המעבדה לעיבוד גיאומטרי של תמונות
Geometric Image Processing Laboratory

TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY

PROJECT IN IMAGE PROCESSING

234329

Real-Time 3D Face Reconstruction

Author:
Shadi Endrawis

Advisor:
Matan Sela

February 18, 2018

Contents

1 Abstract **3**

2 System Description **4**
2.1 Synthetic Data 4
2.2 Technology and Libraries 5
2.3 Hardware 5
2.4 3D Reconstruction Network 5
2.4.1 Deep Neural Network 5
2.4.2 The Iterative Model 6
2.4.3 The Rendering Unit 6
2.5 Video and Real-Time Reconstruction System 7

3 Execution **8**
3.1 Training the Single Image Network 8
3.1.1 The Inputs 8
3.1.2 The Architecture 8
3.1.3 The Hyper-Parameters 9
3.2 Implementation of The Video/Real-Time System 10

4 Results **11**
4.1 Synthetic Evaluation 11
4.2 Single Image Reconstruction 12
4.3 Video Reconstruction 14
4.4 Real-Time Reconstruction 14

5 Conclusions **17**

6 Further Work and Suggestions **18**
6.1 Detail Extraction 18
6.2 Modifying The Model 18
6.3 Modifying the Data 19
6.4 Mobile 19

References **20**

List of Figures

1 The ResNet Model 6
2 The Iterative Model 7
3 Synthetic Examples 11
4 Example of Iterations 12
5 Grayscale Reconstruction Examples 13
6 Single Image Reconstruction Examples 13
7 Video Reconstruction Examples 15
8 Real-Time Reconstruction Examples 16

1 Abstract

Fast and robust three-dimensional reconstruction of facial geometric structure from a single image is a challenging task with numerous applications. In this project, I implemented and experimented with the iterative CNN models proposed in [3] and [4] for extracting geometric structure from a single image.

The fist step was to train a deep neural network that extracts the reconstruction from a single image using synthetic data of faces generated by the model proposed in [1].

The next step in the project was to to take the models trained to perform the reconstruction from a single image and apply them first to videos, and from there to optimize the system for real-time 3D face reconstruction using a webcam.

2 System Description

In this section I will describe all the different software and hardware components of the system I implemented.

2.1 Synthetic Data

It has been well established that usually you need a large amount of labeled data in order to train deep neural networks from scratch. Unfortunately, there's no large dataset that contains examples of images and their 3D reconstruction, which makes it difficult to approach this problem in a typical supervised learning manner.

To solve this problem we will use a model-based approach by modeling the face using the the PCA model proposed in [1]. This model provides us with a method to embed a mesh representation of a face (consisting of $N \sim 30K$ vertices and $M \sim 50K$ triangles) into a 200-dimensional space.

Additionally, similar to [3] and [4] I will use another embedding space of 84 dimensions in order represent the facial expressions.

Overall the model consists of three components. μ_S- The shape of the average face, $A_{id}-$ The identity embedding matrix (size $3N \times 200$) and $A_{exp}-$ The expression embedding matrix (size $3N \times 84$). In order to reconstruct the shape some identity and expression coefficient vectors $\alpha_{id}, \alpha_{exp}$ we perform the following operation:

$$S = \mu_S + A_{id}\alpha_{id} + A_{exp}\alpha_{exp}$$

Once we have a model for the face, the problem becomes simpler. We can produce as much data as we like by generating random coefficients $\alpha_{id}, \alpha_{exp}$ and producing the geometry (the labels) using the model, and images (the inputs) by rendering those geometries. Now the task becomes to train the network to learn those coefficient vectors.

Similar to [4] I also use 6 additional input channels to the network which are RGB PNCC and normal renderings of the geometry with some random perturbation to simulate iterative improvement of the network output (more on that later).

