Dense tracking of human facial geometry

Mikko Ronkainen

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 20.11.2017

Supervisor

Prof. Ville Kyrki

Advisor

Prof. Jaakko Lehtinen

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical
Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Mikko Ronkainen

Title Dense tracking of human facial geometry

Degree programme Master's Programme in Automation and Electrical Engineering

Major Control, Robotics and Autonomous Systems **Code of major** ELEC3025

Supervisor Prof. Ville Kyrki

Advisor Prof. Jaakko Lehtinen

Date 20.11.2017 **Number of pages** 93 **Language** English

Abstract

In recent years it has become possible to train very deep neural networks. Deep convolutional neural networks (CNNs) have been able to extract relevant information from human faces. Deep fully convolutional neural networks (FCNNs) have been able to do dense pixel-per-pixel segmenting of images. This thesis explored whether it is possible to combine these two approaches and train an FCNN using non-realistic synthetic data to do dense pixel-per-pixel geometry tracking of real-world human faces. Training data was generated by rendering because suitable training dataset was not readily available. UV mapping of the underlying 3D model was used as a basis for the geometry mapping. Neural network topology was based on the U-net design, which is an FCNN with skip connections. Loss function was a simple L1 loss between result and target images. UV gradient images were used as an additional loss term. Data augmentation was used to expand the training dataset and to create occlusions. Various visualization methods were developed to help assess the accuracy of the generated geometry. Results were encouraging as the final network successfully generalized to real-world facial images. Faces were accurately segmented out of the backgrounds, and plausible geometry was generated inside them. Data augmentation prevented the network from overfitting. The occlusion augmentation method enabled the network to inpaint geometry under various obstructions, even ones the network had never seen. Temporal stability of the generated geometry was satisfactory but could be improved, especially under occlusions.

Keywords machine learning, deep learning, neural networks, fully convolutional, synthetic data, rendering, texture mapping, uv mapping, data augmentation, occlusions, inpainting

Aalto University
School of Electrical
Engineering

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Mikko Ronkainen

Työn nimi Ihmiskasvojen geometrian tiheä seuranta

Koulutusohjelma Automaation ja sähkötekniikan maisteriohjelma

Pääaine Säätötekniikka, robotiikka ja autonomiset järjestelmät **Pääaineen koodi** ELEC3025

Työn valvoja Prof. Ville Kyrki

Työn ohjaaja Prof. Jaakko Lehtinen

Päivämäärä 20.11.2017 **Sivumäärä** 93 **Kieli** Englanti

Tiivistelmä

Viime vuosien aikana erittäin syvien neuroverkkojen kouluttaminen on tullut mahdolliseksi. Syvät konvoluutionaaliset neuroverkot (CNN) ovat pystyneet analysoimaan ihmiskasvojen piirteitä. Syvät täysin konvoluutionaaliset neuroverkot (FCNN) ovat pystyneet tekemään tiheää pikselipohjaista kuvien lohkomista. Tämä diplomityö tutki mahdollisuutta yhdistää nämä kaksi lähestymistapaa, eli onko mahdollista kouluttaa FCNN epärealistisella synteettisellä datalla tunnistamaan oikeiden ihmiskasvojen geometria tiheästi. Koulutusdata luotiin renderöimällä, koska sopivaa koulutusdataa ei ollut helposti saatavilla. Geometrian mallintaminen perustui renderöintiin käytetyn 3D-mallin UV-kartoitukseen. Neuroverkon topologia perustui U-net -malliseen verkkoon, joka on FCNN hyppy-yhteyksillä. Sakkofunktio oli yksinkertainen L1-sakko tulos- ja kohdekuvien välillä. UV-gradienttikuvia käytettiin lisäterminä sakkofunktiossa. Datan suurentamista käytettiin koulutusdatajoukon näennäisen koon kasvattamiseen ja peitteiden luomiseen. Erilaisia visualisaatiomenetelmiä kehitettiin luodun geometrian oikeellisuuden arvioimiseksi. Tulokset olivat rohkaisevia, koska lopullinen neuroverkko yleistyi onnistuneesti oikeisiin ihmiskasvokuviin. Verkko pystyi lohkomaan kasvot irti taustasta erittäin tarkasti ja niiden tilalle luotu geometria oli uskottavaa. Datan suurentaminen esti menestyksekkäästi neuroverkon ylisovittumisen. Peitteidenluomismenetelmä datan suurentamisen yhteydessä antoi verkolle mahdollisuuden luoda geometriaa näköesteiden alle. Luodun geometrian ajallinen vakaus oli tyydyttävää, mutta voisi olla parempaa erityisesti peitteiden alla.

Avainsanat koneoppiminen, syväoppiminen, neuroverkot, täysin konvoluutionaalinen, synteettinen data, renderöinti, tekstuurikartoitus, uv-kartoitus, datan suurentaminen, peitteet, täyttömaalaus

PREFACE

In late 2013 I came back to Aalto University to continue my studies after a hiatus of five years. While going through the courses, I stumbled upon the new computer graphics courses by Prof. Jaakko Lehtinen. They immediately piqued my interest as the content and course execution seemed very well done. I started to think that this could be my "exit strategy" for my very long overdue studies. Realizing that the course assignments did not have any upper bounds on points awarded (and some nice prizes), I brushed up my graphics maths and went to town with the exercises. After getting quite high scores, Jaakko promised that I could join the graphics group and do my thesis there.

In summer 2016 we started brainstorming different ideas for my final project. I wanted to do something practical while including GPU computing, rendering, and neural networks. Ultimately, we ended up with the subject of this thesis; trying to track human facial geometry densely using fully convolutional neural networks. At that time there were no publications of the exact method we were going to use. Not having too much background in machine learning, I spent the fall of 2016 studying it and familiarizing myself with the various software needed. The better part of 2017 we spent iteratively improving the network model, training data, and augmentations until we had the results we were looking for. In the end, this work contains some math and theory, but even more practical software engineering and exciting visuals. Just how I like it!

I would like to thank Prof. Jaakko Lehtinen and D.Sc. Miika Aittala for advising the thesis and keeping my to-do list full, Prof. Ville Kyrki for supervising, Markus Kettunen and Ari Silvennoinen for fruitful discussions at the office, Timo Aila from NVIDIA Research for good ideas and advice, Remedy Entertainment for models and textures, team Luolamiehet for abundant help during my general studies, remedial massage therapist Timi Hilakari for keeping me physically able to write, and last but not least my family (Jyrki, Maarit, Jenni, Toini, Risto) for supporting and keeping a roof over my head.

The calculations presented in this thesis were performed using computer resources within the Aalto University School of Science "Science-IT" project.

The author wishes to acknowledge CSC – IT Center for Science, Finland, for generous computational resources.

Espoo, 20.11.2017
Mikko Ronkainen

CONTENTS

LIST OF FIGURES

LISTINGS

ACRONYMS

BRDF Bidirectional Reflectance Distribution Function

CNN Convolutional Neural Network

CNTK Microsoft Cognitive Toolkit

CPU Central Processing Unit

CSC Finnish IT Center for Science

ELU Exponential Linear Unit

FCNN Fully Convolutional Neural Network

GPU Graphics Processing Unit

GUI Graphical User Interface

HDD Hard Disk Drive

HDR High Dynamic Range

HTML Hypertext Markup Language

PCA Principal Component Analysis

RELU Rectified Linear Unit

SGD Stochastic Gradient Descent

SSD Solid State Drive

SSHFS Secure Shell Filesystem

1 INTRODUCTION

1.1 RESEARCH BACKGROUND

Machine learning tries to answer the question of how could computers be trained to do useful work without having been explicitly programmed to do so. Machine learning subject area can be divided into many subcategories, and neural networks are one of them. Neural networks are constructed from simple functions which are composed together to form networks. The functional parts of neural networks are inspired by neuroscience, which is the scientific study of nervous systems. Neural networks have been shown to be able to solve many different and difficult problems, especially in the area of computer vision. [1, 2]

Research about automated face detection, i.e., determining if an image contains a face or not, dates back to the beginning of the 1970s [3]. Since then the methods have been improved so that real-time detection of multiple faces from an image or video is possible [4]. In contrast to face detection, face reconstruction strives to extract the underlying 3D model of the facial geometry. It can be done today accurately but usually needs multiple images of the face [5]. Face recognition methods try to detect if a specific person is in the image. Face recognition methods analyze human faces on a deeper level than face detection methods, but the recognition methods do not need to process as much facial detail as the reconstructions methods [6]. Automated single image facial geometry reconstruction is still an active area of research.

In recent years, the capabilities of machine learning methods have greatly increased, and this new era of machine learning has been named "deep learning" [1]. What is new is the massively increased computing power and memory capacity of the newer Graphics Processing Units (GPUs) and the lucky fact that the neural network training and inference map very well to GPU hardware architectures. Coupled with small algorithmic tweaks [1], this has allowed successful training of very deep neural networks with tens of millions of parameters, hence the name deep learning. Deep networks are beneficial because they can automatically learn, from large amounts of raw data, the representations needed for the task at hand [2]. Also, numerous Python-based GPU-accelerated deep learning frameworks have emerged in the last few years which allow rapid implementation and iteration of the network models [7, 8].

Convolutional Neural Networks (CNNs) have been used for image processing since the late 1980s [9]. Lately, they have been increasingly used in, for example, face detection [4] and face reconstruction [10]. CNNs have been shown to be able to extract low dimensional representations of facial pose and appearance parameters which could consequently be used for reconstructing the facial geometry [11]. Very recently, the CNNs have been extended to Fully Convolutional Neural Networks (FCNNs) that could do semantic segmentation, i.e., dense pixel-to-pixel mapping [12, 13]. In other words, FCNNs can take in an image, process it, and output another image.

Training of the deep neural networks in a supervised manner, i.e., with input/output pairs of known data, needs a large amount of annotated material. Training the networks to do face detection is relatively easy because the data annotations are light-weight and easily generated manually [4]. To train the networks to do facial reconstruction is considerably harder because the training data needs to be densely annotated, and the annotations are almost impossible to create by hand [10]. It is possible though to produce human face images and their dense geometry annotations using computer graphics. The obvious problem, in this case, is that the synthetic face images cannot be completely realistic. Instead of trying to generate as realistic face images as possible, it might be possible to go all the way to the other direction and generate a vast number of completely unrealistic facial images with greatly varying colors and textures. This way the only thing that is not randomized is the basic principles of light interaction with the geometry of the face. The network would have to learn to ignore everything else. Tobin et al. [14] proposed an idea called domain randomization in the context of robot machine vision and simple scenes of various objects. They trained their vision neural network using greatly varying non-realistic renders of the scene and got the network to generalize to real-world. They argued that because of enough variation in the training data, the neural network learned to see the reality as just another variation. This thesis has a similar idea, but we came up with it independently.

Abundant processing power and storage capacity is nowadays available in computing clusters, and it is feasible to generate large sets of densely annotated training images by rendering [15]. Modern 3D modelers/renderers like *Blender* [16] have the ability to output realistic images, be completely scripted, and be run from the command line in computing clusters [16]. In addition, it has been shown that the effective sizes of training datasets can be increased by, for example, randomly swapping the color channels of the training images [17]. This is called data augmentation [1]. Generalization in machine learning means that the trained network works well on input it has not seen in the training data. Overfitting means the network has learned the training data well but does not generalize well to unseen input.

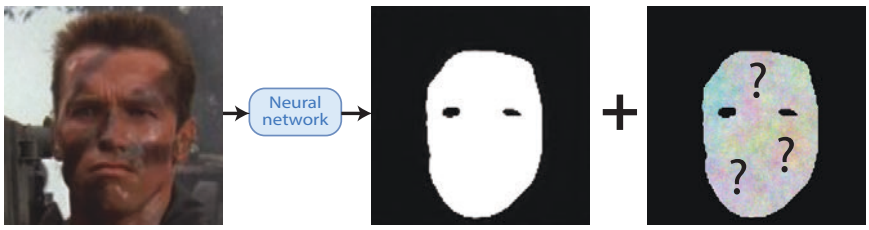


Figure 1.1: The main research goal visualized. A Fully Convolutional Neural Network (FCNN) is given a real-world human face image as an input, and the result should be the face segmented out of the background and densely filled with useful geometry information.

Data augmentation makes it possible to use smaller datasets while preventing the network from overfitting and getting it to generalize well to new data [1].

1.2 RESEARCH PROBLEM

CNNs have been shown to be able to extract relevant information from human faces, and the extension of CNNs, FCNNs, has been shown to be able to do dense pixel-to-pixel mappings in various situations. Could it be possible to train an FCNN to extract human facial geometry information from an image and map it to another presentation densely pixel-per-pixel (see figure 1.1)? An FCNN can be trained in a supervised manner using a large number of densely annotated image pairs, but those are not readily available for real-world human faces. Could it be possible to generate these training pairs by rendering? Could the rendered data be non-realistic and still make it possible for the trained FCNN to generalize to real-world images? What is the mapping that could be generated by rendering and that is possible for the FCNN to learn? Could the rendered training dataset be extended using data augmentation to prevent overfitting and to enable occlusion detection?

1.3 RESEARCH GOALS

The main research goal of this thesis is to train a Fully Convolutional Neural Network (FCNN) using non-realistic synthetic data to do dense human facial geometry tracking on real-world images. If the network is given an image of a real human face as an input, the output should be a same sized image with the face segmented out of the background and filled with a useful dense geometry mapping. The main research goal is visualized in figure 1.1.

This main goal can be divided into seven subgoals:

- Design and implement a suitable network topology and a loss function using a Python-based deep learning framework, e.g., *Microsoft Cognitive Toolkit (CNTK)* [8]. If the topology and loss functions are good enough, the network should produce non-blurry, sharply defined, results with a reasonable number of parameters.
- Design and implement a method to generate a large quantity of non-realistic synthetic training data in a reasonable amount of time. The dataset should be big enough and have a precise enough mapping that the training of the FCNN is possible. Also, rendering a new dataset should take less than 24 hours.
- Make the network generalize well to real-world images of human faces. Even if the network is trained on completely non-realistic data, the network should be able to detect real-world faces and extract relevant geometry information from them.
- Make the feature detection of the network invariant to lighting, texturing, scaling and positioning. Even if the subject in the real-world image is under extreme conditions, the network should continue to output believable segmentation and geometry mapping.
- Enable the network to inpaint believable geometry under occlusions. If the face is, for example, partially occluded by sunglasses, the network should be able to detect them and generate geometry underneath.
- Implement geometry mapping visualizations and, in general, make the evaluation of results easy. It should be possible to see how well the generated geometry matches the real subject, and comparing results between different network versions should be effortless.
- Make the network generate temporally stable geometry mappings when applied to video. When looking at a video, the generated geometry should not flicker and jump around from frame to frame.

1.4 RESEARCH SCOPE

The scope of this thesis includes the neural network design, synthetic data generation, data augmentation, training of the network, geometry mapping visualizations, and evaluation of the results. The facial geometry is identified in the canonical 2D parameterization. Because of time constraints, the extraction of 3D geometry from the parameterization has not been included.

2

BACKGROUND AND RELATED WORK

To give some background to our method this chapter first gives a brief overview of machine learning and neural networks. Then, a selection of face detection, recognition, and reconstruction methods are presented. More emphasis is given to facial reconstruction methods, especially those that have been developed recently and use neural networks and synthetic data in one way or another.

2.1 MACHINE LEARNING AND NEURAL NETWORKS

If we have a parametric function $f(\mathbf{x}; \theta) = \mathbf{y}$, how should the parameters θ be adjusted so that, with input $\mathbf{x}$, the output of the function matches $\mathbf{y}$ as closely as possible? Machine learning, and especially its most common form, supervised machine learning, can be used to solve this problem. In supervised learning, the parameters θ are iteratively adjusted until the output of the function cannot be further made more accurate. To make these kinds of small adjustments, a number of training pairs $(\mathbf{x}_i, \mathbf{y}_i)$ are needed in addition to some metric that tells how well the function is performing. [1, 2]

The function performance can be measured by a separate loss function that tells how far the output of the function is from the desired values. In the case of fitting a simple line function to a collection of points, the loss function could be visualized as the sum of the distances of the points from the line. If the points lie exactly on the line, the loss becomes zero, and further away the points are from the line, the bigger the loss becomes. Especially with more complex functions, the loss could be visualized as a multi-dimensional hilly landscape. Lower values of the loss function are valleys, and higher values are hills. The lower it is possible to travel in this loss landscape, the better the function under optimization performs. With complex functions, the loss landscape has numerous valleys and hills. Even if it seems that the loss is now small, because you are at the bottom of a valley, it is very much possible that there exists another valley somewhere else that is even deeper and thus better. [1, 2]

To adjust the parameters of the function, an algorithm called gradient descent is used. The gradient of the loss function can be visualized as an arrow that, at any point in the loss landscape, points towards the direction of the greatest ascent. To travel towards lower loss function values, that is, towards the bottoms of the valleys, a step into the opposite direction of the gradient should be taken. When repeated enough many times, a loss function minimum is reached. The

problem with gradient descent is that, after reaching a bottom of a valley, the algorithm cannot continue. This means that the optimization could get stuck into a local minimum even though much better minima would be available further away. [1, 2]

If a very accurate gradient is calculated using all the available training data, it is very much possible for the gradient descent algorithm to get stuck at a local minimum. The accurate gradient calculation is also time-consuming and usually needs the whole dataset to be kept in memory. This is not feasible for very large datasets. The Stochastic Gradient Descent (SGD) algorithm solves this problem by calculating a less accurate gradient from a smaller, randomly selected, part of the training dataset. This small selection of training samples is called a minibatch. The more random gradient is faster to calculate, and because it is noisy, it will help the gradient descent algorithm to escape from local minima. [1, 2]

Instead of using just one function as the target of the optimization, a composite of multiple functions, a network, can also be used. These networks are usually called neural networks because the functional parts are loosely inspired by neuroscience. A neural network has an input layer, any number of hidden layers, and an output layer. The adjacent layers can be fully connected to each other with distinct weights for each connection, and the values that flow through the network can be modified with activation functions. The hidden layers are called so because the training data does not give any desired output for them. Instead, when trained, the network is free to come up with its internal representations. The training of a neural network like this is enabled by the backpropagation algorithm. For it to work, one requirement is that the functions that compose the network are differentiable. If this is the case, the gradient calculated at the output layer can be backpropagated through the network, and all the parameters of the network can be adjusted accordingly. [1, 2]

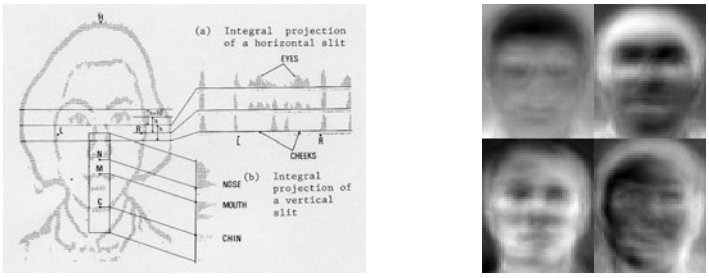
The machine learning algorithms used today are largely the same as in the 1980s. The massive increase of computing power, storage space, and memory amounts in recent years has enabled the training of networks that were previously thought very hard to train. The modern general processing GPUs have been instrumental in making the training process faster, as their architecture suits the task very well. Some other smaller developments, e.g., using the Rectified Linear Unit (RELU) activation function, have also played their part in making the training of deeper networks easier. Modern neural networks can have tens of millions of parameters, and the networks can be very deep with tens, if not hundreds, of layers. Deep learning, the name given to this new era of machine learning, reflects this fact. A good example of the deep learning advantage is the application of deep neural networks by Krizhevsky et al. [18] in the 2012 ImageNet

competition. Their results were impressive, almost halving the error rate compared to the best competitors. [1, 2]

The functions that the network is composed of can also perform filtering using convolutions. These kinds of networks are called Convolutional Neural Networks (CNNs). Convolutions can be though as small filters that are slid over the larger underlying signal and that produce a new, modified, signal. Convolutions are especially well suited for processing images. One of the first applications of CNNs was a method to recognize handwritten digits, developed by LeCun et al. [19]. Many convolutions, with differing filter values, can be applied to the same image. This allows the extraction of different features from the image. The hierarchical nature of the CNNs means that, at the first layers, the extracted features are edges, and then continuing deeper inside the network the features become parts, and parts become objects. The structure of CNNs has been directly inspired by the concepts in visual neuroscience. [1, 2]

An ordinary CNN usually has fully connected layers at the end and will produce a one-dimensional vector as the final output [19]. In Fully Convolutional Neural Networks (FCNNs), the fully connected part at the end is replaced by a convolutional upsampling part [12]. This means that, for FCNNs, both the inputs and outputs can be images. FCNNs have been successfully used for dense image semantic segmentation. Semantic segmentation means understanding the image at the pixel level, i.e., giving each pixel of the input image a meaningful class [12, 20]. The training of FCNNs has been made easier with the introduction of the concept of skip connections. Skip connections connect the downsampling and upsampling parts of the FCNN directly. Skip connections have been shown to help gradients propagate from the output towards the input side of the network, transfer high-frequency detail from the input side to output side, and help to avoid singularities in the loss landscape [1, 20, 21].

