Università degli Studi di Milano
FACOLTÀ DI SCIENZE E TECNOLOGIE

DIPARTIMENTO DI INFORMATICA
GIOVANNI DEGLI ANTONI

CORSO DI LAUREA MAGISTRALE IN
INFORMATICA

FACE PARAMETERIZATION THROUGH NEURAL NETWORKS

Supervisor: Prof. Stefano Ferrari

Elaborato Finale by:
Alfredo Russo
Matr. No. 967771

Academic Year 2023-2024

Contents

Chapter 1

Introduction

The reconstruction of a 3D face model starting from a 2D image is a complex and relevant challenge in the field of computer vision and computer graphics. The ability to accurately recreate a three-dimensional face model from a single two-dimensional image has profound implications in different applications like facial recognition, animation, virtual reality, and augmented reality. This task has different problems explored during the years where the loss of significant information like depth and texture information during image capture are the principal ones. In order to solve these problems, the development of sophisticated algorithms was necessary.

In recent years the advent of machine learning and artificial intelligence has revolutionized different fields especially the one related to computer vision. There are many approaches based on the use of machine learning, trying to improve the result obtained from the sophisticated algorithms previously mentioned. They tried to solve different problems related to the loss of details in the process of generating a 3D face model from a single 2D image exploiting machine learning. Anyway, the majority of those approaches are based on 2D images obtained from real world data. This thesis explores a novel approach trying to not use real world data and thus reconstruct a 3D face model from a single 2D image by leveraging neural networks trained on synthetic data generated through the MakeHuman platform.

MakeHuman is a an open-source software which allows the creation of 3D human model using a wide range of customizable parameters. These parameters range from general attributes like ethnicity and age to specific facial features and play a crucial role in the key focus of this work which is on automating the creation of a large dataset of facial images and corresponding parameters, which are then used to train a neural network for 3D face reconstruction.

The first part of this thesis reviews the existing literature on 3D Morphable Models (3DMMs) and alternative techniques such as Shape-from-Shading (SFS). Then it is explored how the approaches based on neural networks tried to solve the problem previously mentioned related to details loss. In the subsequent chapters this thesis presents the development of a custom plugin for MakeHuman needed for synthetic data generation. The plugin facilitates the selection, combination, and randomization of facial parameters, enabling the creation of a diverse set of human models that can be used for training purposes. In addition, this thesis will investigate the effectiveness of neural networks in parameterizing human faces, with a particular emphasis on the

regression problems associated with predicting facial parameters. Then it explored the architecture and design of the neural network models employed, discussing the challenges encountered during the implementation phase, such as the handling of a large number of parameters and the need for transfer learning techniques.

The results of this approach will be evaluated through a series of experiments and the results will be analyzed to identify potential improvements. This work contributes on the synthetic data generation and neural network-based facial recognition, giving basis to future improvements and more future advanced applications. The following sections will focus on review of related literature, describe the tools and methodologies used, and present the findings and conclusions of this study.

Chapter 2

Literature reviews

The problem of 3D face reconstruction from a 2D image is a challenging one which has gained more and more attention in the field of computer vision and graphics over the years. This interest is due to its wide range of applications, including facial recognition, animation, virtual reality, and augmented reality. However, during the image capture there is a loss of information which necessitates the development of sophisticated algorithms and techniques to obtain a high-quality reconstruction of a 3D model starting from a 2D image.

A pioneer approach was introduced by [1] with the introduction of a statistical method called **3D Morphable Models (3DMMs)** which capture the variability of facial shapes and textures within a dataset of 3D faces scans. This framework allows to generate new faces as a linear combination of learned basis shapes and textures and manipulate them. However, the framework struggles to capture fine-grained details that make every face unique and it is sensitive to the variation in pose and illumination. Another problem is related to obtaining the 3D face scans which is very computationally expensive. Anyway this work inspired numerous other works with the aim of improving the accuracy and robustness of this approach.

An alternative technique called **Shape-from-Shading (SFS)** is then emerged aiming to recover 3D shape from a little variation in shading present in a single image [5]. This approach excels at capturing fine details and geometry, but sometimes it requires strong assumptions about lighting and surface reflectance. The ambiguity of SFS solutions, combined with sensitivity to noise and image artifacts, has posed significant challenges in obtaining accurate and reliable 3D facial reconstructions.

The introduction of the **Deep neural Networks (DNNs)** has revolutionized the field of computer vision and 3D reconstruction isn't an exception. DNNs with their ability to learn complex nonlinear mappings, offer the opportunity to overcome the limitation of the previous approaches presented. Many of these approaches focus on using DNNs in order to predict 3DMM parameters or depth maps from a single 2D image, demonstrating impressive improvements in reconstruction accuracy and efficiency. In the paper presented by [3], it uses an end-to-end DNN framework that directly predicts 3DMM parameters, reducing the computational cost and simplifying the reconstruction pipeline. Another approach based on the use of DNNs is the one presented by [8]. It introduced a coarse-to-fine network architecture where an initial 3D face is estimated using 3DMM

and then is used a refinement network to capture fine details which are the ones that 3DMM struggle to capture. Another similar work is the one explored by [6] where an initial 3D reconstruction is obtained from 3DMM, then two different networks are used, trying to improve the result. The first one focuses on generating a displacement map in order to capture details missing from the initial reconstruction. The second one is in charge of reconstructing hair, a notorious more difficult challenge due to their complexity and variability.

Some approaches, like the one presented by [4], don't rely on the 3DMM framework and thus they leverage on the power of DNNs. The approaches presented are related to the use of two different neural networks, a **Residual Network** and a **MRF-NN (Markov Random Field Neural Network)**. The first one has the aim to predict a depth map from a 2D image, while the second one is in charge of refining the reconstruction by incorporating local geometric constraints. This dual network approach tries to overcome the limitation of traditional methods that may struggle to capture fine-grained details and handle geometric distortions. Another similar approach is the one presented by [7] where the first network generates a depth map, while the second one utilizes SFS techniques, trying to improve the model generated from the first network.

Even if the improvements were significant with the advent of DNNs, several challenges persist in 3D face reconstruction. The main one was related to limited availability of large-scale labeled 3D face datasets for training, undermining the generalization capabilities of these models. Furthermore, the reliance on 3DMM leads to difficulties in capturing fine-grained details and dealing with problems related to different poses and expressions. An attempt to face these problems can be found in the work presented in [9] utilizing synthetic data for training DNNs. The creation of a synthetic dataset is explored in the work presented by [2], which introduces a parametric model for generating diverse face drawings and offers a potential avenue to overcome the limitation of real-world data collection and annotation.

This topic is further explored during this work in order to try to understand how a neural network trained only with synthetic data behaves with real world data.

Chapter 3

Tools for synthetic data generation

The tool used to achieve the goals of creating a suitable set of images and corresponding parameters for training a neural network is MakeHuman. This open-source application allows the creation of human models through a variety of customized parameters. Since the objective is to generate a set of facial images and their related parameters, only a portion of the application will be utilized to accomplish this goal.

3.1 MakeHuman

The models in MakeHuman are defined by a text file with the extension `.mhm`, that contains all the parameters and their corresponding values, along with additional information to customize the model based on the default template. The parameters are categorized into different macro classes, but for the purposes of this work, only two classes are relevant. The first class pertains to macro parameters, while the second class is related to face-specific parameters. The default model is the one where all the face parameters assume value 0 and when it is saved the resulting `mhm` file contains only the information related to the macro details, while the face specific parameters are not included. It is possible to change the appearance of the human model by adjusting the parameters value through the MakeHuman UI. As mentioned previously, the parameters are divided into classes and they are all presented with sliders for easy modification. Face-related parameters range from -1 to 1, while macro detail parameters range from 0 to 1. Due to the large number of face parameters, manually trying different configurations to generate the dataset is impractical. Therefore, it is necessary to automate the generation of all required files and images through code.

MakeHuman is an ideal choice for this work because it supports the development of custom plugins, which can be written in Python, just like the application itself. This capability allows the automation of the dataset creation process. Another advantage is that it allows exporting the created models in various formats such as Collada, Wavefront, or FBX, which can then be imported into Blender for rendering the images. This aspect, along with the reasons why these formats will not be used, will be discussed in section 3.3.2.

3.2 Parameters extrapolation

To start plugin development that will generate data for the dataset, it is necessary to identify the relevant parameters, their names, and their actual identifiers, which differ from the ones used in the UI. MakeHuman provides an API that simplifies access to various functionalities, including retrieving all parameter names related to a human model.

The parameters follow a specific format where the name of the parameter is preceded by the class to which it belongs. For example, if we are dealing with a nose parameter, it will be in the form `nose/parameter-name`. For macro details, the naming convention varies, some parameters use the macro area name **macrodetails**, while others have their own distinct macro area names. More details about this will be discussed in the following sections.

The API, known as the MakeHuman API (mhapi), allows to retrieve all parameters related to a human model and write them to a file. The simple script used to accomplish this is as follows:

```
def write_params_to_file(path):
    f = open(path, "x")
    params = G.app.mhapi.modifiers.getAvailableModifierNames()
    f.write("\n".join(str(i) for i in params))
    f.close()
```

Figure 1: An example code where it is highlighted how it is possible to retrieve a list of all the parameters available in MakeHuman.

This script was executed within the MakeHuman application using a plugin called **scripting**, which allows writing, loading, and saving scripts directly inside the application. This plugin, combined with the **execute** plugin, enables the execution of scripts within the MakeHuman environment.

Once all the parameters were retrieved and written to a file, it was necessary to distinguish the parameters related to human faces from the others. Since the parameters are divided into macro areas, all parameters associated with macro areas relevant to human faces were selected. This process identified a total of **138** parameters related to all components of the human face.

3.3 Plugin development

A plugin in MakeHuman can be described as a set of Python scripts that includes an entry point file called `__init__.py`. Inside this file, there must be a class that inherits from the `TaskView` class, allowing to set the name of the view and create the UI for the plugin. Additionally, a method called `load` is required outside the class. This method specifies the category to which the plugin belongs, adding the plugin's task view to this category. The code looks like the one in the figure 2.

As it is possible to see, the plugin task view is called **Face Parametrization** which is added to the category **Utilities**.

```
class FaceParametrizationTaskView(gui3d.TaskView):
    def __init__(self, category):
        self.gui3d = gui3d
        gui3d.TaskView.__init__(self, category, "Face Parametrization")
        self.__create_ui()

    def __create_ui(self):
        CreateUI(self)

def load(app):
    category = app.getCategory('Utilities')
    category.addTask(FaceParametrizationTaskView(category))
```

Figure 2: This code highlights how it is possible to add a custom tab to MakeHuman through the custom plugin development.

The plugin is organized into two different macro areas. The first area focuses on the creation of various human models, which in the MakeHuman context corresponds to generating different `mhm` files. This part includes various sections that are useful for selecting parameters of interest, as well as options for random selection, generation, and other relevant parameters. The second area involves creating model images based on the previously generated `mhm` files. Both areas will be discussed in more details in the following sections, along with the choices made during the plugin development.

3.3.1 Human generation

The initial approach to generating synthetic data for the dataset involved introducing functionalities for manually selecting parameters through a tree view as it is possible to see in image 3.

This method creates files containing all possible combinations of parameters and their values by specifying a `step` value. The `step` value, a float, determines the interval between values, starting from the lower range bound. For example, if the `step` value is 0.25, the possible values a parameter can assume are $[-1, -0.75, -0.5, \ldots, 1]$. This value is represented by a slider in the UI.

As can be understood, it is computationally expensive to calculate all possible combinations of parameters given a certain step value. For example, if 8 parameters are selected with a step of 0.25, the number of possible combinations is 43,046,721. This number corresponds to the number of `mhm` files, which in MakeHuman represent the number of human models. Another issue with this approach is the memory usage. The size of the generated `mhm` files increases with the number of parameters chosen. For instance, a `mhm` file with 8 parameters is approximately 1 kB in size. Given the previously calculated number of files, the total size would be 43,046,721 kB, which is approximately 42 GB for just 8 parameters and a step value of 0.25. It is possible to observe how the number of combinations increases as the number of chosen parameters increases in the image 4. This approach can be used for handling a small number of parameters, in order to set up the

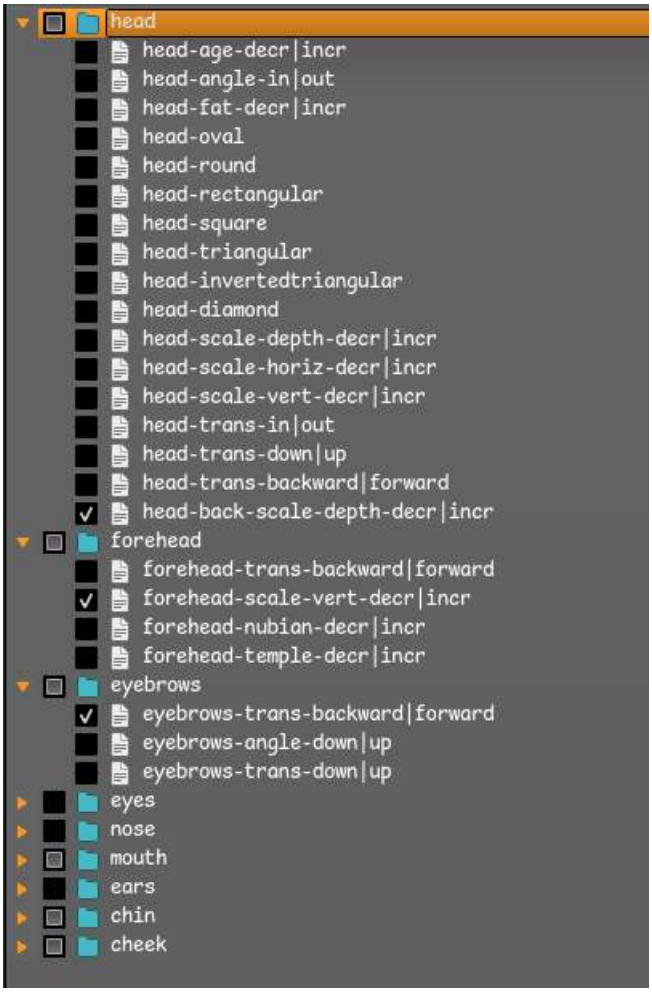


Figure 3: TreeView section.

pipeline and observe how the network behaves with data composed of a limited number of it. This will be discussed deeply in the following chapters.

Inside the tree view, the parameters related to the macro details of the human model are excluded because they use a different range than the usual one. Specifically, these parameters range from [0, 1], whereas the usual parameters range from [-1, 1]. Therefore, within the plugin, there is a dedicated section for selecting the values of these parameters. This implies that the macro details chosen will remain consistent across all text files generated during a specific execution, unlike the face parameters. An overview of this section can be seen in the image 6.

The editable parameters related to the macro details include **ethnicity**, **gender**, **age**, and **weight**. UI elements were chosen according to the type of data to modify. Additionally, buttons are available to randomly select the values of the macro details or reset them to their default values.

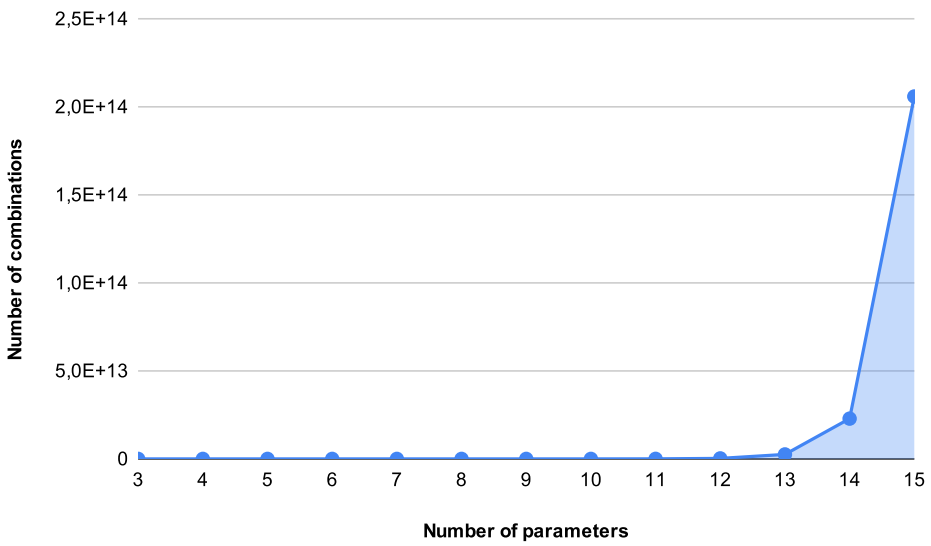


Figure 4: This graph shows how the number of combinations increases as the number of parameters increases.

Another useful functionality is the ability to randomly select the face parameters. This feature is beneficial as it allows for the random generation of the human model, providing insight into how the neural network performs with parameters that are not chosen according to specific criteria. The plugin permits the user to decide the number of parameters to randomly select, which can be any number between 2 and 7. This range was chosen because selecting only one parameter does not allow for the generation of a sequence of files, while attempting to generate files with more than 7 parameters can lead to excessive RAM usage and it can be computationally expensive. The feasibility of this feature is certainly dependent on the processing power of the machine running the plugin. The look of this part can be seen in the image 6.

At a certain point, it became necessary to implement a functionality that allows loading parameters from a JSON file. This was essential because the network was handling too many parameters and could not predict all of them correctly; this issue will be discussed in more detail in section 4.3. The structure of the JSON file is straightforward, as the data models are already organized in a way that makes JSON the ideal solution. Each parameter belongs to a specific class, so the elements inside the JSON file consist of a key representing the parameter class and a value, which is a list of all chosen parameters. To better understand the structure of the JSON file, refer to the image 5.

After loading a file using the path selector within the plugin, the specified choices are displayed in the tree view. The selection of parameters through a JSON file is usually associated with choosing many parameters and ensuring that the selection persists across sessions. Since it often involves numerous parameters, it is impractical to use it with the generation functionality based on the selected step value. Therefore, it is intended to be used with the next functionality, which

```
{
    "parameter_class_1": [
        "parameter_1a",
        "parameter_1b",
        ...
    ],
    "parameter_class_2": [
        "parameter_2a",
        "parameter_2b",
        ...
    ],
    ...
}
```

Figure 5: Example of the JSON file structure for parameter organization.

is the random generation.

