Master's thesis

2020

Ola Lium

Master's thesis

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

Ola Lium

3D Facial Reconstruction from Front and Side Images

June 2020

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

3D Facial Reconstruction from Front and Side Images

Ola Lium

Master of Science in Computer Science

Submission date:	June 2020
Supervisor:	Theoharis Theoharis, IDI
Co-supervisor:	Antonios Danelakis, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

Being able to reconstruct 3D faces from 2D images is useful for a variety of Computer Vision branches, such as Face Analysis and Face Recognition. Recent advancements in the Computer Vision field has enabled the use of CNNs to produce good 3D facial reconstructions. The Position map Regression Network (PRN) is a recent method which produces convincing 3D faces from 2D images using a CNN. PRN uses a single facial image as input and predicts a UV position map, containing the aligned 3D positions from a 3D face. By building on the works made with PRN this thesis proposes a new method which produces 3D faces from *two images*, one front and one side. The method uses a network architecture similar to the PRN network architecture, but is modified to fit two input images and uses more modern CNN components. The proposed CNN is trained on both synthetic and real data. The synthetic data is generated using a synthetic facial generation software. We show that the proposed network is able to predict faces in the MICC Florence dataset with greater accuracy than PRN.

Sammendrag

Å kunne rekonstruere 3D modeller av ansikter fra 2D bilder er nyttig innenfor biometrisk ansiktsgjenkjenning. Nylige fremskritt innen datasyn og dyp læring har muliggjort bruk av nevrale nettverk for å generere rekonstruksjoner av ansikt fra bildedata. En metode som bruker et nevralt nettverk for å rekonstruere 3D ansikt er en metode kalt Position map Regression Network (PRN) [1]. Vi skal i denne avhandlingen bygge videre på arbeidet gjort med PRN og foreslår en ny metode for rekonstruksjon av ansikt fra bildedata. Vår metode bruker to ansiktsbilder, et foran og et fra siden, for å rekonstruere et ansikt. Et sentralt element i vår metode er det nevrale nettverket. For å trene dette nettverket bruker vi både syntetisk og ekte data. Den syntetiske dataen er generert ved hjelp av programvare spesialisert i syntetisk ansiktsgenerering. Vår metode rekonstruerer ansikter fra MICC Florence datasettet med større nøyaktighet enn PRN.

Preface

This master thesis is the result of the work performed over the course of the spring semester 2020 carried out at the Department of Computer and Information Science (IDI), at the University of Science and Technology (NTNU).

I want to thank Theoharis Theoharis and Antonios Danelakis for their knowledgeable guidance and enthusiasm during this spring semester. Secondly I would like to thank Anna Emilie and Håkon for proofreading the thesis. Finally I would like to thank Ninni for helping me with the graphical components in this paper.

Contents

List of Tables

List of Figures

Glossary

300W-LP A large pose (LP) version of the 300 Faces In-the-Wild Challenge (300W) dataset. viii, 1, 20, 23, 24, 26, 30, 33, 39–41

3DMM 3D Morphable Face Models. 13, 14, 18, 20

BFM Basel Face Model. 13, 14, 20, 24–26

CED Cumulative Error Distribution. 36, 38, 39

CNN Convolutional Neural Network. vi, viii, 1, 3, 4, 7, 22, 30, 31, 39–43

CSS Canonical Screen Space. 15, 16, 28

ECS Eye Coordinate System. 15, 28

face3d Python library for processing 3D face models. viii, 23, 24, 26

FaceGen 3D face-generating software. vii, viii, 1, 14, 15, 23, 25–30

Florence MICC Florence 3D face dataset. vii, viii, 1, 2, 12, 21, 34–37, 39, 40, 42

ICP Iterative Closest Point. 17, 35, 40

position map Representation of 3D points in UV space where the RGB components are typically used to store XYZ positions. vii, viii, 3, 16–20, 22–24, 28, 30, 34, 35

PRN Position map Regression Network. vi–viii, 1, 18, 19, 22, 30, 36–41, 43

ReLU Rectified Linear Unit. 9, 31

SAM Statistical Appearance Model used in FaceGen. 14

SCM Statistical Color Model used in FaceGen. 14

SSM Statistical Shape Model used in FaceGen. 14, 25, 26

VCS Viewport Coordinate Space. 15, 16, 28

Chapter 1

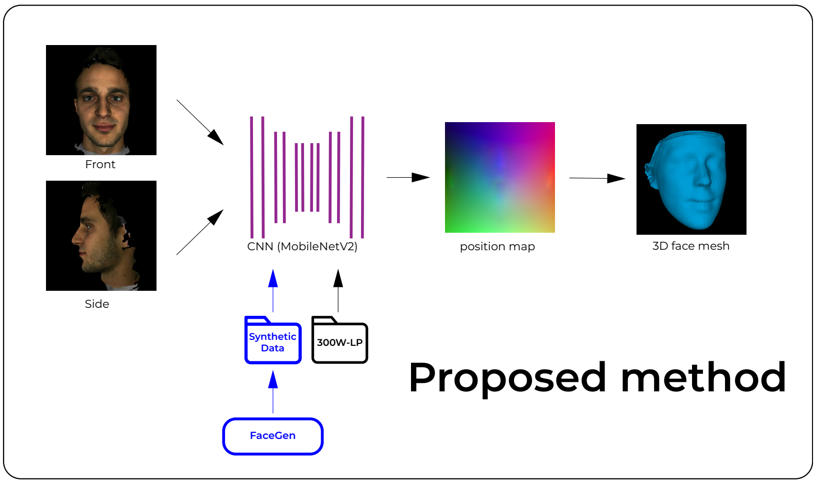
Introduction

Biometric recognition and analysis from 3D facial images is inherently advantageous compared to 2D facial images, as it does not suffer from pose and illumination variations. However, many existing databases consist of only one or more 2D facial images. By accurately reconstructing 3D faces from 2D images we eliminate the need for 3D imaging technology and can make use of the larger 2D facial image databases. Reconstructing 3D faces from 2D images is therefore an important computer vision problem.

By applying recent deep learning techniques several methods have been proposed to solve this problem. A method proposed in [1] has had good success compared to other methods with reconstructing 3D faces from a single input image by utilizing *position maps* to record facial shapes. The method is called Position map Regression Network (PRN) and applies an end-to-end Convolutional Neural Network (CNN) to predict position maps from a single input facial image.

The goal of this thesis is to reconstruct 3D faces from front and side images. By expanding on a previous single image method and utilizing synthetic data we aim to reconstruct 3D faces with greater accuracy. We propose a new method which builds on the work in [1]. The proposed method uses a CNN to map *two* facial images, one front and one side, into a position map. The new CNN is also fitted with more modern network components. The backbone ResNet [4] encoder network in PRN is replaced with inverted residuals components from MobileNetV2 [5]. The proposed method is trained on both synthetic and real data to further increase the performance. For the generation of the synthetic data the FaceGen[1] tool was used, while for the real data, the 300W-LP [7] was recruited. The differences between the proposed method and PRN are outlined in figure 1.1. To assess and compare our proposed method to the PRN, we test the networks on the MICC Florence Dataset [6] by introducing an evaluation pipeline which aligns and calculates the facial reconstruction accuracy.

[1]https://facegen.com/



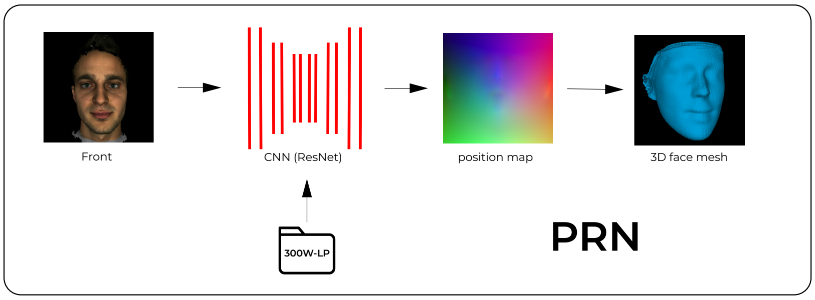


Figure 1.1: Outline of the proposed method (top), compared to the PRN (bottom). The CNN in the proposed method is trained on both synthetic and real data.

1.1 Structure of Thesis

The thesis is structured as follows:

Chapter 1 introduces the work in this thesis.

Chapter 2 covers the necessary background theory.

Chapter 3 describes relevant works and datasets for 3D facial reconstruction.

Chapter 4 contains the proposed method implementation.

Chapter 5 presents the evaluation pipeline and results on the MICC Florence dataset.

Chapter 6 provides the conclusion and outline further work.

Appendix A lists relevant code from our implementation.

Appendix B contains the installation manual for our prototype.

Chapter 2

Background

This chapter covers the necessary background theory to understand the methodology described later in this thesis. Only relevant theory will be covered. To gain a further insight the reader is encouraged to examine referenced sources.

Section 2.1 covers the basic concepts of Convolutional Neural Networks(CNNs)
Section 2.2 examines two relevant CNN architectures
Section 2.3 covers relevant data augmentations for image processing
Section 2.4 looks at example uses of synthetic data
Section 2.5 introduces generative models for faces
Section 2.6 describe the FaceGen software
Section 2.7 defines relevant matrix transformations
Section 2.8 covers UV mapping and UV Position mapping
Section 2.9 give a description of a point cloud alignment algorithm

2.1 Convolutional Networks

If the reader is not familiar with the biological background of Neural networks and the basic Artificial Neural Network perception the reader is encouraged to read Nielsen[8] or Goodfellow et al. [2, p. 164-224]. Unless explicitly stated otherwise, the theory in this section is from Goodfellow et al. [2].

Convolutional networks, also known as convolutional neural networks (CNNs) are neural networks which contain at least one convolutional layer. Typically a CNN contains one or more convolutional layers interspersed with pooling layers and one or more fully connected layersin the end. The following sections 2.1.1-2.1.11 detail the CNN basics.

2.1.1 Convolution operator

The name *convolutional networks* comes from the mathematical operation which these networks use, namely convolution. The convolution operation can be defined as an operation on two functions x and w of a real-valued argument t.

$$s(t) = \int x(a)w(t-a)da. \tag{2.1}$$

The operation is typically denoted with an asterisk: $s(t) = (x * w)(t)$. As the data in computer applications usually are discrete, we define a discrete convolution:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a). \tag{2.2}$$

the x in equations 2.1 and 2.2 is, in CNN terminology, referred to as the input, while the w is called the kernel. The output can be referred to as feature maps. The input in computer applications is usually a multidimensional array of data, while the kernel is usually a multidimensional array of parameters, or weights. The weights are what the learning algorithm is adapting. Assuming that the functions are zero everywhere but in the finite set of point values, the infinite summation can be replaced by a summation of a finite number of array elements. Additionally convolutions are often used over more than one axis at a time, for example over a two-dimensional image. With a two-dimensional image I input and a two-dimensional kernel K the convolution is defined as:

$$s(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n). \tag{2.3}$$

As convolution is commutative, 2.3 is equivalent to 2.4.

$$s(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n). \tag{2.4}$$

The commutative property occurs because the kernels are flipped relative to the input. This way the input index increases with m as the kernel index decreases. In practice a more commonly used function is the cross-correlation function. The function is the same as a convolution, but without flipping the kernel:

$$s(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n). \tag{2.5}$$

An example of a cross-correlation can be found in figure 2.1. As many neural network libraries implement the cross-correlation function, these functions are not differentiated any further and both are referred to as convolutions.

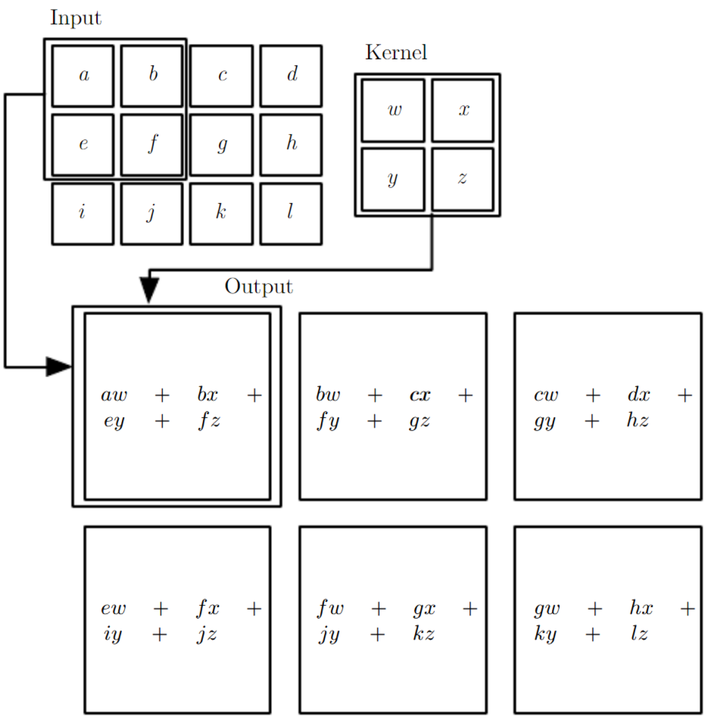


Figure 2.1: Figure from Goodfellow et al. [2] showing an example of a cross-correlation, with input, kernel and output

2.1.2 Fully Connected Layer

The traditional feedforward neural networks, also called Multi-layer perceptrons (MLPs), predict a category based on an input using perceptrons. The input is passed through layers of perceptrons to approximate a function mapping the input to the output. Increasing the number of layers allows the network to approximate more complex functions. Each of the outputs from the previous layer is passed through every perceptron in the following layer. The traditional feedforward neural network uses fully connected layers to predict categories from an output, while CNNs contain at least one convolutional layer.