2.2 Technology and Libraries

- python3 – I choose to write most of the components of my system in python because it allowed for fast prototyping and had many robust libraries and resources/implementations to use. most of the data and mathematical processing was done using numpy.
- tensorflow – I chose this framework to implement the deep neural network part of the system because it allowed easy cross-platform use of GPU resources for the training, and large amount of resources and architecture implementations available online.
- opencv2 – This framework was mainly used to perform some image processing tasks and the video capturing/writing part of the system.
- dlib – This framework was used for the face detection. The implementation of the face detection method in this library is much faster than the one in opencv2 and it was necessary to implement the real-time version of the system.

2.3 Hardware

- Nvidia GPU – A cuda-capable GPU was necessary in order to train the network in a reasonable amount of time.
- Webcam – This is needed for the real-time version of the system, for the most part I used the built-in cam in personal laptop throughout this project.

2.4 3D Reconstruction Network

2.4.1 Deep Neural Network

I used an architecture very close the one proposed for coarse reconstruction in [4], the full architecture can be seen in $Figure$ 1.

The inputs are three RBG 200×200 images, the first is a crop of the original image, the second is a render of the geometry using some canonical coloring called PNCC and the third is another render of the geometry using the normal to vertex as it's RGB color. An example input can be seen in $Figure$ 3.

The output of the network is a vector of 290 numbers, the first 200 represent the coefficients for the identity embedding, the next 86 are the coefficients for the expres-

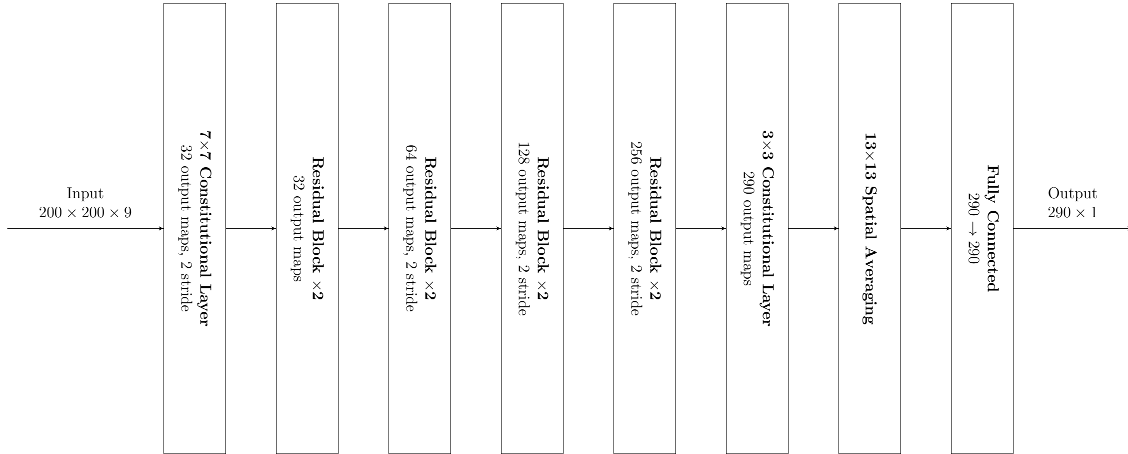

Figure 1: The ResNet model used for the geometry reconstruction. The residual blocks in the networks are the blocks defined in the ResNet architecture proposed in [2].

sion embedding and the remaining 6 are transformation parameters, 3 parameters for rotations 1 for scaling and 2 for translation.

2.4.2 The Iterative Model

In [3] they found out empirically that an iterative system works better for reconstruction. Furthermore, the 6 extra input channels (the PNCC and normal render) require that we actually know the geometry beforehand.

The iterative process starts with the average face geometry to produce the initial PNCC and normal images, then at each iteration we produce a new embedding vector, which we use to produce another set of inputs for the next iteration.

The usually process takes about 3-5 iterations to converge. A diagram of how the iterative process works can be found in *Figure* 2.

Clearly, the network had to be trained to do this specific task of improving the reconstruction from previous iteration, more on that in later chapters.

