UNIVERSITAT DE BARCELONA

Treball final de grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

Graph Convolutional Networks for 3D garment registration

Autor: Ángel Morales Carrasco

Director: Dr. Meysam Madadi
Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, 13 de juny de 2022

Contents

Abstract

The registration of a 3D mesh consists of, given two 3D objects, establishing a correspondence vector for each vertex of one mesh with one of the second one. This process can be used to obtain realistic simulations. Specifically, in this project, I will build the registration of pieces of clothing on realistic models of human bodies. To do so, I will implement an unsupervised machine learning model, using Graph Convolutional Networks, which allow working with non-euclidean data structures, such as 3D objects. This could allow us to build in a fast and efficient way the registration of the garments.

Resum

El registre d'una malla 3D, consisteix en, donats dos objectes 3D, establir un vector de correspondència per a cada vèrtex d'una malla amb un de la segona. Aquest procés pot ser utilitzat per a obtenir simulacions realistes. Específicament, en aquest projecte construiré el registre de peces de roba sobre models realistes de cossos humans. Per a això implementaré un model d'aprenentatge automàtic no supervisat, usant xarxes neuronals convolucionals en forma de graf, que permeten treballar amb estructures de dades no-euclidianes, com ho són els objectes 3D. Això permet construir d'una manera ràpida i eficient el registre de les peces.

Resumen

El registro de una malla 3D, consiste en, dados dos objetos 3D, establecer un vector de correspondencia para cada vértice de una malla con uno de la segunda. Este proceso puede ser usado para obtener simulaciones realistas. Específicamente, en este proyecto construiré el registro de piezas de ropa sobre modelos realistas de cuerpos humanos. Para ello implementaré un modelo de aprendizaje automático no supervisado, utilizando redes neuronales convoluciones en forma de grafo, que permiten trabajar con estructuras de datos no euclidianas, como lo son los objetos 3D. Esto permite construir de una manera rápida y eficiente el registro de las prendas.

Acknowledgements

First, I would like to thank my tutor, Meysam Madadi, for his constant advice and supervision. He introduced this topic to me, and I think he has taught me lots of useful and cool concepts about a field I had much interest in but did not know too much about.

I would also want to thank my family and friends, which have been a motivation to give always the best of myself. In special, to my sister Alicia, who, even if she liked it or not, has always been a referent to me and inspired me to follow my objectives.

Chapter 1

Introduction

1.1 Motivations

Humans have always been obsessed in understanding what is the world we live on, and try to give explanations to what is this reality we seem to perceive.

However, even that we seem to be really far away from understanding it, we have managed to create 3D realities that allow us to imitate the world we see, and even creating new ones. Computers nowadays can renderize 3D objects with thousands of vertices without any effort. This has uncountable applications, from the most artistic ones, like creating video games and movies, to other which could potentially save lives, like medical or car crashes simulations.

In this project, I will be working with 3D objects, and try to construct a model to build the registration of a pair of 3D meshes. The registration of a 3D mesh consists of, given two 3D objects, establishing a correspondence vector for each vertex of one mesh with one of the second one. To achieve that, I will be using Graph Convolutional Neuronal Networks(GCN). The registration process is often part of other complex machine learning pipelines, as the CLOTH3D dataset generation, the first big scale synthetic dataset of 3D clothed human sequences, which will also be used in this project. [1] This could have multiple applications, as creating models for 3D animation software, or hyperrealistic animation generation.

The currently used implementation is the non-rigid Iterative Closest Point(ICP) surface registration [2]. This method, solves the problem analytically, and thus is slow and not very efficient. However, in this project, I will try to implement the registration using neuronal networks and unsupervised learning techniques. From my perspective, this is very exciting, because, with a completely different approach, we are trying to obtain similar results to the existing methods, and once the model is trained it should be way faster than the current methodologies.

Neuronal networks have revolutionized the world of computer science and artificial intelligence. In recent years, we have seen neuronal networks completely change the way of solving well established problems, with a radical change of mentality. Before, we were used to solving complex problems with analytic solutions, using very slow algorithms with high computational complexity that used brute force like techniques to find an optimal solution. However, now, with the so-called machine learning techniques, we have learned that we can *train* models with samples of data or examples, and create techniques to allow the model to learn how to find a local solution. For instance, we have seen the *AlphaZero* chess engine, based on neuronal network models, beat the by tradition most powerful chess engine: *Stockfish*, which analyses in depth the position exploring the different outcomes for each possible move [3]. We are also seeing in the artificial vision field more and more solutions based in Convolutional Neuronal Networks, as image classification or template matching.

From my perspective, this is why I found this topic so interesting, as we have recently discovered the potential of neuronal networks, and for each project there are multiple indirect applications, that we do not even imagine now. Besides, from a personal point of view, I think that doing this project could teach me tons of concepts about machine learning modeling and how to use some widely used machine learning libraries as *tensorflow*.

1.2 Objectives and challenges

This project aims to explore the challenges and possibilities to implement the surface registration with a Graph Convolutional Network(GCN) model.

To do so, we will need to define and implement two GCN models, using unsupervised learning techniques. To do so, I will define some loss functions, which will allow the models to learn how to solve the problem.

The definition of these functions will be critical, as we must use the power of the GPU parallelism to be able to train our models in a reasonable amount of time.

As well, since for each model I will be using multiple loss functions, one key aspect will be how to balance these losses *weights* to be able to return the expected outputs.

After that, I will implement the training pipeline, preprocessing the dataset data to train the models.

Then, to summarize, my main contributions to this project will be:

- Definition and implementation of the Graph Convolutional Network models.
- Implementation of the various loss functions to train the model.
- Losses coefficients balancing to train the models.
- Training pipeline definition and implementation.

1.3 Document structure

This document will be divided in several chapters that will go through the research and work done in this project. The different chapters will be:

- **Introduction**: this section, a brief introduction of the topic. It goes through all the motivations that make this topic interesting and useful, and establishes the scope of this project.
- **State of the art**: a brief explanation of the available registration solution, to give context to why do we want to define another solution with a completely different approach.
- **Theoretical Background**: an introduction to neuronal networks basic concepts, to be able to understand how the Graph Convolutional Neuronal Network models could be able to solve our problem.
- **Methodology**: detailed explanation of the design of our approach to solve the registration problem, going through all the model details and all the decisions taken to define this solution.
- **Results**: contextualization of the dataset and required tools needed to generate the results. Later, we share the results, doing a critical interpretation of them and explaining the decisions taken to generate them.
- **Conclusion**: finally, we analyze the whole project, going through possible improvements and future work related with it, and taking some conclusions about the work done.

Chapter 2

State of the art

2.1 Non-Rigid ICP registration

As mentioned above, the non-rigid Iterative Closest Point(ICP) registration [2] is the currently used method for surface registration. Registering two surfaces means finding a mapping between a template surface and a target surface that describes the position of semantically corresponding points.

Indeed, this algorithm is very complex, and its implementation is complex and very costly computationally. In this section, we will try to explain briefly the basis of this algorithm.

This implementation is based on the ICP algorithm, that basically, given two point clouds, aims to find a transformation that gets the source point cloud closer to the target. To achieve that, at first, we need to find for each point of the source, find the nearest neighbor on the source, a process, that even it can be optimized using several techniques, shows us the high computational demand of this method. After that, we need to estimate the combination of rotation and translation, using a root-mean-square point to point distance metric minimization technique, which will best align each source point to its match found in the previous step. Then, the obtained result will be a point cloud that is closer to the target one. Afterwards, we can repeat all this process until we obtain a point cloud closer enough to the target. The ICP algorithm can be defined as:

- For each point in the source, find the closest on the target.
- Find the transformation that minimizes the distance between closest points.
- Set result as source garment and repeat the first step until the distance is lower than some value.

However, the non-rigid ICP registration method, even that uses the ICP algorithm, introduces a series of restrictions to improve the registration results. The non-rigid ICP registration loops over a series of decreasing stiffness weights, and incrementally deforms the template towards the target. This stiffness weights help the algorithm to obtain closer point clouds but not only based on the nearest neighbor on the target garment, as it allows vertices to move in directions not necessarily going directly for its nearest neighbor.

This stiffness term is added to the global cost function, and what it does is penalizing differences in the transformations of neighbor vertices. This really helps to obtain better results, as we want near vertices to be registered to similar points, and at the start of the iterative process we could find that some neighbor vertices have very different nearest neighbors, so we want to penalize them going into opposite ways. To achieve that, we begin the iterative algorithm with a really high stiffness weight associated to the stiffness cost function. In the following iterations, we start to reduce progressively the stiffness weight, until we find a result close enough to the target point cloud. We can see an illustration of this process on Figure 2.1.

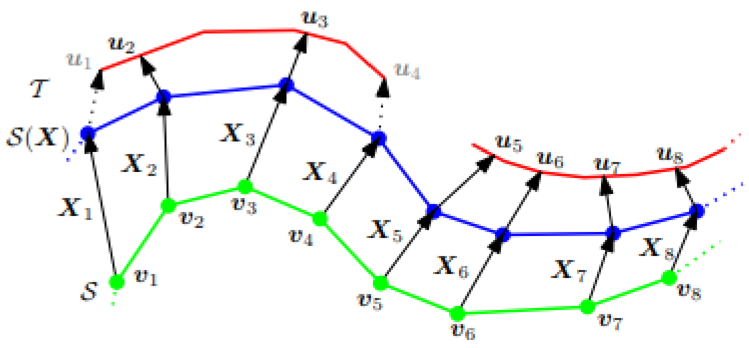


Figure 2.1: The template surface S (green) is deformed by locally affine transformations (X_i) onto the target surface T (red). The algorithm determines closest points (u_i) for each displaced source vertex ($X_i v_i$) and finds the optimal deformation for the stiffness used in this iteration. This is repeated until a stable state is found. The process then continues with a lower stiffness. Due to the stiffness constraint, the vertices do not move directly towards the target surface, but may move parallel along it. (Image extracted from [2])

This method can generate good registration results, succeeds for a wide range of initial conditions, and handles missing data robustly.

