Master's thesis

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

Sindre Thomassen

3D Face Reconstruction Using Facial Image Sequences

Master's thesis in Computer Science
Supervisor: Theoharis Theoharis
Co-supervisor: Antonios Danelakis
June 2022

NTNU
Kunnskap for en bedre verden

Sindre Thomassen

3D Face Reconstruction Using Facial Image Sequences

Master's thesis in Computer Science
Supervisor: Theoharis Theoharis
Co-supervisor: Antonios Danelakis
June 2022

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

NTNU
Kunnskap for en bedre verden

Abstract

As the field of 3D face reconstruction keeps evolving, one trend seems to stay the same. Reconstruction is usually done given a single or a small set number of input images. This work proposes a way to conduct 3D facial reconstruction with machine learning given a sequence of 2D facial images as input. It aims to find a way to take an arbitrary amount of facial images as input where the quality of the results improves with each new image presented to the network. The proposed model shows potential, although it suffers from overfitting due to limitations concerning the available dataset. Further work discusses ideas for improvement concerning the dataset used, and aproposes improvements for the proposed network architecture.

Sammendrag

Etterhvert som 3D ansikts rekonstruksjon utvikler seg er det en trend som tilsynelatende går igjen. Rekonstruksjon blir stort sett gjort ved bruk av ett enkelt bilde, eller et lavt fast antall bilder som input. Dette arbeidet foreslår en måte å gjøre 3D ansikts rekonstruksjon ved hjelp av maskinlæring gitt en sekvens bestående av ansikts bilder som input. Vi forsøker å finne en måte å ta inn ett vilkårlig antall ansikts bilder som input, hvor hvert nytt bilde fører til et forbedret resultat generert av nettverket. Den foreslåtte modellen viser potensiale, men lider av overfitting på grunn av begrensninger med tanke på det tilgjengelige datasettet brukt. Videre arbeid diskuterer ideer for å forbedre datasettet i tillegg til forbedringer med tanke på den foreslåtte nettverks arkitekturen.

Preface

This thesis is the product of the work done during spring 2022 in the field of 3 dimensional facial reconstruction. This work has been done as part of a masters program in computer science.

I would like to extend my gratitude to my supervisors professor Theoharis Theoharis and Post-doctoral Fellow Antonios Danelakis, both for the great support given throughout not only this assignment, but also the preliminary work associated with it, as well as their rapid response time when asked for help. I would also like to thank Rolf harald Dahl at IDI DRIFT for granting access to a lab computer with the needed hardware to work on the assignment, as well as quickly responding and fixing technical issues should they arise.

Contents

List of Figures

List of Tables

1 Introduction

Simple 2D reconstruction networks are able to create highly realistic reconstructions of human faces using a basic encoder-decoder architecture. Notable examples from recent times like Deepfakes are able to reconstruct high-quality faces to such a degree it starts blurring the line between real and synthetic data. By training a common encoder to encode any given face, with a specialized decoder trained on a single given face, the network is able not only to reconstruct a face from images of the said face but able to abstract ANY given face and decode it to a face of the users choosing, effectively transposing a chosen face onto any given subject.

Considering this, it is only natural to start thinking about 3D facial reconstruction and advances in this field as well. Compared to 2D reconstruction, some aspects would improve from a 3D upgrade, such as better biometric identification given light and pose independence, as well as additional security as 3D data is less susceptible to forgery[1]. With the 3D upgrade however, challenges also arise. How do we best represent a 3D face in our machine learning models? What kind of data do we train on for the best results? What type of networks do we use? Is the simple encoder-decoder architecture enough also in 3 dimensions, or do we need to consider other alternatives as well?

In recent years, notable works within the field of 3D reconstruction include PRN [2] by Feng et al. from 2019, who approached the problem of predicting accurate 3D models by representing the 3D data as 2 dimensional by using UV maps. Further work based on this was presented in [3]. Whereas the model proposed in [2] uses a single image, [3] modifies the architecture to take two images in addition to updating the residuals used in PRN with inverted residuals presented in MobileNetV2 [4]. Other work with reconstruction in general, not strictly focusing on facial reconstruction, is the 3D-R2N2 network proposed by Choy et al. in [5], where they are able to accurately reconstruct a low-resolution representation of any 3D object with multiple images from different angles.

Common for these works within 3D facial reconstruction is the use of 1-2 images, mostly from given positions. In this thesis we propose a model based upon [3] which uses LSTM layers to introduce the aspect of memory. The encoder has been modified to include several LSTM layers instead of some of the inverted residuals, thereby adding a temporal dimension to the input. We are then able to feed it multiple images across time to hopefully increase the quality of the final prediction with each new image introduced as each new image could contain new information the model didn't have previously, or confirm old information to a greater extent. The proposed network will also attempt to predict 3D point clouds directly instead of going through the position maps.

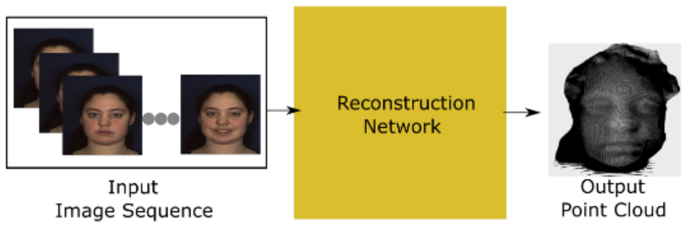


Figure 1: Illustration of suggested pipeline

2 Background

This section contains basic information about the technology and core concepts necessary for this masters project.

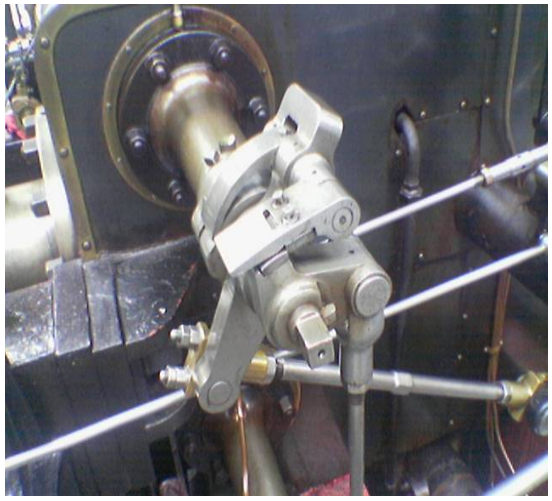
- Subsection 2.1 describes a Convolutional Neural Network and some core concepts related to, including the convolutional operator, activation functions, and pooling.
- Subsection 2.3 describes Residual Neural Networks used for better performance in deep learning.
- Next subsection 2.4 goes on to shortly describe Recurrent Neural Networks.
- Subsection 2.5 describes the basic architecture of an encoder-decoder network.
- Subsection 2.6 describes the work of a loss function in machine learning and gives some examples of different loss functions.
- Subsection 2.7 goes through the MobileNet architecture to explain how the inverted residual blocks are used.
- Finally, subsections 2.8 and 2.9 presents information about the Open3D and OpenCV2 libraries used for processing multi-dimensional geometries and images respectively.

2.1 Convolutional Neural Networks

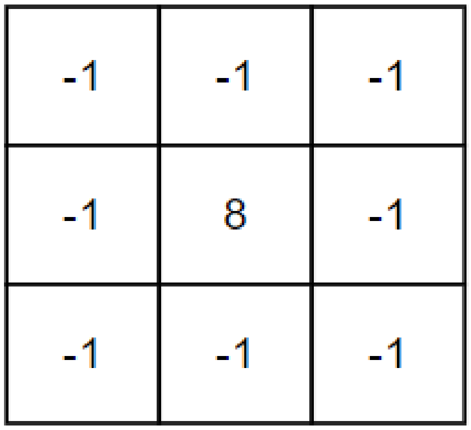
Convolutional Neural Networks (CNN) are simply put neural networks consisting of at least one convolutional layer. Unlike a dense layer where matrix multiplications are used to process the input data, in a convolutional layer the convolutional operator is used. Unlike regular neural networks with dense layers, a CNN does not only work with simple vectors but works with 2-3D matrix representations of input data, making it ideal for working with images. When working with image processing we would sometimes like to preserve the spatial information of the image, such as which certain features can be found next to another. A matrix representation of an image in combination with the convolutional operator, the convolutional layer achieves this preservation to a certain extent.

2.2 Convolutional Layer

The input of a convolutional layer in image processing is typically an image or a set of feature maps as resulting from a previous convolutional layer. The idea behind a convolutional layer is to process the input image and extract features present in the image, using a set of trainable weights called a kernel. From the extracted features the network can then better predict what is present in the image and not.

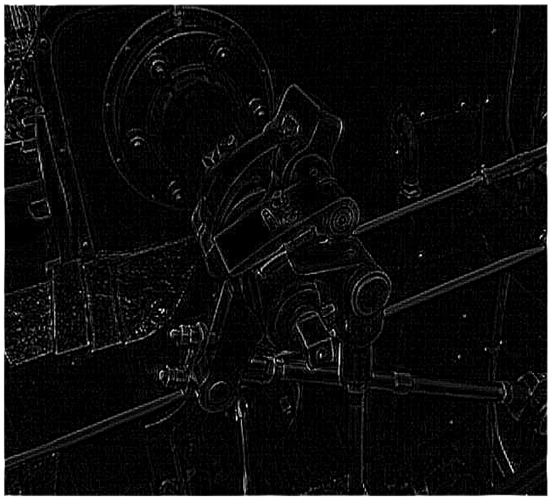


(a)



-1	-1	-1
-1	8	-1
-1	-1	-1

(b)



(c)

Figure 2: 2a: Original image from wikipedia. 2b: Laplacian edge detection kernel. 2c: Extracted feature map.

Figure 2 shows a simple convolutional demo which applies a Laplacian edge detector to the original image to extract edges in the image. In a convolutional layer many kernels would be present to generate an array of different feature maps as the network looks for a multitude of features to base it's prediction on.

[6] states that a convolutional layer typically consists of 3 separate parts, starting with the convolutional operator.

2.2.1 Convolution operator

In image processing, a convolution is the process of using a input image and a convolutional kernel to extract certain features from the input image. The convolutional operator across both x- and y-axis is defined in [6] as equation 9.

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{1}$$

Convolution is commutative, a property resulting from flipping the kernel relative to the input, which means equation 9 can be written as equation 2[6].

$$S(i,j) = (K * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2}$$

Because this property is not strictly necessary for machine learning many implementations use cross correlation instead, defined by equation 3 from [6].

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+mj+n)K(m,n) \tag{3}$$

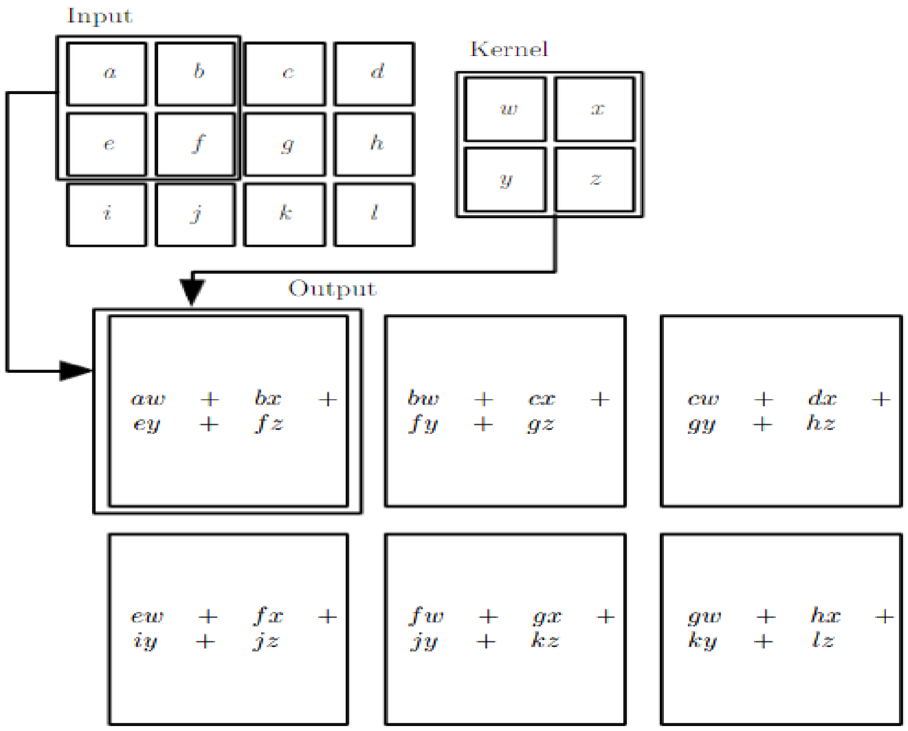


Figure 3: Convolution illustration from [6]. Place the kernel in each possible location of the image, and calculate a pixel value for the results by summing the products together.

Cross-correlation is the same operation as convolution, only without flipping the kernel before use. Whether cross-correlation or real convolution is used does not have a significant impact on the final model as during training the network will adapt its weights to better perform with its given operations.

In practice, convolution can be applied by iterating the kernel over each valid overlapping position in the original image, and for each position, calculate the new pixel value by summing the product of each overlapping value in the kernel and its current position in the image.

2.2.2 Activation Layer

The second step through a typical convolutional layer is the activation layer. The activation layer comes after the convolutional layer and uses a non-linear activation function to further modify the given input. In machine learning, there are numerous activation functions to choose from when designing a model. Some more well-known ones include ReLU and Leaky ReLU, Sigmoid, and Tanh.

The ReLU function is an activation function that simply returns the biggest value of an input x and 0. The ReLU function is defined by equation 4

$$ReLU(x) = max(0, x) \tag{4}$$

As this function simply compares the input value to 0 and returns the bigger value, this function is very fast and easy to compute and has great learning potential. The ReLU function is not without its problems however. The ReLU function is susceptible to the dead neuron problem[2] where some neurons cease to update. This could be because of one very big change in weights where the resulting weights for a neuron is a high negative value, resulting in the neuron only returning 0 values later and becoming unable to update again. This can also happen if weights and biases are initialized poorly, for example with a lot of zero values, resulting in neurons only returning zero values and struggling to update themselves. As the derivative of ReLU is 0 for negative values, the network can not learn from negative activation values.

In addition to improved initialization, the Leaky ReLU activation function is a modified version of ReLU designed to improve upon this problem. By modifying the ReLU function to not return 0 for negative values, but rather use a tiny slope to return a fraction of the negative input, the derivative is no longer 0 for negative values. This helps keep neurons alive and lets the network learn from negative values. The Leaky ReLU activation function is defined in equation 5

$$L\_ReLU(x, \alpha) = max(\alpha * x, x) \tag{5}$$

The sigmoid function is another commonly used function. This function takes the input and returns a value between 0 and 1. This is useful when working with data where all desired values are within this range, such as when working with probabilities. The Sigmoid activation function is defined by equation 6

$$Sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{6}$$

[2]Simple description of dead neural problem https://towardsdatascience.com/neural-network-the-dead-neuron-eaa92e575748

As the inputs of the sigmoid function get higher or lower, the derivative of the function starts to reach 0, thus making the activation function particularly prone to disappearing gradients.

