Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Personalized Hand Model Parameter Estimation Using Deep Neural Networks

Bachelor's Thesis

Author
Dávid Komorowicz

Advisor
Márton Tóth
Kornél Kis(Robert Bosch Kft.)

December 7, 2018

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Komorowicz Dávid*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2018. december 7.

Komorowicz Dávid
hallgató

Kivonat

A virtuális valóságban való elmerülés elérése jelenleg egy kiemelten aktív kutatási terület. A középpontban mostanáig főleg a képernyőfelbontás és a nyomkövető rendszerek pontosságának növelése állt, amíg a felhasználói élmény személyre szabása kevesebb figyelmet kapott.

A személyre szabás fontos eleme a virtuális valóság átélésének: szerepe az, hogy megelőzze a játékos elidegenedését. Egy ismeretlen személy kezét látni a sajátunk helyett furcsa és zavaró lehet, ami megakadályozhatja, hogy a felhasználó valóban beleélje magát a történetbe.

Jelen dolgozat tárgya egy olyan rendszer részének kifejlesztése, ami képes a felhasználó kezéről készített színes képet egy digitális kézmodellé alakítani. Megvizsgáltuk a publikusan elérhető kézadatbázisokat, és azt találtuk, hogy nem létezik olyan, ami ízületi annotációkat tartalmaz kinyújtott pózban. Ezért már létező adatbázisok felhasználásával létrehozunk egy, a céljainknak megfelelő, új adatbázist. Kéz generálására átalakítunk egy már létező testrekonstrukciót végző modellt, és kicseréljük egy részét különböző korszerű architektúrákra. Betanítjuk a neurális hálót az újonnan létrehozott adatbázison, kiértékeljük az egyes konfigurációk teljesítményét és megvizsgáljuk a "transfer learning" hatását. Azt tapasztaljuk, hogy a formáért felelős paraméterek gyenge felügyelete és bizonyos szabadsági fokokon a kényszerek hiánya természetellenes kezeket generálását eredményezi, két dimenzióban viszont jól alkalmazható ízületek detektálására.

Abstract

Immersion in Virtual Reality is an actively researched topic. A lot of effort has been put into improving display resolution and tracking quality, yet, personalization of the experience has received less attention.

Personalization in a Virtual Reality experience is important to avoid alienating the player. Seeing someone else's hands instead of the player's own feels uncanny and causes discomfort, which can break immersion. This work aims to change the character's hands to resemble those of the player in order to make Virtual Reality offer a truly personal experience, thus making it more immersive.

In this thesis, we present the design of a system that aims to capture the user's hands and turns it into a digital model from a single RGB image. We examine the publicly available hand datasets and find that none of them contain joint annotations with flat hand poses. Therefore we combine existing sources to create a new dataset suitable for our needs. We modify an existing model for Human Mesh Reconstruction to accept hand data and replace part of it with different state of the art architectures. We train the neural networks on the newly created dataset, evaluate the performance of each configuration and examine the effects of transfer learning. We find that the weak supervision on shape and the lack of supervision on certain degrees of freedom make it unable to produce plausible 3D meshes but work well for 2D joint detection.

Chapter 1

Introduction

Recent advances in Artificial Intelligence, specifically Deep Learning, made it possible to take personalization to the next level. It enables automated personalized asset creation for interactive audio-visual experiences such as games and Virtual Reality (VR). A couple of examples are modeling the player's face from a single color photo[9] and generating facial animation from waveform [12][25].

1.1 Motivation

In the majority of computer games, a generic model is shown as the body and hands of the playable character instead of one tailored to the user. This may serve as an important narrative element but it can also distance the player from the character.

Seeing someone else's hands move instead of the player's own causes discomfort. This is especially relevant in Virtual Reality, where the control is, literally, in the player's hands. Hand-object interaction is in the center of a great virtual reality experience.

Although this sounds like a small detail, it can really break the immersion. Currently, most games apply clever tricks, like covering the player's hands (**Figure 1.1**) or showing gloves instead as a workaround.

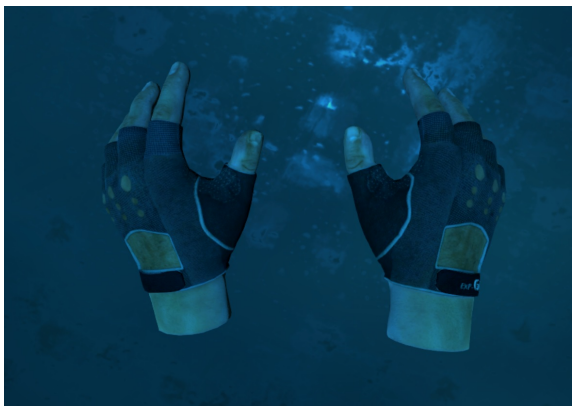


Figure 1.1: Covering the hands of the player to reduce discomfort
(The Gallery: Call of the Starseed)

However, hands differ in shapes and sizes, making these workarounds only a partial solutions to the problem.

1.2 Problem statement

The goal of this thesis is to develop part of a system (**Figure 1.2**), a neural network, that is responsible for capturing the user's hands, converting it into a personalized model and display it in-game (**Figure 1.3**).

The requirements for the system is ease of use and the resulting hand model should be compatible with existing graphics pipelines, driven by the same animation for every personalized model and a common texture space.

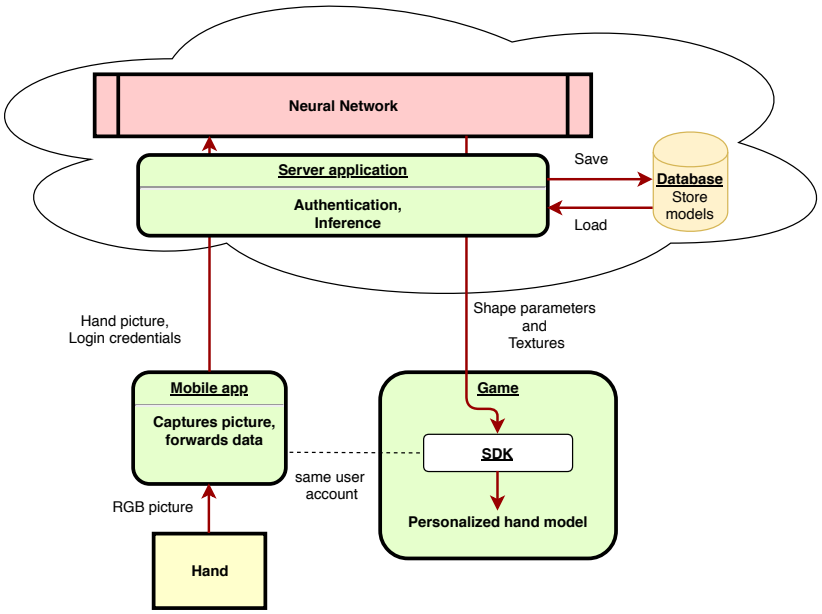


Figure 1.2: Overview of the system

The goal is to develop a deep neural network that can extract *pose* and *shape* parameters from an input picture.

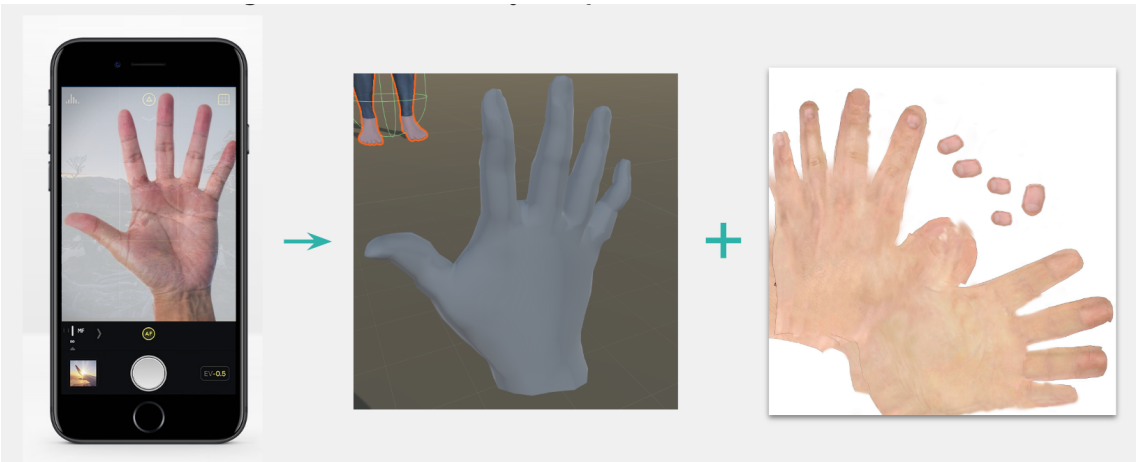


Figure 1.3: Mobile application concept

A similar task has been researched for full body parameter extraction which will be utilized and extended in this thesis to work with hand models.

1.3 Overview

The rest of this work is structured as follows. Chapter 2 introduces the original models about deformable hand modeling, parameter extraction from images and synthetic data generation. Chapter 3 describes and compares the most widely used Deep Learning frameworks. Chapter 5 introduces publicly available hand datasets, compares them and details the creation of a new one from the combination of them suitable for our needs. Chapter 4 describes the setup and configuration of used tools and infrastructure. Chapter 6 describes the necessary modifications to the original model to be utilized with hand models and set up for training. In Chapter 7 we evaluate the results of the trainings. Finally, we draw some conclusions in Chapter 8.

Chapter 2

Related works

2.1 Morphable model formulation

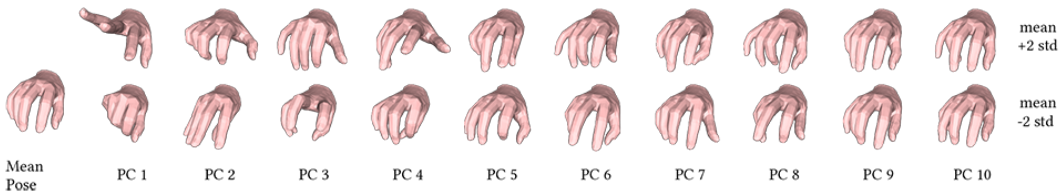


Figure 2.1: Mano model: Principal Components of the pose space

Mano[21] (*hand Model with Articulated and Non-rigid defOrmations*) is a generative hand model based on the *Skinned Multi-Person Linear* model (SMPL)[13]. The model is learned from around 1000 high-resolution 3D scans of 31 subjects in a wide variety of hand poses (**Figure 2.1**). The model is realistic, low-dimensional, captures non-rigid shape changes (deformation intrinsic to a particular person that make them different from others) with pose (kinematic deformations due to a skeletal posture), is compatible with standard graphics packages, and can fit any human hand. Mano deformations are modeled as a combination of skinning and linear blendshapes. The vertices of the final model is given by $M(\vec{\beta}, \vec{\theta})$ as follows:

$$M(\vec{\beta}, \vec{\theta}) = W(T_P(\vec{\beta}, \vec{\theta}), J(\vec{\beta}), \vec{\theta}, \mathcal{W}) \tag{2.1}$$

$$T_P(\vec{\beta}, \vec{\theta}) = \bar{\mathbf{T}} + B_S(\vec{\beta}) + B_P(\vec{\theta}) \tag{2.2}$$

Where a skinning function W (in this case Linear Blend Skinning) is applied to an articulated rigged hand mesh with shape T_P, joint locations J defining a kinematic tree, pose $\vec{\theta} \in \mathbb{R}^{3K}$, shape $\vec{\beta} \in \mathbb{R}^{10}$, and blend weights $\mathcal{W}$.