Designing and training the neural networks, especially the deep ones, would be very time-consuming if the training code had to be reimplemented at low-level with every design iteration. To get fast enough training speeds, using GPUs is necessary. The training code has to be executable on GPUs, and efficient transfer of training data to the GPUs needs to be implemented. To help make the neural network design process, code generation for the GPUs, data transfer to the GPUs, and the training process easier, numerous software frameworks have been developed in recent years. Good examples of these software frameworks are *Microsoft Cognitive Toolkit (CNTK)* [8], *Tensorflow* [7], and *PyTorch* [22]. With these frameworks, the neural network can be designed at high-level with the Python scripting language. Changes to the network topology can be made easily, and the framework will take care of converting the high-level network description into optimized low-level code that is ready to be run on GPUs. [7, 8, 22]



(a) [3] (b)

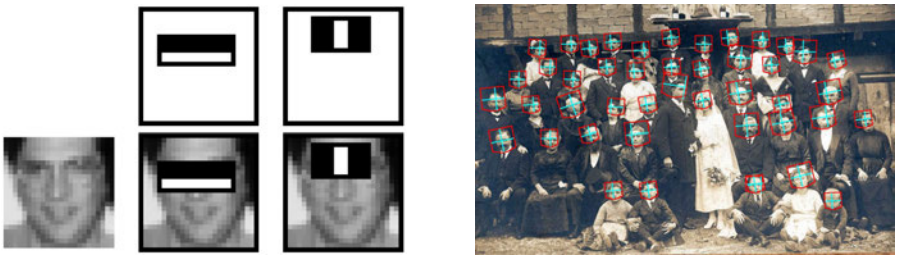
Figure 2.1: One of the first face detection systems was developed in the early 1970s by Sakai et al. [3]. Their algorithm was based on analyzing slices of a binary image of the head as shown in (a). Face recognition based on eigenfaces, as proposed by Turk et al. [23], was one of the first commercially viable methods for face recognition. Visualizations of eigenfaces are shown in (b) (Copyright of AT&T Laboratories Cambridge).

2.2 FACE DETECTION AND RECONSTRUCTION

The techniques for processing human faces in images are broadly classified into three categories by Datta et al. [6]: face detection, face recognition, and face reconstruction. Face detection tells us if there exists a human face in an image in the first place, and can give the approximate location and size of the face. Face recognition goes a step further as it can identify the actual person in the image. Face reconstruction does not necessarily have anything to do with identification but rather finding out the underlying facial geometry, i.e., the facial shape, of the human pictured.

One of the first automated face detection systems was developed in the early 1970s by Sakai et al. [3]. The input was a 5-bit grayscale image with a resolution of 140x208. The image was processed with an edge detecting filter and then thresholded to obtain a binary image containing contours of the face. See figure 2.1a for an example. The binary image was then scanned slice-by-slice from top to bottom. Each slice was analyzed while an elaborate state machine kept track whether the image contained a human or not. The face had to be centered in the image and could have only a small amount of tilt in any direction. This method did not work at all if the person in the image had glasses or a beard.

The method of using eigenfaces for recognizing humans from pictures was introduced by Turk et al. [23] in 1991. It was one of the first accurate and fast enough methods to be used commercially. The method captured the relevant variation in a collection of facial images, that is, the principal components of the distribution, into eigenvectors. The eigenvectors can be visualized as ghostly faces, eigenfaces, as is shown in figure 2.1b. Any picture of a human face could then be de-



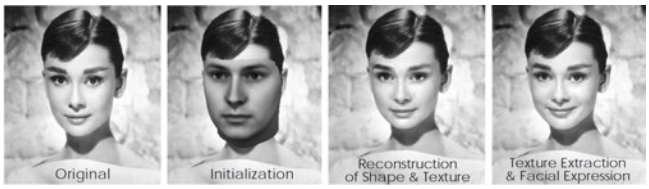
(a) [24] (b) [4]

Figure 2.2: Face detection speed was greatly increased by a method introduced by Viola et al. [24]. The method was based on fast evaluation of rectangular filters and machine learning. Some filters are visualized in (a). Convolutional neural networks were used by Osadchy et al. [4] for fast face detection and pose estimation. (b) shows how the algorithm performed on a somewhat difficult image.

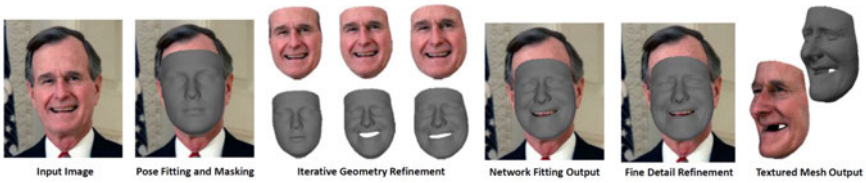
constructed into a linear combination of eigenvectors, and inversely, reconstructed with the same linear combination of the same eigenvectors. The face recognition could be done by comparing the weights of the linear combinations as the weights would be similar between two images of the same person.

The speed of detecting faces and generating the bounding boxes around them was dramatically improved by Viola et al. [24] in 2001. Their proposed method was based on the idea of the integral image, rectangular feature filters, and machine learning. The integral image, or summed area tables, was an intermediate representation of the image that allowed rapid summation of pixels in arbitrary rectangular areas. The rectangular feature filters are illustrated in figure 2.2a. The filter calculated the sum of the pixels in the white area which was then subtracted from the sum of the black area. The filter could thus find intensity variations between arbitrary rectangular areas rapidly. Hundreds of thousands of possible combinations of these filters existed for a 24x24 pixel detection window, and machine learning was used to select few thousand of the most relevant filters for human face detection. Applying the resulting combination of filters to images was fast, and faces could be detected and tracked near real-time with contemporary hardware.

In 2007, Osadchy et al. [4] published a novel method for simultaneously detecting faces and estimating their poses in real-time using convolutional neural networks. Figure 2.2b shows how the network was able to detect multiple faces in one image including their yaw from left to right and in-plane rotation. Their network topology was similar to the one used by LeCun et al. [19] in 1998 to recognize handwritten digits. The training data consisted of real human faces which were manually annotated with the pose data. The method could do



(a) [25]



(b) [10]

Figure 2.3: Facial geometry and texture reconstruction using a 3D morphable model was proposed by Blanz et al. [25]. The steps of their algorithm are shown in (a). Richardson et al. [10] used a CNN and an iteration process to fit the morphable model to the input image. The steps are shown in (b).

face detection and pose estimation at the same time quicker and more accurately than previous methods that did the tasks separately.

In 1999, Blanz et al. [25] introduced a new technique to reconstruct human facial geometry from a single image using a 3D morphable face model. They started by scanning hundreds of real human faces with a 3D laser scanner. The scan produced accurate vertex positions and colors. The generated 3D face models were processed so that they all came into full correspondence with each other. Using Principal Component Analysis (PCA) decomposition, an average face model and a set of basis face vectors were created. PCA was also done for the vertex color data. This allowed a parametric creation of 3D face models and their textures. The process of reconstructing the facial geometry from a single image started with a coarse manual alignment of the average face model over the image. An analysis-by-synthesis loop was then repeated where the face model was rendered over the image, and the factors of the principal components were adjusted until an optimization minimum was reached. Finally, additional detailed facial texture information was extracted from the image as the color PCA components did not contain enough high-resolution data. Steps of this process are shown in figure 2.3a.

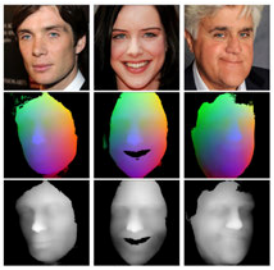
More recently, in 2016, Richardson et al. [10] proposed a method to extract 3D morphable model parameters from real-world images using a CNN. The steps of their algorithm are visualized in figure 2.3b. They started by aligning the average morphable face model over the image using another posing algorithm. The face was segmented out of the background and fed to the CNN along with a rendered version of the current morphable face model. The CNN was trained to out-

put a correction term to the morphable model parameters to make the rendered face model match the segmented real face image better. The face model rendering and parameter correction calculation were repeated iteratively multiple times to increase the quality of the reconstruction. The CNN was completely trained on synthetic data generated using the 3D morphable face model. Because fine facial details could not be extracted using this method, they were later captured with a separate shape-from-shading algorithm.

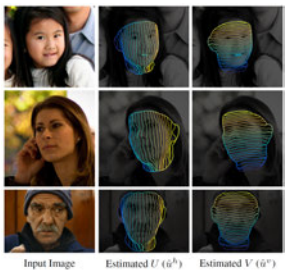
Richardson et al. [26] improved their previous method of [10]. The problem with their previous method was that it needed separate initialization of the average face model over the input image and that the method could not extract fine details without an external algorithm. The improved algorithm could do both in one go using two connected networks, a CNN and an FCNN. The CNN was applied iteratively to the input image and a coarse geometry using a 3D morphable model was recovered. A depth map was generated from this model, and the map was fed to the FCNN along with the input image. The FCNN was trained to do fine detail reconstruction, i.e., shape-from-shading, within the depth map in one shot without iteration. The CNN was trained in a supervised manner with synthetic data rendered using the 3D morphable model, and the FCNN was trained in an unsupervised manner using real-world images.

Kim et al. [11] introduced a method that could estimate a wide variety of 3D morphable model parameters from a real-world image in a single shot using a CNN. The estimated parameters included facial pose, facial shape, facial expression, skin reflectance, and scene illumination. No iteration was needed to refine the results. Before feeding the real-world images to the network, their variety was reduced by segmenting the faces out of the backgrounds using facial landmark detection. The network was trained exclusively on synthetic training data derived from the 3D morphable model. To get the parameter distributions of the synthetic training data to better model the real-world distributions, a novel breeding method was introduced that used real-world facial images to update the training set. A network trained with the breeding outperformed a network trained only on synthetic data.

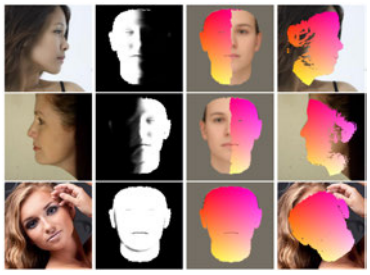
Tewari et al. [27] were able to train an autoencoder CNN completely unsupervised using only real-world images. No separately generated synthetic data was used. The CNN was able to extract 3D morphable model parameters including facial pose, facial shape, facial expression, skin reflectance, and scene illumination in a single iteration. The method relied on a novel analytical and differentiable decoder layer that used the morphable model parameters to reconstruct the input image. It was possible to train the network in an unsupervised manner because the loss could be backpropagated through the decoder layer. With a method like this, the real-world distribution of the mor-



(a) [28]



(b) [29]



(c) [30]

Figure 2.4: The method of Sela et al. [28] generated dense correspondence and depth map images. Their results are shown in (a). Güler et al. [29] generated dense correspondences using quantized regression which resulted in outputs shown in (b). Yu et al. [30] proposed a method that could estimate dense 2D pixel flow between the input image and a front-facing template mesh. Visualization of their method is shown in (c).

phable model parameters could be captured the best, and the network could generalize well to real-world images.

After the work on this thesis had started, multiple papers were published that had similar ideas regarding CNNs and dense geometry generation. Sela et al. [28] proposed an FCNN that could map a real-world human face image to a dense correspondence image and a dense depth map image. Examples of these images are shown in figure 2.4a. In contrast to most of the previously mentioned methods, this method did not use a 3D morphable model in the reconstruction process. The correspondence and depth map images were then used with a separate algorithm to extract the 3D mesh of the face. The network was trained with synthetic data with a wide range of facial shapes, facial poses, facial materials, lighting conditions, and background textures. A surprising result was that even though the synthetic data was created with a limited generative model, the network could generalize beyond the limited scope of the training material.

Güler et al. [29] trained an FCNN to estimate dense correspondences between image pixels and a 3D template mesh projected onto a 2D deformation-free space, or in other words, a UV space. Their approach was very similar to ours, but instead of plain regression, they

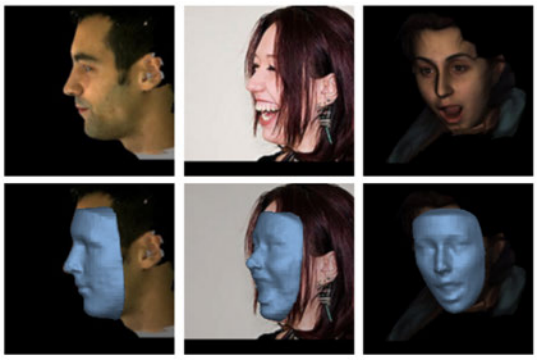


Figure 2.5: Results of the method proposed by Jackson et al. [31]. They used FCNNs to regress from the input image into a 3D volume directly.

used quantized regression. The mappings generated by this method are illustrated in figure 2.4b. The training dataset was generated from real-world images that had existing facial landmark annotations. The dataset generation was done by first fitting the 3D template mesh over the image using the landmarks and then rasterizing the mesh using colors representing locations in the deformation-free space.

Yu et al. [30] used an FCNN to generate dense facial correspondences between the input image and a 3D morphable face model. Two images were generated from the input image: a 2D flow image and a matchability mask image. The 2D flow image estimated the flow between the input image pixels and a synthetic rendering of an average frontal face. The matchability image indicated which correspondences were valid inside the average face. These images are visualized in figure 2.4c. In the beginning, the network was trained with synthetic data generated from a 3D morphable model. Later, the network was refined using annotated real-world images. Simple rectangular occlusions were also added to the training data which made the model more robust against obstructions over the faces.

In late 2017, Jackson et al. [31] published a novel method that used FCNNs for direct 2D-to-3D facial geometry reconstruction. No intermediate fitting steps were used. The training dataset consisted of real-world images and corresponding 3D binary volumes that modeled the faces in the input images. Inside the volume, if a voxel was inside the face, it was given a value of 1, and 0 otherwise. The 3D binary volumes were created using a 3D morphable model that was already fitted to the input images. The actual 3D facial geometry was recovered by generating the iso-surface of the regressed binary volume. Results of this method are shown in figure 2.5.

3 OUR METHOD

A synthetic 3D model has a canonical, continuous, and dense 2D parameterization which is not dependent the 3D model pose or deformation. Pose means the translation and rotation of the model, and deformation means the independent movement of the model vertices. We used the 3D model to render large quantities of randomized, deformed, and non-realistic training data. The training data consisted of randomized facial images and corresponding 2D parameterization images. Using this data, we trained a neural network to detect faces from images and to identify the pixels inside the faces with the 2D parameterization. This gave a 2D correspondence between the face pixels and the 3D model.

This chapter goes into the details of our method, and the chapter is divided into sections as follows:

- Section 3.1, general workflow, describes the repeated high-level work that was needed to iteratively improve the results.
- Section 3.2, UV color mapping, explains how we used the 2D parameterization of the head model for the geometry mapping.
- Section 3.3, synthetic data generation, explains the details necessary to create a diverse enough training dataset.
- Section 3.4, neural network design, shows the final topology of the network and the particular features of its layers.
- Section 3.5, loss function design, illustrates how the training sample flowed through the network and how the final loss value was calculated from the resulting images.
- Section 3.6, data augmentation process, details every augmentation method we used to increase the effective size of the training dataset, and especially how the occlusion augmentations were created.
- Section 3.7, training process, presents the software architecture used for the training.
- Section 3.8, visualization methods, details all the techniques we used to generate evaluation images from the network outputs.
- Section 3.9, results evaluation, explains the technical details behind managing the evaluation and inspection of a large number of training runs and result images.

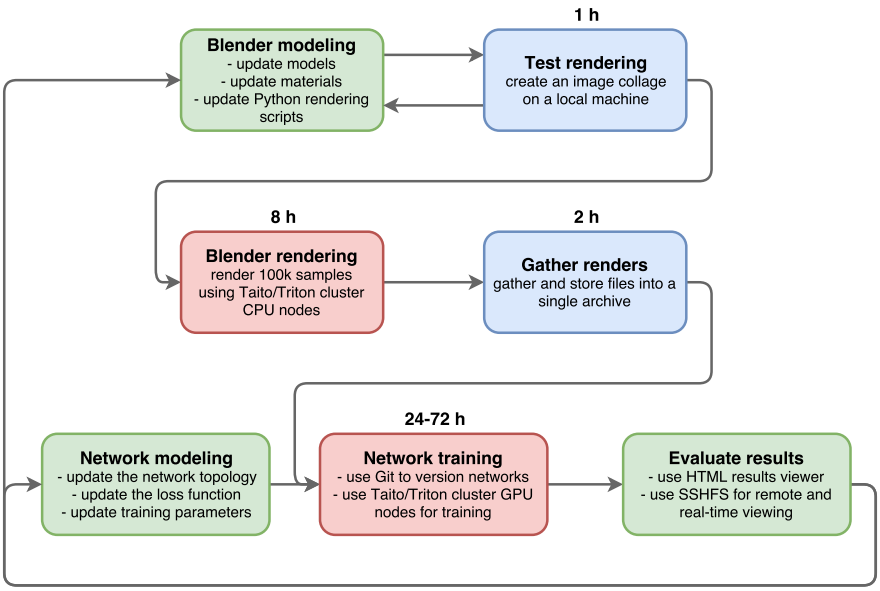


Figure 3.1: Overview of our research workflow. Green boxes mean manual work that did not have an exact duration. Blue boxes mean small-scale non-parallel jobs. Red boxes mean massively parallel and long-running jobs on either the CPU or GPU nodes in the Aalto Triton or CSC Taito computing clusters.

3.1 GENERAL WORKFLOW

Figure 3.1 depicts the general workflow of our method from a higher level. We started by editing the head model and associated materials in the *Blender* [16] 3D computer graphics toolset. After having adjusted rendering parameters, a test rendering was executed on the local computer. This usually took about one hour and in the end generated a large collage with hundreds of renderings combined. Using this collage, we then determined if further rendering adjustments were necessary. If they were, this adjust-render-evaluate method was repeated until the renderings looked satisfactory.

Updated rendering related files and scripts were then uploaded to the Aalto Triton or CSC Taito computing clusters. There, a massively parallel rendering job was launched that generated 300 000 individual rendered images. The job took on average eight hours to finish. This big set of files was then gathered together and compressed into a single archive, taking over 10 gigabytes of storage space.

While the rendering was underway, we manually tuned the neural network parameters. The validity of the changes was tested on a local computer using quick test runs. We created up to 20 different training configurations where usually a single change was made to either network, loss, or augmentation parameters. After the training dataset generation had finished, we sent it together with 20 different training jobs to the GPU nodes in the computing cluster.

After the models had been under training for a maximum of three days, we started to evaluate the results. The evaluation was done by mounting the result directory to a local computer using Secure Shell Filesystem (SSHFS) and then looking at the generated evaluation images using the custom-made browser-based viewer. We compared the results from all the 20 different jobs visually and by using some simple evaluation metrics. After we had come to conclusions with the current results, we started the process from the beginning by going back to updating the network and rendering models. In the end, this process was repeated about 15 times which means that around 300 24–72 hour training runs were performed.

