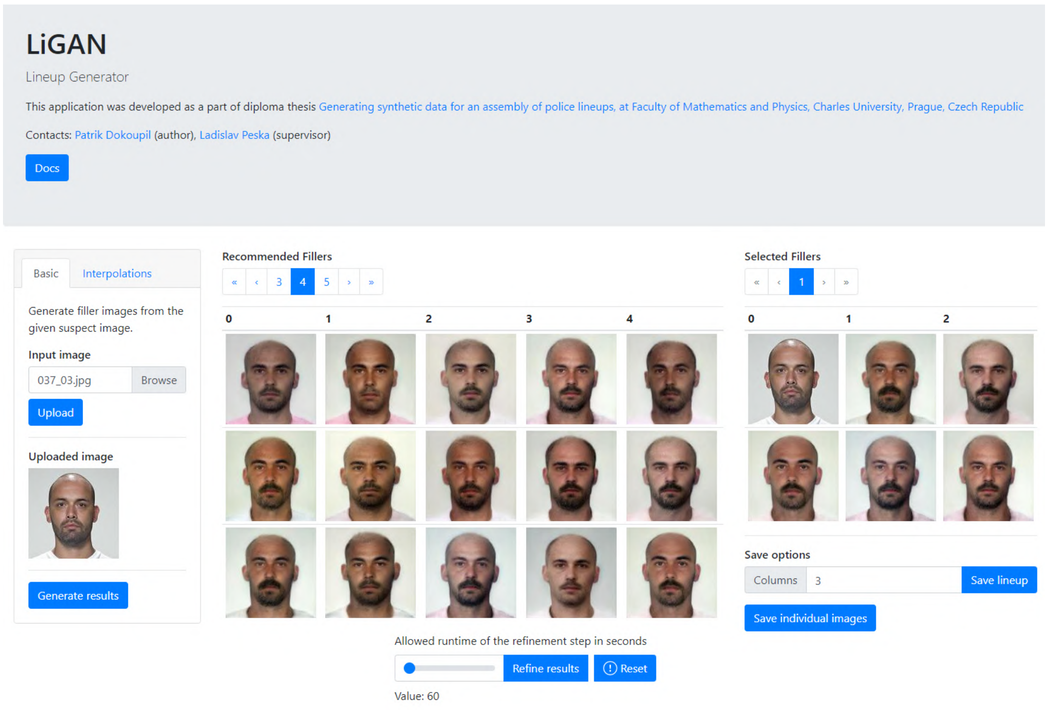
037_03.jpg

Allowed runtime of the refinement step in seconds

Value: 60

Figure 5.38: Context menu invokend on one of the generated images. Example input image is from FRLLS dataset.

Assume that the user searched through the results and selected several images as fillers. These selected fillers are then displayed in the right part of the screen as illustrated in Figure 5.39.


LiGAN

Lineup Generator

This application was developed as a part of diploma thesis Generating synthetic data for an assembly of police lineups, at Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

Contacts: Patrik Dokoupil (author), Ladislav Peska (supervisor)

Generate filler images from the given suspect image.

Input image

037_03.jpg

Allowed runtime of the refinement step in seconds

Value: 60

Figure 5.39: LiGAN UI when user has already selected some fillers. Example input image is from FRLLS dataset.

Figure 5.39 shows a screen where the user has already selected 6 images as fillers. The user is also able to click on any of the selected fillers to invoke another context menu which allows removing the image from selected fillers. Furthermore, it should be mentioned, that every time an image is uploaded it is automatically added to the resulting lineup (and to the selected fillers) because it is often the case that it should be present there. However, the user is still able to remove the image from the selected fillers.

Once the user is satisfied with the filler selection, the final lineup could be generated by pressing *Save lineup*. An example of (cropped) generated lineup is shown in Figure 5.40.

