FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

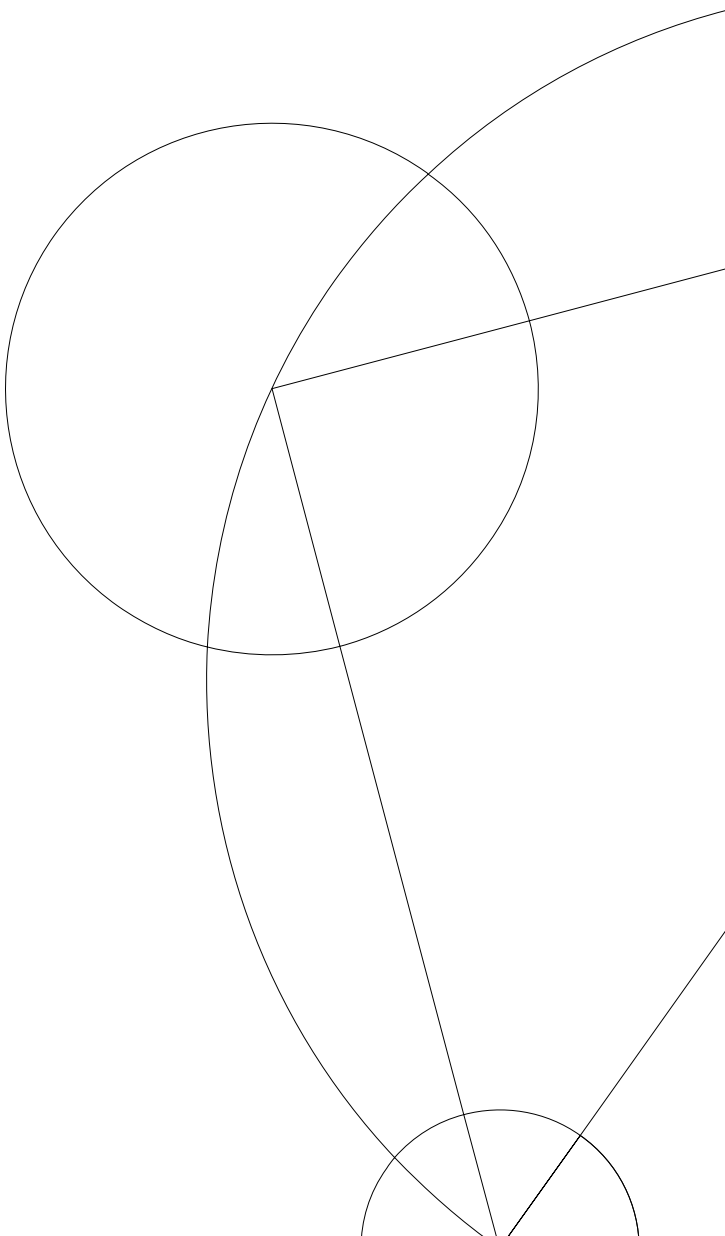
Bachelor Thesis

2D Articulated Human Pose Estimation

Using Explainable Artificial Intelligence

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

June 11, 2021



Supervisor
Kim Steenstrup Pedersen kimstp@di.ku.dk

Abstract

In this thesis we implement and train the Stacked hourglass developed by Newell *et al.* [15]. The network is trained, validated and tested on the Microsoft 2017 COCO dataset [14]. This is followed by an interpretation of the developed model. In the interpretation we first off verify Newell *et al.* [15] and Olsen's [18] claim, that the skip-connections are used for recreating details that are lost during the encoder-phase. Then, we analyse the structure of the latent space. In correlation to this we conclude, that the model has learned the difference between moving and stationary people, learned the difference between almost fully-annotated people and not fully-annotated people, as well as use the first principal component to know if a given person is sitting down or standing up. During the interpretation we further conclude, that there are some redundancy and misplaced training samples in the latent space of the model. Finally, we use the obtained knowledge about the model to improve the performance of the model and remove redundancy in the model. This is done by modifying the architecture of the model to include an autoencoder with a reduced information bottleneck.

