Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Kristian Stamland

3D Face tracking using Geometric Deep Learning

Master's thesis in Computer Science
Supervisor: Theoharis Theoharis
Co-supervisor: Antonios Danelakis
July 2021

NTNU
Kunnskap for en bedre verden

Kristian Stamland

3D Face tracking using Geometric Deep Learning

Master's thesis in Computer Science
Supervisor: Theoharis Theoharis
Co-supervisor: Antonios Danelakis
July 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

Face tracking has been an active field of study, especially with the advent of Deep learning and Convolutional Neural Networks. Most of the active research has been in the realm of 2D images and real time videos. This thesis looks into face tracking in the realm of 3D meshes and point clouds instead, using existing techniques from the field of Geometric deep learning.

We propose an application for the segmentation version of the PointNet++[16] network for the purpose of performing face tracking on 3D meshes. It adds a post-processing step that extracts a yes or no answer for face detection purposes, as well as extracting predicted face regions on the mesh for tracking purposes.

We use a mixture of the BU3DFE[21] facial emotion recognition dataset together with a proprietary dataset made with the FaceGen[7] software and the ModelNet10[19] dataset from Princeton. Our proposed solution achieves an Intersection over Union score of about 98% on the validation dataset, and a face detection rate of 99.7%. The solution struggles with achieving similar results on more complex inputs, e.g. full head meshes instead of face-only meshes.

Sammendrag

Fjessporing er et aktivt forskningsemne, spesielt etter at dyplæring og konvulsjons neurale nettverk kom på banen. Den aktive forskningen forholder seg for det meste til 2D bilder og videoer. Denne avhandlingen ser på fjessporing på 3D modeller og punkt skyer ved å bruke eksisterende Geometrisk Dyplærings teknikker.

Vi foreslår bruk av segmenteringsdelen av nettverket PointNet++[16] for fjessporing på 3D modeller. Vi har lagt til et etterbehandligssteg som gir et Ja eller Nei svar for ansiktsgjenkjennelse, samt uthenting av predikerte ansiktsregioner fra nettverket.

Det benyttes en blanding av BU3DFE[21] datasettet sammen med et proprietært datasett som er skapt av FaceGen[7] programvaren og ModelNet10[19] datasettet fra Princeton. Løsningen vår ender opp med en Snitt over Union score på 98%, og en ansiktsgjennkjennelses rate på 99.7%. Den sliter med å oppnå liknende resultater på mer kompleks data, for eksempel hodemodeller kontra ansiktsmodeller.

Acknowledgements

First of, I would like to give thanks to my supervisors, Antonios Danelakis and Theoharis Theoharis, for all the help they have provided over the course of writing this thesis. I would also like to give thanks to my family for supporting me through it all, as well as helping me lift the burdens of this past year. Finally I would like to thank my fellow students, both friends and acquaintances, for sharing what has been a wonderful experience with me.

Table of Contents

List of Figures

1 Introduction

Deep learning has been used for multiple different applications, from object recognition to segmentation. Most of these applications have been in lower dimensional domains, such as the two-dimensional image space. In recent years there have been an increased interest in trying to generalize deep learning methodology to higher-dimensional domains. This thesis looks into using some of these methods for the purpose of face-tracking on 3D-mesh data.

Face tracking in two-dimensions has seen a lot of interest over the years, and is usually one of the important parts in modern face-recognition algorithms. Before convolutional neural networks were a thing, face tracking was done using mathematical functions[17]. However, with the advent of deep learning and convolutional neural networks, we have seen multiple papers[1][18] utilizing these methods to quickly and accurately find faces in images.

As face-tracking on 3D-mesh data is a topic in its infancy, there exists little prior work on the topic. There are, however, many papers looking into the field of Geometric Deep Learning. A paper[3] proposes using spectral graph theory to define a convolution on graphs. The method uses graph Fourier transforms for filtering in the signal domain instead of the vertex domain. It also suggests a pooling type called Graph Coursing, where courser and courser levels of a graph are constructed. This is used in tandem with a pooling layer on graph signals, more detail is given in the Background section. Another method[12] suggests using local geodesic systems of polar coordinates to extract patches from non-euclidean manifolds, which are then passed through filters and linear and non-linear operations.

PointNet is a neural network that takes a point cloud as input and performs either object classification or shape segmentation[15]. This network was further revised afterwards to improve the usage of local structures in the model, allowing for better results on shape segmentation[16]. We theorize that using the shape segmentation network, we can perform both face detection and face tracking by using a binary class dataset for training the network to recognize points as either being included in a face, or excluded. This is then used as the input for a post-processing step that extracts regions from the input that are defined as faces. It also returns a Yes or No answer based on whether or not a region was found to indicate if a face has been detected.

2 Background

2.1 Deep Learning

Deep learning is a sub-set of machine learning that takes inspiration from the biological structure of the brain. The main concept behind machine learning is training a set of weights based on some data input. This helps us achieve self-learning algorithms that can be trained to perform specific tasks. Deep learning bases it self around creating neural networks, which are multi-layered algorithms where each layer feeds into the next. The layers each represents some form of algorithms, such as simple addition, convolutions, with more. This type of network is called a Multi-Layer perceptron.

2.1.1 Loss function

A loss function gives us a score we can use during training of the network. The function returns some value based on the distance between ground-truth and the networks result. Another way of looking at it is the loss function describes the error of the network. One of the simplest loss functions is simply subtracting the output from ground truth.

$$loss(x) = y - x \tag{1}$$

where y is equal to ground truth.

The negative log likelihood (NLL) loss function is a useful loss function on classification networks. It needs the last layer of the network to be a softmax layer. Softmax is a function that forces all classes onto the range [0,1], while also making sure they sum up to 1. The equation for Softmax, where i stands for the given class, the below sum is the sum of all classes, is:

$$f_i(x) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{2}$$

This is useful, as it gives us a probability score for each class, allowing us to make predictions on which class is most likely represented. During training, the probability score of the correct class is sent in to the negative log likelihood loss function, giving us our loss. The reason why NLL is useful when training classification networks, is because of the output space of the function. As the probability goes towards 1, the loss moves towards 0, while when the probability score moves towards 0, the loss tends towards infinity[13].

$$NLL(y) = -log(y) \tag{3}$$

2.1.2 Backpropagation

The training of a model is essentially split into two parts. The first part is the forward pass, which is a normal run through the network. The result from the forward pass is then run through the loss function, giving us a loss value we can use to update the network. We take the loss value, and calculate new weights based for the network. This is done by propagating the loss value back-to-front through the network.

2.1.3 Optimizer

The goal is to find a global minima for the loss value, or at least a local minima. This gives us an optimization problem. Backpropagation uses gradients when updating the weights. Each weight gets updated based on the partial derivative of the gradient defined using the loss value. The part of the training process that calculates these gradients is called an optimizer.

Many optimizers exists, one of the first ones being called gradient descent. Gradient descent is, in its most simple form, the derivative of the loss function. We subtract the gradient from the current weight values, giving us a new value for the weight. However, if we subtract the full gradient from the weight, we might end up overstepping the minima we hope to find. That is where the hyper-parameter called learning rate comes in. We multiply the gradient with the learning rate, and then subtract it from the old weight value. The important part when picking the learning rate is to try to pick a small enough value so that we do not overstep the minima, while having a large enough value so that the time it takes for the network to converge is as quick as possible.

Adam is an optimizer that updates continuously over the training session. Unlike gradient descent, Adam has learning rates on a per-weight basis, as well as a dynamic learning rate instead of it being static [8]. The learning rate is calculated based on the first-degree, i.e. the mean of the gradient, and second-degree moment, i.e. the uncentered variance of the gradient.

2.1.4 Activation Function

As deep learning methods are based in linear algebra, we need a way to represent non-linear functions. This is where the idea of an activation function comes in. An activation function decides whether or not the neuron should feed its result forward in the network or not. The function maps the result from a linear function through a non-linear function. The sigmoid function, for example, maps the output of a neuron to the range [0,1]. As you can see, the sigmoid function clamps the top and bottom end of the neuron output to either 1 or 0, while results near the center are mapped gradually from 0 to 1.

$$S(x) = \frac{1}{1 - e^{-x}} \tag{4}$$

Another widely used activation function is the ReLU activation function. ReLU only allows positive results through. One of the benefits of using ReLU over Sigmoid is the difference in computation cost, as ReLU is only a single max operation compared to the mathematical operations of the Sigmoid function.