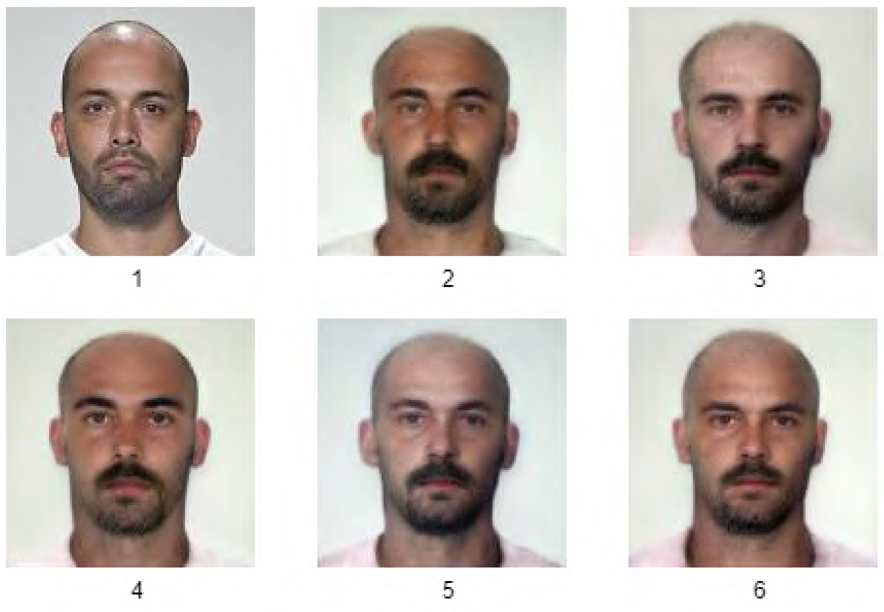
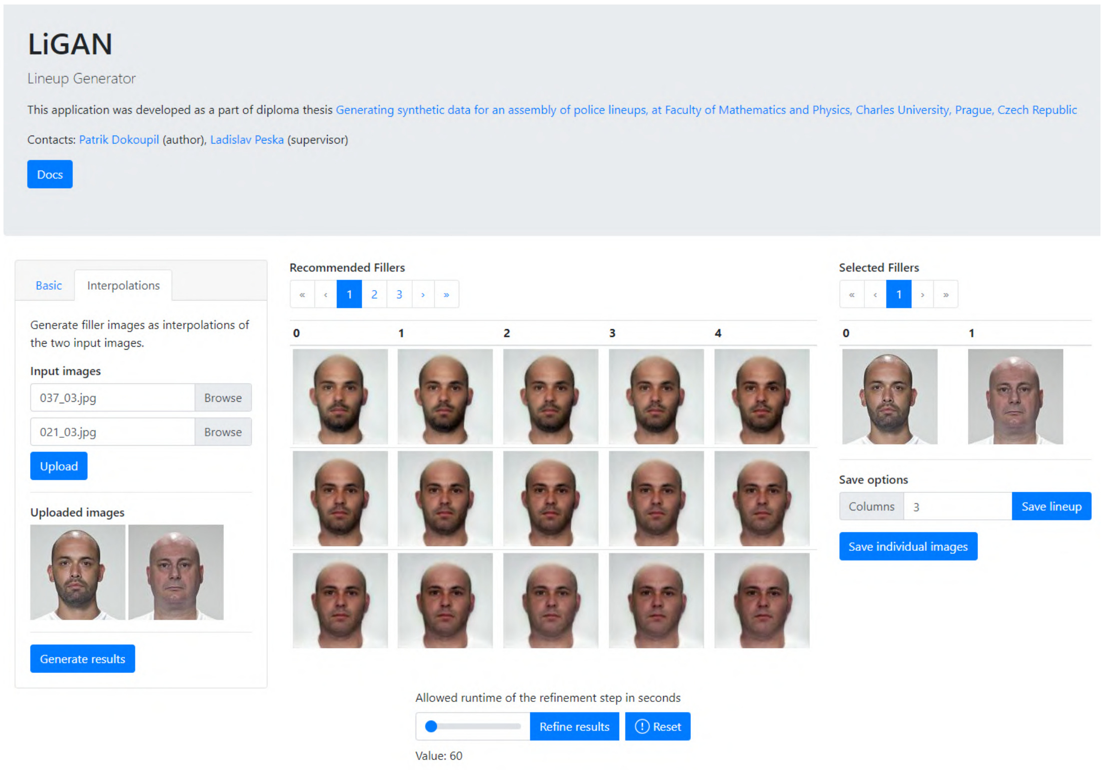


