FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

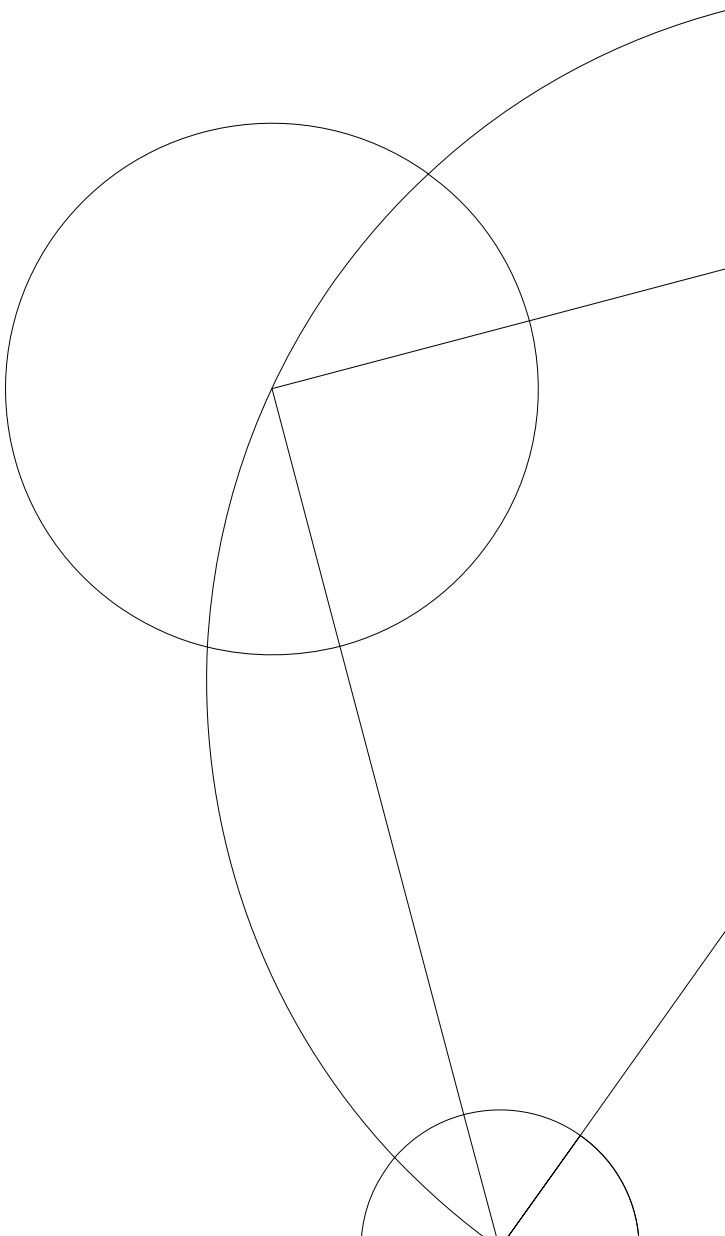
Bachelor Thesis

2D Articulated Human Pose Estimation

Using Explainable Artificial Intelligence

André Oskar Andersen (wpr684)
wpr684@alumni.ku.dk

June 2, 2021



Supervisor
Kim Steenstrup Pedersen kimstp@di.ku.dk

1 Abstract

Contents

1 Abstract **2**

2 Introduction **5**

3 Notation **6**

4 Machine Learning Theory **7**
4.1 Motivation . . . 7
4.2 Machine Learning Paradigms . . . 7
4.3 Evaluation of Machine Learning Models . . . 7
4.3.1 Splitting the dataset . . . 7
4.3.2 Evaluation Metrics for Supervised Machine Learning (Loss Functions) . . . 8
4.4 Neural Networks . . . 8
4.4.1 Feedforward Neural Networks . . . 8
4.4.2 Convolutional Neural Networks . . . 13

5 Autoencoders, Stacked Hourglass, PCA and K-Means **15**
5.1 Autoencoders . . . 15
5.2 Stacked Hourglass . . . 15
5.2.1 Motivation behind using the Stacked Hourglass . . . 15
5.2.2 The Residual Module . . . 16
5.2.3 The Hourglass . . . 16
5.2.4 The Stacked Hourglass . . . 17
5.3 Principal Components Analysis (PCA) . . . 17
5.4 K-Means Clustering . . . 18

6 The Dataset **19**
6.1 The COCO Dataset . . . 19
6.2 Data Preprocessing . . . 20
6.2.1 Creating the test dataset . . . 20
6.2.2 Preprocessing the images . . . 20
6.2.3 Handling the labels . . . 21

7 Experiment **22**
7.1 Configuration Details . . . 22
7.2 Results . . . 24
7.3 Training Details . . . 25

8 Interpreting the Model **26**
8.1 Motivation . . . 26
8.2 Verifying the Effects of Skip-Connections . . . 27
8.3 Shape Analysis of the Latent Space . . . 27
8.4 Using Clustering to Seperate the Latent Space . . . 29

9 Improving the Model **35**
9.1 Motivation . . . 35
9.2 Configuration Details . . . 35
9.3 Results . . . 36

10 Discussion **38**
10.1 Summary of Obtained Results 38
10.2 Comparison of Models 38
10.3 Hvorfor er mine resultater dårligere/bedre end Newell/Camilla? 38
10.3.1 Forskelle 39
10.4 Future Work 39

11 Conclusion **40**

12 References **41**

2 Introduction

It is common knowledge, that the real-world use of artificial intelligence and machine learning is growing rapidly. With this grow the need for accurate computer vision models is also growing. One widely used usage of computer vision is *human pose estimation*, where a machine learning model is used for estimating the pose of one, or multiple, humans. These models have many real-world applications, such as motion analysis, augmented reality and virtual reality [25].

As the complexity of these models has increased, the models have started to work more and more as a "black box", where it can be difficult to understand how the models work and why they work as they do. This can lead to problems such as distrust, redundancy or difficulty with improving the performance of the models [20].

The goal of this thesis is thus to develop a model for human pose estimation, as well as interpretating the developed model with respect to getting an understanding of how the different parts of the model works, as well as checking for any redundancy in the model.

In the remainder of this thesis, Section 4 introduces the basic machine learning theory and Section 5 introduces the most important algortihms used throughout the thesis. In Section 6 the used dataset and its preprocessing is described. We then describe our development of a model and its respective results in Section 7, which is then explored and interpreted in Section 8. In 9 we then use our knowledge of the model to improve the performance of it. We then discuss our approach and results in Section 10. Lastly, we conclude our results in Section 11.

3 Notation

The notation used throughout this thesis is summarized below

x	A scalar
$\boldsymbol{x}$	A vector
$\boldsymbol{X}$	A matrix
$\boldsymbol{x}_i$	The ith element of a vector $\boldsymbol{x}$
$\boldsymbol{X}_{ij}$	Element located at row i column j in matrix $\boldsymbol{X}$
$\mathbb{R}$	The set of real numbers
$\mathbb{R}^n$	The set of n-dimensional vectors of real numbers
$\mathbb{R}^{n \times m}$	The set of $n \times m$-dimensional matrices of real numbers, where n is the amount of rows and m is the amount of columns
$\lvert \cdot \rvert$	Cardinality
∇f	Gradient of f
$\nabla_{\boldsymbol{x}} f$	Gradient of f with respect to $\boldsymbol{x}$
$\frac{\partial y}{\partial x}$	Partial derivative of y with respect to x
$\mathcal{O}$	Big O-notation
$\odot$	Element-wise multiplication
$\mathcal{N}\left(\mu, \sigma^2\right)$	Normal/Gaussian distribution with mean μ and standard deviation σ^2
$D(a, b)$	An arbitrary distance function, that computes the distance between a and b

4 Machine Learning Theory

Throughout this section the theory of machine learning that will be used in this thesis is described and explained. In Section 4.1 we describe the general motivation behind using machine learning. Then, in Section 4.2 we explain the three most common paradigms in machine learning and how we will use some of them. In Section 4.3 we give a brief overview of how to evaluate a developed model. Lastly, in Section 4.4 we describe the mathematics behind feedforward- and convolutional neural networks .

4.1 Motivation

It can be difficult for humans to recognize certain patterns and trends in data. This becomes more difficult the greater the quantity of the data is, which is becomming more and more common with the rapidly growing topic of *Big Data*. For this reason, computers are often used instead of humans to recognize patterns and trends in the data by analyzing the data, which is what is called *Machine Learning*. In this thesis, we will use machine learning in section 7 to develop a model to estimate the 2D pose of a single human in an image. Later, in section 8, we will use machine learning to improve our understanding of the model.

4.2 Machine Learning Paradigms

Machine learning is usally split into the following three paradigms

1. *Supervised learning* where the data consists of features and labels. By analyizing the data the algorithm learns to predict the labels given the features [5]. Supervised learning is further split into *classification* and *regression*. If the value of each label is limited, then the task is a classification task. If the value of each label is not limited, then the task is a regression task.

2. *Unsupervised learning* where the data only consists of features. The algorithm then learns properties of the data, without any provided labels [5].

3. *Reinforcement learning* where the algorithm learns to perform the action in a given environment that yields the highest reward [1].

In this thesis we will make use of supervised learning when developing our model for pose estimation. Later, unsupervised learning is used when we explore our developed model.

4.3 Evaluation of Machine Learning Models

When developing a machine learning model it is important to know how trustworthy the developed model is. This is usually done by testing how good the model is at generalizing unseen data, which is done by making use of *evaluation metrics*.

4.3.1 Splitting the dataset

When developing a machine learning model, the data needs to both create the model, but also to evaluate the model. For the evaluation of the model, one of the two following techniques is usually used

1. *Cross validation* where the data is split into K random non-overlapping chunks of equal size. The model is then trained for K rounds on $K - 1$ of the chunks, where the last chunk is used for evaluating the model. After each round the parameters of the model is reset to ensure one round does not affect another round. After the K rounds the average loss of the K rounds is the loss of the model [17].

2. *Train-validation-test* where the data is split into 3 random non-overlapping chunks. The training dataset is then used for training the model and the validation dataset is used for evaluating the model as it is being developed - this often means, that the *hyperparameters*, the paramters that are not possible to fit from the data, are being tweaked to yield the best validation loss. Lastly, the testing dataset is used as a final evaluation of the model to yield an unbiased evaluation of the model. Once the testing dataset has been used it can no longer be used for evaluating the data, as this ensure an unbiased evaluation [7].

Throughout this thesis the train-validation-test technique will be used over cross validation for evaluating the developed models. This is done, since cross validation is better suited for smaller datasets, as the runtime is much greater than the runtime of the train-validation-test technique.

4.3.2 Evaluation Metrics for Supervised Machine Learning (Loss Functions)

When we have trained a model, we need to somehow evaluate how well the model performs on unseen data. This is usually done by making use of evaluation metrics or *loss functions*. There are many different loss functions, each with their own advantages and disadvantages. One of the most common loss functions for regression is the *Mean Squared Error (MSE)*, defined as

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where y_i is the true value of the ith observation and $\hat{y}_i$ is the estimated value of the ith observation. Thus, MSE measures the average squared difference between the true observation and the estimated observation. The aim of a model is thus to make the MSE as small as possible [8].

4.4 Neural Networks

In recent years *deep learning* and *neural networks* have revolutionized the use of machine learning. In this thesis a neural network will be used for performing the human pose estimation. Throughout subsection 4.4 the theory and mathematics behind neural networks is described and explained.

4.4.1 Feedforward Neural Networks

Algorithm 1 Gradient Descent [24]

Require: Learning rate η
Require: Starting position $\boldsymbol{\theta}$
Require: Function to minimize f
1: **while** stopping criterion not met **do**
2: Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta\nabla f(\boldsymbol{\theta})$
3: **return** $\boldsymbol{\theta}$

Overfitting and Regularization
The main goal of a machine learning model is to generalize well on unseen data. This can often be difficult, as the model simply "remembers" the training data instead of learning the patterns in the training data. In other words, the gap between the training error and the test error is too large, which is a concept called *overfitting*. Certain techniques are designed to reduce the test error - these techniques are collectively called *regurlization* [4].

Gradient Descent
The goal of a machine learning model when training is to minimize its loss. There are different methods to do so, however, the most common algorithms are variants of *gradient descent*, whose algorithm is described in Algorithm 1. The algorithm works by taking a learning rate η, a starting position $\boldsymbol{\theta}$ and a function f as input, where f is the (loss) function to minimize. It then computes the gradient of f with respect to $\boldsymbol{\theta}$, and subtracts the gradient times η from $\boldsymbol{\theta}$. This is done untill a stopping condition is met, such as when the magnitude of the gradient $|\nabla f(\boldsymbol{\theta})|$ is small or untill a maximum amount of iterations has been reached [24].

Online, Mini-batch and Batch methods
When gradient descent is used in machine learning, computing $\nabla f(\boldsymbol{x})$ is usually done by averaging the gradient of each of the n observations of the trainingset, which is called a *batch gradient method* and is computational inefficient, as the cost is $\mathcal{O}(n)$. It is therefore common to use variants of gradient descents, that reduces the cost of computing the gradient. In *online gradient methods* a single observation from the dataset is used to compute the gradient, which brings the cost down to $\mathcal{O}(1)$. In *mini-batch gradient methods* a subset of the dataset is used to compute the gradient, making the cost $\mathcal{O}(|\mathcal{B}|)$, where $|\mathcal{B}|$ is the mini-batch size [24].
Choosing the right batch size can be difficult, however, there are a few guidelines which one can follow [14] [24]

1. Batch gradient descent uses the fewest iterations, however, each iteration takes the longest to compute. On the other hand, in online gradient descent each iteration is the fastest to compute, however, it is also the method that uses the most iterations. Lastly, mini-batch gradient descent combines the two: it uses less iterations than online gradient descent, but more than batch gradient descent, and each iteration takes less time than in the case with batch gradient descent, but longer than in the case with online gradient descent.