The mesh T_P is a function of the pose and shape of the hand, which is the sum of the blend shape functions, supplying vertex displacement values, and the mean shape $\bar{\mathbf{T}}$. The blend shape functions are essentially a linear combination of the blend shapes by its parameters. The shape blend shapes are constructed from the first 10 principal components of aligned

scans and pose blend shapes correct the artifacts introduced by the skinning function W as the function of the relative 3D rotation of $K = 15$ joints of the skeleton.

2.2 Parameter estimation

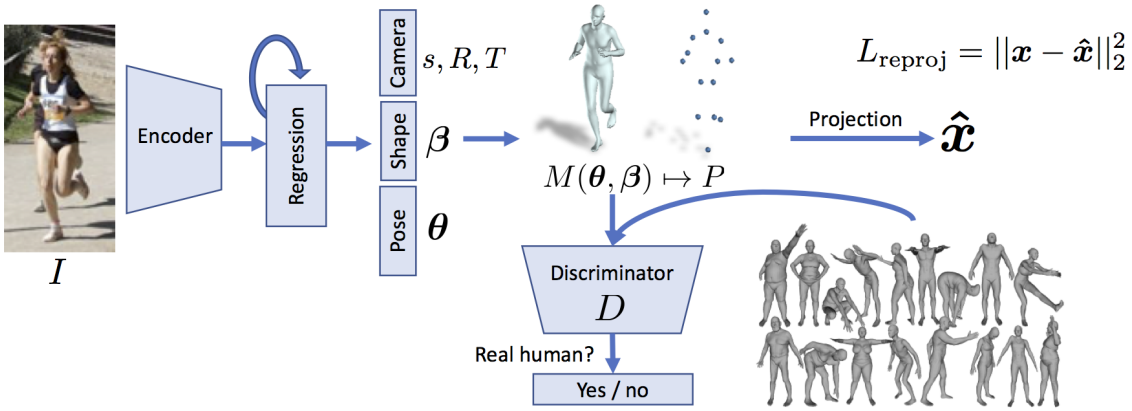


Figure 2.2: HMR architecture

End-to-end Recovery of Human Shape and Pose [11] describe Human Mesh Recovery (HMR), an end-to-end framework for reconstructing a full 3D mesh of a human body from a single RGB image (see fig. 2.2). The main objective is to minimize the reprojection loss between the predicted model and the 2D joint annotations. The network estimates the parameters of the SMPL model, similar to **eq. 2.1**, as well as the camera parameters used for projection. Thus the reconstructed model is expressed as a 85 dimensional vector $\Theta = \{\vec{\theta}, \vec{\beta}, R, t, s\}$. The reprojection loss is calculated as follows:

$$L_{reproj} = \Sigma_i \left\| v_i(\mathbf{x}_i - \hat{\mathbf{x}}_i) \right\|_1 \tag{2.3}$$

Here $\mathbf{x_i} \in \mathbb{R}^{2K}$ is the ith ground truth 2D joints, $\hat{\mathbf{x}}_\mathbf{i}$ is the 2D projection of the predicted joint locations (eq. 2.4) and $v_i \in \{0, 1\}^K$ is the visibility (1 if visible, 0 otherwise) for each of the K joints.

$$\hat{\mathbf{x}} = s\Pi(RX(\vec{\theta}, \vec{\beta})) + t \tag{2.4}$$

where Π is an orthographic projection, $\{s, t, R\}$ are the scale, translation and rotation parameters respectively, totalling $1 + 2 + 3 = 6$ parameters. While $X(\vec{\theta}, \vec{\beta})$ corresponds to the posed skeleton rig corrected by the shape parameters.

The model uses ResNet-50, pre-trained on ImageNet[22], to encode the input image $I \in \mathbb{R}^{224\times224}$ into $\phi \in \mathbb{R}^{2048}$ vector. This is sent to the iterative 3D regressor module (IEF), which outputs Θ with the model parameters, refining them over $T = 3$ iteration, starting from the mean values.

The key idea behind this technique is that the joint locations of the model depend on the $\vec{\beta}$ shape parameters (**eq. 2.1**), therefore optimizing for joint locations will implicitly optimize the shape parameters.

Anthropometrically implausible 3D bodies are penalized by a discriminator trained on ground truth 3D pose and shape data to tell whether SMPL parameters correspond to a real body or not. This serves as a data-driven prior.

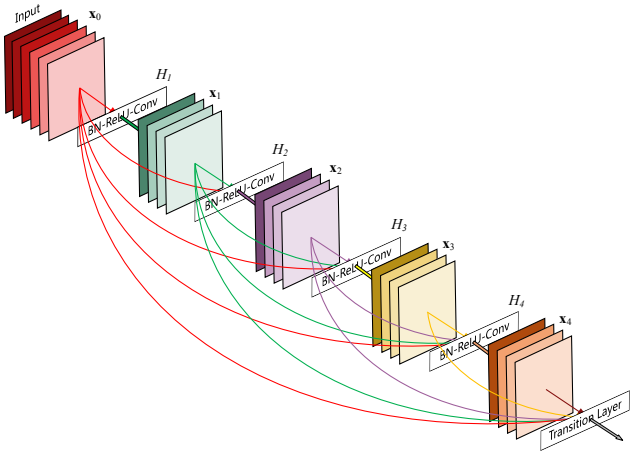


Figure 2.3: DenseNet architecture

DenseNet[8] is a Convolutional Neural Network, the logical extension of ResNet[7], featuring skip connections between intermediate layers. It connects all layers directly with each other in a feed-forward manner (as seen in Figure 2.3), which alleviates the vanishing-gradient problem, strengthens feature propagation, encourages feature reuse, and substantially reduces the number of parameters.

DenseNet achieves state of the art results on a variety of problems including image classification on ImageNet[22].

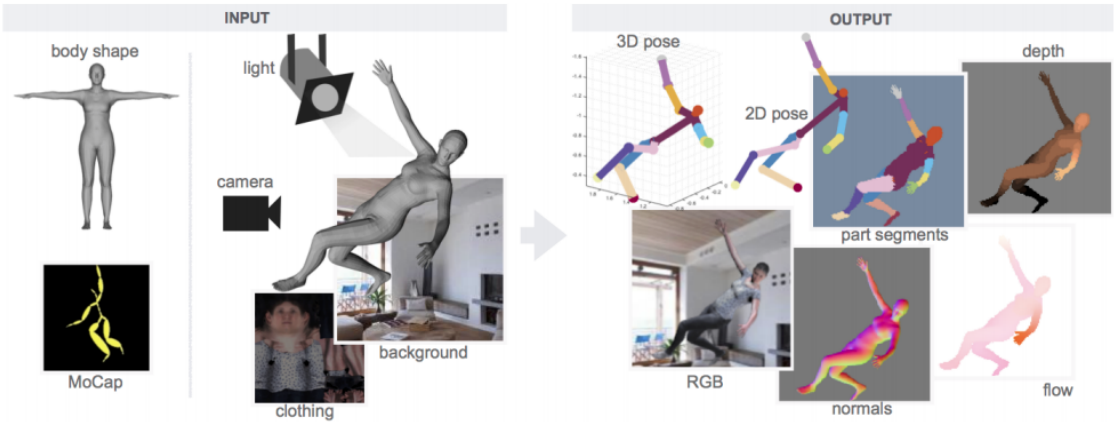


Figure 2.4: Pipeline for generating synthetic training data

Learning from Synthetic Humans[28] train a neural network from synthetic images to estimate human parameters and achieves high accuracy on real-world images. The images are generated using the SMPL model with randomly chosen parameters from ground truth data. This includes textures, lighting direction, camera position and background picture. In addition to color images, this method is able to generate normal, depth, optical flow and part segmentation maps. The process is illustrated in Figure 2.4.

Chapter 3

Deep Learning frameworks

In this section we will compare the most popular deep learning frameworks to date. This wide adoption can be attributed to being open source and active community support. All of the ones mentioned have a Python[27] interface.

3.1 TensorFlow

TensorFlow[1] is a library for high performance numerical computation using data flow graphs originally developed and continuously maintained by Google.

TensorFlow has build-in support for a wide range of hardware accelerators, including CUDA GPUs, TensorCores[14], Tensor Processing Units[10] (TPU) and lately general GPU support through the use of OpenCL.

TensorFlow builds a static computation graph prior to running the model. This enables extra optimizations at the cost of flexibility. This is beneficial for huge convolutional neural networks for image processing but makes it almost impossible to implement Recurrent architectures for time series with variable length inputs or outputs. The separation of graph construction and evaluation adds additional complexity which presents a steep learning curve for newcomers.

The graph building phase imposes a significant overhead at runtime which is noticeable when using multiple small networks. Furthermore debugging is extremely difficult, because the computation graph runs natively, on the C++ side. Making it difficult to look into the tensor values at runtime.

Eager execution tries to remedy this shortcoming by using imperative style by evaluating operations without building graphs.

A major differentiating factor compared to other libraries is the support production environments through TensorFlow Serving and inference on mobile devices through TensorFlow Lite. Lite uses quantized weights (8 bit) and low level CPU instructions optimized for mobile. However mobile GPU acceleration still remains untapped.

3.2 Keras

Keras[5] is a high-level neural networks API, running on top of powerful frameworks such as TensorFlow and formerly Theano[26]. It was developed with a focus on enabling fast experimentation. Keras has been designed with ease of use in mind, therefore it falls behind in terms of performance. Keras is now available as part of TensorFlow.

3.3 PyTorch 0.4.1

PyTorch[17] is a Python wrapper for NumPy tensors with GPU support and a package for automatic differentiation. PyTorch is tightly integrated with the Python language, making development and debugging feel natural.

PyTorch builds a dynamic computation graph during execution which enables dynamic architectures and variable length inputs, characteristics of Recurrent Neural Networks. In spite of that it is on par with TensorFlow in terms of performance. This makes it ideal for research, where rapid prototyping is necessary, and even for production in some cases.

A downside of using PyTorch is the need to write separate code for CPU and GPU backends. This is of minor concern currently, as the standard way of training neural networks is using CUDA acceleration, but it is changing in favour of specialized hardware such as TPUs and TensorCores.

PyTorch has poor source code documentation which makes integrated auto-complete tools of minimal use.