Figure 5.40: Example of a lineup constructed in LiGAN.

The user may also use the interpolations scenario instead of the *Basic* one, if that is the case, the page of results shows images for individual steps of the interpolation. The results for interpolation scenario are shown in Figure 5.41.

Because sometimes the model cannot provide reasonable results, it may help to use fine-tuning approach to potentially improve the results. The user is allowed to invoke fine-tuning by pressing *Refine results*, moreover, there is a slidebar next to this button which allows the user to set a time limit for fine-tuning (in seconds) with a value between 60 (default value) and 600 seconds. It should be kept in mind that the time limit is not perfectly accurate and the user should count with about 30 seconds interval around the selected time limit[4]. The results are then shown on a new page in a standard manner as illustrated in Figure 5.42. Finally, it should be mentioned that the refining tasks are only available once some results are generated (either from the *Basic* or *Interpolation* scenario).

[4]This is caused by the way the fine-tuning is performed because some steps can take up to 20-30 seconds without a possibility of being interrupted.



LiGAN

Lineup Generator

This application was developed as a part of diploma thesis Generating synthetic data for an assembly of police lineups, at Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

Contacts: Patrik Dokoupil (author), Ladislav Peska (supervisor)

Generate filler images as interpolations of the two input images.

Input images

037_03.jpg

021_03.jpg

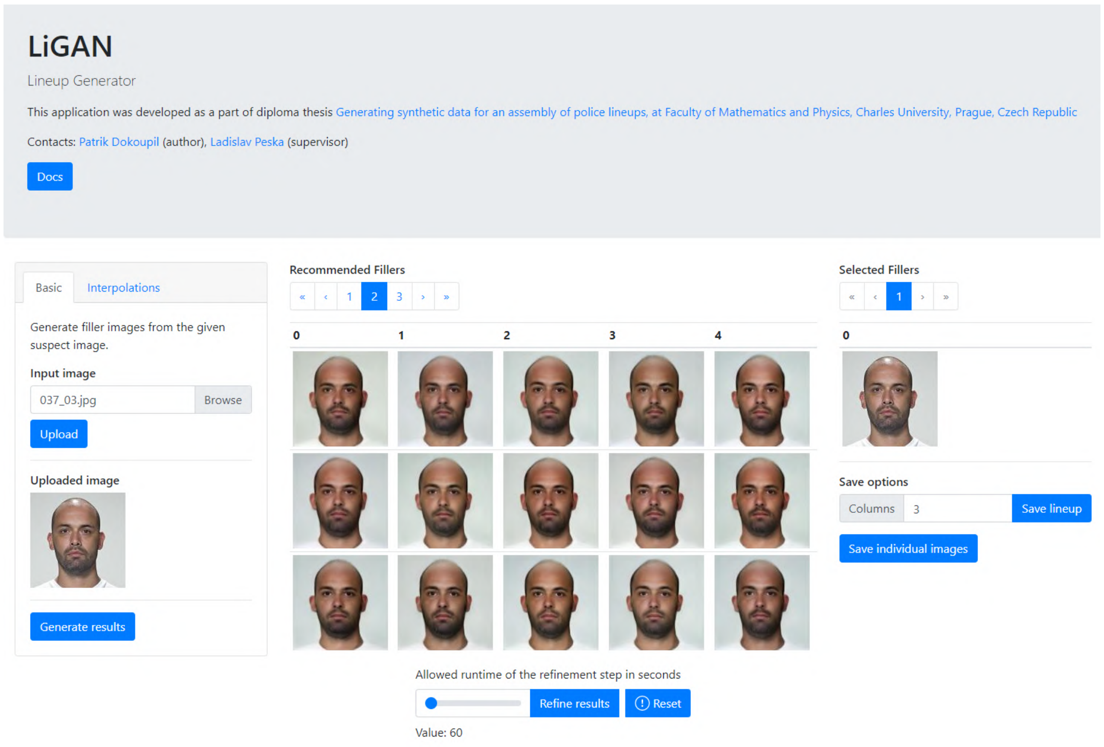
Uploaded images

Save options: Columns 3

Allowed runtime of the refinement step in seconds

Value: 60

Figure 5.41: Results for the interpolation scenario. Example input images are from FRLLS dataset.



LiGAN

Lineup Generator

This application was developed as a part of diploma thesis Generating synthetic data for an assembly of police lineups, at Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic

Contacts: Patrik Dokoupil (author), Ladislav Peska (supervisor)

Generate filler images from the given suspect image.

Input image

037_03.jpg

Uploaded image

Save options: Columns 3

Allowed runtime of the refinement step in seconds

Value: 60

Figure 5.42: Refined results. Example input image is from FRLLS dataset.

Conclusion

The main goal of this thesis was to propose a new variation or adjust an existing generative model that could be used for generating synthetic images of people's faces. Then, the main intention was to use these faces for the assembly of police lineups. The main goal was fulfilled because the final model—StyleGAN2 trained on a dataset of missing persons—could be used for the target scenario. Furthermore, there were several sub-goals defined in Section Goals which are re-evaluated below.

1. *The images generated by the model should have a reasonable quality in terms of resolution*—This sub-goal was achieved because the generated images have a resolution of 128×128 pixels that should be sufficient for the task, although there is still a room for an improvement.