2.1.3 Convolutional Layer

Convolutional layers use convolutions to compute their output. The convolutional operation entails three beneficial properties, namely sparse connectivity, parameter sharing and equivariance to translation.

Sparse connectivity

In fully connected layers every input unit interacts with every output unit by a matrix multiplication. However, in convolutional layers the convolution kernels are smaller than the input, each output unit is then only connected to a subset of input units specified by the kernel. This causes sparse connectivity. When passing an image through a convolutional layer the input image might be thousands or millions of pixels, while the kernel which can detect meaningful image features might be tens or hundreds of pixels. The result of sparse connectivity is a reduced model size and fewer mathematical operations.

Parameter Sharing

In a fully connected layer each weight is used only once when computing the output, while in convolutional layers each weight is used multiple times. This is called parameter sharing and further reduces the storage needed by the network. As each kernel is used on every input unit, the learning algorithm has to learn far fewer weights.

Equivariance to translation

Convolutional layers also have the property of equivariance to translation. The particular form of parameter sharing found in convolutional layers means that any translation changes in the input causes the same translation in the output. This property is very useful when detecting edges in an image input, as the edge location is simply translated together with the image, if the input image changes position.

2.1.4 Pooling layer

Typical convolutional network layers consists of three stages. At the first stage, the layer performs several convolutions to produce a set of linear activations. The output of the linear activations are then passed through a nonlinear activation function. For the third and final stage of a typical convolutional network layer we use some sort of pooling function to further modify the output from the nonlinear activation function.

The pooling function produces a summary statistic of nonlinear activation outputs. An example of a pooling function can be the max pooling function. This function simply reports the maximum nonlinear activation output within a set neighbourhood.

2.1.5 Transposed Convolutional Layer

A transposed convolution is a transposed convolution operation. A transposed convolutional layer is similar to a convolutional layer, but uses transposed convolution matrices to calculate its output. Transposed convolutional layers usually use feature maps predicted by a neural network to predict an aspect of the input image that produced the feature maps.

Transposed convolutional layers are for instance used in deconvolutional networks to predict the original input image from a set of feature maps [9].

2.1.6 Training a CNN

The weights of CNNs are trained similarly to fully connected neural networks. After calculating the network prediction loss by the selected cost function, typically the squared loss, the update gradient is calculated and passed backwards through the network through backpropagation.

2.1.7 Transfer Learning

Using a previously trained network to initialize the weights or predict features relevant for another network is called transfer learning. If a prediction application lacks labeled training data or want to make use of a large dataset transfer learning is applicable. Transfer learning is particularly relevant in image processing problems as datasets can contain millions of pictures [10], and labeled data can be hard to come by. There are two main transfer learning approaches, using a pretrained CNN as a feature extractor and fine-tuning a pretrained CNN [3].

By fine-tuning a pretrained CNN the classification layers of the pretrained CNN is replaced and trained on new data relevant to a new problem. Some of the upper layers of the pretrained network might be freezed to reduce overfitting. Using a lower learning rate is crucial to limit big gradient updates in the pretrained layers. If we want to train a pretrained network on a large dataset fine-tuning can be an effective approach [3].

2.1.8 Generalization, Overfitting and Underfitting

The goal of a CNN is to perform well on new, previously unseen relevant inputs. The ability to do well on unseen inputs is called generalization. Observing the generalization is simply done by labeling a percentage of the input data as validation data, on which the network never optimizes, but simply evaluates. When the network is unable to further lower the error cost of the validation data, or validation loss, the network should not optimize any further.

A network is overfitted if the network has low error on the training data, but has a high error on new unseen data. Overfitting occurs when the gap between these loss values are too large. If the network is unable to find a low error value on the training error, the network is suffering from underfitting. By increasing or reducing the number of layers, and network parameters, we may control the capacity of the network. This capacity may affect the networks likelihood of overfitting or underfitting.

2.1.9 CNN Hyper Parameters

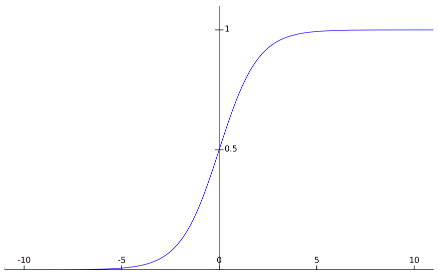
This section details relevant hyper parameters for the network implementation later in this thesis 4.3.2. Hyper parameters are the variables defining the network architecture and learning parameters. Selecting the best hyper parameters for a neural network is not a trivial problem, it is done through experimentation and intuition. An approach to find the best parameters can be through a grid search. A grid search manually or automatically trains the target network with all relevant hyper parameter combinations and select the best one. To save resources and time, an alternative approach is to first set the hyper parameters based on experience and intuition, and then grid search the hyper parameters you see most relevant.

Network Architecture

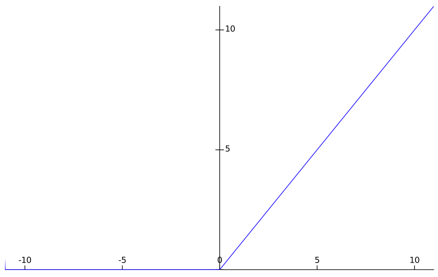
When deciding on the architecture of the network the appropriate number of units and the number of layers is important. Increasing the number of layers in CNNs generally produce better result, but at a certain depth the gain of prediction accuracy drops as the network is harder to optimize. Also, increasing the depth of the network increases the number of weights to optimize, which will affect the performance of the network. The size of the different layers also affects the network's ability to learn. To maximize the network generalization it is important to find the best ratio between depth and size. The depth and size of the network is found through experimentation and careful monitoring of the validation loss result. It is also useful to use previous successful network architectures to find inspiration when creating a new network.

Activation Functions

Activation functions define the output value of a unit from it's inputs. We will go through the relevant two activation functions relevant for the network implementation later in this thesis. Plots of the activation functions is found in figure 2.2



(a) Sigmoid



(b) ReLU

Figure 2.2: Plots of activation functions from the CS231n webpage[3]

Sigmoid

The Sigmoid activation function is defined as $\sigma(x) = 1/(1+e^{-x})$. The function takes a numerical input at squashes it into a number between 0 and 1. Large negative numbers become 0 while large positive numbers become 1. One issue with the sigmoid function is that it saturate and kill gradients, as the gradient for 0 or two is almost zero. The sigmoid function is also not zero centered, causing undesirable learning behaviour [3].

ReLU and Leaky ReLU

A commonly used activation function in recent years is the Rectified Linear Unit (ReLU). It is defined by the function $f(x) = max(0,x)$. The activation is 0 if the input value is below 0, and the same as the input otherwise. It has been found to greatly increase learning compared to the sigmoid function, and is computationally inexpensive. A problem with the ReLU function is dead units. If the input to a ReLU unit is too great, the unit may never be able to update the unit again [3].

A proposed solution to the dead ReLU unit problem is the Leaky ReLU. Instead of returning zero when the input is smaller then 0, the network instead outputs a small negative slope. This is thought to relieve the dead ReLU unit problem, but results have varied [3].

Convolution Kernels

For each network layer, the number of kernels and their size needs to be specified. Increasing the kernel sizes increases the capacity of the network, but in turn increases the storage requirements of the network. Balancing the size and capacity of the network is important to improve network generalization [11].

When applying the specified kernels to the input of each layer, the stride of the kernels needs to be specified. In the example in figure 2.1, the stride is 1, as the kernel moves one space for each convolution. To reduce the output size, the stride can be increased, for example by moving the kernel 2 steps over the input for each kernel operation. Using a stride of 2 in the example in figure 2.1 would produce 4 outputs instead of 6. However, increasing the stride might discard information as some inputs are not covered by the kernel. This is useful if the input is of too high resolution and the goal is to down sample the input data. Selecting the correct stride for a network layer is important to down sample the input data when relevant.

A problem when applying convolution kernels is that some information might be lost as the kernel is applied fewer times at the perimeter of the input [11]. By padding the input with extra information, typically zeros, zero-padding, the effective size of the input is increased the kernel is applied the entire input. Zero-padding preserves information and the spatial dimension.

Learning rate, threshold

When training a network the learning rate also needs to be specified. The learning rate is used by the optimizer to calculate the size of the optimization gradient. The best learning rate depends on the optimizer, ans is found empirically. The learning rate might also change depending on how long the network has trained for. There are several approaches for modifying the learning rate during training. One way to change the learning rate is to cut it in half every X epochs. This makes the gradient size smaller and leads to the network making smaller adjustments during the later stages of training.

2.1.10 Regularization

To increase the generalization of the network, several regularization techniques have been proposed. In this section two regularization techniques relevant for the network implementation later in this thesis are covered.

Data augmentation changes the training data to increase the networks performance on new input data. By augmenting the data the dataset size can be increased and more relevant real world cases can be covered. Data augmentations relevant to the network training implementation in 4.3.3 are covered in section 2.3.

Monitoring the networks performance on validation data, and stopping the network training when the validation loss stagnates is another regularization technique. This technique is called early stopping and is easy to implement. After each training epoch, the validation loss is measured. If the validation loss is the lowest recorded, the network instance is saved, and the training continues. If the validation loss is higher than the lowest recorded validation loss, the network instance is not saved and the training continues. If the network does not improve over a set threshold, the training is terminated.

2.1.11 Adam Optimizer

After defining a loss function a optimization algorithm is applied to minimize the loss. In this thesis the Adam [12] optimizer is used. Adam combines the techniques of previous relevant optimization algorithms and requires little tuning. It is an adaptive learning algorithm and is closely related to RMSProp [13] and AdaGrad [14].

Adam uses minibatches to increase performance, as the gradient is computed over batches instead of the whole dataset to support parallelization. Adam also implements momentum through exponential weighted moving averages, to reduce the chance for the algorithm to be stuck in a local minimum. By finding the relevant momentum and using the user defined learning rate we find the scale of the gradient which is then used to update the network. The algorithm is robust, but may encounter some problems if the gradients have significant variance. A solution can be to increase the batch size.

2.2 CNN Architectures

In this section two relevant network architectures are covered, namely ResNet [4] and MobilenetV2 [5]. These architectures are relevant to the network implementation 4.3.2 later in this thesis.

2.2.1 ResNet

ResNet is a residual learning framework to better train deep neural networks. ResNet greatly improved the accuracy on the ImageNet [10] and CIFAR-10 [15] datasets amongst others. When adding a layer to a neural network the capacity of the network is usually increased, but at a certain point the added layers decrease the networks error. This increase of error is not caused by overfitting, but rather the networks inability to optimize the deepest parts of the network. ResNet introduces an approach to improve the optimization of deep parts of the network.

In a residual network the output of a network layer is the result of both the layer output and the original input. This is called residual mapping and is visualized in figure 2.3. Residual mapping is realized by using "shortcut connections". Shortcut connections simply skips one or more layers. In residual layers the shortcuts allow the input to be added to the layer output. This operation is computationally inexpensive and does not require additional parameters.

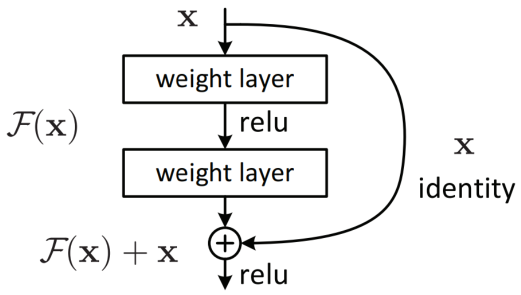


Figure 2.3: Residual network layer, from *Deep Residual Learning for Image Recognition*[4]

2.2.2 MobileNetV2

MobileNetV2 [5] is a network architecture particularly well suited for light weight, mobile, deep neural networks. The network builds on the first paper introducing MobileNets [16].

The main contribution of MobileNetV2 is its building block, the inverted residual with linear bottleneck. This module takes a low-dimensional representation, expands it to a high dimension and then filters it with a lightweight depthwise convolution. The output features are then projected back to a low dimensional representation. Figure 2.4 is a visualization of the feature maps in an inverted residual layer.

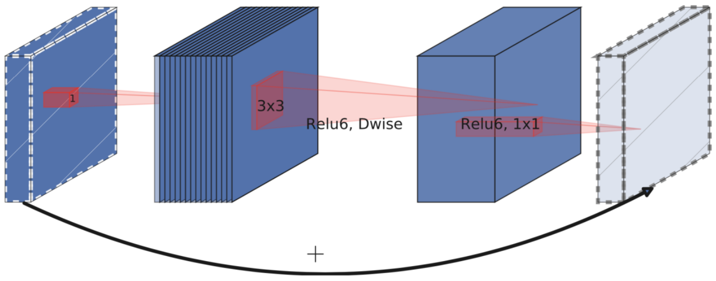


Figure 2.4: Inverted residual layer from *MobileNetV2: Inverted Residuals and Linear Bottlenecks*[5]

2.3 Data augmentation

If there is an insufficient training data foundation or the training data is imbalanced data augmentation is applicable. By applying augmentations to the training data the data foundation can be improved. Within the 3D face reconstruction field image data is used as input to predict 3D vertices. Augmentation of the input image data is therefore relevant. Augmenting the input image must also be reflected in the corresponding ground truth vertices. Translating, scaling and rotation must be done to both the image input and the ground truth vertices to maintain any image to vertex alignment. In figure 2.5 three relevant augmentation techniques are presented to improve generalization; image rotation, image color channel scaling and image dropout.