The tanh function is similar to the sigmoid function as it also takes any input and returns a value in a range of -1.0 to 1.0. Like sigmoid, tanh also suffers from the vanishing gradient problem as high and low input values causes the derivative to become 0. Due to the output value range including negative values, the tanh function is slightly easier to optimize compared to the sigmoid function. Tanh is defined by equation 7.

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{7}$$

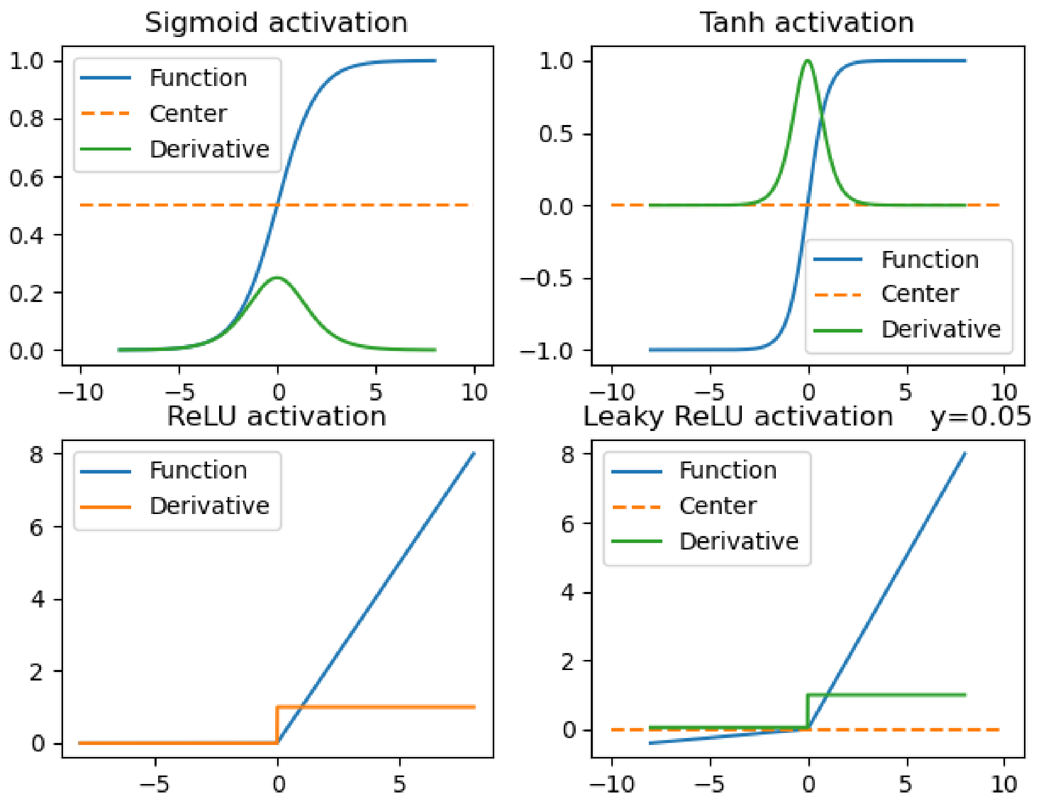


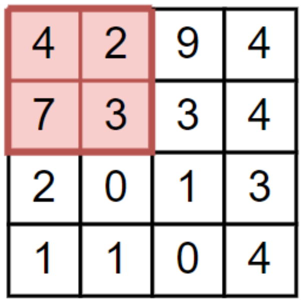
Figure 4: Graphs of activation function

Figure 4 shows a graph of the four functions mentioned with their derivatives.

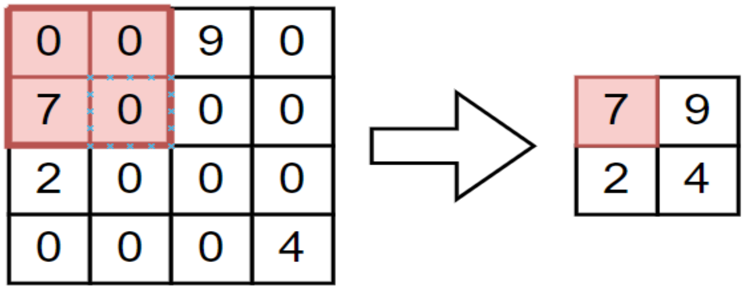
2.2.3 Pooling

At the third stage of a convolutional layer, the pooling layer generates a small summarized version of the input[6]. Examples of pooling methods are max pooling, where the new value is defined as the highest value in a set of adjacent pixels; and average pooling, where instead of taking the highest value, all values in the set of adjacent values are averaged.

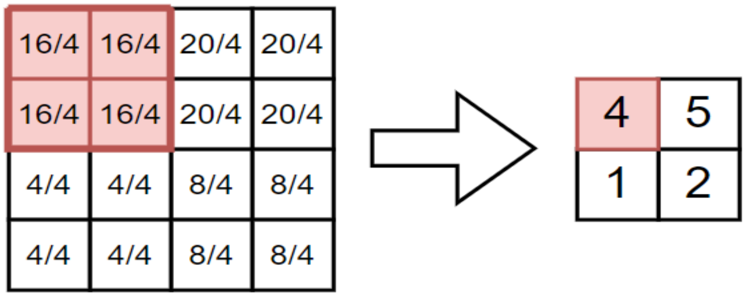
The pooling operation is done similar to the convolutional operations. By using a kernel of a set size, one iterates the kernel across the input image with a stride equal to the kernel size and calculate the new value from the values inside the area covered by the kernel.



(a) Image values before pooling.



(b) Max pooling illustration.



(c) Average pooling illustration.

Figure 5: Figure demonstrating two pooling methods on input (a) using a 2x2 kernel. (b) demonstrates max pooling, where the top left regions highest value is 7. (c) demonstrates average pooling, where the top left region adds up to 16, which is divided by 4 to get 4.

2.2.4 Transposed Convolutions

Transposed convolution is an upsampling technique, taking in a number of smaller feature maps and generating an upsampled image. Where a convolutional layer takes in an image and extracts certain features, a transposed convolution takes in the feature maps and tries to predict a possible output relating to the given feature maps. For image reconstruction and encoder-decoder architectures, this is useful as we want to extract features that then are used to create a new image with a spatial resolution on par with the original input image.

The process is similar to that of regular convolutions. An input of size WxH is convoluted by a kernel of size KxK. Unlike a regular convolution however, the transposed convolution is done by placing the kernel at the same position as each individual pixel, where the entire kernel is multiplied by the single pixel value. This area in the output will then be a KxK area representing the one pixel value in the input. Next, move the kernel to the next pixel. When moving the output area, some of the area will overlap with the area of the first pixel. Once this is done for each pixel in the input, every multiplication is summed together where the areas overlap and we have our upsampled feature map.[3]

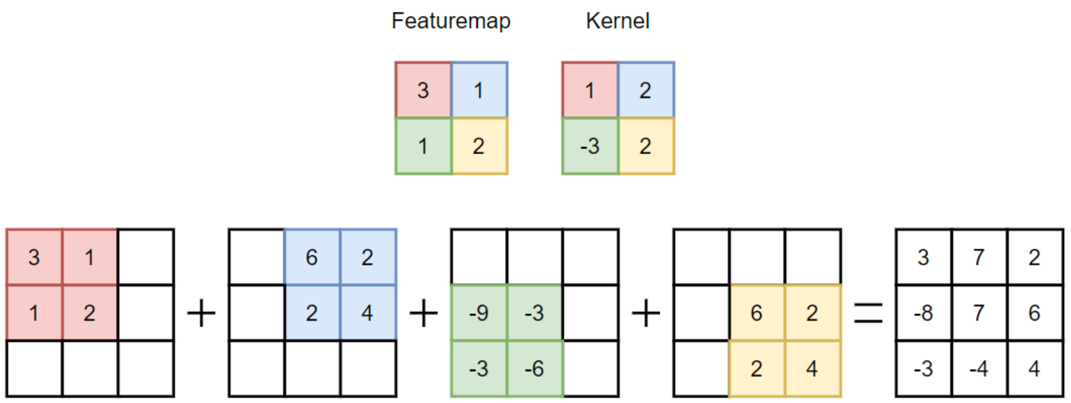


Figure 6: Transposed convolution illustration.

Image 6 demonstrates a transposed convolution on a 2x2 feature map using a 2x2 kernel and a stride of 1.

2.3 Residual Neural Networks

Residual neural networks (ResNet) were first proposed in "Deep Residual Learning for Image Recognition"[7] from 2015 by Kaiming He et al. Their paper proposes a neural network that aims to reduce the problems of vanishing and exploding gradients. This is done by letting the model skip certain layers using residual blocks. A residual block typically consists of convolutional layers as a regular CNN but introduces the addition of a shortcut. The shortcut will in practice add the

[3]https://towardsdatascience.com/transposed-convolution-demystified-84ca81b4baba

unmodified input of the block to the final result of the convolutional layers, thus preserving information that might have been lost otherwise and the network is less likely to get very high, or infinitely small gradients when conducting backpropagation.

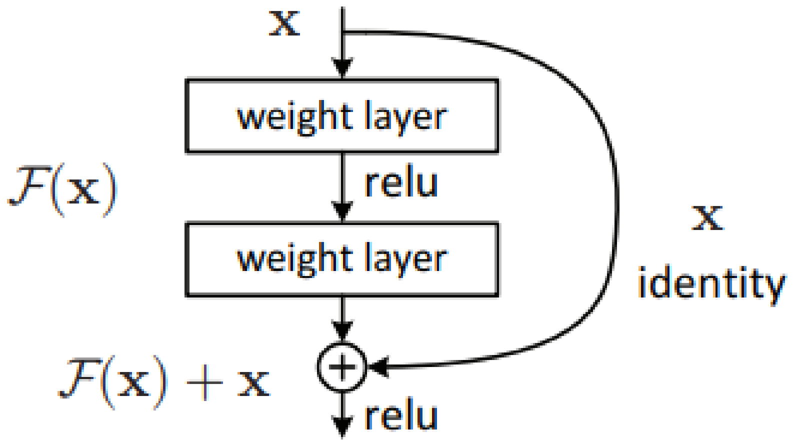


Figure 7: Residual block as illustrated in [7]

Figure 7 shows a typical residual block, featuring two weighted layers interspersed with the ReLU activation function. An identity transformation acts as the shortcut, which simply adds the input to the output of the two convolutional layers and creates the final result.

2.4 Recurrent Neural Networks

Recurrent Neural Networks (RNN) are neural networks with added temporal connections, making it able to handle sequential inputs. Utilizing an internal hidden state, an RNN is able to "remember" previous inputs and potentially give better predictions. Because of RNN's ability to take in sequential data over time, it might be possible to improve upon a prediction given more context around the input already received, and hopefully get a more accurate result the more data that is presented.

2.4.1 Long Short Term Memory

Long Short Term Memory (LSTM) is a successor of RNNs introduced in [8] from 1997. What sets LSTM apart from regular RNN is the addition of control gates in the LSTM structure, resulting in a more capable memory that is able to remember over longer periods of time. The control gates allow the LSTM cells to train parameters to better know when to forget, update and use the information stored in the memory cell.

Further improvements have been done with regards to LSTM in the form of Gated Recurrent Units (GRU) which allows for more refined control of which parts of the new input are included as well as how much of the previous memory state should be ignored when making new predictions.

2.4.2 Convolutional LSTM

A convolutional LSTM is an LSTM cell where all computations are done using convolutions rather than matrix multiplications. This way preserves some spatial adjacency information and helps make better predictions when working with images or other multi-dimensional data where spatial information is important.

2.5 Encoder-Decoder Networks

Encoder-decoder networks are neural networks consisting of two parts. An encoder which takes in the output and encodes it into a state. The decoder then takes the state generated by the encoder and tries to construct the desired output from this. The way the encoder-decoder architecture is designed makes it a viable option for reconstruction and generative networks. Some notable works using an encoder-decoder architecture among others include PRN-net[2], [3], and Deepfake which have been talked about a lot in recent years due to it's uncanny ability to reconstruct any given face the decoder is trained to generate.[4]

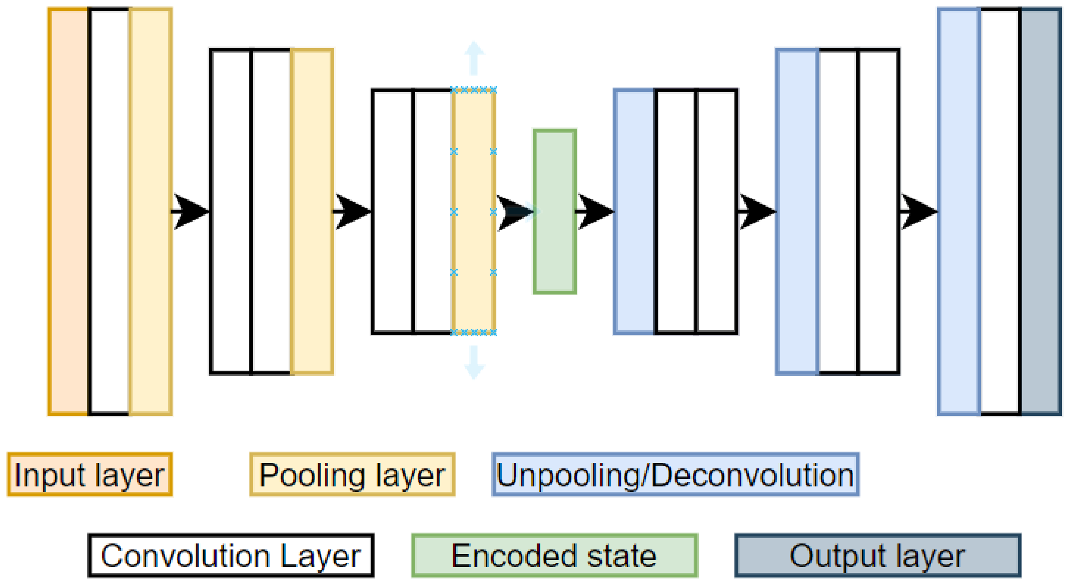


Figure 8: Simple example of an encoder-decoder neural network.

Figure 8 shows a simple example of a encoder-decoder network. From the left, the encoder takes the input and performs convolutions and pooling, downsampling the input to a final 'state'. This state is then taken as input by the decoder, which performs unpooling, transposed convolutions, or applies other methods of upsampling with additional convolutions. The results from the final layer is the final output of the network. Training an encoder-decoder network will focus on reconstructing or generating the desired result from the encoded state generated by the encoder.

[4]https://www.alanzucconi.com/2018/03/14/understanding-the-technology-behind-deepfakes/

In the case of this thesis, the encoder's job is to take in a sequence of images and encode them into a set of feature maps, which are then decoded into a 3D point cloud consisting of vertices for a facial mesh.

2.6 Loss Functions

For a machine learning model to be able to learn at all, we need a metric to measure the model's performance at any given point. This is done using a loss function. The loss function shows how far away from the ground-truth the model's predictions are, and from there it can be adapted further to minimize this loss.

There are multiple loss functions available for use in machine learning, with some being better suited for reconstruction than others. A well-known loss function is the Binary Cross-Entropy loss function which is commonly used for classification networks, but not that useful for regression models which is the topic of this paper. We also have several loss functions more suited for regression models, with the one used here being the Mean Squared Error (MSE) loss function.