2. *The generated face images should not be easily distinguishable from images of real people*—This sub-goal was also fulfilled because as can be seen from user study described in Section 5.1, the users were not substantially better at identifying fake images than guessing at random. Although there exist images that are very easily identifiable as fake, it is important that there are enough difficultly distinguishable images.

3. *Generated images should be diverse enough*—This sub-goal was fulfilled, although it should be mentioned that the diversity is at the level of diversity of the training dataset. Especially, this means that the generator is not able to generalize beyond the training dataset.

4. *The output of the model should be controllable*—This sub-goal was fulfilled thanks to the encoder that was implemented as a part of this thesis.

Beyond the crucial requirements described above, there were two more "soft requirements" whose evaluation follows:

- *The model should be capable of generating images of people with rare facial features*—This extra sub-goal was not fully fulfilled. The problem is with rare facial features that were not present in the generator's training dataset because in that case, the generator is not able to generate them. To some extent, this problem could be mitigated by the fine-tuning approach from Section 3.4.4 but the results are rather controversial (i.e. it helps only in some cases).

- *The control over the model's output should be achieved by allowing seeding the model with an image*—This extra sub-goal was achieved, again, thanks to the use of the encoder.

To summarize, all the essential goals were fulfilled, although there are still some possibilities for improving the whole model as will be described in Section Future work. Apart from these goals, one extra achievement was accomplished—showing a proof of concept application called LiGAN which incorporates all of the implemented models and makes them available for easy use by the end-user.

Future work

As it was already mentioned, although all the goals were fulfilled, there is still some room for improvements. There are three main areas in which this thesis and its contribution could be improved and which should be considered for future work. These areas are listed below, ordered from most important to least important.

1. *Extending training dataset*—The problems with reconstruction of some of the images that were shown in previous Chapter 5 were mostly related to the fact that the training dataset was not diverse enough. It would be interesting to train the generator on a more diverse dataset, including photographs of persons of multiple ethnicities. The simplest way would be to take the existing *Faces* dataset and extend it by the images from different sources. A possible source of images would be databases of missing persons from different countries (preferably from countries outside of Europe).