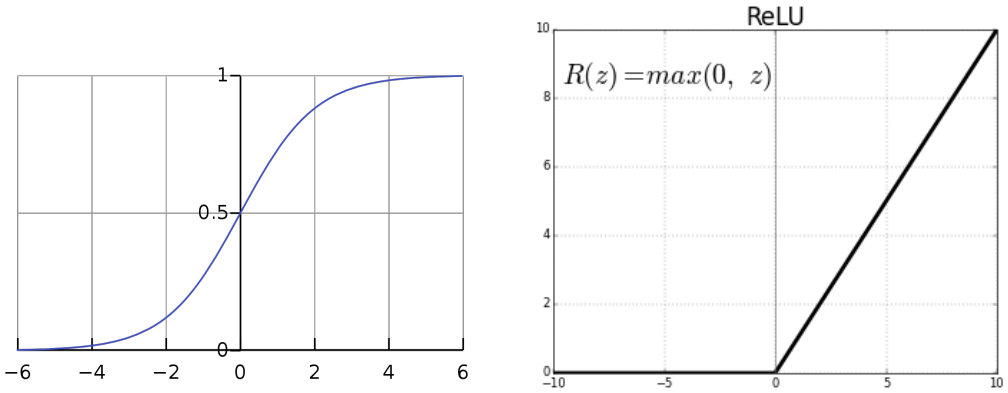


Figure 1: The plotted sigmoid function (left) and ReLU function (right)

$$ReLU(x) = max(0, x) \tag{5}$$

Other differences between ReLU and Sigmoid is that with ReLU, you do not have vanishing gradients for values above 0, as the gradients for high output values tend towards zero for Sigmoid, while being unaffected on ReLU. Sigmoid has the advantage, however, of preventing blow up of activation values, as the output is restricted to the range [0,1] while ReLU is unrestricted for values above 0[5].

2.1.5 Convolution layers

A convolution is defined as a mathematical operation that describes how one function modifies another. The input for a convolution is usually two functions, and the output is a new third function. In deep learning, the input for a convolution has generally been a multi-dimensional array of data, generally referred to as a Tensor, and a second multi-dimensional array of weights, referred to as a Kernel. A convolution over a 2d-image can be defined as:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{6}$$

The result of a convolution layer is often called a feature map, as it shows where the kernel resulted in an activation, giving us a clue as to what features the kernel is filtering for.

2.1.6 Pooling

Pooling layers are layers that perform some sort of aggregating function on the input. This helps with down-sizing the inputs for later layers, and can help with giving us a more general feature map. This is due the aggregated result being of a lower resolution compared to the original input.

The two most used pooling layers are the Max pool and the Average pool. Max pooling returns the maximum value inside the given pooling size. For instance, if we have a pool size of 2, and the values inside a 2x2 part of an image are 10, -2, 3 and 4, the result from the max pool operation will be 10. Similarly, average pooling returns the average value within the average pool filter. Using the same example, we get 3.75 as the returned value of the average pooling. Doing pooling with a filter size of 2 will effectively half the resolution of the input, meaning if we have a 512x512 image, and run it through the pooling layer, we end up with a 256x256 image.

One of the nice things about using convolution layers in conjunction with pooling layers is that when we get to the fully-connected part of the network, we have reduced the original resolution, making it less costly to do the computation of the fully-connected layers.

2.1.7 Training

Training of neural networks use most of the parts described above. The loss function and optimizers are for example parts mostly used during training. It is during training that the network model is made to fit a given problem. As mentioned earlier, this is where we try to minimize the loss value of the network. Training takes time, but there are things we can try to utilize to minimize the time investment. For example, since most neural networks are an application of linear algebra, we can perform the training on more than one data at the time. For instance, if we were training a model on object detection on images, we can send multiple images through the network at once. This is called batching.

We also need to have a predefined dataset we can do the training on. The dataset will include a bunch of different data, the more data we have and the more varied the data is will help with specializing the network. We train the model on the training set multiple times. When we send data through the network, we call it an iteration of training. Iterations are usually done on batches, where the size of the batches is decided before training starts. The higher the batch size, the faster the training will be. The limiting factor here is the memory needed for higher batches. When the network has finished iterating through the dataset, we have finished what is called an epoch. We generally train the network over multiple epochs. The number of epochs a network needs to converge can vary from network to network.

2.1.8 Validation

Training is not the only thing we need to do with the network. We also need to ensure that the network gives good results when non-training data is run through it. This is where the validation step comes in. First we need to define a validation data set. This data set is similar to the training data set, as it contains some data that has a pre-defined result attached to it. The validation step is very similar to the training step, as we iterate through the data set. The difference is that during validation, we don't update the weights of the network.

This thesis performs validation after each training epoch, but it is possible to do validation after any given amount of epochs, to potentially speed up training.

2.2 Geometric Deep Learning

The current state of the art deep learning methods work mostly on lower dimensional data, i.e. 1-dimensional or 2-dimensional. Geometric deep learning is an attempt to generalize deep learning to higher-dimensional data, such as 3D meshes or graphs.

The term Geometric Deep Learning was first introduced in a paper[2] describing the concept, and summarizing multiple papers on the topic. The paper is continuously updated to include summaries of new papers.

2.2.1 Graph Convolutions

There have been several proposed solutions for generalized graph convolutions. One paper suggests using something called a Chebychev spectral graph convolutional operation[3]. The operation uses spectral graph theory and introduces the idea of using Graph Fourier Transformations, performing convolutions in the Signal domain instead of the vertex domain.

We first define a graph as $G = (V, \xi, W)$, where V is a finite set of n vertices, ξ is a set of edges and $W \in R^{n \times n}$ is a weighed adjacency matrix encoding the connection weight between two vertices. A signal $x : V \longrightarrow R$ defined on the nodes of the graph can be regarded as a vector in R^n, where x_i is the value of x at the i^{th} node. The graph Laplacian is defined as $L = D - W \in R^{n \times n}$ where $D \in R^{n \times n}$ is the diagonal degree matrix with $D_{ii} = \sum_j W_{ij}$. L has a complete set of orthonormal eigenvectors $\{u_l\}_{l=0}^{n-1}$, called the graph Fourier modes, and their associative set of real non-negative eigenvalues $\lambda_{l=0}^{n-1}$. From this, we can define the Fourier base $U = [u_0, ..., u_{n-1}] \in R^{n \times n}$. We can see then that the Laplacian is diagonalized by U, i.e. $L = U^T \Lambda U$, where $\Lambda = diag([\lambda_0, ..., \lambda_{n-1}])$ [3]. The fourier transform of a signal x is so defined: $\hat{x} = U^T x$, and the inverse fourier is defined as $x = U\hat{x}$

From this, we define the convolution operation $*_G$ as $x *_G y = U((U^T x) \odot (U^T y))$. $\odot$ is defined as the element-wise Hadamard product[3, p.03]. Signal x is therefore

filtered by filter g_θ as:

$$y = g_\theta(L)x = g_\theta(U\Lambda U^T)x = Ug_\theta(\Lambda)U^T x \tag{7}$$

The paper goes into greater detail on how the filters are parameterized, and how learning is done on the filters. It also goes into detail on how to perform the convolution operation in parallel[3, p.04].

2.2.2 Pooling

Pooling is an important part of convolutional neural networks as it aggregates the results from the convolutions, allowing us to create aggregated feature maps of different resolutions. This is useful, because it allows us to capture both global and local structures on the data. The graph convolution network[3] provides a concept for pooling, combining the idea of Graph coarsing and pooling on graph signals.

The graph coarsing step uses the coarsing phase of the Graclus multilevel clustering algorithm[4]. The algorithm computes successive coarser versions of a given graph, and is able to minimize several popular spectral coarsing objectives, which we can choose the normalized cut from. At each coarsing level, we pick an unmarked vertex i and match it with its unmarked neighbors j that maximizes the local normalized cut $W_{ij}(1/d_i + 1/d_j)$. The two matched vertices are then marked, the coarsened weights are set as the sum of their weights. This matching is performed until all points have been iterated over[3, p.04]. This algorithm can be done fairly quickly, and generally ends up dividing the number of nodes by approximately 2.

After the graph coarsening, the coarsened version and the input graph are not arranged in any meaningful way. A direct application of the pooling operation would therefore have to store the matched vertices in a table, which would be fairly memory inefficient. The paper suggests arranging the vertices in such a way such that the pooling operation is similar to a 1D pooling operation in efficiency. The first step is to create a balanced binary tree, and the second step is to rearrange the vertices. After coarsening, each node has either two children, or one, i.e. a singleton, based on if it was matched on the finest level or not[3, p.04]. Nodes that have been disconnected during the coarsening steps, are added as children to nodes that only have one child such that each node has two children. This is a balanced tree because each node is either a regular node with two regular node, or with one singleton and one detached node, or a detached node with two detached node as children. Input signals are initialized with e neutral value on the detached nodes, e.g. 0 when using ReLU and max pooling. Ordering the nodes arbitrarily on the coarsest level, and propagating this order to the finest level, produces a regular ordering in the finest level. This means that the adjacent nodes are hierarchically merged on the coarser levels. Performing pooling on the rearranged graph signal will be analog to pooling a 1D signal[3, p.04-05].

3 Related work

3D face tracking on 3D-meshes is a relatively new field of study, and as such no real state of the art exists. However, face tracking in general has seen quite a bit of progression over the years. Some of the challenges faced by 2D face tracking systems are pose variation, viewpoint variation, occlusion and illumination variation [17, Challenges in Face Tracking].

A comparative analysis of various face recognition and tracking systems[18] from 2021 presents a comprehensive history of various face tracking networks. Deep multi-pose[1] sends a face image through multiple pose-specific deep CNNs to generate multi-pose features. NAN[20] proposes using a Neural Aggregation Network for face recognition. While this is not face tracking, face tracking is part of the issue with face recognition.