3.2 UV COLOR MAPPING

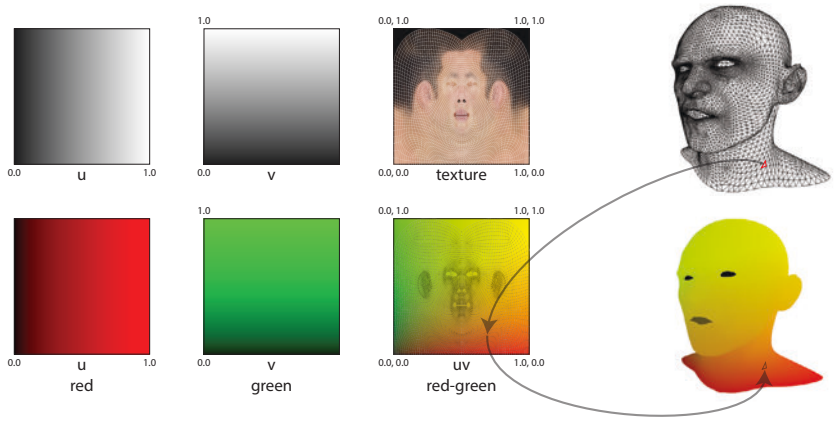


Figure 3.2: UV color mapping. Same idea as in texture mapping but the texture is replaced by the procedural color representation of the UV coordinates. The 3D triangle mesh is ”opened up“ and flattened to the 2D UV space so that the triangles do not overlap.

The key insight to our method is understanding the UV color mapping we used. Figure 3.2 illustrates this idea. A 3D model consists of triangles, and each triangle has three vertices. These vertices each have at least two attributes: a 3D world position and a 2D UV coordinate. The position can have any arbitrary values, but the UV coordinate is constrained to values between 0.0 and 1.0. It could be thought that the 2D UV values are real-valued positions on a square whose side length is 1.0. When a 3D model is created, its UV mapping is generated at the same time automatically with some manual tuning afterward. The UV map generation can be visualized as opening the 3D model up and then warping and flattening the triangles onto the 2D UV square. The flattening is done so that the triangles do not overlap in the UV space, that is, each triangle is mapped to a unique area. After the UV mapping has been done, it does not change even if the 3D world positions of the model vertices are changed. The change

could be caused by, for example, expression animation or by applying scaling or deformation operations to the 3D model. Figure 3.2 illustrates the UV dimensions both in grayscale and with color; red for the U dimension and green for the V dimension. If these separate color channels were combined, it would result in a color gradient shown in the red-green square. The color choice has been arbitrary; it affects only on the visualizations of the results.

Usually, the UV mapping is used to texture the 3D model using textures like one in the texture square of figure 3.2. We replaced this normal texture with the red-green-yellow gradient texture generated procedurally straight from the UV coordinates. This resulted in the red-green-yellow renders of heads as seen in, for example, figure 3.10. Looking at a render like that, each pixel of the head tells precisely where that pixel belongs to in the UV space. Now, if we can generate this same UV color mapping from an arbitrary face image, we can then use the mapping to sample textures, create own visualizations using static or procedural textures, swap faces between two people, or recreate the 3D mesh of the face.

Many corresponding input and UV images can be rendered using just one head model while deforming it in many ways. Even if the appearance the model changed, the UV mapping would stay the same. These input and UV image pairs can then be used to train a neural network to do the mapping. The network can become invariant to facial geometry changes and would be able to map very different facial geometries to the same underlying UV mapping.

3.3 SYNTHETIC DATA GENERATION

For generating the images in the training dataset, we decided to use *Blender* [16]. The choice was quite easy, as Blender is free and open source, and it has comparable features to commercial 3D modeling software. In addition, Blender comes with the Cycles rendering engine which has all the features we needed. We could use the nodes material system to design arbitrary materials and output resulting renders in multiple file formats. Blender is heavily based on the Python scripting language, and that allowed broad access to the internals of the program with the use of external scripts. This meant that we could completely automate and randomize the rendering process from start to finish. Finally, blender supports Linux and running from the command line without opening a Graphical User Interface (GUI), which meant the rendering of a lot of images could be parallelized on a computing cluster.

We received the head model and its textures from Remedy Entertainment. The dataset also contained 178 different expressions for the head model. A basic scene with the head, single directional light, and a background was set up. Materials for the head and background

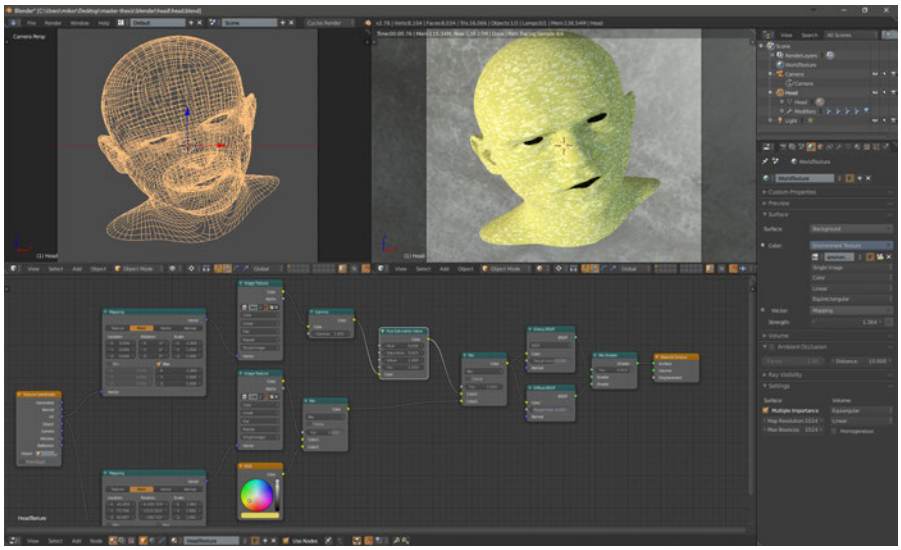


Figure 3.3: A Blender modeling session. On the top left is the wireframe of the model, on the top right is the rendering result, and on the bottom is the current material modeled with the nodes system.

were designed using the Blender nodes system for materials. See figure 3.3 for an example of a typical editing session.

For the realistic backgrounds, we obtained 50 High Dynamic Range (HDR) environment textures with a resolution of 8000x4000 from *sIBL Archive* [32]. These textures had equirectangular projections, which meant that they could be projected 360 degrees around the head model without distortion. Two samples of environment textures can be seen in figure 3.4. For non-realistic face textures, 70 different textures, with an average resolution of 1600x1600, were obtained from *Textures.com* [33]. Most of these textures were of natural materials and some were completely synthetic. Examples of these material textures can be seen in figure 3.5, along with the realistic face texture and the $1/f$ noise texture.

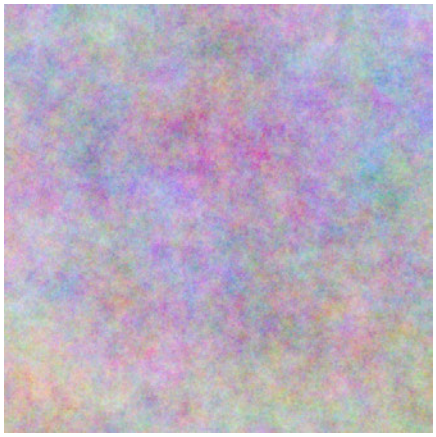
We started with very simple gray renderings (see figures 3.6 and A.1) and then proceeded to more complex materials as it became evident that with the simpler materials it was not possible to successfully train the network. The idea behind using the $1/f$ noise textures (see figures 3.7 and A.2) was that the $1/f$ noise follows the power spectra of natural images [34]. The results were not that good, and we further experimented with different kinds of materials: completely realistic and completely non-realistic (see figures 3.8 and A.3). In the end, we ended up with a head material that was a mixture of realistic face textures and non-realistic material textures. The material texture was first blended with a random solid color, and this blend was further blended with a realistic face texture. The blending amounts were also randomized. The actual Bidirectional Reflectance Distribution Function (BRDF) of the head material was a random blend between diffuse

Figure 3.4: Two examples of environmental textures with equirectangular projections. These were used for backgrounds when rendering and for generating occlusion masks when augmenting.

and glossy BRDFs. Samples of images with these blends can be seen in figure 3.9. This figure also has the final amount of randomization in all the rendering parameters. Here is a detailed list of all the randomizations we did per each rendered training sample:

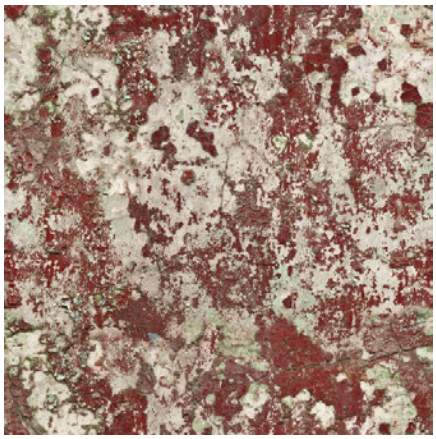
- The head model scale was randomized in all three dimensions separately using a uniform random distribution.
- The head model rotation was randomized around all three axes separately using a Gaussian random distribution with zero mean. The standard deviations were set so that the 99.9th percentiles of the angles were as follows: pitch angle $\pm 40°$, roll angle $\pm 45°$, and yaw angle $\pm 90°$.
- The head model was randomly deformed with Blender's simple deform modifiers of twist, bend, taper, and stretch.
- The head model was randomly deformed with Perlin noise [35].

(a) Realistic texture



(b) 1/f noise texture

(c) Material texture 1



(d) Material texture 2

(e) Material texture 3

(f) Material texture 4

Figure 3.5: A sample of different textures used for rendering the head model.

- The camera focal length was randomized using a uniform random distribution.
- The camera shift in the XY-plane was randomized using a Gaussian random distribution with zero mean. The standard deviation was set so that the 99.9th percentile shift moved the head model at most two-thirds out of the frame.
- The direct light direction was randomized using a uniform random distribution. The direction was restricted so that the light never shone behind the head model.
- The noise texture scale and translation were randomized using a uniform random distribution. The scaling was identical in all three dimensions.
- A random environment texture was selected from a set of 50. Its scale and rotation were randomized using a uniform random distribution. The scaling was identical in all three dimensions, and the rotation was different for each of the three axes.
- A random material texture was selected from a set of 70. It was rotated and scaled in the same manner as the environment texture.
- A random realistic facial texture was selected from a set of 2. The texture was randomly flipped around the vertical axis.
- A random solid color was selected which was blended with the material texture with a random amount. This resulting mix of a solid color and a material texture was then blended with the selected realistic facial texture with a random amount.
- The head material BRDF mix between diffuse and glossy was randomized. Also, the roughness of the glossy material was randomized.
- A random expression was selected from a set of 178 and applied to the head model. Because there were much more expressions with an open mouth, the randomization process was biased so that there would be a 50/50 split between expressions with an open mouth and expressions with a closed mouth.
- The Blender Cycles renderer seed value was randomized.

One training sample consisted of three images: the input image, the target UV image, and the target mask image. All the images had a resolution of 128x128 pixels, and they were created from the same randomized rendering parameters. The UV and mask images were rendered with the help of another mask image that segmented out only the frontal face area of the head model. This was done because

Figure 3.6: Head renders with gray background and face textures. A collage of these renders can be seen in figure A.1.

Figure 3.7: Head renders with 1/f noise background and face textures. A collage of these renders can be seen in figure A.2.

Figure 3.8: Head renders with realistic background textures and non-realistic face textures. A collage of these renders can be seen in figure A.3.

Figure 3.9: A collage of head renders with realistic background textures and mixed material/realistic face textures. These have the final amount variation between all the randomized rendering parameters.

Figure 3.10: A collage of head renders with the input, target UV and target mask images. These image triplets formed the training samples.

we were not interested in teaching the network the geometry of the neck or back of the head. Eyes and inside of the mouth with teeth were not included with the original head model, and we did not have assets to model them, so they were left out of the renders. Eyes and inside of the mouth were rendered as black in all three of the training sample images. The resulting image triplets are illustrated in figure 3.10.

Our rendering loop started by doing all the rendering parameter randomizations. It then rendered the same scene three times, adjusting internal blender settings to generate the different images in the training sample. A random 16 character string was created, and with it, the file names for current rendering sample. Our file naming scheme allowed later the grouping of the rendered files into training samples, whose order could be randomized easily without having to worry about mixing the images.

All the rendering results were stored in floating-point precision. We used the EXR file format with 16-bit floats and ZIP compression. Halving the precision from 32 bits to 16 bits did not affect the results, but reduced the size of the training dataset by a factor of two. The additional built-in ZIP compression reduced file sizes even further, as the UV and mask images were composed of somewhat repeating values. The floating-point precision was essential, especially with the UV images, where the pixel needs to store a real-valued coordinate in the UV space. At first, we used 8-bit precision with the UV images, as is used in the PNG format, and the resulting network did not perform well. In addition, the physically based lighting of the Cycles renderer and HDR environment textures generated HDR input images. We wanted to save this information so that the data augmentation process would have as much information available as possible.

Antialiasing with 64 samples per pixel was used for input and mask images but disabled for the UV images, where only the center of the pixel was sampled. At the internal facial geometry boundaries, for example between nose tip and upper lip when looked from above, two adjacent pixels could have a considerable distance in the UV space. This is why we reasoned that antialiasing the UV images did not make sense, as it would blur the boundary and generate an invalid UV value. When sampling the pixel centers, it could be possible to find a pixel where the center is inside the model geometry, but part of the pixel is already outside. We did not want to include these pixels in the UV image. We decided to render the mask image with antialiasing which would make these boundary pixels grayscale. These grayscale values could then be thresholded out, and a little bit smaller mask could be created. This mask has pixels that are all 100% inside the facial geometry and could be used to remove all ambiguous pixels from the UV image.

The rendering speed was about one training sample per second on a modern four-core desktop machine. Most of the time was spent rendering the input image; the UV and mask images were much faster to render in comparison. We kept the sample count per pixel low, at 64, on the input image to keep rendering times in control. Low sample counts caused some noise to be left in the input images, but we reasoned that it does not matter as noise is added anyway in the augmentation process.

Rendering 100 000 training samples would have taken too long on a single machine, so the rendering was offloaded to Aalto Triton and CSC Taito computing clusters. The CPU utilization of a single Blender rendering process was rather low. It was found out that an optimal combination was to request a job allocation with eight cores and then launch eight separate Blender rendering processes which each was told to use four threads. This way the rendering could reach near 100% CPU utilization, and we witnessed the fastest total rendering

times. Each job output the rendered images into the local temporary storage, and after having finished rendering, stored all the images into a tar-archive and sent the file to a shared work directory. After all the jobs had finished, the smaller archives were gathered together and combined into one single large archive. The size of an archive containing 100 000 training samples was around 10 gigabytes.

3.4 NEURAL NETWORK DESIGN

At the time of starting this project, there were two machine learning Python frameworks that we looked at, *Tensorflow* [7] and *Microsoft Cognitive Toolkit (CNTK)* [8]. It quickly became clear that the syntax of Microsoft Cognitive Toolkit (CNTK) was simpler, and the examples and tutorials were easier to understand. Back then, Tensorflow did not have an official well-working GPU implementation on Windows. In contrast, CNTK natively supported GPU training without syntax changes on both Windows and Linux. CNTK also promised easy-to-use multi-GPU training out-of-the-box. The main development of this project was to be done on Windows, and all the training was to be done on Linux servers. So, it was an easy decision to go with the CNTK framework.

Our final network topology is illustrated in figure 3.11. The design relied heavily on the U-net design proposed by Ronneberger et al. [20]. The decision to go with the U-net design was influenced by our prior good experiences with it. We decided to call our network UV-net as an homage to its inspirator and because the network did UV mapping. The network was an FCNN with skip connections. The input was one image, and the output was three different images of the same size. The skip connections were a vital part of the network, as they were able to preserve the high frequency detail of the input image. Without the skip connections the training process was not able to converge to any satisfactory result.

In figure 3.11 the cyan boxes represent the data flowing through the network, the tensors. Tensors are multidimensional arrays of numerical values. In our case, the first tensor was a 128x128x3 floating-point array representing an RGB color image with a resolution of 128x128. The tensor then got processed through the seven levels of the network. At the contracting side of the network, the spatial size of the tensor got reduced but, feature map size increased. In the middle, the tensor size was 2x2x525. At the expanding side, the spatial size was gradually increased, and feature map size decreased. At the end, the network output a tensor with a size of 128x128x4. This included two channels for the UV image and two channels for the two different mask images: normal mask and occluded mask. The skip connections were implemented by first making sure that the spatial dimensions of

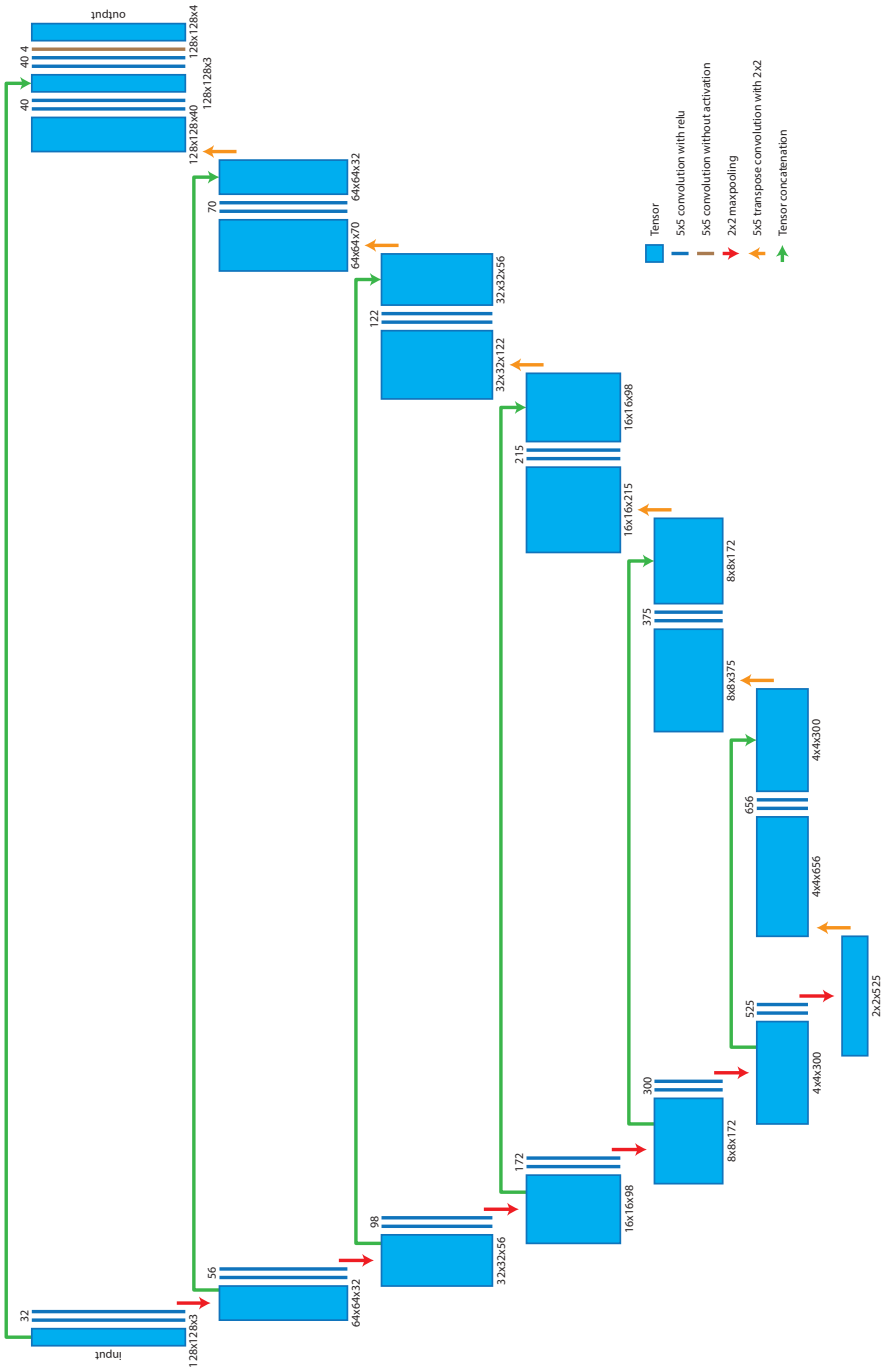


Figure 3.11: The final topology of the neural network. It was a fully convolutional network with skip connections between down and upsampling portions. Downsampling was done with max-pooling and upsampling with transpose convolutions. Numbers above the convolution layers convey their feature map sizes.