In the UI, there is a checkbox that, when checked, reveals a section below it. This section allows the user to select the number of `mhm` files to generate based on the chosen parameters. The values for all selected parameters are generated randomly using a MakeHuman functionality, which generates parameters value randomly but within certain constraints to ensure the creation of plausible models. The overall appearance can be seen in the image 6.

3.3.2 Render human through Makehuman

The plan is to use Blender to render the model created in MakeHuman and take a picture of it. MakeHuman provides the functionality to export the model in various formats, such as **Wavefront**, **Collada**, and **FBX**.

Wavefront is one of the most widely used formats, making it the best choice if you are working with different applications that need to use this file and it doesn't consume much memory. However, due to its simplicity, this format does not support rigging, animations, or other advanced features. The **Collada** format is open-source, so it can be used freely and supports advanced features like animation and rigging. However, it can be very verbose, resulting in files that consume a lot of memory. The **FBX** format falls in between the two previously mentioned formats. It offers all the functionalities of Collada, and the memory occupied by these files can vary according to the model complexity. However, it is not an open format, which can be problematic when working with different applications. Anyway it is supported by both MakeHuman and Blender so this wouldn't be a problem. Even though the FBX format might seem the best choice, the Wavefront format is chosen. This is because it is the simplest format, which helps to speed up performance. Initially, the approach involved exporting the model in one of these formats and importing it into Blender using **bpy** (Blender Python API) to capture an image of the model.

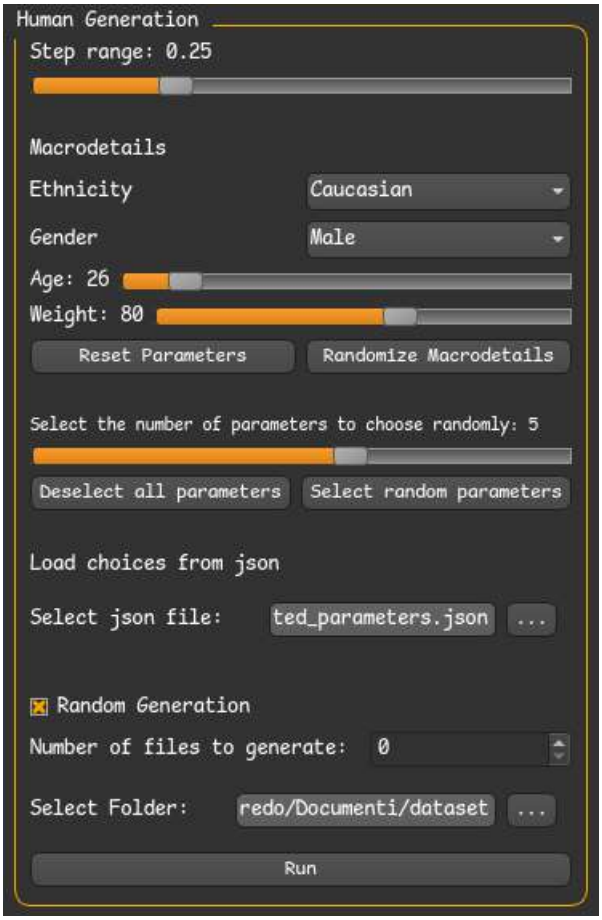


Figure 6: Human generation box.

Bpy can be seen like Blender as a python module, allowing users to perform most Blender tasks through scripting. This capability is particularly well-suited to our needs, as it enables the generation of images via the MakeHuman UI. It is used for opening a file in one of the supported formats in blender, then a light is positioned in front of the model loaded and the camera moved in the right position as well as its rotation adjusted properly. Once it is all set up the picture is taken and the file produced is saved accordingly to the path passed in. Every time a file is open and the model loaded inside blender, in order to load the next file and take the picture of the model, it is necessary to remove the previous model loaded otherwise the RAM will be filled resulting in an application crash.

The UI related to the image generation is very simple, consisting of a field to select the folder containing the previously generated text files and another field to select the resolution of the images that have to be generated. The resolution which can be selected, just goes from 16x16 to 512x512. This choice was made since they are the most useful resolutions to build up the pipeline and understand if the hardware used can handle the model learning process at specific resolution.

There is no option to choose the destination for saving the images, as they will be saved in the same location as the text files. An overview of the UI can be seen in the image 7.

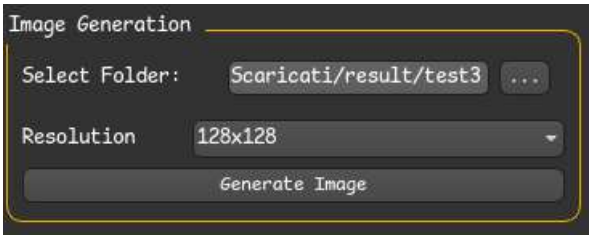


Figure 7: Image generation box.

MakeHuman, through its API, allows exporting the human model loaded within the application in various formats, besides the one previously mentioned. To test the feasibility of this approach, the Wavefront format was used. However, there is a significant issue, to export a model in the desired format, it must first be imported into the MakeHuman application. This process consumes a substantial amount of time and represents a bottleneck in the image generation pipeline. Additionally, the time required by the MakeHuman API to export the model is considerably longer than the time needed by the Blender plugin, which will be discussed in section 3.3.3. The duration of loading and exporting the model is not influenced by the number of parameters in the `mhm` file, and consequently, the model they describe, as the execution time does not improve regardless.

Due to inefficiency of the model loading inside MakeHuman this approach cannot be used since it took too much time to export all the files in the desired format and then take a picture of them.

3.3.3 MakeHuman Plugin For Blender 2

The approach explored during the previous section didn't yield to an optimal and usable results, so between the different solution explored to this problem, one of them is trying to use the **MakeHuman plugin for blender** or **MPFB2**.

In order to extend Blender functionalities it allows importing addons. The addons generally can be divided into two different groups, the ones developed by the Blender developers which are seen as "Official addons" and the one developed by the community. The official ones are already inside the application and in order to use them it is only needed to activate them through the Blender UI. The community ones have to be first installed in order to be used and this can be done in the proper section inside the Blender application.

MakeHuman is a standalone GUI application which was released more than 15 years ago, meanwhile Blender gained a lot of notoriety and it was improved a lot over time. So the idea of the MakeHuman community was to exploit the blender functionality to further improve the human generation. In order to do that they developed the first version of the plugin for Blender in 2018. This version was not intended to replace MakeHuman at all and it allows importing the models created thanks to MakeHuman in Blender in order to perform advanced modifications that are not allowed in MakeHuman.

Then a second version of the addons was developed in 2022 and it can be seen as a MakeHuman reimplementation as a Blender addons. This plugin just uses Blender functionalities for the majority of the 3D modelling tasks. It allows to do all the modification which MakeHuman is capable of, hence even the functionality to load a model from a mhm file which is the needed functionality.

MPFB2 is composed of different services which are basically classes containing only static methods, which means that there is no need to instantiate them. The service interested is called **HumanService** which allows to parse the mhm file and create a mesh which is loaded inside Blender allowing to use bpy to take a picture of the face. This addon is developed with the intention to use it through Blender UI with the design task to make it usable even through python scripts. However, it doesn't work right out of the box. The first step is to install and enable this addon in bpy. To do this, I used two different scripts. The first script handles the installation of the addon, while the second script is used to enable it. The installation script only needs to be run once, but the enabling script must be executed every time before using **MPFB2**. To obtain the mesh, the **HumanService** needs that inside bpy has to be installed and enabled another addons which is related to handle Wavefront, thus it is only installed once using the previous script and enabled every time MPFB2 is used.

At this stage, another problem arises, the addon has a memory leak of approximately 88 MB every time it is used. Since the addon is used to load multiple models at every execution, it results in significant memory waste. This, in turn, causes an increasing slow down each time a picture is taken, leading to the application crashing. The addon is very articulated and finding where the memory leak happens is a really hard task to solve. The memory leak happens even if no service is used with an increase of memory wasting according to the actions performed. To face this problem, the solution was to keep only the **HumanService** script and the ones which it depends on. Doing that, the memory leak disappeared and finally it was possible to include the functionality of loading `mhm` files and taking a picture of the models generated inside the MakeHuman plugin.

3.3.4 Results

Thanks to the **MPFB2** approach, the performance improved significantly compared to the initial approach. In order to generate 200 photos with a resolution of 128x128, from the related mhm files, the time needed is around 328 seconds which means approximately 6 minutes. For generating 500 pictures, the time required is about 765 seconds, which is less than expected. As it is possible to see in the graph 8 the time is quite linear as the number of the photos to generate increases.

In order to generate 200 photos with the first approach it is needed 481 seconds which means 8 minutes, while for generating 500 photos 1231 seconds are needed, around 20 minutes. Even in this case the growth is linear, even if it is more steep, see graph 9. The steepness of the graph just highlights that the first approach scales worse than the MPFB2 one.

Both of the previous times are taken considering all the parameters related to the face, which are 138. In the figure 10a and 10b it is possible to see the generation time of the images with only 7 parameters involved, which is considerable less than the previous ones, but only for the approach which involves the addon. Anyway the result obtained is the same, with the MPFB approach having a better time execution.

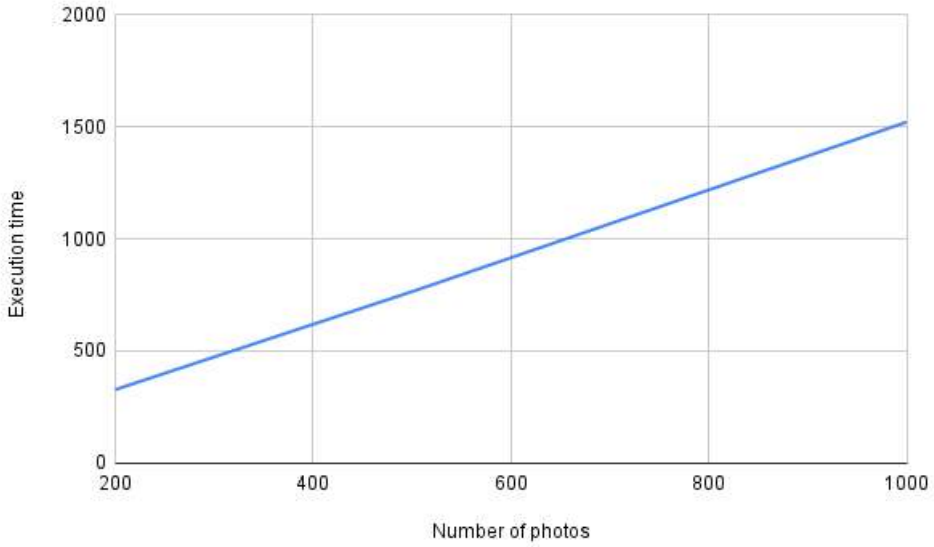


Figure 8: MPFB approach.

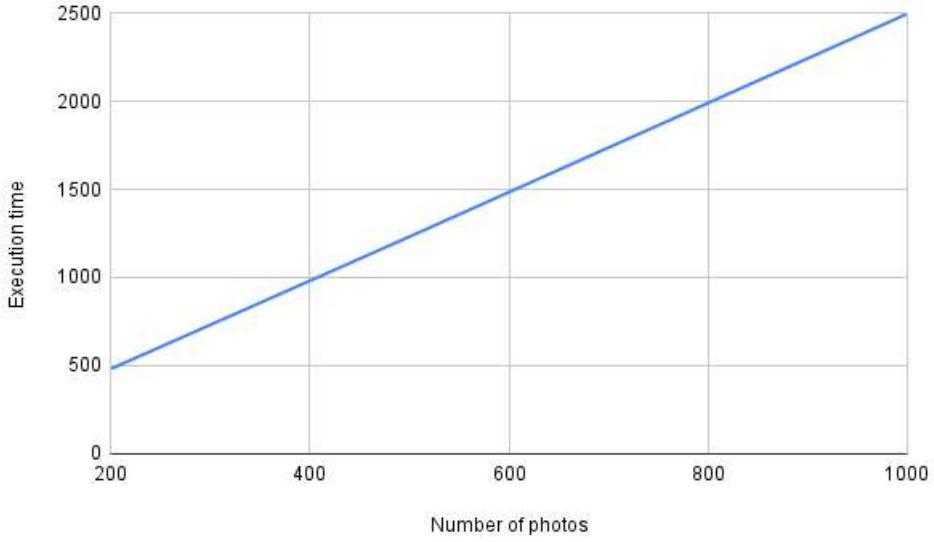
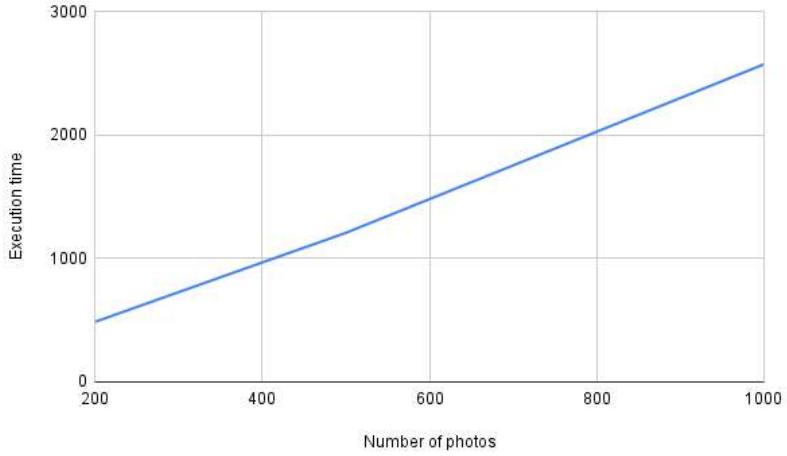
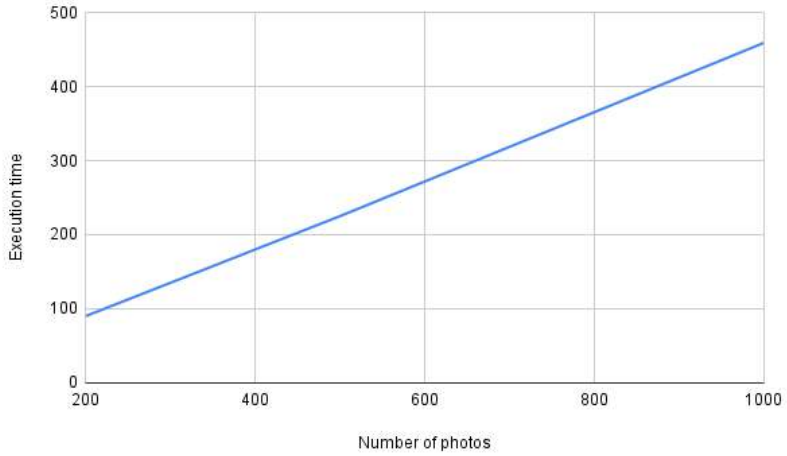


Figure 9: First approach.

The graph related to the first approach highlights that the time is not related to the number of parameters involved, at least for this approach. Taking in account all the parameters in order to generate 1000 images it is needed 41 minutes, while for generating the same amount of images but involving only 7 parameters the time needed is around 43 minutes. The time which involves less parameters just needs more or the same execution time in order to complete the task.



(a) First approach



(b) MPFB approach

Figure 10: Execution times.

Chapter 4

Model Architecture and Design

The problem faced by the model is related to predicting face parameters starting from an image. The parameters are related to the ones in the MakeHuman application, thus the problem is a regression one.

4.1 Regression problems

Regression is a technique used in machine learning in order to model the relationship between a dependent variable and one or more independent variables. There are different kinds of regression with different characteristics and applications. They are:

- **Linear regression**: Aim to model the relationship among the independent variables and the dependent variable as linear combination of the independent variables like the following:

$$y = B_0 + B_1 x_1 + B_2 x_2 + ... + B_n x_n + \epsilon \tag{1}$$

  where ϵ is the error between the real value and the one predicted.

- **Logistic regression**: It is usually used for binary classification.

- **Polynomial regression**: It is very similar to the linear regression with the addition of polynomial terms:

$$y = B_0 + B_1 x + B_2 x^2 + ... + B_n x^n + \epsilon \tag{2}$$

- **Ridge regression**: It's a kind of regression which includes a form of penalization in order to reduce overfitting. It is useful when dealing with a lot of independent variables.

$$y = B_0 + B_1 x_1 + B_2 x_2 + ... + B_n x_n + \lambda \sum_{i=1}^{n} \beta_i^2 + \epsilon \tag{3}$$

- **Lasso regression**: It is similar to the ridge regression but it uses a different kind of penalization which leads coefficients to zero, reducing the overfitting.

$$y = B_0 + B_1 x_1 + B_2 x_2 + ... + B_n x_n + \lambda \sum_{i=1}^{n} |\beta_i| + \epsilon \tag{4}$$

- **Multivariate regression**: It is an extension of the linear regression where there are more than one dependent variable to predict. It can be expressed as:

$$\mathbf{Y} = \mathbf{XB} + \mathbf{E} \tag{5}$$

  where:

  - **Y** is the matrix of the dependent variables.
  - **X** is the matrix of the independent variables.
  - **B** is a coefficient matrix which describes the relationship between the dependent and independent variables.
  - **E** is a matrix error which represents the difference between the observed values and the ones predicted.

The problem faced can be classified like a **Multivariate Regression** since there are more than one dependent variable to be predicted starting from a set of independent variables. More deeply, the independent variable can be seen as the image pixels, even if they won't be treated individually according to the model chosen. Instead the 138 parameters that have to be predicted can be seen like the dependent variables. Since the number of dependent variables that have to be predicted are more than one, as previously said, the problem is a Multivariate one.

4.2 Neural Network

The neural network chosen for facing this problem is a **Convolutional Neural Network (CNN)**. This kind of network is particularly suitable for the task faced, since the images are high dimensional data which this kind of network can handle well thanks to its convolutional layers. These layers use the convolution in order to extrapolate information from the image like borders, textures and other features.

4.2.1 Convolution Operation

In the field of image processing the **convolution** operation is used to apply a certain filter to images or other image transformations. This operation is a mathematical one and consists of sliding a matrix called kernel or filter over all the pixels of the images performing a dot product at each point where the filter overlaps the image.

The matrix related to the kernel is chosen arbitrarily according to the kind of operations wanted. There are different kinds of kernel matrix and each of them permits to perform different image transformations. An example of the convolution can be seen in the image 11.