3.4 Comparison

These frameworks have become more similar over the years, except the shortcomings detailed above.

We chose PyTorch for its intuitive coding style and flexible architecture which doesn't hinder its performance.

Chapter 4

Tools and infrastructure

This chapter details the development of setup of numerous tools that help streamline the training and evaluation of neural networks.

4.1 Docker

Docker is a program that performs lightweight operating-system-level virtualization, also known as "containerization". A *container* is an isolated environment for development and deployment instantiated from an *image*. A docker container can access the available CUDA capable graphics cards with the NVIDIA runtime[1]. The architectural overview is illustrated in Figure 4.1.

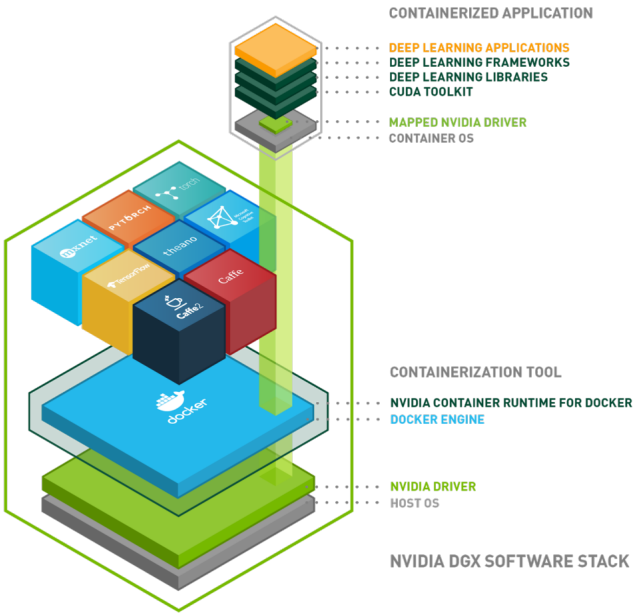


Figure 4.1: Overview of running a deep learning model with docker utilizing Cuda

[1] `https://github.com/NVIDIA/nvidia-docker` (Accessed 24.11.2018)

A docker image can contain the development environment with all of its dependencies. This can then be deployed on other operating systems, in the cloud, with the exact same behaviour[2]. This property enables truly reproducible research[3].

The code used in this thesis can be run with the following docker image:

`https://hub.docker.com/r/dawars/bscthesis/` (Accessed: 05. 12. 2018)

4.2 TensorBoard

TensorBoard is a web tool, developed by Google alongside TensorFlow[1], for inspecting and understanding trainings.

As the most basic usage, the loss values can be logged every N iteration and compared with other runs, detect overfitting, etc. (see Figure 4.2)

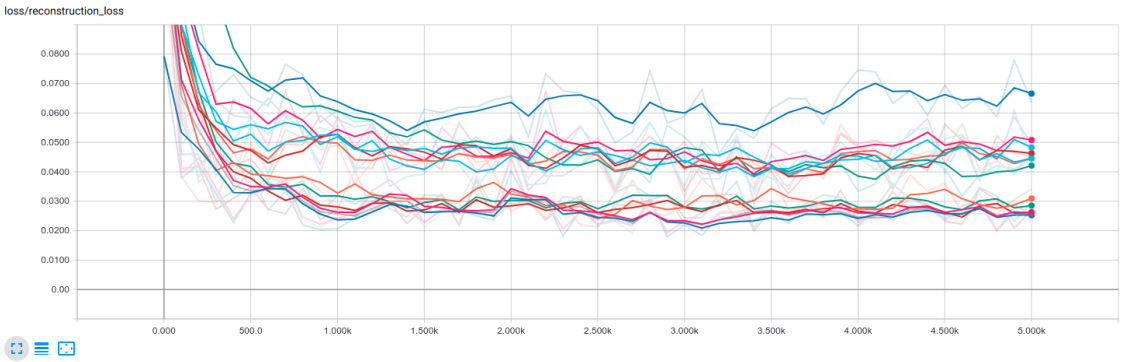


Figure 4.2: TensorBoard losses of different runs

Beside numerical values, other data types can be added, like images, audio, histograms, distributions and embeddings. Histograms are useful to monitor the weights of the network to detect the occurrence of vanishing or exploding gradients.

TensorFlow has native support for TensorBoard but other frameworks are supported through a community driven project named tensorboardX.

`https://github.com/lanpa/tensorboardX` (Accessed: 05. 12. 2018)

4.3 Jupyter notebook

Jupyter Notebook is an open-source web application that allows the creation of interactive documents for Python, among others.

It is ideal for rapid prototyping, when only part of the code needs to be re-run at a time. A major benefit of using it is the ability to display plots and images interleaved with code.

[2]Cuda is only supported under Linux

4.4 Headless rendering

Headless rendering is the process of rendering images without opening a window. This is especially challenging on servers, because a graphical interface is not available. A typical rendering process works as follows: The application queries the X server for creating an OpenGL context, once the context is available, OpenGL can talk directly to the GPU, bypassing the X server. This is illustrated in Figure 4.3.

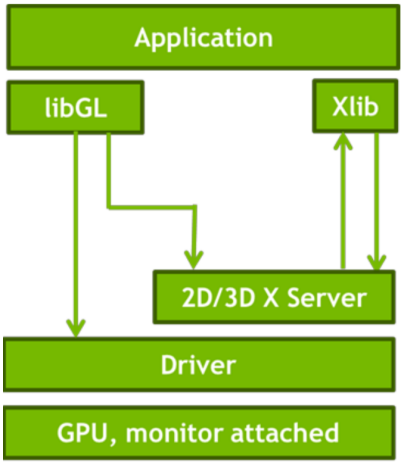


Figure 4.3: Program accessing OpenGL through X server[3]

On a typical server setup, where a display is not attached and the server is accessed through SSH, the X server is not running. Without it, programs cannot access the GPU via OpenGL.

We can configure it however, to start regardless of display availability, using the following configuration:

```
Section "Screen"
    Identifier     "nvidia"
    Device         "nvidia"
    Option         "UseDisplayDevice" "none"
EndSection
```

Listing 4.1: X server configuration

During training we wanted to examine the quality of the predictions by rendering the model and log in TensorBoard periodically. For that we didn't find a sufficiently fast and simple Python library. We needed a library that can create an OpenGL context without opening a window and render images as fast as possible not to hinder the training process.

We developed a library, called *libPaprika* to fulfill these requirements. It is designed to render one model with constant topology and varying vertex locations with minimal overhead. It takes as input the list of vertex positions and corresponding index list, similar to the OBJ file format. An image of 256 renders can be seen in Figure 4.4.

The MANO[21] model doesn't supply normals, therefore they had to be calculated manually for shading. The first attempt was naively calculating the vertex normals using the cross product for every vertex in a loop. This turned out to be too slow for our use-case. A signifiant improvement was achieved by replacing the for loop with a vectorized cross product in NumPy[6].

[3]Source: `http://www.nvidia.com/content/PDF/remote-viz-tesla-gpus.pdf` (Accessed 09.11.2018)

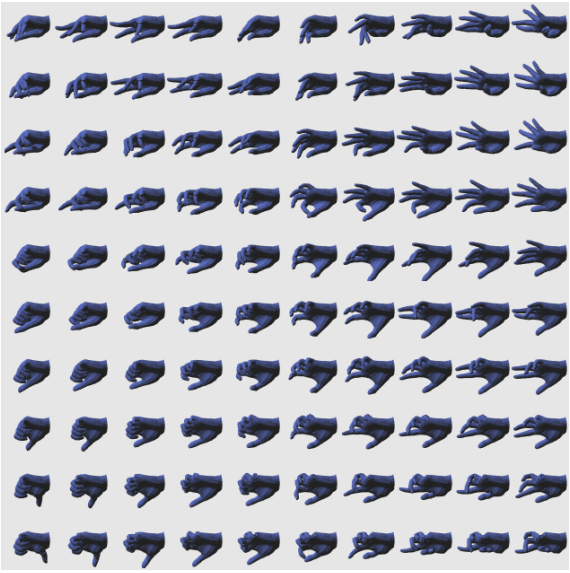


Figure 4.4: Image rendered with libPaprika

The final solution was putting the normal calculations in the geometry shader. This led to an improvement with a significant margin, at the cost of visual quality. In the geometry shader, only the face normals can be calculated, making smooth shading impossible.

The different methods were benchmarked by rendering the above image consisting of 256 mesh renders with supersampling level 8. The results are shown in Table 4.1.

Method	Time
meshrender library	60 sec
for loop with cross product	15 sec
vectorized cross product	7 sec
geometry shader	0.2 sec

Table 4.1: Time of rendering 256 images with 8 supersampling enabled.

This fulfills our requirements, but it could be further accelerated by using direct memory mapping between PyTorch Cuda Tensors and OpenGL. This way the vertex data wouldn't have to be copied to the CPU and then back to the GPU causing additional latency.

4.5 Training server

The neural networks were trained on the server with the following components:

- Intel Core i7 7700 CPU
- 32 GB DDR4 RAM
- GeForce GTX 1080 Ti - 11GB VRAM
- 120GB SSD
- 2TB HDD

The server runs Ubuntu 18.04 LTS with Cuda 9.2.

4.6 Remote code execution

Deep learning servers are expensive and aren't suited for day-to-day use or development. Instead a cheaper laptop should be used for development and the code should run remotely on the server. This enables development anywhere with high performance and trainings can even be started on multiple servers.

The most basic method is logging in to the server through SSH and developing from the command line. The connection can be authenticated by private key making it more secure and convenient. A more advanced way is running code in a remote docker container. Docker has a TCP API that exposes the full functionality to the internet.

Using the combination of these methods, the Professional version of PyCharm synchronizes the code between the client and the server and enables remote execution and debugging. The usage of this IDE greatly simplified the development process.

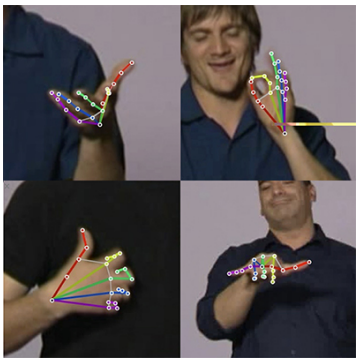
Chapter 5

Datasets

Training a neural network requires a vast amount of data. In this section we explore the publicly available datasets and present how they were transformed to train the utilized neural network architecture.

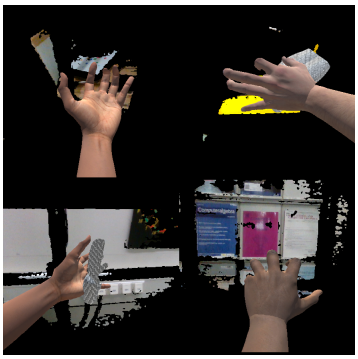
5.1 Public datasets

(a) Mano [21]



(b) HandDB Hand Annotated [23]

(c) HandDB Synthetic [23]



(d) SynthHand [15]

(e) 11k Hands [2]

(f) Bosphorus [29]

Figure 5.1: Public hand datasets.