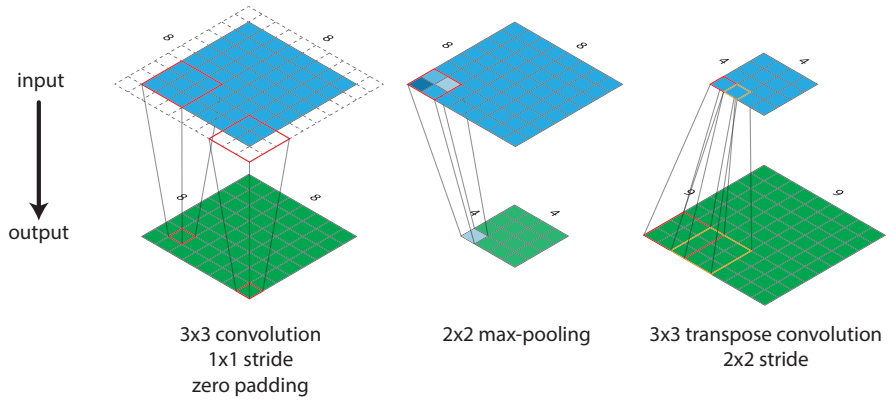


Figure 3.12: A visualization of the different layers used in the network. Zero padding was used with the normal convolution layers to keep the output same size. Max-pooling layers selected the maximum value inside a 2x2 area, effectively reducing the size by a factor of two. Transpose convolutions with a stride of two were used to upscale. 3x3 convolution filter size is used here for illustration purposes; the final filter size was 5x5.

all tensors on the same level were identical and then just concatenating the tensors together along the feature map axis.

Figure 3.12 illustrates the different layers we used to operate on the tensors. Convolution layers with a filter size of 5x5 were used to extract features from the tensors. For example, the first tensor with three feature maps was convolved with 32 different convolution filters to generate a new tensor with 32 different feature maps. Convolutions used a stride of 1x1, and the input was zero padded to prevent the spatial size reduction of the tensors. The actual downscaling of the tensors was done with 2x2 max-pooling layers [1]. Each feature map was scanned through looking at disjoint 2x2 pixel areas at a time. Whatever pixel had the highest value was selected. This reduced the spatial dimensions of the tensors by a factor of two. Max-pooling layers also introduced some invariance towards the internal shifting of features. Upscaling of the tensors was done with transpose convolution layers with a filter size of 5x5. These effectively did the same as the convolution layers but in reverse. If a stride larger than one was used, 2x2 in our case, the spatial dimensions of the tensors were increased. The output size would not be the same as if scaled up by a factor of two. This was remedied by just cropping the tensors back to the right size.

We used exponentially growing feature map sizes as fully convolutional neural networks like ours need large feature map sizes to work well [20]. We started with 32 initial feature map size and increased that by a factor of 1.75 for every level. On the upscaling side of the network, the feature map sizes were derived from those on the downscaling side by multiplying by 1.25. This was because the network

needed more expressivity on the upscaling part for generating dense pixel mappings.

We preferred using two convolution layers in succession with smaller filter sizes to using one convolution layer with a larger filter size. This increased the effective receptive field size of the convolutions while not increasing needed parameter count as much. For example, with two 5x5 filters, the effective receptive field size was 9x9. Two 5x5 filters needed 50 trainable parameters, and one 9x9 filter would have needed 81 trainable parameters.

The convolution layer filter values were prepared using uniform Glorot initialization, described by Glorot et al. [36]. All the layers had the bias term enabled. After testing all the available activation functions, we decided to use the RELU [1]. It was fastest to train and gave as good results as the more recent ones, like the Exponential Linear Unit (ELU) [37]. The last convolutional layer did not have any activation function.

Every part of the network model creation was parameterized. The full network creation code can be seen in listing A.1. This parametrization allowed easy changing of the network topology. Parameters could also be saved to a file and read back later. This made it simple to generate hyperparameter sweep runs on the computing clusters.

3.5 LOSS FUNCTION DESIGN

A fully convolutional neural network like ours can be trained in a supervised manner using input/output image pairs. We had hundreds of thousands of these image pairs thanks to the synthetic training data generation process. The network was first given in an input image, and then the generated output images were compared with known ground truth images. The difference between these images could then be used to iteratively update the network parameters to gradually make the output more and more like the ground truth images.

A general overview of the image data flow during processing one training sample and calculating its loss is illustrated in figure 3.13. The process was started by reading in the current training sample which consisted of input, target UV, and target mask images. The target mask image was thresholded so that the grayscale pixels were turned to black. The target UV image was multiplied with this new slightly smaller target mask image. This removed all the ambiguous border pixels from both the target UV and target mask images.

The original input image and the thresholded target UV and target mask images were then input into the data augmentation process. The process output an augmented input, target UV, and target mask images. In addition, it generated two new mask images: an occluded target mask image and an eroded target mask image. The occlusions

in the occluded target mask image reflected the colored occlusions generated onto the augmented input image. The eroded target mask image had a morphological erosion image processing operation applied which increases the size of the black areas and decreases the size of the white areas.

The augmented input image was given to the neural network which generated three output images: result UV, result mask, and occluded result mask. The UV image generated was not masked straight out of the network, so it was multiplied by the augmented target mask to create the final masked result UV image. Result mask and occluded result mask images were left as is.

A gradient image describes how much pixel values change between adjacent pixels. We knew that the UV image should mostly have very smooth changes within itself, i.e., no bumpiness or splotchiness in the gradient images. This is because the UV color mapping process inherently generates smoothly varying colors from the smooth UV space. Occasional sharp jumps might be generated by, for example, the nose geometry. Because the UV image should be smooth, its gradient image should, therefore, contain only small gradient values and the sum of the pixels in the gradient image should be low. Thus, low gradient image pixel sum could be used as an additional loss term to help steer the training process towards smoother result UV images.

Gradient images were calculated from both the augmented target UV image and the non-masked result UV image. Gradients were calculated in both X and Y directions for both of the U and V channels, which generated four new images. Figure 3.13 depicts the gradients as vector magnitudes for U and V channels using only one image per each. A problem with the gradients was large values at the edges of the face, mouth and eyes. We were not interested in these as they did not introduce any new actionable information to the process. We remedied the problem by multiplying the gradient images with the previously generated eroded target mask image. This had the desired effect of removing the large edge gradient values from the gradient images.

Figure 3.13 labels the images, between which the loss was calculated, with green and red. The final loss value calculation method is shown in figure 3.14. The difference between two images was calculated by doing a pixel-by-pixel subtraction. In the difference image, green values depict positive values and red negative values. The absolute operation was then applied to the image resulting in positive values only. All the pixel values were summed over the image resulting in one scalar value. A total of seven of these scalar values were produced, and they were summed together to form the final loss value. To summarize, our loss function was an L1 loss of the UV, UV gradient, mask, and occluded mask images. We tried different combinations of L1 and L2 losses and, in the end, the L1 loss always

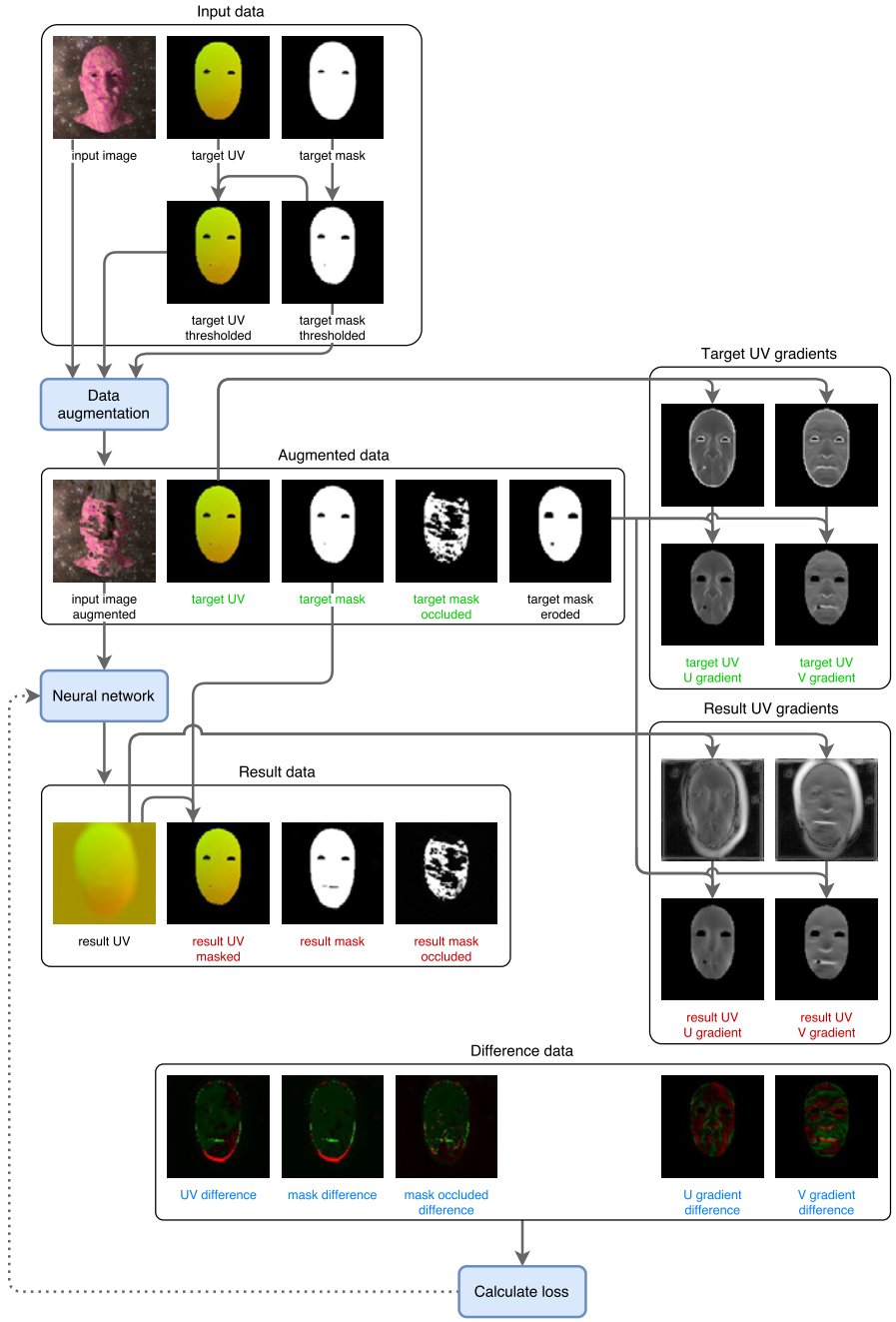


Figure 3.13: An overview of the data flow when evaluating one training sample. The input data is first preprocessed and then fed to the data augmentation process. From the augmented data, the augmented input image is given to the neural network. The network will output result data which is slightly post-processed. The difference data calculation has not been visualized using arrows but with colors. The result images with red labels are subtracted from the target images with green labels to produce the difference images with blue labels. The actual final loss value is calculated from the difference images with the method illustrated in figure 3.14. The images on the right side of the figure on this page are the UV gradient images. Gradient images tell how much pixel values change when moving from one pixel to the next. The eroded target mask is used to eliminate large gradient values that exist on, for example, the outer edges of the face.

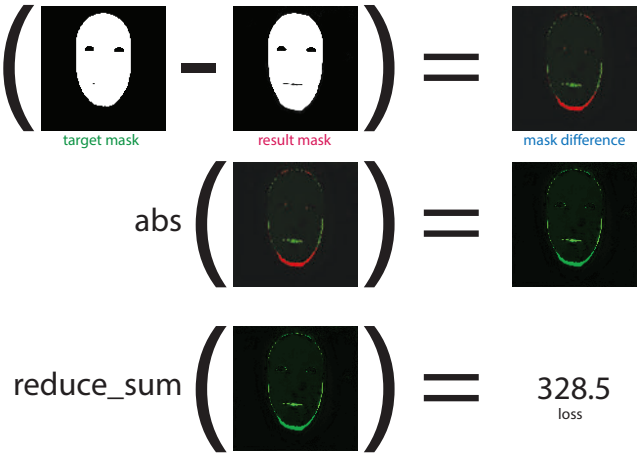


Figure 3.14: A visualization of the L1 loss function evaluation for the images. The difference between the images was calculated by a pixel-wise subtraction. Positive values are depicted in green color and negative in red color. The absolute operator turned all pixel values to positive, and the reduce-sum operator summed up all the pixels together to one scalar value.

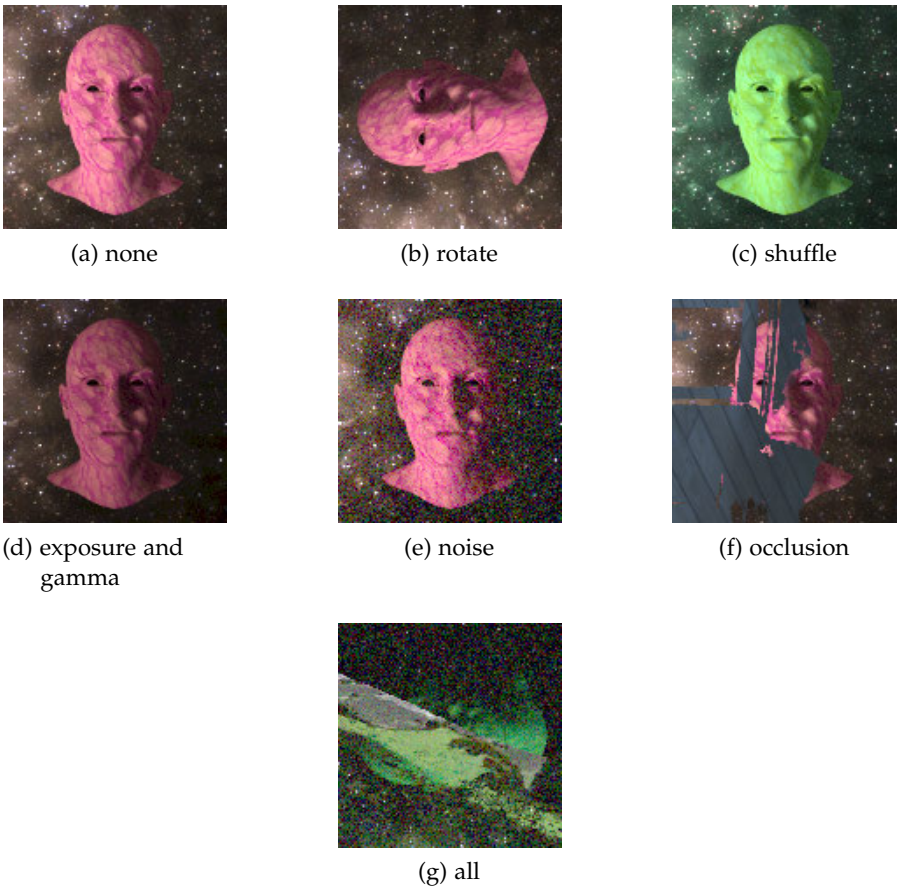
outperformed the L2 loss when it came to the sharpness of the resulting images. The L2 loss tended to produce blurred mask edges and blotches inside the UV mapping. Full code implementation of the loss function can be seen in listing A.2.

To optimize the loss, we used the Adam optimization method [38] provided with CNTK. We started with a learning rate of 0.0001 and a momentum of 0.9. After the loss improvements had tapered off, the learning rate was reduced to 0.00001 and momentum increased to 0.99. Also, the magnitude of the noise augmentation was gradually decreased down to zero during the training process.

3.6 DATA AUGMENTATION PROCESS

Without data augmentation, the network did overfit quite quickly. That is, the network learned the training dataset well, but would not generalize to test or real-world images. Data augmentation expanded the existing training dataset many times over. We were able to train a network continuously for ten days without overfitting by combining all our augmentation methods shown in figure 3.15: rotate, shuffle, exposure, gamma, noise, and occlusions. If the input image in figure 3.15 is compared to the final image with all the augmentations enabled, it is evident that the modifications to the original image could be drastic.

Rotation augmentation (see figure 3.15b) randomly picked a rotation of 0, 90, 180, or 270 degrees. The input, UV, and mask images were rotated. It should be noted that flip augmentations could not



(a) none (b) rotate (c) shuffle

(d) exposure and gamma (e) noise (f) occlusion

(g) all

Figure 3.15: Examples of different types of input image augmentations. The final image (g) is the same as (a), but with all augmentations at their extreme.

be used in our method, as, for example, flipping along the vertical axis would have made the UV mapping ambiguous. This was not a problem with rotations. The shuffle augmentation (see figures 3.15c and A.4) separated the RGB color channels of the input image, shuffled them randomly, and then combined them back into a new image. This effectively increased the training dataset size by a factor of six. The shuffling was done on per color channel, not per pixel. The exposure augmentation multiplied the input image with a random number. The gamma augmentation raised the input image to a random power. These two were used to increase the brightness variation of the input image (see figures 3.15d and A.5). The noise augmentation (see figures 3.15e and A.6) added Gaussian noise with random magnitudes to the input image. This was one of the most important augmentations as it made the model much less prone to overfitting. Adding noise made the model generalize better, be less sensitive to small variations in input, and aided the training process to avoid local minima.

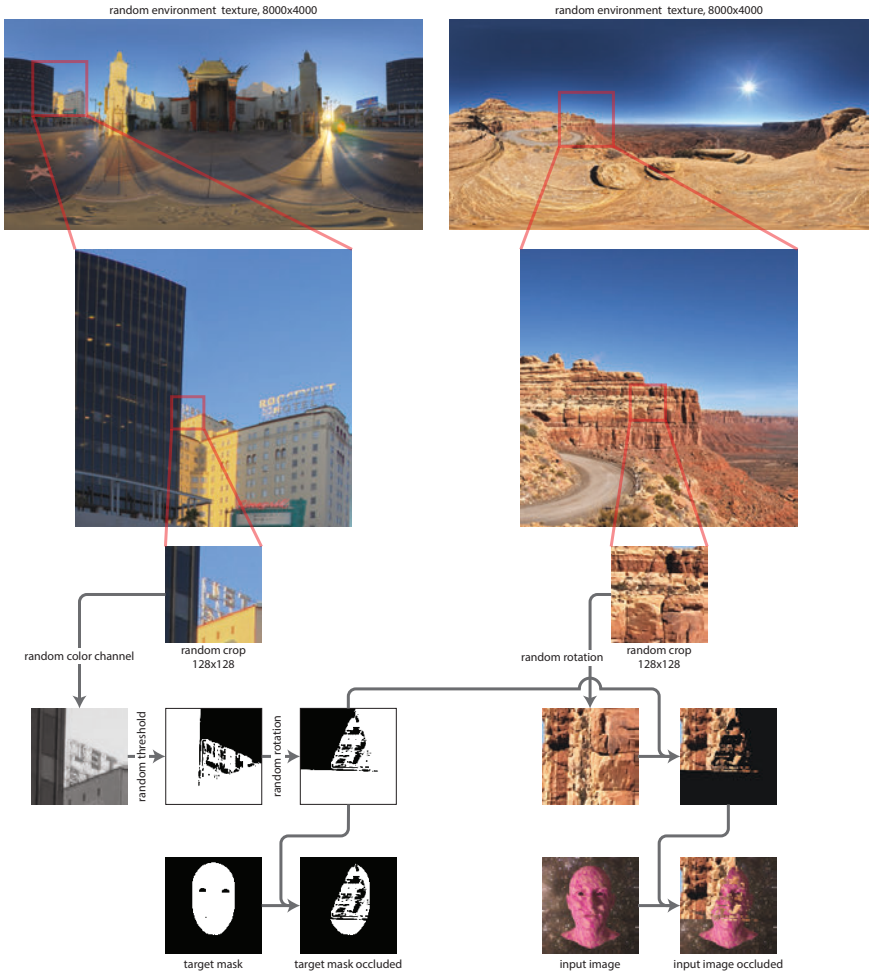


Figure 3.16: An overview of the occlusion generation process. Two environment textures were selected at random, and a small random area was selected from each of them. These two small images were then used to create the occlusion mask and a color for it.

As we evaluated the results, we noticed that any occlusion in front of the face usually degraded the mapping results considerably, usually completely breaking the tracking. Occlusions were, for example, long hair, beards, mustaches, sunglasses, hats and random objects. We did not have assets to add these to the training data, and it would have been infeasible to try to model every kind of possible occlusion. The only way to try to fix this was to add occlusions to the training data, on-the-fly, using augmentation.