2. *Make the LiGAN application production ready*—Regarding the LiGAN application, there are still some parts that could be done better and more efficiently as its current form is only a proof of concept. Notably, it would be beneficial to improve performance and stability for the scenario when multiple users use the application (e.g. more efficient task scheduling among the container jobs). Also, it would be reasonable to make the application more easily deployable and think over deployment outside of the GPU cluster. This also includes cleanup and simplification of some parts of the code and architecture (e.g. getting rid of volume bindings which are not very suitable for the production environment). Another important aspect that should be addressed before the application will be production-ready is security. The application security should be analyzed and the possible issues should be resolved. For example, it will be important to introduce some kind of authentication presumably by introducing user accounts.

3. *Train the models on higher resolution*—It would be worthy to try to train all the models on slightly larger resolution, probably on 256×256 pixels, because the *Faces* dataset is available in this resolution so it would not be necessary to find a new dataset. The target resolution could be set even higher assuming that a dataset of high enough resolution is available. It is possible that using higher resolution would help in producing images with higher quality and also improving the performance of the facial feature prediction (some features might be barely observables for low resolution). However, it should be kept in mind that higher resolution could also have inversed effect on result quality, simply because some defects could be more visible when resolution is larger.

Bibliography

[1] Travis M. Seale-Carlisle and Laura Mickes. Us line-ups outperform uk line-ups. *Royal Society open science*, 3(9):160300–160300, Sep 2016. 27703695[pmid].

[2] Curt A. Carlson, Alyssa R. Jones, Jane E. Whittington, Robert F. Lockamyeir, Maria A. Carlson, and Alex R. Wooten. Lineup fairness: propitious heterogeneity and the diagnostic feature-detection hypothesis. *Cognitive research: principles and implications*, 4(1):20–20, Jun 2019. 31197501[pmid].

[3] Identification Procedures: Photo Arrays and Line-ups, 06 2017. https://www.criminaljustice.ny.gov/pio/press_releases/ID-Procedures-Protocol-Model-Policy-Forms.pdf, last accessed on 05/14/21.

[4] New York State Photo Identification Guidelines, 04 2011. https://www.criminaljustice.ny.gov/ops/training/other/story_content/external_files/photoarrayguidelines.pdf, last accessed on 05/14/21.

[5] Roy Malpass and Patricia Devine. Eyewitness identification: Lineup instructions and the absence of the offender. *Journal of Applied Psychology*, 66:482–489, 08 1981.

[6] Steven E. Clark and Ryan D. Godfrey. Eyewitness identification evidence and innocence risk. *Psychonomic Bulletin & Review*, 16(1):22–42, Feb 2009.

[7] Ryan Fitzgerald, Heather Price, Chris Oriet, and Steve Charman. The effect of suspect-filler similarity on eyewitness identification decisions: A meta-analysis. *Psychology Public Policy and Law*, 19, 05 2013.

[8] R. C. L. Lindsay, Joanna D. Pozzulo, Wendy Craig, Kang Lee, and Samantha Corber. Simultaneous lineups, sequential lineups, and showups: Eyewitness identification decisions of adults and children. *Law and Human Behavior*, 21(4):391–404, Aug 1997.

[9] Jennifer Tunnicliff and Steven Clark. Selecting foils for identification lineups: Matching suspects or descriptions? *Law and human behavior*, 24:231–58, 05 2000.

[10] Gary Wells, SM RYDELL, and EP SEELAU. The selection of distractors for eyewitness lineups. *Journal of Applied Psychology*, 78:835–844, 10 1993.

[11] Stephen Darling, Tim Valentine, and Amina Memon. Selection of lineup foils in operational contexts. *Applied Cognitive Psychology*, 22:159 – 169, 03 2008.