Mano 5.1(a) contains 1000 high-resolution 3D scans of hands of 31 subjects in a wide variety of hand poses. The scans contain texture information in the form of vertex color. Furthermore each scan contains a low-poly mesh registration with the same topology as described in Chapter 2. Several scans contain hand-object interaction where the held item is colored green. The only annotations belonging to the scans are contained in the file names. This contains a unique id for each subject and the side of the hand (left or right).

HandDB Manual 5.1(b) contains 1912 training and 846 testing in-the-wild images with manual joint annotations. The images have varying aspect ratios and sizes, often containing self occluding hand poses. Each sample is accompanied by a json file with annotations. The file contains annotations for the hand and body joints, as well as bounding boxes for the head and hands.

HandDB Synthetic 5.1(c) contains, in all, 14,261 annotated synthetic images with hand joint annotations in accompanying json file. These images exclusively contain hands. They are constructed by rendering a synthetic hand over a real background.

SynthHands 5.1(d) contains synthetic hand image sequences from an egocentric viewpoint. Each frame contains color as well as depth information. The data is constructed from 10,000 randomly offset background images and hand renders driven by 63,530 frames of real hand motion performance. The images are annotated by 3D joint locations for 21 keypoints including wrist and finger endings. There are images both with and without object interaction. The background is not continuous, often containing large black holes.

11K Hands 5.1(e) is a collection of 11,076 high resolution (1600×1200 pixels) hand images of 190 subjects, showing both dorsal and palmar sides with a uniform white background. The hands are in a flat pose, several subject wear rings or watches and in some cases markings and bruises are visible. Each image is annotated by the *gender*, *age* and *skin color* of the subject and a set of information of the captured hand, i.e. right- or left-hand, hand side (dorsal or palmar), accessories, imperfections.

Bosphorus 5.1(f) consists of 4,846 hand images of 918 subjects, showing the palmar sides. 160 among them have hand images with time lapses of several months. The data was acquired using a commercial scanner with hands placed flat on the glass plate. The file name of the images contain a unique subject id and whether the image contains the left or right hand. Furthermore the real world, physical dimensions can be easily calculated.

5.2 2D Dataset generation

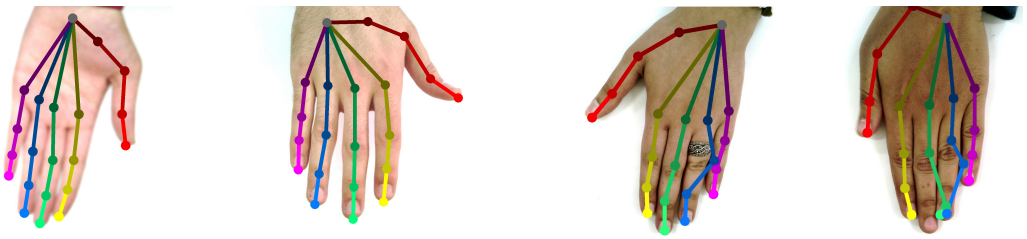
For our use case, we need the hands in a flat position, showing either the palmar or dorsal side, with joint annotations. In addition we require high resolution images centered on the hands for potential future texture extraction. Samples from a diverse set of subjects are desired with regards to age, sex, skin color.

- Mano is a 3D dataset, yet it doesn't contain joint annotations and doesn't have 2D images which could be used as training data.
- HandDB Manual has a wide variety of image sizes and the hands usually only take a small portion of it.
- HandDB Synthetic has a wide variety of hand poses and only a small portion contains flat poses, usable for our training.
- SynthHands is constructed from an egocentric viewpoint which is not ideal and samples with flat hand poses would have to be selected somehow.
- 11k Hands doesn't contain joint annotations. The hands are positioned in a way that the wrist is not visible in the frame most of the time. As a consequence of the rapid capture process, there are a lot of pictures with motion blur in them.
- Bosphorus doesn't contain joint annotations. The shape of the hands are deformed and the background contains a lot of noise due to the capturing process, also only contains the palmar side.

Since there is no dataset that satisfies all of our requirements at this time, we needed to create one from the ones available to us.

11k Hands and Bosphorus fulfill most of the requirements, they both consists of flat hand poses and are high resolution. We decided on **11k Hands**, because it contains more images with more variety. Diversity is often an issue in datasets but this one represents the majority of people all around the world.

We acquire the missing joint annotation via the hand detector module in OpenPose[4] joint detector. The detector takes as input an image, a bounding box and the side of the hand to detect. We use the Python interface easily call it for every image. We set the bounding box to be the full image and read the side of the hand accompanying the metadata csv file of the dataset. The result is a json file containing the joint locations with certainty and an image with the predicted joints drawn on it.



(a) Blurry image (b) Wrist not in image (c) Ring (d) Finger mismatch

Figure 5.2: Typical detection errors.

After manually examining the results, we found some common detection errors as shown in Figure 5.2. We created a program to efficiently filter out these errors. The program goes through the images in the specified directory, displays the image and waits for either a 'y' or 'n' key to save or discard the image respectively. The state machine of the program is detailed in Figure 5.3.

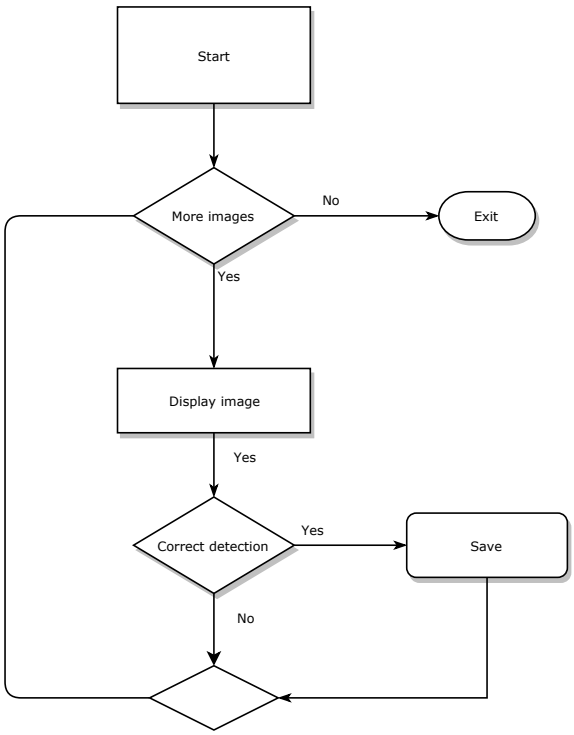


Figure 5.3: State machine of supervision program

The supervision is done in a two phases. During the first phase the correctness of the wrist joint 5.2(b) and quality of the images 5.2(a) were checked quickly. This phase more than halved the usable images. The second, more thorough phase was run on the remaining images and examined the accuracy of the finger joints. The results of each phase is summed up in Table 5.1.

Phase	Time	Resulting images
initial state	-	11,076
OpenPose detection	15 min	11,076
1^{st} pass	1h	5,906
2^{nd} pass	2h	5,773

Table 5.1: Process of manual detection supervision

At the end, the remaining data was split into training and validation sets in a 80-20 proportion. Images from subjects in the training set doesn't appear in the validation set and vice versa. The final dataset contains 4,618 and 1,155 images respectively.

The final data is stored in two Pickle files as a list of the data samples. Pickle (.pkl) is a standard Python format for data exchange.

A data sample is a dictionary of two values, the filename including extension, called "filename" and an ordered list of the joint detections, called "joints". 21 keypoints (Figure 5.4) are detected, including the finger endings and the wrist joint. Such detection contains the absolute position of the joints and the certainty of the detection and is given in the form (x, y, p). The position is given as an integer in pixels, x ranging from 0-1600 and y ranging 0-1200. The position is accompanied by the certainty of the detection. The certainty (p) is a rational number between 0 and 1 inclusive, where 0 indicates impossibility and 1 indicates absolute certainty.

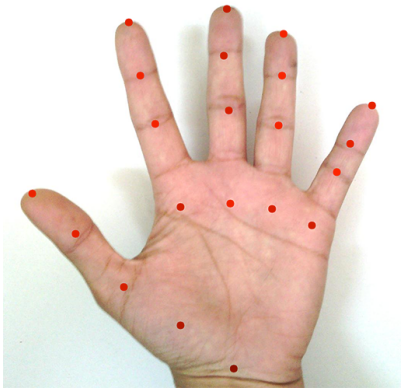


Figure 5.4: Sample from the 11K Joints dataset with detection certainties (darker=more uncertain)

5.3 3D Dataset generation

Capturing high-quality data with 3D annotation is both hard and very expensive. Mano is the only publicly available 3D hand dataset (Section 5.1), however it doesn't contain either 3D shape and pose parameters, or 2D images. We want to train our network with 2D images and 3D labels, namely model parameters, to be able to learn recovering 3D models from 2D input images.

5.3.1 Synthetic data generation

To create 3D annotated samples, we create images in both a top-down and a bottom-up manner. We utilize the 3D scans from MANO[21], which contain texture information and low poly registrations to render synthetic images based on *Learning from Synthetic Humans*[28].

Top-down

Rendering synthetic images enables us to control several properties of the result: The static background, the lighting, the position and rotation of the hand scans. An example can be seen in Figure 5.5.

The process of rendering images with randomized parameters can be automated in Blender using a Python script.

A disadvantage of this method is that the number of unique shapes and poses is limited by the scans in the dataset. An even bigger problem is the lack of ground truth model parameters which makes this by itself unusable.

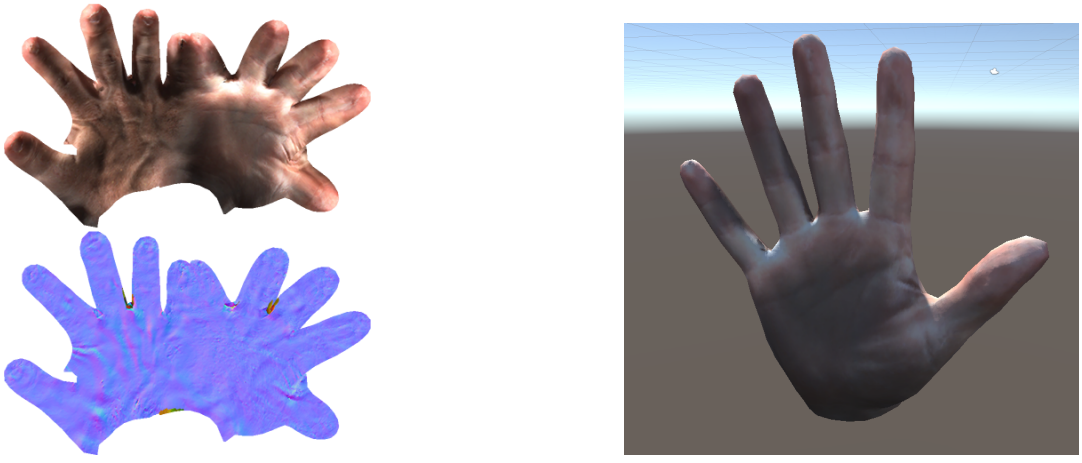
Figure 5.5: Rendered synthetic data sample

Bottom-up

One way to work around this limitation is to generate the meshes from the Mano equation **(eq. 2.1)** instead of the original scans. We can generate any number of unique meshes from the pose and shape parameters.