By rotating the image in the xy-plane the network is be able to predict vertices from faces in different angles better. An identical rotation is applied to the ground truth vertices to align the face vertex image coordinates correctly. Image color channel scaling helps the network see faces in different color and light settings. This is achieved by scaling each color channel in the input image by a random factor. To simulate occlusion randomly sized black boxes are applied to the input data. Dropout, similar to the network regularization technique which deactivate certain activation units [2, p. 255], increases the networks ability to predict partially occluded faces.

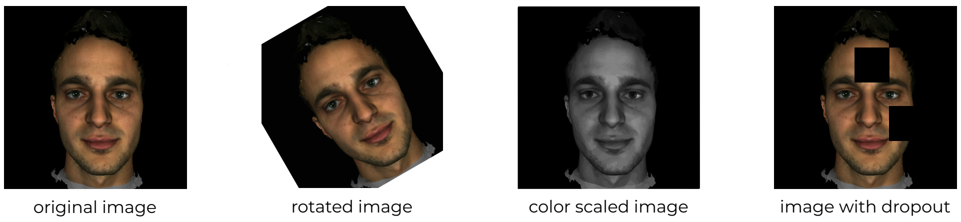


Figure 2.5: Image from the MICC Florence dataset[6] and some example augmentations

2.4 Synthetic Training Data

Data augmentation can be useful to reduce network overfitting and to increase the dataset size. Another way of increasing the dataset size is through creating and using synthetic data. Generating synthetic data using a generative model can improve the training data foundation. Following are two example papers which use synthetic data to increase network performance and a paragraph on how synthetic data can be generated for the facial reconstruction problem.

The SOMAnet [17] is an early example of using synthetic data to improve neural network performance in the person re-identification problem. With a human body generator they were able to render a 100K instance dataset called the SOMAset. Using this data, the network was able to generalize on real world inputs and achieve state-of-the art performance.

In [18] synthetic data is used, together with real data, to train a facial recognition network. By generating synthetic faces with a face image generator they were able to reduce the dataset bias and consequently increase the performance of their neural network. They also showed that transfer training, using first synthetic then real-world data, increased the performance of their network.

To generate synthetic faces for the 3D facial reconstruction problem a generative model is needed. The generative model needs to generate the 3D face models and a way to render them into images. Section 4.2.3 details the implementation of a generative 3D face model.

2.5 3D Morphable Face Models

The concept of 3D Morphable Face Models (3DMMs) was introduced in the 1999 by Volker Blanz and Thomas Vetter [19]. A 3DMM is a generative 3D face model where the shape, illumination, projection and texture parameters can be modified based on a probability density. The 3DMM also serves as a way to parameterize human faces by splitting different features and expressions into vectors. Creating 3DMMs is inherently difficult as constructing such models requires a 3D scanner, several hundred individual face scans and the computation of dense correspondence between the scans. Several 3DMMs are available today [20], but this thesis will focus on one of the most popular ones, the Basel Face Model [21].

The Basel Face Model (BFM) was introduced in 2009 as a public 3DMM. The BFM parameterize face pose, lighting, imaging and identity parameters. This model was further improved in 2017 in the paper 'Morphable Face Models - An Open Framework'[22]. The original one will be used in this thesis. When referring to the BFM we hereby refer to the BFM from 2009. The BFM can generate an unlimited number of 3D faces by sampling from a statistical distribution. Thus, the BFM serves as a standardized generative model of a human face. The ground truth of several 3D face datasets have been transformed into BFM parameters [7]. A 3D face mesh can thus be generated synthetically using the parameters provided. Regressing these 3DMM parameters instead of just N number of 3D vertices with a neural network has also been done to predict 3D faces[23].

Mathematically the BFM describes a face by its shape and texture as indicated by 2.6 and 2.7 [24].

$$S = \overline{S} + A\alpha, \tag{2.6}$$

$$T^{(l)} = \overline{T}^{(l)} + B\beta \tag{2.7}$$

Here S is the 3D face vertices, $\overline{S}$ is the mean shape of the BFM, and the α is the shape parameters corresponding to the 3D shape bases A defining an unique face. The $T^{(l)}$ is the texture of the face defined within the mean shape $\overline{S}$. $\overline{T}^{(l)}$ represents the mean texture, with B being its texture bases and β the texture's unique parameters.

2.6 FaceGen

FaceGen [1] is a 3D face generating software available through a license. FaceGen has created its own 3DMM using 273 high-resolution 3D face scans. The face model is parameterized through 80 dimensions of shape, and 50 dimensions of color. The FaceGen 3DMM is able to produce different mesh topologies through composite statistical appearance models(CSAMs), or just SAMs. A SAM is composite of a statistical shape model(SSM) and a statistical color model (SCM). A SAM is able to generate random faces and render these faces with a mesh topology. FaceGen also provides mesh integrating tools for generating SAMs for any mesh topology layout. Example faces generated with FaceGen are shown in figure 2.6.

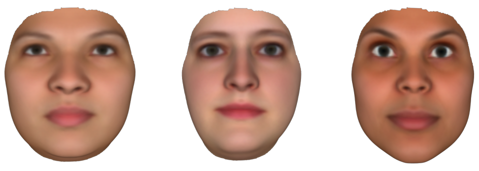


Figure 2.6: Synthetically generated FaceGen faces rendered with the *Preview* SAM

The face generation and rendering pipeline using the FaceGen SDK is outlined in this paragraph. First a SAM is chosen from one the provided FaceGen SAMs. The *Preview* CSAM for example defines the mesh of a face. After navigating to the SAM folder running the **fg3 random** command generates a random face. The **fg3 construct** then generates a 3D mesh and texture image from the generated face. The constructed face mesh and texture image can then be rendered with **fg3 render**. A short description of the different commands is found below.

[1] https://facegen.com

fg3 random

To create a random face a XML-file defining some settings needs to be set up. The `fg3 random setup` produces such a XML-file where the pose, rendering and output settings are specified. An example setting which can be modified is the face rotation range. After specifying the relevant settings the `fg3 random run` command produces a random face.

```
fg3 random run <XML file><output label><number of faces>
```

This command returns a face in the .FG file format. A .FG file is a binary file containing a face coordinate.

fg3 construct

`fg3 construct` takes a SAM, .FG face file and output file name and produces a 3D mesh and texture image.

```
fg3 construct <sam> <face>.fg <out>
```

The output is a mesh file in .tri format and a texture image.

fg3 render

Similarly to the `fg3 random` command the rendering settings are specified by an XML-file. The pose, lighting and camera pose is specified in the file. By passing an argument to the `fg3 render` the face pose and camera projection parameters can be saved for later. These parameters will be necessary to find the image coordinates of all the vertices of the rendered face. The command is thus:

```
fg3 render <XML file> -s <save param file> <mesh>.tri <texture>.jpg
```

The result is a rendered image of the face as well as XML files with the pose and camera projection settings.

2.7 Transformations

In order to transform the FaceGen vertex coordinates into image coordinates relevant transformations need to be defined. The transformations should take the vertices from the eye coordinate system (ECS) to the viewport coordinate space (VCS) via the canonical screen space (CSS). To convert vertices from the ECS to the CSS an extended viewing transformation is used. A viewport transformation will be described in order to further conver the vertices from the CSS to to the VCS.

2.7.1 Extended Viewing Transformation

Given a viewing volume's clipping planes a perspective transformation matrix can be created. The Extended Viewing Transformation 2.8 as described in Theoharis et al. [25, p. 138] is suiting as the truncated pyramid view volume is not necessarily symmetrical about the z_e-axis. The transformation matrix is described below.

n_o = near clipping plane,
f_o = far clipping plane,
b_o = y coordinate as bottom clipping plane intersects with the near clipping plane,
t_o = y coordinate as top clipping plane intersects with the near clipping plane,
l_o = x coordinate as left clipping plane intersects with the near clipping plane,
r_o = x coordinate as right clipping plane intersects with the near clipping plane,

$$M_{ECS\to CSS}^{PERSP} = \begin{bmatrix} \frac{2n_0}{(r_0-l_0)} & 0 & 0 & 0 \\ 0 & \frac{2n_0}{(t_0-b_0)} & 0 & 0 \\ 0 & 0 & \frac{n_0+f_0}{(f_0-n_0)} & \frac{2n_0f_0}{(f_0-n_0)} \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.8}$$

2.7.2 The Viewport Transformation

A viewport is the rectangular part of the screen where the contents are displayed. A viewport can be described by its bottom-left-nearest $[x_{min}, y_{min}, z_{min}]^T$ and top-right-furthest $[x_{max}, y_{max}, z_{max}]^T$ corners. With these variables vertices can be converted from CSS to VCS using a viewport transformation. The Viewport Transformation 2.9 as described in Theoharis et al. [25, p. 141] is used.

$$M_{CSS\to VCS}^{VIEWPORT} = \begin{bmatrix} \frac{x_{max}-x_{min}}{2} & 0 & 0 & \frac{x_{max}+x_{min}}{2} \\ 0 & \frac{y_{max}-y_{min}}{2} & 0 & \frac{y_{max}+y_{min}}{2} \\ 0 & 0 & \frac{z_{max}-z_{min}}{2} & \frac{z_{max}+z_{min}}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.9}$$

2.8 UV Mapping

Later in this thesis *UV Position mapping*, a version of *UV mapping*, will be used to store the 3D vertices of faces in UV space. UV mapping and UV Position mapping is therefore briefly explained in this section.

UV mapping is the process of mapping a 2D image to a 3D model's surface. *U* and *V* typically denote the two axes of the 2D texture image. This way of mapping or storing information has been utilized to express textures [25], height maps [26], and geometry images [27]. Another take on UV mapping is UV Position mapping.

2.8.1 UV position mapping

A UV Position map is a representation of 3D points in UV space [1] [27] where the RGB components are typically used to store XYZ positions. In [1] the 3D positions of points from a 3D face mesh aligned with a corresponding facial image is stored in UV space. The Position mapping is expressed in 2.10

$$Pos(u_i, v_i) = (x_i, y_i, z_i) \tag{2.10}$$

The (u_i, v_i) is the UV coordinate of the ith point in a 3D face mesh and (x_i, y_i, z_i) is the corresponding 3D position. The (x_i, y_i) represents the corresponding 2D position in a RGB facial image, while z_i represents the depth of this point. An image from [1] which neatly illustrates a Position map is found in figure 2.7.

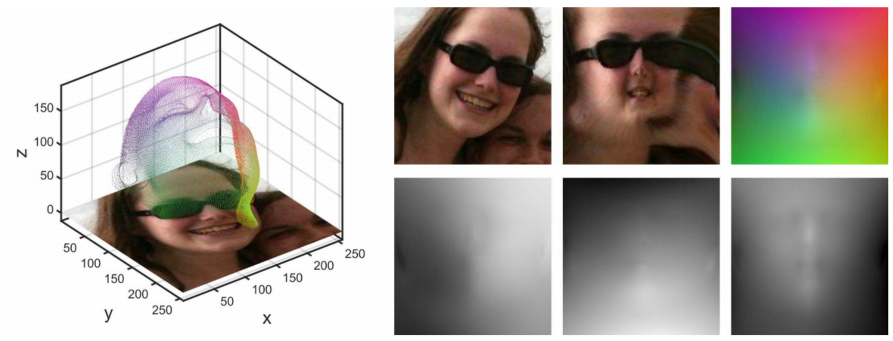


Figure 2.7: The left image shows the 3D mesh plotted on the input image. The top-left image of the 6 image boxes on the right shows the input image, the top-center shows the extracted texture in UV space, and the top-right shows the corresponding UV Position map. The bottom images visualize the XYZ channels of the UV Position map.

2.9 Iterative Closest Point

In the evaluation pipeline later in this thesis 5.2 the Iterative Closest Point (ICP) algorithm is utilized. The ICP is therefore briefly explained in this section.

ICP is a method for aligning 3D objects introduced by Besl and McKay (1992) [28]. The goal of the algorithm is to transform the point set X to the point set P. The algorithm produces a rigid transformation with a translation vector t and rotation matrix R. By applying the R and t to X we can align the point set to P. Initially, the ICP algotihm starts with an estimation R and t. The points in X are then matched with the closest neighboring points in P. A rotation matrix R' and transformation vector t' is then added to the initial R and t, $R = R * R'$, $t = t + t'$. The new R and t are then evaluated. If the transformation meets the convergence criteria the algorithm terminates, if not the algorithm make another iteration.

Chapter 3

Related Works

3.1 3D Face Reconstruction

The deep learning field has been more and more researched in the last years and consequently there have been multiple publications in the 3D facial reconstruction field recently. In 2020 alone, several papers have been released on the subject [29] [30], specially targeting facial texture reconstruction. Several methods focus on the single view 3D face reconstruction[1], but there are some papers with multi-view approaches as well [31] [32]. To the best of the author's knowledge, there are no recent publications which specialize in *only two viewpoints*, front and side, as multi-viewpoint methods generalize to deal with viewpoints from arbitrary angles.

Different 3D face reconstruction methods have been suggested. Many partly depend on a reference 3DMM or the mean shape of the 3DMM to predict a face through regressing 3DMM parameters instead of vertex coordinates [23][33]. The Volumetric Regression Network (VRN) [34] introduced a straightforward way to map input image pixels to a full 3D facial structure unrestricted from any face model space. The paper defines a complex network structure which predicts voxel data. The Position map Regression Network(PRN) [1] builds on the idea of mapping input data unrestricted from model space and predicts Position maps from input images.