[12] Gary Wells, Margaret Kovera, Amy Douglass, Neil Brewer, Christian Meissner, and John Wixted. Policy and procedure recommendations for the collection and preservation of eyewitness identification evidence. *Law and Human Behavior*, 44:3–36, 02 2020.

[13] Ladislav Peska and Hana Trojanova. Towards similarity models in police photo lineup assembling tasks. In Stéphane Marchand-Maillet, Yasin N. Silva, and Edgar Chávez, editors, *Similarity Search and Applications*, pages 217–225, Cham, 2018. Springer International Publishing.

[14] Ladislav Peska and Hana Trojanova. Towards recommender systems for police photo lineup. In *Proceedings of the 2nd Workshop on Deep Learning for Recommender Systems*, DLRS 2017, page 19–23, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[16] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014.

[17] T. Hastie, R. Tibshirani, and J.H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer series in statistics. Springer, 2009.

[18] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943.

[19] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958.

[20] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:303–314, 1989.

[21] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[22] Milan Straka. Introduction to Deep Learning. `https://ufal.mff.cuni.cz/~straka/courses/npfl114/1920/slides.pdf/npfl114-01.pdf`, last accessed on 05/14/21.

[23] Sho Sonoda and Noboru Murata. Neural network with unbounded activation functions is universal approximator. *Applied and Computational Harmonic Analysis*, 43(2):233–268, 2017.

[24] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. JMLR Workshop and Conference Proceedings.

[25] Zhiyuan Li and Sanjeev Arora. An exponential learning rate schedule for deep learning. In *8th International Conference on Learning Representations,*

ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020. OpenReview.net, 2020.

[26] Jiawei Zhang. Gradient descent based optimization algorithms for deep learning models training. *CoRR*, abs/1903.03614, 2019.

[27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[28] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[29] RMSProp. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf, last accessed on 05/14/21.

[30] Kaifeng Lv, Shunhua Jiang, and Jian Li. Learning gradient descent: Better generalization and longer horizons. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2247–2255. PMLR, 06–11 Aug 2017.

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

[32] Data augmentation — Tensorflow Core. https://www.tensorflow.org/tutorials/images/data_augmentation, last accessed on 05/14/21.

[33] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[34] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6:107–116, 04 1998.

[35] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[36] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[37] Davi Frossard. VGG in TensorFlow. https://www.cs.toronto.edu/~frossard/post/vgg16/, last accessed on 05/14/21.

[38] S. Liu and W. Deng. Very deep convolutional neural network based image classification using small training sample size. In *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*, pages 730–734, 2015.

[39] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 936–944, 2017.

[40] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 2672–2680, Cambridge, MA, USA, 2014. MIT Press.

[41] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[42] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 214–223. PMLR, 06–11 Aug 2017.

[43] Martín Arjovsky and Léon Bottou. Towards principled methods for training generative adversarial networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[44] Kevin Roth, Aurélien Lucchi, Sebastian Nowozin, and Thomas Hofmann. Stabilizing training of generative adversarial networks through regularization. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 2018–2028, 2017.

[45] Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. Unrolled generative adversarial networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[46] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[47] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[48] Augustus Odena, Christopher Olah, and Jonathon Shlens. Conditional image synthesis with auxiliary classifier gans. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 2642–2651. PMLR, 2017.

[49] Miles Brundage, Shahar Avin, Jack Clark, Helen Toner, Peter Eckersley, Ben Garfinkel, Allan Dafoe, Paul Scharre, Thomas Zeitzoff, Bobby Filar, Hyrum S. Anderson, Heather Roff, Gregory C. Allen, Jacob Steinhardt, Carrick Flynn, Seán Ó hÉigeartaigh, Simon Beard, Haydn Belfield, Sebastian Farquhar, Clare Lyle, Rebecca Crootof, Owain Evans, Michael Page, Joanna Bryson, Roman Yampolskiy, and Dario Amodei. The malicious use of artificial intelligence: Forecasting, prevention, and mitigation. *CoRR*, abs/1802.07228, 2018.