2. A batch size that is of power of 2 can offer in better runtime for some hardware. A batch size that is often used for larger models is 16, however, they typically range between 32 and 256.

3. Smaller batch sizes can offer a regularizing effect, as it is difficult for the model to "remember" the complete dataset from batches that does not represent the whole dataset.

Algorithm 2 RMSProp [4]

Require: Learning rate η
Require: Decay rate ρ
Require: Starting position $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}
Require: Function to minimize f
Initialize accumulation variables $\boldsymbol{r} = \boldsymbol{0}$
while stopping criterion not met **do**
  Sample a minibatch of m observations from the training set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
  Compute gradient: $\boldsymbol{g} = \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i f\left(\boldsymbol{x}^{(i)}\right)$
  Accumulate squared gradient: $\boldsymbol{r} = \rho\boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$
  Compute parameter update: $\Delta\boldsymbol{\theta} = -\frac{\eta}{\sqrt{\delta+\boldsymbol{r}}} \odot \boldsymbol{g}$
  Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
return $\boldsymbol{\theta}$

Optimization Algorithms
Online, mini-batch and batch gradient descent are all optimization algorithms used for estimating the minimum of a function. One problem of these algorithms is, that the learning rate can be difficult to choose. Therefore, there have been developed a range of various optimization algorithms that uses a separate learning rate for each parameter and automatically adapt these learning rates. One of which is *RMSProp*, which has been visualized in Algorithm 2. The algorithm works by using an decaying average that discards knowledge from the past, so that it can converge after finding a convex bowl. The algorithm uses a hyperparameter ρ, that controls the length scale of the moving average [4].

Algorithm 3 Stochastic Gradient Descent [4]

Require: Learning rate η
Require: Initial parameter $\boldsymbol{\theta}$
Require: Function to minimize f
1: **while** stopping criterion not met **do**
2: Sample a minibatch of m observations from the traning set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(n)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
3: Compute gradient estimate: $\hat{\boldsymbol{g}} = \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i f\left(\boldsymbol{x}^{(i)}\right)$
4: Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta\hat{\boldsymbol{g}}$

Another important optimization algorithm is *Stochastic gradient descent* (*SGD*). Unlike RMSProp, SGD does not adapt the learning rate over time, but instead keeps it fixed. The algorithm of SGD has been visualized in Algorithm 3. SGD is very closely related to the algorithm behind gradient descent, however, instead of updating the parameters for each sample, SGD instead uses the mean of n samples. The momentum algorithm works by accumulating a decaying moving average of the past gradients and continuing to move in their direction.

Algorithm 4 Stochastic Gradient Descent with Momentum [4]

Require: Learning rate η
Require: Momentum parameter α
Require: Initial parameter $\boldsymbol{\theta}$
Require: Initial velocity $\boldsymbol{v}$
Require: Function to minimize
1: **while** Stopping criterion not met **do**
2: Sample a minibatch of m observations from the traning set $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(n)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$
3: Compute gradient estimate: $\boldsymbol{g} = \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i f\left(\boldsymbol{x}^{(i)}\right)$
4: Compute velocity update: $\boldsymbol{v} = \alpha\boldsymbol{v} - \eta\boldsymbol{g}$
5: $\boldsymbol{\theta} = \boldsymbol{\theta} + \boldsymbol{v}$

Momentum
One problem with SGD is how slow it often can be. For this reason *momentum* is often used to accelerate learning. The algorithm behind SGD with momentum has been visualized in Algorithm 4. The algorithm works by introducing two new variable; $\boldsymbol{v}$, which is the direction and speed of which the parameters move through the parameter space, and $\alpha \in [0, 1)$, which describes how quickly the contribution of previous gradients decay. Common values of α are $0.5, 0.9$ and 0.99 [4].

Batch Normalization

Batch normalization is a reparametrization method, which is applied to individual layers. If $\boldsymbol{x} \in \mathcal{B}$ is an input to the batch normalization, BN, then batch normalization is done by the following

$$\text{BN}(\boldsymbol{x}) = \boldsymbol{\gamma} \odot \frac{\boldsymbol{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}}{\hat{\boldsymbol{\sigma}}_{\mathcal{B}}} + \boldsymbol{\beta}$$

where

$$\hat{\boldsymbol{\mu}} = \frac{1}{|\mathcal{B}|} \sum_i \mathcal{B}_i$$

and

$$\hat{\boldsymbol{\sigma}} = \sqrt{\epsilon + \frac{1}{|\mathcal{B}|} \sum_i (\mathcal{B} - \boldsymbol{\mu})_i^2},$$

which makes the minibatch have 0 mean and unit variance. $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are then used to make the mini-batch have an arbitrary mean and standard deviation and are two parameters that needs to be learned when the network is being fitted. This helps the network to converge, as the batch normalization keeps centering the mean and standard deviation of the mini-batches [24].

Epoch

An *epoch* is an iteration through the whole dataset during fitting of the network. Multiple epochs are often needed to reach the minimum of the loss function [24].

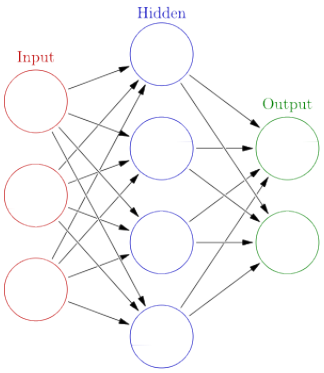


Figure 1: Visualization of a feedforward neural network with a single hidden layer [23]

The Architecture and Forwardpropagation

One of the most common types of neural networks are *feedforward neural networks*, where the data flows unidirectionally through the network. Such a network is visualized in Figure 1. The network is built up of three types of components: the *input layer*, the *hidden layers* and the *output layer*. Each layer is built up of *units*, also called *neurons* (which are visualized as circles in Figure 1), where each neuron has a *bias* assigned to it, and is connected to one or two other layers through *edges* (which are visualized as arrows in Figure 1), where each edge has a *weight* assigned to it. Hidden layers are connected to two other layers - one before the hidden layer and one after the hidden layer - where the input layer is only connected to the next layer in the network and the output layer is only connected to the previous layer in the network.

We can define the network mathemaically by letting $a_n^{(i)}$ denote the value of the nth node in the ith layer, $w_{m,n}$ denote the value of the weight of the edge connecting the nth node in the ith layer to the mth node in layer $i+1$ and $b_n^{(i)}$ denote the bias corresponding to the nth node

in the ith layer.
When data flows through the model it follows the following formula

$$\boldsymbol{a}^{(i+1)} = g^{(i+1)}\left(\boldsymbol{z}^{(i+1)}\right)$$

where

$$\boldsymbol{z}^{(i+1)} = \boldsymbol{W}^{(i+1)}\boldsymbol{a}^{(i)} + \boldsymbol{b}^{(i+1)},$$

$\boldsymbol{W}^{(i+1)}$ is the weights between layer i and layer $i+1$ defined by

$$\boldsymbol{W}^{(i+1)} = \begin{pmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,0} & w_{m,1} & \cdots & w_{m,n} \end{pmatrix},$$

$\boldsymbol{a}^{(i)}$ is the values of the nodes in the ith layer defined by

$$\boldsymbol{a}^{(i)} = \begin{pmatrix} a_0^{(i)} \\ a_1^{(i)} \\ \vdots \\ a_n^{(i)} \end{pmatrix},$$

$\boldsymbol{b}^{(i+1)}$ is the values of the biases of layer $i+1$ defined by

$$\boldsymbol{b}^{(i+1)} = \begin{pmatrix} b_0^{(i+1)} \\ b_1^{(i+1)} \\ \vdots \\ b_m^{(i+1)} \end{pmatrix}$$

and g is an *activation function*, that is typically applied element-wise [4] [19]. One often used activation function is the *rectified linear activation function* (or *ReLU* for short) defined by

$$g(x) = \max\{0, x\}.$$

The ReLU-function is very close to being linear, making the function keep many of the properties of linear functions that make them easy to optimize and generalizing, which are two great advantages of using the ReLU-function. Another great advantage of using the ReLU-function is stated by the *universal approximation theorem* which states, that a feedforward network with a linear output layer and at least one hidden layer with the ReLU-function (or another activation function from a wide class of activation functions) can approximate any continuous function on a closed and bounded subset of $\mathbb{R}^n$ (and actually some functions outside of this class), as long as the network has enough hidden neurons [4].

Backpropagation
Backpropagation is an algorithm used to compute the gradient of the network. It is used together with an optimization algorithm, such as RMSProp, to train the model by minimizing the training loss of the model. Backpropagation happens after data has flowed through the model from the input to the output, and works by computing the gradient of each parameter sequentially from the output to the input of the model. The procedure makes heavily use of the *chain rule* from calculus, which states, that if we let $\boldsymbol{x} \in \mathbb{R}^m$, $\boldsymbol{y} \in \mathbb{R}^n$, g be a function that maps from $\mathbb{R}^m$ to $\mathbb{R}^n$ and f be a function that maps from $\mathbb{R}^n$ to $\mathbb{R}$, then, if we let $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$, we can then compute $\frac{\partial z}{\partial x_i}$ by

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{y_j}{x_i}$$

[4]. If we use this to find the gradient of each parameter, we will find, that the partial derivative for each weight is

$$\frac{\partial L}{\partial w_{jk}^{(i)}} = \frac{\partial z_j^{(i)}}{\partial w_{jk}^{(i)}} \frac{\partial a_j^{(i)}}{\partial z_j^{(i)}} \frac{\partial L}{\partial a_j^{(i)}} = a_k^{(i-1)} g'^{(i)} \left(z_j^{(i)} \right) \frac{\partial L}{\partial a_j^{(i)}}$$

and the partial derivative of each bias is

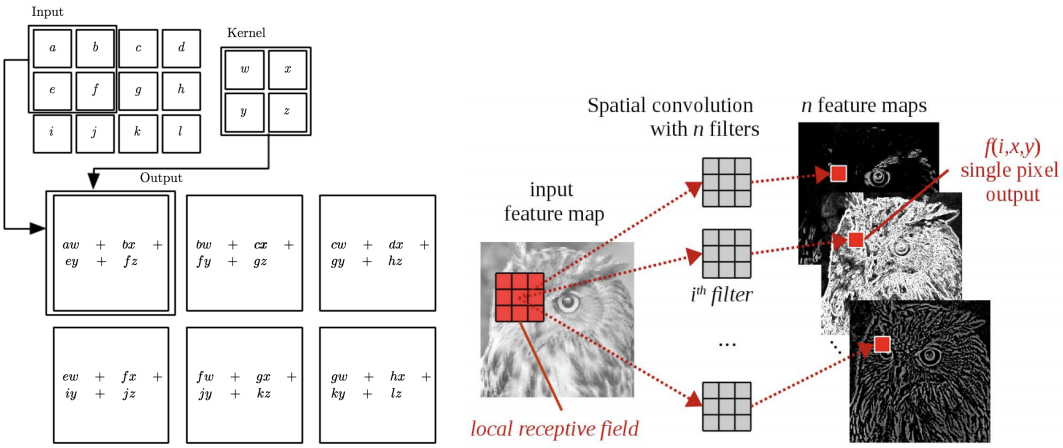
$$\frac{\partial L}{\partial b_j^{(i)}} = \frac{\partial z_j^{(i)}}{\partial b_j^{(i)}} \frac{\partial a_j^{(i)}}{\partial z_j^{(i)}} \frac{\partial L}{\partial a_j^{(i)}} = g'^{(i)} \left(z_j^{(i)} \right) \frac{\partial L}{\partial a_j^{(i)}}$$

where for both cases

$$\frac{\partial L}{\partial a_j^{(i)}} = \sum_{j=0}^{n_i - 1} w_{jk}^{(i+1)} g'^{(i+1)} \left(z_j^{(i+1)} \right) \frac{\partial L}{\partial a_j^{(i+1)}}.$$

if $\boldsymbol{a}^{(i)}$ is not the output-layer. Once the partial derivative of all weights and biases has been found, the gradient vector can be formed and an optimization method can be used to optimize the parameters of the model [18].

4.4.2 Convolutional Neural Networks



(a) Example of a 2D convolution [4]

(b) Example of a convolution with multiple filters [15]

Figure 2: Convolutions visualized

Feedforward neural networks introduced in 4.4.1 can be used for pattern recognition within images, however, they are usually not used for this task. Consider a colored input image of dimension 64×64. If we were to use a feedforward neural network on this image, each neuron in the first hidden layer would be connected to the input layer through 12.288 weights. Not only would this use a lot of computational power and time to train, however, a network of this size would also be prone to overfitting. Instead, *convolutional neural networks* (also known as CNN's) are usually used. A CNN usually consists of *convolutional layers*, *pooling layers* and *fully-connected layers*, where the fully-connected layers are analogous to the layers in a feedforward neural network [12].

Convolutional layers

A convolutional layer is composed of a set of *kernels* (also known as *filters*), which are matrices of weights of dimension $k \times k$, where k usually is 5, 3 or 1, and each weight is a parameter

for the model to be learned [15]. Each kernel is used on the input to produce a *feature map*. The kernels are applied to the input by "sliding" over the input (where the step size is called *stride* and is usually by default equal to 1). Each $k \times k$ grid of the input (called the *local receptive field*) is then used to compute the dot-product between the grid and each kernel, which is then placed in the corresponding feature map of each kernel, as visualized in Figure 2. When all of the feature maps have been computed, the feature maps are stacked together, forming a tensor, which is then returned by the layer and a activation function can be applied.
As described previously, by using a convolutional layer we can dramatically decrease the amount of weights used by the layer. If we were to use a 3×3 kernel on a colored image, we would reduce the amount of weights on each neuron in the following layer from 12.288 down to just 27, reducing both the training time and making the network less prone to overfitting [12].