Lian, Z. et. al[9] proposes a system for real time multiple face tracking using a multiple object tracking algorithm. The paper uses Multi-task Convolution Neural Network (MTCNN) is used to detect faces. Multiple features are added on top of this to help with occlusion and rapid object movements, such as appearance features and motion features.

PointNet[15] introduced a network that performs either shape classification or shape segmentation on Point clouds. This network was further built upon in the paper PointNet++[16] which proposes a new network that performs multiple iterations of the original PointNet on progressively more aggregated point sets. We think that the segmentation part can be used to perform face tracking on 3D meshes by using the vertices as a point cloud.

3.1 PointNet

PointNet is a neural network that takes a point cloud as its input, and then performs either object classification or part segmentation. Previous networks on point clouds or mesh data used to either change it to a regular 3D grid of voxels, or a collection of 2D images before sending it through the network. The problem with these transformations is that it makes the data larger, and can introduce artifacts that might lead to obscuring of natural occurring invariances in the data.

The crux of PointNet is the use of max pooling. Max pooling gives the network interesting points, allowing it to train and learn global features. In essence, the network learns how to pick out interesting points, and the reasoning for their selection. These points are then sent to the final fully-connected layer, creating the global descriptor for either the entire shape (object classification) or is used to create per-point labels (shape segmentation). [15, p.01]

3.1.1 Properties of Point Sets (point clouds)

The PointNet paper highlights three properties about the point cloud, or the point set, input of the network. The first property is that the set of points is unordered. Unlike 2D arrays of data or volumetric grids, the point set is just an unordered collection of points. This means that the network needs to be invariant to N! permutations for a 3D point set of size N.

The second property the paper highlights is the interaction between points. Each point has a distance metric for each other point, and we can therefore utilize the neighboring points to create meaningful subsets of points.

Finally, the paper highlights the need to be invariant to transformations on the point set. The learned representation of the point set should be invariant to certain transformations. [15, p.03]

3.1.2 Network Architecture

The network architecture builds around three key modules. The first module is the max pooling operation. There are mainly three ways to deal with the solving the problem of creating a model that is invariant to the input permutation. The first solution is to order the input. The second is to use the input as a sequence to train an RNN, augmenting the training data by all kinds of permutations. The third way is to use a symmetric function to aggregate information from the points. Max pooling has the nice property of being symmetrical, meaning no matter how you order the input, you will get the same output, and the output is an aggregate of the input. [15, p.03]

Now the next key module is the global information aggregation, which is the layers following the max-pooling layer. These global features can be connected to a MLP for global shape classification, which is what forms the Shape Classification part of the network. However, in the case of shape segmentation, the global features are fed forward in the network to after the local point features have been extracted.

The third key module is the local information aggregation. The network creates local point features for each point. It then combines these local features with the global features extracted, and uses both of these features to predict per-point quantities that rely on local features and global semantics. [15, p.04]

Figure 2 shows the overall structure of the PointNet network. The first part of the network, the T-Net, transforms the input in such a way that it aligns the input to a canonical space before feature extraction. This helps making the network invariant to rigid transformations and other various geometric transformations. [15, p.04]. This is then followed by the first key module mentioned, the max pooling, then the second key module, global information aggregation, and then finally the local information aggregation and classifications.

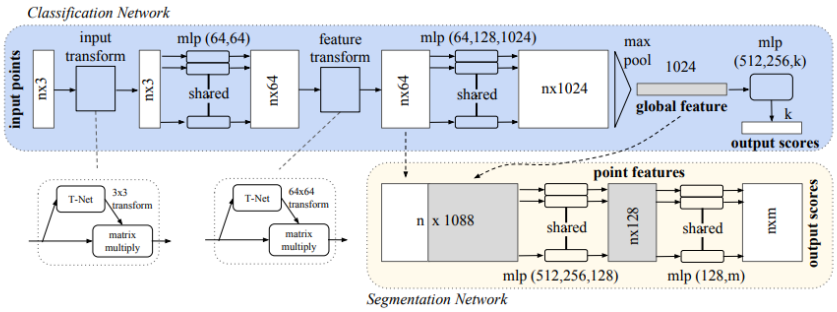


Figure 2: PointNet structure [15]

3.2 PointNet++

The authors of PointNet released a new paper the following year, called PointNet++. PointNet does not capture the local structures that are induced by the metric space the input points live in. This new network attempts to solve it by recursively applying PointNet on nested partitions of the input set. The two issues that PointNet++ addresses to perform this are how to generate the partitioning of the input, and how to abstract sets of points or local features via a local feature learner. The partitioning has to produce common structures across partitions so that the local feature learners can share their weights. PointNet is used as the local feature learner in PointNet++[16, p.01-02].

3.2.1 Network Architecture

The network structure of PointNet++, as seen in Figure 3, shows that the local feature learner, PointNet, runs after sampling and grouping. These layers serve a similar purpose as the Max Pool layer in the original PointNet, however the aggregation happens in steps, moving towards lower-resolutions. Figure 3 shows that we perform two of these sampling and grouping stages, giving us two aggregation steps.

3.2.2 Hierarchical Point Set Feature Learning

The original PointNet uses a single max pooling layer that aggregates the whole input set, while the new model introduces a new set of layers that perform a set abstraction instead. This set abstraction layer is made of three key components: A sampling layer, a Grouping layer and a PointNet layer[16, p.03].

The sampling layer selects a set of points from the input. It first uses iterative farthest point sampling (FPS) to get a subset of points where the next point in the set is the farthest point from the points before it in the set. I.e. X_{i_j} from the set $\{X_{i_1}, X_{i_2}, ..., X_{i_m}\}$ is the farthest point from the points in the set $\{X_{i_1}, X_{i_2}, ..., X_{i_{j-1}}\}$.

The grouping layer takes in a point set of size $Nx(d+C)$, where N stands for number of points in d-dimension and with C-dimension features. It also takes in a set of centroids of size $N'xd$. These inputs are then turned into groups containing new

sets of size $N'xKx(d+C)$, where K stands for the number of neighbouring points to the centroid points. K varies across groups. Neighbouring points in the metric space are defined by their metric distances.

The layer uses Ball query, which finds all points within a radius of the given point. An upper limit of K is set during implementation.

The PointNet layer consumes the input from the grouping layer. Before the groups are given to the PointNet layer, the points in the local region are translated to a local frame of reference relative to the centroid point. This allows the network to capture point-to-point relations in the local region[16, p.03-04].

3.2.3 Robust Feature Learning under Non-Uniform Sampling Density

It is common for point sets to include non-uniform density in different areas. The features learned in dense areas might not generalize to more sparsely sampled regions. Models trained on sparse point clouds might not recognize more fine-grained structures.

To combat this issue, PointNet++ proposes density adaptive PointNet layers that learn to combine features from regions of different scales when the density of the input samples changes. The paper proposes two density adaptive layers.

The first type of density adaptive layer is called **Multi-scale grouping** (MSG). This layer applies grouping layers with different scales, followed by similar PointNets to extract features from each scale. The different sized features are then concatenated to form a multi-scale feature[16, p.04].

The second type is called **Multi-resolution grouping** (MRG). The MSG approach is computationally expensive, since it runs local PointNets on large scale neighborhoods for each centroid point. This gets worse on denser sampled inputs.

In MRG, features of a region at some level L_i is a concatination of two vectors. The first vector is obtained by summarizing features at each subreion in the level below using the set abstraction level. The other vector is the feature obtained from processing all the raw points of the local region using a single PointNet.

When the density of a local region is low, the second vector should have a higher weight than the first, because the features from the sub-regions will suffer from the sub-regions being performed on even sparser points leading to sampling deficiency. The first vector will, however, provide information of finer details when the density of the local region is high, as it can inspect the points in higher resolutions recursively on lower levels.

3.2.4 Point Feature Propagation for Set Segmentation

Since the network aggregates data into a lower resolution, we need a way to get back to the original set for segmentation purposes. This is where the idea of k-Nearest-

Neighbor interpolation comes in[16, p.05]. We propagate features from sub-sampled points onto the original points.

This is done by propagating point features from $N_l x(d + C)$ points to N_{l-1} where N_{l-1} and N_l, where $N_l \leq N_{l-1}$, represent point set sizes of the input and outputs of set abstraction level l. The formula uses inverse distance weighted averages for interpolation based on k nearest neighbors. The interpolated features on the new point set are then concatenated with skip linked point features from the set abstraction level. The concatenated features run through what the paper calls a unit pointnet, similar to a one-by-one convolution in regular CNNs. This then runs through a few shared fully-connected layers and ReLU layers to update each point's feature vector. This process is repeated until we reach the original set of points[16, p.05].