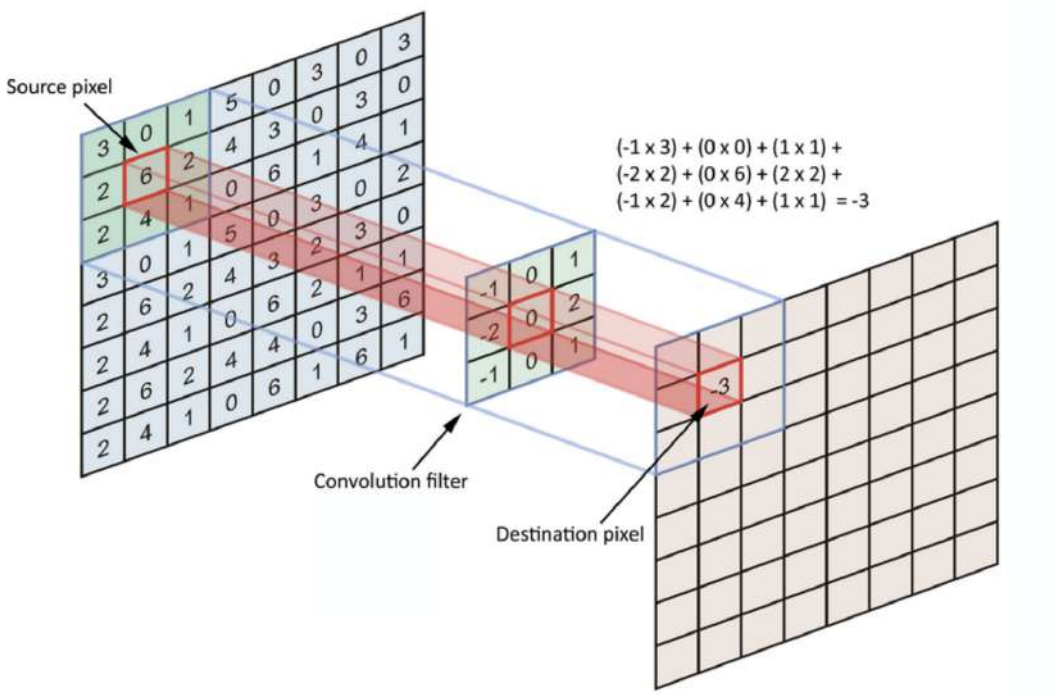


Figure 11: Convolution example.

During the convolution, one of the problems to face is when the center of the kernel is overlapped to a border pixel of the image. As it is possible to understand, doing that leads to having some values of the kernel matrix out of the image matrix, so it is needed to handle the problem, otherwise the convolution cannot be done properly. There are different techniques which allows to handle this problem and they are:

- **Zero Padding**: This is the simpler method, which involves adding zeros around the edges to enable the convolution process. Anyway can add artifacts along the border of the resulting image.
- **Reflection Padding**: The pixels at the border are duplicated and reflected outwards. In this case the artifacts are introduced only if the pixels at the border are not homogeneous.
- **Replication Padding**: In this method the pixels on the border are just duplicated and added outside the matrix. It is really simple, but it can add artifacts if the borders are really complex.
- **Circular Padding**: The edges of the image appear connected as if the image were wrapped around a cylinder. It is not a good choice with non-periodic images.
- **Valid Padding**: No padding is added and the convolution is done only where possible and thus not at the border, but the resulting image has a smaller size.
- **Constant Padding**: It is really similar to zero padding, but the value added can be chosen by the user and it can be different from zero.

There is no one-size-fits-all solution and all of them have pros and cons, it depends on the situation. So every time it is needed to choose the best techniques according to the kind of data involved and the objectives to reach.

4.2.2 Transfer Learning

Transfer learning is a technique which allows to use a pre-trained model and adapt it to a specific task. The first thing to do is to choose a pre-trained model according to the task to face. Then the last layer of the model is removed, since it is specific to the original model, and replaced with a new layer specific to the task to do. In this scenario the goal is to predict the 138 parameters of the human face so the last layer of the model should be removed and replaced with one according to the needs. Then the model has to be trained with new data.

This technique is really good if the dataset is not large enough. Since the model already knows the relevant feature, the time required for reaching a good result is significantly reduced than training a model from zero. Anyway the majority of the pre-trained models have been trained using a dataset composed of real images which is far from the purpose of my work. The goal of it is to train a model with synthetic human faces and observe how it works with real ones. For this reason this technique has not been used and the choice of creating a new model is made. Of course, the model is not created from scratch, the **PyTorch** library is used to achieve the goal.

4.2.3 PyTorch

PyTorch is an open-source library with a focus on machine learning. This is really widespread for its simplicity and flexibility. The building blocks of PyTorch are the tensors which are similar to the numpy arrays but they can be used on the GPU in order to speed up the computation. When dealing with machine learning, the ability to use the GPU is really important since it significantly reduces the execution time of the operations to perform. This library allows to define in a very straightforward way a neural network and work with it. To accomplish this, a class must be created that inherits from one of the modules provided by the library, specifically `nn.Module`, and the necessary methods must be defined. In the following section the structure of the model created using PyTorch is explored.

4.2.4 Structure

The structure of the network is a common one, it is composed of convolutional layers, pooling blocks and fully connected blocks. During the training the image goes through all these layers and blocks and the model is supposed to catch the feature from the image and generate the desired output, 138 face parameters. An example of a generic network can be seen in the image 12.

The neural network consists of three convolutional layers, each conducting convolutions on the input image. This process aims to extract features from the image while eliminating redundant information. For instance, the three convolutional layers may be employed to extract general edges and textures, capture more intricate features like shapes and eyes, and identify specific facial features such as the position and shape of the mouth, nose, and eyes. The number of convolutional layers can vary according to the difficulty of the task faced. The beginning layers learn basic

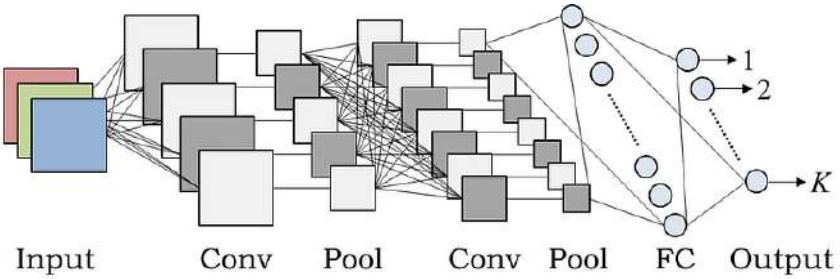


Figure 12: CNN example.

features like border and texture, while the deeper layers capture more complex features previously mentioned. The more layers are, the more the model learns complex features and its ability to generalize on new data is improved. However, having a lot of convolutional layers is not always a good thing since they require more computational power, and if the dataset is not large enough, the model can easily overfit. Overfitting is a term used to describe when a model learns the details of the training data too well, so it fails to generalize on new data. In **PyTorch** when it is created a convolutional layer, it is needed to pass the following parameters:

- **Input channels**: It refers to the channel of the input image. If it is an RGB image then the channels are 3.
- **Output channels**: This one refers to the number of the channels wanted in output which refers to the number of filters applied during the convolution in these layers. Every filter produces an output channel.
- **Kernel size**: It is the size of the kernel matrix which is slid over the image. To specify this, it is also possible to use a tuple if the kernel has different dimensions for height and width.
- **Stride**: The stride just denotes how many pixels the kernel is moved every time a convolution is done. If the stride is one means that the kernel is moved one pixel at time. If the stride is increased the size of the output will be decreased.
- **Padding**: This number refers to how many pixels it is needed to add around the image. As previously mentioned inside the section 4.2.1, this is really important in order to perform the convolution and manage its output size.

Then the model is composed of two **pooling blocks**. These layers have the aim to reduce the dimension of the image preserving the most important features. Thus a pixel should represent the image as the best as it can. The operation performed here is similar to the convolutional one indeed the needed parameters in Pytorch are the same, but there are no padding techniques since the aim of this operation is to reduce the image size. As it is done for the convolution, there is a sort of kernel, which is called **spatial extent**. It is slid over the image and the resulting value is determined according to the technique chosen. There are different kind of them and they are:

- **Mean pooling**: The resulting value is calculated as the average from the values involved.
- **Max pooling**: The maximum value is chosen between the ones involved.
- **Min pooling**: The minimum value is chosen between the ones involved.

It is possible to see in the image 13 how the pooling works, in this case the **Max pooling**.

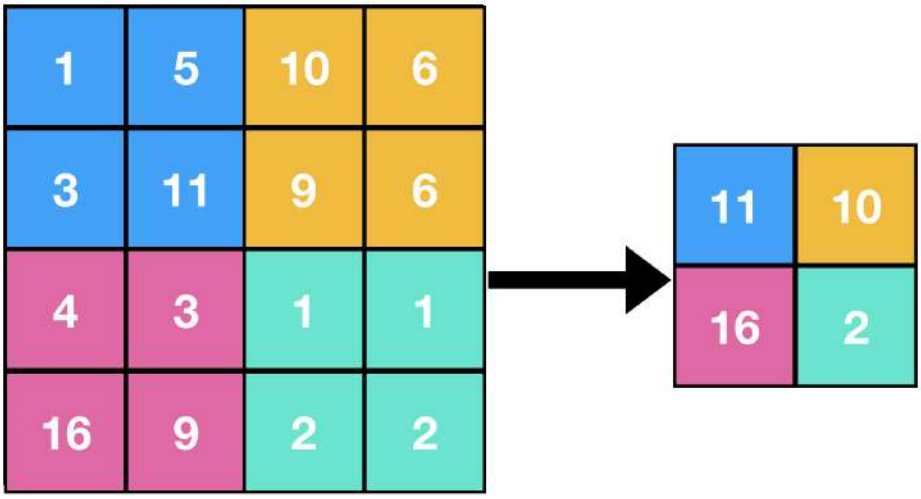


Figure 13: Max pooling example.

At the end, we have the fully connected blocks, which receive data that has been flattened beforehand. The flattening is done in order to reduce the dimension of the data processed by the convolutional and pooling layers in a mono-dimensional array. These blocks have the aim to aggregate data in order to create a global representation of them. The last block usually creates an output according to the one needed. It has to be specified during the creation of it and in this case the output has to be 138 values which refers to all the parameters to predict.

Inside the class related to the model it is needed to implement also the **forward** method. It describes how the input is processed and goes through all the layers of the network. Every time the input goes through the convolutional layer and before being processed by the pooling layer a function called **ReLU** is used. The goal of this function is to decide which neurons have to be activated and which one needs to be inactive according to the input received. The function is defined as:

$$ReLU(x) = max(0, x) \tag{6}$$

This function returns the input if it is positive, 0 otherwise. There are various types of activation functions, but this one is generally preferred due to its simplicity and ease of calculation, making it less computationally expensive.

Between the fully connected block a technique called **dropout** is used in order to prevent model overfitting. This technique consists of disabling randomly certain amounts of neurons in a specific layer. In order to perform this technique, a value has to be specified which is the probability p of a neuron to be disabled or not. As well as preventing overfitting, it also allows the model to learn how to better generalize not seen data.

Another important thing related to the model is the **loss function** used. It is really important since it allows to measure how far the predicted values are from the actual ones. There are different kinds of loss function and the choice of it depends on the problem faced. The problem faced during the work, as mentioned in the section 4.1, is a regression one and for this kind of problem is usually used the **Mean Square Error (MSE)**. It is defined as the average of the squares of the differences between the predicted values and the actual values:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{7}$$

where:

- n is the number of values.
- y_i is the real value for the parameter i.
- $\hat{y}_i$ is the predicted value for the parameter i.

This loss function is particularly suitable for the problem faced since there are 138 parameters that have to be predicted and MSE provides a clear estimate of the average difference between predicted and actual values. Since it is the quadratic of the differences, it penalises the bigger mistakes than the little ones and this is really good for the purpose of the work since a bigger error means a completely wrong representation of the face model. The aim of the training process is to minimize this value over the epoch trying to achieve **convergence**.

The **optimization function** plays a critical role in training neural networks by minimizing the loss function through iterative adjustments of the model's weights. Gradient descent is a fundamental optimization algorithm used to minimize the loss function by updating the weights in the direction of the negative gradient. The Adam (Adaptive Moment Estimation) optimizer is used due to its effectiveness in handling large datasets and complex models such as CNNs leading to accurate predictions of the 138 face parameters. In PyTorch when the optimizer is instantiated it is needed to specify a parameter called **learning rate**. This parameter is really important for the purpose of the training. This has the aim to decide how much the model weights are changed in response to the estimated error. A smaller learning rate can lead to a more precise convergence but can require more time for learning, while a bigger one can accelerate the learning process but can lead to not optimal performance of the model. This parameter needs to be chosen carefully as it can change the learning result of the model. Convergence occurs when the loss stops decreasing significantly with further iterations. At the beginning, the loss is usually very high because the model makes random predictions. During the epochs, the model learns more and more, and once it has learned everything possible, the loss stops decreasing significantly, indicating that convergence has been achieved.

In order to prevent the overfitting of the model, another technique is used called **early stopping** in addition to the dropout one. It consists of stopping the training before the model starts to go in overfitting. In order to understand if the model is going in overfitting, a different set of data than the one used for the training is used. This has a direct implication on how the dataset is organised, but this will be discussed in the chapter 4.3. Anyway, this technique is related to the loss function, as previously said this function is used in order to understand how far are the predicted values from the actual ones. Through the training epoch this information denotes if the model is improved or not. If the loss isn't improved, it means that the model has started to overfit the training data. The number of epochs in which the loss doesn't improve is called **patience** and it is used to understand when it is needed to stop the train. When it is interrupted the best model obtained during the latest epochs is saved. This technique is really useful to handle the overfitting problem, as well as improving the performance since the training is interrupted when the model finishes improving.

4.3 Dataset

The creation of a dataset through the use of MakeHuman is the goal of this project. The dataset has to be composed of a couple of files, the `mhm` files and the related images. In order to generate this kind of data it is used the MakeHuman application and the related plugin developed and described in the chapter 3. The idea is to have the file and the image with the same name but with different extensions, the images must be in the `PNG` format.

The dataset must adhere to a specific structure, consistent with the developed model. Various techniques discussed previously necessitate a certain organization of the data. The main directory can contain as many sub folders as needed, but a specific pattern must be followed at some point. This pattern includes three main folders, **train**, which contains the data used for training; **test**, which stores all the data used for testing; and **validate**, where all the data required for early stopping techniques are found. The model is evaluated on this data, thus the loss is calculated on them and if the model doesn't improve over time the training process is stopped. When the data are created, they are divided into three folders according to the following percentage:

- **train**: 80%
- **test**: 10%
- **validate**: 10%

Typically, for a classification problem, the dataset consists solely of images. To determine if the model has accurately predicted the class, the label of the folder where the images are stored is used. Nevertheless the problem faced in this project is not a classification one but a regression one and the dataset is more complicated. It is really trivial to implement and use a dataset for the classification problems, however it's not the same for a regression problem.

4.3.1 PyTorch implementation

The complexity of the dataset led to the creation of a custom dataset class which inherits from the one in PyTorch. It allows the creation of two kinds of dataset types which are the **iterable dataset** or **map-style dataset**.

The iterable dataset is used for dataset which don't have a fixed size where the data are generated dynamically. In order to create this kind of dataset it is needed to inherits from the `IterableDataset` class in PyTorch and define the method `__iter__` where it is possible to define the behaviour wanted. Anyway for the project purpose it is used a map-style dataset.

According to the PyTorch documentation the map-style dataset is the one that implements the `__getitem__` and `__len__` protocols. The first method allows access to the elements of the dataset through an index or a key since in this kind of dataset the index can also be a string rather than a number. This method permits also to specify the behaviour wanted when processing the data. In this case every time it is called, an image and the related parameters need to be returned. The image is straightforward since it has only been converted to RGB without the alpha channel. However, it is important to specify how to handle the parameters, as the method returns two parameters, the image and the set of face parameters.

In order to keep the management of the parameters not too complex it used a simple array filled with the values of the parameters itself. Thus when a `mhm` file is processed it is produced an array with the values of the parameters read from the file. An example of the array produced can be seen in the image 14. Of course the values can be in the range [-1, 1] according to what is said in the chapter 3.

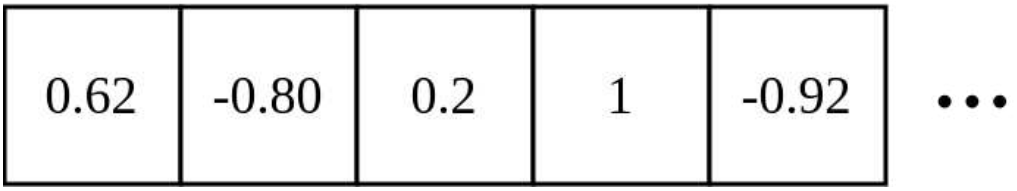


Figure 14: Array example.

The length of this array is always the same and it is 138 like the number of parameters to predict. An important thing is how it is possible to distinguish which parameter a value corresponds to. Actually the model doesn't need this kind of information since it is able to infer it during the learning process, since it can associate to a specific value a corresponding change in the image. Anyway, in order to keep track of the parameters and understand the prediction of the model, a way to distinguish them is needed. In order to do that is used a dictionary where the key is the parameter name and the value is the position of the parameters inside the array. An example of the dictionary structure can be seen in the image 15.

When the file is processed and the array is generated, the method returns a pair consisting of the parameter array and the associated image. It may occur that not all parameters are always present in the mhm file. In such instances, if a parameter is missing, the array is populated with the value 0 at the corresponding position.

```
parameters_to_index = {
    "cheek/l-cheek-bones-decr|incr": 0,
    "cheek/l-cheek-inner-decr|incr": 1,
    "cheek/l-cheek-trans-down|up": 2,
    "cheek/l-cheek-volume-decr|incr": 3,
    "cheek/r-cheek-bones-decr|incr": 4,
    "cheek/r-cheek-inner-decr|incr": 5,
    "cheek/r-cheek-trans-down|up": 6,
    "cheek/r-cheek-volume-decr|incr": 7,
    "chin/chin-bones-decr|incr": 8,
    "chin/chin-cleft-decr|incr": 9
    ...
    ...
    ...
}
```

Figure 15: An example of the dictionary used in order to keep track of the parameters.

The `__len__` method allows to define how to calculate the length of the dataset and, consequently, the number of elements it contains.

When the dataset is created using the MakeHuman plugin, a CSV file is generated, containing the parameters present in the generated files. This file is used during the test phase in order to know which parameters are involved for generating the graphs which show how far the predicted values are from the actual ones. Specifically, it is used to identify which parameters are involved and, consequently, which ones need to be displayed in the graph. During the creation of the dataset this file is processed and the parameters involved are saved in a list which will be used later during the testing phase.