Pooling layers
Pooling layers are layers used to reduce the dimension of the input. The most common pooling layer is the *maxpooling*-operation. The operation works by considering each $k \times k$ grid, like in the case with the convolutional layer, in which the maximum entry in that grid is being inserted into the output [12].

Nearest Neighbour Upsampling
Sometimes we want to increase the size of an image. This is done by making use of *upsampling*

Algorithm 5 Nearest Neighbour Upsampling [16]

Require: Input image X of size $m \times n$
Require: Wanted output size $xm \times yn$
1: Create empty image O of size $xm \times yn$
2: **for each** pixel, p, in X **do**
3: $\quad i, j =$ index of p in X
4: $\quad$ Insert p at index (xi, yj) in O
5: **for each** empty pixel $p \in O$ **do**
6: $\quad$ Let p be the value of the nearest neighbour
7: **return** O

(also known as *interpolation*) techniques. One of the most common upsampling techniques is *nearest neighbour upsampling*, whose pseudocode has been written in 5. The algorithm starts off by taking an image, X, of size $m \times n$, as input, which we wish to upsample to size $xm \times yn$. The algorithm then loops over each pixel, p, in X, finds the corresponding index, i, j, of p in X, and places p at index (xi, yj) in the output image O of size $xm \times yn$. When this is done, it assigns each of the empty pixel in O the value of their nearest neighbour, making each pixel in O have a value, and then returns O [16].

5 Autoencoders, Stacked Hourglass, PCA and K-Means

In this section the various algorithms and architectures used throughout this thesis is described and explained in details. The section starts off with Section 5.1, where we will be giving a brief introduction to autoencoders. Throughout Section 5.2 the Stacked hourglass, used for pose estimation, is described. Then, in Section 5.3 the algorithm of Principal Components Analysis is described. In the last section, Section 5.4, K-Means clustering is described.

5.1 Autoencoders

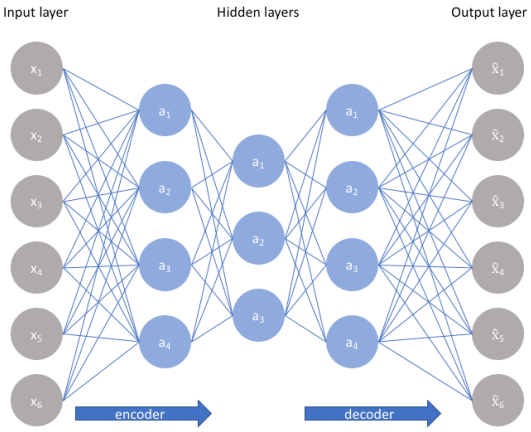


Figure 3: Visualization of an undercomplete autoencoder [9]

Autoencoders is a class of neural networks that are trained to output the input of the model. An autoencoder consists of two parts: the *encoder* and the *decoder*, where each part usually consist of multiple hidden layers. When data is fed to the autoencoder, the data is passed through the encoder, which then passes the data to the decoder, where the data is again processed and finally returned [4].

If all of the layers of the autoencoder have the same dimensionality, the network can easily learn how to copy the input to its output. For this reason we often talk about *undercomplete* autoencoders instead, where the dimension of the output of the encoder is smaller than the dimension of the input and output of the network, as visualized in Figure 3. By making use of an undercomplete autoencoder, the encoder learns how to encode the input to a lower dimensionality, forcing the network to learn the most important features of the training data [4].

To make the autoencoder more robust in relation to small variances, some noise is usually added to the input of the data during training. The noisy training data is then passed through the network and compared to the non-noisy training data, when the loss is computed [4].

5.2 Stacked Hourglass

When performing the pose estimation in section 7, we will be implementing and using the *Stacked hourglass* described by Newell *et al.* [11]. The following description and explanation of the architecture is based on an interpretation of Newell *et al.* [11] and Camilla Olsen [13].

5.2.1 Motivation behind using the Stacked Hourglass

We have decided to make use of the Stacked hourglass described by Newell *et al.*, as it is an architecture that has shown state-of-the-art results. At the same time the architecture of

the network is similar to the architecture of autoencoders, making the architecture useful for encoding the data into a lower dimension, which can be useful in section 8, when we will be doing the interpretation of the model.

5.2.2 The Residual Module

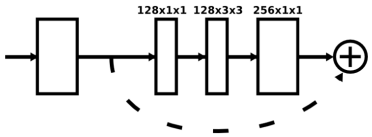


Figure 4: Visualization of the residual module [11]

The Stacked hourglass makes heavily use of so-called *residual modules*, one of which is visualized in Figure 4. The module works by taking an input, which is sent through a 1×1 and a 3×3 convolution, each with 128 channels. Then, the 128 output featuremaps are sent through a 1×1 convolution with 256 channels. Lastly, element-wise addition is then used to add the 256 output featuremaps to the input of the module, which the module then returns. All convolutions are followed by an acitvation function and are *same convolutions*, meaning the output featuremaps are of the same dimensions as the input featuremeaps.

5.2.3 The Hourglass

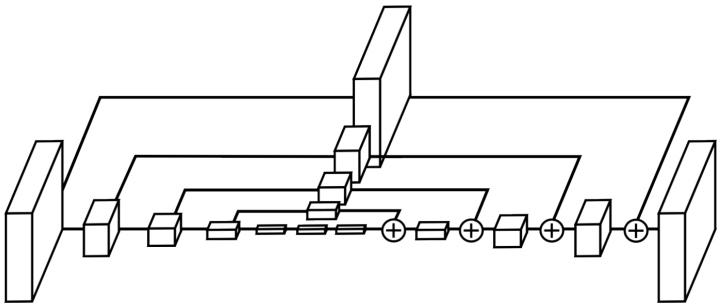


Figure 5: Visualization of a single hourglass [11]

The Stacked hourglass consists of hourglasses, where each hourglass is split into an encoder, where the featuremaps is downsampeld, and a decoder, where the featuremaps are upsampled. The hourglass is symmetric, in the sense, that it has an equal amount of downsampling layers in the encoder as there are upsampling layers in the decoder. In Figure 5 a single hourglass han been visualized, where each box is a residual module.

The hourglass works by using residuals and max poolings to process features down to a low resolution. Then, nearest neighbor upsampling is used to upsample the featuremaps until the featuremaps have the same dimensions as the input of the hourglass. Before each max pool in the encoder, the network branches off and applies a residual. The output of this residual is then added back element-wise to the corresponding level in the decoder, which helps to ensure that lost information from the encoder is kept. This is then fed into a residual in the decoder.

Between the encoder and decoder the network has a bottleneck, where no downsampling or upsampling happens, instead only residuals are processing the featuremaps. After the decoder two 1×1 convolution layers er applied to produce the final predictions of the network.

5.2.4 The Stacked Hourglass

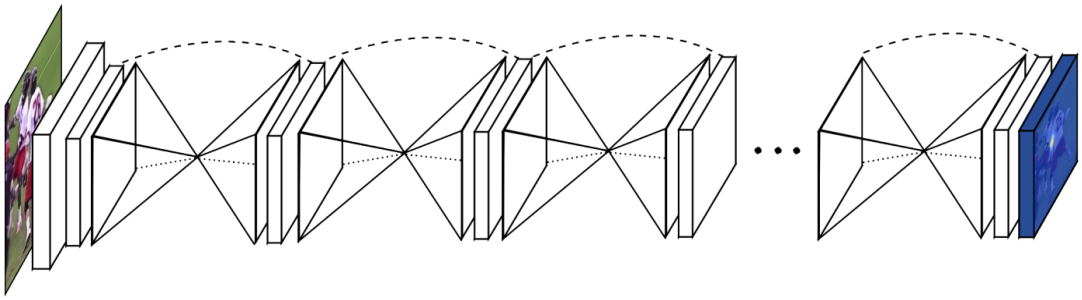


Figure 6: Visualization of the Stacked hourglass [11]

The full network is build by stacking multiple hourglasses end-to-end, making the output of one hourglass be the input of the next hourglass, as shown in Figure 6, which makes each hourglass reevaluate estimates. To evaluate each hourglass, intermediate supervision is used by applying a loss to each hourglass' intermediate prediction.

The input of the network is a 256×256 RGB-image. To lower the memory usage, the network starts off with a 7×7 convolution layer with stride 2, followed by a residual module and max pooling to bring the resolution down to 64×64, which is then input to the first hourglass.

By the end the whole network outputs n heatmaps corresponding to the n joints it should predict for a single person. The prediction of a joint is thus the maximum activation of the corresponding heatmap.

5.3 Principal Components Analysis (PCA)

Algorithm 6 PCA [17]

Require: Input data $\boldsymbol{Y} \in \mathbb{R}^{N \times D}$, with feature columns $\boldsymbol{y}_1, ..., \boldsymbol{y}_N$.
Require: Wanted output dimensions k

1: Let each feature column have zero mean by subtracting the corresponding mean, $\bar{y} = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{y}_n$, from each feature column
2: Compute the sample covariance matrix $\boldsymbol{C} = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{y}_n \boldsymbol{y}_n^T$
3: Find the D eigenvector/eigenvalue pairs of the covariance matrix
4: Find the eigenvectors, $\boldsymbol{w}_1, ..., \boldsymbol{w}_k \in \mathbb{R}^D$, corresponding to the k highest eigenvalues
5: Let $\boldsymbol{W} = [\boldsymbol{w}_1, ..., \boldsymbol{w}_k]$, that is, the $D \times k$ matrix created by placing the k eigenvectors alongside one another
6: Let $\boldsymbol{X} = \boldsymbol{Y}\boldsymbol{W}$ be the projection of Y down to k dimensions
7: **return** $\boldsymbol{X}$

It is very common, that a given dataset has an enormous amount of dimensions. This is quickly become a problem, as it can be difficult to visualize or it can lead to other problems, such as models becoming too complex, thus being prone to overfitting. This is a common phenomen called the *Curse of Dimensionality* [5]. For this reason multiple techniques have been developed for reducing the dimensions of a given dataset. We have already seen in Section 5.1, how autoencoders can be used as non-linear dimensionality reduction. Another very common technique for reducing the dimension of a dataset is *Principal Components Analysis* (*PCA*).

PCA is an unsupervised linear projection method used for reducing the dimension of a dataset

from D down to k dimensions. The algorithm works by finding the k orthogonal vectors that maximizes the variance of the input data. [17]

The pseudocode of the algorithm has been visualized in Algorithm 6. The algorithm starts off by finding the sample covariance matrix $\boldsymbol{C}$. It then finds the k vectors, that maximizes the variance of the input data. The k vectors that maximizes the variance are the k eigenvectors with the corresponding highest eigenvalues. Thus, the projection with the first eigenvector captures the most variance, the projection with the second eigenvector captures the second most variance, and so on. The projection down to k dimensions then happens by stacking the k eigenvectors, forming a $D \times k$ matrix, where D is the input dimensions, which is then multiplied with the input data, resulting in the projected data. [17]

5.4 K-Means Clustering

Often we want to find patterns in data that has not been labelled. One common group of techniques for this purpose is the *clustering algorithms*, that are used to group observations into clusters, such that observations from the same cluster are more similar, than observations from different clusters. One common clustering technique is *K-Means*.

K-Means is a unsupervised method used for clustering observations into K groups of similar observations, such that no observation occurs in multiple clusters. The algorithm uses distance as a measure of similarity, such that observations closer to each other are more likely to being grouped to the same cluster, than two observations far appart. In the middle of each cluster is a synthetic observation (that is, not a real observation), called the *centroid*, which is defined as the mean of the cluster. The pseudocode of the algorithm has been visualized in Algorithm 7. The algorithm is an iterative process, which works firstly by assigning each observation to the closest centroid. Next, each centroid is updated accordingly. This is done until the assigning of each observation is unchanged [17].

K-Means is guaranteed to converge to a local minimum of the total distance between the objects and their corresponding centroid, however, it is not guaranteed to reach the global minimum. This only depends on the initial position of the centroids. To partly overcome this problem it is common to run the algorithm multiple times with different random initial positions of the centroids and use the best solution as the final output [17].

Algorithm 7 K-Means [21]

Require: Input data $\boldsymbol{X} \in \mathbb{R}^{n \times m}$
Require: Amount of clusters k
1: Select k points as initial cluster centers $\boldsymbol{C}_1, ..., \boldsymbol{C}_k$
2: **while** not converged **do**
3: **for** $1 \leq 1 \leq n$ **do**
4: Map point p_i to its nearest cluster center $\boldsymbol{C}_j$
5: **for** $1 \leq j \leq k$ **do**
6: Compute centroid $\boldsymbol{C}'_j$ of the points nearest $\boldsymbol{C}_j$
7: **for** $1 \leq j \leq k$ **do**
8: Set $\boldsymbol{C}_j = \boldsymbol{C}'_j$

6 The Dataset

To perform the pose estimation, we need some data on which to train, validate and test our model. Throughout this section the used data and the relevant preprocessing is described. The section consists of two parts: Section 6.1, where a brief overview of the dataset is described, and Section 6.2, where the preprocessing of the data is described.

6.1 The COCO Dataset

Notice how the image contains multiple people, each with their own keypoints and amount of joints labeled

Figure 7: Example of an image from the COCO dataset with the keypoints drawn on [10]

The data needed for our model has to fit to our problem and has to be annotated, as our model will perform supervised learning. There are multiple datasets that fits these requirements. One of these datasets is the Common Objects in Context (COCO) dataset [10], which we will be using. The dataset contains annotations for different purposes, however, for our pose-estimation-task, only the keypoint annotations of human bodies are needed. An example of such a picture with the keypoints labeled can be seen in Figure 7.

The annotation of each person consists of an array with a length of 51, which annotates 17 keypoints of a person. Thus, each joint corresponds to three sequential elements in the array, where the first and second indices corresponds to the x and y-location of the joint in the image, and the third index is a flag, v, indicating the visibility of the joint in the image. v has three outcomes: if $v = 0$, the joint is not labeled, if $v = 1$, the joint is labeled but not visible, and if $v = 2$, the joint is visible and labeled.