After understanding the basics of this method, we could easily guess that its implementation; because of its nature and the multiple costly operations performed in each iteration, as nearest point finding and the cost function minimization, will be slow. This can clearly be a huge restriction, because, if we want to register multiple point clouds, we are forced to wait the necessary time to do all these operations and could cause a clear overhead in other projects that need to use a proper registration technique.

Chapter 3

Theoretical background

3.1 Neuronal Networks

Artificial neuronal networks have generated a lot of excitement in Machine Learning research and industry, thanks to many breakthrough results in speech recognition, computer vision, text processing and many other fields. They have been proven to be an excellent model for solving complex problems. Neuronal networks usually take long times to fully train and adjust all the model parameters, but once this job is done, the model can provide outputs in a practically constant time [4].

3.1.1 Biological inspiration

Indeed, Neuronal Networks try to emulate the brain behavior, which has been proven to be very good at solving quickly very complex problems. The human nervous system is formed by more than 86 billion neurons, which have multiple connections between them. These are connected by the axon of the neuron, which connects to the dendrites of other neurons, and with a chemical process called synapses, it can communicate and give information to other neurons. We can see a representation of a neuron in Figure 3.1.

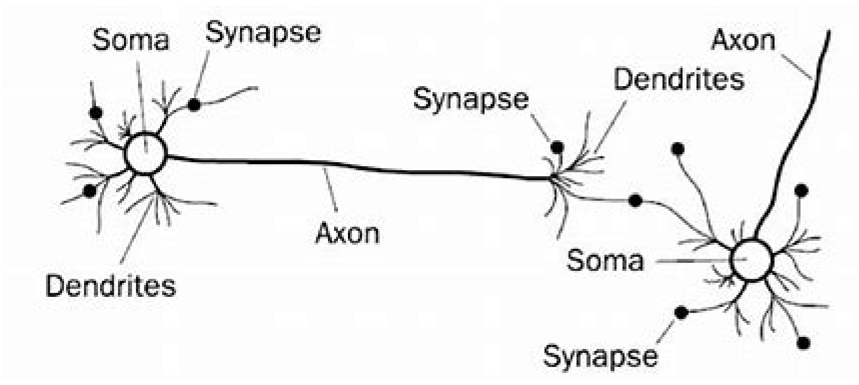


Figure 3.1: Biological neuron model

Then, based on this real complex biological neuronal model, we can construct simple mathematical models that intend to emulate the behavior of it.

3.1.2 The perceptron

The basic unit of every Neuronal Network model is the perceptron. The perceptron can be easily described as a set of input signals x, a set of weights w and a bias b. Then, given a vector of inputs x, we can define the output of a perceptron as:

$$p(x) = (\sum_{i=1}^{m} x_i w_i) + b$$

We can represent this as in Figure 3.2.

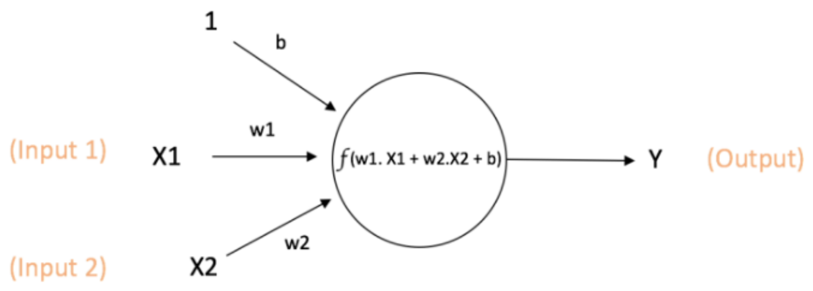


Figure 3.2: Perceptron model image (extracted from [4])

3.1.3 Activation functions

With this simple model, we are just obtaining an output that is a linear combination of the input values. However, most of the world data is not linear, and we want some way to be able to introduce non-linearity to the output of our model. To do that, we can use what we call an activation function, which enables us to adjust the range of output values that we could obtain.

Then, given an activation function f, the output of our perceptron would be:

$$p(x) = f((\sum_{i=1}^{m} x_i w_i) + b)$$

We can use multiple activation functions, and these are some that could be useful in different scenarios:

- Linear activation or no activation: $f(x) = x$, this function does not change the output, and it can be used in some cases when we want to obtain a huge range of values. We can see its shape at Figure 3.3

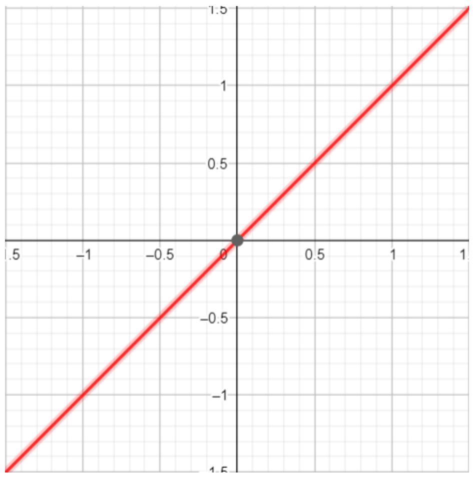


Figure 3.3: Linear activation

- Sigmoid: $f(x) = \frac{1}{1+e^{-x}}$, no matter how big or small the input values are, it adjusts the output between 0 and 1. We can see its shape at Figure 3.4.

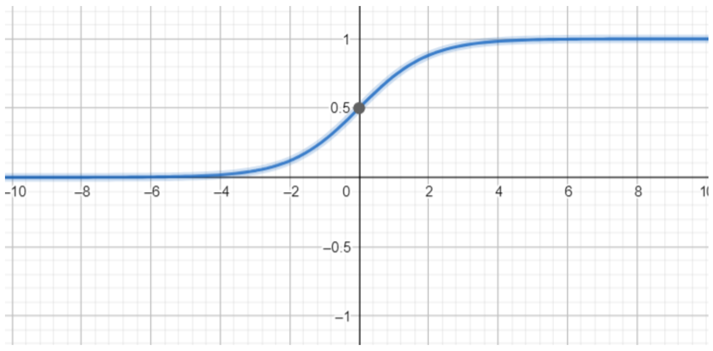


Figure 3.4: Sigmoid activation

- Rectified linear activation (Relu): $f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$, always outputs positive values, and it is widely used as training with Relu is often faster than using others as sigmoid. We can see its shape at Figure 3.5.

There are lots of other widely used activation functions, that can have various uses for training machine learning models.

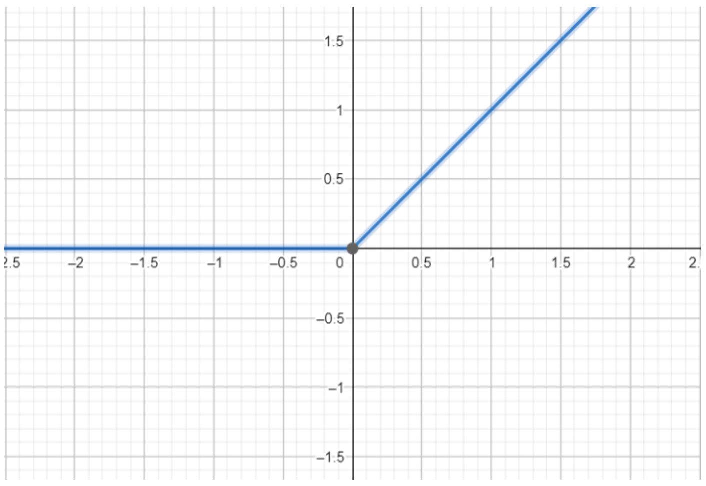


Figure 3.5: Relu activation

3.1.4 Multilayer Neuronal Networks

The perceptron by itself cannot describe very complex problems, as them can only learn linear functions.

However, we can use the concept of perceptron and build more complex networks of perceptrons, creating multiple layers with several perceptrons connected between them, so this way the output of one layer of neurons will be the input of the following one, as shown in Figure 3.6.

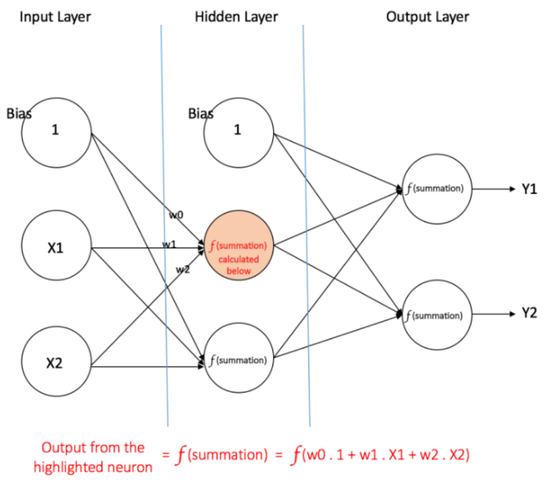

Input Layer
Hidden Layer
Output Layer
Bias
1
Bias
1
X1
X2
w0
w1
w2
f(summation) calculated below
f(summation)
f(summation)
f(summation)
Y1
Y2
Output from the highlighted neuron = f(summation) = f(w0 . 1 + w1 . X1 + w2 . X2)

Figure 3.6: Multilayer Neuronal Network (extracted from [4])

3.1.5 Training the models

Once we have defined our neuronal network model, we will have a model with several weights and biases. These parameters will totally determine which will be the output of our model after introducing an input. Taking into account the nature of the neuronal networks and its topology, it is very likely that the number of parameters of these models will be very big. That is why we need to define a mechanism to automatically adjust the model parameters. To do that, we will use several input examples, and based on the given output and a metric that says us how good is this output, we will *train* the model parameters. If we do this several times, we will see that the model is capable of learning the problem and now knows how to give outputs that are closer to the expected ones.

Here is where the training backpropagation algorithm comes to play [5]. This algorithm is widely used for training machine learning models. Basically, what this algorithm does is to compute the gradient of the loss function (the function that tells how close is the prediction to the result we want to get), with respect to the model parameters. This allows to adjust these parameters iteratively to minimize the loss function. We will not give many details of the actual implementation, as it is very technical, and it is practically transparent in most machine learning frameworks as *tensorflow*.