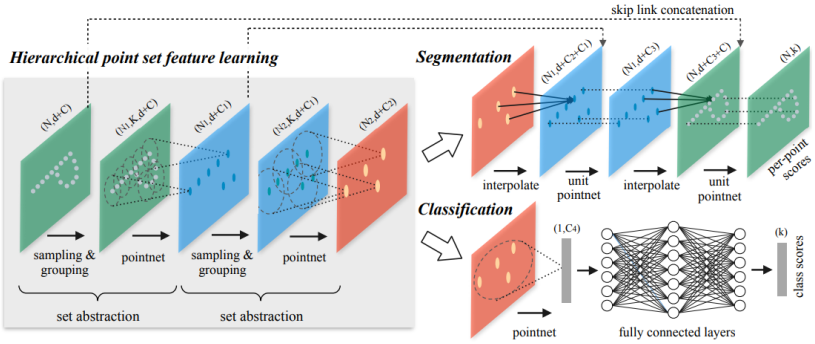


Figure 3: PointNet++ visualization [16]

4 Methodology

4.1 Tools

4.1.1 Pytorch

PyTorch[14] is a rich python library that contains a multitude of tools used for machine learning applications. The library is open-source and maintained by FaceBook. It is built to be quick to iterate with, and the code is made to be Pythonic. PyTorch supports running models either on CPUs or GPUs using CUDA, allowing for high performance computation on Single-Instruction Multi-Data stream processors to speed up training and evaluation times.

This thesis creates a model using Pytorch, with some of the more important parts being the Adam optimizer, and the built-in loss functions.

4.1.2 PyTorch Geometric

The standard library included with PyTorch contains many useful pre-made functions, but is more generalized and useful towards deep learning compared to geometric deep learning. This is where PyTorch Geometric[6] comes in handy.

PyG contains implementations of multiple different geometric deep learning papers. It is an extension to PyTorch, and uses the same structure for the network. Most importantly, the PyG library includes the original PointNet as a pre-built module.

This means that we can create an implementation of the PointNet++ network, without having to build the original PointNet structure also from scratch. The Pytorch Geometric project page also includes some examples, and we use some of these to build our network.

4.2 PointNet++

This thesis uses the PointNet++ network for per-point prediction. The examples included in Pytorch Geometric contains an implementation of PointNet++, one example of the classification network and another with the segmentation network. We build our implementation on top of this example implementation. The main differences that this thesis implements is the usage of a binary class dataset, and some changes to how the validation is performed.

The various modules of PointNet++ are created using regular python classes. The set aggregation layer, represented by the SAModule class, uses the built-in implementations of the PointNet convolution, called PointConv, and Pytorch geometric's implementation of both the Iterative Farthest Point sampling, fps, and Ball query, called radius. This is followed by the Global Set Aggregation module. The module takes the groups from the last set abstraction layer, and performs global max pool

to aggregate the samples.

The FPModule class performs knn-interpolation, taking the results from the previous set abstraction levels. The first FPModule accepts both the output from the global SAModule, and the output from the second set abstraction level. The next FPModule takes the output from the previous FPModule, and the output from the first set abstraction level. The final FPmodule takes the output from the second FPModule and the original point set input from the network. This is then run through a couple of fully-connected layers before ending with a LogSoftmax layer for predictions. LogSoftmax is used instead of regular Softmax because we are training the model using the NLL loss function. The LogSoftmax operation performs the log operation together with the softmax operation, so the returned value would be $log(Softmax(x))$

4.3 Post-processing step for face detection and tracking

PointNet gives us a per-point score, classifying the point as either "Face" or "Not Face". This result in and of itself is not the actual face detection or tracking result, so therefore we need to perform a post-processing step to extract our detection and tracking result.

This thesis proposes using a simple algorithm that creates face regions based on positive points and their neighbours, pseudo-code in python shown in Figure 4. The input for the algorithm is the set of per-point predictions, i.e. the class with the highest score per-point, the original list of points and a connection set. The connection set is a python dictionary where point-indices are used as keys, and an iterable set of indices representing each point connected to the key are used as a value.

The algorithm iterates over each point, keeping a list of points visited so as to not re-do computations. It checks whether or not the point is classified as "Face" or "NotFace". If the point is classified as "NotFace", the algorithm moves on to the next point. Otherwise, an expansion queue is created, which is a list of point-indices. We also create a region set, representing the vertices connected to the currently tracked face. From the connection set, the expansion queue is populated with the points connected to the current index. We then iterate over the expansion queue, removing the first point in the queue, adding it to the points visited set. We check if the removed point is classified as a "Face" point, if it is, we add all its connections to the expansion queue, and add the point to the current face-region. Once the expansion queue is empty, we check whether or not the number of points in the region set exceeds a predefined threshold to be classified as a face. If it classifies as a face region, we save it to a set of face regions.

When the algorithm has iterated over all the points in the input, we check if there were any regions large enough to be classified as a face. This is how the network does face detection. The face tracking result is the set of face regions.

```python
def containsFace(predictions, connections, pos,
    region_threshold=40):
    points_visited = set()
    face_regions = []
    for i, point in enumerate(pos):
        if i in points_visited:
            continue
        points_visited.add(i)
        region = []
        if predictions[i] == "NotFace":
            continue
        region.append(i)
        expansion_queue = []
        expansion_queue.extend(connections[i])
        while expansion_queue:
            expansion_point = expansion_queue.pop()
            if expansion_point in points_visited:
                continue
            if predictions[expansion_point] == "Face":
                region.append(expansion_point)
                expansion_queue.extend(connections[expansion_point])
            points_visited.add(expansion_point)
        if len(region) > region_threshold:
            face_regions.append(region)
    return len(face_regions) > 0, face_regions
```

Figure 4: Pseudo python code for face detection and tracking post processing algorithm

4.4 Network Training

The network is trained using the Adam optimizer, as described in the Background section. We use the Negative Log Likelihood loss function, as the network is a binary classification network. The network supports training on various batch sizes, and we discuss in the Results section how we trained the model multiple times using differing batch sizes.

Each training epoch runs iterates over the training data in a random order each time, in an attempt to increase randomness in a hope to combat overfitting. After each epoch, we perform validation testing on the validation dataset, and compute the IoU for the network on the validation set. We also create a checkpoint for the network, so that training can be split into multiple sessions in the event where the system resources are needed for other tasks.

The network is trained on a system with a Ryzen 5 2600x 3.5ghz CPU, 32 GB of RAM and a Nvidia RTX 3070 8GB GPU.

4.5 Metrics

There are many metrics that are used to evaluate object detection systems. Some of the more common ones are mean Average Precision, Intersection over Union and F1 score. Intersection over Union (IoU) in classic 2D object detection is described as the area of the intersection over the area of the union, but as this thesis works with 3D input, and not 2D images, we use the definition from set theory. Intersection of two sets equals the number of elements that have the same value at a given position. $A \cap B = \{a, c\}$ given set A being defined as $\{a, b, c\}$ and set B being defined as $\{a, d, c, e\}$. The union of these two sets would be $A \cap B = \{a, b, c, d, e\}$. If we take the intersection over union for this example, we get:

$$IoU = \frac{|\{a, b\}|}{|\{a, b, c, d, e\}|} = \frac{2}{5} = 0.4 \tag{8}$$

resulting in an IoU of 40%. This metric is useful, as it tells us how many of our predicted points are in the original set. With this metric, we can make some assumptions as to the tracking performance of the network. If we have a high IoU, that indicates that the network has done a good job predicting the per-vertices category on the inputs, but if the IoU is low, it indicates that our per-vertices predictions are largely incorrect.

The second metric we use, is face detection rate. This is a simple metric where we perform face detection on the verification set, and sum up the number of correct results. We then divide the number of correct results with the total number of inputs in the dataset, and we end up with a percentage score for detection rate.

4.6 Datasets

There exits no real dataset that is built explicit for the task this thesis wishes to address. As such, multiple other datasets have been used in conjunction to create a diverse data set that the network can train on.

4.6.1 BU3DFE

BU3DFE is a dataset containing 100 different face identities. Each face identity contains multiple expressions, leading to a total of 2500 face models[21].Each face mesh consists of between 3000 and 5000 vertices, The dataset is primarily used for 3D face emotion recognition, and each identity contains various face poses representing differing emotions. The dataset comprises of 60 female identities and 40 male, giving a 60/40 distribution over the genders.

As this thesis is about face tracking, and not emotion recognition, the face meshes are instead used as is. Each vertex of the meshes gets classified as a "Face" point. The network consumes point sets, as such only the vertices are used, and edge data is only used in the post-processing face-detection step. We split the dataset in to

two parts, using a split of 80/20. we use 80 identities for the training dataset, while the remaining 20 identities are used for validation testing. The test set contains 10 identities from each gender, i.e. a 50/50 split compared to the datasets 60/40 split.

4.6.2 FaceGen_1k

FaceGen1k is a proprietary dataset that contains 1000 face models that have been generated using the FaceGen software[7]. The models in this dataset are all face models with 5094 vertices, and are randomly generated using the FaceGen Artist software.

The face models have all their vertices labeled as "Face" during the initial set up of the network. We split the dataset in to two parts, the first part consisting of the first 800 faces, the second part consisting of the last 200 faces. The first part is used for training the model, while the second part is used for validation testing.