Since late 2018 some papers have been able to beat the PRN on certain test datasets. One of these papers is [32]. Instead of directly predicting representations of 3D face vertices, the authors implement a complex network which learns on the Image-level loss such as skin estimation loss and the Perception-level loss for deeper features of the face. The paper also proposes a multi-image confidence score system which outperforms basic shape averaging. This type of multi-network approach to fit different aspect of a face; depth, shape, lightning and texture has been used in several recent papers [29][35].

Taking into account that this thesis proposes a method that builds on the PRN approach we will now go through the main contributions made with PRN.

3.1.1 PRN

The work in [1] proposes a method that both reconstructs 3D facial structure and provides dense alignment from a single picture. The proposed method is still performing good compared to recent papers[1]. The method utilizes a PRN to predict UV Position maps, as described in section 2.8.1, to represent a 3D facial structure with alignment information. Predicting a Position map is advantageous as the spatial adjacency information among points is preserved. Also predicting each point would require a fully connected layer connected to each point, which would result in a big number of network parameters. The PRN is light-weight and spends only 9.8ms to process an image to generate a UV Position map on a modern GPU [1].

Network architecture

PRN utilizes an encoder-decoder network, and predicts a Position map from unconstrained 2D images. An encoder-decoder network extracts features from an image input and decode the features found into a goal output, typically of the same size as the input. The Encoder part of PRN consists of one convolutional layer followed by 10 residual blocks which reduce the input image into 512 feature maps. The decoder network contains 17 transposed convolutional layers 2.1.5 to generate the predicted position output. The resulting network structure can be found in figure 3.1.

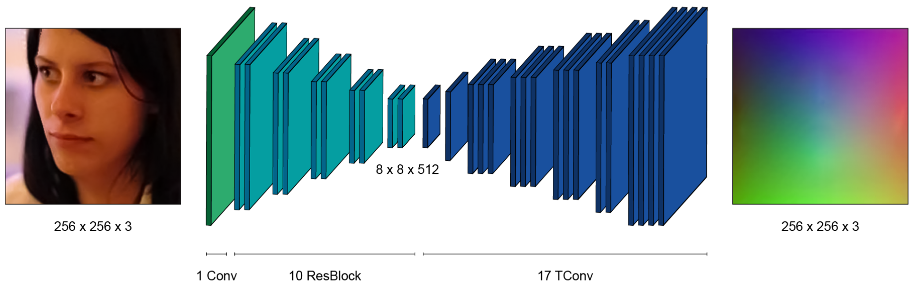


Figure 3.1: Network architecture of PRN, left is the input image, and right is an illustration of the predicted Position map

Loss function

The loss function for PRN measures the mean squared error (MSE) between ground the truth Position map and the predicted Position map with a weight mask to increase the

[1]https://paperswithcode.com/sota/3d-face-reconstruction-on-florence

importance of features in central regions of the face. The weight mask is visualized in figure 3.2 with these weights: (**subregion1** : **subregion2** : **subregion3** : **subregion4**) = (**16:4:3:0**). The Loss function is defined in equation 3.1 where the $Pos(u,v)$ denotes the predicted Position map and $Ps(u,v)$ denotes the ground truth Position map with the weight mask $W(u,v)$.

$$Loss = \sum ||Pos(u,v) - Ps(u,v)|| \cdot W(u,v) \tag{3.1}$$

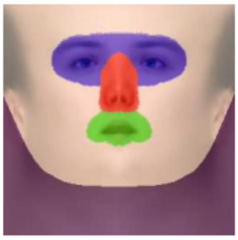


Figure 3.2: colored texture map with the training weights. **Subregion1** = 68 keypoints, **Subregion2** = (purple, red, green), **Subregion3** = face, **Subregion4** = neck.

3.2 Datasets

Producing 3D models with corresponding images is a challenging and costly task. In this section two relevant 3D facial reconstruction datasets are introduced.

3.2.1 300W (-LP)

The 300 Faces in-the-wild challenge dataset (300W) [36] was created for a facial landmark localization challenge in 2013. The 300W dataset includes the datasets AFW, LFPW, HELEN, IBUG and XM2VTS [36] with a standardized keypoint annotation as shown in figure 3.4. There are more than 3500 individuals photographed in the 300W dataset.

One of the limitations, except from its size, is the lack of extreme yaw angled poses in the dataset. The dataset lacks faces with yaw angles in the [45°, 90°] range. The 300W-LP (LP = Large Pose) dataset [37] is an extension of the 300W dataset and addressed the lack of annotated training data with yaw angles in the [45°, 90°] range. The authors fitted the faces in 300W with BFM parameters and rotated the fitted faces with yaw angles up to 90° in k steps, with k typically in the $[10,15]$ range. One rotating example can be found in figure 3.3. The resulting dataset is called 300W-LP and contain image and corresponding 3DMM parameters as described in BFM. The 300W-LP dataset consists of 122,450 image samples and serves as a good source for training data with respect to 3D face reconstruction. One issue with 300W-LP is that the BFM parameters are fitted based on only 68 keypoints. As a result the fitted 3D face mesh is not entirely accurate.

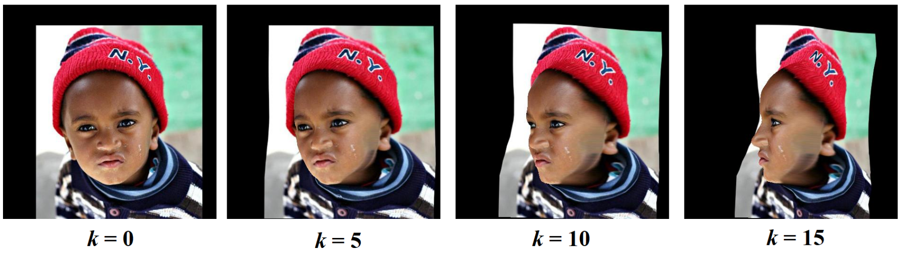


Figure 3.3: 300W-LP dataset, yaw angle rotated k number of times

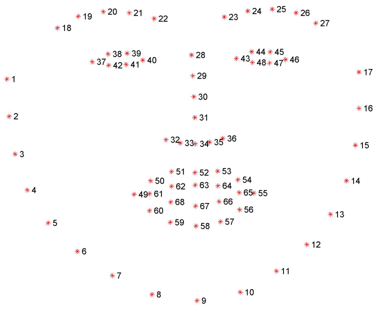


Figure 3.4: Keypoint annotation in 300W-LP.

3.2.2 Test datasets for 3D face reconstruction

The MICC Florence dataset [38] consists of high-resolution 3D facial scans, images and HD videos of 53 people. It is commonly used as a test dataset for 3D face reconstruction methods. The dataset contains accurate and complete 3D models of faces and is used as a metric of comparison between 3D reconstruction solutions[2]. Sample faces can be seen in figure 3.5.

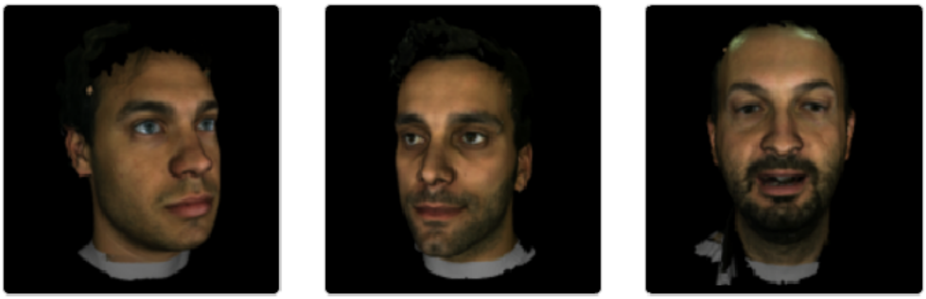


Figure 3.5: Example faces from the Florence dataset

[2]https://paperswithcode.com/sota/3d-face-reconstruction-on-florence

Chapter 4

Methodology

This chapter explains the implementation of our pipeline for reconstructing 3D facial meshes from front and side images. The main component of the proposed pipeline is the CNN. The CNN is fed a concatenated image matrix and predicts the corresponding Position map. Both real and synthetic training data is used to train the network. The data generation process, CNN implementation and CNN training are covered in this chapter.

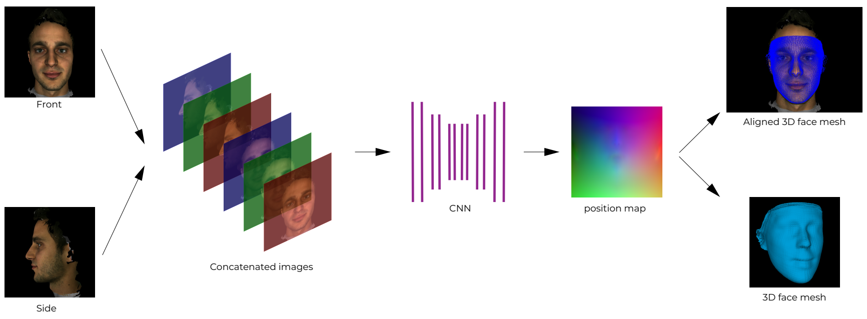


Figure 4.1: The proposed 3D facial reconstruction pipeline.

4.1 Proposed Pipeline

The proposed pipeline builds on the PRN implementation [1] and is outlined in figure 4.1. The pipeline is built around a CNN which is fed front and side images and produces a Position map. The input images are simply concatenated before being fed into the network as visualized in figure 4.1. The network then predicts a position map. The network implementation is explained in section 4.3. Using the predicted Position map, the 3D vertices are extracted and reconstructed into a facial mesh using the face3d[1] library.

[1]https://github.com/YadiraF/face3d

4.2 Training Data Generation

In order to train our network we need to generate training data. The input data of our network are two images, one facing the front, and one facing the left or right. The network predicts a Position map which maps the face mesh vertices to one of the input images, in our case the front facing image. The training data pairs for our network contain two images of a face as input, and one Position map as ground truth. To generate training dataset pairs we use the 300W-LP dataset and synthetic data from FaceGen. The 300W-LP dataset contains images with Basel Face Model (BFM) parameters defining the shape, expression and pose. From these parameters the 3D face mesh and UV Position map can be generated using the Face3d library. With FaceGen we synthetically generate renderings of random faces and their corresponding 3D face mesh. We then generate the Position maps for the generated images by applying the rendering transformation to a corresponding 3D face mesh which is rendered into a UV Position map. By using the 300W-LP dataset and FaceGen we generate more than 60K data pairs to train our CNN.

4.2.1 Front and Side Face Definition

We define a front facing image as an image of a face with a yaw angle in the $[-45°, 45°]$ range, and a side face image as an image of a face with a yaw angle not in this range. We limit the range of face yaw angles for side angles to be in the $[-100°, -45°]$ and $[45°, 100°]$ ranges as large portions of the face are occluded at any greater yaw angle.

4.2.2 300W-LP Dataset

To generate data pairs from the 300W-LP dataset we generate and transform the vertices as defined in the BFM parameters and keypoint information each image is accompanied by. The BFM parameters are used to generate a 3D facial mesh using the mesh topology layout defined in the Face3d library. Using the provided keypoint information we crop the face and save the cropping transformation parameters. This cropping transformation is then applied to the generated 3D facial mesh vertices to align the mesh to the new cropped image. With the transformed vertices the Position map is rendered in UV space. The pipeline is largely similar to the implementation in [1] and is visualized in figure 4.2. The python implementation of this pipeline is found in Appendix A.1.

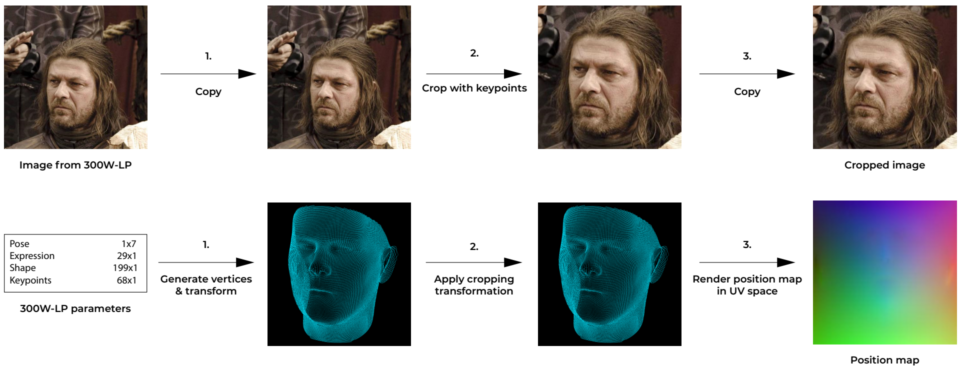


Figure 4.2: The 300W-LP training data generation pipeline. We first generate the vertices and transform them to the correct pose using the Face3d library(1.). We then crop the image and the generated vertices according to the given keypoints(2.). Finally we render the Position map in UV space(3.).