2.4.3 The Rendering Unit

The rendering unit seen in *Figure* 2 is a made of a lightweight renderer provided by *Matan Sela* written in C++. The renderer was then further modified, optimized and imported to python to work with the rest of the components of the system.

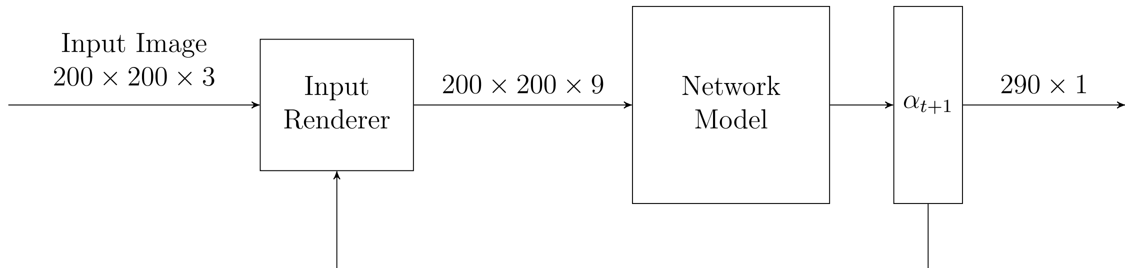

Figure 2: The iterative reconstruction system.

2.5 Video and Real-Time Reconstruction System

The first and the most trivial step I took to to move from a single image to video is to apply the single image system on each frame of the video, this is obviously very wasteful and could never be fast enough to transition into real-time.

Under the assumption that the face doesn't change dramatically from frame to frame in a video, we can use the geometry from the previous frame to construct the input channels instead of having to start with the average face and perform several iterations to get to the desired reconstruction, which allows us to perform the input construction and run the network only once per frame.

This optimization alongside several other optimization to how the rendering is handled allowed the system work in real-time using my personal laptop at 13-14 fps.

There is definitely more room for improvement in the performance in order to achieve higher frame rate, but this seemed reasonable enough for this point in time.

3 Execution

In this section I will explain in detail the steps taken in order to achieve the final results, and some insight on a few failed attempts.

3.1 Training the Single Image Network

3.1.1 The Inputs

In order to train the network to correct itself from previous iterations, the data needs to simulate the fact that the PNCC and normal renders are from geometries that aren't similar to the ground truth be close to it.

The way the synthetic examples for the network are generated:

- Randomly generate the ground truth coefficients α_{gt} and transform parameters, this is the output to the network.
- Randomly generate a texture and background image to render the first input image using the geometry that corresponds to α_{gt}.
- Randomly generate perturbation coefficients α_q.
- Linearly interpolate the ground truth and the perturbation $\alpha = p\alpha_{gt} + (1-p)\alpha_q$ where q is uniformly sampled from $[0, 1]$.
- Render the PNCC and normal images using the geometry that corresponds to α and crop the textured image accordingly.

Examples of those images can be seen in $Figure$ 3.

3.1.2 The Architecture

Since this was the first time for me working with tensorflow, the first step was to learn to setup an environment which allows me to experiment with different models and hyper-parameters.

Once that was done, I ran many experiments with different models to test for accuracy and performance, models such as: different inception models, ResNeXt, MobileNet, FaceNet and others.

I found that in terms of accuracy the model did not have as much of an effect as I had expected, and since the evaluation was mostly visual, it was sometimes difficult

to tell the difference between the different models, sometimes the more advanced and complicated models performed even worse.

Speed wise however, there was a huge difference when it came to models like inceptions and ResNeXt which reduce the number of parameters but there are many more convolutions, and ultimately they were not worth it for the final system since the final goal was to run the system in real-time and speed was paramount.

Ultimately, I went an architecture that is very close to the coarse net in [4], and used the standard ResNet architecture defined in [2].

3.1.3 The Hyper-Parameters

All of the hyper-parameters of the system were tuned in a cross-validation process.