[50] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 4401–4410. Computer Vision Foundation / IEEE, 2019.

[51] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of stylegan. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 8107–8116. IEEE, 2020.

[52] Ken Shoemake. Animating rotation with quaternion curves. *SIGGRAPH Comput. Graph.*, 19(3):245–254, July 1985.

[53] 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. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 6626–6637, 2017.

[54] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 586–595. IEEE Computer Society, 2018.

[55] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells III, and Alejandro F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2015 - 18th International Conference Munich, Germany, October 5 - 9, 2015, Proceedings, Part III*, volume 9351 of *Lecture Notes in Computer Science*, pages 234–241. Springer, 2015.

[56] Animesh Karnewar and Oliver Wang. MSG-GAN: multi-scale gradients for generative adversarial networks. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 7796–7805. IEEE, 2020.

[57] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[58] Ishaan Gulrajani, Faruk Ahmed, Martín Arjovsky, Vincent Dumoulin, and Aaron C. Courville. Improved training of wasserstein gans. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5767–5777, 2017.

[59] Ali Razavi, Aäron van den Oord, and Oriol Vinyals. Generating diverse high-fidelity images with VQ-VAE-2. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 14837–14847, 2019.

[60] Minhyeok Lee and Junhee Seok. Controllable generative adversarial network. *IEEE Access*, 7:28158–28169, 2019.

[61] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *CoRR*, abs/1411.1784, 2014.

[62] TensorFlow. `https://www.tensorflow.org/`, last accessed on 05/14/21.

[63] PyTorch. `https://pytorch.org/`, last accessed on 05/14/21.

[64] Pátrání po osobách - Policie České republiky. `https://aplikace.policie.cz/patrani-osoby/Vyhledavani.aspx`, last accessed on 05/14/21.

[65] Poszukiwani. `http://poszukiwani.policja.pl/`, last accessed on 05/14/21.

[66] Pátranie po osobách, Ministerstvo vnútra SR - Polícia. `https://www.minv.sk/?patros-index`, last accessed on 05/14/21.

[67] opencv/how_to_train_face_detector.txt at 3.4.0 · opencv/opencv · GitHub. `https://github.com/opencv/opencv/blob/3.4.0/samples/dnn/face_detector/how_to_train_face_detector.txt`, last accessed on 05/15/21.

[68] OpenFace. `http://cmusatyalab.github.io/openface/`, last accessed on 05/15/21.

[69] Brandon Amos, Bartosz Ludwiczuk, and Mahadev Satyanarayanan. Openface: A general-purpose face recognition library with mobile applications. Technical report, CMU-CS-16-118, CMU School of Computer Science, 2016.

[70] TFRecord and tf.train.Example — TensorFlow Core. `https://www.tensorflow.org/tutorials/load_data/tfrecord`, last accessed on 05/14/21.

[71] tf.data: Build TensorFlow input pipelines — TensorFlow Core. `https://www.tensorflow.org/guide/data`, last accessed on 05/14/21.

[72] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. StyleGAN2 — Official TensorFlow Implementation, 2020. `https://github.com/NVlabs/stylegan2`, last accessed on 05/14/21.

[73] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

[74] Kimmo Kärkkäinen and Jungseock Joo. Fairface: Face attribute dataset for balanced race, gender, and age. *CoRR*, abs/1908.04913, 2019.

[75] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 2019.

[76] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[77] Classification on imbalanced data — TensorFlow Core. `https://www.tensorflow.org/tutorials/structured_data/imbalanced_data`, last accessed on 05/14/21.

[78] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 2999–3007. IEEE Computer Society, 2017.

[79] Qizhe Xie, Minh-Thang Luong, Eduard H. Hovy, and Quoc V. Le. Self-training with noisy student improves imagenet classification. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 10684–10695. IEEE, 2020.