In PyTorch, **DataLoaders** are used to handle datasets efficiently, allowing data to be loaded in parallel and thus reducing training execution time. They also permit dividing the data into batches. When a DataLoader is instantiated, it is possible to specify whether to use shuffling or not. This feature allows shuffling the data during different epochs to prevent the model from learning a specific data order. Thanks to this feature at every epoch it is possible to find a different order and elements inside a batch.

With the implementation of the dataset class, the model is ready as well as the pipeline, allowing for testing it. It's advisable to start with a few parameters to determine if the model is sufficiently complex to predict parameters value accurately.

4.3.2 Parameters selection and first results

In order to understand if the pipeline works and the model as well, it is needed to decide which parameters to involve in the first testing. Since the image resolution is very low, it's necessary to choose parameters that are easily recognizable, meaning that different values should result in clear

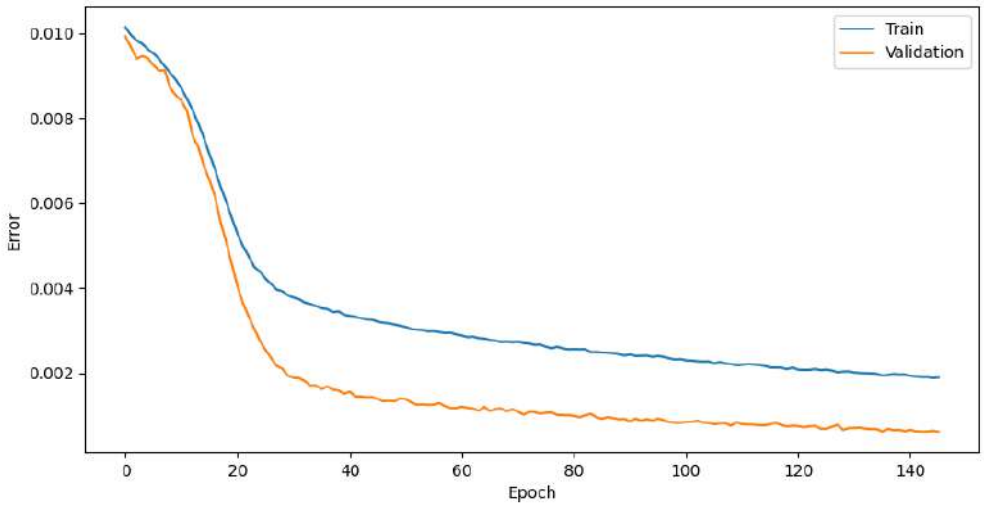


Figure 16: Loss graph.

and noticeable changes in the images. The parameter which best fit this behaviour are related to the ones that changes the position of the feature like nose, eyes or mouth, so it is chosen to check how the model work with only 3 parameters related to the nose which are the **horizontal position**, **vertical position** and **depth**.

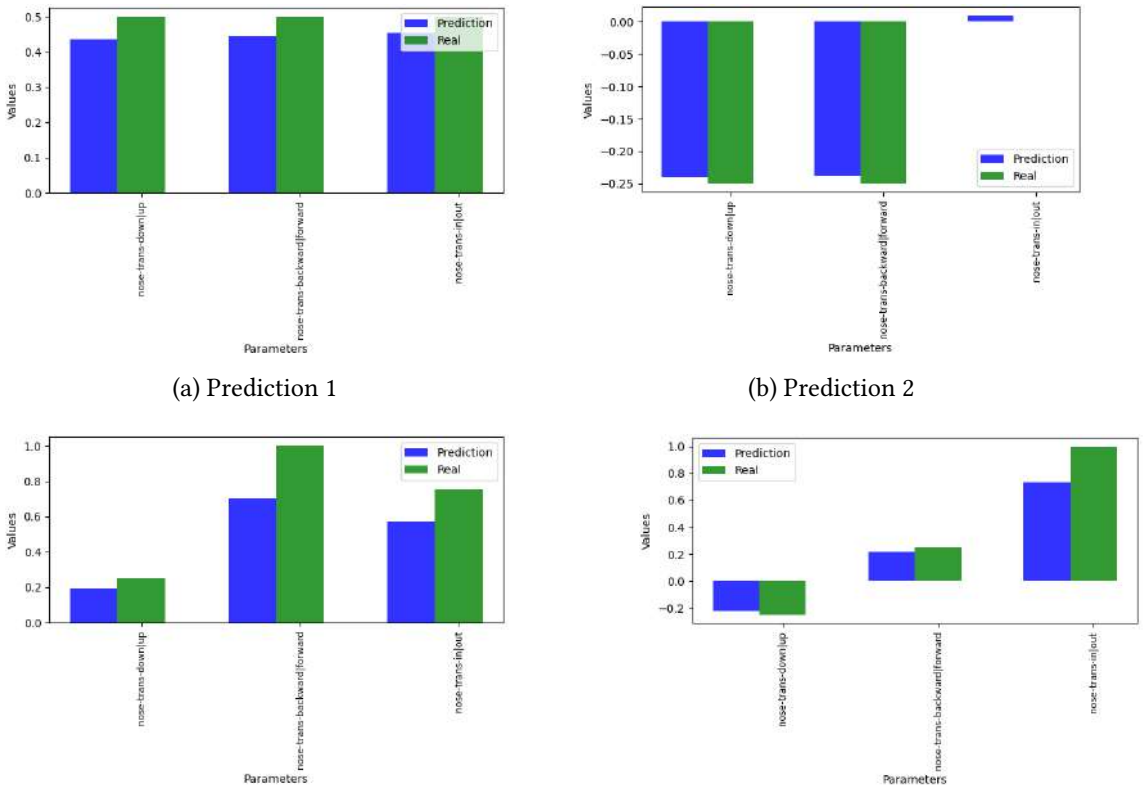
The dataset is generated according to the parameters chosen and the method used to generate it is the combinatorial one with a step of 0.25. The generation of the `mhm` files is really fast as it takes only 1 seconds to be performed with a total of 729 files generated which represents all the possible combinations according to the parameters and the step decided. The dataset, as previously mentioned, is divided into three folders: train, test, and validate. It also created the `info.csv` file where the information on which parameters are involved in the dataset are stored. The combinations are generated in sequence, meaning that the first file consists of all parameters set to the value -1, the second is composed by the first parameter with a value of -0.75 and the other two with a value of -1 and so on. In order to not preserve this order, before generating the files, they are shuffled. To generate all the images it took 332 seconds, approximately 6 minutes. At this stage it is possible to start the model train.

The model is trained with a learning rate of 0.00001, a train loader with a batch size of 64 while validation and test loaders with a batch size of 32. For the early stopping techniques the patience chosen is 10 which means that after 10 times that the loss is not improving, the training process is stopped. The execution time for the training is 104 seconds, approximately 2 minutes with a training and validations loss which can be seen in the image 16.

Both the training curves decrease a lot in the beginning while over the epochs they stabilize gradually. This means that the model is converging so it is learning. The curve related to the validation loss is slightly lower to the train one which means that the model generalizes well from the data and it is not going in overfitting. The training is performed in 146 epochs which is when

the early stopping mechanism is activated due to the validation loss.

After training, the test phase is performed with an overall MSE of 0.0005. In the images 17 it is possible to see how the predicted values are far from the actual ones. They are only some predictions among all the images in the test dataset.



(a) Prediction 1

(b) Prediction 2

(c) Prediction 3

(d) Prediction 4

Figure 17: Some model predictions.

As it is possible to see, the predictions of the model are really close to the actual ones, meaning that the pipeline works. In order to be sure of that, it is possible to check how the image generated from the `mhm` file predicted by the model is far from the actual one. This can be seen in the image 18 which reflect the results obtained previously since the images are practically identical.

Thanks to these results it is possible to say that the pipeline works and it is now possible to proceed with further experiments which will be explored in the next chapter.

(a) This image is created using the MakuHuman plugin, where three nose-related parameters are manually selected and their corresponding values are randomly generated.

(b) This is the image obtained from the `mhm` file generated from the values predicted by the model.

Figure 18: This is a comparison of the actual image and the one generated starting from the values predicted by the model. It is important to notice that the area involved is only the one related to the nose and the 3 parameters involved, since all the other parameters are the default one in both the images.

Chapter 5

Experiments and Results

Once verified that the pipeline works with nose parameters, the same thing is done with the mouth parameters and eye parameters. The results obtained are very similar to those of the model, which is able to predict the parameter values with a very small error.

There is something which can be specified related to the synthetic human faces generated, when a human model is created if a value for a certain parameter is not specified the default one is used. This means that if there are no parameters value specified, the human model produced is the default one. This is really relevant since if there are only some values involved in the creation of a human model, like previously, the other parameters value should be 0 and the model has to predict them accordingly. The problem is that different parameters related to various facial features are interdependent. This means that if the values for both the nose and the eyes are modified, the resulting human model can differ from the one where only the nose values are changed, even if the nose values remain the same. The interaction between the nose and eye parameters leads to a different generated image. This doesn't happen for all the face parameters but only with the ones interdependent, for instance a mouth parameter is not interdependent with an eye one.

In order to understand how the model works with more parameters and thus if it scales well, a dataset with all nose parameters is used. The parameters related to the nose are 21 and obviously it is not possible to generate all the combinations, then the random approach is used. Thanks to this approach, it is possible to generate a dataset as large as desired since it is possible to specify the number of elements wanted. The dataset generated is composed of 4000 pairs divided in three folders with the percentage specified in the previous chapter. In order to generate all the files the time required is about 2.5 seconds, while for generating all the images the time needed is more, around 45 minutes.

The model is trained with a learning rate of 0.0001 and a batch size of 128 for all three data loaders. The train is performed in 111 epochs with a validation loss of 0.001 and a train loss quite similar, it is possible to see them in 19. The graph highlights that the model is actually learning, even if the result is not as good as the one in the section 4.3.2.

The train is stopped thanks to the early stop mechanism according to the validation loss with a patience of 10. After training, the test phase is performed with the loss resulting in the same of the previous ones and an overall execution time of 11 minutes.

Some of the results obtained by the training model can be seen in the image 20.

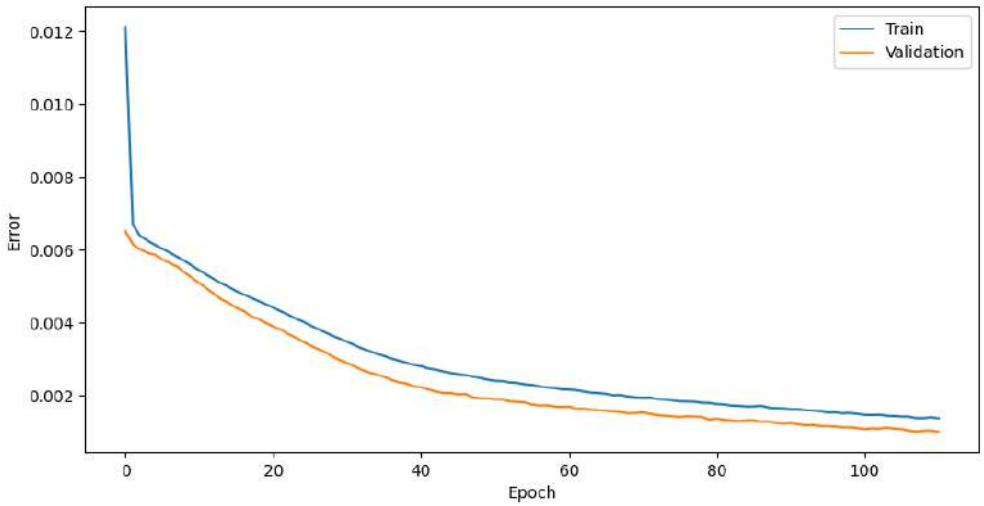
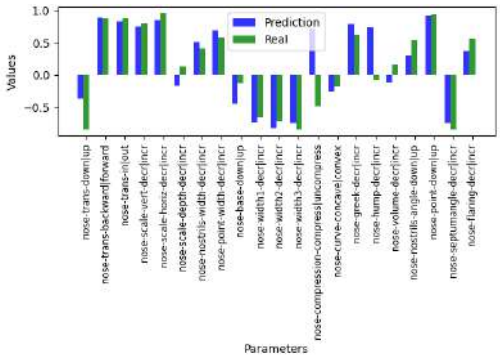


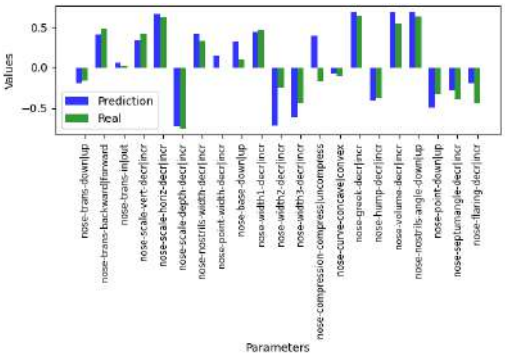
Figure 19: Loss graph.

The overall result highlights that the model is scaling well, anyway there are some parameters where the model predictions are a little too far from the actual values. These results are the ones which best highlights the overall predictions of the model, since quite all the graphs present the same behaviour. The parameters not predicted well are not always the same and this underlines two things. The image has very low resolution and lacks significant details. Consequently, changes in certain parameters may result in alterations that are not very noticeable in the image, making it difficult for the model to detect them. This leads to some parameters where the predictions are usually far from the actual one. Another issue is that parameters of the same feature are more interdependent than others, making it difficult for the model to understand the changes they cause. This will be highlighted more in the following section.

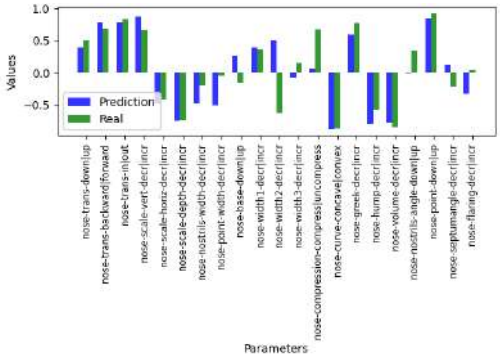
Additionally, it is important to assess how the model's predictions perform when dealing with images generated from those predictions and check how far they are from the starting images. This can be seen in the image 21 and it is possible to understand that the model is not correctly predicting the image even if the predictions highlighted by the graphs are not so bad.



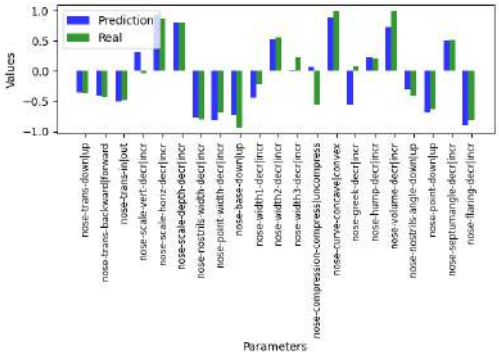
(a) Prediction 1.



(b) Prediction 2.



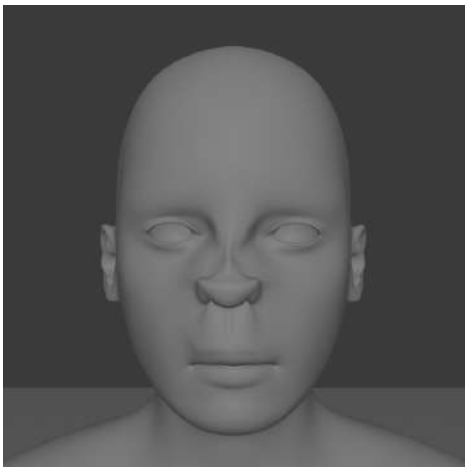
(c) Prediction 3.



(d) Prediction 4.

Figure 20: Some model predictions.

(a) This image is created using the MakuHuman plugin.



(b) This is the image obtained from the values predicted by the model.

Figure 21: This is a comparisons of the actual image and the generated one.

5.1 Gradual training

Since the previous results are not so good, thus the model is not able to correctly predict the image in input, a new training approach is tried in order to achieve better results.

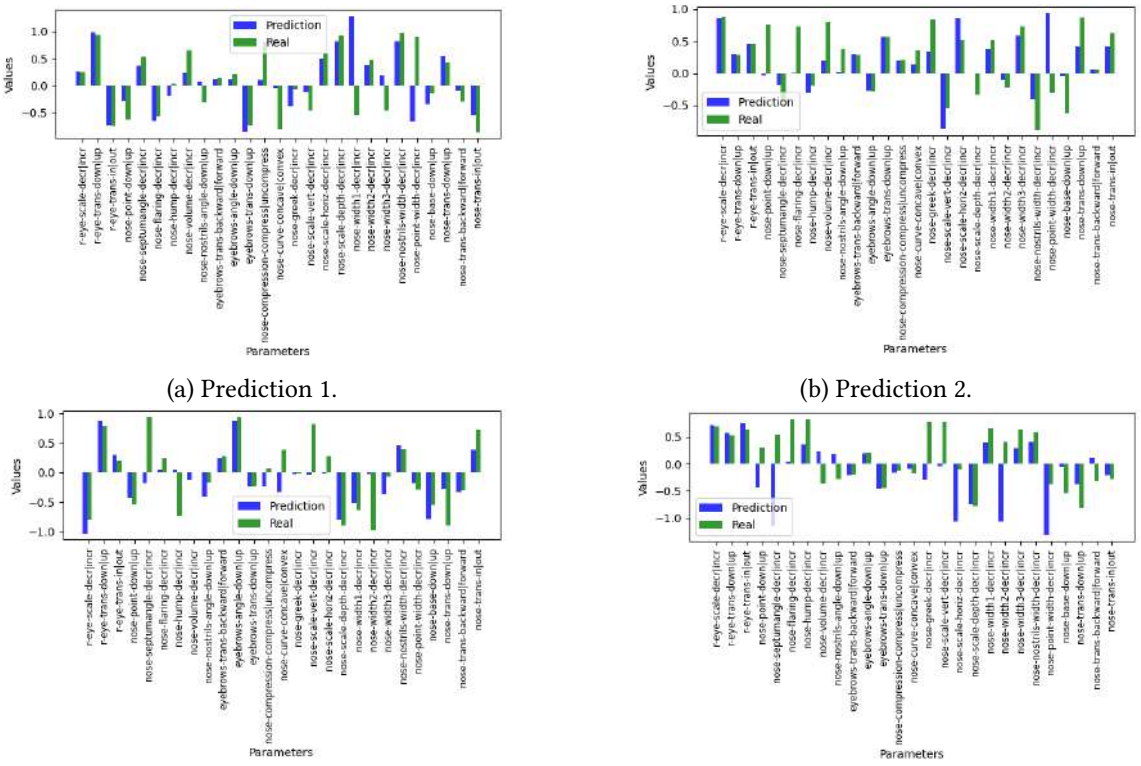
The dataset class created and described in the chapter 4 is really flexible and permits to create different kinds of dataset. The one created is composed by smaller dataset where each of them is related to parameters of a certain face feature. The features chosen are related to nose, eyebrows and eye, where the majority of them are related to the nose. The facial features and parameters are chosen based on their interdependence, as the nose can influence the eyes, which in turn can affect the eyebrows. An overall structure of the dataset can be seen below:

```
dataset/
├── nose_1/
│   ├── train/
│   ├── test/
│   ├── validate/
│   └── info.csv
├── eyebrows/
│   ├── train/.
│   ├── test/
│   ├── validate/
│   └── info.csv
├── nose_2/
├── eyes/
├── nose_3/
└── test_random/
    ├── test/
    └── info.csv
```

As it is possible to see, the data generated are related to a single face feature. In the first small dataset related to the first parameters of the nose there are only `mhm` files and related images where the overall human face is the default one except for the nose parameters chosen. This behaviour is the same for the other face features and parameters chosen. The last folder called *test_random*, instead, contains all `mhm` files and the related images of all the parameters chosen in the previous folders. This is done in order to understand if it is possible to train the network with data where only few parameters at time are modeled and then test the network with face models where all the parameters are included.