- Loss Function – As mentioned in [4], the loss function is a function between geometries rather than the coefficient vectors, and that is due to the fact that the identity and expression basis are not orthogonal. The loss function for the geometry proposed in the paper was:

$$\left\| [A_{id}|A_{exp}](x - \hat{x}) \right\|_2^2$$

  And and a simple MSE for the transformation parameters. Empirically I found that $\ell_1 - norm$ works better than $\ell_2 - norm$ for the geometry.

- Regularization – I tried two types of regularization:
  - Weight Decay: while it did make the training process slightly smoother and allowed me to train for more epochs, there were no clearly noticeable improvements.
  - Dropout: I tried to apply dropout in many different ways to either dense or convolution layer with different drop rates. However, I was disappointed that even tough it prevented the network from over-fitting in terms of the loss value, the visual results were significantly worse in most cases.

  Contrary to what usually happens, regularization didn't help for this particular problem and therefore the final model didn't include either method.

- Optimizer – Initially I used ADAM optimizer with exponentially decaying learning rate. I later found that this was affecting the learning process negatively and started using a piece-wise constant learning rate, where I lower the learning rate when the loss value plateaus.

- Early Stopping – As previously mentioned before, regularization techniques didn't work out for me, therefore, I couldn't train the network a large number of epochs. I found that the network would start to severely over-fit after roughly epoch 10, so I have to apply early stopping to achieve good results.

3.2 Implementation of The Video/Real-Time System

The transition from a single image to video is relatively easy, I simply took the single image system and applied a single iteration on each frame, where the geometry for the PNCC and normal is the one calculated in the previous frame (the first frame starts with the average face).

The real-time setting however, is not as simple. Systematically it works similarly to the video reconstruction, but now the production of the rendered images and the forward pass of the network has to be fast enough to run for each frame in a reasonable rate.

Apparently, the network can be applied with a very high frame rate, the main bottleneck of this system is the rendering unit. I made several optimizations in C++ side of the code to avoid unnecessary operations and make the renderer more task specific. However, most of the time was being wasted in the transition from python to C++, and for that problem I used a library called *ctypes* which allowed me to run the C++ renderer from python, in addition to the face that *numpy* had several mechanisms to pass *numpy* arrays to C++ through *ctypes* in a very efficient manner.

For the face detection part, I initially used the detector in *opencv*2 to implement the video system, but it turn out to be extremely slow when I wanted to transition to real-time. Therefore, I instead used the detector from the *dlib* library which is significantly faster and can be run in a real-time setting.

Since the face detection is independent in each frame the box that the detector returns changes size, which causes an undesirable jitter effect. In order solve this applied some smoothing to the frames by averaging between the box from the previous frame and the current one. I also applied this smoothing to the geometries.