The loss function takes in the ground-truth and the predictions as arguments, from which it calculates the euclidean distance between each point in the ground-truth and the corresponding point in the prediction before the sum of all distances is averaged with regards to the number of points. The MSE loss function is defined in equation 8 with N being the number of predicted points.

Combining the loss function with an optimization algorithm, the model is able to update its trainable parameters using the derivative of the loss function and the chain rule, back propagating through the network and updating the weights and biases used to generate the prediction.

$$MSE(p, y) = \frac{1}{N}\sum_{i=1}^{N}(y_i - p_i)^2 \tag{8}$$

2.7 MobileNet

2.7.1 MobileNet V2

MobileNet V2 introduced a new building block, called inverted residuals. These inverted residuals aim to be more lightweight and perform better on mobile devices. The inverted residuals achieve this by splitting regular convolutions into a depthwise separable convolution. A depthwise separable convolution consists of a convolutional operation split into two parts. First, a depth-wise convolution is applied by using one KxKx1 kernel for each channel, followed by a point-wise convolution to generate the feature map.

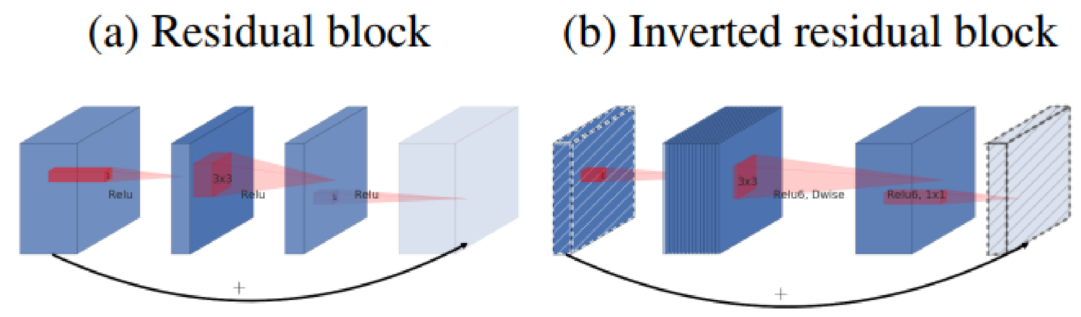


Figure 9: Inverted residual as compared to regular residual block. Illustration from [4]. Where regular residual blocks connect the layers with many channels, the inverted residual blocks connect the bottlenecks[4]

As MobileNet V2 focuses on implementing a lightweight mobile network, depthwise separable has two main benefits above regular convolutions: Fewer trainable parameters, and lower computational cost.

For a regular convolution one would need a single kernel K of size (k*k*c) for each output channel, resulting in N kernels to generate N feature maps. The total trainable parameters for a regular convolution can then be defined by equation 9

$$\#P_{convolution} = K_{width} * K_{height} * C * N \tag{9}$$

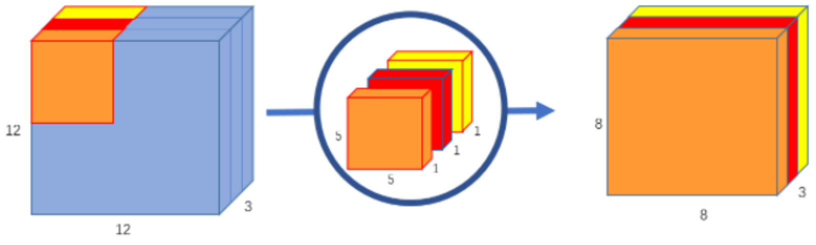
Here C is the number of input channels, while N is the number of feature maps in the output. Depthwise separable convolutions however initially use a single kernel of size (k*k*1) for each input channel to perform a depthwise convolution, resulting in an output with the same amount of channels as the input. Next, a (1*1*c) kernel is used for a pointwise convolution to generate the final feature map. To generate N feature maps in the output, N kernels of size (1*1*c) is used. The total number of parameters for a depthwise separable convolution can then be defined by equation 10

$$\#P_{separable} = K_{width} * K_{height} * C + C * N \tag{10}$$

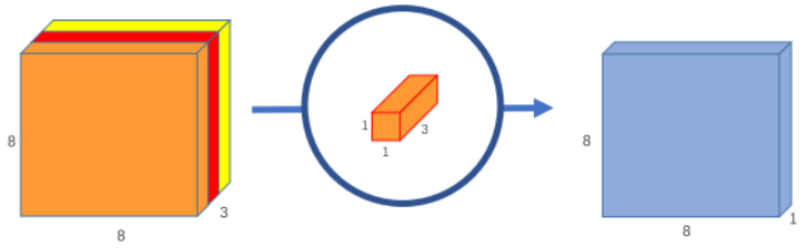
By using depthwise separable convolutions, the computational cost is also reduced[4]. [4] shows that transforming a given input tensor I of shape h*w*c to an output tensor J of size h*w*d using a kernel K of size k*k*c*d has a computational cost S defined by equation 11 for regular convolutional layers, and defined by equation 12 for depthwise separable convolutional layers.

$$S_{convolutional} = h * w * c * d * k * k \tag{11}$$

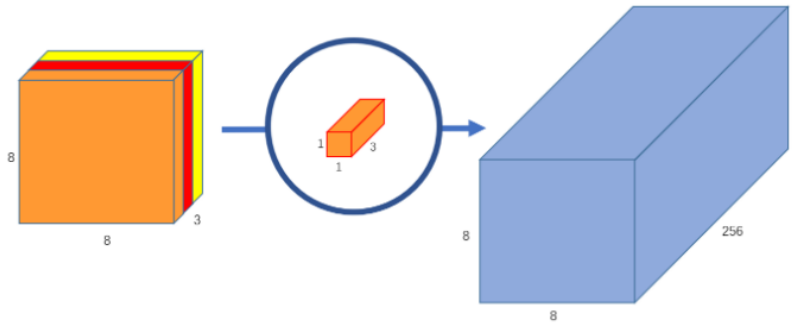
$$S_{separable} = h * w * c(k^2 + d) \tag{12}$$



(a) Step 1 of depthwise separable convolution.



(b) Single pointwise convolution



(c) Pointwise convolution for each new feature map (256)

Figure 10: Illustration of a depthwise separable convolution.Illustrations taken from "A Basic Introduction To Separable Convolutions"
In (a) three kernels of size 3x3x1 is used to apply convolution to each channel of the input image. The result is a 8x8 output with 3 channels. (b) shows a single pointwise convolution which merges the input channels and outputs a feature map. To get an output with a higher number of channels, multiple 1x1x3 kernels are used. (c) uses 256 kernels of size 1x1x3 to output 256 feature maps of size 8x8.

2.7.2 MobileNet V3

The third version of MobileNet attempts to further improve the MobileNet architecture in [9], this time by utilizing Squeeze and Excite blocks, a method proposed in [10] from 2019. The squeeze and excite block first uses global average pooling to reduce each feature map into a single value. Then comes two dense layers, the first using ReLU activation and the second using the sigmoid activation function. The resulting vector is then used to weigh each channel of the output by multiplying each feature map to it's corresponding value in the vector resulting from the dense layers. By weighting the layers in this manner, the model is able to learn which features are more important in certain areas in the network, and find dependencies between certain features[10].

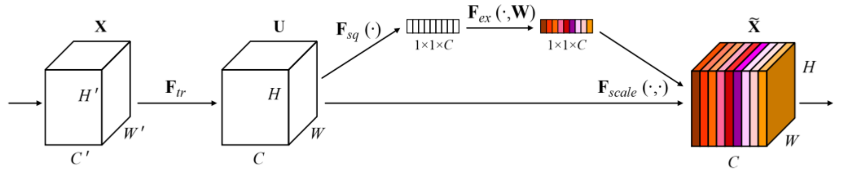
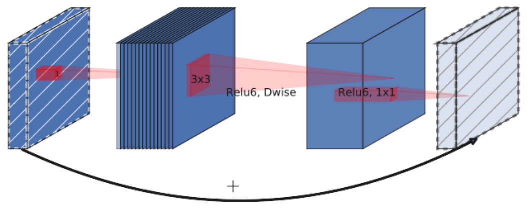
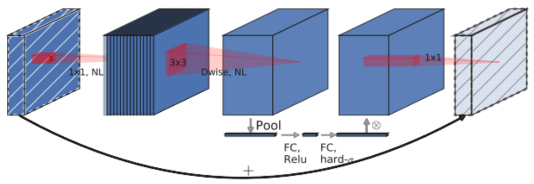


Figure 11: Visualization of a Squeeze and Excite block. After a depthwise global pooling, two dense layers generate a weight vector. The output feature maps are then multiplied with the vector before moving on through the network. Illustration from [10]



(a) MobileNet V2 block



(b) MobileNet V3 block

Figure 12: (a): A MobileNet V2 block. Depthwise convolution followed by point wise consolution. (b): MobileNet V3 block. Same as V2 block, but with the addition of a Squeeze and Excite block between the depthwise and pointwise convolution. Both images from [9]

2.8 Open3D

Open3D is a python tool library for working with 3D models[5]. The library is able, among other functionalities, to read triangle meshes and point clouds. These geometries can then be processed further with the Open3D library. Amongst other operations, one can use Open3D to subsample a triangle mesh, resulting in a point cloud with a set amount of points[11].

2.9 OpenCV2 CascadedeClassifier

OpenCV2 is a python library used for image processing[6]. OpenCV2s CascadeClassifier provides a simple and easy to implement/use way to conduct face detection during preprocessing of the dataset. Instead of training a new network just for finding faces in images, using the cascade classifier with pretrained classifier XML file[7] is used during the preprocessing of the dataset to find the faces in every image. This is used because of the available dataset, where this should be an

[5]http://www.open3d.org/
[6]https://opencv.org
[7]Pretrained classifier obtained from https://github.com/opencv/opencv/tree/master/data/haarcascades