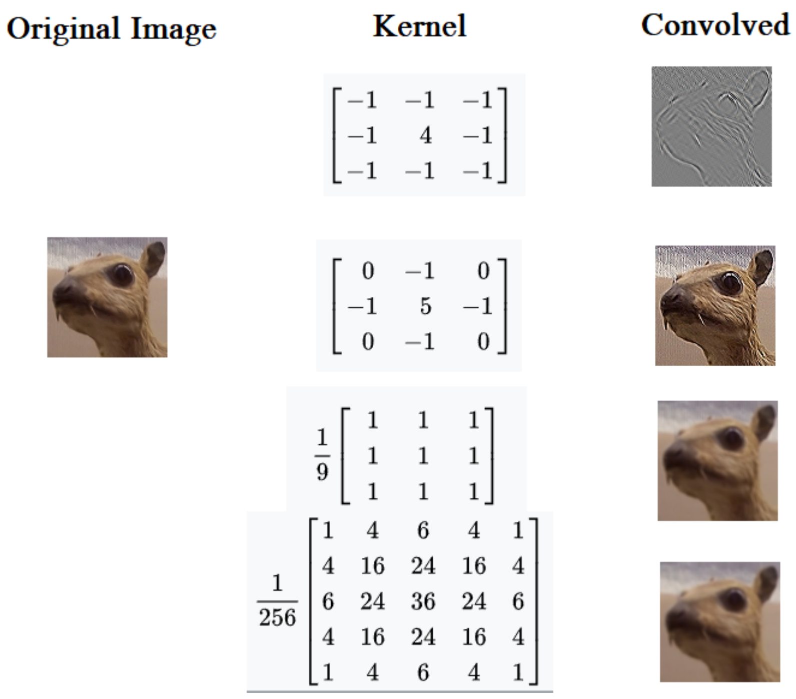
3.2 Convolutional Neuronal Networks

Simple multilayer neuronal networks work well when the input data size is reasonable, and the number of input values is small. However, there are problems that require much bigger inputs. For instance, when dealing with images, if we had one with high definition resolution (1920x1080x3 pixel values), and we would want to use each pixel value as an input, we would have 6220800 input neurons. Then, if we added some layers to the model, we will see that our model has several millions of parameters to train. This model would be impossible to train in any reasonable amount of time.

Nevertheless, Neuronal Network models excel performing artificial vision tasks, as image segmentation, classification and others. Then, to be able to train our models when we input images, we can use the concept of convolution and down sampling to reduce the dimensionality of our inputs. This way, we will be able to train a fully connected neuronal network with the down sampled feature space generated by the convolution layers. A convolution is a simple mathematical operation that can be used to extract some features from images [6]. If $g(x, y)$ is the filtered image, $f(x, y)$ is the original image, and w is the filter kernel and each element of the kernel is inside $-a \leq dx \leq a$ and $-b \leq dy \leq b$, a convolution can be defined as:

$$g(x, y) = \sum_{dx=-a}^{a} \sum_{dx=-b}^{b} (w(dx, dy) \cdot f(x - dx, y - dy))$$

When we apply these convolutions all over the image pixel by pixel using a sliding window, we can obtain very interesting results as using the correct filters we may extract some features of the image as seen in Figure 3.7.



$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 4 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Figure 3.7: Image convolved using different 3x3 kernels (extracted from [6])

Then, using this convolution concept, we can define a series of convolution layers on our model, that use several kernels to transform the input image. We can also add some *pooling* layers, that select certain values from the convolved images to reduce the dimensionality even more. The interesting thing about this is that we do not even have to think about what the values of the kernels should be to extract those significant features that could give information to our fully connected layers, as we can treat the kernel values as they were parameters of our model like the neurons weights and biases, thus, we can also train them to learn how to recognize our problem. We can see a typical Convolutional Neuronal Network [7] shape in Figure 3.8.

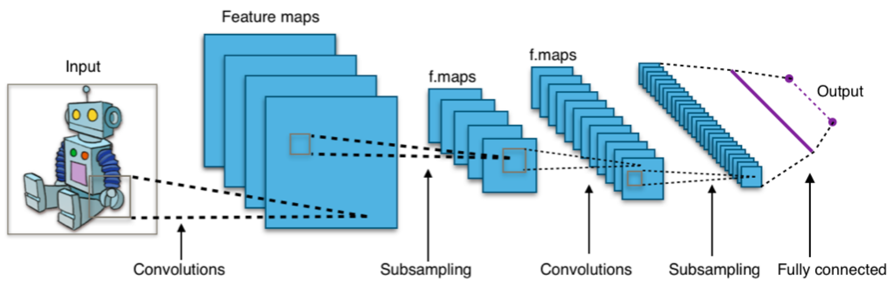


Figure 3.8: Representation of the shape of a typical Convolutional Neuronal Network model (extracted from [7])

3.3 Graph Convolutional Neuronal Networks

Convolutional Neuronal Networks, work with Euclidean data. That is why they excel working with images, which are often in the $\mathbb{R}^3$ space. In this case, we want to work with 3D objects, which are a non-euclidean data structure, which can be defined as a graph. That is why we will be using Graph Convolutional Networks, a Neuronal Network model which takes advantage of some concepts of the simple Convolutional Neuronal Networks, but can work with graph shaped data. [8] [9]

However, we can use some concepts of the Convolutional Neuronal Networks to be able to define our graph based Neuronal Networks, that will be able to work with non-euclidean data. As we see in Figure 3.9, indeed two-dimensional euclidean data, as images, can also be represented as a graph. We can reuse the concept of convolution to define a model that applies *convolutions* to our graph. The only difference here, is that we cannot define any kernel to convolve our graphs, because the number of neighbors of every vertex is not regular.

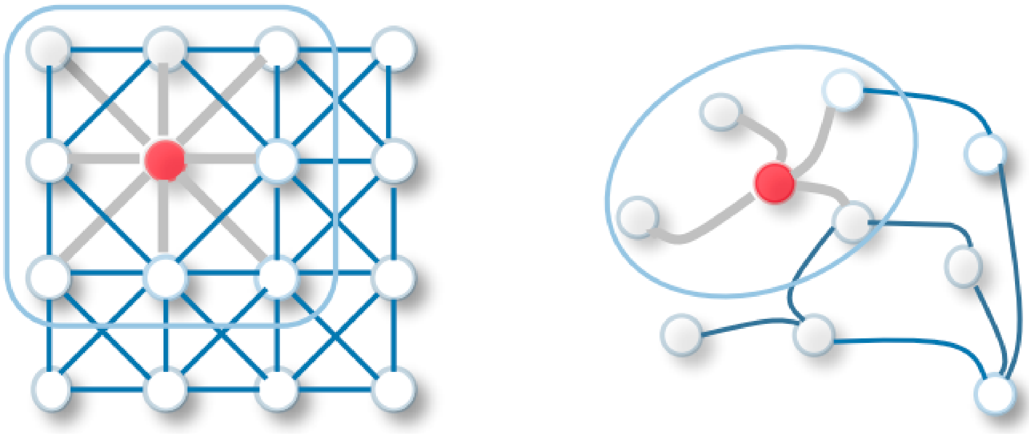


Figure 3.9: Representation of a two dimensional euclidean structure (left), versus a non-euclidean graph (right) (extracted from [9])

Then, we could just simply define the graph convolution as the weighted sum of all adjacent vertices multiplied by a weight, that will be our model parameter to this convolution. This way, we will be able to convolve any graph, no matter how many vertices does it have and how many adjacencies every vertex has. Then, having our graph G, two sets of weights w_0 and w_1, a bias b, the activation function f, and being $A*$ the normalized laplacian matrix that defines all the graph adjacencies, the graph convolution can be defined as:

$$\forall V \in G : convolveNeighbors(G) = \sum_{i=0}^{N} V \cdot (A*)_i \cdot w_1$$

$$convolve(G) = f(G \cdot w_0 + convolveNeighbors(G) + b)$$

Then, we can train our models setting the w_0, w_1 and b as additional model trainable parameters, and combining it with a fully connected neuronal network at the end of the graph convolution layers.

At first, we could think that the power of the graph convolutions is limited, as we can only convolve using the nearest neighbors of each vertex. However, if we use multiple layers, indeed the values of vertices further away from the local neighborhood have an influence in the result, so with multiple graph convolutional layers we will be able to detect global and local patterns that will help our model to detect and solve problems.

Chapter 4

Methodology

4.1 Model design

To build a Graph Convolutional Network model that learns how to build the 3D garment registration, I will need to define a model and a training pipeline.

To do that, we will propose two different models, a *stretching* model, and a registration model. In our case, we are loading garment samples from a dataset, which have a particular shape and position, that could be more or less arbitrary. To solve our problem, we are actually interested in the topology of our garment, and we do not really care what is the initial state of the garment. For instance, we could have multiple 3D models, that represent the same piece of clothing, but its initial state is different, then if we directly applied a model to register these garments we would obtain different results, even that it is the same cloth. That is why, before registering the garment, we want to build a model that stretches it, and tries to smooth the garment shape. If we do that this way, ideally, for different 3D objects that represent the same garment with different initial states, the stretching model should be able to output the same stretched version for the two different samples.

After that, we will use the output of the stretching model, and define a second model to learn the actual registration.

Then we have to consider how are our models going to learn to solve the problem. In our case, we do not have a training dataset which has the initial garment and an associated output to train with, so for each sample we do not know *a priori* which will be the desired output. This is why we will be using **unsupervised** learning techniques. To do that, we will have to think and define a set of what we will call the loss functions of our problem.

These functions will take the output state of the model, and evaluate some metrics that we want to minimize in our problem, and then, computing the gradients of these functions, we will be able to train the model in order to minimize the loss functions.

4.2 Stretching Model

As mentioned before, the stretching model will be processing the input garments and construct an smoothed version of them.

First, we have to define the number of layers and shape of the Graph Convolutional Network that we are going to use to construct this model.

Indeed, choosing a good model topology is a really hard task, as it is very difficult to estimate in an objective way which network shape will give better results, and as Neuronal Networks take a lot of time to train, we cannot test so many configurations. Then, in our case, we will be choosing a not very complex configuration that we tested out that was giving reasonable results, and we will stick to it during all the project.