4.6.3 ModelNet10

ModelNet is a dataset from the university of Princeton[19]. The goal for the dataset is to provide 3D models for various research fields, like computer vision and computer graphics. It contains various categories of models, and each category is split into a training set and a test set. The dataset was first introduce in the paper for the 3D ShapeNets network[19].

The dataset comes in three varieties, ModelNet10, ModelNet40 and ModelNet. The first two varieties are each a subset of the full ModelNet set. This thesis utilizes the ModelNet10 variety, as the main focus for the proposed system is face tracking. ModelNet10 provides the network with a variety of models that do not contain faces, and therefore should not return any face points. During the dataset setup, each vertices from the ModelNet models gets labeled as "NotFace".

4.6.4 FaceGen_100

FaceGen100 is another generated dataset using FaceGen[7]. This set uses generated head models, instead of face models. The dataset is used to check the models ability to track faces, not detect them. However, the dataset has not been fully marked, so no real metrics have been used with it. The resulting face regions have instead been saved and then been imported for use in Blender with a custom script that selects all vertices of a given region. Appendix A goes into further detail.

5 Results

5.1 Preliminary Results

The network was first trained on a dataset using the BU3DFE faces, FaceGen_1k and ModelNet10. 80% of the BU3DFE set was used for training, with 20% reserved for validation testing, while FaceGen_1k was split 50/50 for training and validation. As stated in Section 4.6.3, the ModelNet10 dataset is already split into test sets and training sets. Only the Bed and Desk sets from Section 4.6.3 were used in this initial model.

Figure 5 shows us that the network quickly moves towards low loss values while training, implying that the model is moving towards convergence. The network was trained for 60 epochs in an attempt to find out how many epochs it would need before reaching convergence.

While the network achieves a high IoU score, reaching the high nineties on the validation test set as seen in Figure 6, further testing on face models from FaceGen100, Section 4.6.4, returns unimpressive results. As FaceGen100 contains full head models, the expected result would be to only get face points on parts of the model, namely the face. However, running one of these meshes through the trained network returns a full set of face points. As this is not a validation set, there is no statistic for the IoU on these meshes.

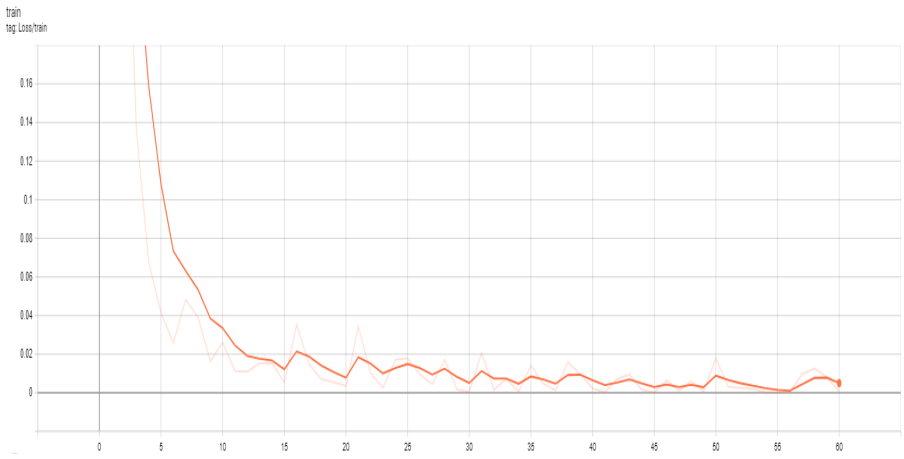


Figure 5: Train loss over time (First trained model)

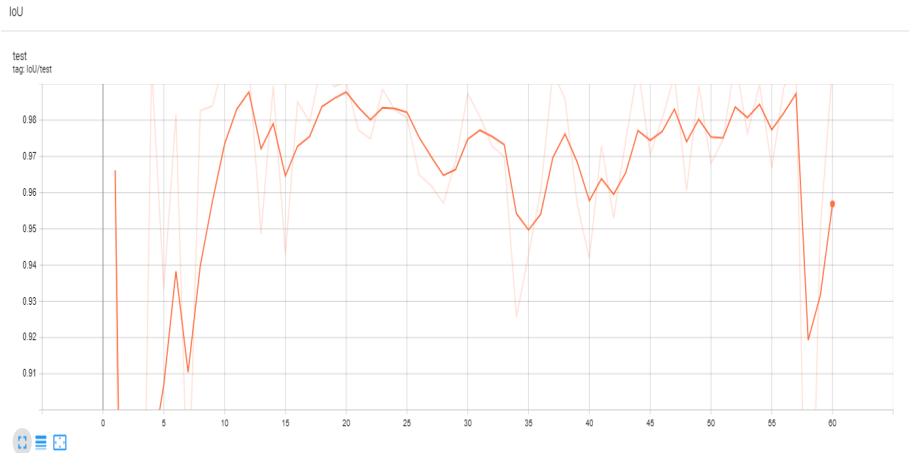


Figure 6: IoU value of Validation Testing over time (First trained model)

5.2 Face detection results

None of the models have been trained with face detection as an optimization problem. The per-point scores used for face tracking has been used for training instead. However, the post-processing algorithm does provide a Yes or No answer on whether or not a point cloud is predicted to contain a face or not. As such, we ran validation set through the post-processing step, looking at the number of correct guesses versus the number of point clouds in the validation set.

The first trained model ends up having a face detection accuracy of 99.6% on the validation set. The second trained model has a face detection accuracy of 91.9% on the validation set. The third trained model has a face detection accuracy of 99.7% on the validation set.

5.3 Face tracking results

Due to the preliminary results, we added the rest of ModelNet10 to the dataset and started training the network again. This time we trained it for 190 epochs instead of 60. The batch size used for the training of these networks were 4. Unlike the first network, we saw initially worse results on the validation set. However, after around 100 epochs we started to see better results, as seen in Figure 7. After 190 epochs, the IoU looked good at around 95%. We did the same validation testing as we did with the first model, looking at the tracking results on meshes from the FaceGen100 dataset. Where the first network marked everything as face points, this network marked everything as "NotFace" points. Figure 8 shows the loss value over time for the second trained network. The runs from epoch 70 and up ran with a batch size of 6, instead of 4, causing something of a reset on the loss value.

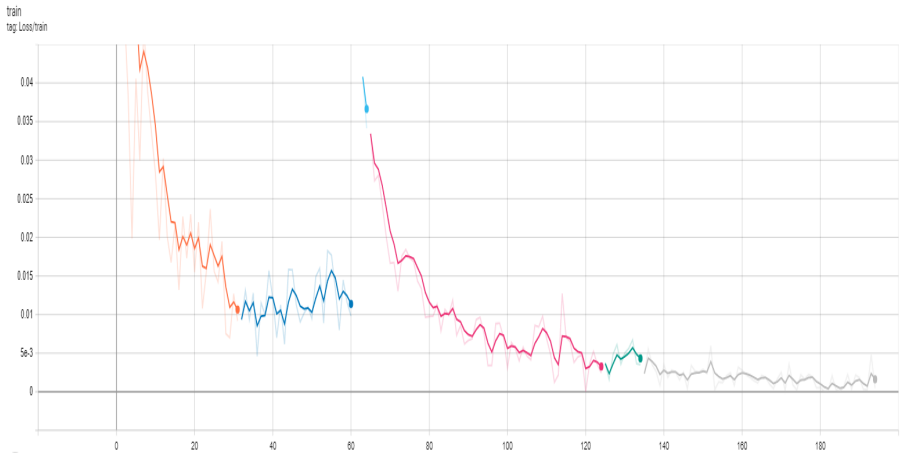


Figure 8: The traiing loss value over time for the second model.

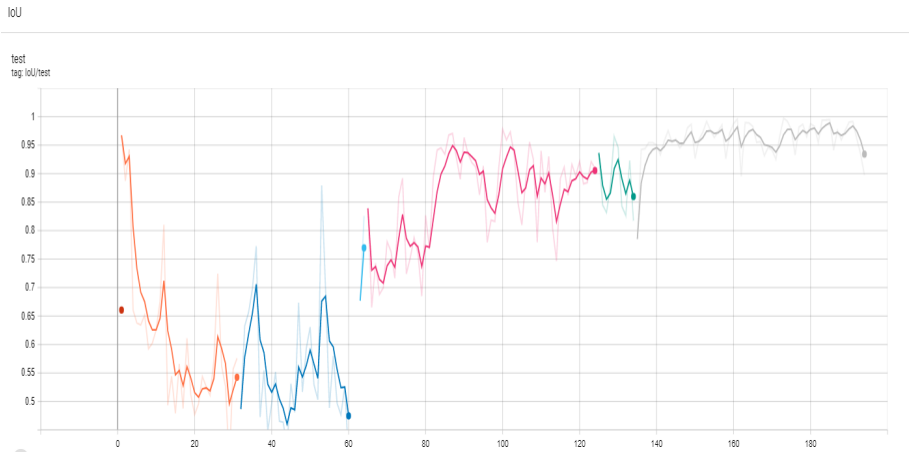


Figure 7: IoU value of Validation Testing over time (Second trained model). The network was trained over different periods of time, causing TensorBoard to mark them as different runs.