To render the Mano model we need texture coordinates, which we take from the SMPL[13] model and textures.

We acquire textures by transferring the vertex color information from the original scans to a common texture space. We exploit the fact that each scan has a corresponding low poly registration with the same topology. The transfer process is looking for the texture coordinates corresponding to the closest point of the low poly model and copies the color there. This is done in Meshlab[18] using its "Vertex to Texture" filter. Additionally the extra geometry can be baked into a normal map. The results are shown in Figure 5.6.



(a) (b)

Figure 5.6: a) Baked normal and diffuse texture b) Textured model

Using this method we can generate virtually an unlimited number of unique training samples with respect to shape, pose and texture. One disadvantage is the ability to generate unrealistic images, given arbitrary parameters. We don't know the manifold of hand pose and shape over the parameter space, therefore we cannot guarantee realistic looking results.

5.3.2 Parameter Extraction

To acquire the correct parameters we attempt the extract this information from the low poly scan registrations. Since they have the same topology, it is possible to minimize the pair-wise distance between each vertex as the function of shape and pose for each scan i:

$$Loss_i(\vec{\theta_i}, \vec{\beta_i}) = \left\| Mano(\vec{\theta_i}, \vec{\beta_i}) - LowPoly_i \right\|_2^2 \tag{5.1}$$

The error is minimized using Stochastic Gradient Descent[20]. The optimization process consists of three stages: first only optimizing the pose parameters to get a rough approximation of the pose quickly, secondly only optimizing the shape parameters and lastly jointly optimizing both to fine-tune the model **Figure 5.7**.

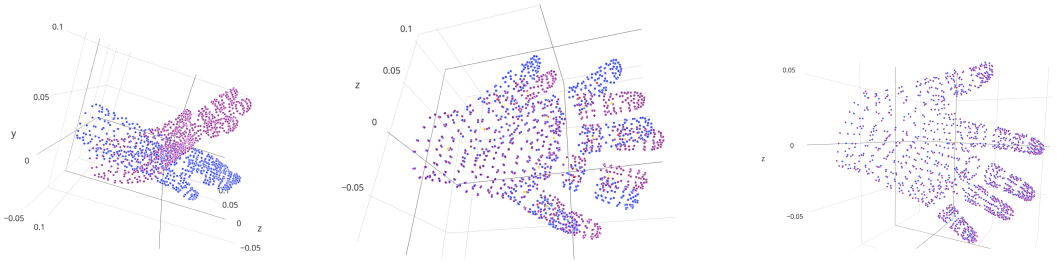


Figure 5.7: Progression of mesh fitting.

The results can be seen in Figure 5.8. This method works reasonably well for extracting the pose parameters, however the acquired shape parameters are not precise enough for our purposes.

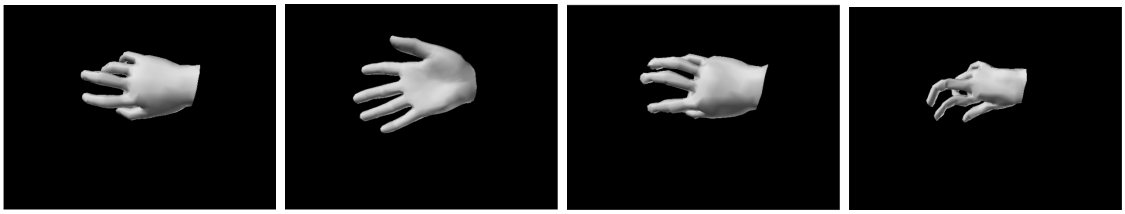


Figure 5.8: Results of parameter extraction.

To improve accuracy, we take into account the fact that there are multiple scans of the same subject. $LowPoly_{i,j}$ denoting the i^{th} scan registration of the j^{th} subject. Therefore the shape parameters are shared between these fittings. We assume that sharing the shape parameters between multiple scans reduce the effects of noise in the measurements and act as a regularizer.

$$Loss_j(\vec{\theta}_{1...n}, \vec{\beta}_j) = \sum_{i=1}^{n} \left\| Mano(\vec{\theta}_i, \vec{\beta}_j) - LowPoly_{i,j} \right\|^2 \tag{5.2}$$

We optimize it with gradient descent both directly and by taking the the elements of the summation iteratively, the results are recorded in TensorBoard (Figure 5.9). The images are rendered with libPaprika (see Sec. 4.4).

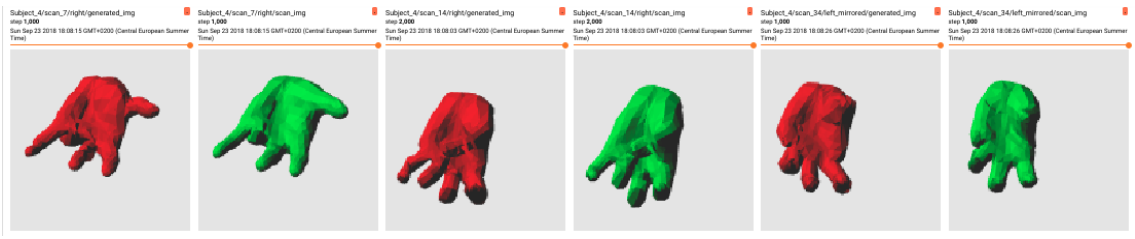


Figure 5.9: Joint Mano parameter fitting

This results in better quality fittings but in spite of that, the shape parameters are still not precise enough.

From this we conclude that scan registrations are not in the hand shape space, even though they were learned from them, hence the unrealistic results even with small error values.

Chapter 6

Model

The model we are using is a modified version of the HMR framework, presented in Chapter 2. In this chapter we describe the modifications necessary to enable the prediction of hand models with our newly created dataset.

6.1 Architecture

The model used in this thesis is the modification of the network proposed in "End-to-end Recovery of Human Shape and Pose" [11] designed for estimating shape and pose parameters from full body images. We started from an unofficial re-implementation of the network in PyTorch which can be found here:

`https://github.com/MandyMo/pytorch_HMR/` (Accessed: 05. 12. 2018)

We replaced the SMPL model with the MANO model and changed the number of parameters as follows:

Parameters	SMPL	MANO
Vertex count (N)	6,980	778
Number of joints (K)	23+1	15+1
Pose parameters ($\vec{\theta}$)	69	45
Shape parameters ($\vec{\beta}$)	10	10
$\Theta = \{\vec{\theta}, \vec{\beta}, R, t, s\}$	85	61

Table 6.1: Changing the number of parameters

Integrating Mano into the program posed a couple of issues. The supplied Pickle file with the model parameters are created in Python 2 which is incompatible with Python 3 which we are using. Furthermore, the Pickle file contains instances of the ChumPy package which is not available for Python 3 and had to be removed.

We replaced the visibility coefficient from eq. 2.3 to the certainty of the joint annotations. In our case there are no occluded joints but there are small errors in the labels. This way keypoints with low certainty have a lower influence on training.

Because of the lack of ground truth data for shapes, we encourage the shape parameters to stay in plausible bounds $\vec{\beta} \in [-1, 1]$ by introducing an L1 loss: $L_{shape} = \left\|\vec{\beta}\right\|_1$.

Due to the lack of 3D ground truth data, the discriminator module is removed.

6.2 Data loader

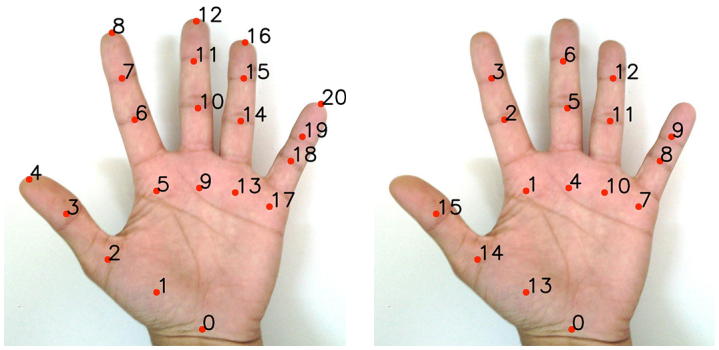
The data loader is responsible for reading the dataset from disk and pre-processing the samples to match the requirements of the neural network. The 11kJoints (Section 5) dataset is loaded by a custom dataloader implemented in the 'hand_joint_dataloader.py' file. Its input is the path to the data itself, containing the joint annotation and the path to the 11K Hands images.

A sample from the dataset contains the filename of the image with extension (e.g. Hand_0010041.jpg) and the joint locations in pixel coordinates relative to the image along with the detection certainty ranging from [0-1].

The thumb has only 2 joint anatomically but the Mano model uses 3 for simplicity. The third joint corresponds to the metacarpal bone, therefore its location can significantly vary between the skeleton of the model and the detections. To avoid this problem, the certainty of the metacarpals are manually set to a low value (0.2).

First the image is loaded, based on the filename, and centered by cropping along the bounding box of the joint positions with a predetermined border. Then the image and joint locations are resized according to the network architecture (224 for ResNet, DenseNet and 299 for Inveption-v3).

The list of joints are remapped to match the order of the Mano model and the finger ending joints are left out, as shown below.



(a) OpenPose joint order (b) Mano joint order

Figure 6.1: Joint order of different models.

In the final step joint locations and image intensity values are normalized between [-1,1] and the image is converted from a HWC to CHW format by swapping the dimensions. The dataloader then shuffles the processed samples and assembles them in mini-batches in the required format.

These processing steps are implemented in the 'image_utils.py' file to avoid code duplication and for testability. Tests are written in a Jupyter notebook for every function, where the outputs can be instantly compared to the expectations.

6.3 Training

We train a number of variants of the base architecture, shown in Table 6.2. In the first training, we use only the reprojection loss (eq. 2.3), for the rest of the trainings we use a weighted sum of the two already introduced losses:

$$Loss_{total} = w * L_{reproj} + L_{shape} \tag{6.1}$$

Where we chose w to be 100 to "dominate" the optimization process.

We train the network in a variety of configurations, as shown in Table 6.2. In addition to the original encoder module, ResNet, we train the network with DenseNet and Inception-v3[24] architectures.

To examine the effects of transfer learning, we train the network in three different modes. First, the weights are initialized with the pre-trained weights on ImageNet and fine-tuned during training (*pretrained*). This is achieved by setting the pretrained parameter to True when creating the model. The second variation differs from the first in that the weights of the encoder are "frozen", so that they don't change during training (*freeze*). And lastly we train the network from scratch with random weight initialization (*random*).