The small dataset usually includes 3 or 4 parameters at time and thus they are generated with the combinatorial approach with a step value of 0.25. The *test_random* one, instead, is created with the random approach since it includes all the parameters in the other folders and it is not possible to generate it with the combinatorial approach because of the high computational cost.

The dataset class created, as previously mentioned, is very flexible. To create the data loaders,



(a) Prediction 1.

(b) Prediction 2.

(c) Prediction 3.

(d) Prediction 4.

Figure 22: These are some model predictions obtained by the model trained with the gradual approach.

it scans all the files in the sub directories of train, test, and validate, and then generates the corresponding data loader for each of them. At the end only three data loaders are created related to the train, test and validate one.

The test executed on the files which are related to the small dataset produces a good result like the ones in the section 4.3.2. The results obtained with the dataset which include all the parameters involved are not so well as the previous ones. Some parameters are predicted well while others are not. This can be for two main reasons, the first is that, as previously said, the images resolution is too low and some changes in parameters cannot produce relevant changes in the images so they cannot be perceived by the network. The second one may be that some parameters can influence others. Since the model is trained on the smaller dataset where only parameters of the same feature are explored, it is possible that it is not able to catch the changes derived from interdependent parameters. This problem happens even for the parameters of the same feature and it is accentuated with interdependent parameters of different features.

It is possible to see some predictions of the model in the image 22, which highlights that this approach cannot be feasible since a lot of parameters predicted are really far from the actual one, some of them are predicted in the opposite side which means that even if the actual parameters

(a) This image is created using the MakuHuman plugin.

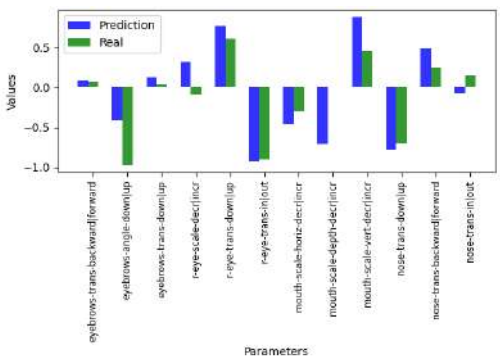
(b) This is the image obtained from the values predicted by the model.

Figure 23: This is a comparison of the actual image and the generated one.

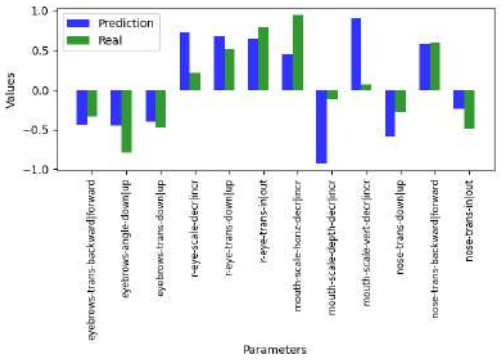
value is a positive number, the predicted one is negative. In order to be sure that the final result reflects the one highlighted by the graphs, in the image 23 it is possible to see how the image predicted by the model is far from the actual one.

In order to understand if this result is correlated to the kind of parameters chosen and the interdependence of them another test has been done with fewer and different parameters chosen according to dependent features. The chosen features are related to the eyebrows, right eye, nose, and mouth. A total of 12 parameters were selected, 3 for each feature. These parameters are the most significant, ensuring that the results obtained do not suffer from issues related to low image resolution and changes that are difficult for the model to perceive. The image 24 highlights that the challenge with this approach isn't due to image resolution, but rather the interdependence of the parameters. This complexity makes the approach seem unfeasible, particularly given the goal of training a model to predict all 138 facial feature parameters. This is confirmed by the image 25 generated from the model.

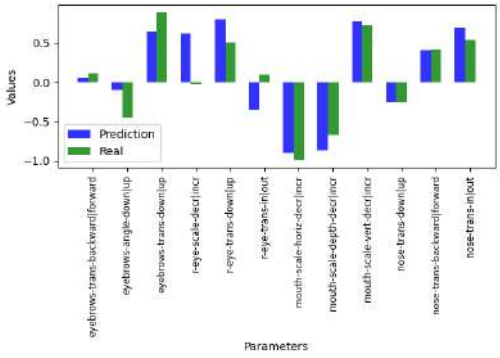
For the reasons above this approach is not further explored in favour of another one where it is tried to predict all the parameters involved facing the problems previously described.



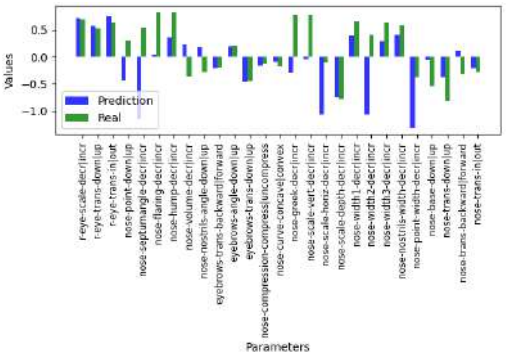
(a) Prediction 1.



(b) Prediction 2.

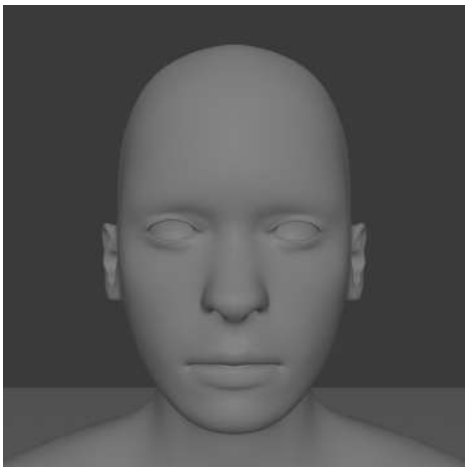


(c) Prediction 3.



(d) Prediction 4.

Figure 24: These are some model predictions obtained by the model trained with the gradual approach with less parameters than the previous one.



(a) This image is created using the MakuHuman plugin.

(b) This is the image obtained from the values predicted by the model.

Figure 25: This is a comparison of the actual image and the generated one.

5.2 All parameters prediction

Since the previous approach didn't work well, it is necessary to understand how the model works with all the parameters involved and how to face the problem presented in the previous section.

The first thing to do is to generate the dataset where all the parameters are involved. In order to do that it uses the random approach to generate the `mhm` files. The dataset is composed of 10000 pairs where the human models are created using all the parameters. This is divided into 6 pools where 4 of them are composed of 2000 pairs while the other two are composed of 1000 pairs. This division isn't based on specific criteria but is done to facilitate dataset generation and prevent crashes. Generating all 10,000 pairs at once would be time-consuming, so splitting the process into parts is more manageable. In order to generate a pair of 2000 `mhm` files it is needed 3.2 seconds which means that for generating all the 10000 pairs the time needed is 32 seconds, half a minute. The time required for the generation of this kind of file is not always time consuming, while the bottleneck of the generation phase is of course the image generation which time can vary according to the number of parameters involved and in this case, since all the parameters are involved, the time needed is the maximum. In order to generate 2000 images the time required is **3439 seconds**, around **57 minutes**. This means that for generating all the 10000 images the time required is **5 hours**.

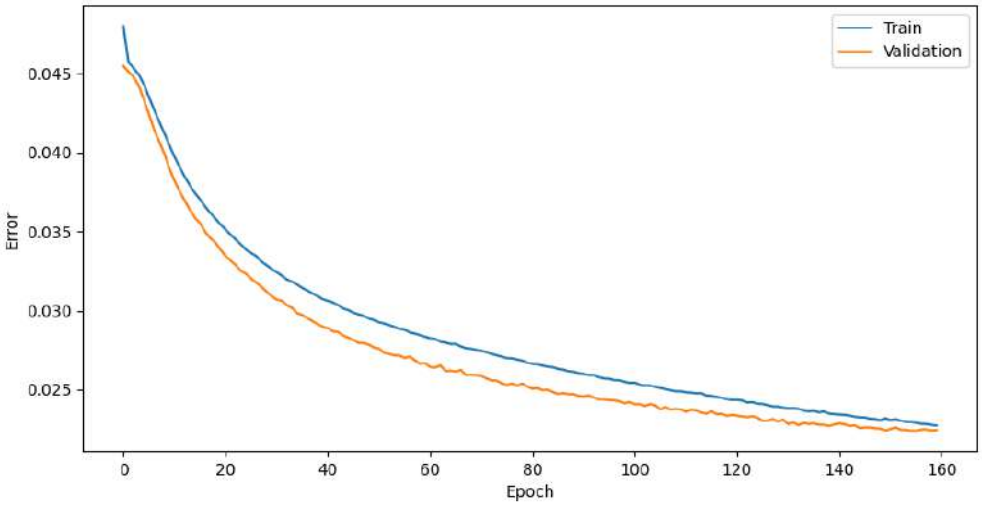
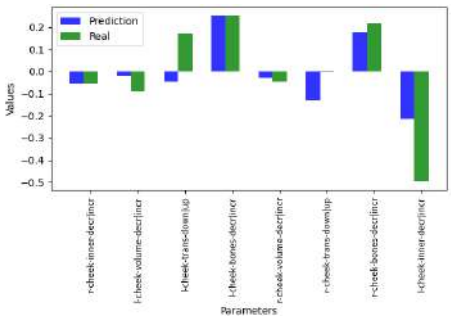


Figure 26: Loss graph.

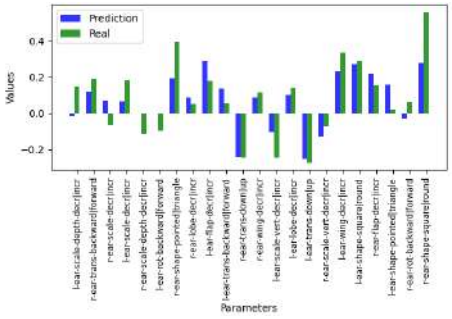
Once the generation is done, the model training is performed. The first attempt is done using a learning rate of 0.0001, a patience of 10 and a batch size of 128 for all the three data loaders. The training time is not as quick as the previous ones introduced in the previous sections, this is because of the size of the dataset which, as previously said, is composed of 10000 pairs which are visited in all the training epochs. Visiting this amount of pairs requires time even for a single epoch and this is amplified by the number of epochs needed and performed during the training.

The number of epochs performed are 160 with a training loss of 0.023 and a validation loss of 0.22 which can be seen in the image 26. The time needed for this training is **4020 seconds**, around **67 minutes** with a resulting loss of 0.022.

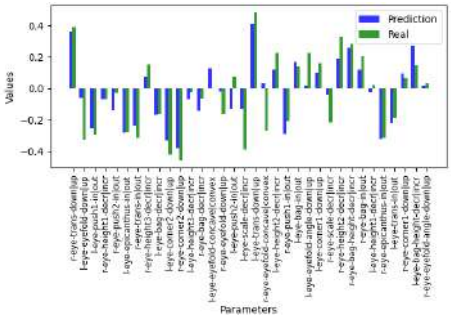
The result obtained is not quite good since a high loss means that a lot of parameters are predicted with a significant error which can be seen even in the image generated by the trained model. Since the parameters involved in this training and the corresponding results involve all the parameters, the graph generated which shows the prediction rather than the actual value is divided by class. Some of the results can be seen in the image 27.



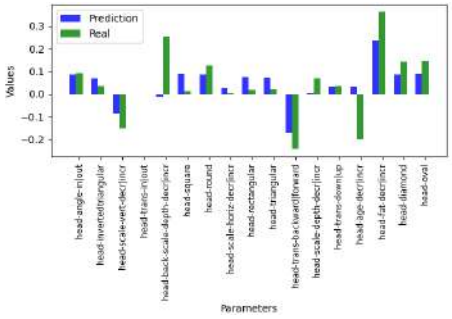
(a) Comparisons of cheek values.



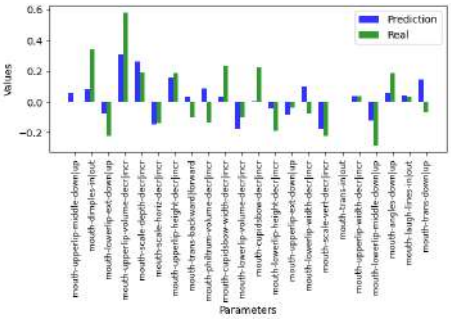
(b) Comparisons of ears values.



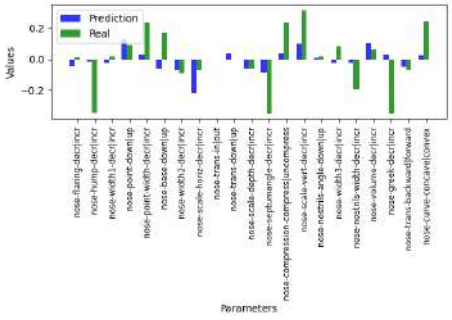
(c) Comparisons of eyes values.



(d) Comparisons of head values.



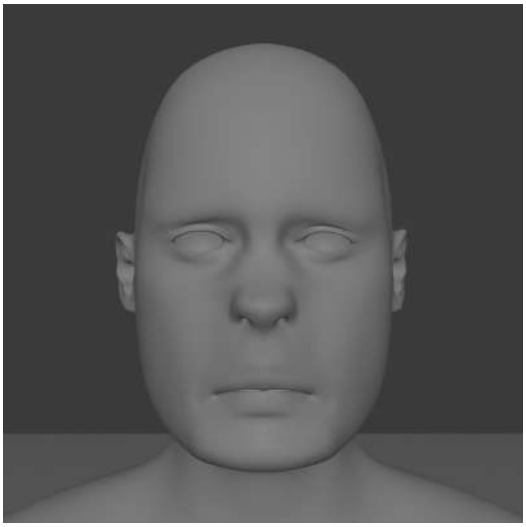
(e) Comparisons of mouth values.



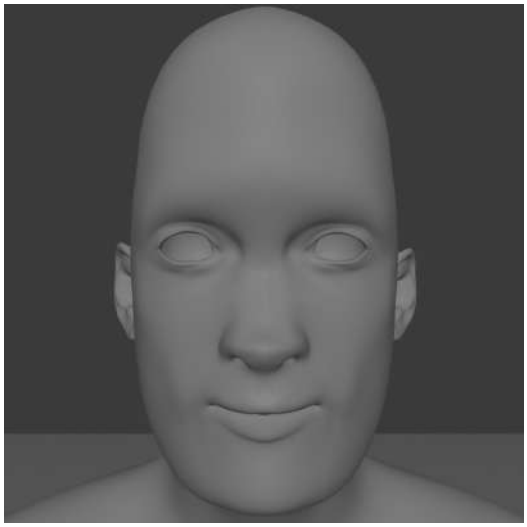
(f) Comparisons of nose values.

Figure 27: Some model prediction of the model trained with all parameters.

In order to better understand how the model behaves, it is important also to notice how the image generated from its prediction is distant from the actual one. This can be seen in the image 28. How it is possible to see the images confirms the result obtained, thus the model is not able to correctly predict the parameters of the input image.



(a) This image is created using the MakuHuman plugin.



(b) This is the image obtained from the values predicted by the model.

Figure 28: This is a comparisons of the actual image and the generated one.

Since the results obtained are not so good, another try is done using a lower learning rate which is 0.0001. This is done because maybe with the previous learning rate the model can oscillate around the minimum. With a lower learning rate the number of time needed for the training is obviously more than the previous one. The number of epochs needed are **623 epochs** with an execution time of **9318 seconds**, around **2 hours and half**. The validation and training loss can be seen in the image 29 and they are respectively **0.023** and **0.026**. The resulting loss, instead, is **0.023**. Some of the predictions obtained can be seen in the image 31 and they are consistent with the result obtained. Even in this case the image 30 confirms the results and thus they are not satisfactory.

In this case, the results obtained are still not satisfactory, and this could be due to various issues. The first two are already discussed in the previous sections, while the new one can be that the model structure is too trivial to capture so many features and thus the parameters value related to them. In order to understand if it is a real problem, it is chosen to identify the parameters which produce more change in the images and training the model only with the selected ones.