Adding rectangular occlusions to the training data, filled with either solid colors or noise, did not improve results. These tended to bias the network to detect every occlusion as a rectangular one. Instead, we decided to use a method that would generate more natural-looking occlusions. The general overview of our method is shown in figure 3.16 and examples of results in A.7. We obtained 50 HDR environment textures from *sIBL Archive* [32]. The textures were of very

Figure 3.17: A collage of head renders with all the augmentations applied. Images like these were ultimately fed to the network in the training process. Figure 3.9 is the same collage without any augmentations.

high resolution, 8000x4000 being the most common one. First, we selected two random pictures from the set. Then, we independently selected a random 128x128 crop from each. The first crop would be used to generate the occlusion mask and the second one would be used to apply color to the mask. This two-step approach was used to increase the randomness of the process, as the mask and its resulting color were not linked together.

Figure 3.18: A collage of augmented head renders with the input, target UV, target mask, and occluded target mask images. The augmentation process mainly touched the input image and generated the occluded target mask. Figure 3.10 is the same collage without any augmentations.

To generate the occlusion mask, a random color channel was selected from the first cropped image and then thresholded. Our thresholding method did not generate binary images, but grayscale images with gradients. This enabled the creation of transparent occlusion masks on top of the input image. The process of creating the threshold image was repeated until the ratio of black to white pixels was within some predetermined range. This range was established by visually inspecting collages of thresholded images and adjusting the target range until there were about equal amounts of black and white. An example of this kind of collage can be seen in figure A.8. After successful thresholding, the image was randomly rotated and then multiplied with the target mask to generate the occluded target mask. The random crop from the other texture was also randomly rotated and multiplied with the thresholded image. The resulting color image was then blended over the original input image to create the final occluded input image.

Our rendered data was in floating-point precision, the previously mentioned augmentations were not restricted in range, and our occlusions were based on HDR textures. This all meant that our final aug-

mented input image could have values outside the 0.0–1.0 floating-point range. The real-world images that we used for evaluation were in the JPEG or PNG formats. They had 8-bit color channel precision, that is, discrete values between 0 and 255. If a color value like this was converted to floating-point by dividing by 255.0, the result was a value between 0.0 and 1.0 with 255 steps. As a result, for the two final augmentations, we used clipping and quantization. Clipping restricted the color value range between 0.0 and 1.0, and quantization compressed values between this range to 255 distinct values.

A collage of input images with all augmentations applied can be seen in figure 3.17. Figure 3.9 is the same collage without any augmentations. The augmented collage shows that the final training material could be exceedingly non-realistic and hard even for a human to understand. The occlusion method was able to generate a diverse set of colored masks, and sometimes the method even covered the whole face in the image. Figure 3.18 shows how the whole augmentation process modified the training sample images in general. It can be compared with figure 3.10. Most of the augmentation methods only touched the input image, the exception being rotation which rotated all images. Occlusion augmentation did not touch the target mask but created a new one, the occluded target mask. This enabled the neural network to detect occlusions while still generating a full, non-occluded, mask image.

3.7 TRAINING PROCESS

The training process was started by first thinking on a high level what modifications we wanted to do to the network model, loss function, and data augmentation. Because the computing clusters we used had a limit of 20 simultaneous GPU jobs, at most 20 different combinations were created. Usually, only a single parameter was changed. Each combination was separately committed to a Git version control repository and tagged with a logical and unique tag name. Then separate jobs, with the tag name as a parameter, were sent to the computing cluster. The first thing the job did was to create a clean working environment and clone the network code from the Git repository with the given tag name. It would also copy and decompress the 100 000 training samples to a local job-specific temporary directory. The job generated a unique run-ID using the tag name and current time. This ID was used to create an output directory to which all the results of the current job would be saved. Doing it this way made it absolutely sure that the network was actually trained with the parameters and training data that it was supposed to. Also, when looking at the results, we could be sure that result files in a specific directory belonged to a specific training run.

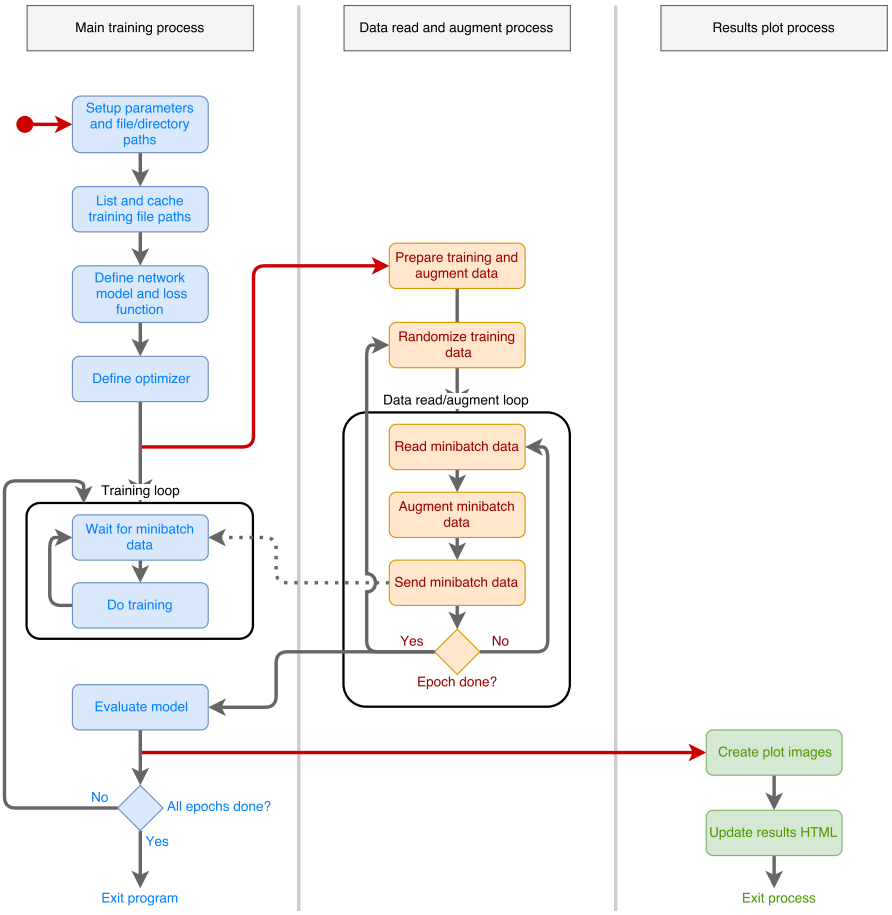


Figure 3.19: An illustration of the software processes used during the training and the data flow between them. Red arrows mean starting of a new process.

After the initial setup was completed, the job launched our Python-based training script. A general overview of the software processes launched at this moment is given in figure 3.19. The script first listed all files and their paths in the training dataset, grouping them into training sample triplets. CNTK was initialized, and it compiled the static graph of the network and readied the target GPU for training. The main training loop started to run after that.

All of the data augmentation was implemented in pure Python code, and that proved to be a performance bottleneck. One minibatch contained 40 samples of data and augmenting it took on average the same time as training it. This meant that the GPU utilization could only reach 50% at best. The problem was solved by moving the data augmentation to a separate process using the Python multiprocessing module. By doing this, we were able to read and augment the data asynchronously with the training and increase the GPU utilization to almost 100%.

The data reading and augmentation process started by caching the most used files into memory, for example, the environment textures

which were used to generate occlusions. The list that contained all the training sample paths was randomized, and the data reading and augmenting loop was entered. The loop started by reading in one minibatch worth of training samples and then augmenting them. The ready minibatch of data was sent to the main training process by using pipes in the Python multi-processing module. This loop was continued until the end of the training sample path list was reached, or a predetermined number of training samples had been processed. At this point, an epoch was signaled as done. The data reading and augmenting process returned to the sample path list randomization and started the loop again.

Meanwhile, the main training process was reading the minibatches of data from the pipe and sending them to the GPU to be processed. This was continued until the epoch done signal was received from the data read and augment process. At this point, the model was evaluated with different evaluation images. The result images were sent to a new plotting process, again, because of Python performance reasons. Plotting all the result images, even after optimizations, took upwards of 30 seconds. The plotting process also updated the result Hypertext Markup Language (HTML) file. Next, the main training process returned to the training loop if not enough epochs had been done or the maximum training time limit had not yet been reached. If the limits had been reached, the main process saved the current model parameters to a file and exited.

3.8 VISUALIZATION METHODS

The generation of the UV mapping allowed us to create various visualizations. They were used to assess the accuracy and stability of the mapping. Accuracy meant how close the generated geometric features matched the actual real geometry and stability how well the features stayed in place while applying the network to video. Any visualization that could be created in the 2D UV space could be easily transferred onto the face. We created four different visualizations: texture projection, inverse texture projection, grid line projection, and grid points projection. Face swapping between two different people and 3D mesh generation should be possible using the mappings, but we did not implement them.

3.8.1 *Texture projection*

Texture mapping, using the original texture that came with the model that was used to generate the training data, was an obvious choice for the first visualization. This projection could be used to assess the accuracy of the generated geometry and how similar the looks between the real human and the textured face were. An overview of the tex-

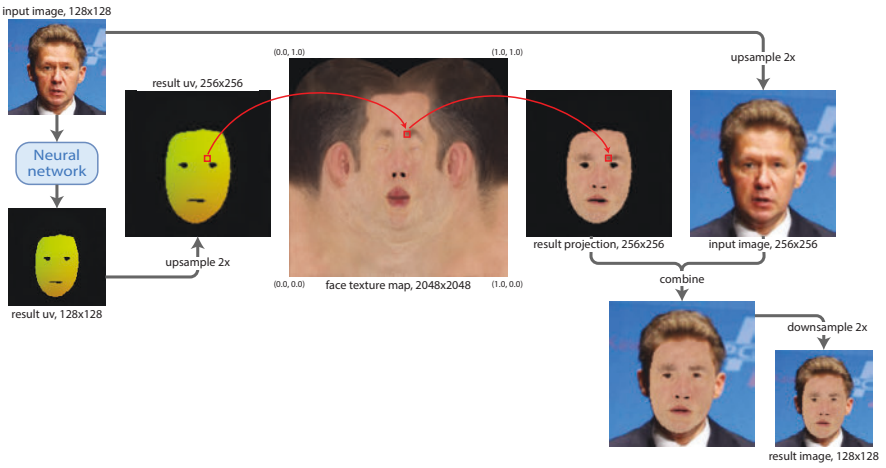


Figure 3.20: Texture projection on the input image. The mapping generated by the network was used to sample a texture, and the result was applied back to the input image.

ture projection process is given in figure 3.20. This method could not take lighting into account, and that is why the textured faces looked flat.

First, a UV map image was created from the input image using the neural network. The UV image was upscaled by a factor of two, and all of its non-black pixels were looped through. Using the UV coordinates encoded to the red and green channels of the pixel, a texture look-up was made. The pixel color value that was read from the texture was written to a new image into the same position as the original pixel in the UV image. The input image was then also upscaled, and the projected image was applied on top of it. Finally, the composite image was downscaled by a factor of two. The images were scaled up and down to introduce a simple antialiasing effect. Without the scaling, the results looked rough and shimmered when applied to a video.

3.8.2 *Inverse texture projection*

As the name suggests, the inverse texture projection did the same thing as texture projection, but in reverse. The result was a texture generated using the pixel color values in the input image. The method is illustrated in figure 3.21. This visualization could be used to assess how stable the mapping into the UV space was, especially with video input. Even if the subject was making expressions or moving their head around, the facial features mapped into the inverse texture should have always stayed still and not move around.

To begin, the UV mapping was generated from the input image using the neural network. All the non-black pixels in the UV image were looped over. The UV coordinate was read from the pixel and

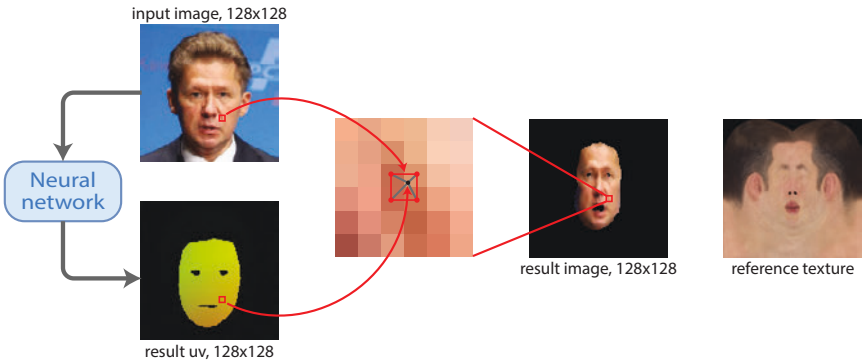


Figure 3.21: Inverse texture projection. The mapping generated by the network was used to transform the input image into the UV domain using bilinear filtering. Result image can be compared to the reference texture.

mapped to a location on the result image, basically just by multiplying the UV values, which were between 0.0 and 1.0, by the dimensions of the result image. This resulted in some real-valued coordinates on the result image which did not correspond to exact pixel locations. A color value was read from the input image from the same location as the UV pixel, and this color was applied to the result image pixels using the real-valued coordinate and bilinear filtering. The result image in figure 3.21 can be compared with the reference texture in the same figure. The reference texture shows how the UV mapping was done with the 3D model we used to generate the training data. The network had learned the UV mapping well if features in the result image were projected to same areas as in the reference texture.

If the subject in the input image was viewed from the side, this visualization should have only projected half of a face. This behavior can be confirmed by looking at row two in figure 4.2. The inverse texture projection could be developed further to generate a full texture of a human face by analyzing a short video where the head is rotated so that it is visible from all sides at least once. The full texture generation should be possible because no matter which way the face is pictured, a pixel from the same physical point of the face should always map to the same point in the UV space and therefore to the same pixel in the result image. When the head is rotated around, it is seen from all angles, and consequently, the texture generation from a video with some post-processing should be possible.

3.8.3 *Grid line projection*

The grid line projection was similar to the texture projection, with the exception that a grid line texture did not exist but was generated procedurally. This visualization helped to understand the contours of the generated geometry mapping and see how they followed the under-

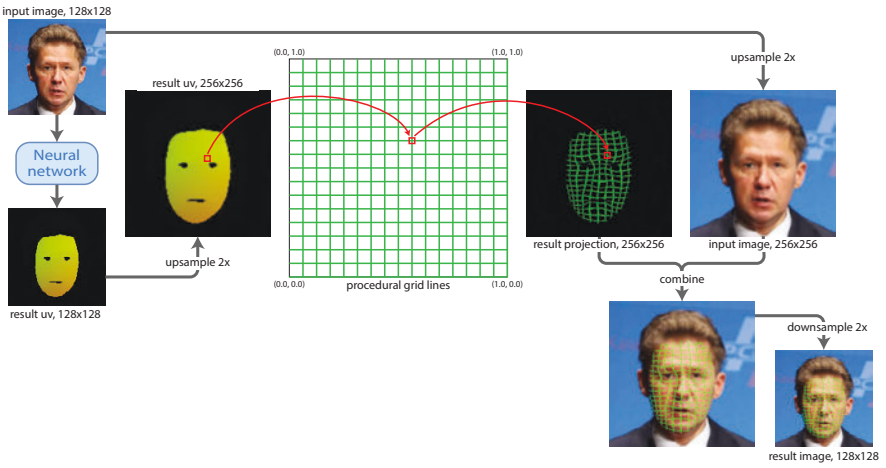


Figure 3.22: Grid line projection on the input image. The mapping generated by the network was used to sample a procedural line grid, and the result was applied back to the input image.

lying real geometry. Grid lines were also beneficial when assessing the temporal stability of the mapping when applied to a video.

To create this visualization, a UV mapping was first generated from the input image. The mapping was then scaled up, and the non-black pixels were looped through. The UV values of the pixels were used to sample a procedural grid in the UV space. The grid was created by first multiplying the UV coordinates, which have values between 0.0 and 1.0, by a grid density number larger than 2.0. From the resulting values, the fractional parts were taken. This resulted in values that go again from 0.0 to 1.0 but multiple times in the original range of 0.0 to 1.0. This formed the base of the grid structure. The new repeated UV coordinate could be used to determine if we were near the left or bottom edges of a square, resulting in L-shaped output. When these L-shapes were stacked together horizontally and vertically inside the grid structure, the procedural grid emerged. The density of the grid and the width of the grid lines could be adjusted without any limit. Next, as with texture mapping, the grid lines were sampled to a new image, combined with an upscaled input image, and then scaled down. The up and downscaling was done for simple antialiasing, as the aliasing would have been obvious with the grid line structure.

3.8.4 *Grid point projection*

The grid point projection procedurally generated small white dots on top of the input image using the UV mapping. This visualization was created to assess precisely the temporal stability of the results as even little jitter in the dots would have been visible. This projection could also be called the inverse mocap projection because the results resembled faces with the motion capture dots painted on. We imple-

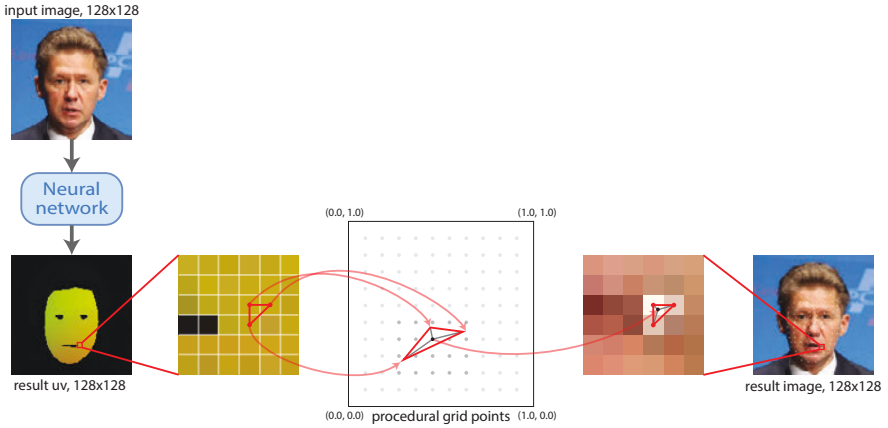


Figure 3.23: Grid point projection on the input image. This could be called the inverse mocap projection. Grid points were procedurally generated and then applied back to the input image using simple filtering.

mented the grid point projection somewhat differently compared to the texture or grid lines projections.

The UV mapping was created from the input image using the neural network, but this time the mapping was not upscaled. First, three adjacent UV pixels were selected from the UV image, forming a triangle. The triangle vertices, or pixels, all had 2D UV coordinates. These vertices were projected onto the UV space forming a new, differently shaped, triangle. The UV coordinate values of the vertices, which were between 0.0 and 1.0, were multiplied by a grid density number that was larger than 2.0. Now, it was imagined that the discrete integer coordinates in this scaled space represented the grid points. The size of a subgrid of these integer points, which completely covered the triangle, was then calculated. This subgrid of points was looped through, testing whether any of the points were inside the triangle using a method that returned barycentric coordinates for the point. If a point was found, the color was added to the input image pixels at the same pixel locations where the original UV triangle was formed. The barycentric coordinates for the point were used to determine the amount of color to be added to each of the input image pixels, giving a simple filtering effect. The UV triangles were generated so that the whole area of the UV image was covered.

3.9 RESULTS EVALUATION

As we iterated over numerous network models, loss functions, and data augmentation methods, we wanted to visually evaluate how the system was performing. We could have used *Tensorboard* [39] to visualize the results and outputs of the training runs, as CNTK had built-in support for it. At the time though, Tensorboard did not have all the

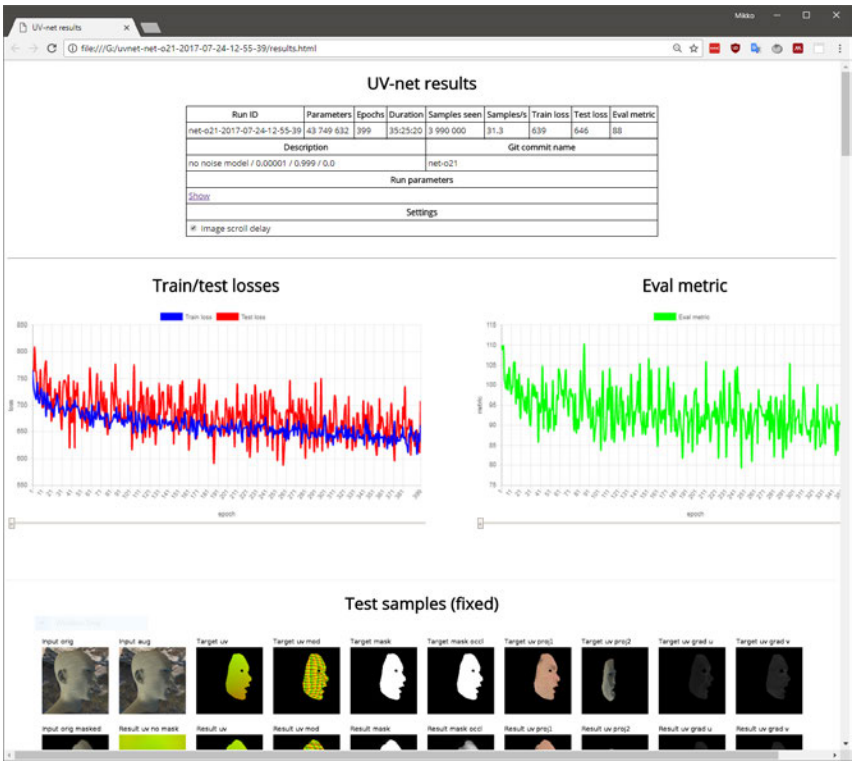


Figure 3.24: The browser-based result viewer. The run-specific parameters and values are on the top. The train loss, test loss, and evaluation metric visualizations are in the middle. The epoch specific evaluation images are after that, see figure 3.25 for a full example.

features needed to visualize the results the way we wanted. We decided to create our own simple HTML page that summarized the most important information about the runs and also allowed easy viewing of large amounts of evaluation images. An example of a result web page is shown in figure 3.24.