The creators of the dataset has already split the data into three parts: a part used for training the model, a part used for validating the model and a part used for testing the model. However, the part used for testing the model is unlabel, hence, it is unusable for our purpose, as our model will be doing supervised learning. As both the training dataset and the validation dataset will be used for training and tuning the model, we will need to create our own hold-out dataset for testing to provide an unbiased evaluation of the final model.

The training and validation sets contains a total of about 123.000 various images. As we only need the images that contain humans, we will be discarding the images without any humans, leaving us with a total of about 66.808 images of humans doing various tasks, with a total of 149.813 humans annotated with keypoints. Each image can contain multiple people, which we need to handle before training our model, as we will be focusing on single-human pose estimation. Besides this, each image also has different resolution and aspect ratio, which we also need to handle, as our model requires the images to have a fixed resolution. Lastly, we

should also do some handling of the labels before training the model, as there could have been some inaccuracies, when the joints were labeled. This especially applies when $v = 1$, that is, when the joint is labeled but not visible, as there are more inaccuracies or uncertainty when labeling a non-visible joint than when labeling a visible joint.

6.2 Data Preprocessing

6.2.1 Creating the test dataset

To create the dataset which will be used for testing we take the training set, since it is the larger of the training and validation set, and sample 5.064 images randomly without replacement, to create a test set. This ensures that the test-set and validation-set are of the same size. This new test set will not be used when training the model nor used when tuning the parameters. Instead, it will only be used to evaluate the very final model.

6.2.2 Preprocessing the images

	Amount of images	Percentage
Training set	124.040	92,45
Validation set	5.064	3,77
Testing set	5.064	3,77
Total	**134.168**	**100**

Table 1: Data distribution

Figure 8: The results of processing the image from Figure 7 with the corresponding labels [10]

We start the preprocessing of the images by creating multiple bounding boxes, where each bounding box surrounds a single person, which is done by making use of the boundig box annotations provided by COCO. Then each bounding box is transformed into a square by making the shorter sides have the same length as the longer sides - this is done to ensure that the aspect ratio of the image is kept, when it is later resized.

It is possible for each bounding box to contain multiple people. This is a problem, as it will confuse our model, since it will not know which person to annotate. An example of this can be seen in the first image of Figure 8. To fix this we center the bounding box around the person it should annotate, making the model annotate the person in the center of the input image, which is done by centering the bounding box with respect to the outermost keypoints of the person.

Since each keypoint does not necessarily lie on the edge of the person, the bounding boxes could result in not all of the pixels of the corresponding person being in the bounding box. For this reason, each bounding box is expanded with 10% in the height and width. If, however, the image cannot contain the expanded bounding box, the bounding box is then expanded as much as possible, while still being a square. If it is the case, that one of the corners of the

bounding box lies outside of the image, then the bounding box is moved either up or down, making the corner of the bounding box be inside the image and keeping the annotated person centered along the x-axis.

When all of the above is done, the image is finally croped to each bounding box, resulting in multiple squared images, each containing an unique person at the center. Each of these squared images are then resized to a 256×256 image. We then center the rgb-values of each image by subtracting the mean rgb of all of the images from the training set from each image. Then, each images is saved as an `.png`.

By doing the data preprocessing as described above, we get the distribution of images displayed in Table 1. In Figure 8, the results of processing the image from Figure 7 are shown with the corresponding labels. Lastly, the data is shuffled to help the developed model generalize the data better.

6.2.3 Handling the labels

Left: The original image. Right: The heatmaps of all the keypoints, fused together to a single image.

Figure 9: An example of the heatmaps of a single image fused together and put over the original image [2]

For each image our model outputs 17 heatmaps, one for each possible joint in the image. An example of such heatmaps fused togehter can be seen in Figure 9.
The ground truth heatmap of a single joint is created firstly by initializing an all-zero 2D array with size 64×64. Next, at coordinate $\left(x \cdot \frac{64}{256}, y \cdot \frac{64}{256}\right)$, where (x, y) is the annotated position of the joint in the original image, a 1 is places, representing the position of the corresponding joint in the output image - by placing the 1 in a heatmap of size 64×64 instead of in a heatmap of size 256×256, which is later resize to 64×64, we ensure, that the 1 is not lost once resized. Next, a Gaussian filter is used to smear out the heatmap, where the standard deviation of the Gaussian filter depends on the visibility, v, of the joint: if the joint is visible, that is if $v = 2$, then the standard deviation is set to be 0.5, whereas the standard deviation of the Gaussian filter is set to be 1 if the joint is not visible, as there there are more uncertainty with the labeling of such keypoints. We do all of this for all of the 17 joints for each image, resulting in the keypoints which will be used for developing our model.

7 Experiment

Throughout the following section the results and configuration details of our trained model are described and explained. In the first section, Section 7.1, the configuration details of the model, as well as the training of the model, are described. In the second section, Section 7.2, the results of training the Stacked hourglass is presented, as well as a short discussion of which version of the model which will be used going forward. In the last section, Section 7.3, we give an overview of the technical details behind training the network, which can be followed in case the reader has any technical problems in case of testing for reproducibility.

7.1 Configuration Details

Our model only consists of a single hourglass. The hourglass consists of 4 down- and upsamples, with 1 residual module between each down- and upsample. In each skip-connection a single residual module is used, and in the bottleneck 3 residual modules are used. Newell *et al.* [11] and Olsen [13] experiment with different amount of hourglasses and with different amount of residual modules in each hourglass. They both come to the conclusion, that stacking multiple hourglasses or using hourglasses with multiple residual modules between each down- and upsample increases the performance of the model. However, as the main purpose of this section is not to create a model with state-of-the-art results, but instead to create a model that can be interpreted and explored, we have chosen to reduce the size of the model. For the same reasons, we will not be developing and testing various configurations of the architecture. Likewise, the purpose of this section is neither to improve the model developed by Newell *et. al* [11], hence why we will be making the same configuration choices as Newell *et. al* [11] and Olsen [13], which are described in the following.

To prevent the model from overfitting we use batch normalization. Newell *et al.* does not describe where to perform the batch normalization, so we follow Olsen [13] and perform the batch normalization before each convolutional layer in each residual module, after the first convolutional layer of the entire network and before the last convolutional layer of the entire network. For the choice of acitvation function we use the $ReLU$-function after each batch normalization. Each max pooling and nearest neighbor upsampling uses a kernel size of 2, which halves and doubles the size of the input, respectively. The full network has been visualized in Figure 10.

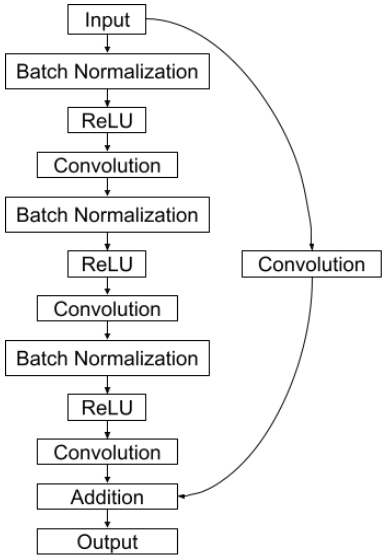
For the initial values of the weights we initialize each weight by sampling from a *Glorot normal distribution* (also known as a *Xavier normal distribution*), described as

$$\mathcal{N}\left(0, \frac{2}{fan_{in} + fan_{out}}\right)$$

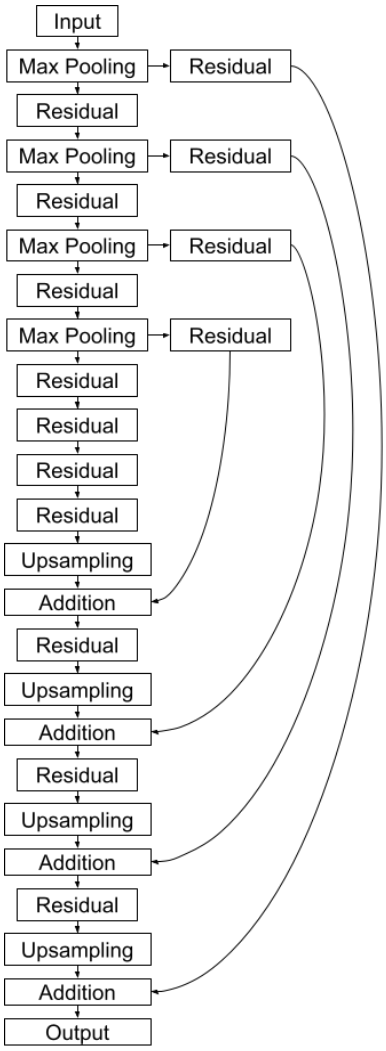
where fan_{in} is the amount of input connections and fan_{out} is the amount of output connections to the layer of the weight [3]. By doing so we make all layers have the same activation variance and gradient variance, essentially helping the model to converge [4].

We make use of a mini-batch size of 16 and no data augmentation, since the dataset is already rather large and captures a lot of the variances. To optimize the network we make use of MSE as our loss function, *RMSPROP* as our optimizer, as well as use a initial learning rate of $2.5e-4$.

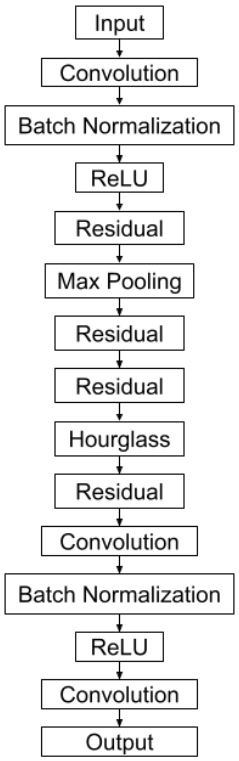
After each epoch we find the validation accuracy of the model by computing the *PCK* between the predictions and the ground truth of the validation dataset, as described by Olsen [13] and the source code of the Stacked Hourglass Network [22], where it is called *heatmap accuracy*. The pseudocode of PCK has been visualized in Algorithm 8. The algorithm works by iterat-



(a) The Residual module



(b) The Hourglass



(c) The Full network

Figure 10: Overview of the used architecture

Algorithm 8 PCK [13][22]

Require: Ground truth heatmaps $heatmaps_{gt}$ of keypoints
Require: Predicted heatmaps $heatmaps_{pred}$ of keypoints
Require: Threshold radius r
Require: Normalization constant c
Let $n = 0$ be the running total of correctly predicted keypoints
Let N be the amount of annotated heatmaps
for each annotated ground truth heatmap, $heatmap_{gt}$, in $heatmaps_{gt}$ **do**
    Let (x_{gt}, y_{gt}) be the $2D$ index of the maximum activation of $heatmap_{gt}$
    Let (x_{pred}, y_{pred}) be the $2D$ index of the maximum activation of the predicted heatmap corresponding to $heatmap_{gt}$
    Let $dist$ be the Euclidean distance between (x_{gt}, y_{gt}) and (x_{pred}, y_{pred}).
    Normalize $dist$: $dist = \frac{dist}{c}$
    if $dist < r$ **then**
        $n = n + 1$
Let $ratio = \frac{n}{N}$ be the ratio of correctly annotated heatmaps
return $ratio$

ing over each annotated ground truth heatmap and the corresponding predicted heatmap. It then finds the Euclidean distance between the maximum activation of a ground truth heatmap and the corresponding predicted heatmap. The distance is then normalized by a constant c and compared to a threshold radius r. The ratio of normalized distances that are less than the threshold r are then computed and returned, yielding the PCK accuracy between the ground truth heatmaps and the corresponding predicted heatmaps [13] [22]. The aim is thus to maximize the PCK accuracy. To produce the final PCK accuracy of the model, the PCK accuracy is computed for each image in the validation dataset. The mean PCK accuracy is then used as the PCK accuracy of the model. For the two constants, c and r, we let c be one tenth of the heatmap resolution size $\left(\text{that is, } \frac{64}{10} = 6.4\right)$ and r be 0.5.

While training the model the PCK accuracy of the model is computed after each epoch, keeping track of the best PCK accuracy. The first time the best PCK accuracy has not improved for 5 continuous epochs, the learning rate is dropped by a factor of 5 permanently, helping the training loss reach a minimum.

7.2 Results

In Figure 11 the evolution of the training loss, validation loss and validation PCK accuracy has been visualized. The model were initially set to train for 100 epochs, however, we decided to stop the training early, as the model clearly started to overfit after 32 epochs, as seen by comparing the training and validation loss, as well as the PCK validation accuracy seemed to have converged.

The reduction of the learning rate happened after 21 epochs. By looking at the validation accuracy in Figure 11 we can see, that the accuracy rapidly increases shortly after the reduction of the learning rate, hinting at the effectiveness of dropping the learning rate.