4.2.3 Synthetic Face Generation with FaceGen

To generate random faces from FaceGen we create and use a shell script. The script is found in listing 4.1. The script takes a numeric input for how many faces it should create. For each iteration a random face is generated using the `fg3 random run` command. The face mesh and texture map for this face is then constructed using `fg3 construct`. The rendering settings for the face mesh is altered through `_fg_generate_settings.py`. This Python program changes the rendering parameters in three FaceGen XML files (`front[.xml]`, `right[.xml]`, `left[.xml]`). The program sets the yaw, pitch, roll, scale and translation of the face to predetermined ranges. These .xml files are then passed to the `fg3 render` command producing renderings of the face with randomized front, left and right sided poses. A SSM with the same vertex layout as BFM is utilized to create a 3D mesh in a .obj file format. Finally the shell script moves the images, 3D mesh and render settings to a target folder.

```
for i in $(seq "$1")
do
        num=$( printf '%05d' $i )
        echo "generating face "$num
        python _fg_generate_settings.py
        fg3 random run _random_settings base_ 1
        fg3 construct Head/Headhires base_0000.fg base_mesh
        fg3 render front -s front
        fg3 render left -s left
        fg3 render right -s right
        mkdir ${num}
        fg3 -s construct BFM/BFM base_0000.fg base_mesh
        fg3 -s triexport ${num}/${num}.obj base_mesh.tri
        mv front.png ${num}/${num}_front.png
        mv left.png ${num}/${num}_left.png
        mv right.png ${num}/${num}_right.png
        mv front.xml ${num}/${num}_front.xml
        mv front_cam.xml ${num}/${num}_front_cam.xml
        mv left.xml ${num}/${num}_left.xml
        mv left_cam.xml ${num}/${num}_left_cam.xml
        mv right.xml ${num}/${num}_right.xml
        mv right_cam.xml ${num}/${num}_right_cam.xml
rm base_mesh*.* base_0000*.* *_pose.xml
done
```

Listing 4.1: Shell script for generating random faces using the FaceGen SDK

Constructing 3D Facial Meshes From FaceGen

The 300W-LP 3D facial meshes are generated using a mesh topology layout as described in the Face3d library. The FaceGen 3D facial meshes need to be converted to the same mesh topology layout. The out-of-the-box Statistical Shape Models (SSMs) from FaceGen are only able to generate meshes with approximately 5000 vertices. The mesh topology layout is also different than the topology layout in the Face3d library. To create FaceGen facial meshes with the same mesh topology the FaceGen mesh integration tools[2] can be utilized. Fitting the base BFM shape's 3D facial mesh to a FaceGen SSM generates a SSM with the same fixed mesh topology as the input BFM facial mesh. Passing this SSM, together with any FaceGen face, to `fg3 construct` then generates a 3D mesh with the Face3d topology layout. The construction of the BFM mesh is done in line 12 in the shell script in listing 4.1.

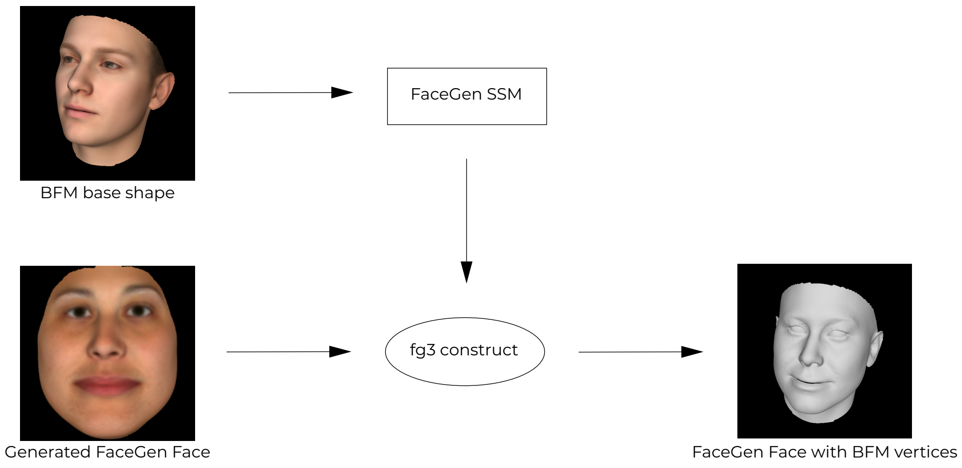


Figure 4.3: Generating 3D facial meshes with a mesh topology layout as describe din face3d.

4.2.4 Output from Synthetic Data Generation

We use the script to generate 10K faces, each face is accompanied with one 3D mesh and rendering of the face from the front, the left and the right as well as the pose settings for each rendering. The faces are rendered with different yaw, pitch and roll angles for the front, left and side face images. The angle ranges are described in figure 4.4. Example outputs are showcased in figure 4.5.

[2]https://facegen.com/dl/sdk/doc/manual/meshintegration.html

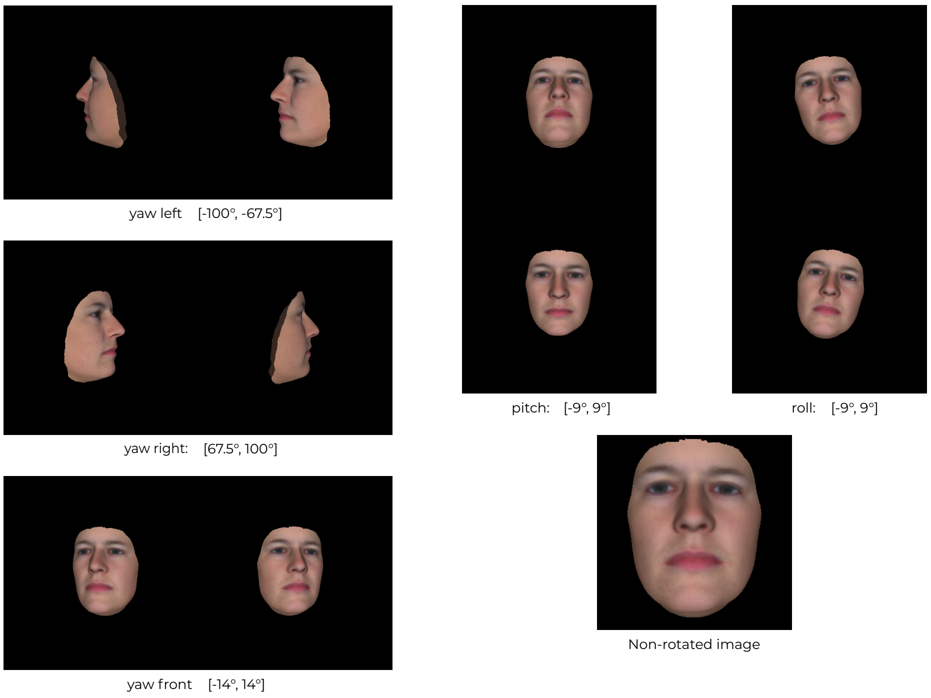


Figure 4.4: A FaceGen face with the different yaw, pitch and roll angle ranges.

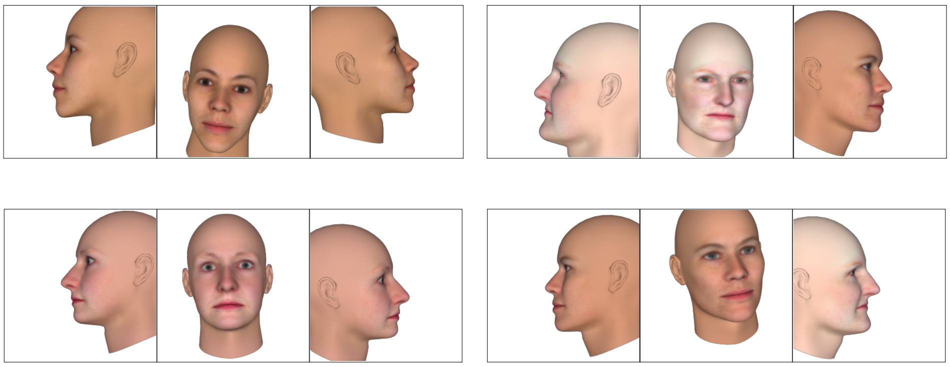


Figure 4.5: Example renderings of FaceGen faces rendered with the pose angles described in figure 4.4.

4.2.5 FaceGen Dataset

To generate training data pairs for the network a Position map for the rendered synthetic facial images needs to be constructed. This is done in two steps. First, the the synthetic 3D facial mesh is transformed to the viewport coordinate system (VCS) corresponding with the facial image rendering settings. Then the vertices are rendered to UV space similarly to the pipeline in section 4.2.2. The training data generation pipeline for FaceGen data is outlined in figure 4.6. With the 10K FaceGen faces we produce 20K training data pairs.

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = M^{VIEWPORT}_{CSS\to VCS} \cdot M^{PERSP}_{ECS\to CSS} \cdot M_{WCS\to ECS} \cdot X_w \tag{4.1}$$

VCS Face Transformation

To generate the 3D facial mesh vertex image coordinates for each synthetic facial image the rendering settings from FaceGen are applied the accompanying 3D face mesh. FaceGen provides the scale, translation, rotation and frustum parameters in the rendering settings. The scale, translation and rotation are applied to the 3D mesh through matrix multiplication. To take the vertices from ECS to CSS an extended viewing transformation is applied as defined in equation 2.8, and to take the vertices from CSS to VCS we apply a viewport transformation as described in equation 2.9. A vertex point $X_w = [x_w, y_w, z_w, 1]^T$ in the mesh is converted into VCS using the equation 4.1. Where the $M_{WCS\to ECS}$ matrix transform defines a scaling, rotation and translation transformation.

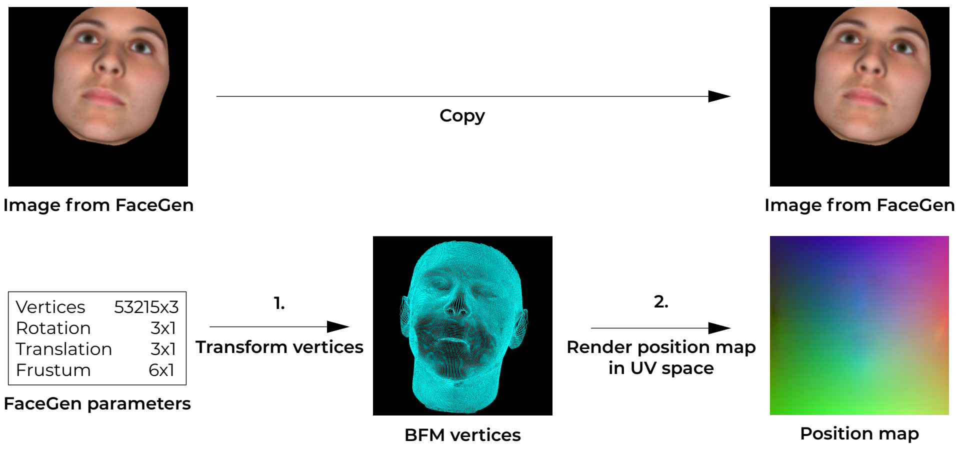


Figure 4.6: The FaceGen training data generation pipeline. A position map is generated for a facial image using the corresponding FaceGen parameters.

4.2.6 Applying Random Background Images

To further improve generalization we apply random background images to the FaceGen images. Inspired by the face generator in [18] a random texture is chosen from the Describable Texture Database [39] and added to a FaceGen image. Example faces with random texture backgrounds are shown in figure 4.7. The python implementation is found in appendix A.2.

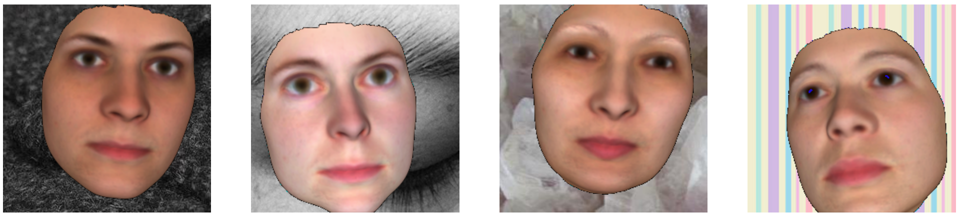


Figure 4.7: FaceGen images with random texture as background

4.3 CNN Implementation

To predict Position maps from input images we implement a new CNN with Keras[3]. We use the original PRN implementation as a starting point when deciding our network architecture. We make some adjustments to allow a concatenated $256x256x6$ image input as well as replacing the ResNet modules with inverted residuals described in section 4.8. We train the network on the synthetic data generated by FaceGen before training it with data from the 300W-LP dataset.

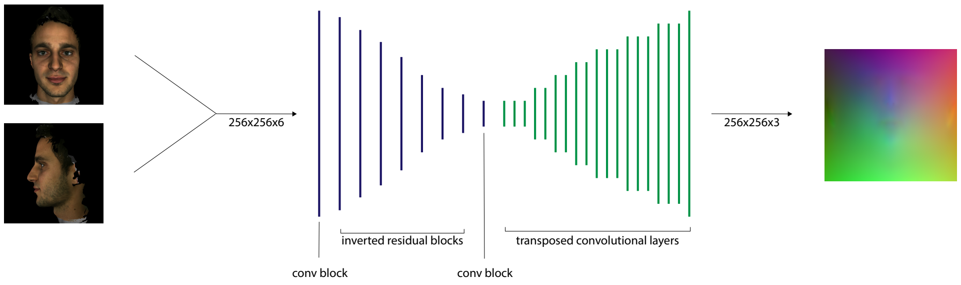


Figure 4.8: Our proposed CNN architecture

4.3.1 Input and Output

The size of the two input images are $256x256x3$ for the height, width and color channels respectively. This is the same as the size used in the PRN implementation. As we, in this thesis, will use front and side facial images instead of one, we concatenate the images, expanding the original image color channel dimensions resulting in an image matrix of size $256x256x6$ as visualized in the second step of the pipeline in figure 4.1. The Position map is of size $256x256x3$, the same as in PRN, which means that the Position map is capable of containing $256 * 256 = 65536$ vertices, this is enough to define a 3D face mesh of great accuracy [1].