The result page listed the basic information about the particular training run: unique run-ID, total model parameter count, total epoch count, total time used, total samples seen, average samples per second training speed, average train and test loss from last ten epochs, average evaluation metric from last ten epochs, textual description of the run, and the unique Git commit tag name or hash associated with this run. Train and test loss history was plotted in the same graph, which helped to recognize overfitting problems. Evaluation metric history was plotted in a separate graph. Evaluation metric was a simpler absolute error between only the target and result UV images. This metric did not change even if loss function was modified. Loss function values could not be compared between runs if the loss function definition changed. In contrast, evaluation metric values could be compared between all the runs, no matter what. The whole parameter structure associated with the run was also printed on the page,

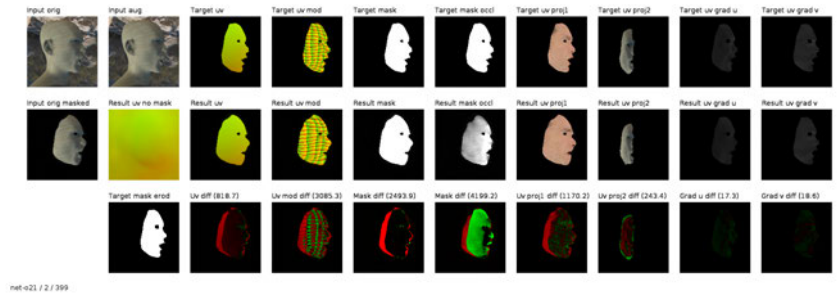


Figure 3.25: An evaluation image that was generated using an image from the test set. These were generated for every epoch and could be used to visually track the performance evolution of the network.

but hidden behind a link. This parameter structure contained every variable that was used to construct the network and could thus be used for verification purposes.

The network was usually evaluated every ten minutes or so, and large evaluation image plots were produced. One example of such an image can be seen in figure 3.25. With these images, we could see how the network performance evolved over time and also cancel training runs that were clearly going nowhere. Per each epoch, the network was evaluated with following as input: six specifically designed synthetic images, one random synthetic image from the current test set, eight specifically selected real-world images, and two randomly selected real-world images. In total, 17 different evaluation plot images were created per epoch. A training run of 72 hours would have around 800 epochs, so in total over 13 000 images could be created.

To help to view this large number of images, two things were done. First, the images were saved with a predictable file name pattern of *{epoch number}-{image number}*. Second, JavaScript, HTML5 canvases, and sliders were used to select and render all these images on a single HTML page. Using the sliders, it was easy to go back in time and see how the network performance evolved over time with each of the different evaluation samples. The HTML page and all the result images resided on the computing cluster work directory, and it was easy to mount that directory to a local computer using SSHFS. This way the downloading of all the evaluation images was prevented and results could be viewed quickly even over a slow connection.

If one moved the slider to see evaluation images of earlier epochs, every epoch image in between would be loaded. This was slow even on computers where the evaluation images were saved locally, not to mention loading them over slow networks. To alleviate this problem, a time delay option was added to the sliders. If one scrolled back in time, the evaluation images would only be loaded after the slider had been stopped for a brief moment. This allowed fast scrolling of evaluation images back in time even over very slow internet connections.

4 RESULTS

Using training data depicted in figure 3.18, we trained the neural network for 72 hours on a NVIDIA K40 GPU in the CSC Taito computing cluster. The original dataset of 100 000 different rendered images was heavily augmented and looped over numerous times. In total, the network saw over 8 million samples during the training. The final model had 43.7 million parameters taking up 175 megabytes of memory. Rotations were left out from the augmentations because they would have necessitated a substantial increase in the network parameter count.

4.1 REAL-WORLD IMAGES

We evaluated the network using selected images from the CelebA dataset [40]. Feeding one image through the model took on average 17 ms on a NVIDIA GTX 980 GPU and 750 ms on an Intel Core i5-3570K CPU. In figure 4.1, from the left, the first image is the input for the network. Next three are the outputs: *result mask*, *occluded result mask*, and *result UV*. The last four images are different visualizations created using the input image and network outputs.

Looking at the *result mask* column in figure 4.1, it is evident that the network was able to do the segmentation well. That is, the model detected what pixels belong to a human face and what pixels to the background. *Result mask occluded* column shows what the network thought is part of a human face but under some occlusion. The masks were not binary as they had some grayscale values. They were most likely caused by our occlusion augmentation method that sometimes generated transparent occlusion masks. Glasses, sunglasses, hats, hair, beards, and mustaches were detected, even though our training data did not have any of them. In some cases, e.g., with the image of Angela Merkel, the occluded result mask was more accurate than the actual result mask. The network was having some trouble especially when hair was occluding the forehead.

The *result UV* column in figure 4.1 shows the actual geometry mapping in the UV space generated by the network. Looking at it does not tell too much, other than that the results look about the same as in the training data. But when applied by doing a *texture projection on input*, it is revealed that the geometry mapping worked, and the network was able to track diverse human faces. The second and fifth rows have very different facial geometries, but the texture projections show that the model was able to differentiate between them. The *grid*

Figure 4.1: Results obtained when the network was applied to real-world images. Leftmost image is the input, and the next three are the outputs of the network. Last four images are all visualizations created using the input image and the network outputs.

Figure 4.2: Results obtained when the network was applied to real-world images with more extreme variations.

lines projection shows the geometric contours of the mapping generated by the network. Together with *grid points projection* these two are not that useful with static images but are very good at determining the stability and accuracy of the tracking when applied to a video. The last column shows the *inverse texture projection* where the facial area of the input image is projected into UV domain using the *result UV* mapping. If processed over time and multiple input images, this could be used to generate a texture from the person pictured.

More extreme situations are illustrated in figure 4.2. The training data did contain expressions with wide open mouths, but not as extreme expressions as in the first row. It shows that the network was able to track a little beyond the training data. Tracking of the face was not lost even at close to right angles, but other disturbances were not well tolerated in these situations. Rendered data contained faces with at most 45-degree angles along the z-axis (out of the paper), and the network was not able to map decent geometries much beyond that. The model worked well even if the subject was halfway out of the frame in any direction. Also, zooming in very close did not pose too much trouble for the network. If the person moved too far away from the camera, the tracking became quickly very unstable. This was mostly caused by the small 128x128 resolution of the input images. The face needed to occupy about one-fourth of the image for successful detection.

Inpainting geometry under the occlusions worked decently. The seventh and eight rows in figure 4.2 show typical results when input images had occlusions. The network detected the occlusions well and was able to generate dense geometry mapping underneath. The problem was that there was usually some warping of the geometry, mouth and eye holes disappeared randomly, and the geometry was not stable. These were most evident when tracking videos with moving occlusions over the faces.

Large variations in brightness and contrast in the input images did not pose problems for the model. Brightness or contrast could be tuned to such extremes that it was difficult for a human to recognize the image. Nevertheless, the network segmented and tracked the faces successfully. In spite of the fact that the training data only contained images with one head, the model was able to differentiate between at least four different faces in one image. There probably is no limit on the number, the only limiting factor, in this case, was the resolution of the input image.

4.2 SYNTHETIC IMAGES

Figure 4.3 shows what happened when we fed the network with similar images we used in training. These images were from the test set; the model had not seen these before. The leftmost image is the orig-

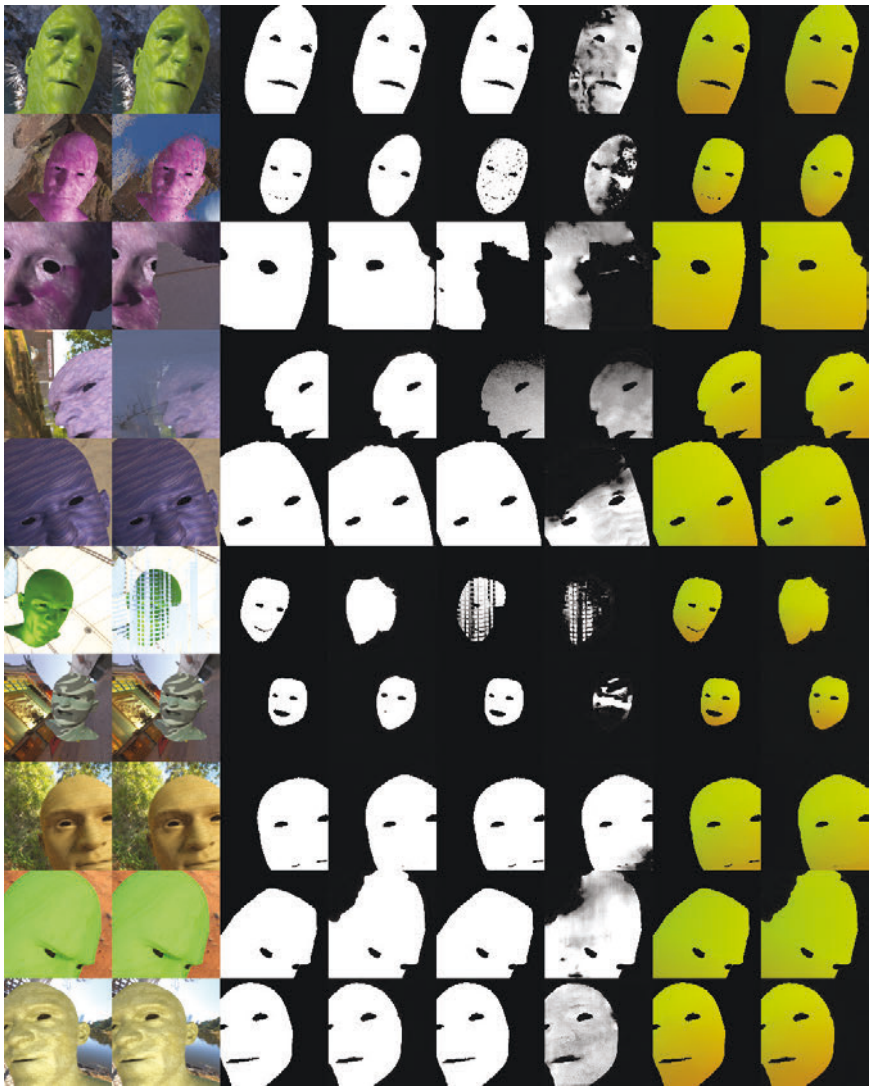


Figure 4.3: Results obtained when the network was applied to synthetic and augmented test images. The second, augmented image, was fed to the network. The result images can then be compared with the ground truth target images.

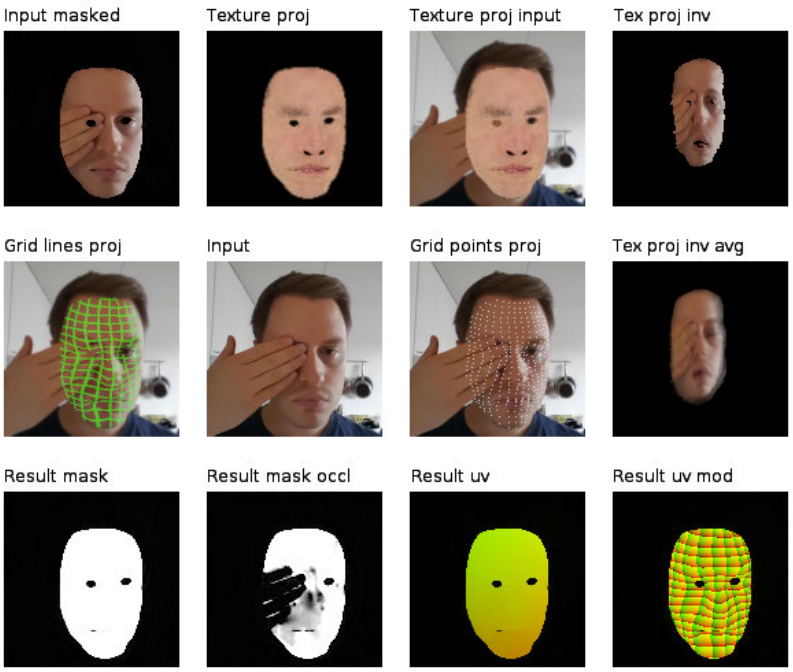


Figure 4.4: A frame capture from the test video. Full videos, which can be used to assess the temporal stability of the tracking, are available on YouTube [41].

inal render and the second is the original render augmented with every augmentation except rotations. Then there are pairs of target and result images. Target images are the known ground truths, and the result images can be compared with them. The model worked well as it was able to segment faces from the backgrounds and detect occlusions even in some extreme cases. Sometimes it was hard even for humans to recognize faces from the augmented images but the network was able to segment them and generate believable geometry.

4.3 VIDEO

To assess the temporal stability of the geometry tracking, i.e., it does not vibrate or warp too much over time, we created a test video containing various difficult scenarios. This video can be viewed on YouTube [41], and a sample frame from the video is shown in figure 4.4. Grid lines projection and grid points projection were most helpful when evaluating the temporal stability of the results. The video confirmed that the network could be applied to a video with satisfactory results. Temporal stability was good when the movement was slow, and the face was well visible without extreme expressions. As soon as the subject tilted too much away from the camera or their expres-

sion was exaggerated, the tracking started to vibrate or even warp to completely wrong geometries. Moving occlusions presented big challenges as the inpainted geometry in these cases was not stable. If there were no subject in the frame, the subject was too far away, the subject was turned away, or their face was occluded, the network tended to "explode." That means that the network output wildly wrong and rapidly changing results instead of just generating black.

4.4 DISCUSSION

The main research problem was whether it is possible to train a neural network using synthetic data to do dense human facial geometry tracking in the real world. We confirmed that it is feasible to train a fully convolutional neural network with skip connections using non-realistic rendered training data and the resulting model generalized to real-world images. Two important design decisions in the network topology were skip connections and upscaling with transpose convolutions. Without skip connections, the network was not able to learn anything useful. At first, we did the upscaling part of the model using matching max-unpooling layers, but they were not able to produce sharp enough results. We tried many combinations of L1 and L2 losses within the loss function and determined that using L1 with everything worked best. L2 loss tended to produce blurry mask edges and splotchy areas inside masks and UV mappings. Using gradient loss did improve learning performance in the beginning but the difference in results diminished when training was continued for extended time periods.

4.4.1 *Synthetic data generation*

Generating the training data using Blender was a success. We received some head models with expression data and textures from Remedy Entertainment, and these were easy to import into the program. We designed new scenes and materials in Blender and found out that the scripting capabilities were excellent. Every possible part of the program was scriptable using Python. This enabled us to randomize every rendering parameter programmatically in any way we wanted. Discoverability was a problem though, as it was sometimes tough to find the actual string of commands that would enable us to modify a specific parameter. Blender worked on Linux and supported rendering from the command line, so the large-scale training dataset generation on Aalto Triton or CSC Taito clusters was made easy.

4.4.2 *Model invariance*

One of the research goals was to make the model invariant to lighting, texturing, scaling and positioning. The idea was that the network could learn to be invariant to everything else than subtle changes in lighting caused by the geometry of the face. We first trained some models using only gray materials (see figure A.1) and some models using only noise materials (see figure A.2). These models worked well with test data similar to their training data and somewhat well with test data with real background and face textures. They performed significantly worse with real-world images. It was evident that the network could not be trained to be invariant with only gray or noise materials. Even with training data consisting of real background and face textures the results were not that good. It was only after we added our occlusion generation method that the model started to work well. It is a good question whether the network did latch on to the subtle changes in lighting or if the facial geometry detection relied on some other features.

4.4.3 *Occlusion augmentations*

Another interesting result was that our occlusion augmentation method improved results significantly. Without occlusion augmentations, the network was less stable even with images without occlusions. With the synthetic occlusion augmentations, the model was able to differentiate, for example, sunglasses and facial hair, even though the occlusions did not have anything resembling them. This meant that our rendered training data did not have to be so diverse. Occlusion augmentations also enabled the network to inpaint geometry. The model could infer the geometry under the occlusion from the surrounding features. Inpainting results were not perfect, especially when applied to moving occlusions in video material. The generated geometry warped around and the eye and mouth holes were sometimes missing from the inpainted geometry. It would be interesting to see if inpainting can be improved and the generated geometry made more believable and stable under moving occlusions.

4.4.4 *Comparison to other works*

During the implementation period of this project, several papers were published which had similar ideas when it comes to using convolutional neural networks for dense human facial geometry mapping. Sela et al. [28] used an image-to-image network trained with synthetic data to create dense depth and correspondence maps from real-world images. They had similar results with the mapping generation. Their synthetic data was not as diverse as ours and did not have occlusion

augmentations, so occlusions in input data did pose a problem. Also, their method did not inpaint occluded geometry. Güler et al. [29] did dense shape regression using a fully convolutional network with convincing results. They did not use completely synthetic data but a database of landmarked real-world images to generate the needed dense mappings for the training. Their method also extended to full body tracking and the method was almost real-time. It was not clear if their training had occlusion augmentations or how well the network could deal with facial obstructions and inpaint geometry. Richardson et al. [26] proposed a novel two-part architecture for detailed face reconstruction. Their model used in part synthetic rendered data, and the training did not include occlusion augmentations. Their method demonstrated some robustness against occlusions. This was, in part, because they did not do straight image-to-image mapping, but reconstructed a 3D mesh from the input image with the help of a morphable model. Jackson et al. [31] also did 3D mesh reconstruction, but with a network architecture that did the mapping from a 2D image straight to a 3D volume. Their model was resilient to occlusions, but they had the same kind of problems with temporal stability as our method. To conclude, it seems that completely non-realistic training data has not been used in the training of these facial geometry reconstruction networks. Also, it seems that no occlusion generation method like ours has been used in any of the other works.

4.4.5 *Reliability, validity and meaningfulness*

The reliability of our results was good, as the network training was successfully repeated hundreds of times with similar network structure and training data. Our primary method used to evaluate the results was calculating the difference between result images and previously rendered ground truth images. This gave a clear indication if the modifications made to the model were improving things. Also, we created various evaluation visualizations with real-world images during training runs. We used these to visually validate if the network was learning correctly. Over and underfitting is usually a problem when training neural networks, but we were able to eliminate both with our augmentation strategy.

When it comes to validity, the network implementation code might have included bugs that we did not spot, the model hyperparameter tuning could have missed some unknown optimal combinations, or there could have been a better loss function. These kinds of problems would have only worsened the visual output of method, not wholly invalidate it.

The results were meaningful as our method generalized well to real-world images. Dense mappings generated by networks like ours could be further used to create detailed 3D meshes from single im-

ages. Mappings generated from two different facial images could be used to perform face swapping. UV mapping of the face is an attractive base, on top of which many kinds of visualizations could be built. Our method was fast with evaluation time of 17 ms on average per frame. This enables exciting possibilities when applied to real-time data, like video streams.