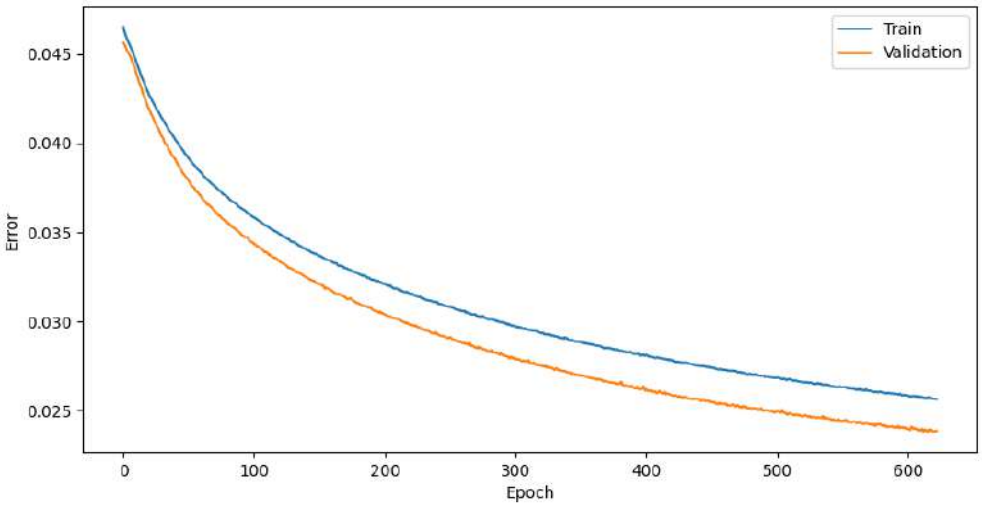
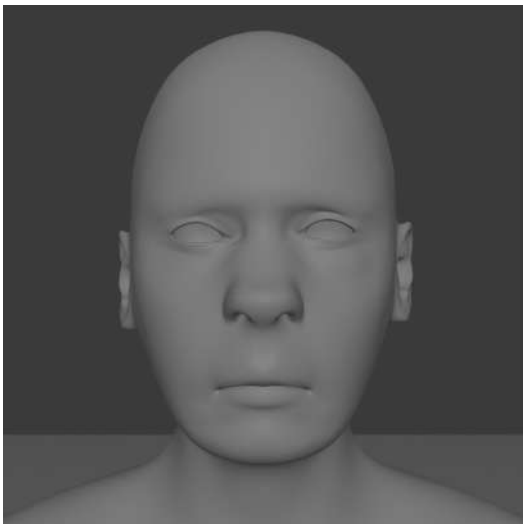
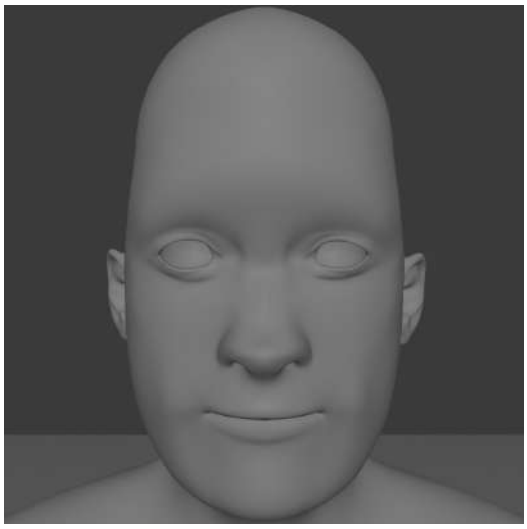


Figure 29: Loss graph.

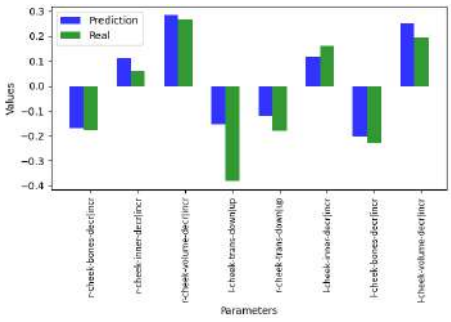


(a) This image is created using the MakuHuman plugin.

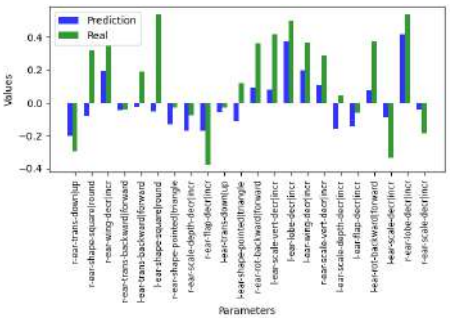


(b) This is the image obtained from the values predicted by the model.

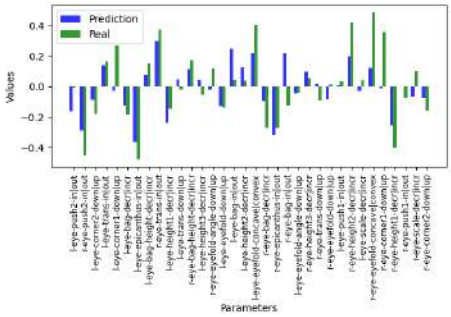
Figure 30: This is a comparisons of the actual image and the generated one.



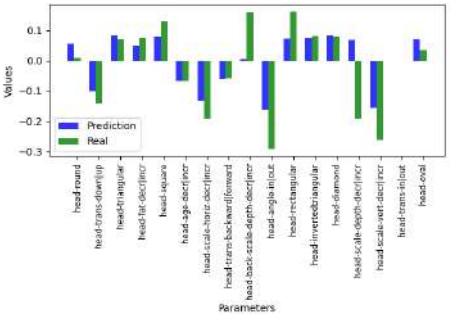
(a) Comparisons of cheek values.



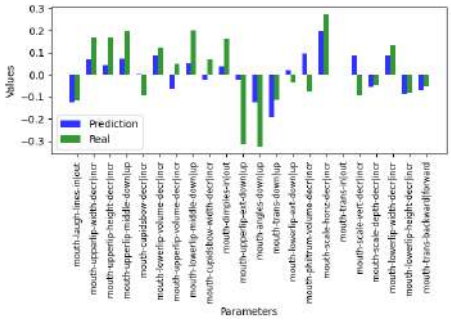
(b) Comparisons of ears values.



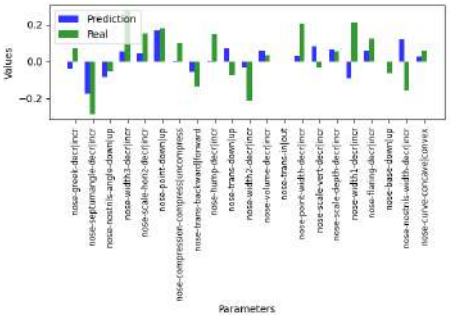
(c) Comparisons of eyes values.



(d) Comparisons of head values.



(e) Comparisons of mouth values.



(f) Comparisons of nose values.

Figure 31: Some predictions of the model trained with all parameters.

5.3 Parameters selection

In order to understand which parameters carry significant changes in the images it is possible to proceed with two approaches. The first task involves generating two images for each parameter, with the face model remaining the default except for the selected parameter. In one image, the parameter value is set to -1, and in the other image, it is set to 1. Then the difference between the images is performed and a rank is created. The second approach, instead, is to check how much an image generated with a parameter assuming one of the values at extreme of the range differs from the default one. In this approach, two differences are calculated, one between the image generated

with a parameter value of -1 and the default image, and another between the image generated with a parameter value of 1 and the default image. These two differences are then summed to obtain the final score for the parameter.

Both the approaches carry with them some common parts like the generation of the images which will be used for ranking the parameters. In order to do that, all the `mhm` files have to be generated and the MakeHuman plugin developed is not suitable for this task. Thus a new script is used exploiting the dictionary seen in section 4.3.1. The dictionary previously mentioned was used in order to assign at each parameter an index which keeps the information of its position within the array used during the model training. Anyway for this purpose it is needed that every parameter is assigned to the class it belongs to, so the new dictionary created needs to have as a key the class name and as value a list of all the parameters belonging to that class. Once this dictionary is created another script is used in order to generate all the `mhm` files for each parameter. This time it is not needed the MakeHuman context since all the parameters names are already available as well as the value they have to assume. Thus for all the parameters two files are generated, one where the value assumed by the parameter is -1 and one where the value is 1. All the files are organised according to class name leading to a precise folders structure. When all the `mhm` files are generated it is also possible to generate all the images related to them. They are organised according to the class and after that, all the `mhm` files are deleted since they are not needed anymore. With all the images generated it is finally possible to score each parameter and rank them.

In order to perform the ranking it is used the **Mean Squared Difference** to assign at each parameter a score. It can be expressed in the following way:

$$MSD = \frac{1}{N}\sum_{i=1}^{N}(\text{img}_1(i) - \text{img}_2(i))^2 \tag{8}$$

where:

- $\text{img}_1(i)$ and $\text{img}_2(i)$ are the i-th pixels of the images.
- N is the total number of the pixels of the images.

Even if in this case it is not needed since the images resolution is always 128×128, the sum is divided by the number of total pixels in order to let it not be dependent on the image resolution.

At this stage, two different approaches are used to determine which one produces the most significant results. Since there are many parameters being compared, and some do not lead to a significant change in the images produced, quartiles are used to select the more relevant parameters. The selected parameters are those within the range from the 25th percentile to the maximum value. Starting from this selection the two approaches are used in order to produce two different dictionaries ordered by the score of each parameter and thus according to the changes they produce in the images.

The results obtained can be seen in the images 32 and 33. The graphs generated are two since the comparison is done starting from one dictionary. If it is considered the dictionary related to the extremes of ranges the parameters are represented according to their position in the dictionary and then compared to their position in the other one created from the differences generated from

the default image. In this way it is possible to understand which parameters are more relevant than others and which approach is better or if they produce quite similar results.

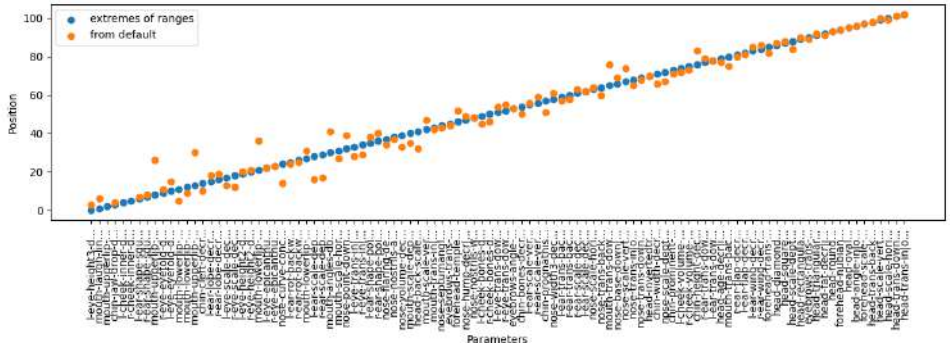


Figure 32: This graph highlights the position of a parameter according to its score where the blue point refers to the results obtained by the extreme of ranges approach and the orange point refers to the results obtained by the from default approach.

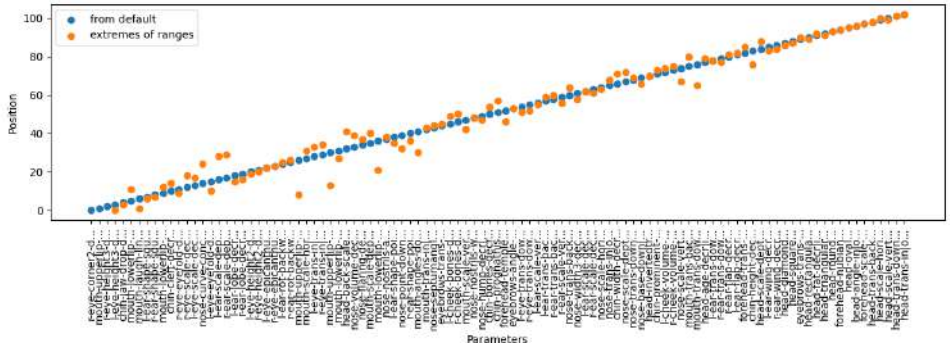


Figure 33: This graph highlights the position of a parameter according to its score where the blue point refers to the results obtained by the from default approach and the orange point refers to the results obtained by the extreme of ranges approach.

The two results indicate that depending on the chosen approach, certain parameters lead to more significant changes in one approach compared to the other. It is also possible that in an approach a parameter is present while in the second one it isn't. This behaviour is due to the fact that a parameter can be present in the 25th percentile according to the approaches. This can be seen in the images when the position of the parameter is highlighted in blue but there isn't its counterpart in orange.

In order to obtain a more clear classification another graph is produced where the parameters are not classified by their position gained by the score but directly by it. The results can be seen

in the image 34 where a point highlights the score obtained with the two different classification methods.

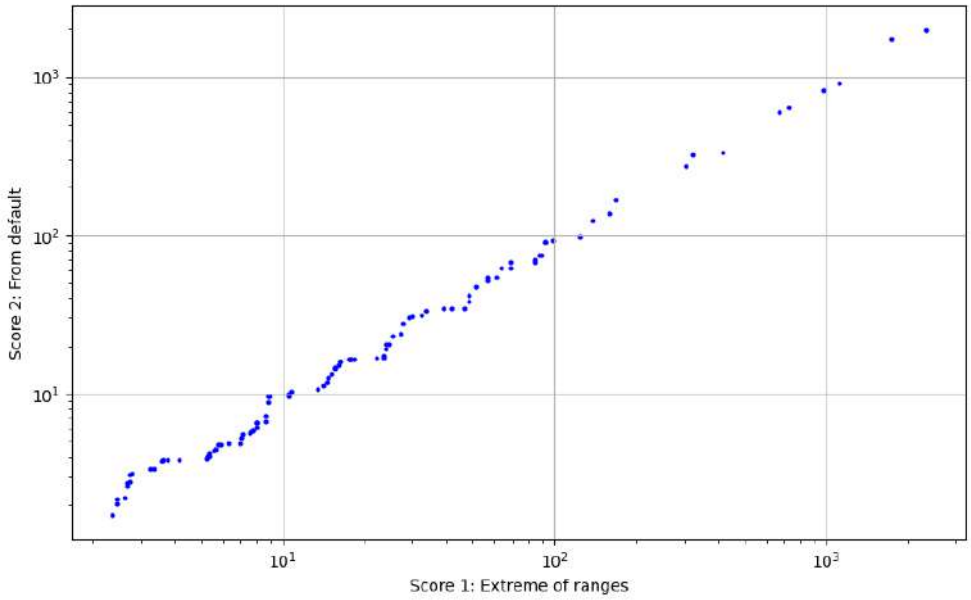


Figure 34: In the graph it is possible to understand the entity of the changes produced by a parameter according to the approach chosen. If a parameter has similar coordinates on both the x and y axes, it means the changes it produces are quite similar in both approaches.

Looking at images the results obtained highlights that according to the approach chosen it is possible that a certain parameter can produce a bigger change. Anyway there is no clear reason to discriminate one approach over another and thus the approach related to the extreme of ranges is chosen.

Once the approach has been chosen, it is time to create a dataset with the parameters and train the network. If it is used the same percentile of the one used for decide which approach to use, the parameters involved are 103. This number is not so far from the number of all parameters which is 138, so proceed in this way is not a good choice. In order to understand how the network performs with fewer parameters it is chosen to train two networks, one with the 25th-75th and the other one with the 50th-100th percentile. In this way for the first choice there are 68 parameters while for the second 69 parameters. In the images 35 and 36 it is possible to see which are the parameters involved in the train respectively.

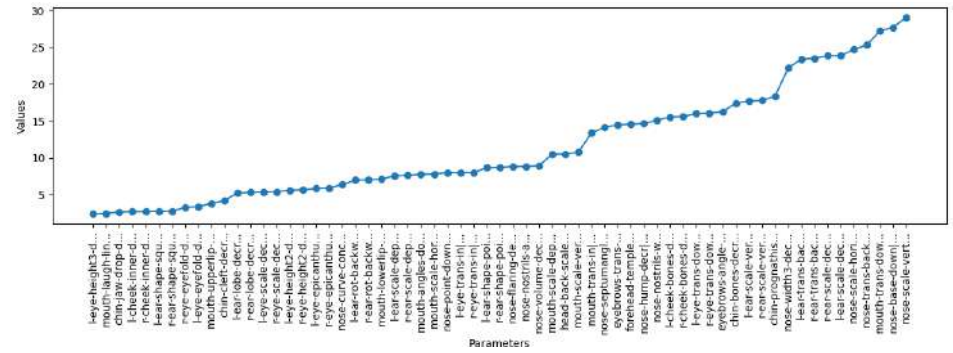


Figure 35: This graph highlights the parameters chosen according to the 25th-75th percentile.

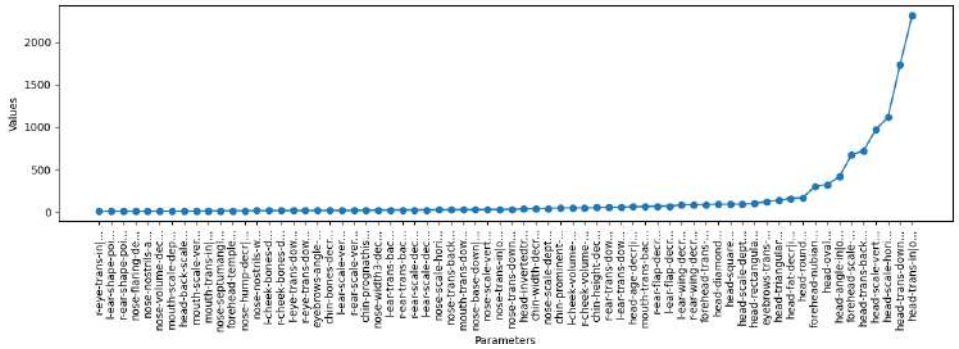


Figure 36: This graph highlights the parameters chosen according to the 50th-100th percentile.

5.4 Models results

There are two different selection of parameters that are used in order to understand if the model is able to predict correctly with fewer parameters involved. To do that it is trained the model with the first parameters selection and then with the other. There is no reason related to this choice.

5.4.1 First selection result

The parameters involved in the training are presented earlier and they are 68. The training is performed in 206 epochs which is when the early stopping mechanism is triggered. The learning rate used is 0.00001 while the batch size for all the three data loaders is 128. The loss of the learning process can be seen in the image 37 which at the end of the training is 0.008177 for the training one and 0.005471 for the validation one. The overall loss at the end of the training instead is 0.005486, quite similar to the validation one. The overall process is done in 3297 seconds which corresponds approximately to 55 minutes.