We trained the network a third time, this time using a batch size of 8 instead of 4. The validation IoU, shown in Figure 9, stayed high through training, but had dips in the 70% range at times. The total number of epochs trained were 290. Figure 10 shows the training loss over time. We reach fairly low values and it looks like the model is converging.

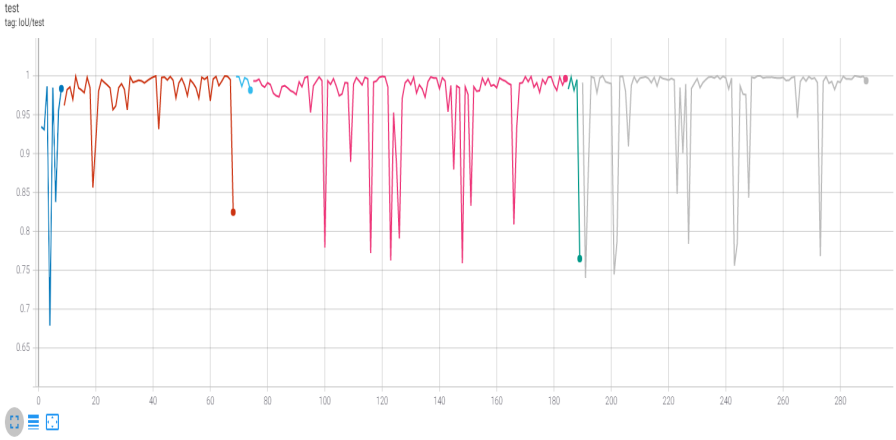


Figure 9: IoU value of Validation testing over time (Third trained model. The network was trained over different periods of time, causing TensorBoard to mark them as different runs.)

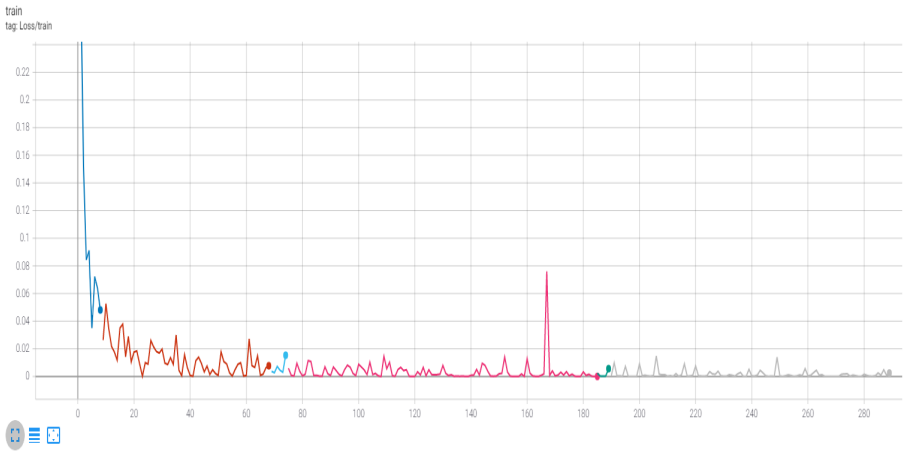


Figure 10: Training loss over time for the third trained network

6 Discussion

The three trained networks, each using a slightly different training configuration, all showed some promise using the metrics defined. They all managed to reach a high value for IoU on the test set defined, which seems to show some promise.

The reason why we trained three separate networks were because while the test set results were good, running full head models from the FaceGen100 dataset showed some flaws with the tracking results on meshes that were a mixture of face and non-face points. These results can be seen in Figure 11. The figure shows that when we ran the post-processing face extraction step on the first network, we ended up getting all vertices classified as FacePoints. This is a somewhat promising result, at least combined with the high IoU on the validation set. It means that the network is capable of categorizing head meshes as containing faces, while also categorizing non-face meshes as not containing faces. However, for tracking purposes, the result is not particularly good, as the network should only extract the face vertices.

The extraction results on the second trained model were, on the other hand, disappointing, as we no longer were able to detect faces in the head meshes. The IoU was at around 88%, so the network was performing well on the face and non-face meshes, but the overall result on the head meshes was noticeably worse. The face detection percentage was also the lowest of all the trained models, achieving 91% compared to the others which were in the high nineties.

The third time we trained the model with a higher batch size compared to the earlier ones, with a batch size of 8 instead of 4. We saw that the IoU value stayed consistenly high during training, rarely dipping below 90%. After the last epoch, the IoU value was at 99%, as you can see in Figure 9. The network managed much better results on the face detection, achieving a score of 99.7%. It also managed to achieve a high face detection score on the FaceGen100 set. The face tracking on the FaceGen100 dataset was more diverse compared to the prior networks, achieving a mixture of face points and non-face points. As you can see in Figure 11, the extracted regions did not correspond well with the actual face region of the mesh.

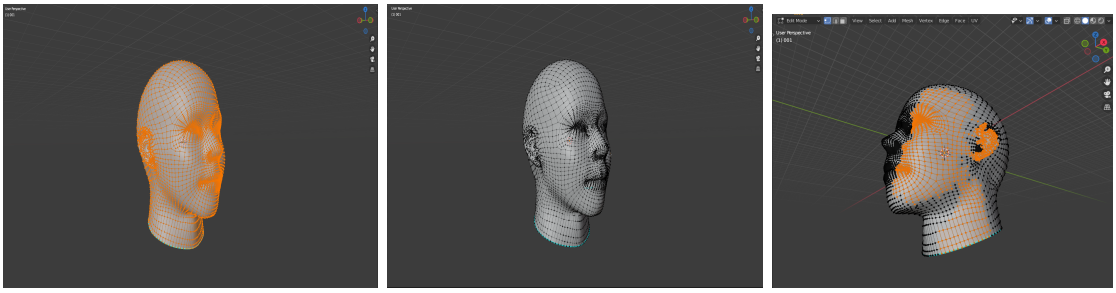


Figure 11: The face regions each network from the first mesh in the FaceGen100 dataset.

6.1 Dataset

One of the challenges with the training of the network is that the dataset only contains face meshes or non-face meshes. This is a problem, because the network does not have proper structures for meshes that contain a mixture of face points and non-face points. We attempted to alleviate this by increasing the mixture of face meshes and non-face meshes, which we did when training the second network.

We saw some more interesting results when we trained the network on the same dataset with a higher batch size and more epochs, i.e. the third iteration of the network. Instead of the results we saw with the first two trained models, we ended up extracting only parts of the model. It could mean that if we trained the network for even longer, we might have seen even better face extraction on the FaceGen100 dataset.

7 Future work

The current network performs admirably on meshes containing one face, and seems capable of detecting whether or not the input is of a face or not. However, the network is not particularly good at tracking faces on more complex scenes.

7.1 Dataset

The current lack of good datasets for the purpose of face tracking on 3D meshes is one of the more important challenges to address in the future. It is necessary to both create task-specific datasets, as well as increasing variety in existing datasets and the number of meshes in them. Creating complex scenes with multiple meshes could potentially be important to improve the training of networks and the expected results.

In the 2D image realm, for example, we have datasets with a magnitude in the order of tens of thousands[10]. The dataset used in this thesis consists of 3500 face meshes, of which 700 were used for validation. More is better, as the model has more data to train and learn from, so this is an important area to expand.

7.2 Bounding Volume Regression

It might be possible to take inspiration from object detection networks in the 2D image realm, like SSD[11] that performs object detection and classification by using bounding box regression as the optimization problem. There might be good reasons to look into using something like the graph convolution network[3] to perform some kind of bounding volume regression, and returning either the vertices in the bounding volume, or just the bounding volume itself for the purpose of tracking faces in 3D meshes.

8 Conclusion

In this thesis, we have introduced an application for the segmentation version of the PointNet++[16] network for the purpose of performing face tracking on 3D meshes. It adds a post-processing step that extracts a yes or no answer for face detection purposes, as well as extracting predicted face regions on the mesh for tracking purposes.

The available resources on 3D face tracking on meshes are few, and the current datasets are few and generally small. We propose using a mixture of the BU3DFE[21] dataset together with a proprietary dataset made with the FaceGen software and the ModelNet10[19] dataset from Princeton. Our proposed solution performs well on the validation set, achieving scores of 99% IoU and 99.7% face detection rate. However, the performance does not translate to meshes that are more complex, and containing both vertices classified as faces and vertices classified as not face points. The results points towards possible future improvements, if more specialized and larger datasets are introduced.

Bibliography

[1] Wael Abd-Almageed et al. 'Face Recognition Using Deep Multi-Pose Representations'. In: *CoRR* abs/1603.07388 (2016). arXiv: 1603.07388. URL: http://arxiv.org/abs/1603.07388.

[2] Michael M. Bronstein et al. 'Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges'. In: *CoRR* abs/2104.13478 (2021). arXiv: 2104.13478. URL: https://arxiv.org/abs/2104.13478.

[3] Michaël Defferrard, Xavier Bresson and Pierre Vandergheynst. *Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering*. 2017. arXiv: 1606.09375 [cs.LG].