4.4.6 *Problems encountered*

We encountered some practical level problems during the research. During training, we needed to read in the small training image files with random access. This always bogged down whatever traditional Hard Disk Drives (HDDs) we were using, and prevented running more than one training instance on one node. We could have solved this by using Solid State Drives (SSDs), but they were not available at the time either in Aalto Triton or CSC Taito GPU nodes. These nodes also had quite old NVIDIA K40 or K80 GPUs which were slow to train the networks. After we had finished most of the needed training runs, both Triton and Taito added new nodes with SSDs and NVIDIA Pascal-based GPUs. Trying to run the training on work desktops was not a huge success either as the desktop computer in question broke twice during long training runs. First a power supply died, and after fixing that the motherboard fried bricking two quite expensive GPUs. Also worth mentioning, the basic operating system libraries in Triton and Taito were somewhat old, and none of our rendering or training scripts ran without significant recompilation efforts. It might be a good idea to try to containerize both the rendering and training software so that they can be run more smoothly on different computing clusters.

4.5 FUTURE WORK

The following lists contain some proposals for future research and work, starting from higher level ideas and then going down to the lower level implementation details.

4.5.1 *High level*

- Temporal stability of the geometry, especially under occlusions, is not perfect. The stability could be tried to be improved by using, for example, recurrent neural networks and video as the training material.
- Reconstruct a 3D face model from the generated mapping. Ways to do this have been demonstrated for example in [28] and [26].

Another way to approach this is to use direct 2D-to-3D volumetric regression introduced in [31].

- Generate a depth map from the input image like in [28]. It should be tested if both UV mapping and depth map can be generated with the same network or if two separate ones are needed.
- Store head pose information when rendering the training set, and then use the pose information in the loss function as an extra term. Additionally, store the head facial landmarks and use these also in the loss function as an aid.
- When doing the data augmentation, randomly insert training samples without any faces. Then use a signal in the loss function to inform whether a face exists in the image or not.
- Obtain better face models with multiple textures and with versatile rigging for expressions. It might be possible find these for *Unreal Game Engine* [42] or *Unity Game Engine* [43]. It would probably improve results if the rendered training data had eyes and inside of the mouth modeled.
- Increase variation in rendering and augmentations. Currently, the network limits seem to reflect the limits of the training data, so increasing variation both in rendering and augmentation is probably a good idea. New augmentations could include blurring, adjusting contrast, and cropping with scaling. Render images that have more than one face. Also, try changing the training set size from 100 000 to smaller or bigger to see what effect it would have on the results.
- Implement real-time evaluation from a web camera video stream. The network evaluation is already quite fast at 17 ms, so making it work on live video should not be a problem. The more significant issue is the performance of the visualizations which are at the moment written in Python and too slow for real-time.
- Improve the inverse texture mapping. At the moment the texture is built over time using simple averaging (see [41]) and the results are not that good. Try to figure out a better way of doing it.
- Test if face swapping between images of two different people using the generated UV mapping produces satisfactory results. It could easily be done in real-time too.

4.5.2 *Lower level*

- Prune the neural network and try to reduce its size using the ideas presented in the Deep Compression paper [44].
- Current network is not optimally designed, and it does not work on arbitrary sized input as it should. This should be fixed, and then it should be tested if the network works on input resolutions different from its training image resolution. Also, evaluate whether it is feasible to process large facial images with the overlap-tile strategy described in [20].
- To better understand what the network is doing, create image collages using the output of different convolutional layers of the model.
- Larger training data resolutions should be tried. Images with a resolution as big as 512x512 have been used successfully [28]. With the newer training frameworks, it might also be possible to train the models with input images that have varying resolutions.
- Enable multi-GPU training within CNTK and test if the 1-bit SGD [45] or Block Momentum [46] algorithms work and speed up training as promised. Alternatively, implement the model with one of the newer deep learning frameworks (like *PyTorch* [22] or *Chainer* [47]) which promise easy multi-GPU implementation and dynamic computation graphs.
- Current training implementation generates intermediate evaluation images every epoch. This was done to understand how the network learned over time. It could now be replaced with one evaluation step at the end, which could consist of generating a few large collage images and rendering a short evaluation video. *Tensorboard* [39] would then also be sufficient for visualizing the training progress.
- Because the inside of the eyes and mouth was rendered as black in the training images, the network might have learned to expect that. Try rendering the insides with random colors.

4.5.3 *Low level*

- Input data was never normalized. Test if dataset normalization (image mean subtraction, standard deviation division) or Batch Normalization [48] would improve results.
- Occlusion mask generation is done at the same resolution as the target image. This causes aliased mask edges. Generate the

mask at a higher resolution and then scale down for an antialiasing effect. Also, modify the algorithm so that it generates binary masks instead of masks with grayscale gradients.

- When the training starts, the large datasets are decompressed into directories, and their file listings are generated. This is quite slow, and the speed could probably be improved by implementing a method to read data straight from the ZIP files in real-time while training.
- Test if the new dilated convolution [49] layers improve performance on either the downscaling or upscaling portions of the network. Also, when downscaling, try replacing the max-pooling layers with strided convolution layers [50].

5

SUMMARY

Recent advances in machine learning, neural network software frameworks, and GPU computing capacity have made it possible to design and teach very deep neural networks. Convolutional Neural Networks (CNNs) have been long used for image processing, and lately, it has been shown that they can be adapted to human face detection and extraction of relevant low-dimensional facial data. An extension of a CNN, the Fully Convolutional Neural Network (FCNN), has been recently shown to be capable of doing dense pixel-to-pixel mappings. This mapping has been used, for example, to do dense semantic segmentations of images. To train FCNNs in a supervised way, a large number of annotated input/output image pairs are needed. Nowadays, because of abundant processing and storage capacity in computing clusters, it is feasible to generate these kinds of input/output image pairs synthetically using rendering software. Also, data augmentation has been shown to expand the effective size of even small training datasets successfully.

Because the FCNN is an extension to the CNN, it is possible to train the former to do dense mapping of human faces. The downscaling part of the FCNN first detects the relevant features of the face and compresses them down to a lower dimensional presentation. The upscaling part of the FCNN then uses the representation to generate a segmentation and a dense mapping of the human face. One problem is to find a dense mapping that would represent the geometry of the face and that could be taught to a neural network. Another problem is to find training data of real-world images with the aforementioned mapping. Creating the mappings by hand would be very time-consuming as at least tens of thousands of training samples would be needed. On the contrary, generating a training dataset like this by rendering would be easy, as access to exact underlying geometry is available.

The main research goal of the thesis was to explore whether it is possible to train an FCNN using non-realistic synthetic data to do dense facial geometry tracking of real-world human faces. Subgoals included designing a method to generate the training data, making the network inpaint geometry under occlusions, making the network temporally stable when applied to video, and designing visualization methods for the generated mappings to ease the evaluation of the results.

To generate the synthetic training dataset, we used Blender to render 100 000 training samples of a randomized head mesh. One sample

included an input image, a UV image, and a mask image. The input image had realistic backgrounds but non-realistic face materials. We decided to use the underlying UV mapping of the head model to render the UV image, which then represented the geometry mapping in our method. The mask image was used to segment the frontal face area out of the input image.

We based the design of our FCNN on the U-net network topology. Seven levels were used and skip connections were added between all corresponding down and upsampling parts. Convolution feature map sizes, convolution filter sizes, and activation functions were slightly adjusted to optimize results. In the end, as a loss function, we used a simple sum of L1 losses between UV, UV gradient and mask images. UV image gradients were used to speed converge and enforce the smoothness of the resulting UV mapping.

Data augmentation was implemented by applying the following to the input image: rotations, color channels shuffles, exposure and gamma changes, Gaussian noise, and texture based occlusions. Augmentation generated one additional mask image, the occluded mask, which was used in the loss function to aid occlusion detection.

To visualize the generated geometry mappings, we implemented four different visualizations: texture projection, inverse texture projection, grid lines projection, and grid points projection. These projections were used to assess the accuracy and stability of the geometry mapping, both in static images and in videos. In addition, we created a custom browser-based results evaluation page that used these visualization images and other illustrations to help assess the performance of the network.

When given real-world human faces, our final FCNN was successful in doing the segmentation and geometry mapping, even though it had been trained on non-realistic synthetic data. The segmenting was, in most cases, good with sharp edges and geometry mapping worked well with non-extreme input images. Our data augmentation strategy successfully expanded the training dataset and prevented the network from overfitting. The occlusion augmentation method also enabled the network to inpaint geometry under obstructions that had not been in the training material, e.g., sunglasses or facial hair. When applied to a video, the temporal stability of the generated geometry was not bad but could be improved. Inpainted geometry was, most of the time, somewhat distorted and not temporally stable.

Future work could include 3D mesh generation from the UV mapping using, for example, morphable models and synthesis-by-analysis. Instead of doing a 2D-to-2D mapping and then 3D mesh reconstruction, it should be possible to train an FCNN to do straight 2D-to-3D volumetric mapping. The temporal stability of the generated geometry was not perfect, especially under occlusions, and could be improved.

REFERENCES

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org/.

[2] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." In: *Nature* 521.7553 (2015), pp. 436–444. DOI: 10.1038/nature14539.

[3] Toshiyuki Sakai, Nagao Makoto, and Kanade Takeo. "Computer analysis and classification of photographs of human faces." In: *First USA—Japan Computer Conference*. 1972, pp. 55–62.

[4] Margarita Osadchy, Yann Le Cun, and Matthew L Miller. "Synergistic Face Detection and Pose Estimation with Energy-Based Models." In: *Journal of Machine Learning Research* 8 (2007), pp. 1197–1215.

[5] Levi Valgaerts, Chenglei Wu, Hans-Peter Seidel, Andrés Bruhn, and Christian Theobalt. "Lightweight Binocular Facial Performance Capture under Uncontrolled Lighting." In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 31 (2012), 187:1–187:11. DOI: 10.1145/2366145.2366206.

[6] Asit Kumar Datta, Madhura Datta, and Pradipta Kumar Banerjee. *Face Detection and Recognition: Theory and Practice*. CRC Press, 2015. ISBN: 9781482226546.

[7] *Tensorflow*. URL: https://www.tensorflow.org/ (visited on 10/10/2017).

[8] *Microsoft Cognitive Toolkit (CNTK)*. URL: https://www.microsoft.com/en-us/cognitive-toolkit/ (visited on 10/10/2017).

[9] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. "Backpropagation Applied to Handwritten Zip Code Recognition." In: *Neural Computation* 1.4 (1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541.

[10] Elad Richardson, Matan Sela, and Ron Kimmel. "3D Face Reconstruction by Learning from Synthetic Data." In: *3Dv* (2016), pp. 461–468. DOI: 10.1109/3DV.2016.56.

[11] Hyeongwoo Kim, Michael Zollhöfer, Ayush Tewari, Justus Thies, Christian Richardt, and Christian Theobalt. "InverseFaceNet: Deep Single-Shot Inverse Face Rendering From A Single Image." In: *arXiv* (2017). arXiv: 1703.10956.

[12] Jonathan Long, Evan Shelhamer, and Trevor Darrell. "Fully convolutional networks for semantic segmentation." In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 2015, pp. 3431–3440. DOI: 10.1109/CVPR.2015.7298965.

[13] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. "Image-to-Image Translation with Conditional Adversarial Networks." In: *Conference on Computer Vision and Pattern Recognition (CVPR)* (2017). arXiv: 1611.07004.

[14] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. "Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World." In: *arXiv* (2017). arXiv: 1703.06907.

[15] Yair Movshovitz-Attias, Takeo Kanade, and Yaser Sheikh. "How useful is photo-realistic rendering for visual learning?" In: *Lecture Notes in Computer Science*. 2016, pp. 202–217. arXiv: 1603.08152.

[16] *Blender*. URL: https://www.blender.org/ (visited on 10/10/2017).

[17] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. "Deep Image: Scaling up Image Recognition." In: *arXiv* (2015). arXiv: 1501.02876.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks." In: *Advances in Neural Information Processing Systems 25* (2012), pp. 1097–1105.

[19] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2323. DOI: 10.1109/5.726791.

[20] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation." In: *Miccai* (2015), pp. 234–241. DOI: 10.1007/978-3-319-24574-4_28.

[21] A. Emin Orhan and Xaq Pitkow. "Skip Connections Eliminate Singularities." In: *arXiv* (2017). arXiv: 1701.09175.

[22] *PyTorch*. URL: http://pytorch.org/ (visited on 10/10/2017).

[23] Matthew A. Turk and Alex P. Pentland. "Face recognition using eigenfaces." In: *1991 IEEE Computer Society Conference on Computer Vision and Pattern Recognition* 3.1 (1991), pp. 586–591. DOI: 10.1109/CVPR.1991.139758.

[24] Paul Viola and Michael Jones. "Rapid object detection using a boosted cascade of simple features." In: *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Vol. 1. 2001, pp. 511–518. DOI: 10.1109/CVPR.2001.990517.

[25] Volker Blanz and Thomas Vetter. "A morphable model for the synthesis of 3D faces." In: *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. 1999, pp. 187–194. DOI: 10.1145/311535.311556.

[26] Elad Richardson, Matan Sela, Roy Or-El, and Ron Kimmel. "Learning Detailed Face Reconstruction from a Single Image." In: *Conference on Computer Vision and Pattern Recognition (CVPR)* (2017). DOI: 10.1109/CVPR.2017.589.

[27] Ayush Tewari, Michael Zollhöfer, Hyeongwoo Kim, Pablo Garrido, Florian Bernard, Patrick Pérez, and Christian Theobalt. "MoFA: Model-based Deep Convolutional Face Autoencoder for Unsupervised Monocular Reconstruction." In: *The IEEE International Conference on Computer Vision (ICCV)* (2017). arXiv: 1703.10580.

[28] Matan Sela, Elad Richardson, and Ron Kimmel. "Unrestricted Facial Geometry Reconstruction Using Image-to-Image Translation." In: *International Conference on Computer Vision (ICCV)* (2017). arXiv: 1703.10131.

[29] Rıza Alp Güler, George Trigeorgis, Epameinondas Antonakos, Patrick Snape, Stefanos Zafeiriou, and Iasonas Kokkinos. "DenseReg: Fully Convolutional Dense Shape Regression In-the-Wild." In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 2614–2623. DOI: 10.1109/CVPR.2017.280.

[30] Ronald Yu, Shunsuke Saito, Haoxiang Li, Duygu Ceylan, and Hao Li. "Learning Dense Facial Correspondences in Unconstrained Images." In: *International Conference on Computer Vision (ICCV)* (2017).

[31] Aaron S. Jackson, Adrian Bulat, Vasileios Argyriou, and Georgios Tzimiropoulos. "Large Pose 3D Face Reconstruction from a Single Image via Direct Volumetric CNN Regression." In: *International Conference on Computer Vision* (2017). arXiv: 1703.07834.

[32] *sIBL Archive*. URL: http://www.hdrlabs.com/sibl/archive.html (visited on 10/10/2017).

[33] *Textures.com*. URL: https://www.textures.com/ (visited on 10/10/2017).

[34] A. Van der Schaaf and J. H. Van Hateren. "Modelling the power spectra of natural images: Statistics and information." In: *Vision Research* 36.17 (1996), pp. 2759–2770.

[35] Ken Perlin. "Improving noise." In: *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. 2002, pp. 681–682. DOI: 10.1145/566570.566636.

[36] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." In: *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)* 9 (2010), pp. 249–256. DOI: 10.1.1.207.2059.

[37] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)." In: *arXiv* (2015). arXiv: 1511.07289.

[38] Diederik P. Kingma and Jimmy Lei Ba. "Adam: a Method for Stochastic Optimization." In: *International Conference on Learning Representations* (2015), pp. 1–15. arXiv: 1412.6980.

[39] *Tensorboard*. URL: https://github.com/tensorflow/tensorboard (visited on 10/10/2017).

[40] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. "Deep Learning Face Attributes in the Wild." In: *Proceedings of International Conference on Computer Vision (ICCV)*. 2015, pp. 3730–3738. DOI: 10.1109/ICCV.2015.425.

[41] *UV-net test videos*. URL: http://bit.ly/uvnet (visited on 10/10/2017).

[42] *Unreal Game Engine*. URL: https://www.unrealengine.com (visited on 10/10/2017).

[43] *Unity Game Engine*. URL: https://unity3d.com/ (visited on 10/10/2017).

[44] Song Han, Huizi Mao, and William J. Dally. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding." In: *International Conference on Learning Representations (ICLR)* (2016). arXiv: 1510.00149.

[45] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. "1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs." In: *Interspeech 2014*. 2014.

[46] Kai Chen and Qiang Huo. "Scalable Training of Deep Learning Machines by Incremental Block Training with Intra-block Parallel Optimization and Blockwise Model-Update Filtering." In: *2016 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 2016, pp. 5880–5884.

[47] *Chainer*. URL: https://chainer.org/ (visited on 10/10/2017).

[48] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." In: *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*. 2015, pp. 448–456. arXiv: 1502.03167.

[49] Fisher Yu and Vladlen Koltun. "Multi-Scale Context Aggregation by Dilated Convolutions." In: *International Conference on Learning Representations (ICLR)* (2016). arXiv: 1511.07122.

[50] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. "Striving for Simplicity: The All Convolutional Net." In: *arXiv* (2014). arXiv: 1412.6806.

APPENDICES

A.1 APPENDIX A NETWORK DEFINITION CODE

Listing A.1: Network definition Python code

```
def create_model(input_image: cntk.Variable, params: Parameters) -> cntk.Function:
  cnf = [int(round(params.model.initial_features * pow(params.model.feature_multiplier, i))) for i in range(0, 7)]
  ucnf = [int(round(params.model.up_factor * i)) for i in cnf]
  fs = params.model.filter_size

  with cntk.default_options(init=cntk.glorot_uniform(), activation=cntk.relu, pad=True, bias=True):
    p1, p2, p3, p4, p5, p6 = None, None, None, None, None, None

    l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=cnf[0])(input_image)
    l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=cnf[0])(l)
    p1 = l = cntk.layers.MaxPooling(filter_shape=(2, 2), strides=(2, 2))(l)

    if params.model.levels >= 2:
      l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=cnf[1])(l)
      l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=cnf[1])(l)
      p2 = l = cntk.layers.MaxPooling(filter_shape=(2, 2), strides=(2, 2))(l)

    if params.model.levels >= 3:
      l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=cnf[2])(l)
      l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=cnf[2])(l)
      p3 = l = cntk.layers.MaxPooling(filter_shape=(2, 2), strides=(2, 2))(l)

    if params.model.levels >= 4:
      l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=cnf[3])(l)
      l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=cnf[3])(l)
      p4 = l = cntk.layers.MaxPooling(filter_shape=(2, 2), strides=(2, 2))(l)

    if params.model.levels >= 5:
      l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=cnf[4])(l)
      l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=cnf[4])(l)
      p5 = l = cntk.layers.MaxPooling(filter_shape=(2, 2), strides=(2, 2))(l)

    if params.model.levels >= 6:
      l = cntk.layers.Convolution(filter_shape=(3, 3), strides=(1, 1), num_filters=cnf[5])(l)
      l = cntk.layers.Convolution(filter_shape=(3, 3), strides=(1, 1), num_filters=cnf[5])(l)
      p6 = l = cntk.layers.MaxPooling(filter_shape=(2, 2), strides=(2, 2))(l)

    if params.model.levels >= 7:
      l = cntk.layers.Convolution(filter_shape=(1, 1), strides=(1, 1), num_filters=cnf[6])(l)
      l = cntk.layers.Convolution(filter_shape=(1, 1), strides=(1, 1), num_filters=cnf[6])(l)
      l = cntk.layers.MaxPooling(filter_shape=(2, 2), strides=(2, 2))(l)

    if params.model.levels >= 7:
      l = cntk.layers.ConvolutionTranspose(filter_shape=(2, 2), strides=(2, 2),
          num_filters=ucnf[6], output_shape=(2, 2))(l)
      l = cntk.layers.Convolution(filter_shape=(1, 1), strides=(1, 1), num_filters=ucnf[6])(l)
      l = cntk.layers.Convolution(filter_shape=(1, 1), strides=(1, 1), num_filters=ucnf[6])(l)
      l = cntk.ops.splice(l, p6, axis=0)

    if params.model.levels >= 6:
      l = cntk.layers.ConvolutionTranspose(filter_shape=(3, 3), strides=(2, 2),
          num_filters=ucnf[5], output_shape=(4, 4))(l)
```

```
    l = cntk.layers.Convolution(filter_shape=(3, 3), strides=(1, 1), num_filters=ucnf[5])(l)
    l = cntk.layers.Convolution(filter_shape=(3, 3), strides=(1, 1), num_filters=ucnf[5])(l)
    l = cntk.ops.splice(l, p5, axis=0)

  if params.model.levels >= 5:
    l = cntk.layers.ConvolutionTranspose(filter_shape=fs, strides=(2, 2),
        num_filters=ucnf[4], output_shape=(8, 8))(l)
    l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=ucnf[4])(l)
    l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=ucnf[4])(l)
    l = cntk.ops.splice(l, p4, axis=0)

  if params.model.levels >= 4:
    l = cntk.layers.ConvolutionTranspose(filter_shape=fs, strides=(2, 2),
        num_filters=ucnf[3], output_shape=(16, 16))(l)
    l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=ucnf[3])(l)
    l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=ucnf[3])(l)
    l = cntk.ops.splice(l, p3, axis=0)

  if params.model.levels >= 3:
    l = cntk.layers.ConvolutionTranspose(filter_shape=fs, strides=(2, 2),
        num_filters=ucnf[2], output_shape=(32, 32))(l)
    l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=ucnf[2])(l)
    l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=ucnf[2])(l)
    l = cntk.ops.splice(l, p2, axis=0)

  if params.model.levels >= 2:
    l = cntk.layers.ConvolutionTranspose(filter_shape=fs, strides=(2, 2),
        num_filters=ucnf[1], output_shape=(64, 64))(l)
    l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=ucnf[1])(l)
    l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=ucnf[1])(l)
    l = cntk.ops.splice(l, p1, axis=0)

  l = cntk.layers.ConvolutionTranspose(filter_shape=fs, strides=(2, 2),
      num_filters=ucnf[0], output_shape=(128, 128))(l)
  l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=ucnf[0])(l)
  l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=ucnf[0])(l)
  l = cntk.ops.splice(l, input_image, axis=0)

  l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=ucnf[0])(l)
  l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=ucnf[0])(l)
  l = cntk.layers.Convolution(filter_shape=fs, strides=(1, 1), num_filters=4, activation=None)(l)

 return l
```