Contents

1 Introduction **6**

2 Machine Learning Theory **8**
2.1 Motivation . . . 8
2.2 Machine Learning Paradigms . . . 8
2.3 Evaluation of Machine Learning Models . . . 8
2.3.1 Splitting the dataset . . . 8
2.3.2 Evaluation Metrics for Supervised Machine Learning . . . 9
2.4 General Machine Learning Terminology . . . 9
2.5 Neural Networks . . . 12
2.5.1 Feedforward Neural Networks . . . 12
2.5.2 Convolutional Neural Networks . . . 14

3 Autoencoders, Stacked Hourglass, PCA and K-Means **17**
3.1 Autoencoders (AE) . . . 17
3.2 Stacked Hourglass (SHG) . . . 17
3.2.1 Motivation behind using the Stacked Hourglass . . . 18
3.2.2 The Residual Module . . . 18
3.2.3 The Hourglass . . . 18
3.2.4 The Stacked Hourglass . . . 19
3.3 Principal Components Analysis (PCA) . . . 19
3.4 K-Means Clustering . . . 20

4 The Dataset **23**
4.1 The COCO Dataset . . . 23
4.2 Data Preprocessing . . . 24
4.2.1 Creating the test dataset . . . 24
4.2.2 Preprocessing the Images . . . 24
4.2.3 Handling the Keypoints . . . 25

5 Experiment **26**
5.1 Configuration Details . . . 26
5.2 Results . . . 28
5.3 Training Details . . . 28

6 Interpreting the Model **30**
6.1 Motivation . . . 30
6.2 Verifying the Effects of Skip-Connections . . . 30
6.3 Dimensionality Analysis of the Latent Space . . . 31
6.4 Using Clustering to Seperate the Latent Space . . . 33

7 Improving the Model **38**
7.1 Motivation . . . 38
7.2 Configuration Details . . . 38
7.3 Results . . . 39
7.4 Training Details . . . 40

8 Discussion **42**
8.1 Summary of the Obtained Results 42
8.2 Comparison of Models 42
8.3 Why did the Autoencoder Improve the Stacked hourglass? 44
8.3.1 Clustering the Latent Space 44
8.3.2 Removing Noisy Features from the Latent Space 45
8.4 Future Work 46

9 Conclusion **47**

10 References **48**

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
$L(a, b)$	An arbitrary loss function, that computes the loss between a and b

1 Introduction

Figure 1: Example of 2D single-person articulated human pose [2]

It is common knowledge, that the real-world use of artificial intelligence and machine learning is growing rapidly. With this growth the need for accurate computer vision models is also increasing. One usage of computer vision is *2D 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 [33].

The goal of human pose estimation is to estimate the pose of one or multiple humans in images or videos. There are different types of human pose estimations, where one of the most common ones is the *articulated* human pose estimation, which is done by estimating the location of various keypoints of the human bodies in an input. The methods within human pose estimation can further be split into $2D$ human pose estimation and $3D$ human pose estimation, which describes the amount of dimensions in the estimations. An example of a 2D articulated human pose of a single person is visualized in Figure 1.

As the complexity of machine learning models have 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 often be a problem, especially in cases where the output of the model can result in a life or death situation of a human. For that reason, understanding how a model works can be very important - this is what is called *explainable AI* (*XAI*). Selvaraju *et al.* [26] argues that there are three cases for using explainable AI: (1) an understanding of the model can help us improve the performance of the model (2) an understanding of the model can help us build trust in the model, as we can understand its strenghts and weaknesses, and (3) an understanding of the model can teach humans how to perform better, in cases where the model outperforms humans [26].