Comparing the training loss, validation loss and the validation accuracy from Table 2 we see, that the there is not an overlap between the models yielding the best training loss, validation loss and validation accuracy. As we in section 8 want to explore a model that performs decently well, we will be using the model with the highest validation accuracy as our model going for-

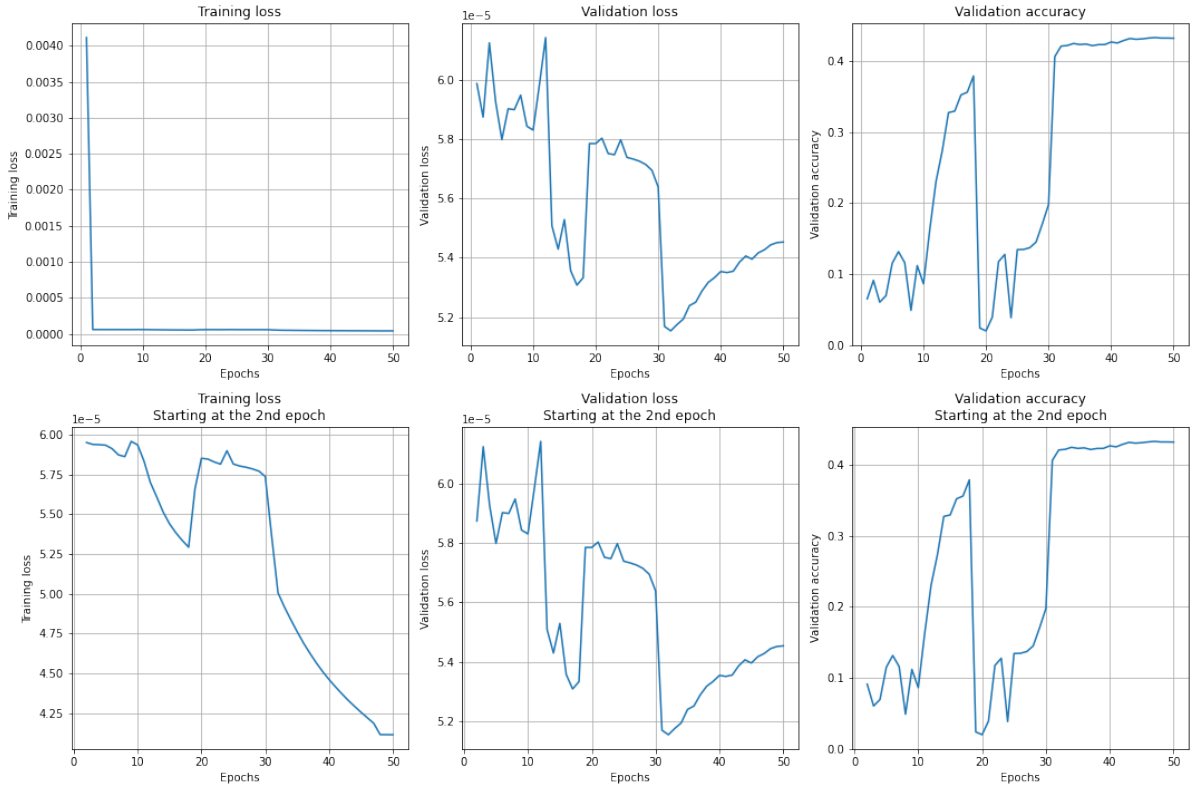


Figure 11: Visualization of the evolution of the training loss, validation loss and validation PCK accuracy of the trained model. Top row shows all of the 50 epochs. Bottom row shows epoch 2 and forward to ease the reading of the training loss

ward. Thus, our model is the model from epoch 47, which has a training loss of $4.19 \cdot 10^{-5}$, a validation loss of $5.43 \cdot 10^{-5}$ and a validation accuracy of 0.433.

7.3 Training Details

The stacked hourglass was implemented in Python 3.8.2 using PyTorch version 1.7.1 and Cuda version 10.2 on a machine using Windows 10 version 20H2, build 19042. The network was trained on an 8 GB NVIDIA GeForce GTX 1070 GPU using a Samsung 840 EVO SSD for data storage. Training the network takes about 70 minutes per epoch, totalling to about 58 hours for 50 epochs.

Description	**Epoch**	**Training loss**	**Validation loss**	**Validation accuracy**
Best training loss	50	$4.11 \cdot 10^{-5}$	$5.45 \cdot 10^{-5}$	0.43
Best validation loss	32	$5.01 \cdot 10^{-5}$	$5.15 \cdot 10^{-5}$	0.42
Best validation accuracy	47	$4.19 \cdot 10^{-5}$	$5.43 \cdot 10^{-5}$	0.433

Table 2: Comparison of the the epochs yielding the best training loss, validation loss and validation accuracy

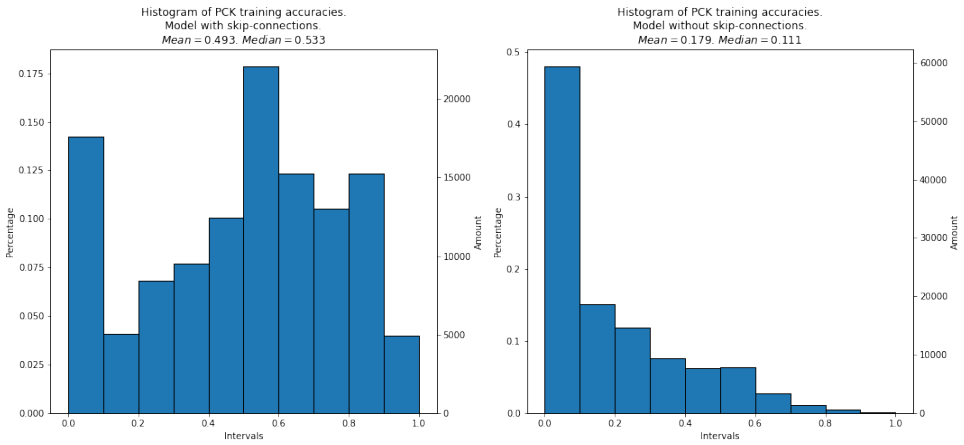


Figure 12: Histogram of PCK training accuracies of the model with the skip-connections enabled (left) and the model with the skip connections disabled (right).

8 Interpreting the Model

In the following section we will be interpreting the model developed in section 7, with the intention of getting an understanding of what the model has learned during training, what the different parts of the model are used for, as well as checking for any redundancy in the model. In Section 8.1 an overview of the motivation behind interpreting the model is given. Section 8.2 then evaluates the effects of the skip-connections of the model. Then, Section 8.3 explores the latent space of the model with respect to getting an understanding of the principal components of the latent space. Section 8.4 also explores the latent space of the model, however, instead it uses clustering to seperate the latent space, which the results are the discussed.

8.1 Motivation

Deep learning models are often complex and work like a *black boxes*. By that it is meant, that when a network is given some input, the model simply just returns some output without any explanation or reasoning behind the output. This can often be a problem, especially in cases where the output of the network can result in a life or death situation of a human. For that reason, understanding and explaining how a network works can be very important - this is what is called *explainable AI* (*XAI*)

Selvaraju *et al.* [20] argues that there are three cases for using explainable AI:

1. When the network performs worse than humans, an understanding of the network can help us improve the performance of the model.

2. When the network is on par with humans, an understanding of the network is trivial for humans to build trust in the network, as we can understand its strengths and weaknesses

3. When the network performs better than humans, an understanding of the network can teach humans how to perform better.

Throughout section 8 we will be exploring and understanding what the model, developed in section 7, learned during training and what the different parts of the model are used for, leading towards improving the performance of the model easier.

8.2 Verifying the Effects of Skip-Connections

Olsen [13] and Newell [11] claims, that the skip-connections are used in order to recreate details that are lost during the encoder-phase. Throughout subsection 8.2 we will be verifying or refuting the claim of the effect of the skip-connections. To do so we will be using two models based on the same network:

1. The trained Stacked Hourglass from section 7
2. The trained Stacked Hourglass from section 7, but with the skip-connections disabled.

Thus, the second model has not been retrained and is identical to the first model, however, without its skip-connections.

In Figure 12 the distributions of the PCK training accuracies of the two models have been visualized. We have decided to make use of the training data for computing the PCK accuracies, as we want to look at the data, that the model has been trained on. By looking at the two distributions we can clearly see how to model without its skip-connections performs much worse, than the model with its skip-connections.

To further understand the decrease of accuracy in the case where the skip-connections are disabled, we have in Figure 13 visualized 20 samples from the training dataset, where the model with skip-connections has an 100% PCK accuracy score. Next to each image the ground truth heatmaps, or the prediction by the model with skip-connections, and the prediction by the model without skip-connections has been visualized.

By looking at Figure 13 we can see, that the model without its skip-connections often struggles with smaller joints, such as the eyes, ears or nose, whereas it performs better, however still not always perfect, on bigger joints, such as the shoulders, hips or knees. This is probably due to the fact, that the details of the smaller joints has a bigger chance of being lost by the max pooling layers in the encoder. Without the skip-connections their information is thus lost, resulting in bad predictions. Thus, we can verify Olsen's [13] and Newell's [11] claims, that the skip-connections are used for recreating details lost in the encoder.

8.3 Shape Analysis of the Latent Space

In subsection 5.2.1 we described how we decided to use the Stacked Hourglass for the pose estimation, as it is similar to Autoencoders. This makes the model useful for encoding the data into a lower dimensional representation of the input data. The space of this lower dimensional representation is known as the *latent space* of the model. By exploring the latent space we can get an understanding of what the model has learned during training.

In the following subsection we will be exploring and explaining the most important components of the training data. By doing so we will get an understanding of how each component contributes to a prediction.

We decided to only use fully-annotated observations, as not fully-annotated observations would add some variance to the data, which would confuse the following procedure. We start off by feeding the fully-annotated training data through the encoder of the model and storing the output of the third residual module in the bottleneck. We decided to make use of training data for this, as we wish to look at what features of the training data that the model has learned. Each output of the bottleneck is a $4 \times 4 \times 256$ tensor, which we flattened to a 4.096 vector. Each vector was then stacked, forming a 7.017×4.096 matrix of the latent space.

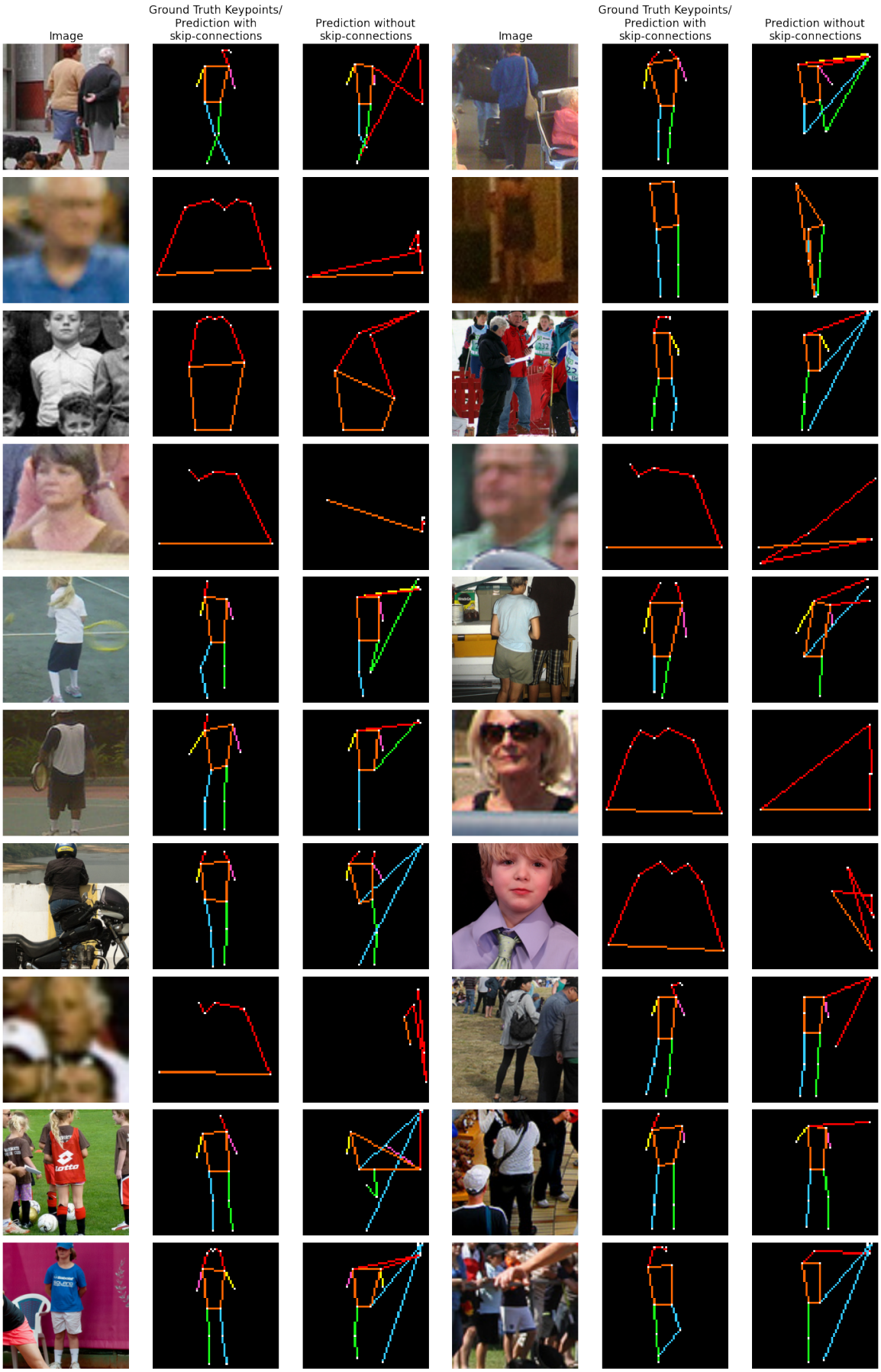


Figure 13: 20 samples of images correctly predicted by the model with skip-connections enabled, the corresponding ground truth heatmap and predictions by the model with skip-connections disabled.

We start off by finding the 4.096 principal components and the corresponding explained variance ratio of the data by using Principal Components Analysis. By doing so we get an understanding of which components for predictions are the most important. Next, the mean coordinate of the principal components, $\bar{\boldsymbol{x}}$ is found. The closest point is then stored. We then explore each principal component by "walking" from $\bar{\boldsymbol{x}}$ along the principal component in positive and negative direction, with various step sizes. After each "walk" the closest point is found and stored. By doing so we can compare the heatmaps of the observations from a "walk" in positive and negative direction, which will give us an idea of what a given component is used for.