adequate way to do this. (Dataset only contains faces looking straight ahead without large poses. Easy to detect). Once the classifier finds a face in the given input image, the image is cropped and rescaled to only contain a 3-channel face image with dimensions 256x256. Cropping and rescaling functions are also provided by open cv2 and are easy to use.

```python
import cv2
# Paths to neccessary resources
IMAGE_PATH = "path_to_image"
DETECTOR_FILE = "haarcascade_frontalface_default.xml"

image = cv2.imread(IMAGE_PATH)
cascade_classifier = cv2.CascadeClassifier(DETECTOR_FILE)

# Find faces in image. Returns a list of positional parameters
# (x_start, y_start, width, height) for all faces found in the image
faces = cascade_classifier.detectMultiScale(image)
for (x, y, w, h) in faces:
    # cv2.rectangle(IMAGE, POINT_1, POINT_2, COLOR, SIZE)
    cv2.rectangle(image, (x, y), (x+w, y+h), (255, 255, 0), 4)

# Display image and found faces in new window.
cv2.imshow('Faces', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Listing 1: CV2 cascade classifier demo

Code snippet 1 shows a simple example use of cv2s cascade classifier. Figure 13 shows a image used with the above script, and the facial image found by the classifier.

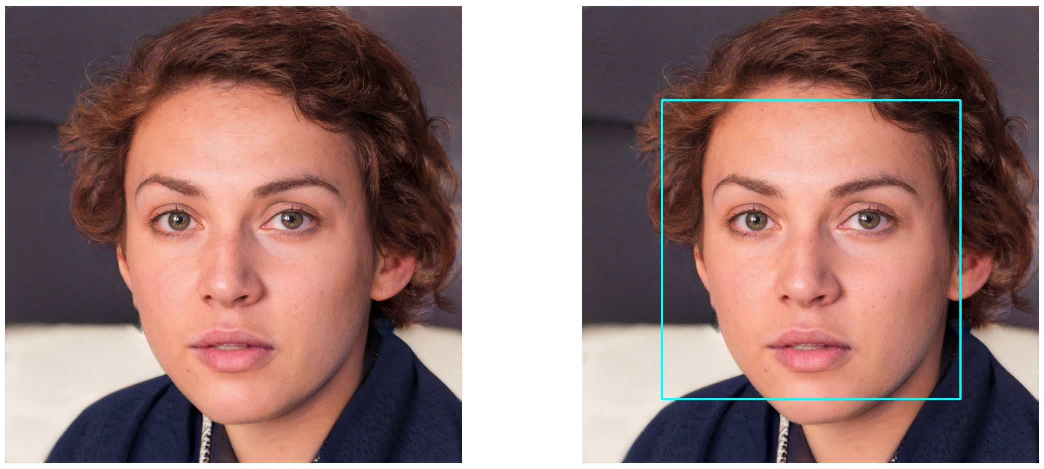


Figure 13: Left: Original image[8]. Right: Face found by detector.

3 Related Works

3.1 PRNet

The Position Map Regression Network, PRN for short, is an approach created by Yao Feng et al. from 2018[2]. The PRN approach is an end-to-end network that is able to conduct dense alignment and facial reconstruction in a single pass, using an approach called UV position mapping. Instead of regressing towards a direct 3D reconstruction, Feng et al. made a 2D representation of a 3D face model. This 2D representation is the UV position map which stores data about the 3D mesh, and the network is then trained to predict these position maps. UV coordinates and UV mapping has been used for things such as textures, bump maps, and normal mapping to name a few, but in PRN, UV space is used to store spatial information about points in a 3D face model where each pixels value in the resulting image can be used to map (r, g, b) values to (x, y, z) space coordinates.

The PRN network uses an encoder-decoder architecture consisting of residual blocks and transposed convolutions respectively to predict the position maps from 2D images. The loss function used is Mean Squared Error (MSE) in combination with a weight mask putting more weight on certain facial points which matters more than other regions of the final result.

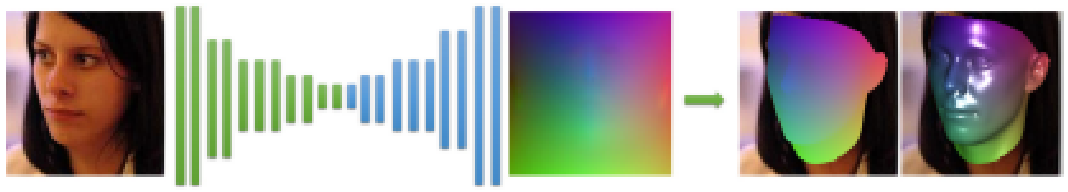


Figure 14: PRNet pipeline illustration. Illustration from [2]. Encoder consisting of residual blocks, decoder of transposed convolutional layers. Predicting UV-position maps and extract mesh from this.

While directly predicting all points in a 3D model can be hard to train as a vector containing all vertices have a problem with spatial adjacency, saving the 3D information in a UV position map is much easier to regress towards. This approach removes the need for multiple dense layers and the need for many more parameters. 5

3.2 3D Facial Reconstruction from Front and Side Images

The work of Ola Lium from 2019 focuses on using UV position maps to predict 3D facial models. With a network based on the works of PRN, Lium was able to improve upon the results by exchanging the existing residual blocks used in PRN with inverted residuals as described in MobileNetV2. The proposed network in [3] uses two input images: one front-facing and one side view image. The ground-truth face models are then adapted to the Basel Face Model (BFM)[12] so as to have a set pose for the network to regress towards.

[8]Image from https://thispersondoesnotexist.com/

Using synthetically generated face data using FaceGen[9] combined with transfer learning on the WLP-300 dataset[10], Lium was able to improve the Normal Mean Error when evaluating on the MICC Florence dataset from 0.0164 as reported by PRN, to 0.0134.

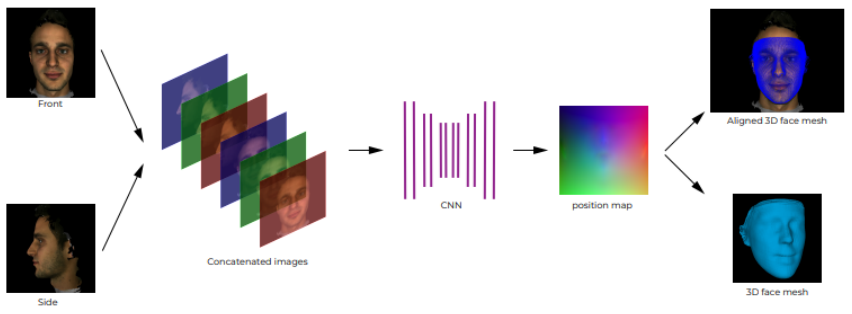
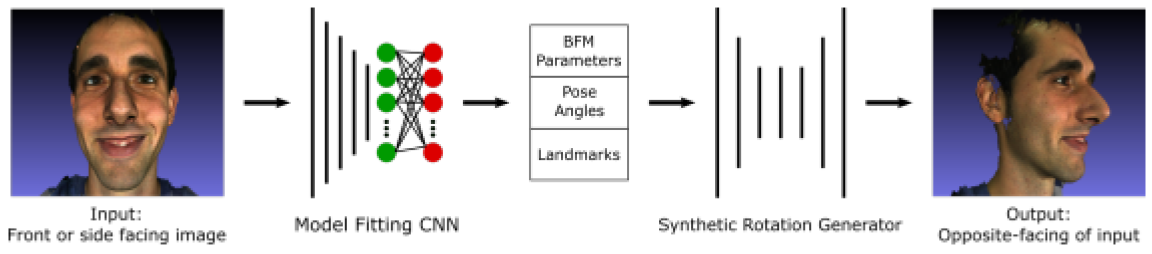
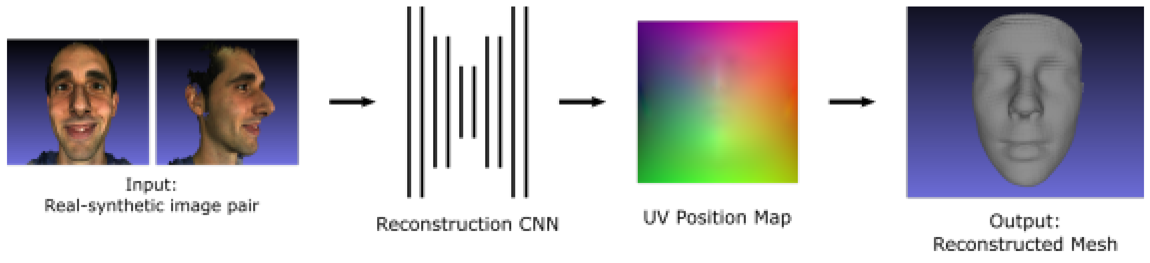


Figure 15: Illustration of network proposed in [3]. Illustration from [3]

3.3 3D Face Reconstruction Based On a Single Input Image



(a) Rotation pipeline



(b) Reconstruction pipeline

Figure 16: Illustration of complete pipeline. Image from [13].

[9]https://facegen.com
[10]http://www.cbsr.ia.ac.cn/users/xiangyuzhu/projects/3DDFA/main.htm

Instead of using two separate input images, [13] implements a second network to synthetically rotate a input image using a technique called rotate and render[14]. This allows then for a single input image to be used, rotated, and then used in the position map predictor. The added rotation network causes this approach to be somewhat slower then previous works.

Although evaluation results did not improve from [3], Yong discusses that one problem is with the dataset. As facegen does not create realistic facial models, as well as there are some issues with realism when it comes to the rotate and render network, having a better dataset would potentially improve the results and potentially improve upon the previously reported NME from [3] and [2].

3.4 3D-R2N2: A Unified Approach for Single and Multi-view 3D Object Reconstruction

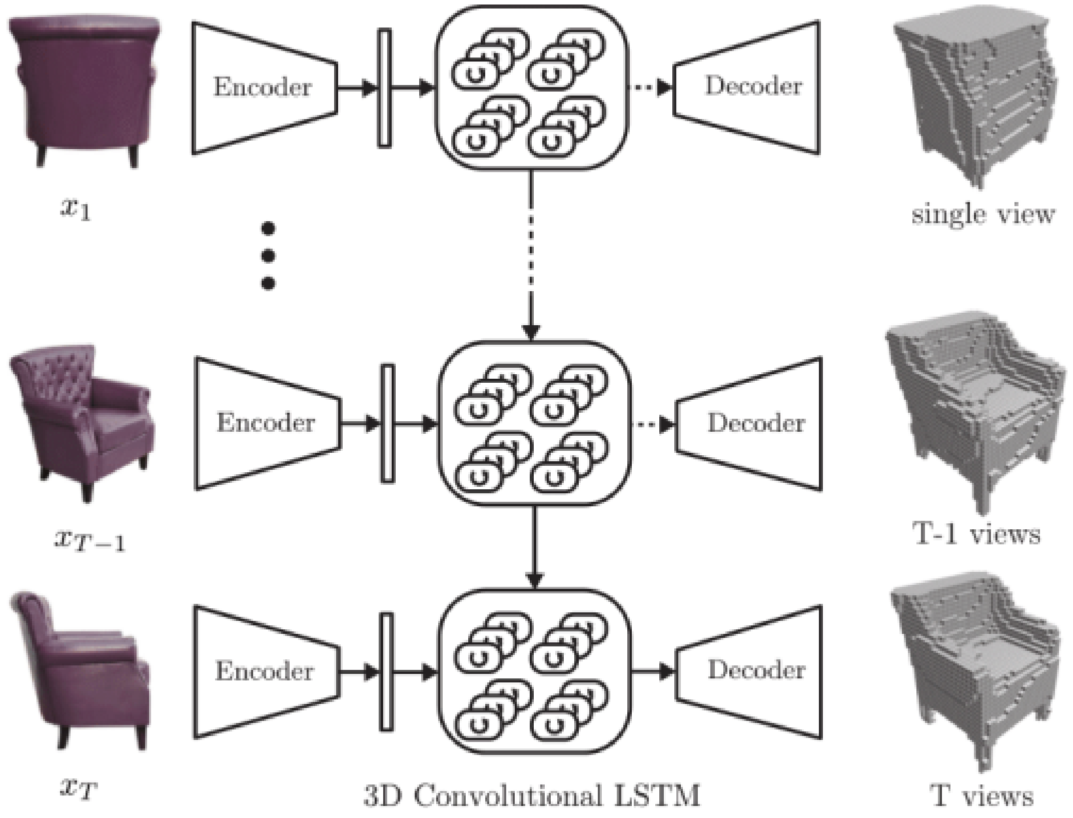


Figure 17: Illustration of the 3D-R2N2 network at work. Illustration taken from [5]. The image shows the model attempting to reconstruct a 3D model of a chair. The more images the model is presented to, the better the results.

In [5], Choy et al. proposed the 3D-R2N2 network in 2016, where they propose a network to reconstruct any 3D object given a sequence of images. This is achieved by using an encoder-decoder architecture with the addition of a 3D LSTM structure between them. This allows for the model to take in a number of images from different angles, improving upon the final voxel grid as more information is presented to the network. This network does not regress UV position maps, but rather performs 3D unpooling to directly predict the final voxel grid for the 3D object. Two models were made in [5]: one using a 3D LSTM structure, and one using Gated Recurrent Units (GRUs).

Although showing promising results, this model reconstructs 3D objects in low quality, with an output dimension of 32x32x32 voxels. This work also is not aimed toward facial reconstruction, but rather at reconstructing the shape of any 3D object, but shows that the use of LSTM cells in reconstruction networks using image sequences is a potentially viable method of reconstruction.

4 Methodology

This work tries to modify [3] and [13] work in order to reconstruct a 3D facial model. Hopefully, the more images presented in the input sequence, the higher the quality of the extracted results. In addition to handling sequential input, this work attempts to not regress towards position maps as before but rather attempts to directly predict the points of a point cloud.

4.1 Pipeline/Architecture

The network architecture is based on the network proposed in [3], with the starting point being the implemented code found on GitHub. The code has been modified through experimentation with different amounts of Convolutional LSTM layers and number of different layers in the encoder and decoder. The model was adapted to handle temporal data by adding TimeDistribution layers and convolutional LSTM layers. The final best-performing model found through this experimentation is outlined in table 1. The input/output shape is given as (T, W, H, C) or (W, H, C) where T is the number of images (timesteps) for each entry in a batch, W and H denote the input spatial dimensions, and C is the number of channels in each image.

Input	**Layer**	**Time Distributed**	**Kernel**	**Stride**	**Output**
8x256x256x3	ConvLSTM2D	-	3	2	8x128x128x32
8x128x128x32	Inverted Residual	Y	3	1	8x128x128x16
8x128x128x32	ConvLSTM2D	-	3	2	8x64x64x24
8x128x128x32	ConvLSTM2D	-	3	2	8x32x32x32
8x128x128x32	ConvLSTM2D	-	3	2	8x16x16x64
8x128x128x32	ConvLSTM2D	-	3	1	8x16x16x96
8x128x128x32	Inverted Residual	Y	3	2	8x8x8x160
8x128x128x32	Inverted Residual	Y	3	1	8x8x8x320
8x128x128x32	ConvLSTM2D	-	3	1	8x8x8x512
8x128x128x32	ConvLSTM2D	-	3	1	8x8x512
128x128x32	Transposed Convolution	-	3	1	8x8x512
128x128x32	Transposed Convolution	-	3	2	16x16x256
128x128x32	Transposed Convolution	-	3	2	32x32x128
128x128x32	Transposed Convolution	-	3	2	64x64x64
128x128x32	Transposed Convolution	-	3	2	128x128x32
128x128x32	Transposed Convolution	-	3	1	128x128x16
128x128x32	Transposed Convolution	-	3	1	128x128x8
128x128x32	Transposed Convolution	-	3	1	128x128x3
128x128x32	Reshape	-	-	-	16384x3

Table 1: Architecture outline

The table is split up into an encoder and decoder part, linked together by a small LSTM block. The initial idea was to keep LSTMs only for this link, but experimentations showed better performance when exchanging some of the inverted residuals with the same Convolutional LSTM layers. The decoder has been reduced to remedy memory restrictions in the hardware. Additionally, the reduction done in the decoder has also contributed to better performance.

The final network from this work changes some of the inverted residuals with pure Keras ConvLSTM2D layers with the corresponding number of output features. Experimentation with architecture shows somewhat improved results in training when using LSTM layers in place of the inverted residuals in some places. Although making for better performance, the inverted residuals are naively replaced, and further work should focus on finding the optimal balance between inverted residuals and LSTM and optimizing parameters for the added LSTM layers instead of using the same parameters as previous residuals.

4.2 Dataset

4.2.1 BU4DFE

The dataset used for this work is the BU4-DFE dataset, courtesy of co-supervisor Antonios Danelakis. The dataset contains a total of 101 different faces. For each face is a total of 6 videos showing different facial expressions, with about 100 frames for each video, resulting in a total of 60.600 images with corresponding 3D facial meshes.

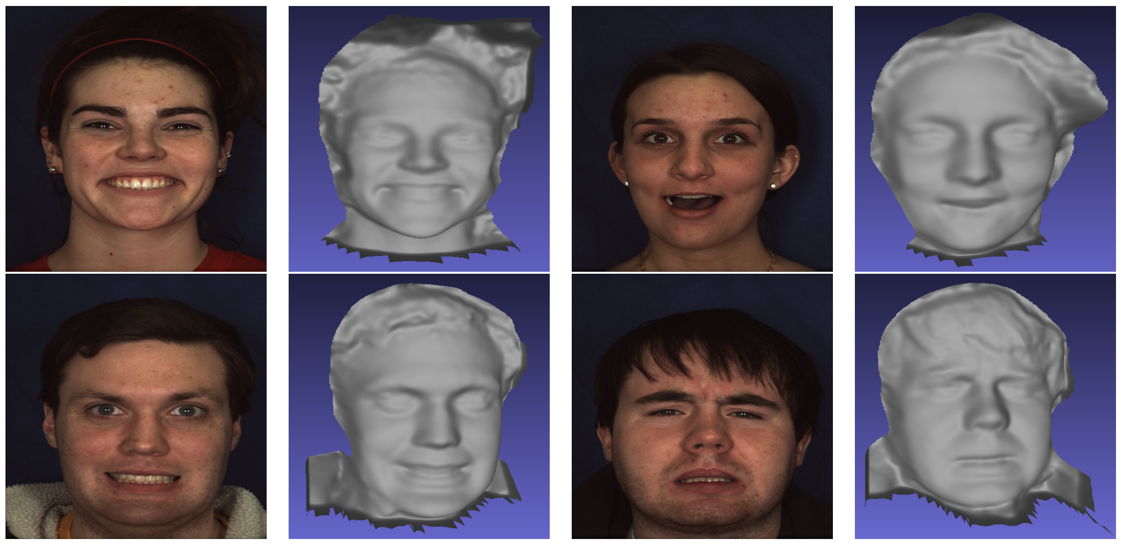


Figure 18: A few example images with corresponding 3D face meshes from the BU4-DFE dataset[15]