Early human pose estimation methods were based on a classic approach. For instance, Felzenszwalb *et al.* uses the Histogram of Oriented Gradient features and Support Vector Machines [7]. More recent pose estimation methods are based on deep learning. Toshev *et al.* introduced *DeepPose*, that uses a cascade of deep neural networks to estimate the location of joints, by starting with an estimation based on the full image, which is then refined using sub-images [28]. Carreira *et al.* developed a self-correcting deep learning model, that works by progressively changing an initial solution by feeding back error predictions [3]. Newell *et al.* introduced the Stacked hourglass, which works by continuously pooling and upsampling the input data to produce a set of heatmaps, where the maximum activation of each heatmap is the location of

the corresponding keypoints [15].

There are also a range of different techniques to explain a developed model. Zeiler *et al.* uses deconvolutional layers to visualize what each convolutional layer has learned [31]. Selvaraju *et al.* uses a gradient based technique to explain the important regions in an image for a prediction [26]. Erhan *et al.* explains the effects an unit, by finding out what inputs that maximize the unit [5].

The goal of this thesis is thus to select a model for human pose estimation, to interpretate the developed model, as well as use the obtained knowledge about the model to modify and improve the model. We will not be aiming for a very accurate initial model, as we will be focusing on interpretating the model instead. For the model of choice for human pose estimation, we will be using the Stacked hourglass by Newell *et al.* [15]. We have decided to make this choice, as the Stacked hourglass 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 model useful for encoding the data into a lower dimension, which can be useful when we will be doing the interpretation of the model. Our technique for explaining of the network will differ from the techniques explained previously. Instead, we will be looking and analyzing the features of the different major parts of the Stacked hourglass to get an understanding of how the model works. Thus, we will not be looking at the minor parts of the Stacked hourglass, such as each convolutional layer, like Zeiler *et al.* does [31], as our model simply is to deep, making it difficult to figure out what role each part plays.

In the remainder of this thesis, Section 2 introduces the basic machine learning theory and Section 3 introduces the most important algortihms used throughout the thesis. In Section 4 the used dataset and its preprocessing is described. We then describe our development of a model and its respective results in Section 5, which is then explored and interpreted in Section 6. In Section 7, we then use our knowledge of the model to improve the performance of it. We then discuss our approach and results in Section 8. Lastly, we conclude our results in Section 9.

2 Machine Learning Theory

Throughout this section the theory of machine learning that will be used in this thesis is described and explained. In Section 2.1 we describe the general motivation behind using machine learning. Then, in Section 2.2 we explain the three most common paradigms in machine learning and how we will use some of them. In Section 2.3, we give a brief overview of how to evaluate a developed model. Lastly, in Section 2.5, we describe the mathematics behind feedforward- and convolutional neural networks .

2.1 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 5 to develop a model to estimate the 2D pose of a single human in an image. Later, in section 6, we will use machine learning to improve our understanding of the model.

2.2 Machine Learning Paradigms

Machine learning is usally split into the at least two paradigmes:

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 [9]. Supervised learning is further split into *classification* and *regression*.

2. *Unsupervised learning* where the data only consists of features. The algorithm then learns properties of the data, without any provided labels [9].

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

2.3 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 to unseen data, which is done by making use of several dataset splits and *evaluation metrics*.

2.3.1 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 non-overlapping chunks. The model is then trained for K rounds on $K-1$ chunks, where the last chunk is used for evaluating the model [22].

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

Algorithm 1 PCK [18][29]

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 $ratio$

best validation loss. Lastly, the testing dataset is used as a final evaluation of the model to yield an unbiased evaluation of the ability of the model to generalize to unseen data. Once the testing dataset has been used it can no longer be used for evaluating the data, as this potentially introduces bias in future evaluations [11].

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.

2.3.2 Evaluation Metrics for Supervised Machine Learning

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. In articulated human pose estimation, one commonly used evaluation metric is the *Percentage of Correct Keypoints* (*PCK*), which states the percentage of predictions that are within a normalized distance of the ground truth.

The pseudocode of PCK is visualized in Algorithm 1. The algorithm works by iterating 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 [18] [29]. The aim is thus to maximize the PCK accuracy.