This is a five layer Graph Convolutional Network, with a Fully Connected layer at the end to reduce the feature space, because as we are working with 3D garments, the input and output graphs must be of shape *N X 3*, being *N* the number of vertices of the outfit, and the 3 Cartesian coordinates of the vertex position. We can see a representation of this model structure in Figure 4.1

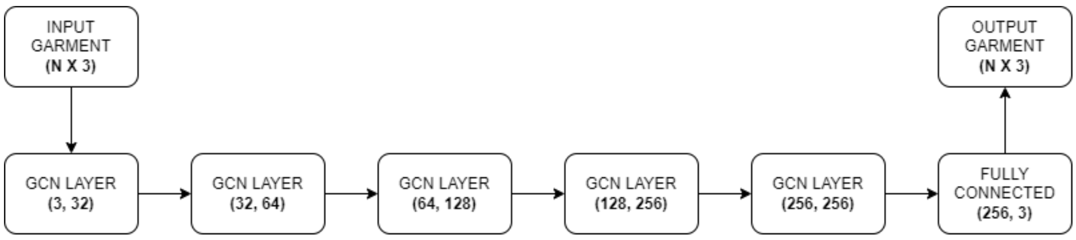


Figure 4.1: Graph Convolutional Network shape

4.2.1 Loss functions

Now, we have to define the loss functions of the network. These functions are the way we have to indicate to the model what do we want to achieve, as in the training stage, the parameters of the model will be adjusted in order to minimize the values of these loss functions. Indeed, the model will try to minimize a single value, the total loss, with will be a combination of all the defined loss functions multiplied by a coefficient. Each loss will have associated a coefficient, that will indicate how much important is this loss in our model. Adjusting these coefficients is not an easy task, as sometimes the behavior of the models while training is unexpected, and we have to take into account lots of factors to adjust these values.

Normal loss

As we commented before, one of the objectives of the first model is to smooth the surface of the garment. To do that, we could consider the normal vectors of the garment. The normal is a vector which is perpendicular to the surface at a given point, as we can see in Figure 4.2 [10].

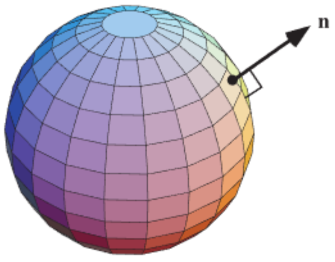


Figure 4.2: Normal vector of a vertex (extracted from [10])

Then, in order to smooth the surface, we want to construct a metric that evaluates how close are the normal vectors of close vertices. This way, we are penalizing irregularities on the garment surface.

A first approach to implement this metric would be to compute the standard deviation over the normals of adjacent vertices of each vertex of the garment, which given a garment with M vertices, and being the N_{ij} the normal of the adjacent vertex j to the vertex i can be defined as:

$$loss(G_M) = \sum_{i=1}^{M} std(N_{i1}, N_{i2}, ..., N_{in})^2$$

Nevertheless, this approach is very difficult to implement efficiently, using the power of the GPU and *tensorflow* functionalities, because with this formulation, we would need to iterate over each vertex of the garment and find all of his neighbors. This shows clearly the difficulty of this problem and defining loss functions, as this loss definition, even that is correct and evaluates the metric we want, is not possible to implement.

However, we can choose a different approach, instead of computing the standard deviations, we can use the dot product, that can be expressed as:

$$dot(a, b) = ||a|| \cdot ||b|| \cdot cos(\alpha_{a,b})$$

Indeed, as the norms of the normal vectors are always one, we know for sure that the dot product will be always $cos(\alpha_{a,b})$. Because of that, this value gives us a really clear measure of how similar two normal vectors are.

Then, we can, in a very efficient way, apply a graph convolution that computes the dot product of all vertices with all his neighbors. At each vertex we will have a value between -1 and 1, 1 meaning that all the adjacent vectors have exactly the same direction, and -1 otherwise. That is why the final metric will be 1 minus the mean of all the vertices' convolution. We can implement this function using *tensorflow* methods as following:

```python
def normal_loss(V, F, L):
  # V: mesh vertices
  # F: mesh faces
  # L: Laplacian adjacencies matrix

  # Compute the Normals
  N = tfg.geometry.representation.mesh.normals.vertex_normals(V, F)

  # Apply convolution to compute dot product
  D = tfg.geometry.convolution.graph_convolution.edge_convolution_template(
  N, L, sizes=None,
  edge_function= lambda x,y: tf.einsum("ij,ij->ij", x, y),
  reduction='weighted',
  edge_function_kwargs={}
  )
  # Sum values to complete dot product and compute the mean
  return tf.reduce_mean(1 - tf.reduce_sum(D, axis=1))
```

Listing 4.1: Normal loss function

Collision Loss

One of the objectives of this models is also to separate the garments a bit from the body. Also, we would consider an incorrect output one that collides with the body mesh, as we do not want the garment inside the body.

For that, we will be using the collision function implemented in the DeepSD [11] code, that given two 3D meshes gives a metric given the number of faces of the two meshes that intersect.

Distance Loss

As we want to separate the garment from the body, we are also defining a metric that computes the differences between the positions of all vertices. To do that, we need to define two *tensorflow* tensors, one with the garment values and with shape $(M, 1, 3)$, being M the number of vertices of the garment; and another with the values of the body with shape $(1, N, 3)$, being N the number of vertices of the body. With *tensorflow* we can use the subtract operand with different shaped tensors, and we will be actually obtaining a $(M, N, 3)$ tensor, which contains all the differences between each body vertex and all the garment vertices. Once we have this result, we can compute euclidean distances all over the tensor and compute the sum of all of them to obtain a metric to indicate the overall distance body to garment. We can implement this function using *tensorflow* methods as following:

```python
def distance_loss(B, G):
  # B: body vertices tensor
  # G: garment vertices tensor

  garment_reshape = tf.reshape(G, (G.shape[0], 1 , G.shape[1]))
  body_reshape = tf.reshape(B, (1, B.shape[0], B.shape[1]))

```

```
#Compute all euclidean distances and sum them
d = tf.reduce_sum(tf.square(1 - tf.reduce_min(tf.sqrt(tf.reduce_sum(tf.
  square(garment_reshape - body_reshape), -1)), -1)))

return d
```

Listing 4.2: Distance loss function

Vertices Position Loss

In most of the cases when training machine learning models, we also have to define some loss functions that we could consider that are not so useful for obtaining the desired output, but help to stabilize the training process. These are the regularizers of our model, and they help us to maintain it stable and avoid extreme over-fitting behaviors.

In our case, if we just used the loss functions above, we could find that the model outputs are not something we would expect, as for instance it could find that returning a completely plane garment would minimize completely the normal loss, or giving a garment really far away from the body to improve the distance loss. That is why we will define a loss that computes the differences between the original garment and the predicted one. If we have two garments, G being the original garment, and P the predicted one, both with N vertices, this loss can be easily defined as:

$$verticesLoss(G, P) = \sum_{i=1}^{N}(G_i - P_i)^2$$

This function can clearly help us to have results that in some way resemble the original garment. Obviously, our main objective here is to transform the original garment, this is why this loss may seem a bit counterintuitive, but we also do not want to obtain an output that does not have any relationship with the original garment.

Edges distance loss

As we did with the vertices position, we can do the same with the edges lengths. We could define another regularizer that penalizes the model for changing a lot the edge lengths. This way we will not return garments with very different shapes from the original one, as we are enforcing the edge lengths to be similar to the input garment. Having two garments, G being the original garment, and P the predicted one, both with N vertices, and being $X_{i,j}$ the edge length of garment X from vertex i to vertex j, we can define this loss as:

$$edgesLoss(G, P) = \sum_{i=1}^{N}\sum_{j=1}^{N}(G_{i,j} - P_{i,j})^2$$

4.3 Register Model

Once we have defined our stretching model, we can start with the model that will actually register the garments.

To do that, first we have to think about how will be the shape of our Graph Convolutional Network Model. As we said before, this is a really complex task, and to keep things simple, we are going to reuse the model shape used in the stretching model (already seen in Figure 4.1). We think this is the best option to start with, as we already tested that this model is able to learn how to stretch our garment, so we will assume this topology is good for recognizing and transforming graph shaped outfits.

4.3.1 Loss functions

After that, we need to define the particular loss functions that will help us to solve our problems. We will be probably reusing some functions that were used in the stretching model, but we will also need to repeat the task of adjusting the coefficients of the different losses that we have defined to obtain the expected results.

Chamfer distance Loss

Foremost, we will define the main function of the model. In our case, our objective is to establish a correspondence between each garment vertex to the body vertices. To do that, we would like to minimize the distance between each body vertex and the closest one in the garment. To achieve that, we can implement an existing metric called Chamfer Distance. Given two point clouds *S1* and *S2* for each point in each cloud, the chamfer distance finds the nearest point in the other point set, and sums the square of the distance up. It can be defined as:

$$ChamferDistance(S_1, S_2) = \frac{1}{|S_1|}\sum_{x\in S_1} min_{y\in S_2}||x-y||_2^2 + \frac{1}{|S_2|}\sum_{x\in S_2} min_{y\in S_1}||x-y||_2^2$$

Seeing its definition, we would expect that when this distance is minimized, we will obtain the registered outfit.

Luckily, to implement this loss, we can use a module from the *tensorflow-graphics-gpu* library. The function *tfg.nn.loss.chamfer_distance.evaluate* is able to return this metric in an efficient time, using GPU parallelism power.

Registration Vectors Similarity Loss

By the registration nature, we would expect that vertices with similar position, will be registered to similar vertices in the body. Thus, we can force this behavior stating a loss function that tries to minimize the differences between the registration vertices of each neighborhood of vertices.

A first approach to implement this metric would be to compute the standard deviation over the registration vectors of adjacent vertices of each vertex of the garment, which given a garment with M vertices, and being the R_{ij} the registration vectors of the adjacent vertex j to the vertex i.

$$loss(G_M) = \sum_{i=1}^{M} std(R_{i1}, R_{i2}, ..., R_{in})^2$$

As we see, this loss definition is practically the same that the one we were defining in the stretching model with the normals. We can also use in this case the trick to be able to use a graph convolution and compute the registration vertices differences using the dot product. We remember that the definition of the dot product can be expressed as:

$$dot(a, b) = ||a|| \cdot ||b|| \cdot cos(\alpha_{a,b})$$

However, in this case, we do not know for sure what will be the lengths of the registration vectors, and in some cases will be bigger than one. So in this case, the dot product will not be the most accurate metric to measure vectors differences, but in practice it still gives us a metric about how similar the vectors are, as $cos(\alpha_{a,b})$ is always multiplying to obtain the result.