Configuration	Batch size	iterations	Duration	parameters	iteration/sec $[\frac{1}{s}]$
ResNet w/o shape loss	64	100k	1d 2h	23,5M	1.0
ResNet pretrained	32	100k	13h	23,5M	2.1
ResNet freeze	32	100k	10,5h	23,5M	2.7
ResNet random	32	187k	1d 1,5h	23,5M	2.0
DenseNet pretrained	16	54k	4,5h	8M	3.4
DenseNet freeze	32	100k	10,5h	8M	2.7
Inception-v3 freeze	32	44k	13,5h	22M	0.9

Table 6.2: Training configurations

When fine-tuning is enabled for the encoder module, we can train DenseNet with only a batch size of 16 and the Inception model with an even smaller batch size. This is due to the high memory requirement of Gradient Descent on very large architectures[19]. Graphics cards have limited memory capacity, compared to on-board memory, in our case 11GB. Most of the memory is taken up by "feature maps", which are the intermediate results during the forward pass. Even though DenseNet has the fewest parameters, it has a lot of connections, hence the name. This explains the high memory requirements. By freezing the encoder parameters, the model fits into memory and it can be trained with bigger batch sizes.

The table clearly indicates that the most significant factor of the number of iterations computed each second (speed) is the batch size. Training with the batch size of 64 achieve the speed of 1 iteration/second, while the size of 32 results in speed between 2-2.7 iteration/seconds and finally a batch size of 16 results in 3.4 iteration/second. Furthermore, in frozen graphs iterations are computed faster, because the back-propagation is not run on the encoder module and the forward pass is usually negligible. This means that the speed of computing an iteration doesn't depend on the number of tunable parameters directly, but rather, the number of connections.

This does not mean, however, that the network is learning faster. ResNet with random weight initialization takes significantly longer to achieve the same precision as the other variants. (More on this in Chapter 7)

At every 500 iteration we evaluate the performance of the network on the validation set. The evaluation loss is calculated on a batch of 32 images. The loss values are logged in TensorBoard as shown in Figure 6.2.

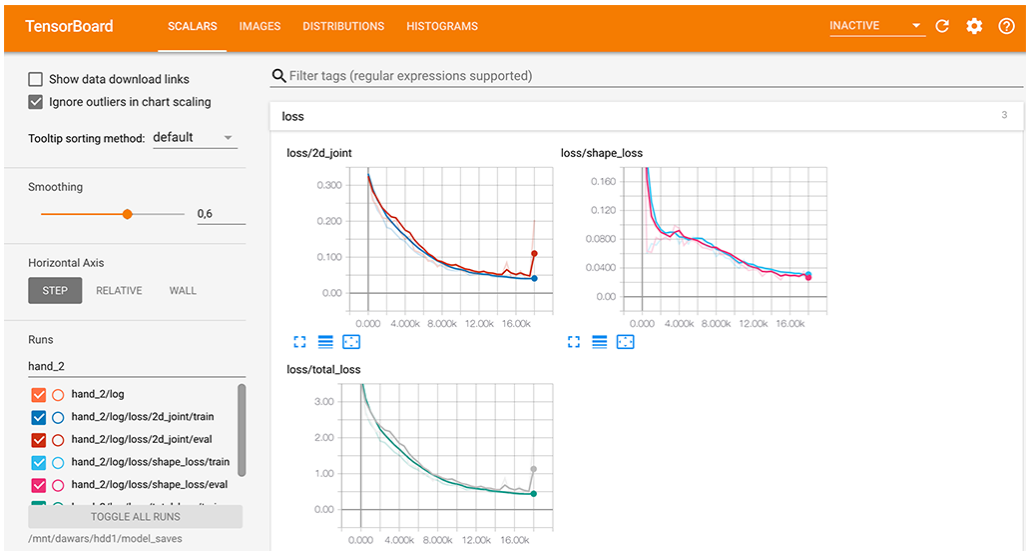


Figure 6.2: Loss values logged in TensorBoard

Additionally we visualize the predicted and ground truth joint locations on a selected set of images. These images show all of the $T = 3$ iterations, which provides insight into the inner workings of the Iterative Error Feedback loop, but for our purposes only the last one is important. The visualizations are also logged in TensorBoard for supervision during training and later inspection. (Figure 6.3)

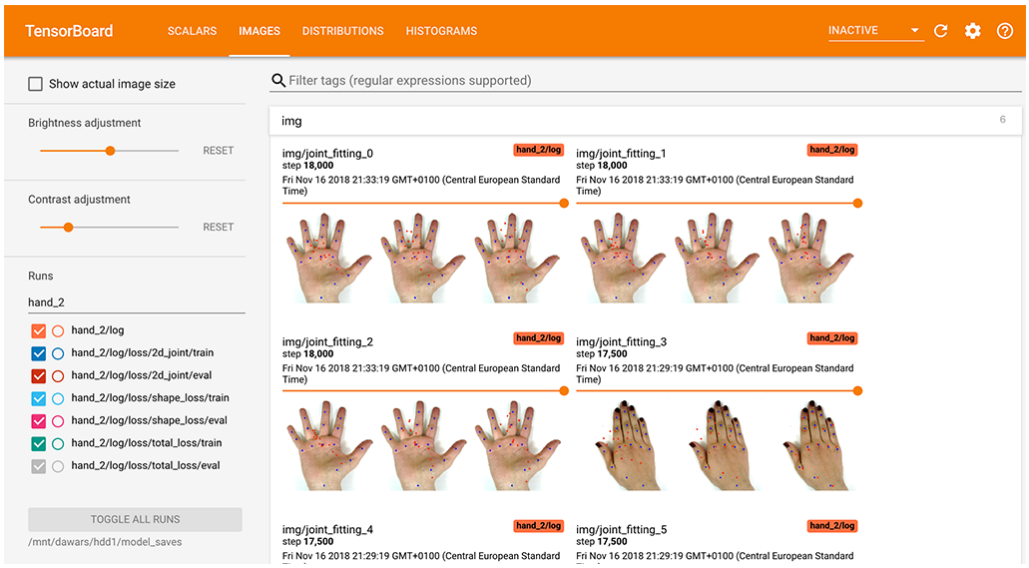


Figure 6.3: Predicted joint locations visualized in TensorBoard

Chapter 7

Evaluations

In this chapter we evaluate the results and compare the different configurations. Table 7.1 shows the loss values at the end of trainings on the training and validation datasets.

#	Configuration	Training loss	Validation loss
1	ResNet w/o shape loss	1.483*	N/A
2	ResNet pretrained	1.675	2.085
3	ResNet freeze	2.535	4.689
4	ResNet random	2.032	2.908
5	DenseNet pretrained	3.044	3.222
6	DenseNet freeze	2.819	5.192
7	Inception-v3 freeze	4.510	6.539

Table 7.1: Training and validation losses (* corrected with w)

As expected, the best training error is achieved by the first configuration which optimizes towards only one loss function. Followed by the configurations where the weights of the encoder aren't frozen, and finally the worst performers are the ones with frozen weights. Furthermore, the data clearly indicates, that the configurations with non-frozen weights have lower validation losses. This means that they generalize better for unseen data samples. This is due to more tunable parameters, learning task specific feature detectors.

2D joint locations With these in mind, the best architecture for 2D joint detection, is the original HMR model, with ResNet, trained with the shape loss (2nd config). This model is tested on an entirely new sample, acquired by the author (Figure 7.1). The image shows the predictions after each ($T = 3$) iteration of the IEF loop.

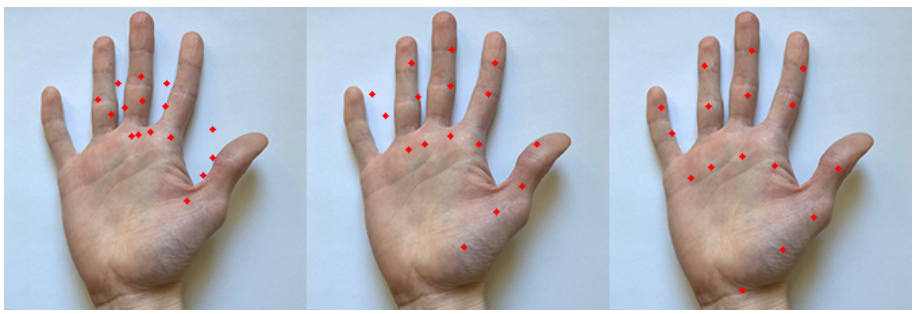
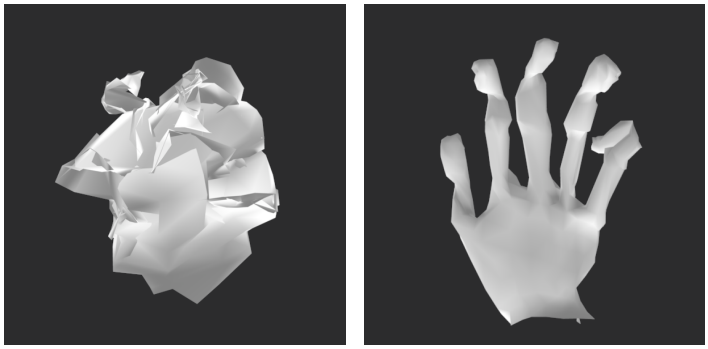


Figure 7.1: Validation on the author's hand

This shows that the model generalizes well, even beyond the validation set. This can be attributed to the robust encoder module (ResNet) and the fact that the input samples are fairly controlled (flat hand, white background).

The progression of the loss values can be seen in Appendix A.1 for each configuration. The validation error significantly deviates from the training error for the frozen models. This reinforces that overfitting is happening, which can be clearly seen in the predictions on the evaluation samples (see Appendix A.2).

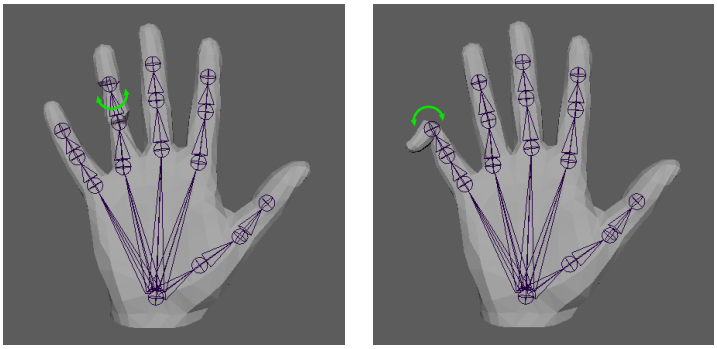
3D mesh Figure 7.2 shows the reconstructed 3D meshes. 7.2(a) is unrecognizable, because the shape parameters explode, (beyond the [-1, 1] bound, up to as high as 5) in order to better match the 2D annotations. This can be attributed to the slight anthropometric differences in the two datasets (Mano, 11k Hands).