4.3.2 Network Architecture

We employ an encoder-decorder network structure to map the input image to the output Position map. The encoder part of our network consists of 1 convolutional layer, followed by 4 inverted residual layers and finally 1 convolutional layer. The inverted residual layers are internally repeated 1-4 times. The decoder part of our network consists of 17 transposed convolutional layers. The network layers are listed in table 4.1. The network has a total of 11,791,273 parameters and is 154MB.

We choose the MobileNetV2 inverted residual blocks instead of ResNet blocks as the MobileNetV2 architecture is newer, lightweight and performs better in image processing

[3]https://keras.io/

tasks [5]. The Inverted residual blocks were also easy to implement as there are several implementations available[4][5].

We set the filter sizes of each layer to reduce the $256x256x6$ input to $8x8x512$ feature maps similar to PRN. The kernel size for the inverted residual blocks are 3, while the kernel size for the transposed convolutional layers are 4. We use zero-padding and ReLU as activation function for 16 layers of our decoder network and Sigmoid for the final one.

We use the Adam optimizer with the same loss function and weight mask as in the original network implementation as described in section 3.1.1. The batch size is set as 16 and we find the best learning rates and learning halving rates empirically for each training step.

Input	Layer	kernel	stride	output
256 x 256 x 6	Convolution	3	2	128 x 128 x 32
128 x 128 x 32	Inverted Residual	3	-	64 x 64 x 96
64 x 64 x 96	Inverted Residual	3	-	32 x 32 x 144
32 x 32 x 144	Inverted Residual	3	-	16 x 16 x 192
16 x 16 x 192	Inverted Residual	3	-	8 x 8 x 576
8 x 8 x 576	Convolution	3	2	8 x 8 x 512
8 x 8 x 512	Transposed Convolution	4	1	8 x 8 x 512
8 x 8 x 512	Transposed Convolution	4	2	16 x 16 x 256
16 x 16 x 256	Transposed Convolution	4	1	16 x 16 x 256
16 x 16 x 256	Transposed Convolution	4	1	16 x 16 x 256
16 x 16 x 256	Transposed Convolution	4	2	32 x 32 x 128
32 x 32 x 128	Transposed Convolution	4	1	32 x 32 x 128
32 x 32 x 128	Transposed Convolution	4	1	32 x 32 x 128
32 x 32 x 128	Transposed Convolution	4	2	64 x 64 x 64
64 x 64 x 64	Transposed Convolution	4	1	64 x 64 x 64
64 x 64 x 64	Transposed Convolution	4	1	64 x 64 x 64
64 x 64 x 64	Transposed Convolution	4	2	128 x 128 x 32
128 x 128 x 32	Transposed Convolution	4	1	128 x 128 x 32
128 x 128 x 32	Transposed Convolution	4	2	256 x 256 x 16
256 x 256 x 16	Transposed Convolution	4	1	256 x 256 x 16
256 x 256 x 16	Transposed Convolution	4	1	256 x 256 x 3
256 x 256 x 3	Transposed Convolution	4	1	256 x 256 x 3
256 x 256 x 3	Transposed Convolution	4	1	256 x 256 x 3

Table 4.1: Table listing all layers in the CNN implementation. The thin line separates the encoder and decoder network parks, but the network is not in any way split up.

[4]https://github.com/d-li14/mobilenetv2.pytorch
[5]https://github.com/xiaochus/MobileNetV2

4.3.3 Training on synthetic data

We train the network on the 20000 synthetic data pairs generated in section 4.2. We split the training data pairs in a training and validation part for training evaluation. The data pairs are shuffled before each epoch to randomize the batches. The training data is augmented before training by rotation, color shifting and image dropout as described in section 2.3. The rotation is set to be in the $[-45°, 45°]$ range, the random color channel scale is between 0.9 to 1.2. We train the network with an initial learning rate of 0.0001. The learning rate is halved every 5 epochs. After each epoch we calculate the loss on the validation data. If the validation loss has not decreased in 10 epochs, the learning is suspended. The python implementation of the training code is found in appendix A.3. We train the network on a computer with Intel Core i7-8700K CPU and a Nvidia RTX 2080Ti GPU. The validation loss is plotted in figure 4.9.

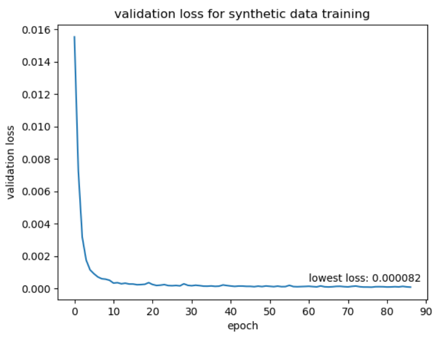


Figure 4.9: Validation loss over synthetic data training

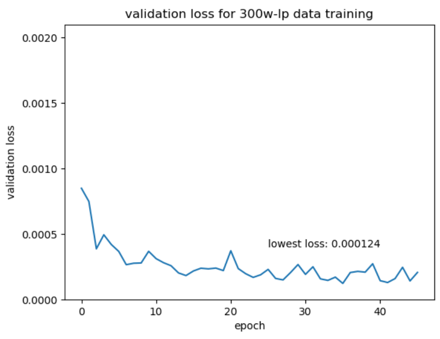


Figure 4.10: Validation loss over transfer training

4.3.4 Transfer Learning with 300W-LP data

To train the network on "real" data we restore the model weights from the model trained in the previous section. We generate training data pairs from the 300W-LP dataset using the front and side definition earlier. For each 300W-LP face we select all images with a face yaw pose within the $[-45^\circ, 45^\circ]$ range. We pair each of these front facing images with one image of the same face with a jaw pose within the $[-100^\circ, -45^\circ]$ and $[45^\circ, 100^\circ]$ ranges. These images are then augmented the same way as the synthetic data before being fed into the network. We set the initial learning rate to 0.00001 and train the network. The learning rate is halved every 5 epochs. The validation loss is plotted in figure 4.10.

Chapter 5

Evaluation

We intend to evaluate the ability of our synthetically trained and transfer trained networks to reconstruct 3D facial meshes from front and side images. We assess our networks using data from the MICC Florence dataset. This chapter presents the evaluation dataset, the evaluation pipeline and the performance results for our networks. The performance of our networks is also compared to the performance of the network in [1] (PRN). Our network also produces a dense facial alignment on the input image, but we will not evaluate the network's facial alignment performance as it falls outside the scope of this thesis.

5.1 Evaluation dataset

We use the MICC Florence dataset as the evaluation dataset. The dataset consists of 2D images, videos and high-resolution 3D scans of 53 subjects. The images, videos and 3D meshes for each face are stored in separate folders. Each subject is scanned from multiple angles. For our evaluation the frontal scans are used. More specifically the .obj file and the corresponding texture data from the `Model/frontal1/obj` folder for each subject is used. Some of the subjects have facial hair, which is included in the 3D scan. The 3D faces used to train the proposed network and PRN do not have facial hair which will increase the error for subjects with facial hair. As the facial hair error is similar for both PRN and the proposed method these subjects are not excluded.

5.2 Evaluation pipeline

We render a front and side facing image from the MICC Florence dataset, predict the corresponding Position map and evaluate the output facial mesh. The images are rendered from extracted .obj files in MeshLab[1] with orthographic projection. After generating a 3D

[1] http://www.meshlab.net/

facial mesh from the predicted Position map we fit the predicted 3D facial mesh to the ground truth 3D mesh provided in the MICC Florence dataset. We align the meshes using an implementation of the ICP algorithm[2]. If necessary, an initial alignment is passed to the algorithm. The ICP implementation outputs the rotation matrix and translation vertex as a homogeneous transformation matrix which maps a point set X to a point set P. After applying this transformation matrix to the predicted 3D face mesh we calculate the evaluation metric of the network.

5.3 Evaluation Metric

The goal of the evaluation metric is to evaluate the networks ability to reconstruct a 3D facial meshes. The evaluation metric measures the difference between two facial meshes. We employ the normalized mean error (NME) of the euclidean distance between the points of the predicted and ground truth 3D mesh to be the evaluation metric. The error function is defined in equation 5.1. $||(p_\mathrm{i}-q_\mathrm{i})||_2$ denotes the euclidean distance between two points, and d denotes the normalization factor. The normalization factor is set to be the bounding box size of the predicted 3D facial mesh.

$$NME = \frac{1}{N}\sum_{i=1}^{N}\frac{||(p_\mathrm{i}-q_\mathrm{i})||_2}{d} \tag{5.1}$$

[2]https://github.com/ClayFlannigan/icp

5.4 Performance of the Synthetically Trained Network

First, the performance of our synthetically trained network is evaluated. We run 20 randomly selected faces from the MICC Florence dataset through the evaluation pipeline using our network from section 4.3.3. We visualize the results by utilizing a Cumulative Error Distribution (CED) curve. The mean NME is presented in table 5.1. Figure 5.1 shows the CED curve for the 20 images from the MICC Florence dataset. Figure 5.2 shows two exemplary predicted 3D facial meshes together with the ground truth meshes.

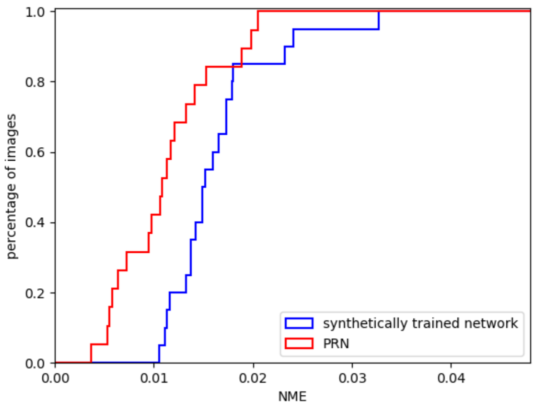


Figure 5.1: CEDs for synthetically trained network and PRN.

	Mean NME
PRN	**0.0134**
Our network	0.0164

Table 5.1: Performance comparison between our syntetically trained network and PRN by looking at the mean NME.

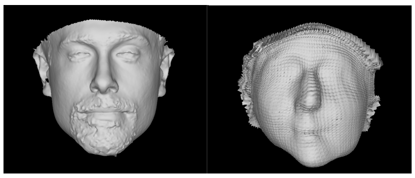
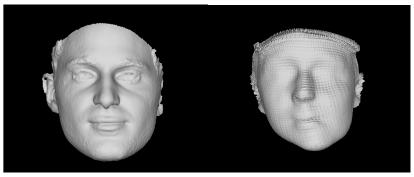



Figure 5.2: Two example 3D facial meshes from the MICC Florence dataset paired with the reconstructed 3D facial mesh made by the synthetically trained network. There are two data pairs where the left image is the ground truth face, and the right image is the reconstructed 3D facial mesh.

The CED curve in figure 5.1 and mean NME in table 5.1 shows that the synthetically trained network performs worse than PRN on the MICC Florence dataset. The NME is consistently higher. By looking at the two examplary outputs of the networks in figure 5.3 the shortcomings of the synthetically trained network become apparent. The reconstructed mesh has an asymmetrical face shape compared to the ground truth facial mesh and there are few visual similarities. There are also artefacts along the top seam edge of the mesh.

A likely explanation for the networks inability to reconstruct convincing 3D meshes is the fundamental difference in the network's training data and the MICC Florence test data. Example images from the test dataset and FaceGen are shown in figure 5.3. The lack of realistic hair, expressions, accessories and skin texture likely reduces the generalization of the synthetically trained network. The facial asymmetry observed in figure 5.2 could be a result of overfitting. If the synthetic training data consistently contains faces with a narrow jaw shape the network could be unable to generate output meshes with wide jaws. Lastly the artefacts in the reconstructed facial mesh are likely the result of a mechanism connected to the generated weight mask used to calculate the loss function. The artefacts occur at the outermost vertices covered by the weight mask described in section 3.1.1. The same artefacts are also found to some degree in the reconstructed meshes from PRN.

Using realistic hair and expressions when rendering synthetic data from FaceGen will likely improve the generalization of the synthetically trained network. A possible solution to the artefact problem could be a modified weight mask with increased or decreased weights at the seam edge vertices.

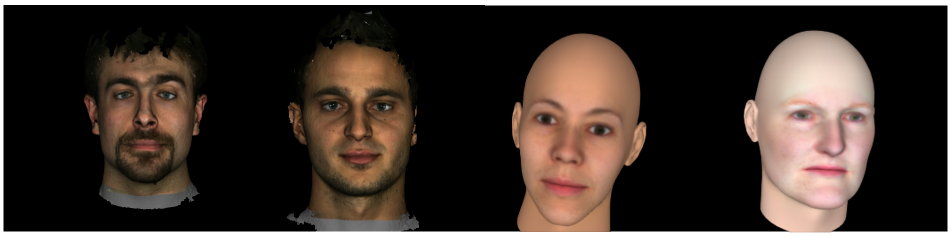


Figure 5.3: Random images from MICC Florence and FaceGen datasets. The two images on the left are from MICC Florence and the two images on the right are from FaceGen.

5.5 Performance of the Transfer Trained Network

To evaluate the transfer trained network described in section 4.3.4 we again run the evaluation pipeline using the same 20 faces that was used in the previous section. The CED curves for our transfer trained network and PRN is plotted in figure 5.4. The mean NME is presented in table 5.2. Figure 5.5 shows the constructed 3D facial meshes from 3 example faces using both the transfer trained network and PRN.