In practice, we have tested that, for our samples, this registration must never be bigger than 10, but she should take into account take this may vary with other samples or datasets.

Then, we can define this loss similarly to what we did with the normal loss, applying a graph convolution to compute the dot products of the neighborhood of each vertex, and then the final metric will be 10 minus the mean of all vertices' convolution. We can implement this function using *tensorflow* methods as following:

```python
def register_vectors_loss(stretched, predicted, L):
  # stretched: vertices of the stretched garment
  # predicted: vertices of the predicted garment
  # L: laplacian matrix of both garments

  register = stretched - predicted

  # Convolution to compute partial products of the dot product
  D = tfg.geometry.convolution.graph_convolution.edge_convolution_template(
  register, L, sizes=None,
  edge_function= lambda x,y: tf.einsum("ij,ij->ij", x, y),
  reduction='weighted',
  edge_function_kwargs={}
  )
  # Compute 10 - dot products and see the means
  return tf.reduce_mean(10 - tf.reduce_sum(D, axis=1))
```

Listing 4.3: Register Vectors similarity loss function

Edge Similarity Loss

We expect the outfit to be registered in a uniform way, so we do not want vertices to very far away from other vertices, thus the registered garment should not have very different edge values. Then, we can simply define a loss that computes the standard deviation over all edge values in the garment. We expect to obtain values close to 0, as we do not want huge differences between edges. Being G_{edges} the list of all the garment edges distance, we can define this loss as:

$$edgeSimilarity(G) = std(G_{edges})$$

Collision loss

As with the stretching model, we also consider that colliding with the body should be penalized. Then the function used in that model will be reused.

However, we should note that in this case the penalty for colliding should be way lower, as we expect some collisions as our objective is to bring the garment really close to the body. Later, we should use a smaller coefficient for the collision loss.

Vertices position loss

As with the first model, we also require some way to stabilize the model. The vertices position of the output should not be very far away from the original vertices. As we commented in the first model, even though our objective is to transform the input, we still need some way to stabilize the training process and regularize it. We will use the same loss function defined in the first model.

4.4 Training pipeline

To define our training pipeline we have reused some code used in the DeepSD project [11] to train the models. In our case, we have defined two python scripts *train_stretch.py* and *train_register.py* to implement the training of each model.

Indeed, the training process is not so complex, we just need to load the required samples, using the DeepSD data management functionalities, and then, for each step in the training process, compute the parameters gradients. We can do that in a very simple and transparent way using the *tensorflow GradientTape* functionality. When we define a section of code with a *GradientTape* watch, all the gradients are automatically computed when different operations are applied. Then, after computing all the loss functions, we can easily obtain the loss gradient with respect to the model parameters, and we can adjust them using a *tensorflow* optimizer, in our case we are using Adam, which is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments.

We should also note that all the losses and operations defined in the training pipeline are prepared to work with multiple batches. That means that we can train with multiple samples at the same time. However, for technical reasons, we are using one batch at the time, because we need huge GPU memory space to compute some loss functions, and if we use more than one sample we may face memory allocation errors.

4.5 Usage of an autoencoder

While training the networks, we realized that the first stages of training are very chaotic. In the first initialization, the values of all the parameters of the model are random, so the first output meshes have also random positions. Sometimes, the outfit would appear inside the body, making it very difficult for the model to learn how to get out of it, because the collision loss would penalize the model if it tried. That, and other factors, made the first steps of training very unpredictable, and generated inconsistent results with the same or similar models.

However, to solve that problem, we could use what is called an autoencoder. This is a simple model that learns how to represent the input. This way, instead of starting training with random values on the model parameters, we would start with some values that are already trained and know how to represent the input, and thus we are not facing the instability of the first training epochs that we were facing before. Autoencoders are widely used in the machine learning world, as they can be useful in many situations and often can improve the performance of the models.

To train our autoencoder we can just use the vertices loss, as we want to minimize the differences of the positions between input and output garments. We can see the difference between the original outfit and the predicted by the autoencoder in Figures 4.3 and 4.4:

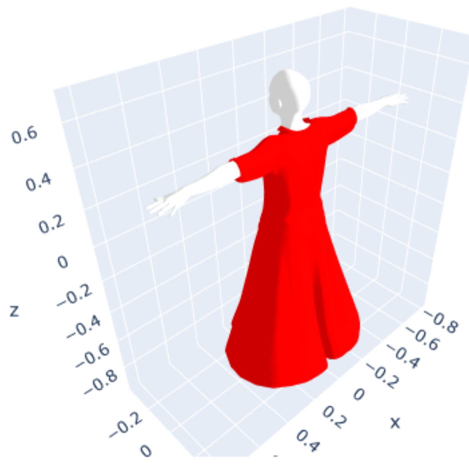


Figure 4.3: Visualization of a dress outfit

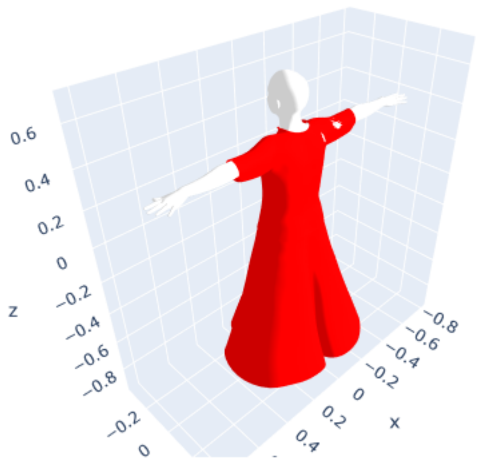


Figure 4.4: Output of the autoencoder when outfit in Figure 4.3 is given as an input

As we can see, the prediction of the autoencoder model is not perfect, but we do not need it to be so. We just need having a model that knows how to reproduce some result that is able to stabilize the first epochs of training.

Chapter 5

Results

5.1 The Dataset

The dataset I will be working with is the CLOTH3D dataset [1]. It is the first big scale synthetic dataset of 3D clothed human sequences. CLOTH3D contains a large variability on garment type, topology, shape, size, tightness and fabric. Clothes are simulated on top of thousands of different pose sequences and body shapes, generating realistic cloth dynamics.

To train the model, we will need loading different outfits, composed of one or more garments, and a body. The CLOTH3D dataset does not contain the body models, because to generate them, it uses the SMPL model [12], which enables us to generate realistic body models with different shape and pose parameters. So, we will need to load the SMPL body shape parameters and generate the body in a rest pose using SMPL.

However, training a model with this dataset it is not so easy, as we have to take into account several technical details, as we want to quickly extract the data, and we do not want to lose any unnecessary time processing the data while training. To do that, I will be using the preprocessing and data extraction scripts used for the DeepSD model [11], which also uses the CLOTH3D dataset to train its model to perform realistic garment animations.

In Figure 5.1 we can see a loaded outfit on top of the SMPL body in rest pose.

5.2 Used Tools

To implement the proposed models, we are going to use the python programming language, combined with the use of several libraries.

5.2.1 General use libraries

We are using several libraries that provide us with some useful functionalities that make the coding easier, and they are very common while coding in python, as *numpy* for nu-

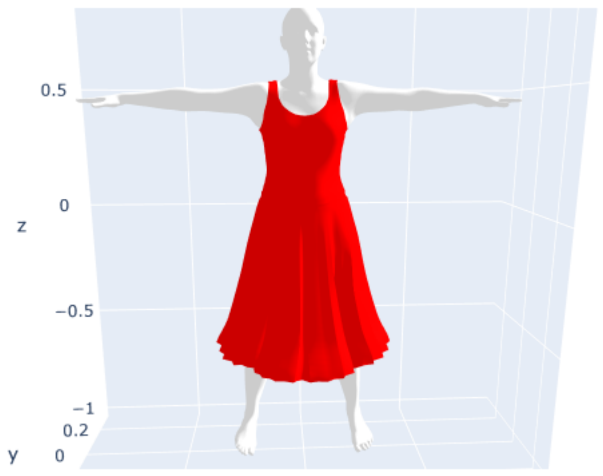


Figure 5.1: Cloth3D outfit sample, loaded with its body model in rest pose

merical operations, *plotly* for 3D meshes visualization, *scipy* for general scientific purposes as sparse matrix operations and others.

5.2.2 Tensorflow

Tensorflow provides a set of functionalities to build and create machine learning models. It allows us to easily create the models, and train the network weights.

The training process of neuronal networks is often very slow, but *tensorflow* provides a key feature that allows us to use a GPU to train the model in a parallel way. This makes the training very fast compared with the CPU implementation.

Also, we will need to define all the loss functions using the *tensorflow* methods, which are also GPU friendly. A correct implementation of the loss functions using the *tensorflow* functionalities can be the difference between hours of training or just a few minutes.

Besides, *tensorflow* has a submodule called *tensorflow-graphics-gpu*, that provides several functionalities to work with graphs implemented with GPU friendly methods, as graph convolutions, and will be widely used in this project.

5.2.3 Google Collab

As mentioned before, using a GPU for training is very fast compared to CPU training. That is why we will be using Google Collab. It provides a virtual environment in which we can run python scripts, and enables us to use for free a GPU to execute them.

This has multiple advantages, as we do not have to configure any physical GPU. The GPU also has good memory specifications (12 GB), and this also will be very helpful as we are processing very big graphs containing garment information.

However, the time that we can use to train models on Google Collab is limited, and this could limit our model parameters tuning.

5.3 Model Results

Once we had defined and implemented the models, it was time to test them and see its results. However, this is a very complex process, as in our case we need to fine tune all the models' hyperparameters, the different loss coefficients. We also have some technical limitations, as to fully train our models, we need to use a GPU with big memory space, and the training procedure takes very long times. In our case, we are using *Google Collab* to generate them, and using this free environment provided by Google, we have limited training time, so we cannot train the models so long.