When doing the walk along a principal component, we made the step size be equal to $c \cdot \sqrt{\lambda}$, where c is a constant and λ is the explained variance of the principal component. By doing so we ensure, that we are not walking too far, ending up with misleading visualizations as there will be far from the end point to the nearest observation. Some of the results have been visualized in Figure 14.

By looking at Figure 14 we can see, that the first principal component is used for determining if the person is sitting or standing up. This is very clear, as the person sits down if we are "walking" in the negative direction from the mean coordinate, and on the otherhand the person straightens up if we are "walking" in the positive direction from the mean coordinate.

We can also see, that principal component 2, 3, 10 and 30 do not have an easy-to-see pattern, like it is the case with the first principal component. This could probably be becuase they explain a very little amount of the variance in the data, resulting in a very small step size, as well as patterns contributing very little to the prediction of the model.

Lastly, we can see, that principal components 50 and onwards do not have any variations in their corresponding results. This results in them acting as noise, and hints towards how the model could be using many fewer filters in the bottleneck.

All in all, by doing the shape analysis of the latent space of the hourglass, we have learned how the model has learned the difference between people sitting down and standing up (and the poses in between), as well as possibly have identified some redundancy, in the form of the model using more filters in the bottleneck, than what might is needed.

8.4 Using Clustering to Seperate the Latent Space

Similar to subsection 8.3, we will be exploring the latent space of the model to further get an understanding of what the model has learned during training, where we again will be using the training observations. Instead of exploring the principal components of the latent space, we will instead be grouping the training observations to get an understanding of how the model relate similar data to each other.

To create the latent space matrix we follow the same procedure as in 8.3, but instead also use not fully-annotated observations. Due to memory constraints only 10.000 random samples were used, resulting in an 10.000×4.096 matrix of the latent space

If we take this latent space matrix, project it down to 2 dimensions using PCA and visualize the samples with their corresponding ground truth heatmaps, we get the plot visualized in Figure 15. The plot only explains about 37% of the variance of the original data, however, we

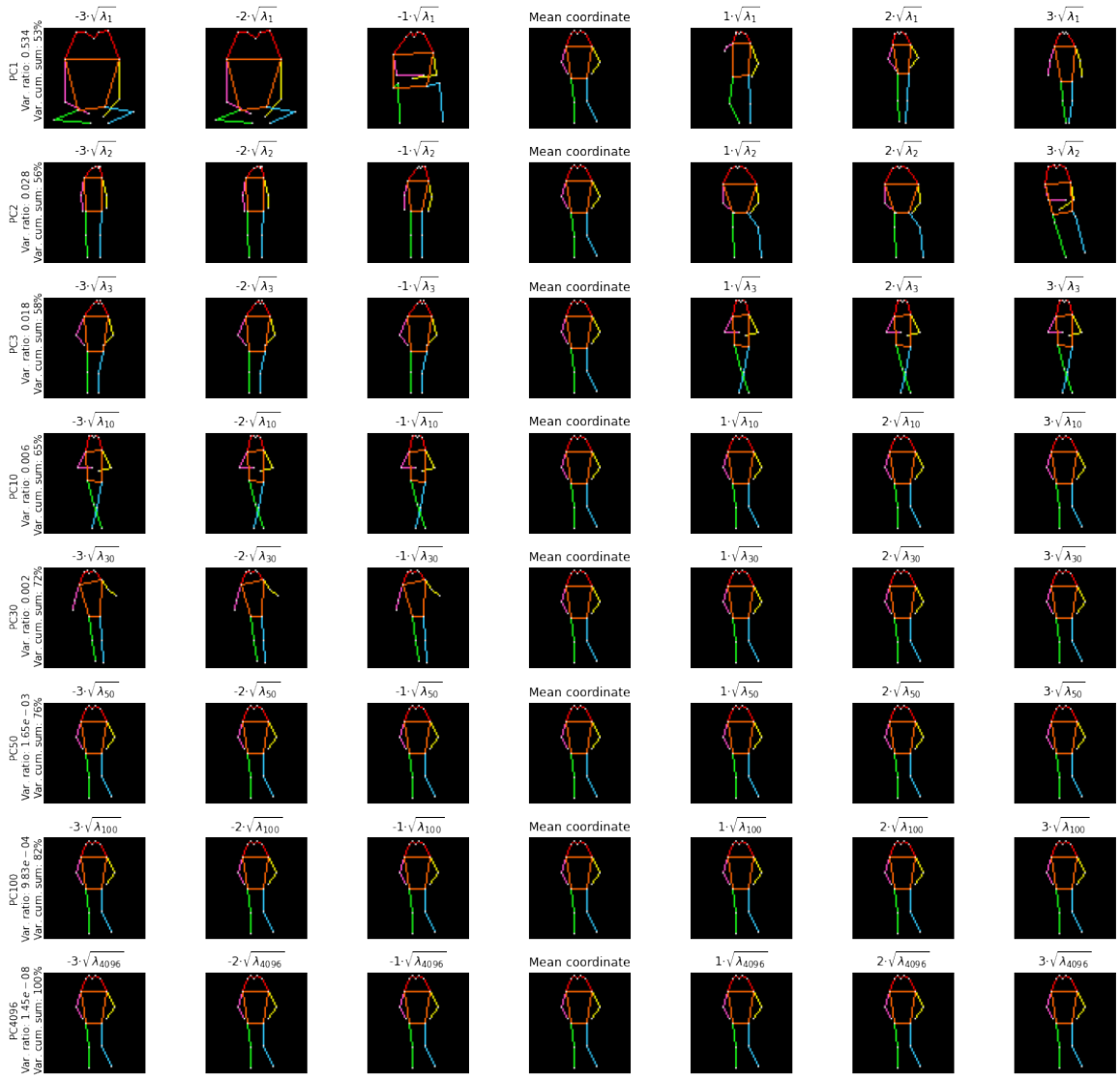


Figure 14: Nearest observations to the end point of "walking" along various principal components with varying step sizes.

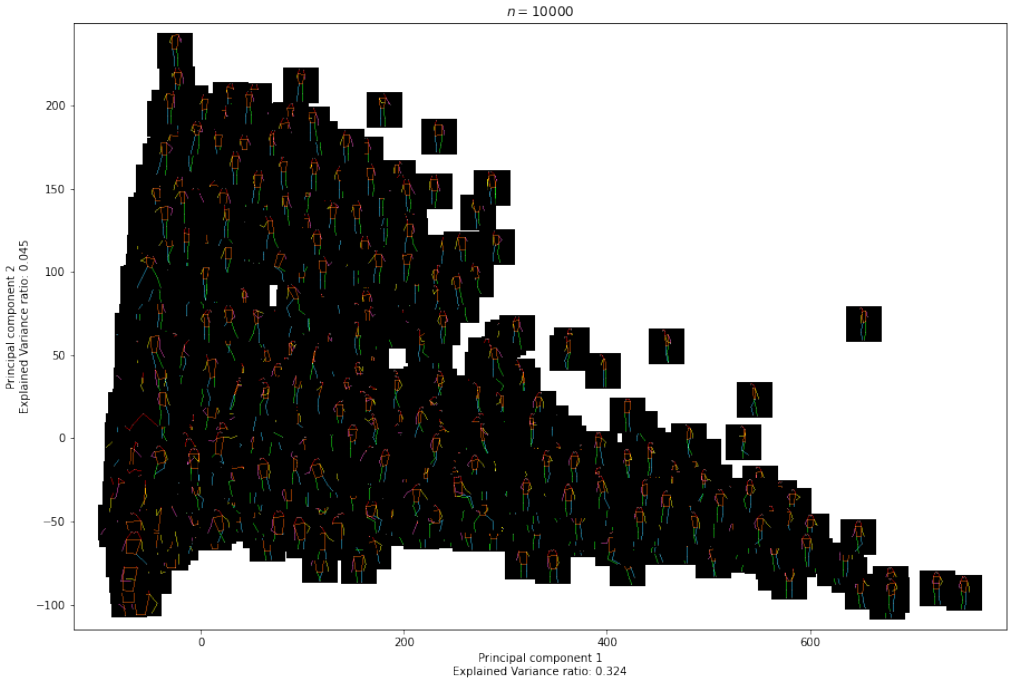


Figure 15: Plot of 10.000 samples of the latent space of the trained model, with the corresponding ground truth heatmaps

can clearly see how there is some specific structure in the data, as samples that are somewhat similar are close to each other, however, with a few outliers.

To see how to model seperates the data in the latent space, we will be using K-Means. Choosing the optimal K can often be difficult, as it is often not clear how many clusters there are in the data. For choosing the optimal k the *Silhouette score* is often computed, following the pseudocode visualized in Algorithm 9. For computing the Silhouette score, various values of K are used for training various K-Means models. After each model has been trained, let a_i be the average distance of the ith sample to the other samples in the same cluster as ith sample. Then, let b_i be the average distance of the ith sample to the samples in the nearest cluster. Ideally, we want $a_i < b_i$, as $b_i < a_i$ means that the ith sample probably has been grouped to the wrong cluster. For that reason, the ith silhouette score is set to $1 - \frac{a_i}{b_i}$ if $a_i < b_i$ or $\frac{b_i}{a_i} - 1$ if $a_i > b_i$. By the end of the algorithm the mean silhouette score is returned. By computing the silhouette score for various values of K, the K with the silhouette score closest to 1 is chosen as the optimal K [6].

When running the K-Means algorithm on the latent space, we use $K = 2, 3, ..., 10$, where the algorithm is retrained 10 times with different initial centroid position for each K. For each run we record the Silhouette score, where the highest Silhouette score for each K has been visualized in Figure 16a. By looking at Figure 16a we can clearly see, how the optimal K for the model is when $K = 2$.

The results of running the K-Means model with $K = 2$ has been visualized in Figure 17. The K-Means model were ran on the latent space in all of the 4.096 dimensions and only each cluster were projected down to 2 dimensions for the purpose of visualization. By looking at Figure

Algorithm 9 Compute Silhouette Score [6]

Require: Clusters $\boldsymbol{C}_0, \boldsymbol{C}_1, ..., \boldsymbol{C}_{k-1}$

1: **for each** cluster $\boldsymbol{C}_i$ **do**

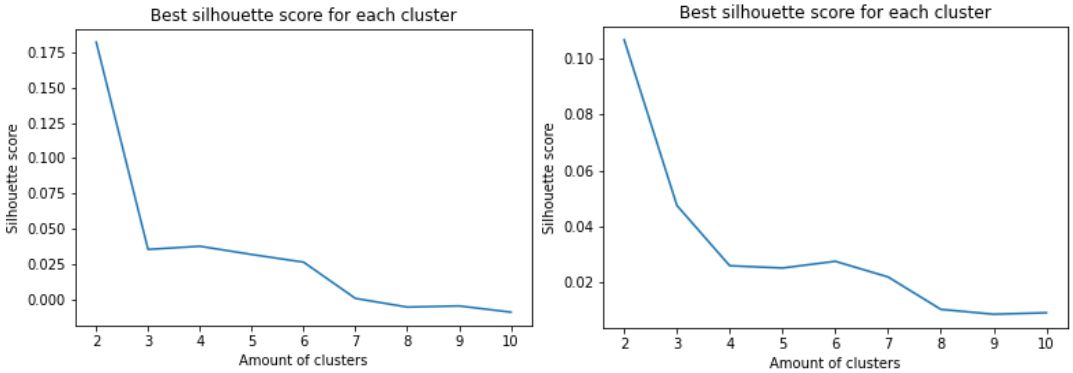
2: **for each** sample $\boldsymbol{x} \in \boldsymbol{C}_i$ **do**

3: Compute the mean euclidean distance from $\boldsymbol{x}$ to the other samples in the same cluster: $a(\boldsymbol{x}) = \frac{1}{|\boldsymbol{C}_i|-1} \sum_{\boldsymbol{y} \in \boldsymbol{C}_i} D(\boldsymbol{x}, \boldsymbol{y})$

4: Compute the mean euclidean distance from $\boldsymbol{x}$ to the nearest other cluster: $\boldsymbol{C}_j$ $b(\boldsymbol{x}) = \frac{1}{|\boldsymbol{C}_j|} \sum_{\boldsymbol{z} \in \boldsymbol{C}_j} D(\boldsymbol{x}, \boldsymbol{z})$

5: Compute the Silhouette of $\boldsymbol{x}$: $s(\boldsymbol{x}) = \begin{cases} 1 - \frac{a(\boldsymbol{x})}{b(\boldsymbol{x})} & \text{if } a(\boldsymbol{x}) < b(\boldsymbol{x}) \\ 0 & \text{if } a(\boldsymbol{x}) = b(\boldsymbol{x}) \text{ or } |\boldsymbol{C}_i| = 1 \\ \frac{b(\boldsymbol{x})}{a(\boldsymbol{x})} - 1 & \text{if } a(\boldsymbol{x}) > b(\boldsymbol{x}) \end{cases}$

6: **return** mean of the Silhouettes



(a) Latent space consists of 10.000 randomly chosen training samples

(b) Latent space consists of 7.017 fully-annotated training samples.

Figure 16: Silhouette score of running various K-Means models on different data from the latent space.