The BU4-DFE dataset is a dataset of videos showing facial expressions coupled with a 3D facial model for each frame of video[15]. The dataset consists of 101 subjects, 58 females and 43 males, all of whom are filmed showing 6 distinct facial expressions: Anger, Disgust, Fear, Happiness, Sadness, and Surprise. Each video has around 100 frames, making a total of 60600 images with corresponding 3D face meshes. Although 101 distinct faces in total, a problem occurred after downloading and extracting the ZIP archive containing the dataset as one of the internal archives was corrupted. After multiple downloads to the computer used for this work, the problem persisted. Because of this, faces labeled F27-F31 (Female subjects 27 through 31) could not be extracted properly and used, leaving 96 subjects for this work.

4.2.2 Preprocessing

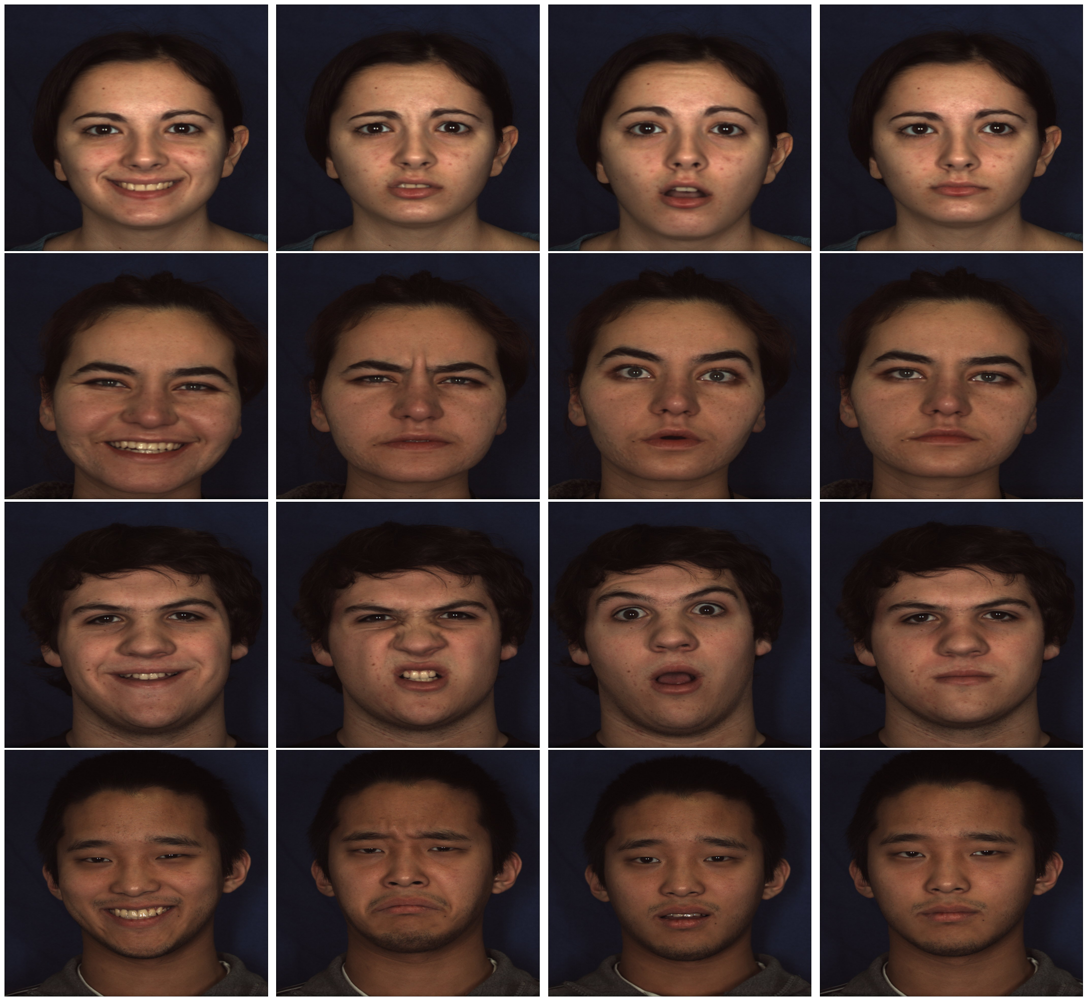


Figure 19: Examples of neutral faces. Columns 1-3 shows different subjects expresing the same emotions, with highly variable results. The fourth columns shows the same subjects with a neutral expression, with less variation in pose and expression. Images from BU4-DFE dataset [15]

As every single frame of video for each subject comes with its own unique facial mesh as ground-truth, the data had to be combed through and select a single ground-truth face mesh for each subject. As facial expressions, and pose to a smaller degree, vary from subject to subject when tasked with making an expression, all subjects have vastly different facial meshes during an emotional display. For the model to generalize and reconstruct different faces better, the data was combed through manually to extract a single neutral front-facing facial mesh for each individual subject. By looking for neutral faces in the images of each subject, a good neutral face was picked and the corresponding 3D mesh was chosen as the ground-truth for this subject. A "neutral" face is here meant a face showing no clear expression and is similar to other ”neutral” faces across all subjects.

As the network is fed images of the same subject, it is desired for the model to construct the same face every time. This allows the model to generalize better and reconstruct the correct face regardless of the expression on display. Selecting similar face meshes with regards to pose and expression across all subjects will increase the network’s ability to generalize, as it is trained to reconstruct the same face with different features, but from the same point of view and expression.

From figure 19 we can see a big variation in expression and pose from each subject expressing the same emotion. To remove some of this variation during training, the neutral faces with a low variation of pose and expression across subjects is chosen as the ground-truth.

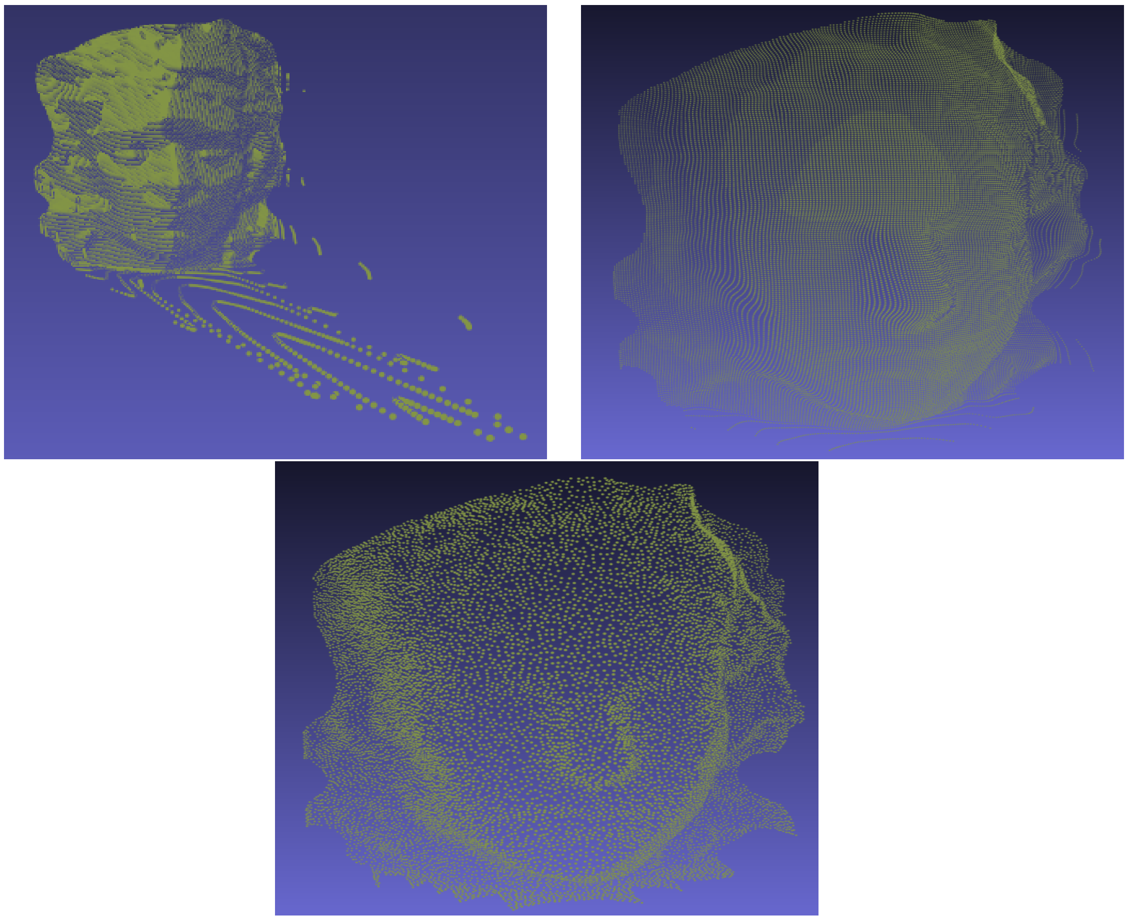


Figure 20: Top left: Raw point cloud with many unreferenced points (noise). Top right: Raw point cloud without unreferenced points. Bottom: Downsampled point cloud used for training.

To let the model predict the positions of the vertices in the ground-truth, the number of points

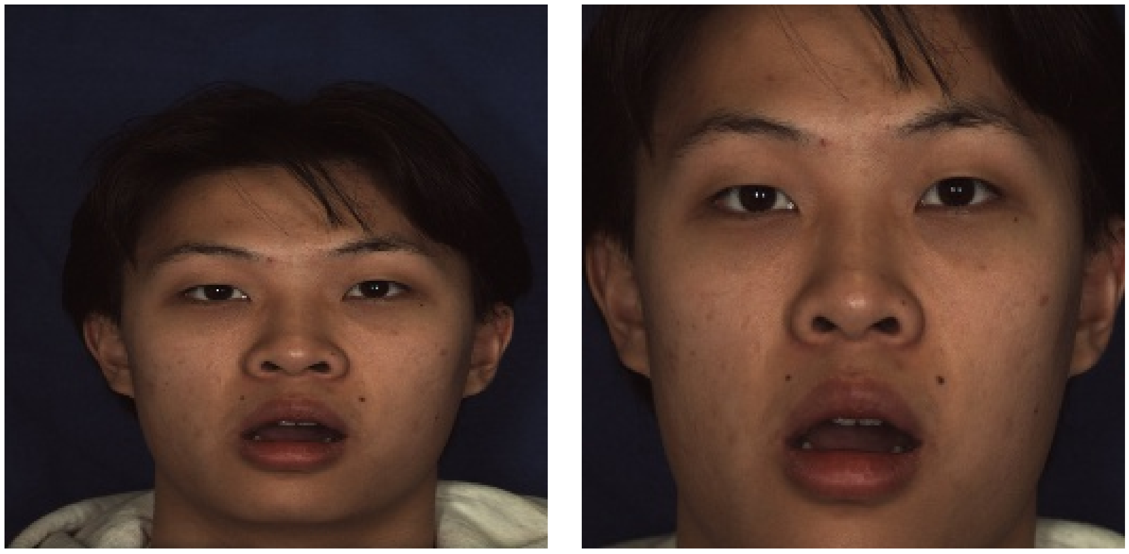


Figure 21: Left: Original image (rescaled to fit on page). Right: Face found by cv2 and rescaled to 256x256.

in the ground-truths would also have to be changed. The number of vertices for all meshes varies vastly, and some points are not referenced at all. Because of this, all ground-truths were processed using the python library open3D to remove unreferenced points. Next, each ground-truth mesh was subsampled down to 2^{14} (16384) points using open3d. This was done to allow the decoder to predict the final result using simple transposed convolutions, upsampling the prediction until the correct number of points are predicted. Manually reviewing the ground-truths after downsampling shows that the ground-truths are still distinguishable and one can see the faces. If the results proves the idea is viable, the point cloud resolution can be increased and adapted for further work.

To save time during training, it was decided to conduct face detection on all images as part of the preprocessing. All images used both during training and evaluation was processed using opencv2 Cascade Classifier to find the face in each image. The face was then isolated, rescaled to size 256x256 and then saved. During training, a datagenerator will construct a batch of batch size B, containing X number of images corresponding to the given number of time steps. Next, the generator will pick X random images of any one subject and the ground-truth of the corresponding subject. The random images are cast to float and normalized to values between 0 and 1 before returning the batch to the training loop.

4.3 Training

For training, the dataset was split into training and testing parts with 70 subjects being used for training and the 26 remaining used for validation each epoch.

For loss calculation we utilize the Mean Squared Error loss function defined by equation 8 which returns the average squared error between all points in the predicted point cloud and their corresponding points in the ground-truth point cloud. Training was done using the Adam optimizer with initial learning rate of 0.0001. The learning rate was halved every 10 epochs, and early stopping was implemented at 30 iterations. Early stopping was initially set to 10 epochs, but due to a problem with overfitting which sadly has not been remedied as of writing, the model stopped training after 2-5 epochs. Due to this, two steps of the training pipeline was changed. First, the early stopping was increased form 10 epochs to 30. Secondly, instead of only comparing the lowest testing losses, we also decided to continue training if the training loss improves before the early stopping limit is reached. This way we could at least look at how the model with improved training loss fairs.

From experiments with batch sizes and number of images, the best combination found thus far seems to be a batch size of 5 with 8 images for each prediction of the model.

Figure 22 shows the loss curve for training and testing for the model defined in table 1. This is the best performing model resulting from experimentation with the architecture.

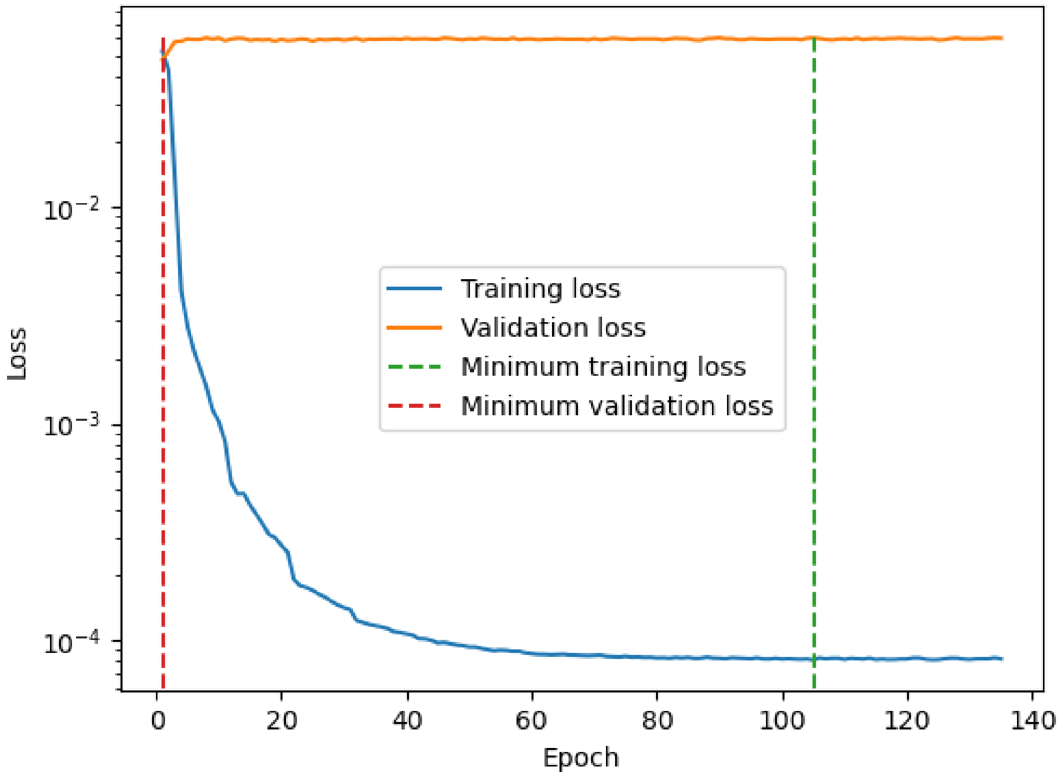


Figure 22: Plot showing training and validation loss during training of the best performing model.

4.4 Discussion

As stated previously, the final model as well as its predecessors has a big problem with overfitting, which is clear from looking at figure 22. This problem could be due to the small dataset used as BU4-DFE only consists of a total of 101 subjects, where 5 were lost during retrieval for this work resulting in a total of 96 different faces. Increasing the number of subjects in the training set could possibly help the model generalize better, but as the current validation set is already very small, the split was kept as is throughout this work. Another potential problem is some ground-truth point clouds having certain features not present in all other ground-truths, such as points representing the subject's hair, shoulders, or collar of a shirt or hoodie. As this is neither present in all ground-truths nor presented within the input images, there is no way for the network to recognize these features. If these deviants are present in the training data, the model will be overfitted to recognize these specific features in the relevant subjects, and these subjects only.

Another point of concern is the reduced decoder. Although showing improved performance during training, it is possible that this could have been one of the causes for overfitting as the more shallow model could have a harder time generalizing data, thus not being able to efficiently reconstruct unseen subjects.

5 Results

5.1 Evaluation Dataset

The dataset used for evaluation is the testing part of the BU4-DFE dataset used during this work. The BU4-DFE dataset has 101 distinct subjects, all of which have approximately 600 facial images across 6 videos making different facial expressions. From these 101, 5 were lost during dataset transfer, leaving us with 96 different subjects. The dataset chosen for evaluation for this work was simply the test data from the BU4DFE dataset. Although small, this should give a good representation of how the trained model is doing on unseen data.

5.2 Evaluation Pipeline

To evaluate the test data, we start by initializing a data generator the same way as during training. This generator is initialized with a batch size of 1, allowing us to iterate through the test data one instance at a time. In each iteration, we extract 8 random images from a single subject, as well as the ground-truth point cloud for the subject. The model makes its prediction based on the input sequence and returns a predicted point cloud. The ground-truth and final prediction are then used together in an Iterative Closest Point (ICP) algorithm to align them before the error is calculated using our evaluation metric. A simple illustration of the evaluation pipeline is presented in figure 23

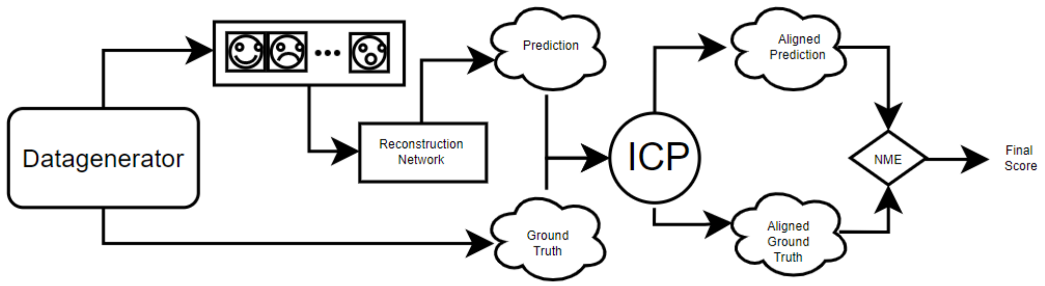


Figure 23: Illustration of the evaluation pipeline.

5.3 Evaluation Metric

The evaluation metric used for this work is Normalized Mean Error (NME). NME calculates the distance from which the prediction deviates from the ground-truth by calculating the squared euclidean distance between all predicted points and their corresponding points in the ground-truth. From there the distances are normalized using a normalization factor, set to the bounding box size of our predictions. Finally, this is averaged by the number of vertices present in the ground-truth and predicted point cloud. NME is defined in equation 13

$$NME = \frac{1}{N}\sum_{i=1}^{N}\frac{||y_i - p_y||_2}{d} \tag{13}$$

5.4 Performance

During training 2 models were saved. One based on validation loss (model weights saved to file each time validation loss improves), as well as one based on training loss (model weights saved to separate file each time training loss improves) due to the network reaching a local minimum on validation loss after 2 5 epochs. Both models were evaluated on the test dataset, and also on the training dataset to see the difference. The NME of the final model is noted in table 2

	Training Model	Validation Model
Training Data	**0.0034183635**	0.0150208668
Testing Data	0.0258034830	0.0155522075

Table 2: Normalized Mean Error for model based on validation loss and training loss on test data and training data.

Unsurprisingly, the best performance is seen from the model based on training loss when evaluating the training data with an NME of 0.0034. However, when evaluating the test data the validation loss model outperforms the training loss model. This is also no surprise since during training we can see from the plot in figure 22 that the validation loss increases as the training loss decrease as is the case when the model overfits to the training data.