[4] Inderjit S. Dhillon, Yuqiang Guan and Brian Kulis. 'Weighted Graph Cuts without Eigenvectors A Multilevel Approach'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 29.11 (2007), pp. 1944–1957. DOI: 10.1109/TPAMI.2007.1115.

[5] Jianli Feng and Shengnan Lu. 'Performance Analysis of Various Activation Functions in Artificial Neural Networks'. In: *Journal of Physics: Conference Series* 1237 (June 2019), p. 022030. DOI: 10.1088/1742-6596/1237/2/022030.

[6] Matthias Fey and Jan Eric Lenssen. 'Fast Graph Representation Learning with PyTorch Geometric'. In: *CoRR* abs/1903.02428 (2019). arXiv: 1903.02428. URL: http://arxiv.org/abs/1903.02428.

[7] Singular Inversions Inc. *FaceGen*. 2021. URL: https://www.facegen.com (visited on 8th July 2021).

[8] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[9] Zhichao Lian, Shuai Shao and Chanying Huang. 'A Real Time Face Tracking System based on Multiple Information Fusion'. In: *Multimedia Tools and Applications* 79.23 (June 2020), pp. 16751–16769. ISSN: 1573-7721. DOI: 10.1007/s11042-020-08889-4. URL: https://doi.org/10.1007/s11042-020-08889-4.

[10] Yiming Lin et al. *MobiFace: A Novel Dataset for Mobile Face Tracking in the Wild*. 2019. arXiv: 1805.09749 [cs.CV].

[11] Wei Liu et al. 'SSD: Single Shot MultiBox Detector'. In: *Lecture Notes in Computer Science* (2016), pp. 21–37. ISSN: 1611-3349. DOI: 10.1007/978-3-319-46448-0_2. URL: http://dx.doi.org/10.1007/978-3-319-46448-0_2.

[12] Jonathan Masci et al. *Geodesic convolutional neural networks on Riemannian manifolds*. 2018. arXiv: 1501.06297 [cs.CV].

[13] Lester James Miranda. 'Understanding softmax and the negative log-likelihood"'. In: *ljvmiranda921.github.io* (2017). URL: %5Curl%7Bhttps://ljvmiranda921.github.io/notebook/2017/08/13/softmax-and-the-negative-log-likelihood/%7D.

[14] Adam Paszke et al. 'PyTorch: An Imperative Style, High-Performance Deep Learning Library'. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[15] Charles Ruizhongtai Qi et al. 'PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation'. In: *CoRR* abs/1612.00593 (2016). arXiv: 1612.00593. URL: http://arxiv.org/abs/1612.00593.

[16] Charles Ruizhongtai Qi et al. 'PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space'. In: *CoRR* abs/1706.02413 (2017). arXiv: 1706.02413. URL: http://arxiv.org/abs/1706.02413.

[17] Amit K. Roy-Chowdhury and Yilei Xu. 'Face Tracking'. In: *Encyclopedia of Biometrics*. Ed. by Stan Z. Li and Anil Jain. Boston, MA: Springer US, 2009, pp. 383–388. ISBN: 978-0-387-73003-5. DOI: 10.1007/978-0-387-73003-5_90. URL: https://doi.org/10.1007/978-0-387-73003-5_90.

[18] Prof. M. Seshaiah and Prof. Shrishail Math. 'Comparative Analysis of Various Face Detection and Tracking and Recognition Mechanisms using Machine and Deep Learning Methods'. In: *Turkish Journal of Computer and Mathematics Education* (May 2021).

[19] Zhirong Wu et al. *3D ShapeNets: A Deep Representation for Volumetric Shapes*. 2015. arXiv: 1406.5670 [cs.CV].

[20] Jiaolong Yang et al. *Neural Aggregation Network for Video Face Recognition*. 2017. arXiv: 1603.05474 [cs.CV].

[21] Lijun Yin et al. 'A 3D facial expression database for facial behavior research'. In: *7th International Conference on Automatic Face and Gesture Recognition (FGR06)*. 2006, pp. 211–216. DOI: 10.1109/FGR.2006.6.

Appendix

A Custom Blender Selection Script

To use the custom blender script that selects vertices from the network output, you first need to install Blender. After that, you open an empty scene. If there are default elements in the scene, you can press A to select all objects, followed by X to delete selected objects. If you then follow Figure 12, you can import the desired .obj model into Blender. Make sure you select Keep Vert Order under the geometry tab, as shown in Figure 13, otherwise Blender will create its own indices for the model, and the selection script will end up selecting unrelated vertices.

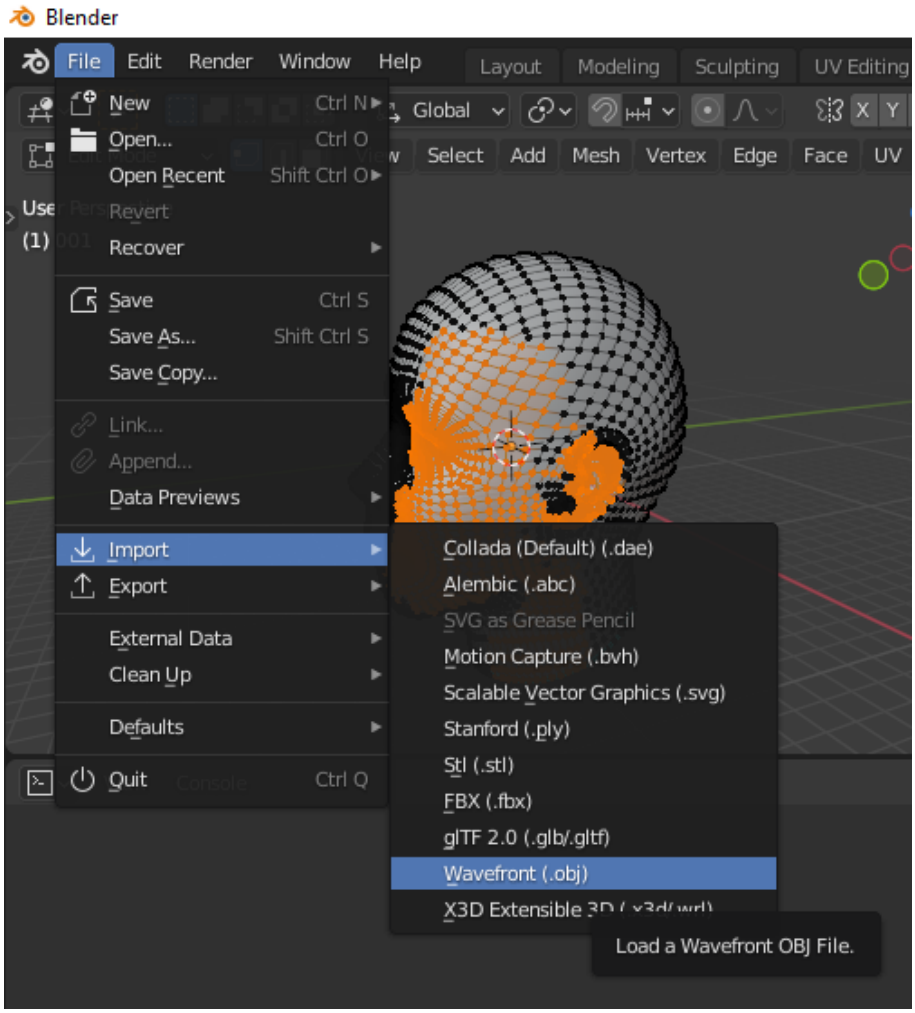


Figure 12: Select the wavefront model file type when importing

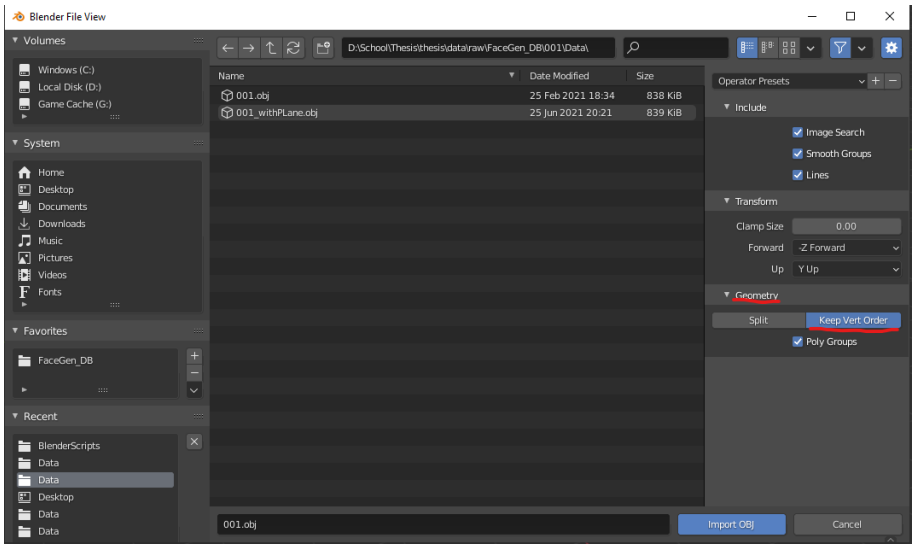


Figure 13: Select the model you want to import, make sure that under geometry that Keep Vert Order is checked.