A.2 APPENDIX B LOSS FUNCTION CODE

Listing A.2: Loss function Python code

```
def cntk_l1_loss(a, b) -> cntk.Function:
  return cntk.ops.reduce_sum(cntk.ops.abs(a - b))

def setup_cntk(params: Parameters) -> Tuple[cntk.Function, cntk.Function, cntk.Function]:
  input_image = cntk.input_variable((3, params.input_size[0], params.input_size[1]))
  target_uv = cntk.input_variable((2, params.input_size[0], params.input_size[1]))
  target_mask = cntk.input_variable((1, params.input_size[0], params.input_size[1]))
  target_mask_occluded = cntk.input_variable((1, params.input_size[0], params.input_size[1]))
  target_mask_eroded = cntk.input_variable((1, params.input_size[0], params.input_size[1]))

  model = create_model(input_image, params)

  result_uv = model[0:2, :, :]
  result_mask = model[2, :, :]
  result_mask_occluded = model[3, :, :]

  target_ux_grad, target_uy_grad, target_vx_grad, target_vy_grad = get_image_gradients_cntk(target_uv)
  result_ux_grad, result_uy_grad, result_vx_grad, result_vy_grad = get_image_gradients_cntk(result_uv)

  target_ux_grad = target_ux_grad * target_mask_eroded
  target_uy_grad = target_uy_grad * target_mask_eroded
  target_vx_grad = target_vx_grad * target_mask_eroded
  target_vy_grad = target_vy_grad * target_mask_eroded

  result_ux_grad = result_ux_grad * target_mask_eroded
  result_uy_grad = result_uy_grad * target_mask_eroded
  result_vx_grad = result_vx_grad * target_mask_eroded
  result_vy_grad = result_vy_grad * target_mask_eroded

  result_uv_masked = result_uv * target_mask

  loss_function = \
      cntk_l1_loss(result_uv_masked, target_uv) + \
      cntk_l1_loss(result_mask, target_mask) + \
      cntk_l1_loss(result_mask_occluded, target_mask_occluded) + \
      cntk_l1_loss(result_ux_grad, target_ux_grad) + \
      cntk_l1_loss(result_uy_grad, target_uy_grad) + \
      cntk_l1_loss(result_vx_grad, target_vx_grad) + \
      cntk_l1_loss(result_vy_grad, target_vy_grad)

  eval_function = cntk_l1_loss(result_uv_masked, target_uv)

  return model, loss_function, eval_function
```

A.3 APPENDIX C COLLAGE IMAGES

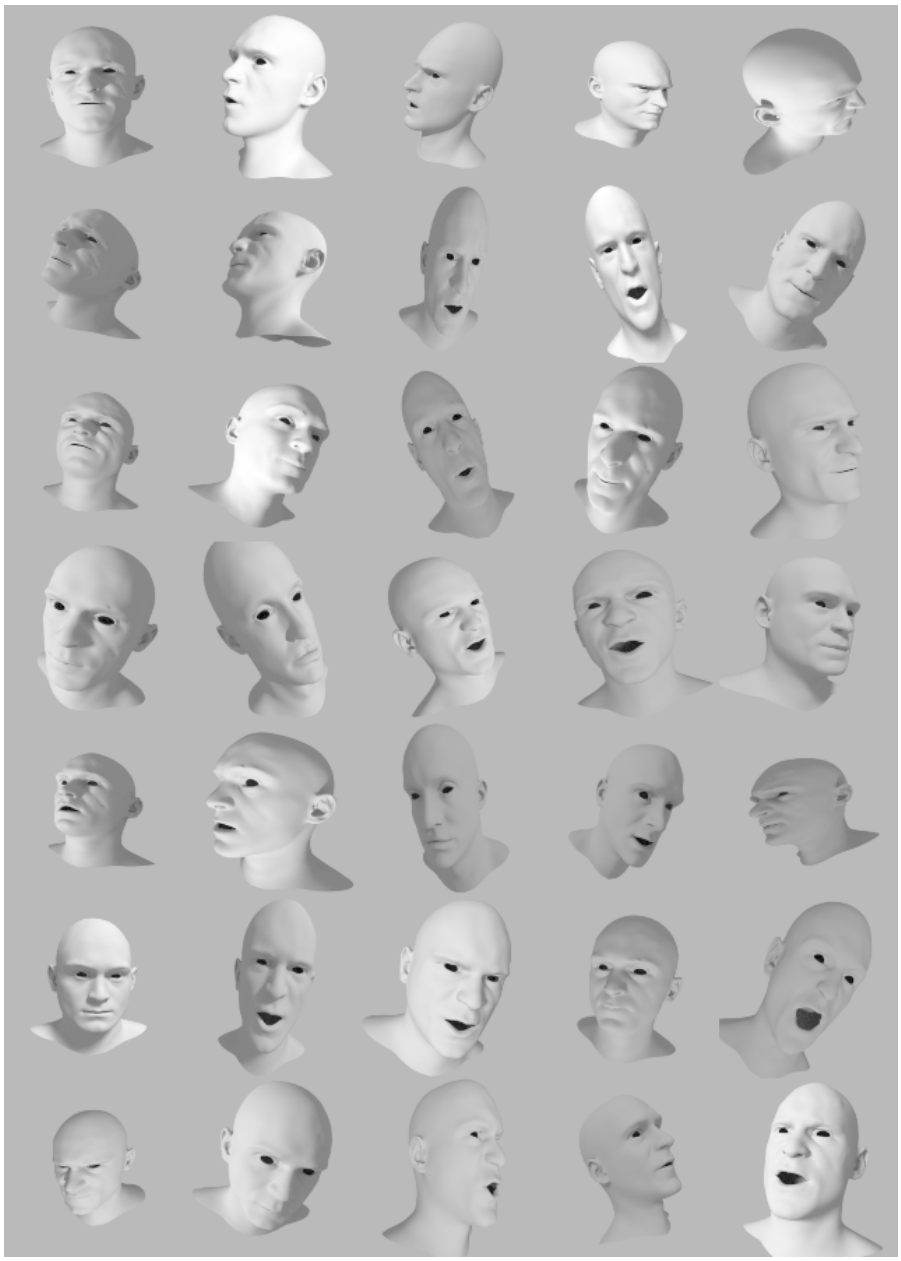


Figure A.1: A collage of head renders with gray textures. These were the most simplistic version of the head renders we used to train the network.

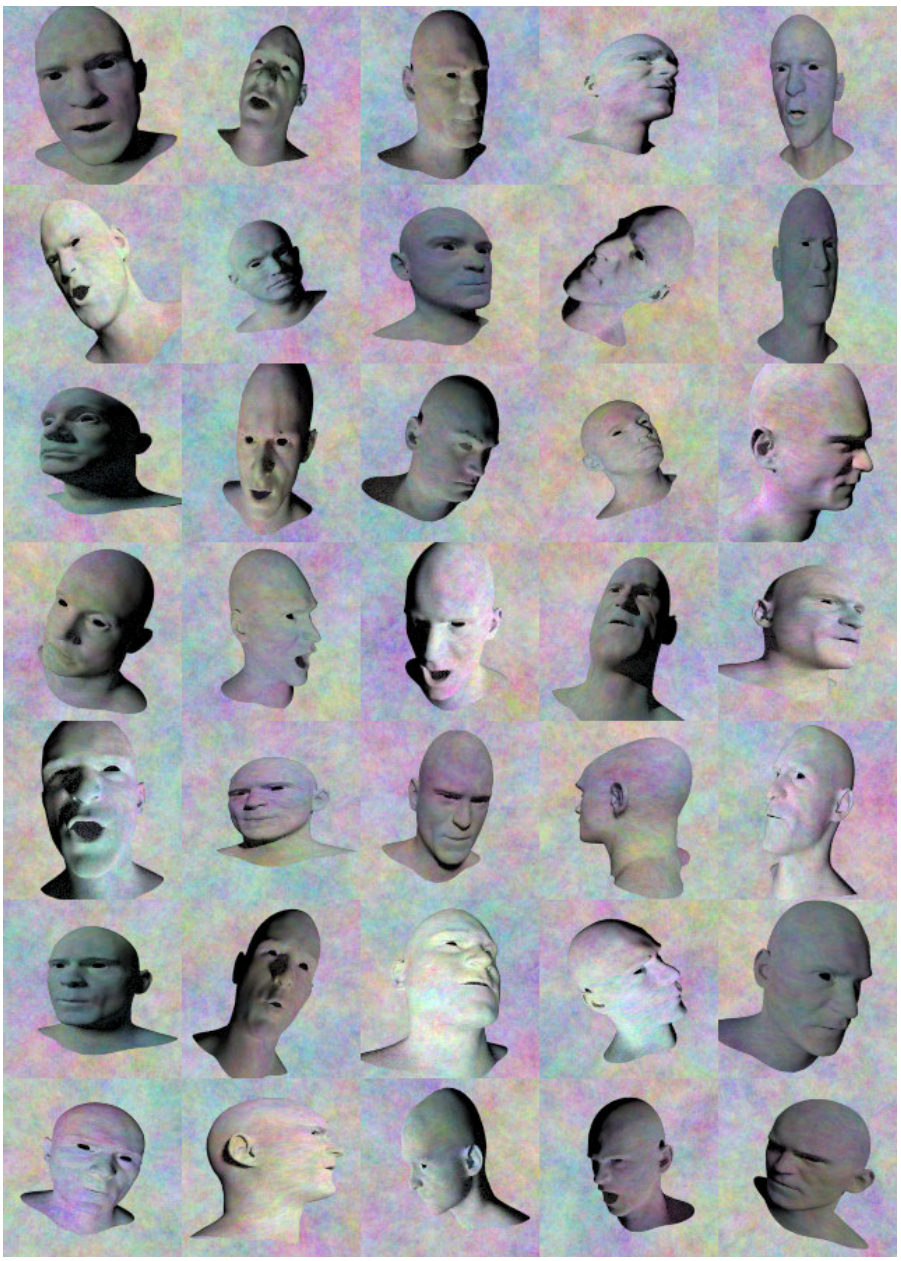


Figure A.2: A collage of head renders with 1/f noise textures. The 1/f noise should follow the power spectra of natural images. These or the gray head renders in figure A.1 did not make the network generalize well to real-world images.

Figure A.3: A collage of head renders with realistic background and non-realistic face textures. Surprisingly, a network trained with images like these performed similarly compared to a network trained with realistic face textures.

Figure A.4: A collage of shuffle augmentations only. This augmentation increased the effective size of the training dataset by a factor of six. The same, non-augmented, image collage can be seen in figure 3.9.

Figure A.5: A collage of exposure and gamma augmentations only. The variation in image brightness is more evident when compared with the same, non-augmented, image collage in figure 3.9.

Figure A.6: A collage of noise augmentations only. Noise augmentation like this helped the training process and made the network generalize better to real-world images. The noise is more evident when compared with the same, non-augmented, image collage in figure 3.9.

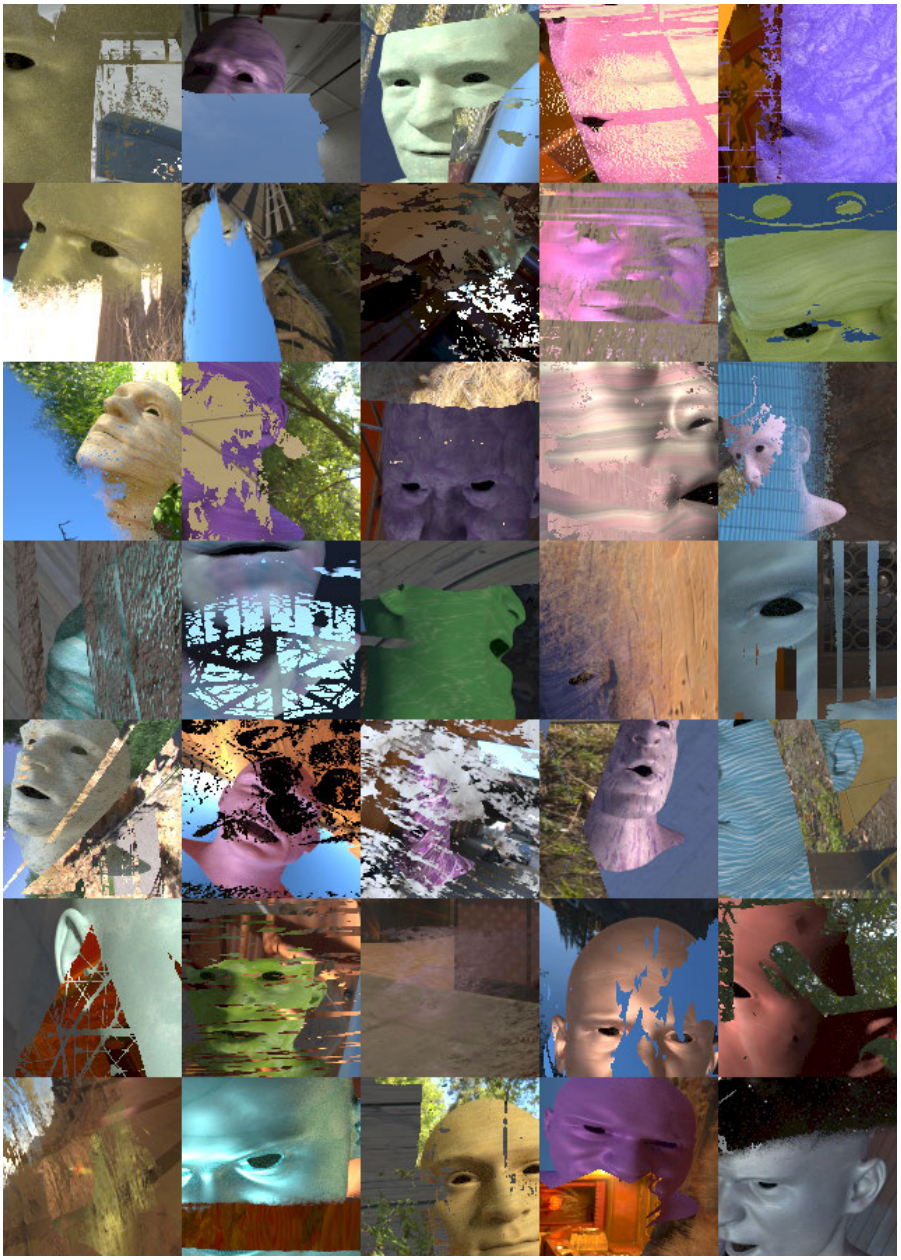


Figure A.7: A collage of occlusion augmentations only. By using occlusion masks generated from realistic textures, we were able to make the network detect arbitrary occlusions and inpaint geometry underneath them. This image collage can be compared with the same, occlusion-free, image collage in figure 3.9.

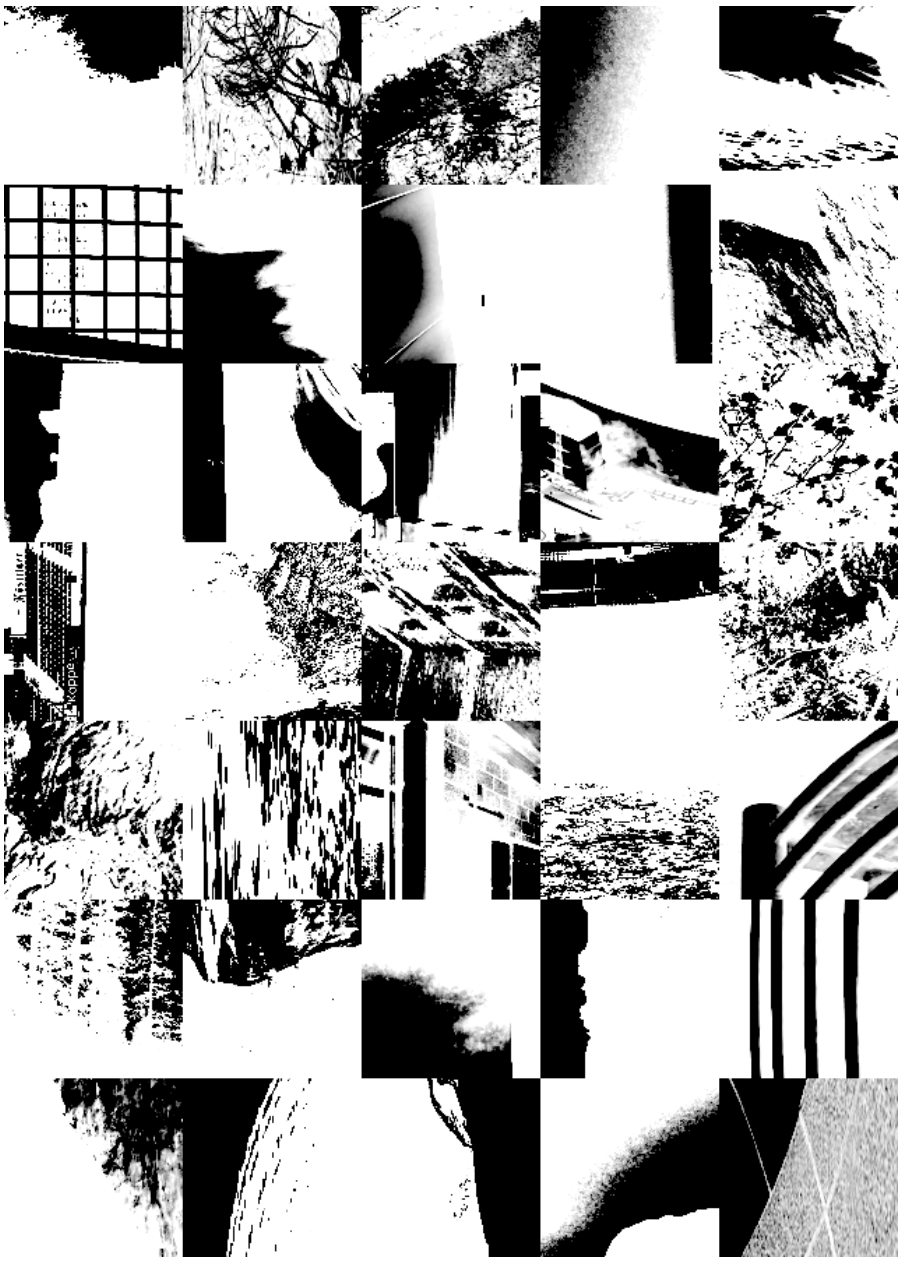


Figure A.8: A collage of occlusion masks. These masks were used in the occlusion creation process depicted in figure number 3.16. The mask generation method was adjusted so that the ratio of white-to-black pixels was visually about 50/50 in larger scale.