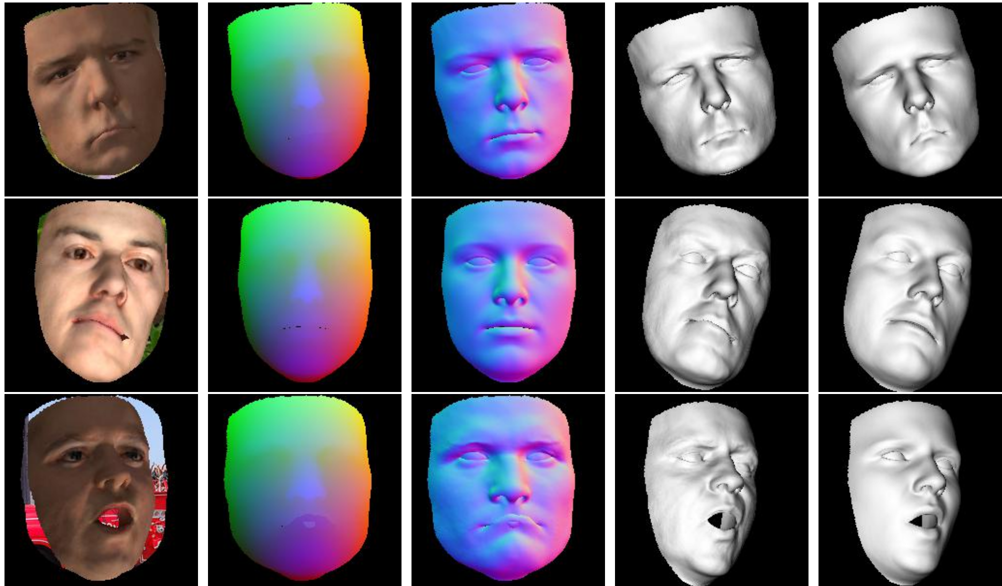

Figure 3: Examples of input/output of the reconstruction network, each row is a different example. The first three columns are inputs to the network, the fourth column is the expected geometry and the fifth one is what the network predicts

4 Results

In this section I will present the final results of the different parts of the project. Most of the evaluation was done in a visual manner because there were no other reliable evaluation metrics to rely upon.

4.1 Synthetic Evaluation

Clearly, the first step in the evaluation process would be to test the network on the synthetic data similar (but not the same) to the one it was trained on.

A few examples of those results can be seen in *Figure* 3, as we can see the network achieves a very close (but not exact) reconstruction on the synthetic data.

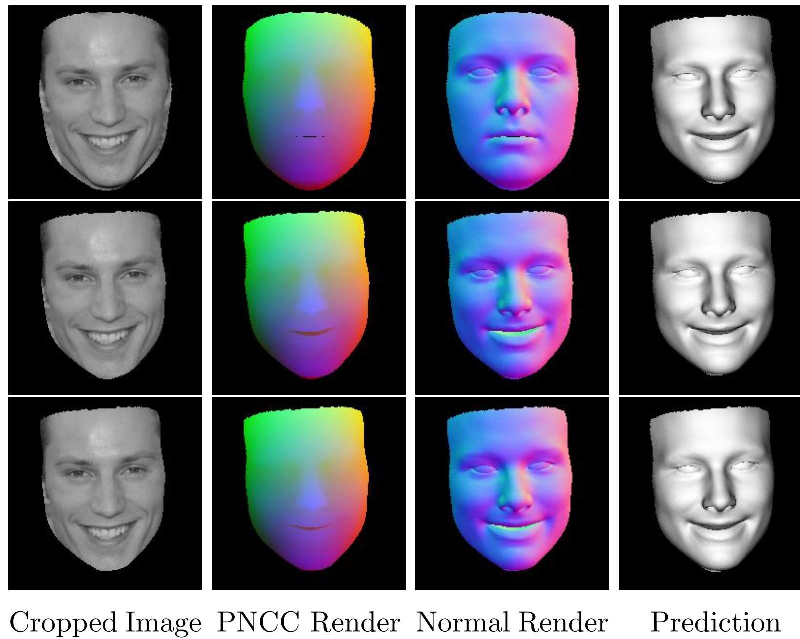

Figure 4: Example of the iterative model, the first row is the initial input with the average geometry, and the last row shows the reconstruction after 3 iterations.

4.2 Single Image Reconstruction

In order to test the single image reconstruction I used images from various face detection and recognition datasets in addition to various photos that I found online.

In *Figure* 4, you can see how the iterative process works.

Figure 5 shows the final reconstruction from single gray scale images.

Figure 6 shows the final reconstruction from single images of celebrities. Here I show the reconstruction on top of the original image to show how the model fits the shape of the face in the original image.

This evaluation alongside the synthetic one were the main criteria according to which I judged the quality of the reconstruction model before t I moved on to implement the video and real-time systems.