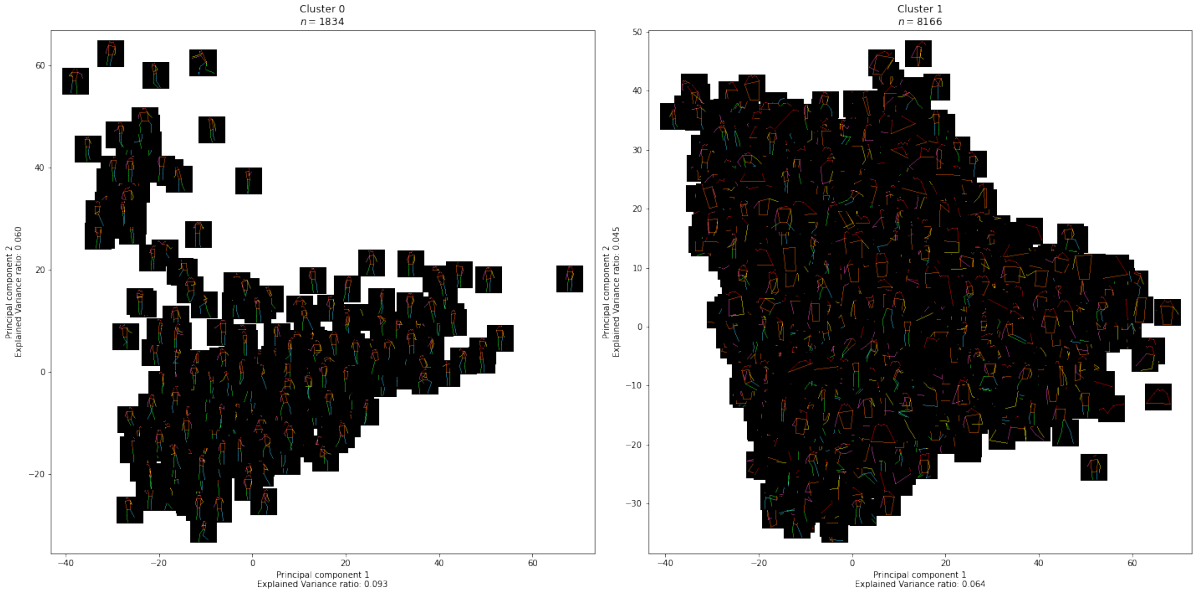


Figure 17: The resulting clusters of running a K-Means model with $K = 2$ on the latent space consisting of 10.000 random training samples

17 we can see how the two clusters has different content: where Cluster 0 focuses more on almost fully-annotated samples, Cluster 1 focuses more on samples that have a lot of keypoints missing. This is also easy to see if we look at the ground truth heatmaps of the samples closest to the centroids of the two clusters, as visualized in Figure 18. By doing so we can see, that the ground truth heatmap of the closest sample to the centroid of Cluster 0 almost has all of its joints annotated, whereas the ground truth heatmap of the closest sample to the centroid of Cluster 1 only consists of 2 keypoints.

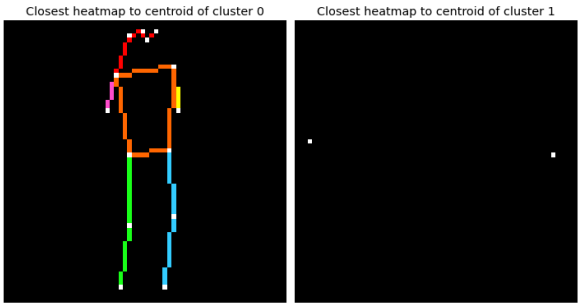


Figure 18: Closest points to the centroids of the two clusters from running K-Means on the latent space consisting of 10.000 random training samples

Although there are differences in the two clusters in Figure 17, there are still quite a lot of missclassified samples. To overcome this problem we remove all of the not fully-annotated samples and instead use all of the 7.017 fully-annotated samples of the training set, again fed through the network and outputted by the third residual module in the bottleneck. By doing so we get the Silhouette scores visualized in Figure 16b, where we again clearly see, that $K = 2$ is the optimal value of K.

The two clusters, resulted by only using fully-annotated samples, have been visualized in Figure 19 and the corresponding closest ground truth heatmaps for the samples closest to the centroids have been visualized in Figure 20. Like before, the K-Means model were ran on the data in full dimension to create the two clusters, which then were projected down to 2 dimensions using PCA for the purpose of visualization. By looking at the figure we clearly see how the content of Cluster 0 contains samples that are stationary, whereas the samples of Cluster 1 carry a lot more movement. This is also the case for the ground truth heatmaps of the samples closest to the centroids, visualized in Figure 20, as we can see, that the heatmap for Cluster 0 is more straighten, whereas the heatmapf for Cluster 1 is more bent and looks like it is in more movement. The two clusters does have a lot less missclassifications, than it was the case with the two clusters in Figure 17. The missclassifications could explain the not-optimal performance of the model as this could mean, that the network has not fully learned the differences between certain positions and where the positions should be placed in the latent space.

All in all we have learned, that model, during training, has learned to distinguish between almost fully-annotated people and not fully-annotated people, as well as learned to distinguish between stationary people and people with a lot of motion. However, the distinguishments are not perfect, as there are some missclassifications, hinting towards the inaccuracies of the model.

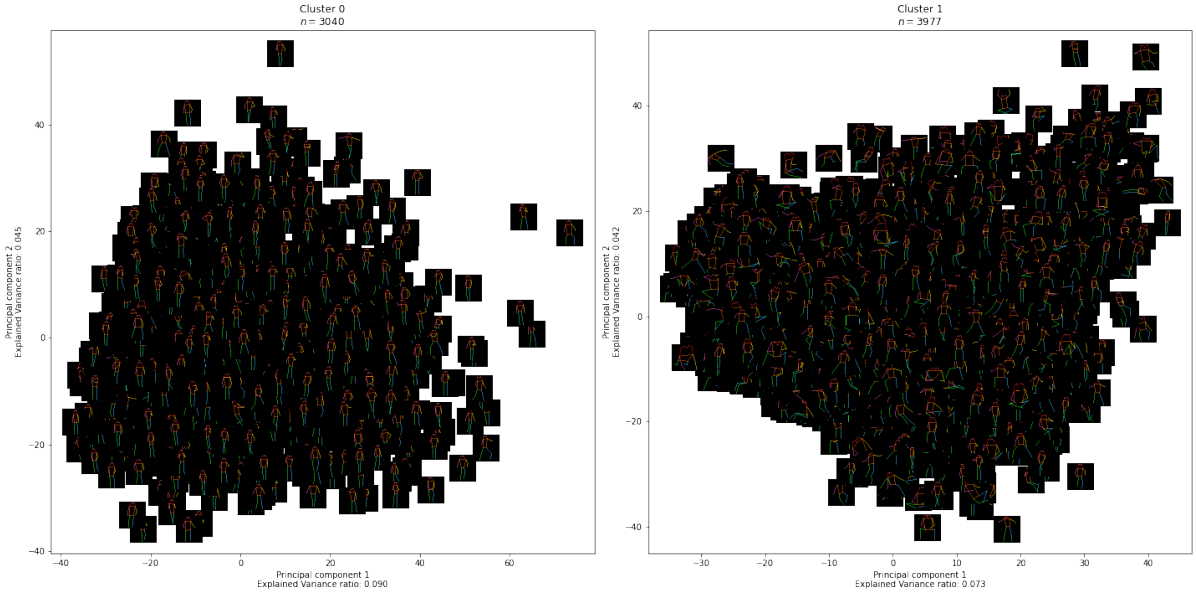


Figure 19: The resulting clusters of running a K-Means model with $K = 2$ on the latent space consisting of 7.017 fully-annotated training samples

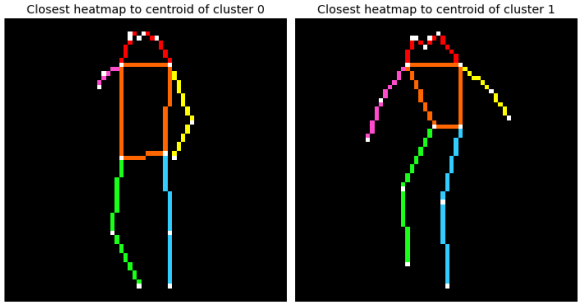


Figure 20: Closest points to the centroids of the two clusters from running K-Means on the latent space consisting of 7.017 fully-annotated training samples

9 Improving the Model

In the following section we will use our knowledge about the developed model to modifie and improve the performance of the stacked hourglass. We will start off in Section 9.1, where we will be giving a brief overview of how the model will be improved. Then, in Section 9.2 the configuration details of the model will be explained and argued for. Lastly, in Section 9.3 the various training details will be described in case of reproduction.

9.1 Motivation

In Section 8 we explored the developed model from Section 7. By doing so we found out, that the latent space of the model has some inconsistency of the placement of the training observations, as well as having some principal components that acts as noise, which could explain the not optimal performance of the model. By helping improving the latent space of the model, as well as removing some of the noise, we could improve the performance of the model, which can be done by making use of an autoencoder.

9.2 Configuration Details

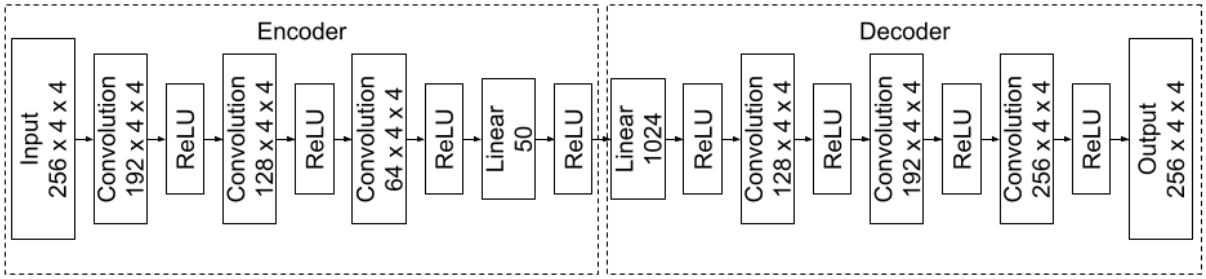


Figure 21: Visualization of the architecture of the developed autoencoder. The numbers in each box represents the dimensions of the output of the corresponding layer.

The developed autoencoder has been visualized in Figure 21. The autoencoder makes use of convolutional layers and linear layers to downsample the input down to only 50 dimensions. We chose 50 as the dimensions of the bottleneck, as we saw in Section 8.3, that principal component 50 and above acts as noise.

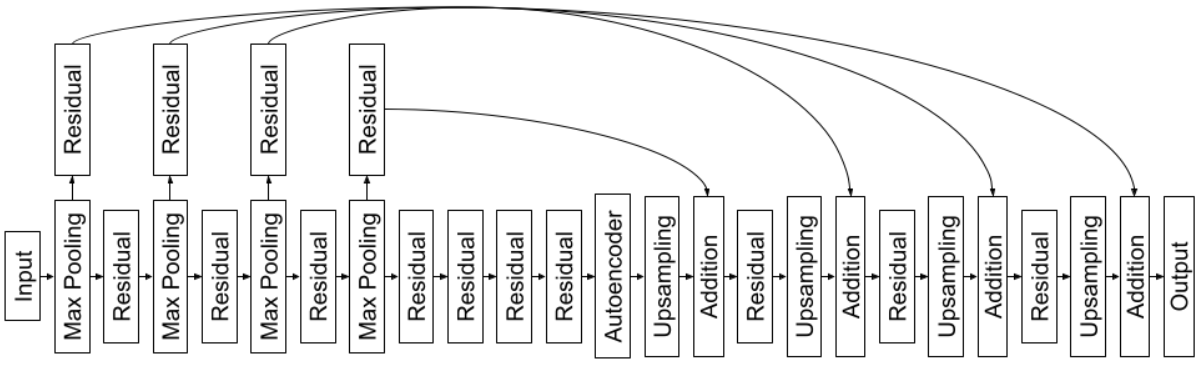


Figure 22: Visualization of the proposed combination of the hourglass and autoencoder for the stacked hourglass.

The autoencoder takes the output of the third residual of the bottleneck of our developed

stacked hourglass as input, hence why the autoencoder will be placed after the third residual to form a new proposed hourglass for the stacked hourglass, as visualized in Figure 22.

The training of the new model consits of two parts to speed up the training. First the autoencoder is trained isolated. Then, the trained autoencoder is placed in the developed stacked hourglass, following the structure of Figure **??**, and the whole network is trained.

The autoencoder is trained using stochastic gradient descent with momentum with $\alpha = 0.9$ and $\boldsymbol{v} = \mathbf{0}$, MSE as the loss function and a learning rate of $5e-4$, which is halved every 25th epoch. To increase the robustness of the autoencoder, we add noise sampled from

$$\mathcal{N}\left(0, x^2 e - 2\right)$$

to each training sample, where x is the value of the training sample. To help the model converge, we sample from a Glorot normal distribution, like in the case with the stacked hourglass.

After the autoencoder has been trained, the whole network is further trained by following Newell *et al.* [11] as described in Section 7.1.