Nevertheless, I think the objective of this project should never be to obtain perfect results. I think the most interesting part should be analyzing some results obtained and trying to understand what is happening, how they can be improved, and provide a strong base to be able to upgrade these models in order to be able to solve the registration problem using machine learning based methodologies.

This is why, in this particular case, we will be generating 5 different models, and in each one we will be using only a particular sample, and try to generate models training with them. The chosen samples from the CLOTH3D dataset are:

- 00016: outfit of trousers and a T-Shirt. Figure 5.2.
- 01643: outfit of a dress. Figure 5.2.
- 02179: outfit of a jumpsuit. Figure 5.2.
- 04846: outfit of a trouser and a top. Figure 5.2.
- 05207: outfit of T-Shirt and a skirt. Figure 5.2.

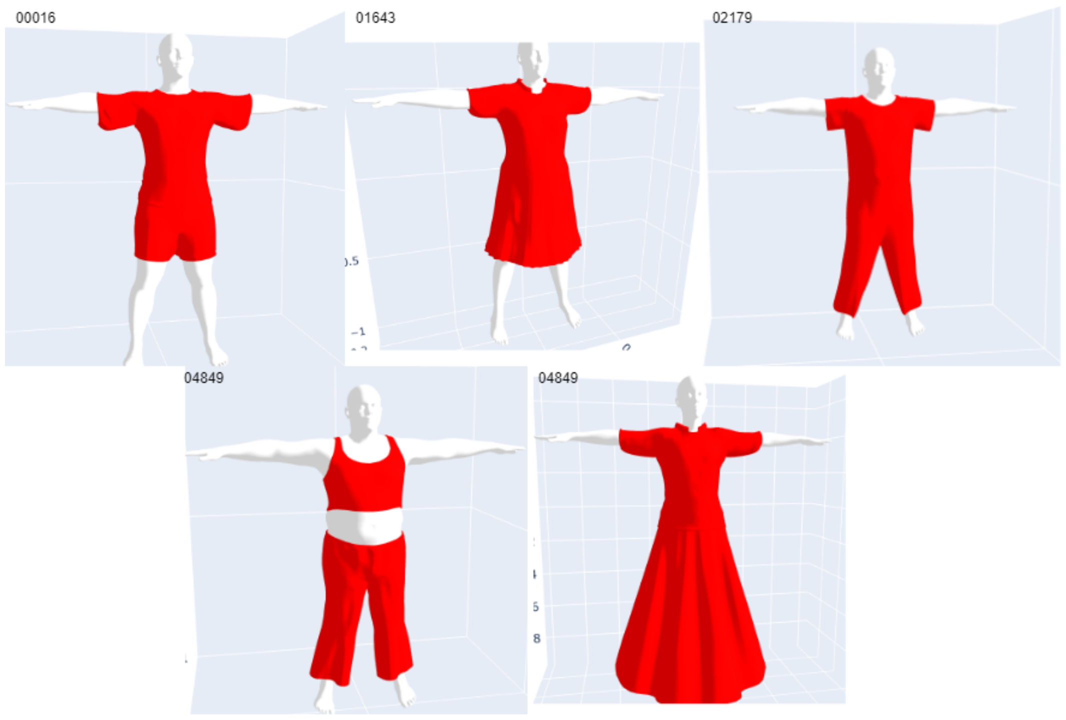


Figure 5.2: Loaded Samples of the different outfit that will be used to train the different models.

5.3.1 Stretching Model Results

First, we have generated the results for the stretching model. It is very difficult to judge in an objective way the correctness of these results, as we do not have a very clear output objective. For the five different samples used, we have created five different models and trained them with only the selected sample 100 epochs. The resulting outputs for each model are:

- 00016: stretched outfit of trousers and a T-Shirt. Figure 5.3.
- 01643: stretched outfit of a dress. Figure 5.3.
- 02179: stretched outfit of a jumpsuit. Figure 5.3.
- 04846: stretched outfit of a trouser and a top. Figure 5.3.
- 05207: stretched outfit of T-Shirt and a skirt. Figure 5.3.

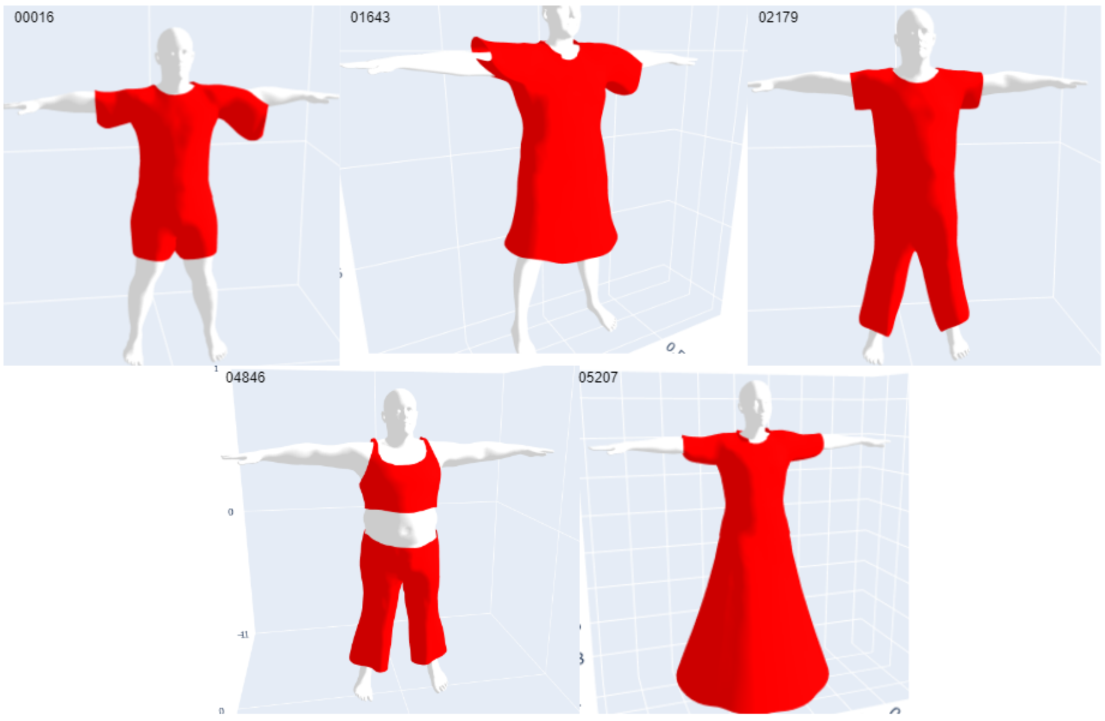


Figure 5.3: Output outfits after stretching.

The used coefficients and the different loss values at the final training epoch(100) can be found in the following table:

	Normal Loss	Collision Loss	Distance Loss	Vertices Loss	Edge Loss	Total Loss
Coefficients	1.0e+3	7.5e+2	1.0e-5	3.0e-1	2.0e+0	-
00016	7.217782	0.36556184	0.100651726	1.2818129	0.63180006	9.597609
01643	10.70532	1.0065246	0.074037805	2.8015463	1.7641408	16.35157
02179	4.807514	0.0029138895	0.12792708	0.36674708	0.1531826	5.458285
04846	5.083509	0.010080771	0.10781584	0.3017217	0.1691572	5.6722846
05207	7.75503	0.0035081136	0.10921732	0.82273716	0.38270578	9.073199

We can also see how the losses values evolve during all the training epochs in Figure 5.4.

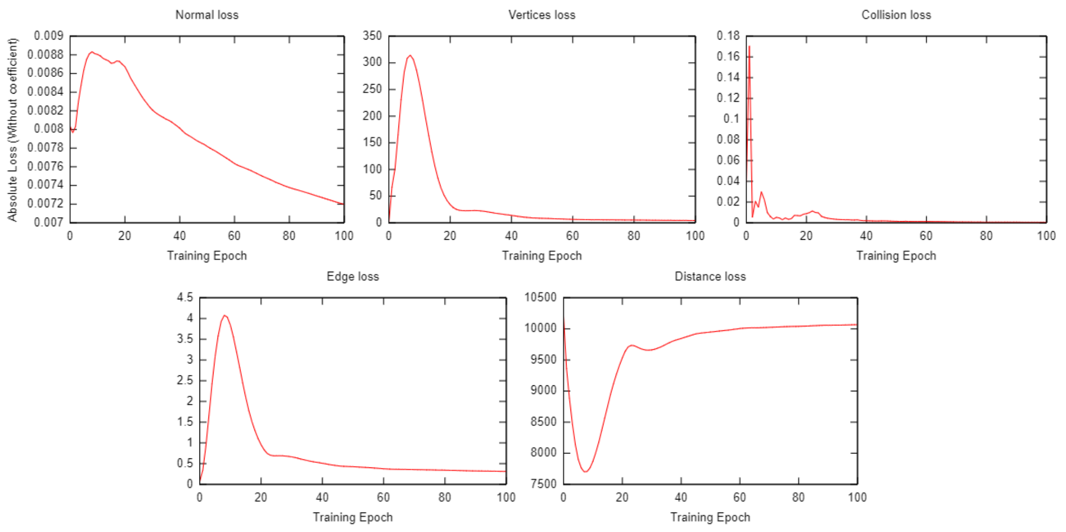


Figure 5.4: Losses values during all training epoch of the stretching model with sample 00016.

In general, the stretching model results should be good for the purpose we want to use it. If we examine the different samples, we can observe that have not changed a lot compared to the input garments. Indeed, we have seen that this helps the second model performance, as we do not want a garment in a very different position or very far away from the body, as it would cause problems later when registering the garment. That is why we have chosen to give a very small coefficient to the distance loss, and very significant coefficients to the vertices and edge regularizers. The main loss, the normal loss, is the one that has higher coefficients, and we can clearly see its work as it has smoothed the garment's shape. We can clearly see the work it has done in Figure 5.5.

However, if we observe the losses table, we can see some clear unexpected behaviors. We see that some losses have unexpected values, for instance, we see that the collision loss has very high values in some samples, as in the sample 01643. We would expect high loss values in other losses, as the regularizers or the normal loss, as we know for sure that it is impossible to obtain outputs with all these losses being 0, but in theory we would never want any face of the outfit to collide with the body.