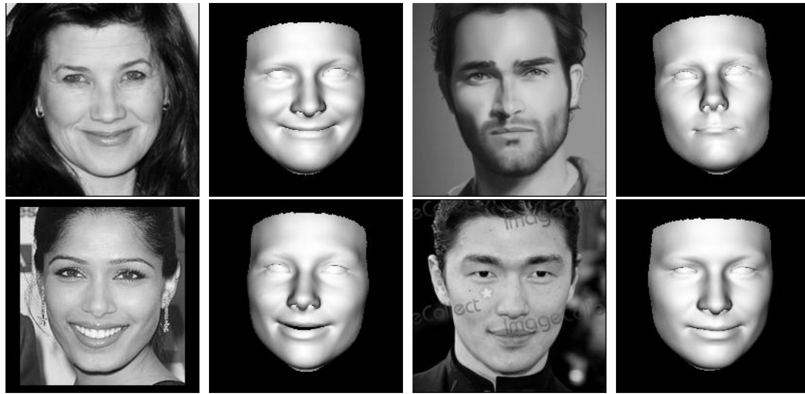

Figure 5: Reconstruction examples of single images. The images were taken from a dataset of grayscale face images.

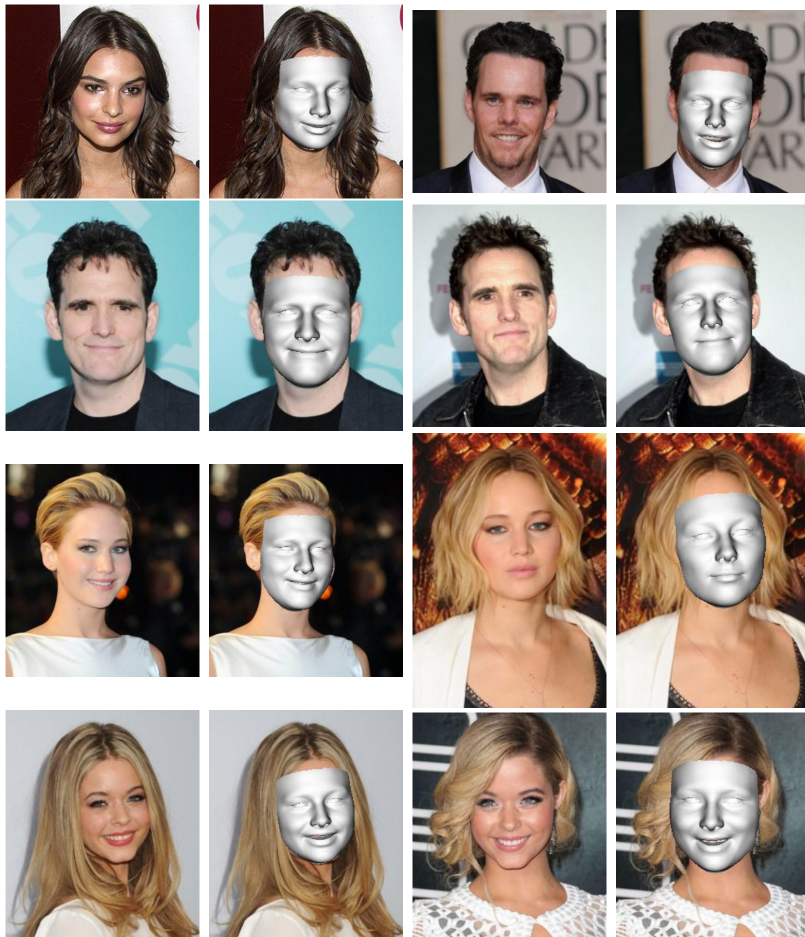

Figure 6: Reconstruction examples of single images. The images are random celebrity photos.

4.3 Video Reconstruction

For the evaluation of the video reconstruction system I picked videos of celebrity interviews from youtube and other sources online.

I did the reconstruction on samples of the videos that I found because using the video as is proved to be difficult because of all the jump-cuts and changing angles that are usually in these videos.