(a) Without shape loss (b) With shape loss

Figure 7.2: 3D hand parameter predictions

In 7.2(b), the shape is more plausible, which is thanks to the L1 loss on the shape parameters. The pose, however, contains unnatural joint rotations. This is because there are degrees of freedom in the model that are not constrained by any loss functions. There are two causes: the 2D joint annotations determine the position of the joints, but not the orientation along the bone (Figure 7.3(a)) and the finger endings are not constrained at all (Figure 7.3(b)).



(a) Unconstrained bone rotation (b) Unconstrained finger endings

Figure 7.3: Unconstrained degrees of freedom

Chapter 8

Conclusions

In this work, we created a joint annotated hand database by combining existing sources. We created tools and set up infrastructure to streamline the development process. We modified the HMR model to accept hand data and replaced the encoder module with different state of the art architectures, then we trained the neural network configurations on the newly created dataset. We evaluated the performance of each configuration and examined the effects of transfer learning.

We conclude that the network can learn the task of joint prediction and generalize well beyond even the evaluation data. This is partly due to the encoder module trained on ImageNet, but for best performance, it should be fine-tuned during the training of the regressor module.

We don't get plausible 3D results, because the problem is under-constrained and we lack any 3D supervision.

8.1 Future work

As our goal is the development of a mobile app, the trained model could be exported using ONNX[1] and inference could be run on device using Caffe 2[2], respecting user privacy and complying with GDPR for European users.

As mobile phones with dual cameras are getting a wider adoption, it is possible to easily acquire depth maps either from binocular disparity or via more sophisticated cameras such as the TrueDepth camera of the iPhone X. This extra information could be used to reconstruct more precise hand shapes.

After the start of writing this thesis, a new paper was published improving on shape parameter extraction[16]. According to it, silhouettes and the semantic segmentation of body parts are the most important factors for shape estimation. Using a segmentation map as an intermediate step may increase the quality of the predictions. Such maps can also be synthetically generated as shown previously.

[1] `https://github.com/onnx/onnx` (Accessed: 06. 12. 2018)
[2] `https://caffe2.ai/` (Accessed: 06. 12. 2018)

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[2] Mahmoud Afifi. Gender recognition and biometric identification using a large dataset of hand images. *CoRR*, abs/1711.04322, 2017. URL `http://arxiv.org/abs/1711.04322`.

[3] Carl Boettiger. An introduction to docker for reproducible research, with examples from the r environment. *ACM SIGOPS Oper. Syst. Rev.*, 49, 10 2014. DOI: `10.1145/2723872.2723882`.

[4] Zhe Cao, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Realtime multi-person 2d pose estimation using part affinity fields. In *CVPR*, 2017.

[5] François Chollet et al. Keras. `https://keras.io`, 2015.

[6] Travis E and Oliphant. A guide to NumPy, 2006. URL `http://www.numpy.org/`.

[7] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *ArXiv e-prints*, December 2015.

[8] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. Densely Connected Convolutional Networks. *ArXiv e-prints*, August 2016.

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

[10] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. Vazir Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie,

M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. *ArXiv e-prints*, April 2017.

[11] Angjoo Kanazawa, Michael J. Black, David W. Jacobs, and Jitendra Malik. End-to-end recovery of human shape and pose. In *Computer Vision and Pattern Regognition (CVPR)*, 2018.

[12] Tero Karras, Timo Aila, Samuli Laine, Antti Herva, and Jaakko Lehtinen. Audio-driven facial animation by joint end-to-end learning of pose and emotion. *ACM Trans. Graph.*, 36(4):94:1–94:12, July 2017. ISSN 0730-0301. DOI: 10.1145/3072959.3073658. URL http://doi.acm.org/10.1145/3072959.3073658.

[13] Matthew Loper, Naureen Mahmood, Javier Romero, Gerard Pons-Moll, and Michael J. Black. SMPL: A skinned multi-person linear model. *ACM Trans. Graphics (Proc. SIGGRAPH Asia)*, 34(6):248:1–248:16, October 2015.

[14] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter. NVIDIA Tensor Core Programmability, Performance Precision. *ArXiv e-prints*, March 2018.

[15] Franziska Mueller, Dushyant Mehta, Oleksandr Sotnychenko, Srinath Sridhar, Dan Casas, and Christian Theobalt. Real-time hand tracking under occlusion from an egocentric rgb-d sensor. In *Proceedings of International Conference on Computer Vision (ICCV)*, 2017. URL http://handtracker.mpi-inf.mpg.de/projects/OccludedHands/.

[16] Mohamed Omran, Christoph Lassner, Gerard Pons-Moll, Peter V. Gehler, and Bernt Schiele. Neural body fitting: Unifying deep learning and model-based human pose and shape estimation. In *3DV*, September 2018.

[17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[18] Guido Ranzuglia, Marco Callieri, Matteo Dellepiane, Paolo Cignoni, and Roberto Scopigno. Meshlab as a complete tool for the integration of photos and color with high resolution 3d geometry data. In *CAA 2012 Conference Proceedings*, pages 406–416. Pallas Publications - Amsterdam University Press (AUP), 2013. URL http://vcg.isti.cnr.it/Publications/2013/RCDCS13.

[19] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. *ArXiv e-prints*, February 2016.

[20] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951. ISSN 00034851. URL http://www.jstor.org/stable/2236626.

[21] Javier Romero, Dimitrios Tzionas, and Michael J. Black. Embodied hands: Modeling and capturing hands and bodies together. *ACM Transactions on Graphics, (Proc. SIGGRAPH Asia)*, 36(6), November 2017.

[22] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. DOI: 10.1007/s11263-015-0816-y.

[23] Tomas Simon, Hanbyul Joo, Iain Matthews, and Yaser Sheikh. Hand keypoint detection in single images using multiview bootstrapping. In *CVPR*, 2017.

[24] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the Inception Architecture for Computer Vision. *ArXiv e-prints*, December 2015.

[25] Sarah Taylor, Taehwan Kim, Yisong Yue, Moshe Mahler, James Krahe, Anastasio Garcia Rodriguez, Jessica Hodgins, and Iain Matthews. A deep learning approach for generalized speech animation. *ACM Trans. Graph.*, 36(4):93:1–93:11, July 2017. ISSN 0730-0301. DOI: 10.1145/3072959.3073699. URL http://doi.acm.org/10.1145/3072959.3073699.

[26] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL http://arxiv.org/abs/1605.02688.

[27] G. van Rossum. Python tutorial. Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 1995.

[28] G. Varol, J. Romero, X. Martin, N. Mahmood, M. J. Black, I. Laptev, and C. Schmid. Learning from Synthetic Humans. *ArXiv e-prints*, January 2017.

[29] Erdem Yoruk, Ender Konukoglu, Bulent Sankur, and Jerome Darbon. Shape-based hand recognition. *Image Processing, IEEE Transactions on*, 15:1803 – 1815, 08 2006. DOI: 10.1109/TIP.2006.873439.