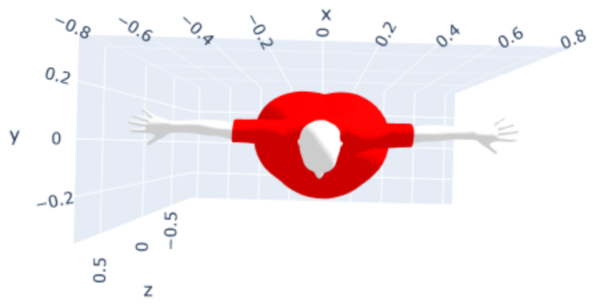


Figure 5.5: Sample 05207 stretched, seen from a top view to observe how the dress shape has been smoothed significantly.

Then, we could think that one solution to this would be to increase drastically the collision coefficient. But, this only shows the complexity of the problem, and why adjusting the coefficients is so hard. Even that, we would not want any collision at the final model, this loss is a part of the learning process. If we penalized completely any collision, the model would not be able to tell any difference between results that are close to being correct, and those who are completely wrong. In practice, what we see if we increase the collision coefficient excessively, is that the gradients of the loss completely push the outfit far away from the body to avoid any collisions. And surprisingly, it is sometimes not even able to avoid all the collisions, as we can see in Figure 5.6.

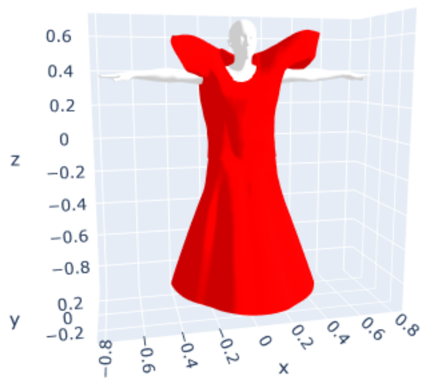


Figure 5.6: Sample 05207 stretched, but using collision loss coefficient 7.5e+4. Because of the high collision penalty, the model tries to minimize the number of faces that collide with the garment, instead of trying to adjust the arms in the dress.

In this example, we can clearly see the nature of neuronal network models. These models are very good at finding **local** minimums, but this does not ensure us that this minimum is the global. While working with these models, we should always be aware of this behavior, and try to deal with it.

In our case, what this problem causes is some artifacts on the outputs, as we observe some collisions in the outputs or some results that seem to have certain parts that are not well stretched. We can clearly see that on the sample 01643 5.3. However, we should ask ourselves if these problems have any influence on the second model, and the reality is that they have few or none. We have to remember that in this first model, what we only want is to generalize the outfits to make the second model output independent of the initial garment state. So, these little artifacts will not cause any problem, and we can consider that the outputs of this first model are quite correct.

5.3.2 Register Model Results

After defining and obtaining the results of the stretching model, we can start testing and obtaining results of the registration model. That is one of the reasons why tuning this model is a particularly difficult task, as we do not only need to adjust its hyperparameters, we also know that it directly depends on the output of the other model.

As with the previous model, we have used the 5 selected samples and generated one model for each one, and trained them 100 epochs. The resulting outputs of the model are:

- 00016: registered outfit of trousers and a T-Shirt. Figure 5.7.
- 01643: registered outfit of a dress. Figure 5.7.
- 02179: registered outfit of a jumpsuit. Figure 5.7.
- 04846: registered outfit of a trouser and a top. Figure 5.7.
- 05207: registered outfit of T-Shirt and a skirt. Figure 5.7.

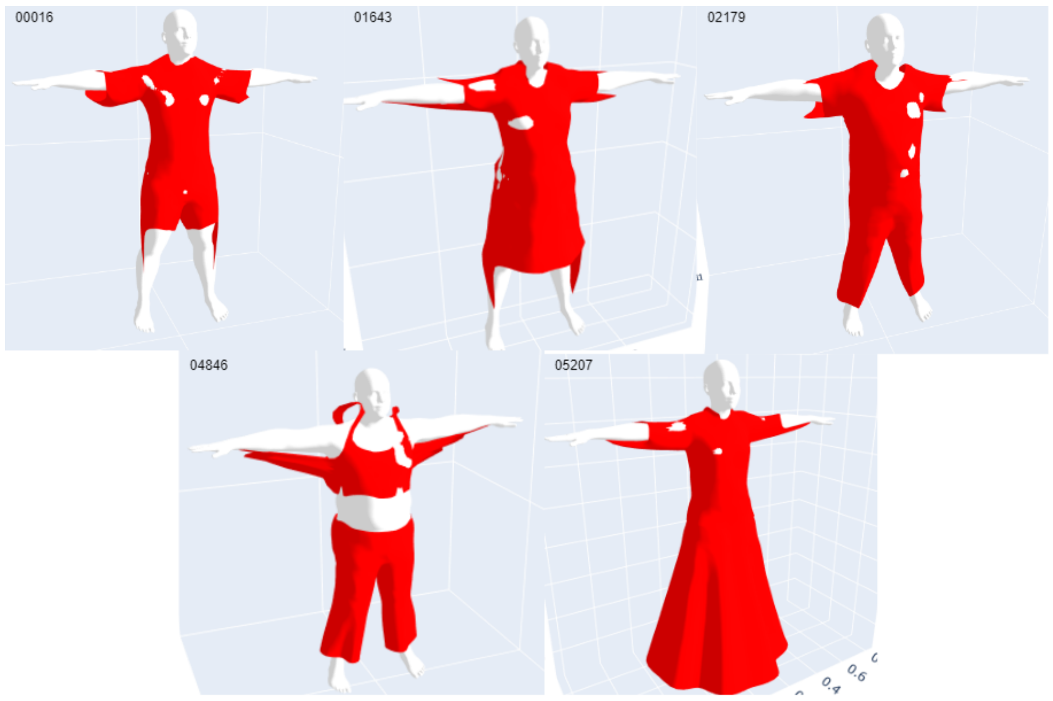


Figure 5.7: Output outfits of the registration model.

The used coefficients and the different loss values at the final training epoch(100) can be found in the following table:

	Chamfer D. Loss	Vertices Loss	Register Vect. Loss	Collision Loss	Edge Loss	Total Loss
Coefficients	5.0e+2	1.0e+0	1.0e-1	1.0e+0	2.0e+1	-
00016	7.4765115	4.7933254	0.99992514	0.060638268	0.13096592	13.461367
01643	5.883686	5.031674	0.99986124	0.04409196	0.24721038	12.206524
02179	3.7493124	3.374235	0.9999687	0.07047841	0.11340179	8.307397
04846	4.329788	5.1768384	0.99996024	0.05137386	0.2648859	10.822847
05207	9.41104	3.3697863	0.9999596	0.015292238	0.16460073	13.960679

We can also see how the losses values evolve during all the training epochs in Figure 5.8.

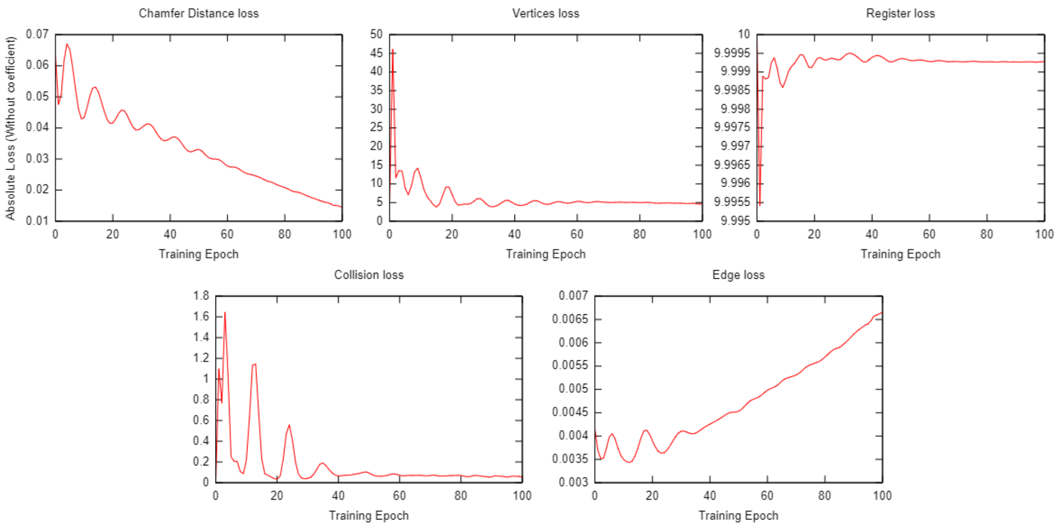


Figure 5.8: Losses values during all training epoch of the registration model with sample 00016.

At this point, looking at the generated results, we can see clearly that they have lots of artifacts, and they seem to be far away from the fully registered garments that we were expecting. However, I feel we should not be very disappointed, as I believe that by examining them we can extract much more insights about the complexity of this problem and what can be done to improve the results. Nevertheless, this model is still giving some correct results at certain parts of the body as we can see in Figure 5.9, so we should take notes on that and see which parts are failing and why.

Observing Figure 5.9, we can clearly see that the garment is following the body shape, and it is getting really close to it. We can see that even in some parts is getting inside the body, but we were expecting that and it is not bad for our results as we really want the position of the body and garment vertices to be practically the same, and that is why we have given a small coefficient to the collision loss for this model.