Figure 7 shows examples of a couple of frames from those videos.

4.4 Real-Time Reconstruction

There are several modes that I can run the real-time system in, which mainly affect the frame rate and how fast the capture is. I found that faster frame rate doesn't necessarily look better because the face detector is rather noisy.

Figure 8 shows some screen shots from the real-time reconstruction.

Figure 7: Reconstruction examples of video. The videos were taken from a celebrity interviews on youtube.

Figure 8: Reconstruction examples of real-time video capture.

5 Conclusions

In summary, the scope of this project included implementing the face reconstruction system from single image first using a deep learning approach, and then try to take the single image results and build a video/real-time system.

I showed that the methods for reconstruction a single image from the literature can work in real time, and it can capture the general structure of the face well.

There's much more work that could be done such as extracting fine details, improving the model or even trying to experiment with a non-model based approach (more information in the next section).

similar to the case of the single image this technology has many potential interesting uses such as motion capture, video processing and other computer vision tasks.

6 Further Work and Suggestions

In this section I will go over potential improvement and expansion ideas to the work I have done in the project.

6.1 Detail Extraction

Currently the system only extracts a coarse structure of the face and the reconstruction still doesn't bare much resemblance to the actual face.

One way to overcome this shortcoming, is to implement a network similar to the fine network defined in [4], however, this approach might be to costly in terms of speed and could impede the real-time version.

Another approach that is most performance friendly is to use a more classical approach to the shape from shading problem in order to extract the fine details.

6.2 Modifying The Model

One of the most significant bottlenecks in the quality of the reconstruction is the strength of the model. The current model simply takes PCA [1] a set of faces.

One way to build a more expressive model, is to build an auto-encoder, which consists of a fully connect neural network with a single hidden layer. In other words, it contains two matrices $W^{(1)}_{N\times h}$, $W^{(2)}_{h\times N}$ which transform the geometry into the representation space and vice versa, which N is the size of the geometry and h is the dimension of the representation space.

Such auto-encoder is trained using standard gradient-based procedures such as $NoiseContrastiveEstimation$. and the goal of the auto-encoder is to reproduce the geometry that was given as an input, therefore, we can use a loss similar to the one used for the reconstruction network.

The assumption here is that this model can learning stronger representations because it's not constrained by linearity like PCA.

Another advantage of such approach is that now we can update the embedding matrices during the training of the reconstruction network, which might yield much more impressive task specific results. This approach is usually practiced in fields like NLP but it is rarely used in computer vision.

6.3 Modifying the Data

As we can see from the second example in $Figure$ 3, the produced PNCC and normal render that simulate the iterative process are quite harsh, and do not resemble the original geometry at all.

This is mainly a result of the fact the linear interpolation factor p is sampled uniformly from $[0, 1]$, which means there are instances where the result is almost entirely the perturbation.

I did not produce the data myself for this project but it might be interesting to test the affect of sampling p differently (uniformly from [0.5,1] for example).

I could also be interesting to experiment with different losses that in-cooperate that perturbation, like the triplet loss used in face detection for example, such that in our task we try to get as close as possible to the ground truth geometry while simultaneously trying to get further from the random perturbation.

6.4 Mobile

Another interesting direction to take this project to is mobile. technically it should be hard to take what I have already done and simply use in a mobile application, but obviously some works need to be done on the speed and memory requirements of the system.

I found that smaller and faster architectures such as MobileNet can achieve decent results. Additionally, other techniques like quantization can be used to reduce the size of the network in memory.

References

[1] 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*, SIGGRAPH '99, pages 187–194, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

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

[3] E. Richardson, M. Sela, and R. Kimmel. 3d face reconstruction by learning from synthetic data. In *2016 Fourth International Conference on 3D Vision (3DV)*, pages 460–469, Oct 2016.

[4] Elad Richardson, Matan Sela, Roy Or-El, and Ron Kimmel. Learning detailed face reconstruction from a single image. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.