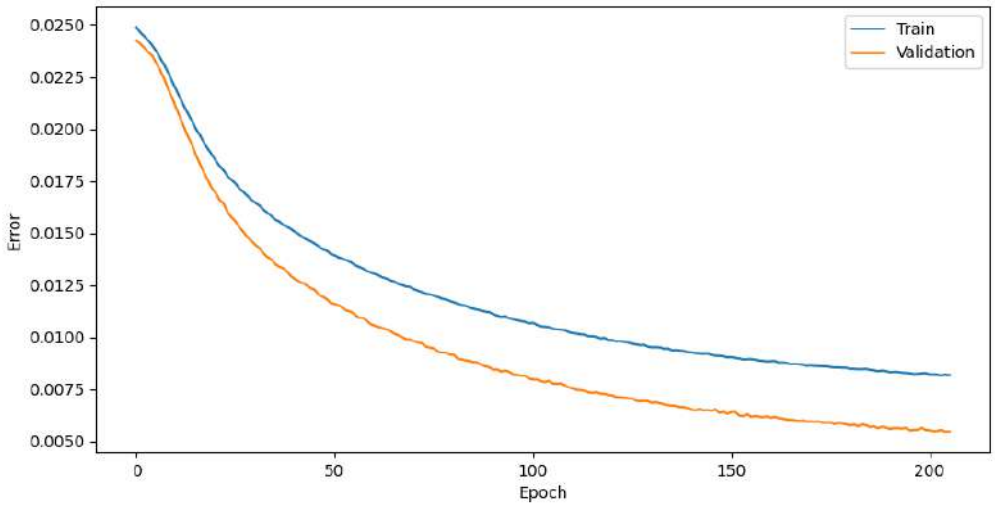


Figure 37: Loss graph.

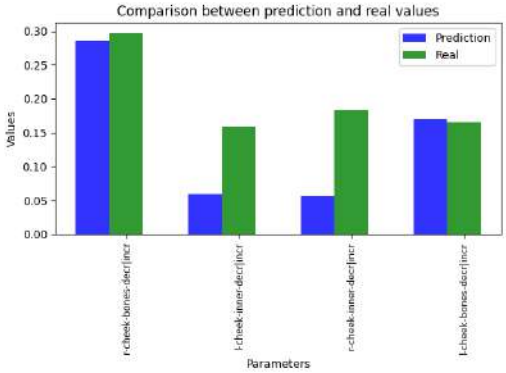
The graphs related to the prediction of the model against the real values can be seen in the image 38. Since there are a lot of parameters, they are divided by class, the graphs highlight that according to the parameters to predict, the model is able to predict a value which is near to the real one or not. Seeing the overall result highlighted in these graphs, it seems that even in this case the results obtained are not so good even if they are slightly better than the one obtained when the model had to predict all the parameters.

Anyway the results are not so bad if it is compared an image generated by the prediction of the model against the one given in input the it. It is possible to see it in the image 39.

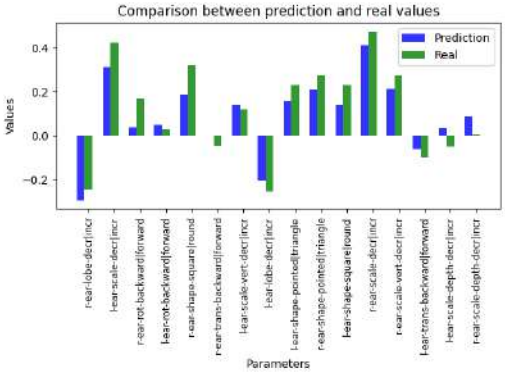
The two images generated are quite similar and it is really difficult to catch the differences between them so it is possible to say that the result achieved is quite good despite the graphs previously presented. There are some differences between the two images related to different parts like mouth, nose, head shape and ears, anyway they follow the result obtained during the training which is an average prediction mismatch of 0.005 between the real values and the predicted ones.

5.4.2 Second selection result

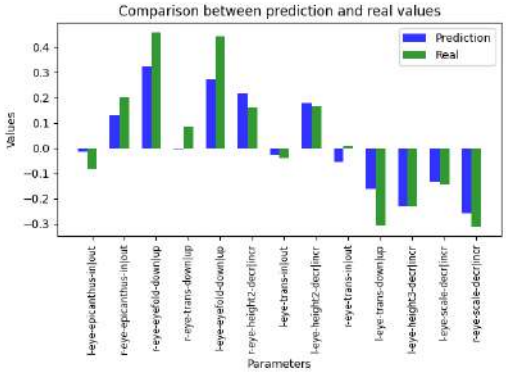
This time the parameters involved in the training are 69, quite similar to the previous one. The training is completed in 374 epochs with a train loss of 0.006882 and a validation loss of 0.005808. How the loss changed over the training can be seen in the image 40. It is completed in 68 minutes approximately with a result loss of 0.0058. Even in this case the training is done with a learning rate of 0.00001 and a batch size for all the loaders of 128.



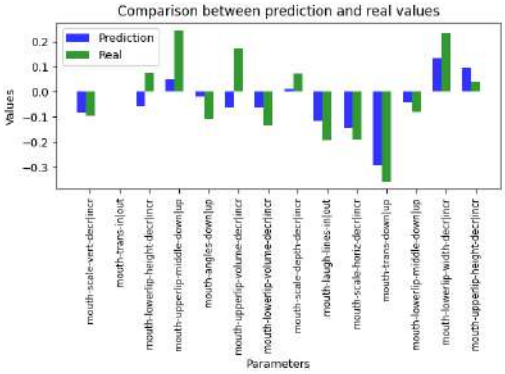
(a) Comparisons of cheek values.



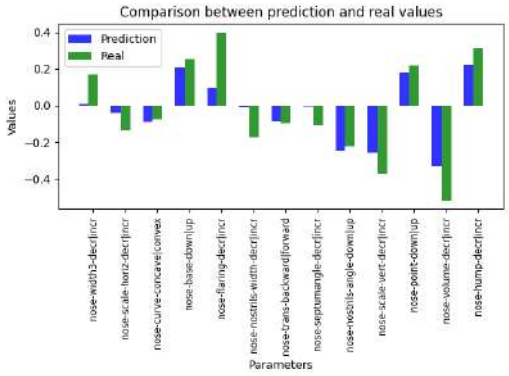
(b) Comparisons of ears values.



(c) Comparisons of eyes values.



(d) Comparisons of mouth values.



(e) Comparisons of nose values.

Figure 38: Comparisons of the real and predicted values.

(a) This image is created using the MakuHuman plugin.

(b) This is the image obtained from the values predicted by the model.

Figure 39: Comparison between the image generated using the MakeHuman plugin and the one generated by the model.

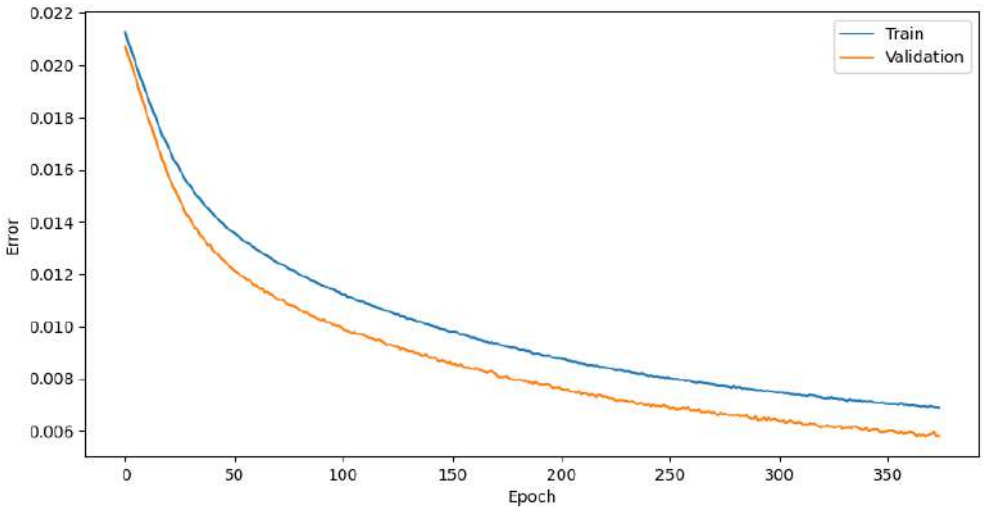
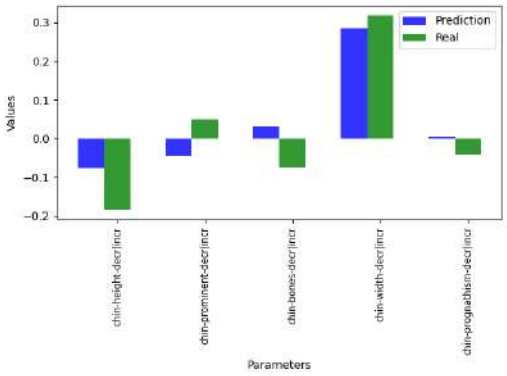
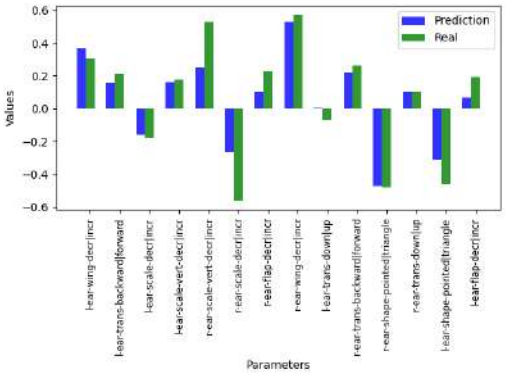


Figure 40: Loss graph.

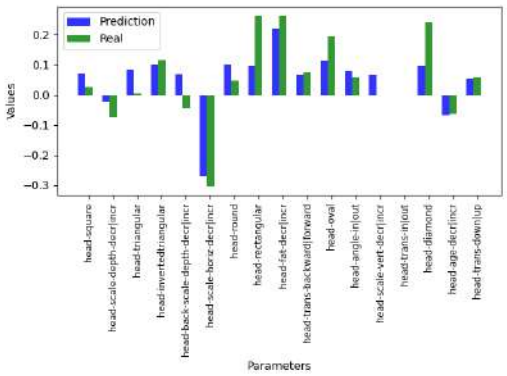
Some results of the model prediction can be seen in the image 41. Even in this case the most relevant one for each relevant class is taken. Exactly like the previous results, the results obtained highlighted by the graph are not so good and they are in line with the loss mentioned. Anyway



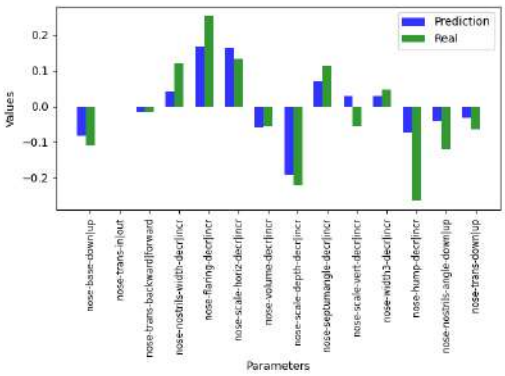
(a) Comparisons of chin values.



(b) Comparisons of ears values.



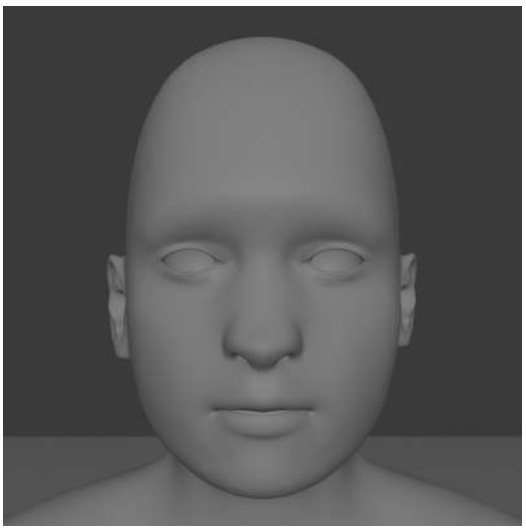
(c) Comparisons of head values.



(d) Comparisons of mouth values.

Figure 41: Comparisons of the real and predicted values.

the images generated by the model which produced these results are not so bad, it is possible to see them in the image 42. There are some differences related to the head shape and the nose, but they are not so noticeable to human sight.



(a) This image is created using the MakuHuman plugin.

(b) This is the image obtained from the values predicted by the model.

Figure 42: Comparison between the image generated using the MakeHuman plugin and the one generated by the model.

At this stage it is possible to say that with less parameters involved, the network is able to predict quite correctly the human face parameters, thus it is possible to achieve a better result using a more complex network as well as images with higher resolution. This will be discussed further in the chapter 6.

5.5 Real images prediction

The previous approaches lead to the creation of two different models according to the parameters selected for each of them. Now it is possible to test how this trained model works with real face images. To test these two models, I used a photo of myself that I modified to better match the expected input of the models. The starting photo, which will be modified, can be seen in the image 43a.

In this photo a reflection removal is applied as presented in the 43 Zhang_2018. After that it is tried to use Gimp AI. In order to do that, the photo which the original resolution was 1020x1280, is scaled to 768x1024 which is a compatible resolution with Gimp AI. It is then added an alpha transparency layer and it is removed the alpha channel related to the region of the eyebrows. At this stage it is possible to use the Gimp AI which is able with two iterations to produce the result in the image 43b. Anyway the image produced lost a lot of details, which can lead to not obtaining

a suitable result for creating a proper input for the network, so it is chosen to proceed in another way.

It is used another Gimp plugin called gmic-inpaint. Starting from the photo 43a the eyebrows are colored with red, resulting in the image 43c. Then a filter is applied though the plugin previously mentioned which leads to the result in the image 43d. At this stage it is removed the region related to the eyes coloring them with pink and then the image is transformed into a grayscale one like in the figure 44.

The photo is almost ready to be used as input for the model. However, its resolution needs to be changed to 128x128 pixels, as expected by the model. An important detail is the distance between the top of the image and the head; the model expects some space in this area. If it is not present the network produces a very bad result related to the shape of the head. Additionally, the current image is rectangular, while the expected one is squared, so further modifications are necessary when adjusting the resolution. To make the image resemble the expected input, the shoulder area has been extended by painting the same gray color of the chest until the edges of the image, resulting in the final image shown in 45.

This image is used as input for both the two model trained and the results achieved can be seen in the image 46. Looking at them, the model which generates a human face model that is more similar to the input image is the first one. This is obviously because in that selection there are more parameters which permits to control the appearance of certain features like eyes, nose, mouth and ears which are the ones that most influence the similarity to the input image. Instead the second model produces a more accurate human face model for the features related to the head and its shape since the first one has very few parameters related to it.

The results obtained tell us that this approach can be used in order to predict the parameters related to the face features of the MakeHuman application and so to create a 3D model starting from a 2D image. The main problem to face up is the one related to the complexity of the network since the one presented is not able to handle all the 138 parameters. It is also important to understand how the model works, training it with higher resolution images. Anyway this approach can be further explored and improved in order to properly predict a face model from a single image.

(a) This is the starting photo to modify in order to have an input similar to the one expected from the network.

(b) This is the result obtained processing the images using Gimp AI.

(c) This is the image where an eyebrows mask is applied which will be passed in to the Gimp plugin gmic-inpaint in order to apply the filter.

(d) This is the result obtained applying the filter through the gmic-inpaint plugin.

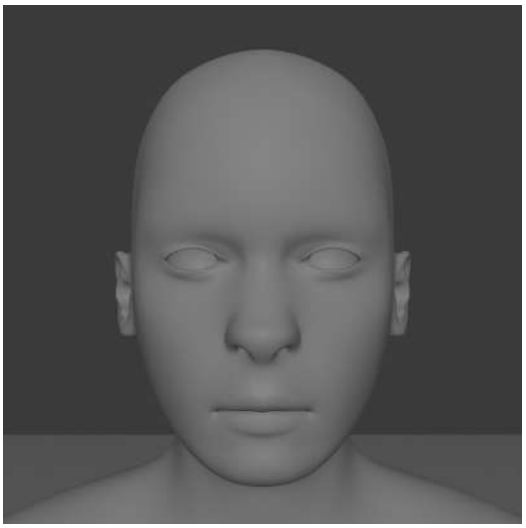
Figure 43: A series of images modified in order to obtain a valid input for the neural networks.

(a) Image where the eyes are darkened using a pink color.

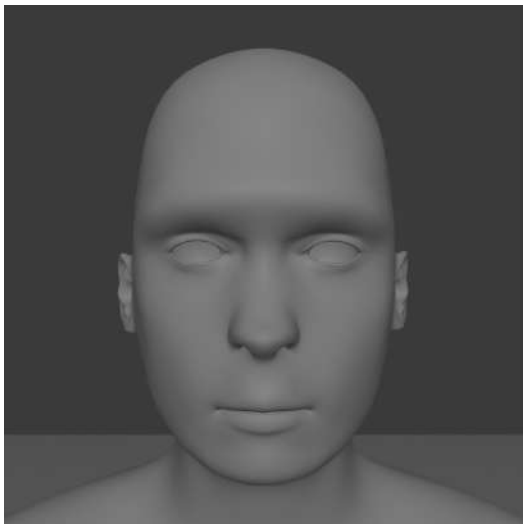
(b) The gray scale image obtain from the one to the left.

Figure 44: The result obtained after image processing.

Figure 45: The resulting image, scaled properly, in order to use it as input for the model.



(a) This is the output predicted by the model trained with the 25th-75th percentile.



(b) This is the output predicted by the model trained with the 50th-100th percentile.

Figure 46: The two results obtained using the two different models created.

Chapter 6

Future improvements

Blender is a really powerful tool which can help in order to produce a photo realistic models. During the project the model produced as well as the images derived from these are not photo realistic. They are created starting from the `mhm` files thanks to the MPFB2 plugin for Blender and this produces a model without texture and eyes. The model produced has a gray skin which can obviously be a problem when the model has to deal with real images. So one of the improvements which it is possible to do is to use all the Blender functionalities to produce a better human model to take a photo of. This can lead to less effort in order to prepare the real images for the network and check how the model behaves.

One of the problems encountered during the project's development is related to image resolution. The networks in this study are all trained with 128x128 images, which introduces issues concerning the information each pixel conveys. It is clear that if higher resolution images were used for training, each pixel would carry more information, thereby resolving the issue of the network's inability to detect changes caused by varying certain parameters. Anyway it can introduce another problem which is the one related to the high computational cost as well as the execution time. Therefore, it is essential to find a compromise between resolution and computational cost. The goal is to identify a resolution that provides enough information for the network to capture even the smallest changes induced by varying parameters while maintaining a reasonable processing time. This balance ensures that the network can perform effectively without excessive computational demands.