After you have imported the model, you need to open the scripting tab in Blender, and open the SelectFaces.py script. In the face array, paste in the copied face region from the post-processing step in the network. Make sure to select the model before pressing run on the script. Figure 14, Figure 15 and Figure 16 show these steps.

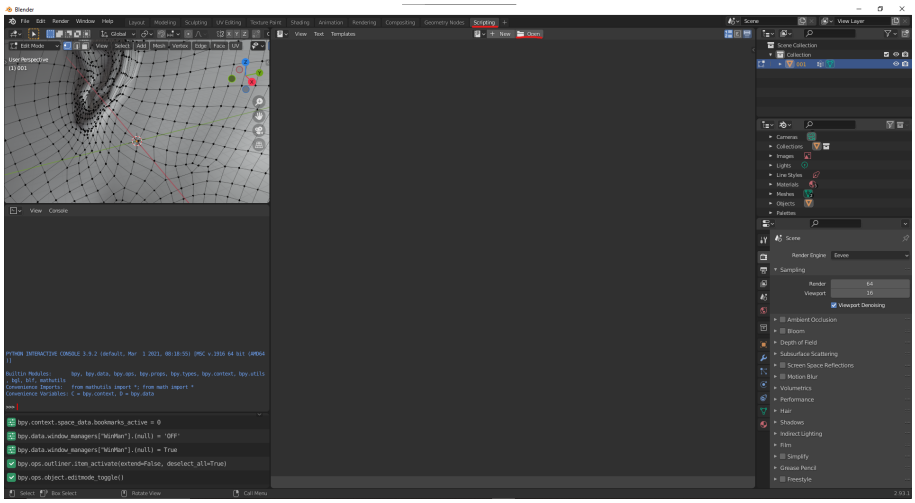


Figure 14: Open the scripting tab

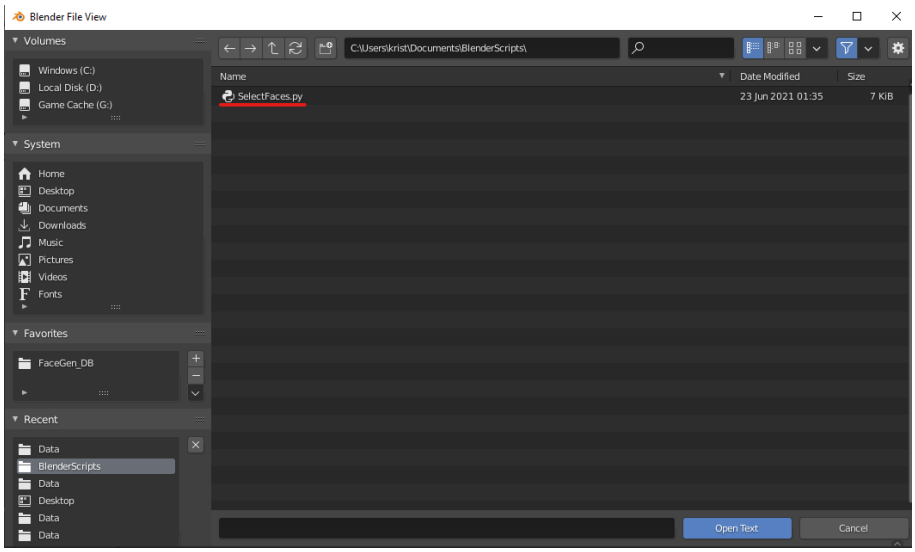


Figure 15: Find the script in the project folder.

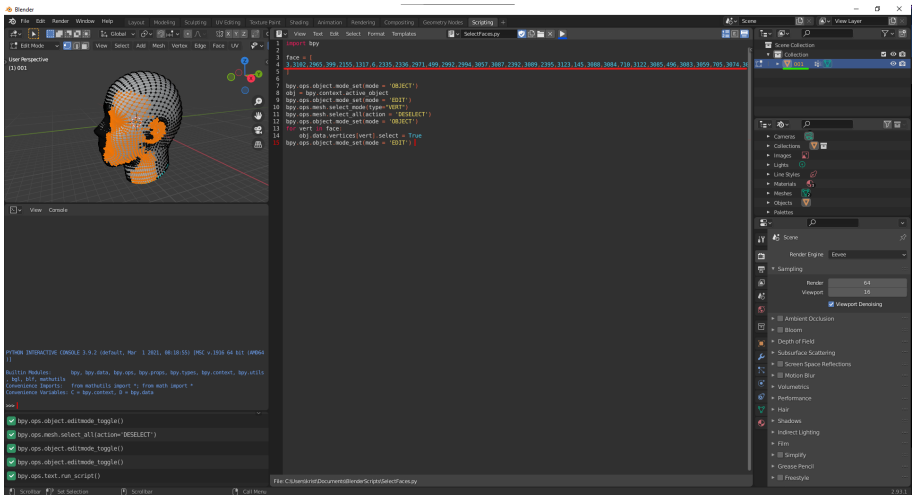
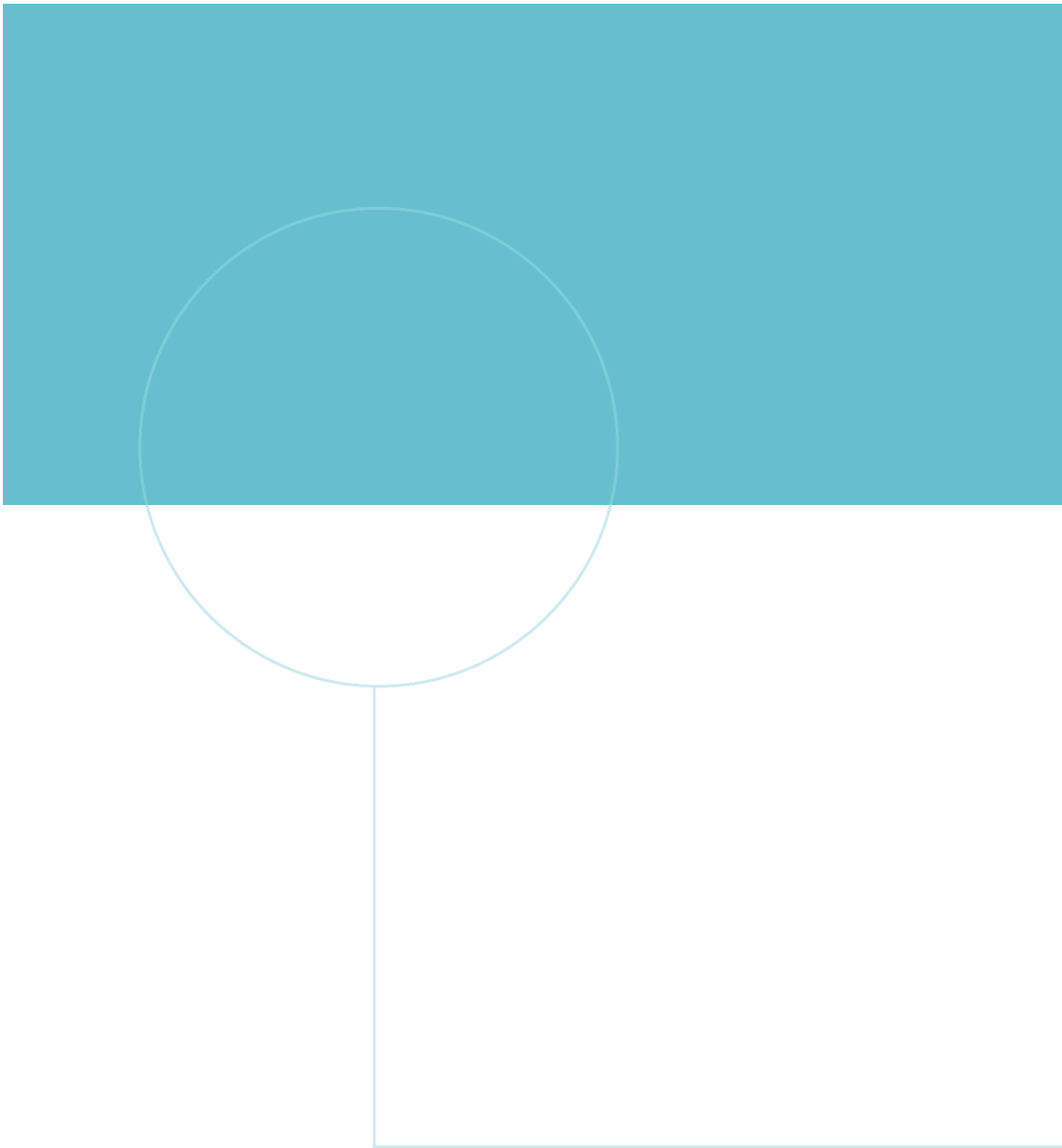


Figure 16: Select the model(green underline). Paste the indices from the post-processing step (red underline). Press the run button(blue underline)



NTNU

Kunnskap for en bedre verden