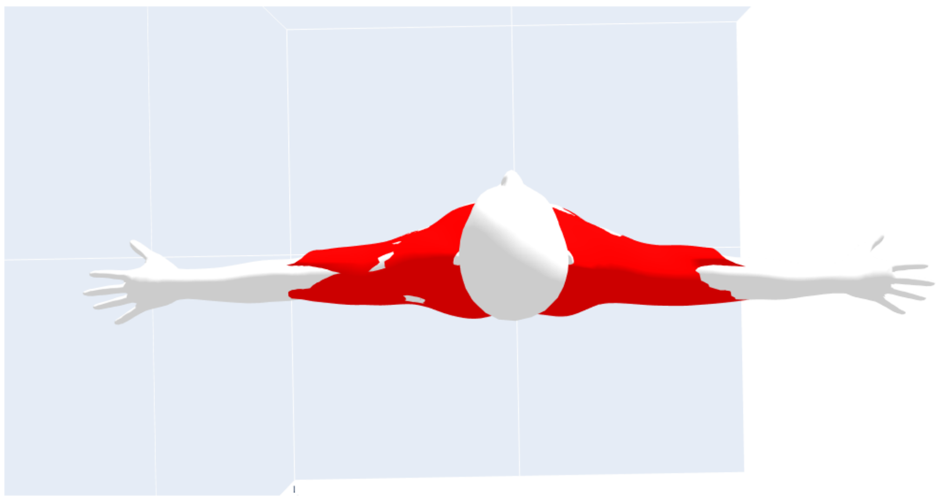


Figure 5.9: Output of the registration model with sample 00016, from a top close view.

However, even that in the middle parts of the body we see that the model may be registering well the outfit, we see that it clearly struggles in other parts as near the head, arms and legs. The registration here becomes more tricky, as the model needs to find a solution at points where there can be multiple vertices that minimize the chamfer distance loss. For instance, as we see clearly in sample 04846 5.7, some vertices that should be registered to the body, get registered to the head or arms. This is caused because at some moment of the training, the model has found that going closer to the head or arms was minimizing the total chamfer distance, and as we saw in the first model, the neuronal network models are very good and finding local minimums, but once they have found it, is difficult to make the model change and find the global minimum that would totally minimize the loss.

Similarly to what we saw in the previous model, in this case what we would want is to have a model that fully minimizes the chamfer distance loss. If the model was able to do that, we would have practically perfect results, as when the chamfer distance is very close to 0 that means that the vertices position is practically the same. However, what we have found in practice, is that we still need a strong support of the vertices position regularizer to obtain a coherent result. If we increase the chamfer collision loss too much without the help of the regularizers, we get very unstable results, and even that it could be very counterintuitive to give such importance to a loss that is only pushing us to stay with the original inputs, this is the only way to get a stable training. We can see the results when we increase too much the chamfer distance coefficient in Figure 5.10.

If we examine the output samples on detail, what we can see is that the model is clearly doing some registration, but it is clearly struggling to generalize for all the garment parts. As mentioned before, simpler garments parts get registered more or less in a coherent way, but when it has to face parts that initially were very far away from the garment or have more complex topologies as skirts or dresses, it is not able to register all the outfit in

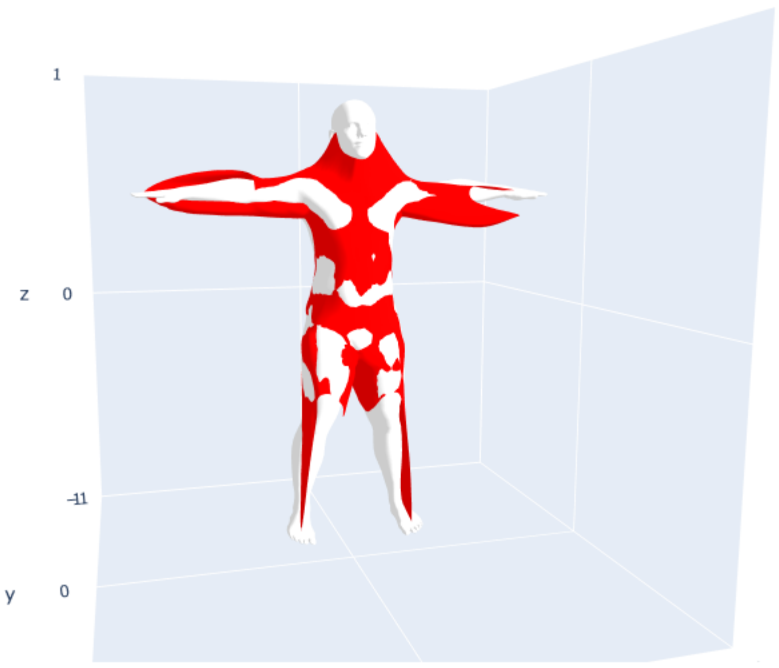


Figure 5.10: Output of the registration model with sample 00016, with chamfer distance loss coefficient set to 5.0e+4. We see that the model over stretches the garment to try to minimize the global chamfer distance.

the same way.

Then, we can conclude that the results are not correct, but they are a clear basis of what a machine learning based model could do to implement the registration.

From my point of view, the main chamfer distance loss is not enough to find and optimal solution to this problem, and I think other well thought losses could definitively help the model to solve the problem in a more correct way.

Finally, we should also be aware of that this model is an unsupervised based model. That means there is not any target output that we want to obtain, and indeed the final obtained losses are a direct consequence of how we have defined the losses and which coefficients we have assigned to each of them. So, the final obtained errors are not a very good way to judge how the model is doing, as in part we are creating these errors. That is why it is so difficult to tune the losses coefficients and to define the loss functions, as it is very hard to objectively judge how the model is doing automatically and without human supervision.

Chapter 6

Conclusions

6.1 Future Work

As we commented before, there is wide range of improvement of the obtained results, that could be implemented to obtain much more accurate outputs.

First, we could play with a more complex network shape. As we said, we used a very simple Graph Convolutional Network, as it was already returning some results that made us understand that it was able to learn the problem. However, including more layers, with higher number of features, could definitively help the models to solve the problem, even thought the training and computation times would increase.

In our case, also we were limited by the available computational resources, as using *Google Collab* we can only train the models for a reduced amount of time, and we could not train the model with the full dataset, to be able to generalize for all different samples, instead of training five different models, overfitting one particular sample.

Another possible improvement to the model, could be to introduce in some way the body shape to the registration network. In our cases, as we are only training with one sample at the time, the model can learn implicitly the body shape and position, but the reality is that the bodies shape vary a lot, and we could even think of using different poses and try to register them to train our models. That is why, I think that, if we want to train with the full CLOTH3D dataset, we must include in some way the body in the model inputs, as if we do not do that, the model cannot learn how to register an outfit to a body it is not knowledgeable about.

In addition, we saw that the registration model certainly struggles with skirts and dresses. This is caused because the topology of the body and garment is very different, in one case the garment has one overture, but the body has two legs. That problem was discussed during the project development, and the proposed solution is to *smooth* the body legs and create a join between the two legs to create a one leg body. This way, the registration results for this particular samples would be improved.

Also, as commented in the results sections, I feel like the registration model still needs other loss functions that help the model to register the garments, as I feel like it really struggles to minimize the chamfer distance, and the other losses are not able to create a big impact on the results.

Furthermore, I would like to state that in this particular case we have found that adjusting the model coefficients is a really hard task, as subtle changes on them could cause very unexpected behaviors. Maybe, we could define some techniques to dynamically adjust the coefficients while training, but I believe that this definition should be very well thought, as it is not an easy task because we have to progressively increase or reduce the coefficients but establishing some metrics to avoid getting chaotic results.

Finally, I think the problem is still open to new clever ideas, as for instance we could think of defining another model to judge how well the registering model is doing the registration, or generating some samples that are already registered with other methods and train the model comparing with them.

6.2 Conclusions

To conclude, I think that this project could establish a clear basis on how a Neuronal Network based model could be able to solve the registration model efficiently. But, at the same time, it also shows the high complexity of designing this type of solutions. While implementing the solutions, we had to constantly review the proposed initial designs and review them to fulfill the project objectives. From my point of view, that also shows that we are still at the tip of the iceberg of the machine learning knowledge, as we still have to figure out how to optimize or make faster the design of the models, as for instance, it is very hard to determine which model shape will be better to solve our problem. Even if we achieve so, I think that this also shows that humans are and will be completely necessary while designing machine learning based algorithms, because, even if we are trying to design models that learn by themselves, we are the ones who can define the problems and mechanisms to the models to be able to learn new and demanding problems.

From a personal perspective, I would like to add that I am very satisfied with the work done in this project. I had always been interested in Neuronal Networks, and even we have seen some basic concepts about them during the degree, this was my first opportunity to engineer a complex machine learning model and to actually implement it with widely used tools in the machine learning world. Definitively, this has given me a much more scientific approach to this kind of problems, and I believe it may be useful to me as it has given me some ideas of other models or problems that can be solved with this kind of approaches.

Bibliography

[1] Bertiche, H., Madadi, M. & Escalera, S. *CLOTH3D: Clothed 3D Humans.* (2019)
https://arxiv.org/abs/1912.02792

[2] Amberg, B., Romdhani, S. & Vetter, T. *Optimal Step Nonrigid ICP Algorithms for Surface Registration.* (2007)
https://gravis.dmi.unibas.ch/publications/2007/CVPR07_Amberg.pdf

[3] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K. & Hassabis, D. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.* (2017)
https://arxiv.org/abs/1712.01815

[4] ujjwalkarn *A Quick Introduction to Neural Networks.* (2016)
https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/

[5] Wikipedia *Backpropagation.* (2022)
https://en.wikipedia.org/wiki/Backpropagation

[6] Wikipedia *Kernel (image processing).* (2022)
https://en.wikipedia.org/wiki/Kernel_(image_processing)

[7] Wikipedia *Convolutional neural network .* (2022)
https://en.wikipedia.org/wiki/Convolutional_neural_network

[8] Kipf, T. & Welling, M. *Semi-Supervised Classification with Graph Convolutional Networks.* (2016)
https://arxiv.org/abs/1609.02907

[9] Mayachita, I. *Understanding Graph Convolutional Networks for Node Classification.* (2020)
towardsdatascience.com

[10] Weisstein, Eric W. from Wolfram MathWorld *Normal Vector .* (2022)
https://mathworld.wolfram.com/NormalVector.html

[11] Bertiche, H., Madadi, M., Tylson, E. & Escalera, S. *DeePSD: Automatic Deep Skinning And Pose Space Deformation For 3D Garment Animation.* (2020)
https://arxiv.org/abs/2009.02715

[12] Loper, M., Mahmood, N., Romero, J., Pons-Moll, G., Black, M. *SMPL: A Skinned Multi-Person Linear Model.* (2015)
https://files.is.tue.mpg.de/black/papers/SMPL2015.pdf