For visual reconstruction, the training loss model was used. The reconstructions of the validation loss model were unsurprisingly low as it was only able to train for 5 epochs before it stopped updating. This was the case when performing reconstruction on both test and training data. To get somewhat high-quality results, the training loss model was therefore chosen to present the reconstruction results as it shows the potential of the proposed network. Figure 24 shows images, prediction, and ground-truth for two different subjects from the test data. The results of reconstruction are not of the quality we had initially hoped for, but not surprising given the overfit. Figure 25 however, although overfitted to the training data, shows the potential of the network as it is able to reconstruct high-quality faces at all. This figure shows images, prediction, and ground-truth for 2 subjects from the training data. The reconstructions are here very closely similar to the corresponding ground-truth and given more time to refine the architecture and a bigger more varied dataset we believe the model would be able to better generalize and also make better predictions on unseen data.

5.5 Discussion

The final results were unfortunately not as expected. Overall performance on unseen data results in reconstructions that is not very similar to the ground-truth, or faces at all, as the model is quick to overfit to the training data. However, the reconstructions using training data show high-quality results similar to the ground-truths, showing the potential given further work on architecture.

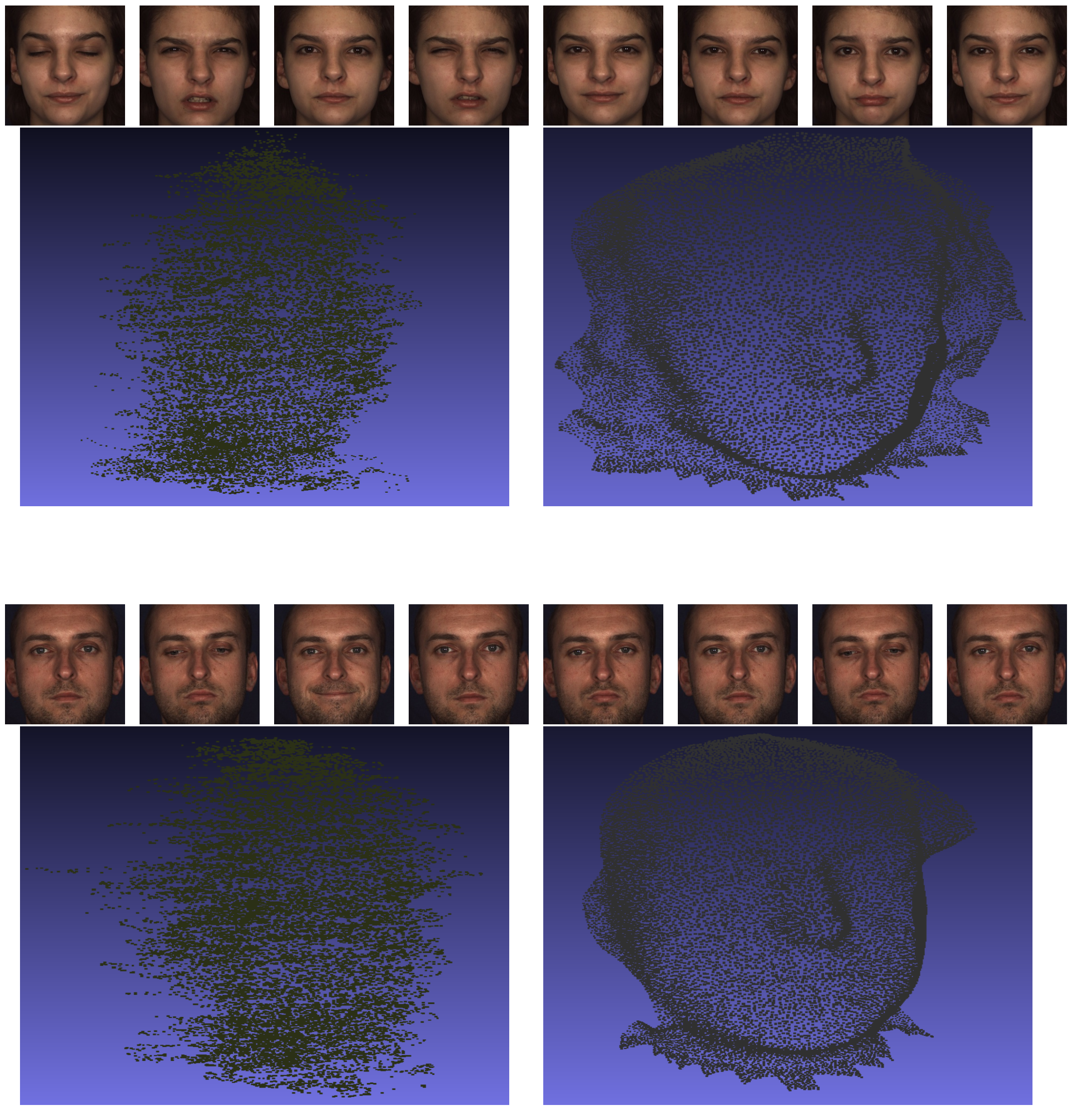


Figure 24: Two sets of demo images generated from subjects from the test set. Top rows: Input face images. Bottom lefts: Trained model prediction. Bottom rights: ground-truth point cloud.

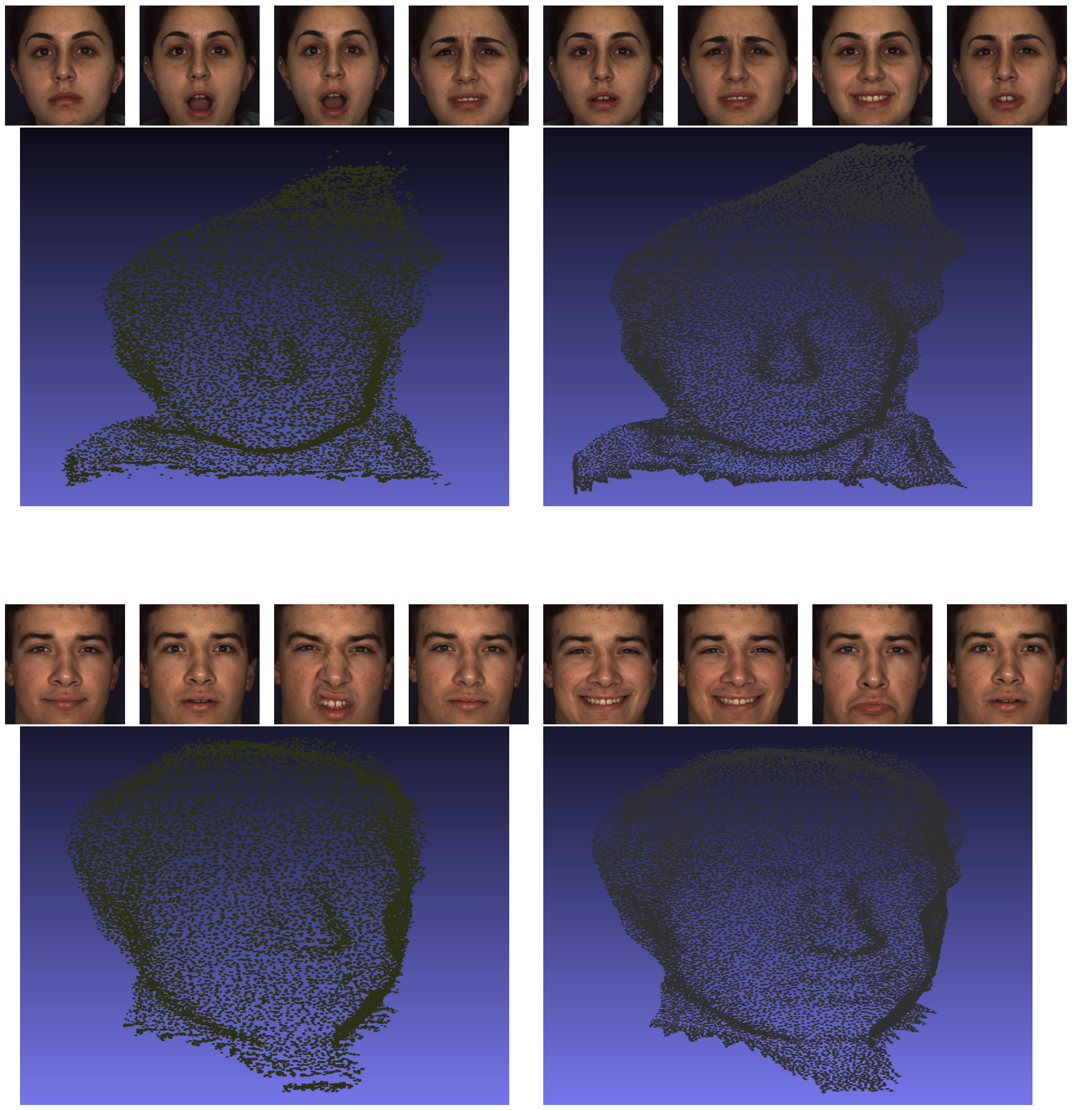


Figure 25: Two sets of demo images generated from subjects from the training set. Top rows: Input face images. Bottom lefts: Trained model prediction. Bottom rights: ground-truth point cloud.

The cause for the overfitting model is most likely due to the low amount of data available in the BU4-DFE dataset and neglecting augmentations during training, which would have increased the diversity in the data and forced the model to rely less on specific features. From image 25 we also see an example where the subject's ground-truth has some outliers. The subject's hair in figure 25 can be seen as a set of points sticking out of the face mesh, something which isn't present in most other ground-truths. Additionally, the ground-truths used for training include more vertices than only those representing the subject's faces. As seen in image 25 it seems that part of the shoulders and collar of the subject's clothes has become part of the point cloud during data generation, which could damage results as it is not present in all ground-truths, and neither are these features present in the input images. A possible solution to this would be finding a dataset where the 3D ground-truth meshes with points strictly representing facial information and not clothes or hair.

Another potential source for the unexpected performance might be the way LSTM has been introduced to the network. Initially, LSTM was used strictly in between the original encoder and decoder, but during experimentation, performance started to improve after changing some inverted residuals for more LSTM layers. This process was done naively however by removing the inverted residual in question and replacing it with a new LSTM layer and keeping the same strides, feature maps, and kernel. In addition to this, the decoder had to be heavily reduced in size as the LSTM layers take up too much memory for the decoder to go unchanged. By optimizing the balance between LSTM and inverted residuals and finding the best layer parameters, performance could see an increase. In addition, the model might be able to generalize better than with the current architecture by also finding a way to increase the complexity of the decoder as a deeper model is more suited for more complex operations such as reconstruction.

6 Conclusion and Further Work

6.1 Conclusion

In conclusion, the network proposed in this thesis shows does not show the initially expected high performance on unseen data. Due to the small dataset and peculiar nature of the ground-truth point clouds where some include features not present in others, the network struggles to generalize and adapt to unseen data. However, from the visual results of the training data, we still believe the addition of LSTM has the potential to improve facial reconstruction. From the reconstructed images from training data, we can clearly see that the network is capable of reconstructing high-quality faces. Because of this, we believe that using the proposed method given further work to improve generalization can introduce an improvement to the current state of 3D facial reconstruction. Further work to improve generalization and performance includes working with an improved dataset with more subjects and larger poses in images and more similar ground-truths, optimizing the relations between LSTM and inverted residuals in the encoder, and expanding the decoder. Further elaboration on this in subsection 6.2.

6.2 Further Work

6.2.1 Optimizing LSTM

When experimenting with different numbers of inverted residuals and LSTM layers, inverted residuals was simplistically removed in favor of more LSTM layers, where the LSTM layer inherited the initial residual block's parameters. This includes the same kernel and stride size, as well as the number of feature maps. As the two different components do not work in the same fashion, further work should experiment more with different values to these parameters to find a combination that works better with the LSTM layers. In addition, experimentation should include positioning of the newly added LSTM layers in the decoder.

Some experimentation was also aimed toward allowing an arbitrary number of input images to be sent as input, but a simple solution to this was not found. Using Keras meant having to specify the exact input dimensions on network initialization, and thus allowing for continuous input to update a single reconstruction was not achieved. Some ways to achieve this might be writing an implementation from scratch, taking variable input dimensions into account, writing a highly customized feed-forward loop that loops through a number of images before returning the final results, or a sequence of generated meshes. Because of time constraints, this was not done during this work.

6.2.2 Improving Decoder

Given the resources needed for the encoder, the decoder had to be downsized for the available hardware to be able to run the model. Further work should focus on finding the optimal relationship between inverted residuals and LSTMs in the encoder, but should also focus on increasing the complexity of the decoder to better decode the given 8x8 feature maps. A way to do this could entail acquiring better hardware or make reductions in the encoder where this does not hurt performance.

6.2.3 More data

The dataset used, BU4-DFE, consists of only 101 different faces, and all looking straight ahead making different expressions. Would like to see work on using a different dataset with more subjects and larger poses, in addition to more similar ground-truths across subjects. The BU4-DFE dataset as shown includes some problematic subjects where the subject's hair and partly clothes are present in the ground-truth. An optimal dataset would therefore be a bigger dataset with more subjects, with images spanning a variety of viewing angles and expressions. In addition, the ground-truths should be of similar orientation or present a simple way of orienting the ground-truths in a similar manner. The ground-truths should strictly contain vertices representing facial features, and not hair or clothes where this would pose a problem such as shown with the top subject in figure 25.

During this work, no transformations (i.e. translations, random rotations, dropout, etc.) were used during training. This should also be considered in later works to increase the variation in training data and increase generalization.

In addition to more data, further work should focus on faces across multiple poses. The BU4-DFE dataset contains only front-facing images, but working with side images and images from a lot of different angles the network could gain more information about a subject by seeing it from different orientations.

6.2.4 Number of images

The original idea was the more images made available to the model, the better the results. This was difficult to experiment with as the more images used during training, the more memory needed by Cuda. For a long time during experimentation, 5 images were the maximum number of images the network could manage before a memory error occurred. After some modifications to the batch size and network architecture, the final model was able to handle 8 images. It would be interesting to either get access to better hardware to increase this image limit and see how much it helps or as discussed above, make the entire model not depend on a set amount of images as iterating over images one at a time, saving the intermittent states would be an even better idea. This could be achieved by writing a customized training loop or implementing the network from scratch keeping this feature in mind.

Appendices

A Installation and Running

A-1 Requiernments

- keras-gpu=2.4.3
- cudnn=7.6.5
- cudatoolkit=10.1.243
- python=3.8
- opencv

A-2 Installation

1. Clone github repo at https://github.com/sindrtho/MasterAssignment2022
2. Install dependencies in new environment in one of two ways
   (a) Install dependencies manually using conda or pip
   (b) Create a new conda environment using the ”spec-file.txt” by running:
   `conda create -n <env-name> --file spec-file.txt`
3. Download and setup the dataset
   (a) Manually choose a single ground-truth facial mesh
   (b) Delete all other .wrl files and save all images of a single subject to the same folder.
   (c) Save the corresponding ground-truth with the images
   (d) Change the image paths in the preprocessing scripts and run these
4. Change path variables in training script to fit with the current dataset
5. run training script by ”`python train.py`” in the root directory

A-3 Directory Structure