Another problem faced is the one related to the complexity of the model. The face parameters in MakeHuman are 138 and there are a lot of them which are interdependent and since the network is composed of only three convolutional layers it is possible that the network is not able to detect all the relation between the data in input. It is necessary to create a more complex network and verify how it behaves and the results achieved. In order to create a more resilient network it is also possible to use the technique called Data Augmentation which consists of increasing the quantity and the diversity of the data making changes to the images already present in the dataset. The changes can be rotation, translation, zoom and so on. Of course the original images are kept in the dataset as well.

For producing a more versatile network it is also possible to introduce facial poses. In order to do that it is possible to train the network with facial images where the human model is in a certain

pose. This can be done using Blender with the plugin presented in this work, MPFB2. When a model is imported in blender from a `mhm` file, it is usually in a T-pose with the face in a neutral pose. Anyway the plugin allows importing a pose using OpenPoses. It is needed that the model inside Blender has a rig in order to map the pose on it. Then it is possible to take the photo of the face model and add them to the dataset used to train the network.

It is also important to perform an image preprocessing when it is needed to try how the network behaves with real images. It is important to remove some component to the images like hair, eyebrows, beard or whatever element in the image that can interfere with the prediction of the model. So it is needed to create an automatic process that can produce the same result obtained in the chapter 5.5.

Appendix A

MakeHuman parameters

MakeHuman is a tool that allows to modify the geometry of a human-shaped quadrilateral mesh in terms of features. model parameters allows to are grouped Here the parameters involved in shaping the face features are reported grouped by their modifiers name as exposed in the GUI, with their corresponding key that identifies them in the code.

GUI Category: Head shape		
GUI Label	Parameter Key	Range
Age	head/head-age-decr|incr	[-1.0, 1.0]
Head fat	head/head-fat-decr|incr	[-1.0, 1.0]
Angle	head/head-angle-in|out	[-1.0, 1.0]
Oval	head/head-oval	[0.0, 1.0]
Round	head/head-round	[0.0, 1.0]
Rectangular	head/head-rectangular	[0.0, 1.0]
Square	head/head-square	[0.0, 1.0]
Triangular	head/head-triangular	[0.0, 1.0]
Invertedtriangular	head/head-invertedtriangular	[0.0, 1.0]
Diamond	head/head-diamond	[0.0, 1.0]
Scale depth of parietal side	head/head-back-scale-depth-decr|incr	[-1.0, 1.0]

Table 1: MakeHuman face parameters for Head shape GUI category.

GUI Category: Head size		
GUI Label	Parameter Key	Range
Scale depth	head/head-scale-depth-decr|incr	[-1.0, 1.0]
Scale horizontally	head/head-scale-horiz-decr|incr	[-1.0, 1.0]
Scale vertically	head/head-scale-vert-decr|incr	[-1.0, 1.0]
Move horizontally	head/head-trans-in|out	[-1.0, 1.0]
Move vertically	head/head-trans-down|up	[-1.0, 1.0]
Move depth	head/head-trans-backward|forward	[-1.0, 1.0]

Table 2: MakeHuman face parameters for Head size GUI category.

GUI Category: Forehead		
GUI Label	Parameter Key	Range
Forehead bulge	forehead/forehead-trans-backward|forward	[-1.0, 1.0]
Scale vertically	forehead/forehead-scale-vert-decr|incr	[-1.0, 1.0]
Cranic shape	forehead/forehead-nubian-decr|incr	[-1.0, 1.0]
Temple bulge	forehead/forehead-temple-decr|incr	[-1.0, 1.0]

Table 3: MakeHuman face parameters for Forehead GUI category.

GUI Category: Eyebrows		
GUI Label	Parameter Key	Range
Eyebrows bulge	eyebrows/eyebrows-trans-backward|forward	[-1.0, 1.0]
Eyebrows angle	eyebrows/eyebrows-angle-down|up	[-1.0, 1.0]
Move vert	eyebrows/eyebrows-trans-down|up	[-1.0, 1.0]

Table 4: MakeHuman face parameters for Eyebrows GUI category.

GUI Category: Neck		
GUI Label	Parameter Key	Range
Double neck	neck/neck-double-decr|incr	[-1.0, 1.0]
Scale depth	neck/neck-scale-depth-decr|incr	[-1.0, 1.0]
Scale horizontally	neck/neck-scale-horiz-decr|incr	[-1.0, 1.0]
Scale vertically	neck/neck-scale-vert-decr|incr	[-1.0, 1.0]
Move horizontally.	neck/neck-trans-in|out	[-1.0, 1.0]
Move vertically	neck/neck-trans-down|up	[-1.0, 1.0]
Move depth	neck/neck-trans-backward|forward	[-1.0, 1.0]
Scale depth of nape	neck/neck-back-scale-depth-decr|incr	[-1.0, 1.0]

Table 5: MakeHuman face parameters for Neck GUI category.

GUI Category: Right eye		
GUI Label	Parameter Key	Range
Eye bag volume	eyes/r-eye-bag-decr|incr	[-1.0, 1.0]
Eye bag distorsion	eyes/r-eye-bag-in|out	[-1.0, 1.0]
Eye bag height	eyes/r-eye-bag-height-decr|incr	[-1.0, 1.0]
Eyefold angle	eyes/r-eye-eyefold-angle-down|up	[-1.0, 1.0]
Eye epicanthus	eyes/r-eye-epicanthus-in|out	[-1.0, 1.0]
Eyefold volume	eyes/r-eye-eyefold-concave|convex	[-1.0, 1.0]
Eyefold position	eyes/r-eye-eyefold-down|up	[-1.0, 1.0]
Scale height 1	eyes/r-eye-height1-decr|incr	[-1.0, 1.0]
Scale height 2	eyes/r-eye-height2-decr|incr	[-1.0, 1.0]
Scale height 3	eyes/r-eye-height3-decr|incr	[-1.0, 1.0]
Move outer corner horiz.	eyes/r-eye-push1-in|out	[-1.0, 1.0]
Move inner corner horiz.	eyes/r-eye-push2-in|out	[-1.0, 1.0]
Move eye horizontally	eyes/r-eye-trans-in|out	[-1.0, 1.0]
Move eye vertically	eyes/r-eye-trans-down|up	[-1.0, 1.0]
Scale eye	eyes/r-eye-scale-decr|incr	[-1.0, 1.0]
Move outer corner vert.	eyes/r-eye-corner1-down|up	[-1.0, 1.0]
Move inner corner vert.	eyes/r-eye-corner2-down|up	[-1.0, 1.0]

Table 6: MakeHuman face parameters for Right eye GUI category.

GUI Category: Left eye		
GUI Label	Parameter Key	Range
Eye bag volume	eyes/l-eye-bag-decr|incr	[-1.0, 1.0]
Eye bag distorsion	eyes/l-eye-bag-in|out	[-1.0, 1.0]
Eye bag height	eyes/l-eye-bag-height-decr|incr	[-1.0, 1.0]
Eyefold angle	eyes/l-eye-eyefold-angle-down|up	[-1.0, 1.0]
Eye epicanthus	eyes/l-eye-epicanthus-in|out	[-1.0, 1.0]
Eyefold volume	eyes/l-eye-eyefold-concave|convex	[-1.0, 1.0]
Eyefold position	eyes/l-eye-eyefold-down|up	[-1.0, 1.0]
Scale height 1	eyes/l-eye-height1-decr|incr	[-1.0, 1.0]
Scale height 2	eyes/l-eye-height2-decr|incr	[-1.0, 1.0]
Scale height 3	eyes/l-eye-height3-decr|incr	[-1.0, 1.0]
Move outer corner horiz.	eyes/l-eye-push1-in|out	[-1.0, 1.0]
Move inner corner horiz.	eyes/l-eye-push2-in|out	[-1.0, 1.0]
Move eye horizontally	eyes/l-eye-trans-in|out	[-1.0, 1.0]
Move eye vertically	eyes/l-eye-trans-down|up	[-1.0, 1.0]
Scale eye	eyes/l-eye-scale-decr|incr	[-1.0, 1.0]
Move outer corner vert.	eyes/l-eye-corner1-down|up	[-1.0, 1.0]
Move inner corner vert.	eyes/l-eye-corner2-down|up	[-1.0, 1.0]

Table 7: MakeHuman face parameters for Left eye GUI category.

GUI Category: Nose size		
GUI Label	Parameter Key	Range
Move vertically	nose/nose-trans-down|up	[-1.0, 1.0]
Move depth	nose/nose-trans-backward|forward	[-1.0, 1.0]
Move horizontally	nose/nose-trans-in|out	[-1.0, 1.0]
Scale vertically	nose/nose-scale-vert-decr|incr	[-1.0, 1.0]
Scale horizontally	nose/nose-scale-horiz-decr|incr	[-1.0, 1.0]
Scale depth	nose/nose-scale-depth-decr|incr	[-1.0, 1.0]

Table 8: MakeHuman face parameters for Nose size GUI category.

GUI Category: Nose size details		
GUI Label	Parameter Key	Range
Scale nostrils width	nose/nose-nostrils-width-decr|incr	[-1.0, 1.0]
Scale tip width	nose/nose-point-width-decr|incr	[-1.0, 1.0]
Move base vert.	nose/nose-base-down|up	[-1.0, 1.0]
Scale width 1	nose/nose-width1-decr|incr	[-1.0, 1.0]
Scale width 2	nose/nose-width2-decr|incr	[-1.0, 1.0]
Scale width 3	nose/nose-width3-decr|incr	[-1.0, 1.0]

Table 9: MakeHuman face parameters for Nose size details GUI category.

GUI Category: Nose features		
GUI Label	Parameter Key	Range
Compression	nose/nose-compression-compress|uncompress	[-1.0, 1.0]
Curve	nose/nose-curve-concave|convex	[-1.0, 1.0]
Greek	nose/nose-greek-decr|incr	[-1.0, 1.0]
Hump	nose/nose-hump-decr|incr	[-1.0, 1.0]
Volume	nose/nose-volume-decr|incr	[-1.0, 1.0]
Nostrils Angle	nose/nose-nostrils-angle-down|up	[-1.0, 1.0]
Move tip vertically	nose/nose-point-down|up	[-1.0, 1.0]
Septum Angle	nose/nose-septumangle-decr|incr	[-1.0, 1.0]
Scale nostrils flaring	nose/nose-flaring-decr|incr	[-1.0, 1.0]

Table 10: MakeHuman face parameters for Nose features GUI category.

GUI Category: Mouth size		
GUI Label	Parameter Key	Range
Scale horizontally	mouth/mouth-scale-horiz-decr|incr	[-1.0, 1.0]
Scale vertically	mouth/mouth-scale-vert-decr|incr	[-1.0, 1.0]
Scale depth	mouth/mouth-scale-depth-decr|incr	[-1.0, 1.0]
Move horizontally	mouth/mouth-trans-in|out	[-1.0, 1.0]
Move vertically	mouth/mouth-trans-down|up	[-1.0, 1.0]
Move depth	mouth/mouth-trans-backward|forward	[-1.0, 1.0]

Table 11: MakeHuman face parameters for Mouth size GUI category.

GUI Category: Mouth size details		
GUI Label	Parameter Key	Range
Scale lowerlip height	mouth/mouth-lowerlip-height-decr|incr	[-1.0, 1.0]
Scale lowerlip width	mouth/mouth-lowerlip-width-decr|incr	[-1.0, 1.0]
Scale upperlip height	mouth/mouth-upperlip-height-decr|incr	[-1.0, 1.0]
Scale upperlip width	mouth/mouth-upperlip-width-decr|incr	[-1.0, 1.0]
Cupid's bow width	mouth/mouth-cupidsbow-width-decr|incr	[-1.0, 1.0]

Table 12: MakeHuman face parameters for Mouth size details GUI category.

GUI Category: Mouth features		
GUI Label	Parameter Key	Range
Dimples	mouth/mouth-dimples-in|out	[-1.0, 1.0]
Laugh-lines	mouth/mouth-laugh-lines-in|out	[-1.0, 1.0]
Lowerlip curved shape	mouth/mouth-lowerlip-ext-down|up	[-1.0, 1.0]
Move corners vert.	mouth/mouth-angles-down|up	[-1.0, 1.0]
Scale middle lowerlip	mouth/mouth-lowerlip-middle-down|up	[-1.0, 1.0]
Scale lowerlip volume	mouth/mouth-lowerlip-volume-decr|incr	[-1.0, 1.0]
Scale philtrum volume	mouth/mouth-philtrum-volume-decr|incr	[-1.0, 1.0]
Scale upperlip volume	mouth/mouth-upperlip-volume-decr|incr	[-1.0, 1.0]
Upperlip curved shape	mouth/mouth-upperlip-ext-down|up	[-1.0, 1.0]
Scale middle upperlip	mouth/mouth-upperlip-middle-down|up	[-1.0, 1.0]
Cupid's bow shape	mouth/mouth-cupidsbow-decr|incr	[-1.0, 1.0]

Table 13: MakeHuman face parameters for Mouth features GUI category.

GUI Category: Right ear		
GUI Label	Parameter Key	Range
Move depth	ears/r-ear-trans-backward|forward	[-1.0, 1.0]
Scale ear	ears/r-ear-scale-decr|incr	[-1.0, 1.0]
Move vertically	ears/r-ear-trans-down|up	[-1.0, 1.0]
Scale height	ears/r-ear-scale-vert-decr|incr	[-1.0, 1.0]
Scale lobe	ears/r-ear-lobe-decr|incr	[-1.0, 1.0]
Ear shape (pointed-triangle)	ears/r-ear-shape-pointed|triangle	[-1.0, 1.0]
Ear rotation	ears/r-ear-rot-backward|forward	[-1.0, 1.0]
Ear shape (squared-round)	ears/r-ear-shape-square|round	[-1.0, 1.0]
Scale width	ears/r-ear-scale-depth-decr|incr	[-1.0, 1.0]
Ear wing-shaped	ears/r-ear-wing-decr|incr	[-1.0, 1.0]
Ear flapped	ears/r-ear-flap-decr|incr	[-1.0, 1.0]

Table 14: MakeHuman face parameters for Right ear GUI category.

GUI Category: Left ear		
GUI Label	Parameter Key	Range
Move depth	ears/l-ear-trans-backward|forward	[-1.0, 1.0]
Scale ear	ears/l-ear-scale-decr|incr	[-1.0, 1.0]
Move vertically	ears/l-ear-trans-down|up	[-1.0, 1.0]
Scale height	ears/l-ear-scale-vert-decr|incr	[-1.0, 1.0]
Scale lobe	ears/l-ear-lobe-decr|incr	[-1.0, 1.0]
Ear shape (pointed-triangle)	ears/l-ear-shape-pointed|triangle	[-1.0, 1.0]
Ear rotation	ears/l-ear-rot-backward|forward	[-1.0, 1.0]
Ear shape (squared-round)	ears/l-ear-shape-square|round	[-1.0, 1.0]
Scale width	ears/l-ear-scale-depth-decr|incr	[-1.0, 1.0]
Ear wing-shaped	ears/l-ear-wing-decr|incr	[-1.0, 1.0]
Ear flapped	ears/l-ear-flap-decr|incr	[-1.0, 1.0]

Table 15: MakeHuman face parameters for Left ear GUI category.

GUI Category: Chin/jaw		
GUI Label	Parameter Key	Range
Tone of side chin	chin/chin-jaw-drop-decr|incr	[-1.0, 1.0]
Cleft chin	chin/chin-cleft-decr|incr	[-1.0, 1.0]
Scale chin prominence	chin/chin-prominent-decr|incr	[-1.0, 1.0]
Scale chin width	chin/chin-width-decr|incr	[-1.0, 1.0]
Scale chin height	chin/chin-height-decr|incr	[-1.0, 1.0]
Scale chin angular	chin/chin-bones-decr|incr	[-1.0, 1.0]
Scale chin prognathism	chin/chin-prognathism-decr|incr	[-1.0, 1.0]

Table 16: MakeHuman face parameters for Chin/jaw GUI category.

GUI Category: Left cheek		
GUI Label	Parameter Key	Range
Cheek outer volume	cheek/l-cheek-volume-decr|incr	[-1.0, 1.0]
Scale cheek prominence	cheek/l-cheek-bones-decr|incr	[-1.0, 1.0]
Cheek inner volume	cheek/l-cheek-inner-decr|incr	[-1.0, 1.0]
Move vertically	cheek/l-cheek-trans-down|up	[-1.0, 1.0]

Table 17: MakeHuman face parameters for Left cheek GUI category.

GUI Category: Right cheek		
GUI Label	Parameter Key	Range
Cheek outer volume	`cheek/r-cheek-volume-decr|incr`	[-1.0, 1.0]
Scale cheek prominence	`cheek/r-cheek-bones-decr|incr`	[-1.0, 1.0]
Cheek inner volume	`cheek/r-cheek-inner-decr|incr`	[-1.0, 1.0]
Move vertically	`cheek/r-cheek-trans-down|up`	[-1.0, 1.0]

Table 18: MakeHuman face parameters for Right cheek GUI category.

Bibliography

[1] V. Blanz and T. Vetter. A morphable model for the synthesis of 3d faces. *SIGGRAPH*, pages 187–194, 1999.

[2] J. Day and N. Davidenko. Parametric face drawings: A demographically diverse and customizable face space model. *Journal of Vision*, 19(11):1–12, 2019.

[3] P. Dou, S. K. Shah, and I. A. Kakadiaris. End-to-end 3d face reconstruction with deep neural networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1503–1512, 2017.

[4] X. Fan, S. Cheng, K. Huyan, M. Hou, R. Liu, and Z. Luo. Dual neural networks coupling data regression with explicit priors for monocular 3d face reconstruction. *IEEE Transactions on Multimedia*, 23:1252–1263, 2021.

[5] L. Jiang, J. Zhang, B. Deng, H. Li, and L. Liu. 3d face reconstruction with geometry details from a single image. *IEEE Transactions on Image Processing*, 27(10):4756–4769, 2018.

[6] J. Lee, J. S. Lumentut, and I. K. Park. Holistic 3d face and head reconstruction with geometric details from a single image. *Multimedia Tools and Applications*, 81:38217–38233, 2022.

[7] Y. Lu, M. Sarkis, N. Bi, and G. Lu. From local to holistic: Self-supervised single image 3d face reconstruction via multi-level constraints. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 8368–8375, 2022.

[8] E. Richardson, M. Sela, R. Or-El, and R. Kimmel. Learning detailed face reconstruction from a single image. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5553–5562, 2017.

[9] E. Wood, T. Baltrusaitis, C. Hewitt, S. Dziadzio, T. J. Cashman, and J. Shotton. Fake it till you make it: face analysis in the wild using synthetic data alone. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 3681–3691, 2021.