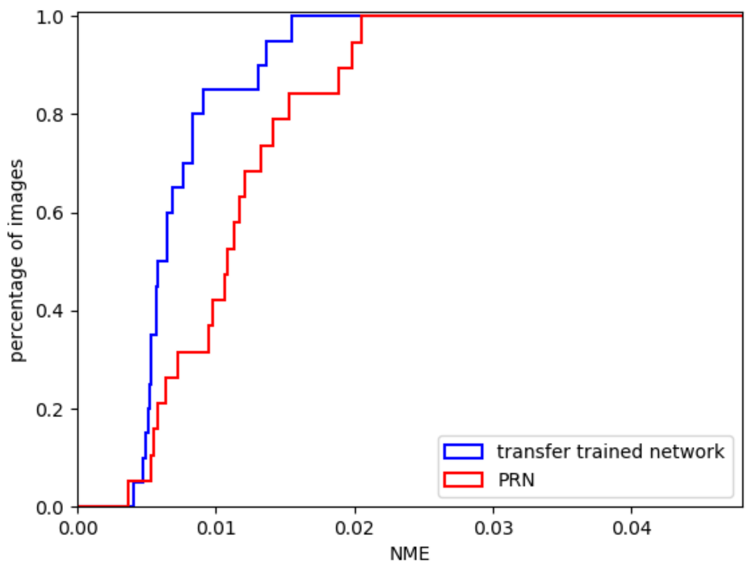


Figure 5.4: CED for transfer trained network and PRN

	Mean NME
PRN	0.0134
Our network	**0.0074**

Table 5.2: Performance comparison between our transfer trained network and PRN by looking at the mean NME.

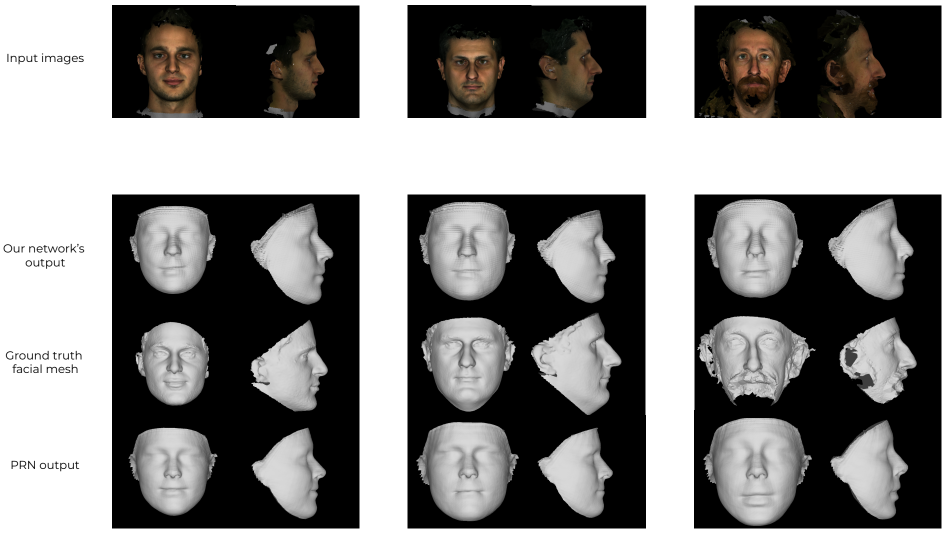


Figure 5.5: Example reconstructed meshes using three faces from the MICC Florence dataset. The top row shows the input images for the networks. PRN uses a single front facing image as input, while our network is fed both the front and side images. The transfer trained network output is shown in the second row, the ground truth facial mesh in the third row and the PRN output in the fourth row.

The CED curve in figure 5.4 and mean NME in table 5.2 shows that our transfer trained network is able to reconstruct more accurate 3D facial meshes than PRN. More than 80% of the reconstructed faces using the transfer trained network have a NME lower than 0.01, while approximately 40% of the faces reconstructed using PRN have a NME lower than 0.01.

When examining the exemplary reconstructed 3D facial meshes in figure 5.5 the difference is not as clear. By comparing the transfer trained networks output with the ground truth facial mesh we see only some similarities. The most distinct facial features are not accurately reconstructed. The reconstructed meshes using our network are more similar to the meshes reconstructed using PRN. There are some differences between our network and PRN around the nose and jawline. The PRN reconstructs similar noses for all the example faces, while our network is able to reconstruct *slightly* more varied nose shapes. The difference is clearest in the rightmost face in figure 5.5. The artefact problems found in the synthetically trained network has largely disappeared. Both networks also predict faces without facial hair. The lack of facial hair is a result of the training data as the synthetic and 300W-LP dataset provide 3D faces without facial hair as ground truth.

The reconstructed faces are largely similar to each other, but are ultimately unable to accurately reconstruct distinct facial structures. One probable explanation for the similarities between our network's and PRN's reconstructed 3D facial mesh is the training data foundation. The transfer trained network is trained on data from the 300W-LP dataset. The ground truth faces in 300W-LP are generated from 68 2D keypoints using a CNN as described in section 3.2.1. By training a network on 300W-LP we risk simply predicting the output of the CNN used to generate the 300W-LP faces. This might also explain the inability for our network to further utilize the additional side image input to greatly improve the performance of our network.

5.6 Potential Sources of Error

5.6.1 MICC Florence Dataset

The evaluation pipeline utilizes the MICC Florence dataset to test the reconstructed 3D facial meshes on ground truth facial meshes. There is a racial bias in the test dataset as all subjects in the dataset are white. This bias might increase or decrease the performance of our proposed network on the MICC Florence dataset. The bias should however affect both our network and PRN similarly as both networks are trained on 300W-LP.

The images used as input when evaluating the proposed network were produced by rendering 3D facial meshes using a mesh processing software. Increasing the number of images by rendering each 3D facial mesh mulitple times at multiple angles as implemented in [1] provides more test data. The process of rendering and aligning the ground truth meshes is however time-consuming and was therefore not implemented in this thesis. Ultimately, renderings from 20 MICC Florence faces should give a sufficient foundation for evaluation and comparison between our networks and PRN.

5.6.2 ICP alignment

The evaluation pipeline utilizes ICP to align a reconstructed 3D facial mesh to a ground truth mesh for evaluation. The ICP requires an initial alignment to align two point sets correctly. By saving the initial alignment parameters for each face reconstructed using PRN to the correct ground truth 3D facial mesh this source for error is largely minimized. As PRN and our network produces 3D facial meshes with vertices in the same value range, the initial alignment is valid for both meshes. If a reconstructed facial mesh is not similar enough to the ground truth mesh the face will however not be correctly fitted. This fitting error is difficult to avoid, but with the initial alignment most faces are fitted correctly. Most of the reconstructed faces using the synthetically trained network and some of the faces with the highest NME in figure 5.4 probably suffers from bad facial alignment. As the initial alignment parameters was found using PRN any ICP fitting disadvantage should be similar for PRN and our networks.

5.7 Discussion

As seen in figure 5.6 the performance for our synthetically trained network is worse than PRN. Using the synthetically trained network weights as initial weights and using two images instead of one resulted in our transfer trained network performing better than PRN. Increasing the quality of our synthetic dataset should not only boost the performance of our synthetic dataset, but also further increase the performance of the transfer trained network. By applying realistic hair, expression and skin texture a synthetically trained network should be able to generalize better.

In the 3D facial reconstruction field the lack of accurately labeled training data increases the importance of synthetic data. Labeling 2D images with corresponding 3D faces is time consuming and difficult. The 300W-LP uses a CNN to create 3D faces using an input 2D images and 68 keypoints. While this simple approach of generating labeled training data produces many data pairs quickly, it is not entirely accurate. The similarity between the transfer trained networks output and the PRN output is likely a result of the data labeling CNN in 300W-LP. With synthetically generated data the ground truth 3D facial mesh is always accurate.

To generate synthetic training data we have utilized FaceGen. FaceGen provides powerful tools to easily generate 3D faces. However, the output from a simple FaceGen SAM without expression, hair and detailed skin texture is not good enough to train a generalized network. A natural extension of the work in this thesis would be to improve the photo realism of the proposed synthetic data generation pipeline output. Additionally, increasing the amount of data, both real and synthetic, should result in an increased performance of our method.

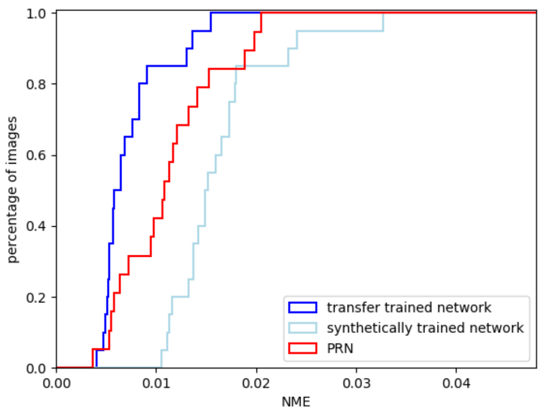


Figure 5.6: CEDs for our networks and PRN.

Chapter 6

Conclusion and future work

6.1 Conclusion

In this thesis we have proposed and evaluated a new method which reconstructs a 3D facial mesh from front and side images using a CNN. The CNN builds on the network proposed in [1] and achieves a lower mean NME on the MICC Florence dataset. Using a concatenated image matrix as input, the network predicts a position map from which a 3D facial mesh is generated. The network is trained on the synthetic data generated using our proposed synthetic data generation pipeline and data from 300W-LP.

6.2 Further Work

To further improve the performance of our proposed method we recommend several improvements.

6.2.1 Network Architecture

The network architecture is largely inspired by the original implementation in [1]. Further experimenting with the network architecture, different hyper parameters and regularization techniques can further improve the generalization of the network and reduce the number of parameters. Especially the decoder part of our network, which currently only consist of transposed convolutional layers, should be optimized. Changing the number of layers, or replacing them could lead to an performance boost.

As optimizing the network architecture might be difficult without understanding the underlying encoder-encoder architecture. The utilization of visualization tools to gain a deeper understanding of the encoder-decoder network is also useful when moving forward. With a better understanding of the underlying network selecting the appropriate hyper parameters for an increase in network performance should be easier.

As we in this thesis has proposed a new CNN architecture and a synthetic data generation pipeline a single-image input CNN version should be made. Hopefully a single-input version of our proposed method is able to perform better than PRN and other proposed solutions.

6.2.2 Training Data

One main issue addressed in this thesis is the lack of accurately labeled training data for the 3D facial reconstruction problem. One improvement would be to utilize accurate training datasets. Another improvement would be to further increase the realism of the proposed synthetic data generation pipeline. Through adding hair, facial expressions, skin texture and head accessories to the training data images the network should be able to generalize better on real world data. FaceGen provides the tools for adding these modifications to synthetic face renderings. Implementing these modifications to the synthetic data generation pipeline is the natural next step to improve the performance of our proposed method.

Appendices

Appendix A

Code

A.1 300W-LP Data Generation

```python
def run_posmap_300W_LP(bfm, uv_coords, image_path, mat_path, save_folder,
    uv_h=256, uv_w=256, image_h=256, image_w=256):

    # load image and fitted parameters
    image_name = image_path.strip().split('/')[-1]
    image = io.imread(image_path) / 255.
    h, w, c = image.shape
    info = sio.loadmat(mat_path)
    pose_para = info['Pose_Para'].T.astype(np.float32)
    shape_para = info['Shape_Para'].astype(np.float32)
    exp_para = info['Exp_Para'].astype(np.float32)

    # generate mesh from shape and expression parameters
    vertices = bfm.generate_vertices(shape_para, exp_para)

    # transform mesh to VCC by applying scale, rotation and transformation
    s = pose_para[-1, 0]
    angles = pose_para[:3, 0]
    t = pose_para[3:6, 0]
    transformed_vertices = bfm.transform_3ddfa(vertices, s, angles, t)
    projected_vertices = transformed_vertices.copy() # orthographic projection
    image_vertices = projected_vertices.copy()
    image_vertices[:, 1] = h - image_vertices[:, 1] - 1

    # crop image with key points
    kpt = image_vertices[bfm.kpt_ind, :].astype(np.int32)
    left = np.min(kpt[:, 0])
    right = np.max(kpt[:, 0])
    top = np.min(kpt[:, 1])
    bottom = np.max(kpt[:, 1])
    center = np.array([right - (right - left) / 2.0,
                       bottom - (bottom - top) / 2.0])
    old_size = (right - left + bottom - top) / 2
    size = int(old_size * 1.5)

    # randomize the cropping size
    marg = old_size * 0.1
    t_x = np.random.rand() * marg * 2 - marg
    t_y = np.random.rand() * marg * 2 - marg
    center[0] = center[0] + t_x
    center[1] = center[1] + t_y
    size = size * (np.random.rand() * 0.2 + 0.9)

    # crop and record the transform parameters
    src_pts = np.array([[center[0] - size / 2, center[1] - size / 2],
                        [center[0] - size / 2, center[1] + size / 2],
                        [center[0] + size / 2, center[1] - size / 2]])
    DST_PTS = np.array([[0, 0],
                        [0, image_h - 1],
                        [image_w - 1, 0]])
    tform = skimage.transform.estimate_transform('similarity', src_pts, DST_PTS)
    cropped_image = skimage.transform.warp(image, tform.inverse, output_shape=(
     image_h, image_w))

    # transform face position(image vertices) along with 2d facial image
    position = image_vertices.copy()
    position[:, 2] = 1
    position = np.dot(position, tform.params.T)
    position[:, 2] = image_vertices[:, 2] * tform.params[0, 0]  # scale z
    position[:, 2] = position[:, 2] - np.min(position[:, 2])  # translate z

    # uv position map: render position in uv space
    uv_position_map = mesh.render.render_colors(uv_coords, bfm.full_triangles,
     position, uv_h, uv_w, c=3)
    uv_position_map = uv_position_map.astype(np.float16)

    # save files
    io.imsave('{}/{}'.format(save_folder, image_name),
              np.squeeze(cropped_image))
    np.save('{}/{}'.format(save_folder, image_name.replace('jpg', 'npy')),
              uv_position_map)
```