2.4 General Machine Learning Terminology

Loss Functions
The training of a machine learning model is done by minimizing a given *loss function*, which measures the error of the model. 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 [12].

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* [8]. One commonly used Regularization technique is *batch normalization*, which we will be using when we in Section **??** wil be developing our model. Batch normalization will be described further in Section 2.5.

Algorithm 2 Gradient Descent [32]

Require: Learning rate η

Require: Initial parameter $\boldsymbol{\theta}$

1: **while** stopping criterion not met **do**

2: Sample an observation from the training set $\boldsymbol{x}$ with corresponding target $\boldsymbol{y}$

3: Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta\nabla_{\boldsymbol{\theta}} L\left(f(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{y}\right)$

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 2. The algorithm works by taking a learning rate η and the initial parameter $\boldsymbol{\theta}$ as input. It then computes $\nabla_{\boldsymbol{\theta}} L\left(f(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{y}\right)$ and subtracts the gradient times η from $\boldsymbol{\theta}$. This is done until a stopping condition is met, such as when the magnitude of the gradient $|\nabla_{\boldsymbol{\theta}} L\left(f(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{y}\right)|$ is small or until a maximum amount of iterations has been reached [32].

Online, Mini-batch and Batch methods

When gradient descent is used in machine learning, computing $\nabla_{\boldsymbol{\theta}} L\left(f(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{y}\right)$ can be done by averaging the gradient of each of the n observations of the training set, 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. Thus, batch gradient descent uses the fewest iterations, however, each iteration takes the longest to compute, whereas in online gradient descent each iteration is computed quickly, however, it also uses more iterations [32].

Choosing the right batch size can be difficult, however, there are a few guidelines which one can follow [19] [32]

1. 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.

2. 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 3 Stochastic Gradient Descent with Momentum [8]

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

Algorithm 4 RMSProp [8]

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

Optimization Algorithms
An example of a mini-batch based optimization algorithm is *Stochastic Gradient Descent* (*SGD*). 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 gradient of n samples to update its parameters.

One problem of the gradient descent algorithm 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 is visualized in Algorithm 4. The algorithm works by using an decaying average that discards knowledge from the past, so that it can converge after finding a convex bowl of the loss function. The algorithm uses a hyperparameter ρ, that controls the length scale of the moving average [8].

Momentum
One problem with SGDis how slow it often can be. For this reason *momentum* is often used to accelerate learning. Momentum works by accumulating a decaying moving average of the past gradients and continuing to move in their direction. This is done 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 [8].

Epoch
An *epoch* is an iteration through the complete dataset during fitting of the network. Multiple epochs are often needed to reach the minimum of the loss function [32].

2.5 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 2.5 the theory and mathematics behind neural networks is described and explained.

2.5.1 Feedforward Neural Networks

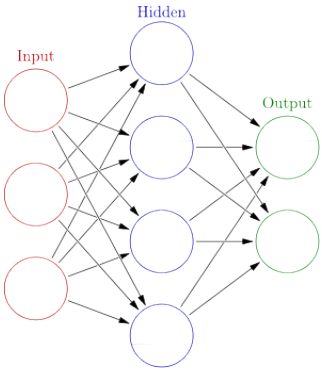


Figure 2: Visualization of a feedforward neural network with a single hidden layer [30]

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 2. The network is a directed acyclic graph and 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 2), 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 2), 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 a non-linear *activation function*, that is typically applied element-wise [8] [24]. The purpose behind using non-linear activation functions is to make it possible for the output of the network to be non-linear. 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 non-linear 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 [8].

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

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

[8]. If we use this to find the gradient of each parameter, we will find, that the partial derivative of the loss L 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 [23].