[80] Elad Richardson, Yuval Alaluf, Or Patashnik, Yotam Nitzan, Yaniv Azar, Stav Shapiro, and Daniel Cohen-Or. Encoding in style: a stylegan encoder for image-to-image translation. *CoRR*, abs/2008.00951, 2020.

[81] Karl Pearson F.R.S. Liii. on lines and planes of closest fit to systems of points in space. *Philosophical Magazine Series 1*, 2:559–572.

[82] scikit-learn: machine learning in Python. https://scikit-learn.org/stable/, last accessed on 05/14/21.

[83] Serving Models — TFX — TensorFlow. https://www.tensorflow.org/tfx/serving/serving_basic, last accessed on 05/14/21.

[84] Flask — The Pallets Projects. https://palletsprojects.com/p/flask/, last accessed on 05/14/21.

[85] BootstrapVue. https://bootstrap-vue.org/, last accessed on 05/14/21.

[86] EventSource - Web APIs — MDN. https://developer.mozilla.org/en-US/docs/Web/API/EventSource, last accessed on 05/14/21.

[87] Fetch API - Web APIs — MDN. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API, last accessed on 05/14/21.

[88] FileSaver.js: An HTML5 saveAs() FileSaver implementation. https://github.com/eligrey/FileSaver.js/, last accessed on 05/14/21.

[89] JSZip. https://stuk.github.io/jszip/, last accessed on 05/14/21.

[90] Lisa DeBruine and Benedict Jones. Face research lab london set, May 2017. https://figshare.com/articles/dataset/Face_Research_Lab_London_Set/5047666/5, last accessed on 05/14/21.

A. Attachment

The contents of the attachment:

- `/src` – folder containing the source codes of the models and LiGAN application.
- `/docs` – folder containing the user and API documentations for LiGAN application (offline version).
- `/setup` – folder containing files for preparing the environment.
- `/README.md` – file describing the contents of the attachment in detail. This file also contains some details about running the LiGAN and training scripts for the models, together with a more detailed description of hyperparameters.

The following links point to additional resources that were not directly included in the attachment:

- Repository with source codes is available at: `https://gitlab.mff.cuni.cz/dokoupipa/ligan`
- Archive with model checkpoints and pre-trained models is available at: `http://herkules.ms.mff.cuni.cz/ligan/models.zip`
- The dataset that was used for training the StyleGAN2 model is available at: `http://herkules.ms.mff.cuni.cz/ligan/dataset.zip`
- The sample of 100 000 images that were generated by the StyleGAN2 model and that were used during some of the experiments (e.g. the user study) in this thesis can be found at: `http://herkules.ms.mff.cuni.cz/ligan/generated_sample.zip`
  Note that this archive also contains a file with attribute annotations produced by CelebA and FairFace models.
- Training logs (raw and TensorBoard logs) that were captured during training the final versions of the models are available at: `http://herkules.ms.mff.cuni.cz/ligan/training_logs.zip`
- Latent vectors needed for PCA fitting inside of the backend container (saved as numpy array) are available at: `http://herkules.ms.mff.cuni.cz/ligan/pca_latents.zip`
- Models that are needed for running the dataset pipeline are available at: `http://herkules.ms.mff.cuni.cz/ligan/pipeline_models.zip`
  Note that these are not our models and they were included only for reproducibility.
- Running instance of the LiGAN application is available at: `http://gpulab.ms.mff.cuni.cz:7022/index.html`

- Online version of user documentation is available at:
  `http://gpulab.ms.mff.cuni.cz:7022/docs`
- Online version of master daemon API documentation is available at:
  `http://gpulab.ms.mff.cuni.cz:7022/master-daemon-api-docs`
- Online version of frontend server API documentation is available at:
  `http://gpulab.ms.mff.cuni.cz:7022/frontend-api-docs`
- Online version of backend server API documentation is available at:
  `http://gpulab.ms.mff.cuni.cz:7022/backend-api-docs`