Listing A.1: Python implementation of 300W-LP dataset generation pipeline

A.2 Random Texture Background

```python
def apply_random_background(image):
    dtd_path = '../Data/dtd/images' # texture dataset
    random_dir_path = get_random_subfolder(dtd_path) # random category
    random_img_path = get_random_subfolder(random_dir_path, is_image = True) # random image within category

    bg_img = io.imread(random_img_path)
    if (bg_img.shape[0] < 256 or bg_img.shape[1] < 256): # should not happen
        return image # return image without background image
    image_with_bg = image[:,:,:3].copy()
    cropped_bg_img = bg_img[:256,:256,:3].copy() # only use the top left 256x256 pixels
    background_mask = np.array(image_with_bg <= [0,0,0]) # if rgb values are 0, background should be shown
    image_with_bg[background_mask] = cropped_bg_img[background_mask]

    return image_with_bg
```

Listing A.2: Python implementation of applying random texture background

A.3 Training Code

```python
def main(args):
    os.environ['CUDA_VISIBLE_DEVICES'] = args.gpu
    batch_size = args.batch_size
    epochs = args.epochs
    train_data_file = args.train_data_file
    learning_rate = args.learning_rate
    epoch_limit = 10
    model_path = args.model_path
    resume_model_path = args.resume_model_path
    resume_model = args.resume

    # set tensorflow session GPU usage
    config = tf.ConfigProto()
    config.gpu_options.allow_growth = True
    sess = tf.Session(config=config)
    set_session(sess)

    # set model saving dir
    save_dir = args.checkpoint
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)

    # initialize TrainData class
    data = TrainData(train_data_file, weight_mask_path = '../Data/uv-data/
     uv_mask_final.png', pre_path = '')

    # set model to train on
    if resume_model:
        model = keras.models.load_model(resume_model_path)
    else:
        if args.model == 'mobilenet':
            model = MobileNetv2_PRN((256,256,6), args.alpha_mobilenet)
        elif args.model == 'prnet':
            network = resfcn256_keras()
            model = network.model
        else:
            raise NotImplementedError

    # set model optimizer and compile
    adam = optimizers.Adam(learning_rate = learning_rate)
    model.compile(optimizer = adam,
                  loss = 'mean_squared_error',
                  metrics = ['accuracy'])

    # initialize logging
    print("\n\nstarting training... \n\n")
    time_now = datetime.now().strftime("%Y_%m_%d_%H_%M_%S")
    fp_log = open("training_logs/log_" + time_now + ".txt","w")

    # calculate number of iterations on training and validation iterations
    num_train_iterations = int(math.ceil(1.0*data.num_training_data/batch_size))
    num_validation_iterations = int(math.ceil(1.0*data.num_validation_data/
     batch_size))

    #initialize early stopping variables
    epoch_val_losses = [9.999]
    times_not_saved = 0

    for epoch in range(epochs):
        reset_train_metrics = True
        model_saved = False
        np.random.shuffle(data.training_data)
        data.set_augmentation(rotate = True, channel_scale = True, dropout = True)

        # train network on training data
        for iters in range(num_train_iterations):
            batch = data(batch_size, data.training_index, data.num_training_data, data.training_data)
            data.training_index = data.get_updated_index(data.training_index, batch_size, data.num_training_data)
            if iters != 0:
                reset_train_metrics = False
            metrics = model.train_on_batch(x = np.array(batch[0]), y = np.array(batch[1]), reset_metrics=reset_train_metrics, class_weight = data.weight_mask)
            stdout.flush()
            stdout.write('\riters:%d/%d epoch:%d,loss:%f,accuracy:%f'%(iters + 1, num_train_iterations, epoch, metrics[0], metrics[1]))

        # calculate validation loss
        stdout.write('\n\rcalculating validation loss...')
        reset_val_metrics = True
        data.set_augmentation(rotate = False, channel_scale = False, dropout = False)
        for iters in range(num_validation_iterations):
            batch = data(batch_size, data.validation_index, data.num_validation_data, data.validation_data)
            data.validation_index = data.get_updated_index(data.validation_index, batch_size, data.num_validation_data)
            if iters != 0:
                reset_val_metrics = False
            val_metrics = model.test_on_batch(x = np.array(batch[0]), y = np.array(batch[1]), reset_metrics = reset_val_metrics)

        # early stopping
        if val_metrics[0] < min(epoch_val_losses):
            epoch_val_losses.append(val_metrics[0])
            model.save(model_path)
            model_saved = True
            times_not_saved = 0
        else:
            times_not_saved += 1
            if times_not_saved > epoch_limit:
                break

        stdout.flush()
        stdout.write('\nvalidation loss:%f validation accuracy:%f \n'%(val_metrics[0], val_metrics[1]))
        fp_log.writelines('[%s] epoch:%d learning rate:%f validation loss:%f, validation accuracy:%f, saved:%s\n'%(datetime.now().strftime("%Y_%m_%d_%H_%M_%S"), epoch, learning_rate, val_metrics[0], val_metrics[1], model_saved))

        # cut the learning rate in half every 5/10/3 epochs
        if ((epoch != 0) and (epoch %5 == 0)):
            learning_rate = learning_rate / 2

    fp_log.close()
```

Listing A.3: Python implementation of training code using Keras

Appendix B

Installation Manual

This appendix outlines the installation manual to run a demo of our proposed method. The installation manual and code is available at GitHub.

B.1 Prerequisites

The following libraries need to be install in order to run the code. We also provide a script to install these libraries using the Anaconda[1] package manager.

```
Python 3.6
Skimage
Scipy
keras-gpu
dlib
opencv2
```

[1]https://anaconda.org/

B.1.1 Anaconda as package manager

We recommend using the Anaconda package manager to install the required libraries. The following Anaconda environment was able to run demo.py on a Windows computer with GTX 1070 GPU.

```
conda create -name face-recon python=3.6
conda activate face-recon
conda install -c anaconda scipy
conda install -c anaconda scikit-image
conda install -c conda-forge dlib
conda install -c conda-forge opencv
conda install -c anaconda keras-gpu
```

B.2 Usage

1. Clone the repository.
   ```
   git clone https://github.com/olalium/face-reconstruction
   cd face-reconstruction
   ```
2. Clone the ICP repository to face-reconstruction folder.
   ```
   git clone https://github.com/ClayFlannigan/icp
   ```
3. Clone face3d repository to face-reconstruction folder.
   ```
   git clone https://github.com/YadiraF/face3d
   ```
4. Download trained model and shape predictor for keypoints.
   Navigate to the ned-data folder
   ```
   cd Data/net-data
   ```
   add these models:

   Shape predictor
   Trained CNN
5. Run the demo
   ```
   cd ../..
   python demo.py
   ```

The demo generates 3D faces for the the image pairs in the `test_images` folder.

Bibliography

[1] Yao Feng, Fan Wu, Xiaohu Shao, Yanfeng Wang, and Xi Zhou. Joint 3d face reconstruction and dense alignment with position map regression network. In *ECCV*, 2018.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`, Accessed: 10.05.2020.

[3] Standford University. Standford cs231n webpage. `https://cs231n.github.io/`. Accessed: 10.05.2020.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[5] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018.

[6] Andrew D. Bagdanov, Alberto Del Bimbo, and Iacopo Masi. The florence 2d/3d hybrid face dataset. In *Joint ACM Workshop on Human Gesture and Behavior Understanding (J-HGBU'11) ACM Multimedia Workshop 2011*, Arizona,USA, 12 2011.

[7] Xiangyu Zhu, Zhen Lei, Xiaoming Liu, Hailin Shi, and Stan Z. Li. Face alignment across large poses: A 3d solution. *CoRR*, abs/1511.07212, 2015.

[8] Michael A. Nielsen. Neural networks and deep learning, 2018. Accessed: 10.05.2020.

[9] Matthew Zeiler, Graham Taylor, and Rob Fergus. Adaptive deconvolutional networks for mid and high level feature learning. volume 2011, pages 2018–2025, 11 2011.

[10] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[11] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. 2020. https://d2l.ai.

[12] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015.

[13] Nitrish Srivastava Geoffrey Hinton and Kevin Swersky. Toronto university slideshow. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. Accessed: 05.06.2020.

[14] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 07 2011.

[15] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012.

[16] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[17] Igor Barros Barbosa, Marco Cristani, Barbara Caputo, Aleksander Rognhaugen, and Theoharis Theoharis. Looking beyond appearances: Synthetic training data for deep cnns in re-identification. *CoRR*, abs/1701.03153, 2017.

[18] Adam Kortylewski, Bernhard Egger, Andreas Schneider, Thomas Gerig, Andreas Morel-Forster, and Thomas Vetter. Analyzing and reducing the damage of dataset bias to face recognition with synthetic data. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2019.

[19] Volker Blanz and Thomas Vetter. A morphable model for the synthesis of 3d faces, 1999.

[20] Bernhard Egger, William A. P. Smith, Ayush Tewari, Stefanie Wuhrer, Michael Zollhöfer, Thabo Beeler, Florian Bernard, Timo Bolkart, Adam Kortylewski, Sami Romdhani, Christian Theobalt, Volker Blanz, and Thomas Vetter. 3d morphable face models - past, present and future. *ACM Transactions on Graphics (TOG)*, 2019.

[21] P. Paysan, R. Knothe, B. Amberg, S. Romdhani, and T. Vetter. A 3d face model for pose and illumination invariant face recognition. In *2009 Sixth IEEE International Conference on Advanced Video and Signal Based Surveillance*, Sep. 2009.

[22] Thomas Gerig, Andreas Forster, Clemens Blumer, Bernhard Egger, Marcel Lüthi, Sandro Schönborn, and Thomas Vetter. Morphable face models - an open framework. 2017.

[23] A. Jourabloo and X. Liu. Large-pose face alignment via cnn-based dense 3d model fitting. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4188–4196, 2016.

[24] Luan Tran and Xiaoming Liu. Nonlinear 3d face morphable model. *CoRR*, abs/1804.03786, 2018.

[25] T. Theoharis, G. Papaioannou, N. Platis, and N. M. Patrikalakis. *Graphics and Visualization: Principles & Algorithms*. A. K. Peters, Ltd., USA, 2007.

[26] Fabio Maninchedda, Christian Häne, Martin R. Oswald, and Marc Pollefeys. Face reconstruction on mobile devices using a height map shape model and fast regularization. *2016 Fourth International Conference on 3D Vision (3DV)*, pages 489–498, 2016.

[27] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. *ACM Trans. Graph.*, 21(3):355–361, July 2002.

[28] P. J. Besl and N. D. McKay. A method for registration of 3-d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, Feb 1992.

[29] Alexandros Lattas, Stylianos Moschoglou, Baris Gecer, Stylianos Ploumpis, Vasileios I. Triantafyllou, Abhijeet Ghosh, and Stefanos Zafeiriou. Avatarme: Realistically renderable 3d facial reconstruction "in-the-wild". *ArXiv*, abs/2003.13845, 2020.

[30] Xueying Wang, Yudong Guo, Bailin Deng, and Juyong Zhang. Lightweight photometric stereo for facial details recovery. *ArXiv*, abs/2003.12307, 2020.

[31] Fanzi Wu, Linchao Bao, Yajing Chen, Yonggen Ling, Yibing Song, Songnan Li, King Ngi Ngan, and Wei Liu. Mvf-net: Multi-view 3d face morphable model regression. *CoRR*, abs/1904.04473, 2019.

[32] Yu Deng, Jiaolong Yang, Sicheng Xu, Dong Chen, Yunde Jia, and Xin Tong. Accurate 3d face reconstruction with weakly-supervised learning: From single image to image set. In *IEEE Computer Vision and Pattern Recognition Workshops*, 2019.

[33] Elad Richardson, Matan Sela, and Ron Kimmel. 3d face reconstruction by learning from synthetic data. *CoRR*, abs/1609.04387, 2016.

[34] Aaron S Jackson, Adrian Bulat, Vasileios Argyriou, and Georgios Tzimiropoulos. Large pose 3d face reconstruction from a single image via direct volumetric cnn regression. *International Conference on Computer Vision*, 2017.

[35] Jiangke Lin, Yi Yuan, Tianjia Shao, and Kun Zhou. Towards high-fidelity 3d face reconstruction from in-the-wild images using graph convolutional networks. *ArXiv*, abs/2003.05653, 2020.

[36] Christos Sagonas, Epameinondas Antonakos, Georgios Tzimiropoulos, Stefanos Zafeiriou, and Maja Pantic. 300 faces in-the-wild challenge. *Image Vision Comput.*, 2016.

[37] Xiangyu Zhu, Zhen Lei, Xiaoming Liu, Hailin Shi, and Stan Z. Li. Face alignment across large poses: A 3d solution. *CoRR*, abs/1511.07212, 2015.

[38] Andrew D. Bagdanov, Alberto Del Bimbo, and Iacopo Masi. The florence 2d/3d hybrid face dataset. In *Joint ACM Workshop on Human Gesture and Behavior Understanding (J-HGBU'11) ACM Multimedia Workshop 2011*, Arizona,USA, 12 2011.

[39] M. Cimpoi, S. Maji, I. Kokkinos, S. Mohamed, , and A. Vedaldi. Describing textures in the wild. In *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2014.