9.3 Results

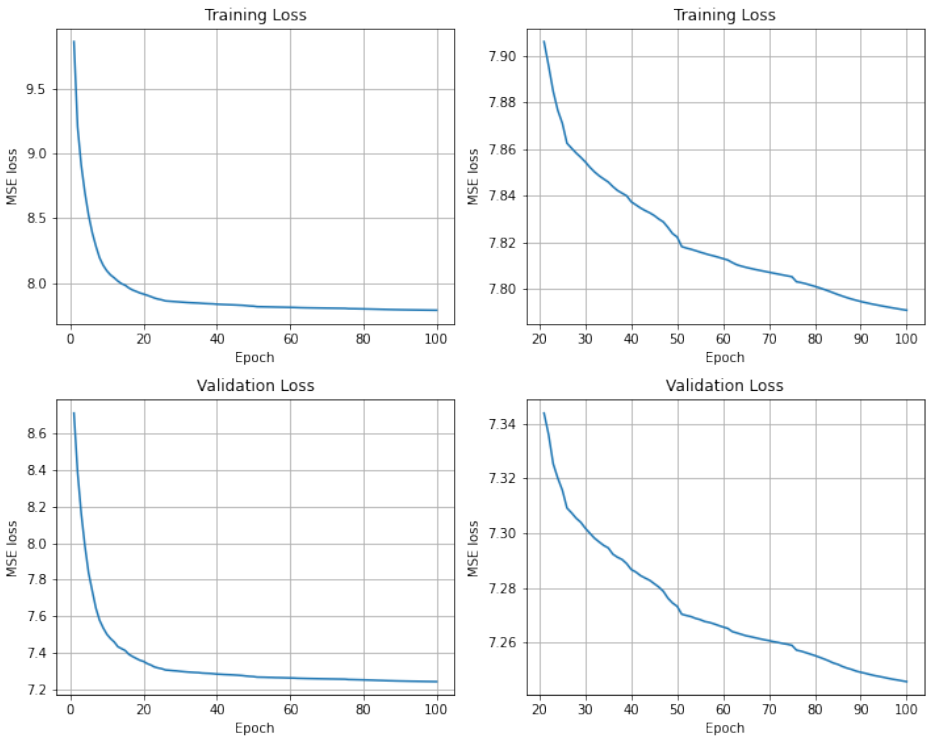


Figure 23: Visualization of the evolution of the training- and validation loss of the autoencoder during training. The left column shows all of the 100 epochs. Right column shows epoch 21 and foward

By training the autoencoder isolated, we get the evolution of the training- and validation loss visualized in Figure 23. We can clearly see, how the model does not start to overfit, as in the case when we trained the stacked hourglass. We decided to stop the training of the autoencoder, as each update only yielded minor changes to the model. The evolution of training the

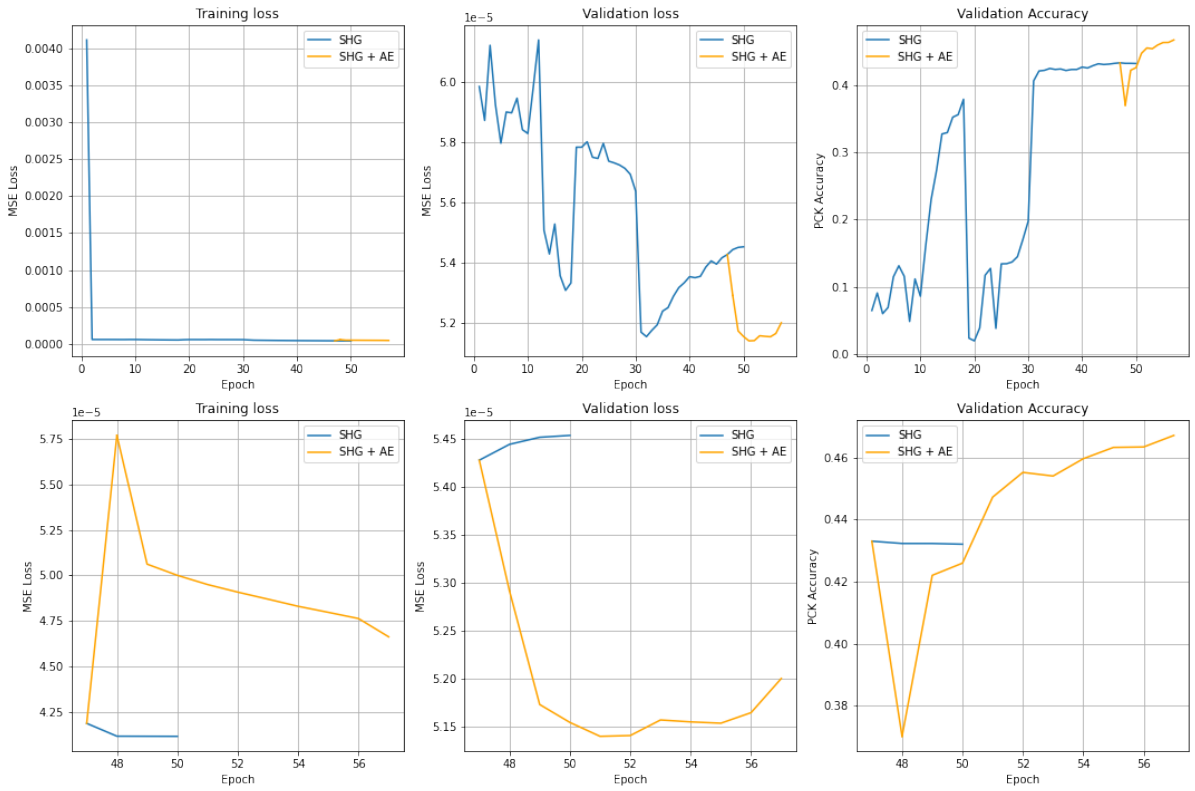


Figure 24: Visualization of the evolution of the training- and validation loss, as well as the PCK validation accuracy of the combination of the stacked hourglass and autoencoder, compared with the evolution of training the orignial stacked hourglass. The top row is of all of the 57 epochs. The bottom row shows epoch 47 and forward.

stacked hourglass with the autoencoder has been visualized in Figure 24. By looking at the figure we can see how the combined stacked hourglass and autoencoder initially performs worse than the original stacked hourglass, however, as the training continues it beats the original stacked hourglass, resulting an validation PCK accuracy of 0.467 - an increase of 7.8% or 0.034 compared with the original stacked hourglass.

To get an unbiased performance evaluation of the two models for comparison, we evaluate our two models on the held-out testing set from section 6. By doing so we get a PCK score of 0.441 for the standard stacked hourglass and a PCK score of 0.473 for the modified stacked hourglass. That is an increase of 7.25% or 0.032, by making use of an autoencoder.

Training Details

Training of the autoencoder, as well as the modified stacked hourglass, follow the training details described in Section 7.3. Training the autoencoder takes about **MANGLER** minutes per epoch, whereas the modified stacked hourglass takes about 70 minuttes per epoch, totalling to about 17.5 hours for the additional 15 epochs.

10 Discussion

- PCK: Scaler ikke med input størrelse
- Glorot Initialisering - kan være dårligt, se EML forelæsning om optimization.
- Acc. var stadig stigende - måske skulle jeg lade den træne længere
- Måske skulle jeg have gjort brug af gap statistics til at finde optimals k
- Curse of dimensionality ved clustering
- Se kap. 14.6 "Learning Manifolds with Autoencoders" i deep learning book. Til sidst er der en diskussion om brugen af Autoencoder til manifold learning
- Euclidisk afstand er måske ikke det rigtige?
- Man kan ikke gøre brug af den rigtige centroid for hvert cluster (grundet skip-connections). De centroids vi har fået kan eventuelt ligge langt væk fra de rigtige centroids, resulterende i misvisende centroids.
- Latent space af AE sættes til 50, idet vi ved shape analysis har set, at de resterende dimensioner er støj. Dette bygger sig dog på fulde skeleter og ikke alle skeleter, som modellen ellers trænes på

10.1 Summary of Obtained Results

In Section 7 we succefully implemented and trained a stacked hourglass, consisting of a single hourglass. We did this by following the configuration details described in Newell *et al.* [11] and Olsen [13]. The developed model has a validation PCK accuracy of 0.433 and a test PCK accuracy of 0.441.

In Section 8 we gained an understanding of how the devleoped model works by exploring the different components of the model. We could verify, that the skip-connections of the model were used for recreating details that are lost during the encoder-phase of the model, as argued by Newell *et al-* [11] and Olsen [13]. We then used PCA to gain an understanding of the structure of the latent space of the model. By doing so we came to the conclusion, that the model had learned the differences between people standing up and people sitting down, as well as possibly discovering some redundancy in the model, as principal componet 50 and above seemed to act as noise. Lastly, we used clustering to gain an understanding of how the model works. By doing so we learned, that the model knows the difference between fully-annotated people and not-fully annotated people, as well as knows the difference between stationary people and moving people. Here we also identified some possible reasons for inaccuracies of the model, as these classifications were not always correct.

In Section 9 we used our knowledge of the model to improve the performance of the model. This was done by developing and training an autoencoder, which was placed in the model. By doing so, both the validation and test PCK accuracy increase to 0.467 and 0.473, respectively.

10.2 Comparison of Models

10.3 Hvorfor er mine resultater dårligere/bedre end Newell/Camilla?

- Forskelle imellem min(e) modeller og deres
- Skal også komme ind på hvorfor mine valg var bedre end de valg de valgte

10.3.1 Forskelle

- Batch normalization - ved ikke placering
- Autoencoder
- Noget andet data
- Gør også brug af $v = 1$
- Anden spredning ved gaussian filter ved $v = 1$
- PCK med fixed normalization konstant straffer folk tættere på kammeraet mere end folk længere væk

10.4 Future Work

If we were to work further with this project, it would be ideal to explore the effects of stacking multiple modified hourglasses end-to-end. By doing so we would not only hope that the performance of the model to increase further, but we would also hope we could obtain the same accuracy as Newell *et al.* experiences [11], however with fewer stacks. For instance, we could hope that by stacking 2 modified hourglasses, we would achieve the same results as Newell *et al.* achieves with 4 standard hourglasses.

11 Conclusion

We have succefully implemented and trained a stacked hourglass, developed by Newell *et al.* [11], consisting of a single hourglass. The network was trained and validated on the Microsoft COCO dataset [10]. We have then interpreted the model by (1) verifying the effects of some of the parts of the model, (2) finding the effects of the principal components of the latent space of the model, and (3) found the effects of the clusters of the latent space of the model. By doing so we have gained an understanding of how the model works, found redundancy in the model, as well as found reasons for the models inaccuracies. Lastly, we used our knowledge of the model to succefully improve the performance of the developed model.

12 References

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Ed. by Michael Jordan, Jon Kleinberg, and Bernhard Schölkopf. Springer, 2006.

[2] Qi Dang, Jianqin Yin, Bin Wang, and Wenqing Zheng. *Deep Learning Based 2D Human Pose Estimation: A Survey*. Tech. rep. 6. Version 24. Tsinghua Science and Technology, 2019.

[3] Daniel Falbel, JJ Allaire, and François Chollet. *Glorot normal initializer, also called Xavier normal initializer*. Keras. URL: https://keras.rstudio.com/reference/initializer_glorot_normal.html. (accessed: 22.4.2021).

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. (accessed: 18.3.2021).

[5] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *Elements of Statistical Learning. Data Mining, Inference, and Prediction. Second Edition*. Springer, 2017.

[6] Bulat Ibragimov. *Modelling and Analysis of Data, Lecture 3 - Clustering*. 12.1.2020. URL: https://absalon.ku.dk/courses/42639/files/4453260?module_item_id=1189839. (accessed: 10.5.2021).

[7] Bulat Ibragimov. *Modelling and Analysis of Data, Lecture 3 - Nonlinear Regression*. 25.11.2020. URL: https://absalon.ku.dk/courses/42639/files/4289570?module_item_id=1145250. (accessed: 17.3.2021).

[8] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning with Applications in R. First Edition*. Springer, 2017.

[9] Jeremy Jordan. *Introduction to autoencoders*. Jeremy Jordan. URL: https://www.jeremyjordan.me/autoencoders/. (accessed: 28.5.2021).

[10] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollar. *Microsoft COCO: Common Objects in Context*. Tech. rep. Microsoft Research, 2014.

[11] Alejandro Newell, Kaiyu Yang, and Jia Deng. *Stacked Hourglass Networks for Human Pose Estimation*. Tech. rep. 1603.06937. University of Michigan, 2016.

[12] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. Tech. rep. 1511.08458. Department of Computer Science, Aberystwyth University, Ceredigion, School of Computing, and Communications, Lancaster University, 2015.

[13] Camilla Maach Brønnum Olsen. "Articulated Pose Estimation of Humans". MA thesis. University of Copenhagen, Department of Computer Science, 2019.

[14] Jens Petersen. *Elements of Machine learning - Optimization in Deep Learning*. 2021. URL: https://absalon.ku.dk/courses/46845/files/4490846?module_item_id=1206787. (accessed: 25.3.2021).

[15] Moacir A. Ponti, Leonardo S. F. Ribeiro, Tiago S. Nazare, Tu Bui, and John Collomosse. *Everything you wanted to know about Deep Learning for Computer Vision but were afraid to ask*. Tech. rep. ICMC – University of Sao Paulo and CVSSP – University of Surrey, 2017.

[16] Mike Pound. *Resizing Images - Computerphile*. Computerphile. URL: https://www.youtube.com/watch?v=AqscP7rc8_M. (accessed: 19.4.2021).

[17] Simon Rogers and Mark Girolami. *A First Course in Machine Learning*. Chapman and Hall/CRC, 2017.

[18] Grant Sanderson. *Backpropagation calculus | Deep learning, chapter 4*. 3Blue1Brown. URL: https://www.youtube.com/watch?v=tIeHLnjs5U8&ab_channel=3Blue1Brown. (accessed: 7.4.2021).

[19] Grant Sanderson. *But what is a Neural Network? | Deep learning, chapter 1*. 3Blue1Brown. URL: https://www.youtube.com/watch?v=aircAruvnKk. (accessed: 18.3.2021).

[20] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrisna Vedantam, Devi parikh, and Dhruv Batra. *Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization*. Tech. rep. 1610.02391. 2019.

[21] Steven S. Skiena. *The Data Science Design Manual*. Ed. by David Gries, Orit Hazzan, and Fred B. Schneider. Springer, 2017.

[22] Princeton Vision and Learning Lab. *pose-hg-train*. URL: https://github.com/princeton-vl/pose-hg-train. (accessed: 5.5.2021).

[23] *What is a Neural Network?* URL: https://deepai.org/machine-learning-glossary-and-terms/neural-network. (accessed: 17.3.2021).

[24] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. 0.16.4. 2021.

[25] Ce Zheng, Wenhan Wu, Taojiannan Yang, Sijie Zhu, Chen Chen, Ruixu Liu, Ju Shen, Nasser Kehtarnavaz, and Mubarak Shah. *Deep Learning-Based Human Pose Estimation: A Survey*. Tech. rep. 2012.13392. IEEE, University of North Carolina and University of Dayton and University of Texas University of Central Florida.