```
root/
├── train.py
├── evaluate.py
├── demo.py
├── Network/
│   ├── facedetect.py
│   ├── mobile3model.py
│   └── res/
│       └── haarcascade_frontalface_default.xml
├── Datagenerator/
│   └── dataset.py
└── Dataset/
    ├── Testing/
    │   └── Individual subject folders
    └── Training/
        └── Individual subject folders
```

B Code

B-1 Preprocessing

B-1.1 Facedetection

```
import cv2
import os
import numpy as np

cv2.ocl.setUseOpenCL(False)

# Simple script  facedetection using opencv cascade classifier
def find_faces(img,
 feature_file='Network/res/haarcascade_frontalface_default.xml'):
 DETECTION_PARAMS = feature_file

 detector = cv2.CascadeClassifier(DETECTION_PARAMS)

 found_faces = detector.detectMultiScale(img, 1.1, 4)

 faces = []

 for (x, y, w, h) in found_faces:
  image = img.copy()[y:y+h, x:x+w]
  faces.append(image.copy())

 if len(faces) == 0:
  return None

 return faces
```

B-1.2 Ground-truth adaption

Original ground-truth files in .wrl format. For ground-truth preprocessing there was made a small script taking a intput file (raw ground-truth file) and a path to the output file (the processed ground-truth file) as parameters.
Run with `python processGT.py <input file path> <output file path>`

```
from sys import argv
from os.path import exists
import open3d as opd
import pymeshlab as pm
import numpy as np

if len(argv) < 3:
 print("Not enough arguments. Need input fle and output filename")
 exit(1)
```

```
input_file = argv[1]
output_file = argv[2]

if not exists(input_file):
 print(f"File {input_file} does not exist")
 exit(1)

ms = pm.MeshSet()
ms.load_new_mesh(input_file)
ms.save_current_mesh(output_file, save_textures=False)

mesh = opd.io.read_triangle_mesh(output_file)
mesh.remove_non_manifold_edges()
mesh.remove_unreferenced_vertices()
opd.io.write_triangle_mesh(output_file, mesh)

print(f"Processed file {input_file} to {output_file}")
```

B-1.3 Image preprocessing

```
import cv2
from Network.facedetect import find_faces
import numpy as np
import os
import tqdm
import shutil

# Change to correct path
# PATH = 'Dataset/BU4DFE/'
# TARGET = 'Dataset/BU4DFE/Preprocessed/'

if not os.path.exists(TARGET):
 os.mkdir(TARGET)

# All folders containing facial images and a single ground-truth .ply file
FOLDERS = os.listdir(PATH)

for folder in FOLDERS:
 print(f"Processing folder {folder}")
 images = [_ for _ in os.listdir('/'.join([PATH, folder])) if _.endswith('.jpg')]
 gt = [_ for _ in os.listdir('/'.join([PATH, folder])) if _.endswith('.ply')][0]
 if not os.path.exists('/'.join([TARGET, folder])):
  os.mkdir('/'.join([TARGET, folder]))

 shutil.copy('/'.join([PATH, folder, gt]), '/'.join([TARGET, folder, gt]))
 for image in tqdm.tqdm(images):
  img = cv2.imread('/'.join([PATH, folder, image]))
  img = find_faces(img)[0]
  img = cv2.resize(img, (256, 256))

  cv2.imwrite('/'.join([TARGET, folder, image]), img)
 print(f"Folder {folder} processed\n")
```

B-2 Network Definition

```
from keras import backend as keras_backend
from keras import layers
from keras.layers import Input, Conv2D, Conv2DTranspose, BatchNormalization
from keras.layers import LeakyReLU, Dense, Reshape, Flatten, ConvLSTM2D
from keras.layers import Add, DepthwiseConv2D, Reshape, multiply
from keras.layers import TimeDistributed, Activation, GlobalAveragePooling2D
from keras.models import Model

T = 1
SQUEEZE_AND_EXCITE = True

# Code adapted from existing code from previous works
# Original code attained from github repo of Ola Lium
# "github.com/olalium/face-reconstruction/blob/master/Networks/mobilenet_v2.py"
# Squeeze and excite block implementation from
# "https://github.com/titu1994/keras-squeeze-excite-network/blob/master/se.py"

def squeeze_excite_block(tensor, ratio=16):
 init = tensor
 channel_axis = -1
 filters = init.shape[channel_axis]
 se_shape = (1, 1, filters)

 se = TimeDistributed(GlobalAveragePooling2D())(init)
 se = TimeDistributed(Reshape(se_shape))(se)
 se = TimeDistributed(
  Dense(
   filters // ratio,
   activation='relu',
   kernel_initializer='he_normal',
   use_bias=False))(se)

 se = TimeDistributed(
  Dense(
   filters,
   activation='sigmoid',
   kernel_initializer='he_normal',
   use_bias=False))(se)

 x = multiply([init, se])
 return x

def _make_divisible(v, divisor, min_value=None):
 if min_value is None:
  min_value = divisor
 new_v = max(min_value, int(v + divisor / 2) // divisor * divisor)
 # Make sure that round down does not go down by more than 10%.
 if new_v < 0.9 * v:
  new_v += divisor
 return new_v


def relu6(x):
 """Relu 6
 """
 return keras_backend.relu(x, max_value=6.0)


def _conv_block(inputs, filters, kernel, strides):
 """Convolution Block
 This function defines a 2D convolution operation with BN and relu6.
 # Arguments
  inputs: Tensor, input tensor of conv layer.
  filters: Integer, the dimensionality of the output space.
  kernel: An integer or tuple/list of 2 integers, specifying the
   width and height of the 2D convolution window.
  strides: An integer or tuple/list of 2 integers,
   specifying the strides of the convolution along the width and height.
   Can be a single integer to specify the same value for
   all spatial dimensions.
 # Returns
  Output tensor.
 """

 channel_axis = 1 if keras_backend.image_data_format()=='channels_first' else -1

 x = TimeDistributed(
  Conv2D(filters, kernel, padding='same', strides=strides))(inputs)
 x = TimeDistributed(
  BatchNormalization(axis=channel_axis))(x)
```

```
    return TimeDistributed(Activation(relu6))(x)


def _bottleneck(inputs, filters, kernel, t, alpha, s, squeeze, r=False):
    """Bottleneck
    This function defines a basic bottleneck structure.
    # Arguments
        inputs: Tensor, input tensor of conv layer.
        filters: Integer, the dimensionality of the output space.
        kernel: An integer or tuple/list of 2 integers, specifying the
            width and height of the 2D convolution window.
        t: Integer, expansion factor.
            t is always applied to the input size.
        s: An integer or tuple/list of 2 integers,specifying the strides
            of the convolution along the width and height.Can be a single
            integer to specify the same value for all spatial dimensions.
        alpha: Integer, width multiplier.
        r: Boolean, Whether to use the residuals.
    # Returns
        Output tensor.
    """

    channel_axis = 1 if keras_backend.image_data_format()=='channels_first' else -1
    # Depth
    tchannel = keras_backend.int_shape(inputs)[channel_axis] * t
    # Width
    cchannel = int(filters * alpha)

    x = _conv_block(inputs, tchannel, (1, 1), (1, 1))

    x = TimeDistributed(
        DepthwiseConv2D(kernel, strides=(s, s), depth_multiplier=1, padding='same'))(x)

    x = TimeDistributed(BatchNormalization(axis=channel_axis))(x)

    x = TimeDistributed(Activation(relu6))(x)

    if squeeze:
        x = squeeze_excite_block(x)

    x = TimeDistributed(Conv2D(cchannel, (1, 1), strides=(1, 1), padding='same'))(x)
    x = TimeDistributed(BatchNormalization(axis=channel_axis))(x)

    if r:
        x = Add()([x, inputs])
```

```
 return x


def _inverted_residual_block(
 inputs, filters, kernel, t, alpha, strides, n, squeeze):
 """Inverted Residual Block
 This function defines a sequence of 1 or more identical layers.
 # Arguments
  inputs: Tensor, input tensor of conv layer.
  filters: Integer, the dimensionality of the output space.
  kernel: An integer or tuple/list of 2 integers, specifying the
   width and height of the 2D convolution window.
  t: Integer, expansion factor.
   t is always applied to the input size.
  alpha: Integer, width multiplier.
  s: An integer or tuple/list of 2 integers,specifying the strides
   of the convolution along the width and height.Can be a single
   integer to specify the same value for all spatial dimensions.
  n: Integer, layer repeat times.
 # Returns
  Output tensor.
 """

 x = _bottleneck(inputs, filters, kernel, t, alpha, strides, squeeze)

 for i in range(1, n):
  x = _bottleneck(x, filters, kernel, t, alpha, 1, squeeze, True)

 return x


def decoder_network(inputs):
 x_e = Conv2DTranspose(
  512, 4, strides=1, padding='same',
  activation=keras_backend.relu)(inputs)  # 8 x 8 x 512

 x_e = Conv2DTranspose(
  256, 4, strides=2, padding='same',
  activation=keras_backend.relu)(x_e)  # 16 x 16 x 256

 x_e = Conv2DTranspose(
  128, 4, strides=2, padding='same',
  activation=keras_backend.relu)(x_e)  # 32 x 32 x 128

 x_e = Conv2DTranspose(
  64, 4, strides=2, padding='same',
  activation=keras_backend.relu)(x_e)  # 64 x 64 x 64

 x_e = Conv2DTranspose(
  32, 4, strides=2, padding='same',
  activation=keras_backend.relu)(x_e)  # 128 x 128 x 32

 x_e = Conv2DTranspose(
  16, 4, strides=1, padding='same',
  activation=keras_backend.relu)(x_e)  # 128 x 128 x 16

 x_e = Conv2DTranspose(
  8, 4, strides=1, padding='same',
  activation=keras_backend.relu)(x_e)  # 128 x 128 x 8
 x_e = Conv2DTranspose(
   3, 4, strides=1, padding='same',
   activation=keras_backend.tanh)(x_e) # 128 x 128 x 3

 return x_e


def get_keras_model(input_shape, N=2**14, alpha=1.0, squeeze=False):
 """MobileNetv2
 This function defines a MobileNetv2 architectures.
 # Arguments
  input_shape: An integer or tuple/list of 3 integers, shape
   of input tensor.
  k: Integer, number of classes.
  alpha: Integer, width multiplier, better in [0.35, 0.50, 0.75, 1.0, 1.3, 1.4].
 # Returns
  MobileNetv2 model.
 """
 inputs = Input(shape=input_shape)

 first_filters = _make_divisible(32 * alpha, 8)
 x_e = ConvLSTM2D(
  first_filters, kernel_size=(3,3), strides=(2, 2), padding='same',
  return_sequences=True)(inputs)


 x_e = _inverted_residual_block(
  x_e, 16, (3, 3), t=1, alpha=alpha, strides=1, n=1, squeeze=squeeze)

 x_e = ConvLSTM2D(
  24, kernel_size=(3,3), strides=(2, 2), padding='same',
  return_sequences=True)(x_e)
```