Appendix

A.1 Losses

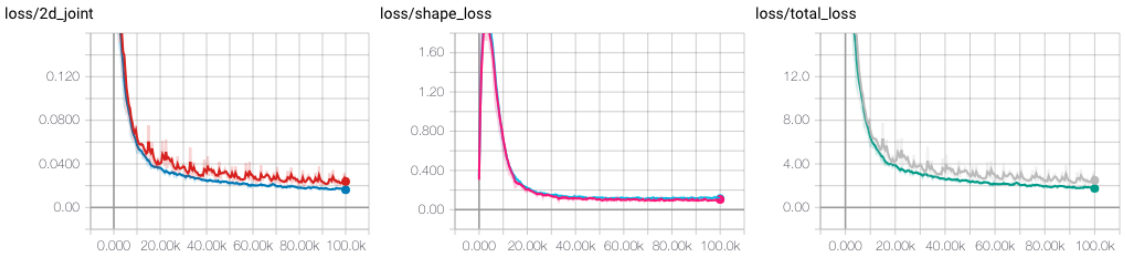


Figure A.1.1: Loss for ResNet, pretrained, fine-tuning, 32 batch size

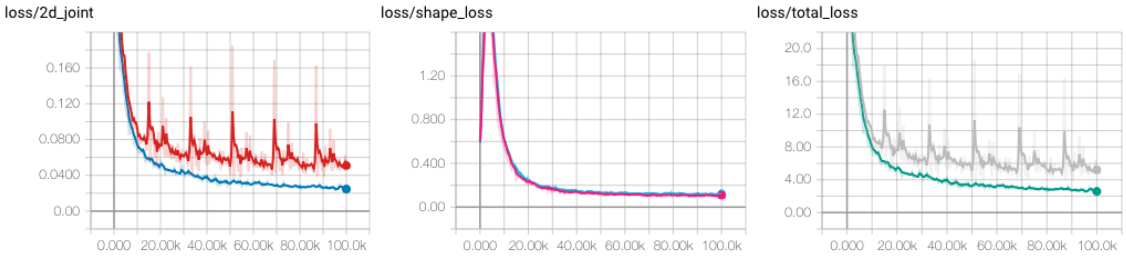


Figure A.1.2: Loss for ResNet, pretrained, frozen, 32 batch size

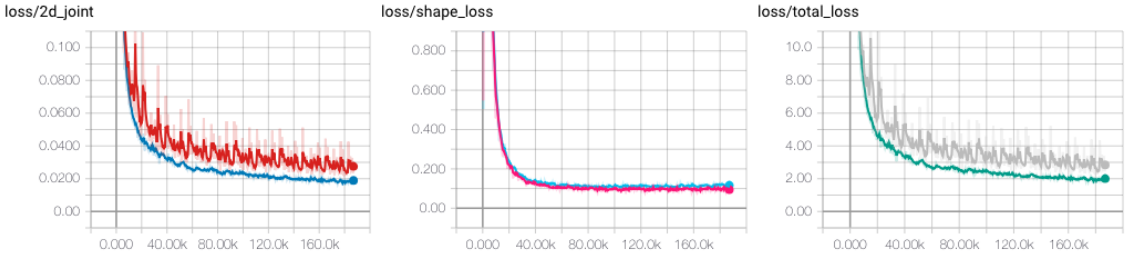


Figure A.1.3: Loss for ResNet, random init, fine-tuning, 32 batch size

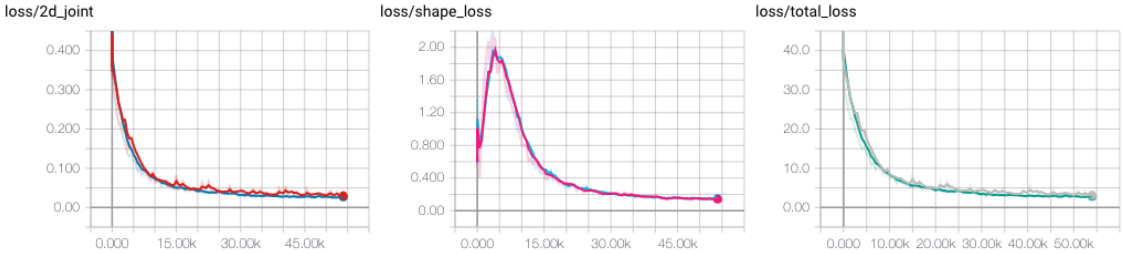


Figure A.1.4: Loss for DenseNet, pretrained, fine-tuning, 16 batch size

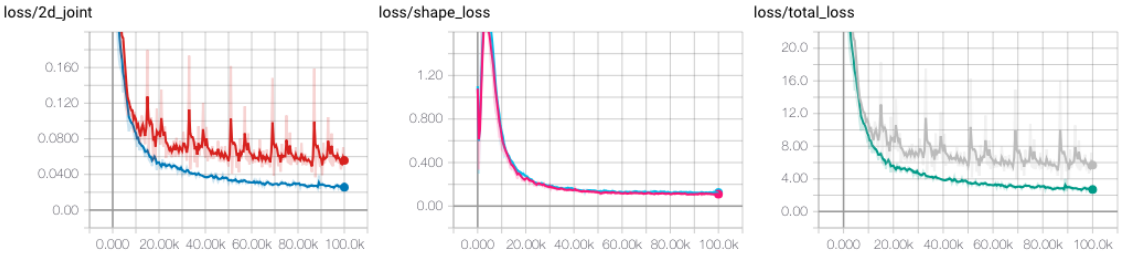


Figure A.1.5: Loss for DenseNet, pretrained, frozen, 32 batch size

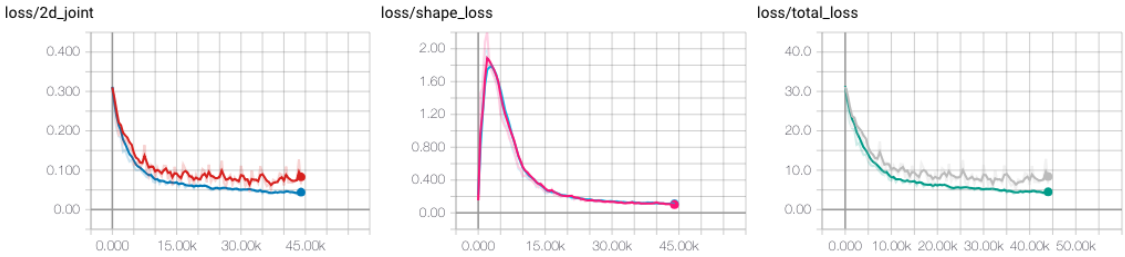


Figure A.1.6: Loss for Inception-v3, pretrained, frozen, 32 batch size

A.2 Predictions on the evaluation set

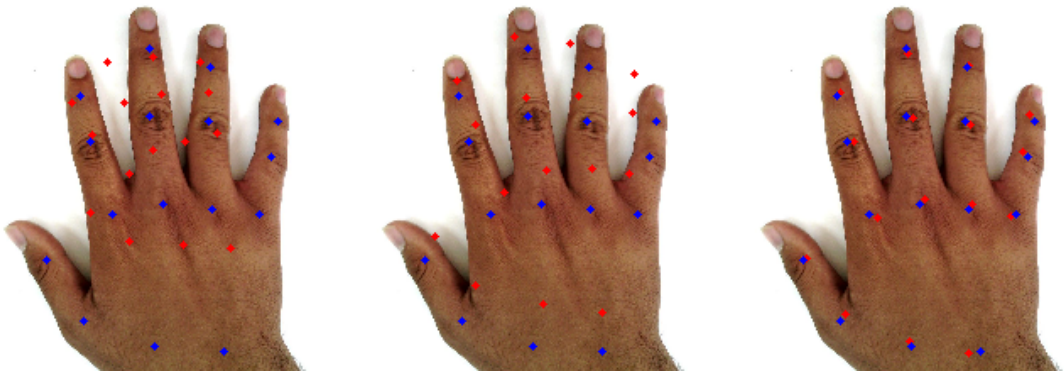


Figure A.2.1: Prediction for ResNet, pretrained, fine-tuning, 32 batch size

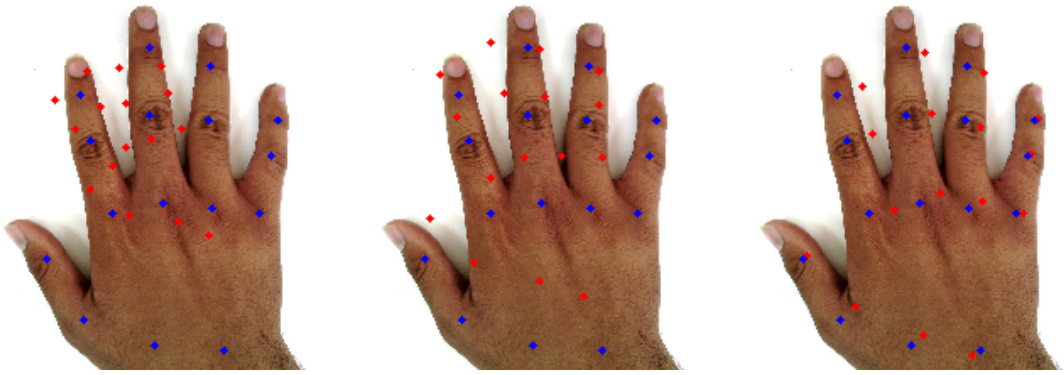


Figure A.2.2: Prediction for ResNet, pretrained, frozen, 32 batch size

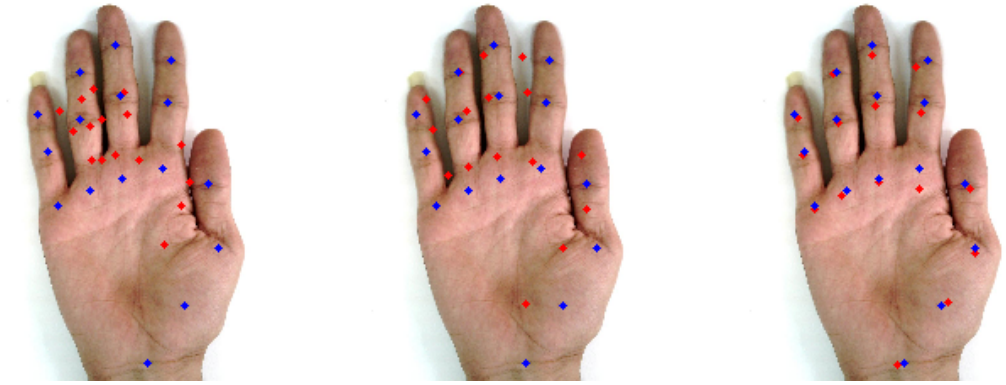


Figure A.2.3: Prediction for ResNet, random init, fine-tuning, 32 batch size

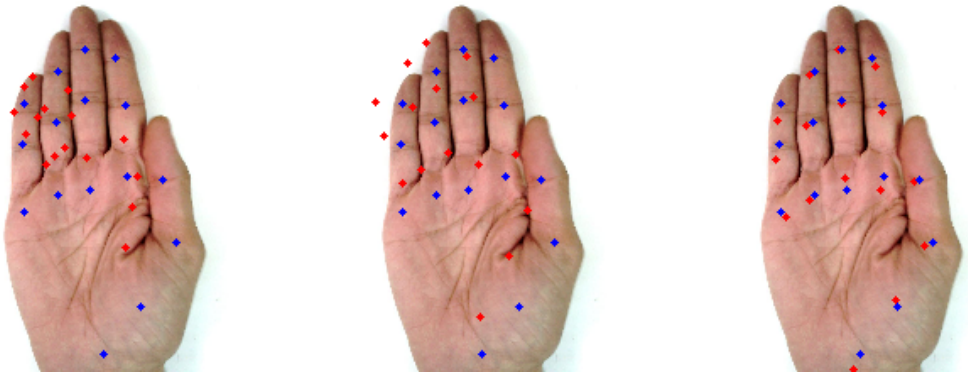


Figure A.2.4: Prediction for DenseNet, pretrained, fine-tuning, 16 batch size

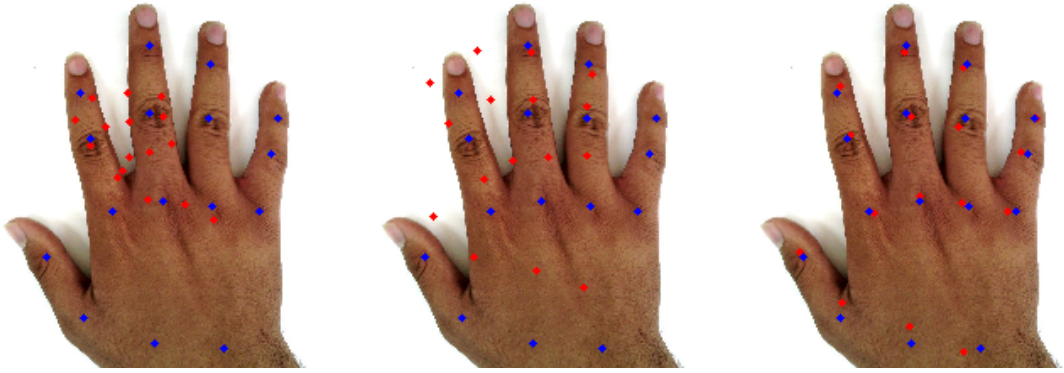


Figure A.2.5: Prediction for DenseNet, pretrained, frozen, 32 batch size

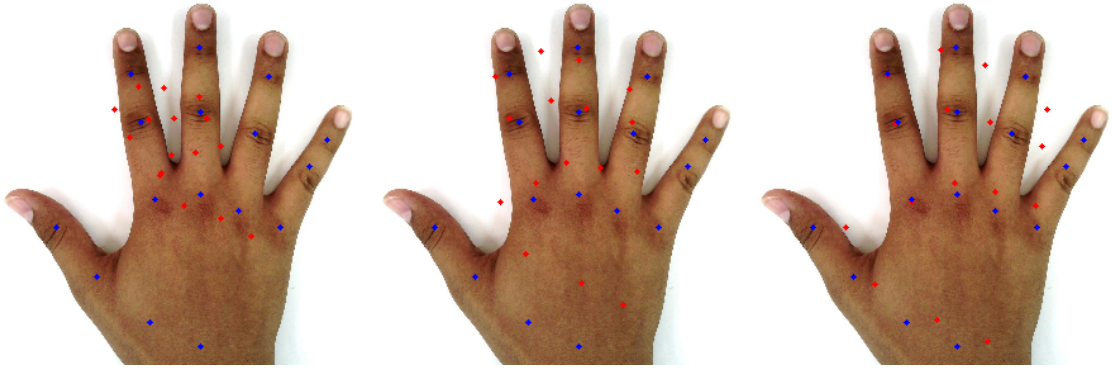


Figure A.2.6: Prediction for Inception-v3, pretrained, frozen, 32 batch size