```
 x_e = ConvLSTM2D(
  32, kernel_size=(3,3), strides=(2, 2), padding='same',
  return_sequences=True)(x_e)

 x_e = ConvLSTM2D(
  64, kernel_size=(3,3), strides=(2, 2), padding='same',
  return_sequences=True)(x_e)

 x_e = ConvLSTM2D(
  96, kernel_size=(3,3), strides=(1, 1), padding='same',
  return_sequences=True)(x_e)

 x_e = _inverted_residual_block(
  x_e, 160, (3, 3), t=6, alpha=alpha, strides=2, n=2, squeeze=squeeze)

 x_e = _inverted_residual_block(
  x_e, 320, (3, 3), t=6, alpha=alpha, strides=1, n=1, squeeze=squeeze)

 if alpha > 1.0:
  last_filters = _make_divisible(512 * alpha, 8)
 else:
  last_filters = 512

 x_e = ConvLSTM2D(
  512, kernel_size=(3, 3), strides=(1, 1), padding='same',
  return_sequences=True)(x_e)

 x_e = ConvLSTM2D(
  512, kernel_size=(3, 3), strides=(1, 1), padding='same',
  return_sequences=False)(x_e)

 x_e = decoder_network(x_e)

 x_e = Reshape((N, 3))(x_e)

 model = Model(inputs, x_e)

 return model

if __name__=='__main__':
 model = get_keras_model(input_shape=(5, 256, 256, 3))

 print(model.summary())
```

B-3 Datagenerator

```
import keras
import numpy as np
import os
import cv2
import open3d as opd
from Network.facedetect import find_faces

def point_cloud_normalization(pc):
 center = np.mean(pc, axis=0)
 pc -= center
 furthest = np.max(np.sqrt(np.sum(abs(pc)**2, axis=1)))
 pc /= furthest

 return pc

def origin_translation(pc):
 p = np.array([np.min(pc[:, i]) for i in [0, 1, 2]])
 pc -= p
 return pc

class LSTMDataset_V3(keras.utils.Sequence):
 def __init__(self, path, images=5, batch_size=1, norm=True, N=27508):
  self.N = N
  self.func = point_cloud_normalization if norm else origin_translation
  self.path = path
  self.batch_size = batch_size
  self.n_img = images
  self.__find_images__()
  self.on_epoch_end()

 def __len__(self):
  return int(np.floor(len(self.indicies)/self.batch_size))

 def __getitem__(self, index):
  X, y = self.__data_generation(index)
  return X, y

 # Finds all jpgs as X and single ply file as ground-truth.
 # All different faces in its own sunfolder.
 # Subfolder for subject x contains all images of subject x
 # as well as a single ply file used as ground-truth.

 def __find_images__(self):
  print("Loading dataset.")
  self.item_pairs = []
  for folder in os.listdir(self.path):
   images = [
    '/'.join([self.path, folder, file]) for file
    in os.listdir(self.path+'/'+folder) if file.endswith('.jpg')
    ]

   # Finds the ground-truth .ply file
   ground_truth_file = [
    '/'.join([self.path, folder, file]) for file
    in os.listdir('/'.join([self.path, folder])) if file.endswith('.ply')
    ]

   gt = self.func(np.array(opd.io.read_point_cloud(*ground_truth_file).points))

   self.item_pairs.append((images, gt))

 def on_epoch_end(self):
  inputs = []
  gts = []

  for Xs, Y in self.item_pairs:
   Y = np.expand_dims(Y, axis=0)
  # For each subject, shuffle all images and create batches of size n_image
   for _ in range(5):
    np.random.shuffle(Xs)
   batches = [
    Xs[i:i+self.n_img] for i in range(0, len(Xs), self.n_img)
    if len(Xs[i:i+self.n_img]) == self.n_img
    ]

   for batch in batches:
    inputs.append(batch)
    gts.append(Y)

  self.inputs = inputs
  self.truths = gts

  indicies = [i for i in range(len(inputs))]
  for _ in range(np.random.randint(3, 10)):
   np.random.shuffle(indicies)

  self.indicies = indicies

 def __data_generation(self, index):
  X = np.zeros((self.batch_size, self.n_img, 256, 256, 3), dtype=np.float32)
  Y = np.zeros((self.batch_size, self.N, 3))
```

```
  for batch in range(self.batch_size):
   images = self.inputs[self.indicies[index + batch]]  # Images
   Y[batch] = self.truths[self.indicies[index + batch]] # ground-truth

   for i, entry in enumerate(images):
    image = cv2.imread(entry).astype(np.float32)
    cv2.normalize(image, X[batch, i], 0.0, 1.0, cv2.NORM_MINMAX)
  return X, Y
```

B-4 Training Loop

```
import tensorflow as tf
import numpy as np
from Datagenerator.dataset import LSTMDataset_V3
from keras.layers import Input
from keras.models import Model
from Network.mobile3model import get_keras_model
import tensorflow as tf
import keras
import tqdm
from datetime import datetime as t
import time

physical_devices = tf.config.experimental.list_physical_devices('GPU')
assert len(physical_devices) > 0, "Not enough GPU hardware devices available"
config = tf.config.experimental.set_memory_growth(physical_devices[0], True)

BATCH_SIZE = 5
N_IMAGES = 8
SHAPE = (N_IMAGES, 256, 256, 3)

TRAINING_PATH = 'Dataset/BU4DFE/Training'
TESTING_PATH = 'Dataset/BU4DFE/Testing'

training_generator = LSTMDataset_V3(
 TRAINING_PATH, images=N_IMAGES, batch_size=BATCH_SIZE, N=2**14)
testing_generator = LSTMDataset_V3(
 TESTING_PATH, images=N_IMAGES, batch_size=BATCH_SIZE, N=2**14)

print("Initializing model")

SQUEEZE_AND_EXCITE = True
model = get_keras_model(SHAPE, N=2**14, squeeze=SQUEEZE_AND_EXCITE)

print("Model initialized")
```

```
EPOCHS = 300
lr = 0.0001

opt = keras.optimizers.Adam(learning_rate=lr)
loss = keras.losses.MeanSquaredError()

best_loss = float('inf')
best_train = float('inf')
savepath = 'model.h5'
counter = 0  # Counter for keeping track of non improving epochs
THRESHOLD = 30 # Threshold for early stopping

for epoch in range(EPOCHS):
 total_training_loss = 0.0
 training_NME = 0.0

 start = time.time()
 print("Start epoch %d at %s" % (epoch+1, t.now().strftime('%H:%M:%S')))
 for index in tqdm.trange(0, len(training_generator)):
  X, Y = training_generator.__getitem__(index)

  with tf.GradientTape() as tape:
   logits = model(X, training=True)

   loss_value = loss(logits, Y)
   total_training_loss += float(loss_value)
  grads = tape.gradient(loss_value, model.trainable_weights)
  opt.apply_gradients(zip(grads, model.trainable_weights))

  print(f"Training step {index+1}:\n\tLoss: {float(loss_value)}\n")

 stop = time.time()
 d = stop-start
 total_training_loss /= len(training_generator)

 print(f"""Finished epoch {epoch+1}
  Final loss: {float(total_training_loss)}
  Finished at {t.now().strftime('%H:%M:%S')}""")

 training_generator.on_epoch_end()


 ##### Validating model on validation set #####

 print("Validating:\n")
 validation_loss = 0
```

```
# metric = 0.0

for index in tqdm.trange(0, len(testing_generator)):
 X, Y = testing_generator.__getitem__(index)
 logits = model(X, training=False)
 loss_value = loss(Y, logits)

 validation_loss += loss_value

validation_loss /= len(testing_generator)

print(f"Average validation loss epoch {epoch+1}: {float(validation_loss)}")

testing_generator.on_epoch_end()
##### Writing information to file #####

with open(f'results.txt', 'a') as file:
 file.write(f'Epoch {epoch+1}\n')
 file.write(f'Finished after {d} seconds\n')
 file.write(f'Average training loss: {total_training_loss}\n')
 file.write(f'Average validation loss: {validation_loss}\n\n')


if epoch > 0 and epoch % 10 == 0:
 print(f"Halving learning rate from {lr} to {lr/2.0}")
 lr /= 2.0
 opt.learning_rate = lr

if validation_loss < best_loss:
 print(f"""Loss improved from {best_loss} to {validation_loss}.
 Saving model to {savepath}""")

 best_loss = validation_loss
 model.save(savepath)
 counter = 0

# Have not been able to keep the model from overfitting
# Save additional model based off of training loss
elif total_training_loss < best_train:
 print("Training loss improved. Saving train model")
 counter = 0
 best_train = total_training_loss
 model.save("trainmodel.h5")
else: # Early stopping if loss does not improve in THRESHOLD epochs
 counter += 1
 if counter >= THRESHOLD:
  print(f"""Early stopping at epoch {epoch+1}.
  Loss not improved in {THRESHOLD} epochs.""")
  exit(0)
```

B-5 Demo Code

```
import cv2
import numpy as np
import keras as k
from Network.facedetect import find_faces
import matplotlib.pyplot as plt
import open3d as opd

cap = cv2.VideoCapture(0)

# Replace IMAGES_PATH with path to dataset used for demonstration.
# Takes in path to single subjects images, NOT dataset root directory.
IMAGES_PATH = 'Dataset/BU4DFE/'
IMAGES_PATH = [
 '/'.join([IMAGES_PATH, f]) if f.endswith('.jpg') for f in os.listdir(IMAGES_PATH)
 ]

# Replace MODEL_PATH with path to saved model used for demo.
#If model uses more or less images, change N
MODEL_PATH = 'Path to model'
N = 8

IMAGES = [IMAGES_PATH[i]
 for i in np.random.choice(len(IMAGES_PATH-1), N, replace=False)]

frames = [cv2.imread(f'Dataset/Disgust/F001/{image}.jpg') for image in IMAGES]
input_images = []

for frame in frames:
 faces = find_faces(frame)

 face = faces[0]

 input_images.append(faces[0])

batch = np.zeros((1, N, 256, 256, 3), dtype=np.float32)

for i, face in enumerate(input_images):
 face = face.astype(np.float32)
 face = cv2.resize(face, (256, 256))
 cv2.normalize(face, batch[0, i], 0, 1, cv2.NORM_MINMAX)
```

```
    """ Uncomment 3 next lines to overview the selected images """
    # cv2.imshow('face', batch[0, i])
    # cv2.waitKey(0)
    # cv2.normalize(face, batch[0, i], 0, 1, cv2.NORM_MINMAX)

cv2.destroyAllWindows()

model = k.models.load_model(MODEL_PATH)
model.compile()

preds = model.predict(batch)[0]

pc = opd.geometry.PointCloud()
pc.points = opd.utility.Vector3dVector(preds)
opd.io.write_point_cloud('demo.ply', pc)
```

B-6 Iterative Closest Point and NMSE

The Normalized Mean Squared Error implementation used for evaluation during this work is the same used by Lium in [3] with some slight modifications to adapt it to the current work. The original code can be found here[11].
The ICP implementation used is the same used in [3]. The original implementation can be found at
https://github.com/ClayFlannigan/icp/blob/master/icp.py

B-7 Evaluation

Run
`python evaluate.py <batch size> <# of images> <data path> <model path>` with optional parameter `[results file]` if results are to be written to file.

```
from sys import argv
import tensorflow as tf
import numpy as np
from Datagenerator.dataset import LSTMDataset_V3
import keras as k
from keras.layers import Input
from keras.models import Model
from Network.mobile3model import get_keras_model, relu6
import tensorflow as tf
import keras
import tqdm
from datetime import datetime as t
import time
```

[11] https://github.com/olalium/face-reconstruction

```
from Evaluation.evaluate_predictions import prediction_evaluater
from Evaluation.evaluate_predictions import apply_homogenous_tform
from Evaluation.icp import icp

if __name__=="__main__":
 physical_devices = tf.config.experimental.list_physical_devices('GPU')
 assert len(physical_devices) > 0, "Not enough GPU hardware devices available"
 config = tf.config.experimental.set_memory_growth(physical_devices[0], True)

 if len(argv) < 5:
  print("""please run script with parameters: batch_size, n_images,
  data_dir, model_weights, [result_file]""")
  exit(1)

 BATCH_SIZE = int(argv[1])
 N_IMAGES = int(argv[2])
 PATH = argv[3]
 MODEL = argv[4]


 filename = None
 if len(argv) > 5:
  filename = argv[5]
  print(filename)

 model = k.models.load_model(MODEL, custom_objects={'relu6':relu6})
 model.compile()

 loss = keras.losses.MeanSquaredError()
 evaluation_metric = prediction_evaluater()

 testing_generator = LSTMDataset_V3(
  PATH, images=N_IMAGES, batch_size=BATCH_SIZE, N=2**14)

 total_loss, metric = 0.0, 0.0

 for index in tqdm.trange(0, len(testing_generator)):
  X, Y = testing_generator.__getitem__(index)
  logits = model(X, training=False)
  loss_value = loss(Y, logits)

  total_loss += loss_value

  metric += evaluation_metric(np.array(logits[0]), np.array(Y[0]))
```

```
total_loss /= len(testing_generator)
metric /= len(testing_generator)

print(f"""Average evaluation loss: {float(total_loss)}
 Average NME: {metric}\n\n""")

if filename:
 with open(filename, 'w') as file:
  file.write(f'Average loss: {float(total_loss)}\nAverage NME: {metric}')
```

Bibliography

[1] David Zhang and Guangming Lu. *3D biometrics: Systems and applications.* Jan. 2013. Chap. 1, pp. 3–17. ISBN: 978-1-4614-7399-2. DOI: 10.1007/978-1-4614-7400-5.

[2] Yao Feng et al. "Joint 3D Face Reconstruction and Dense Alignment with Position Map Regression Network". In: *ECCV*. 2018.

[3] Ola Lium. "3D Facial Reconstruction from Front and Side Images." In: (2020). URL: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2777587.

[4] Mark Sandler et al. "Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation". In: *CoRR* abs/1801.04381 (2018). arXiv: 1801.04381. URL: http://arxiv.org/abs/1801.04381.

[5] Christopher B. Choy; Danfei Xu; JunYoung Gwak; Kevin Chen; Silvio Savarese. "3D-R2N2: A Unified Approach for Single and Multi-view 3D Object Reconstruction." In: (2016).

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. Chap. 9, pp. 326–366. URL: http://www.deeplearningbook.org.

[7] K. He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90. URL: https://doi.ieeecomputersociety.org/10.1109/CVPR.2016.90.

[8] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[9] Andrew Howard et al. "Searching for MobileNetV3". In: *CoRR* abs/1905.02244 (2019). arXiv: 1905.02244. URL: http://arxiv.org/abs/1905.02244.

[10] Jie Hu, Li Shen, and Gang Sun. "Squeeze-and-Excitation Networks". In: *CoRR* abs/1709.01507 (2017). arXiv: 1709.01507. URL: http://arxiv.org/abs/1709.01507.

[11] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. "Open3D: A Modern Library for 3D Data Processing". In: *arXiv:1801.09847* (2018). URL: http://www.open3d.org/.

[12] Pascal Paysan et al. "A 3D Face Model for Pose and Illumination Invariant Face Recognition". In: *2009 Sixth IEEE International Conference on Advanced Video and Signal Based Surveillance*. 2009, pp. 296–301. DOI: 10.1109/AVSS.2009.58.

[13] Yong Bin Kwon. "3D Face Reconstruction Based On a Single Input Image." In: (2021). URL: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2831554.

[14] Hang Zhou et al. "Rotate-and-Render: Unsupervised Photorealistic Face Rotation from Single-View". In: *CoRR* abs/2003.08124 (2020). arXiv: 2003.08124. URL: https://arxiv.org/abs/2003.08124.

[15] Lijun Yin et al. "A high-resolution 3D dynamic facial expression database". In: *2008 8th IEEE International Conference on Automatic Face Gesture Recognition*. 2008, pp. 1–6. DOI: 10.1109/AFGR.2008.4813324.