Batch Normalization

Batch normalization is a reparametrization method, which is applied to individual layers in a neural network. 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

$$\hat{\boldsymbol{\mu}}_{\mathcal{B}} = \frac{1}{|\mathcal{B}|} \sum_{\boldsymbol{x} \in \mathcal{B}_{\boldsymbol{x}}}$$

and

$$\hat{\boldsymbol{\sigma}}_{\mathcal{B}} = \sqrt{\frac{1}{|\mathcal{B}|} \sum_{\boldsymbol{x} \in \mathcal{B}} (\boldsymbol{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}})^2 + \epsilon},$$

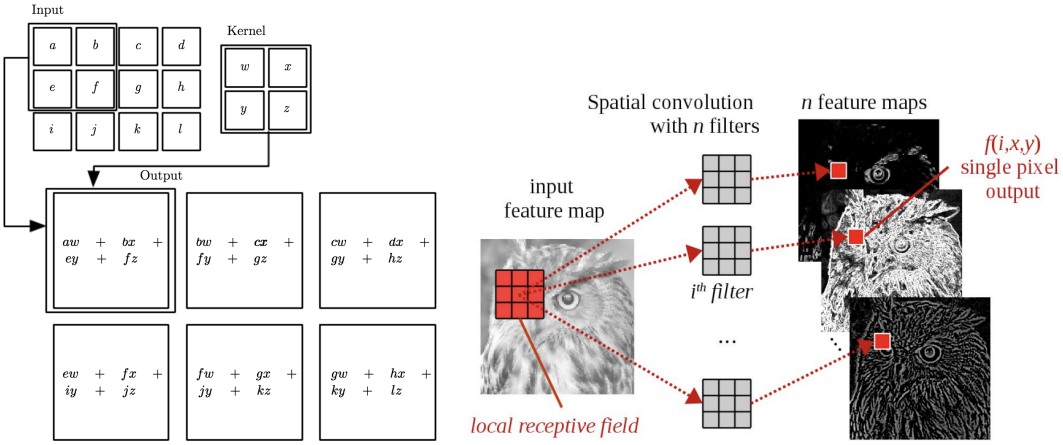
which makes the minibatch have 0 mean and unit variance. $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are often trainable parameters, that are used to make the mini-batch have an arbitrary mean and standard deviation. This helps the network to converge, as the batch normalization keeps centering the mean and standard deviation of the mini-batches. The small constant $\epsilon > 0$ is simply used to avoid division by zero. [32].

2.5.2 Convolutional Neural Networks

Feedforward neural networks introduced in 2.5.1 can be used for pattern recognition within images, however, they are usually not used for this task. Consider a colored input image with 3 channels 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. By using a CNN the high correlation between a pixel and its neighbours is also captured, essentially leading to better results. 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 [17].

Convolutional layers

A convolutional layer is composed of a set of *kernels* (also known as *filters*), which are matrices of weights of dimension $c \times k \times k$, where c is the amount of channels (or amount of filters to be applied) and k kernel-size, usually 5, 3 or 1. The convolutional layer thus consists of c filters, each of $k \times k$ parameters, where each weight is a parameter for the model to be learned [20]. Each kernel is used on the input to produce a *feature map*. The kernels are applied to the input



(a) Example of a 2D convolution [8]

(b) Example of a convolution with multiple filters [20]

Figure 3: Convolutions visualized

by "sliding" over the input (where the step size is called *stride*). 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 3. When all of the feature maps have been computed of a convolutional layer an 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 $3 \times 3 \times 3$ convolution on a colored image with 3 channels, 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 overfit [17].

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 [17].

Algorithm 5 Nearest Neighbour Upsampling [21]

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

Nearest Neighbour Upsampling

Sometimes we want to increase the size of an image. This can be done by making use of *upsampling* (also known as *interpolation*) techniques. One of the most common upsampling

techniques is *nearest neighbour upsampling*, whose pseudocode has been written in Algorithm 5. The algorithm starts off by taking an image, X, of size $m \times n$, as input, which we wish to upsample to size $xm \times yn$, where x and y are positive whole numbers. 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 [21].

3 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 3.1, where we will be giving a brief introduction to autoencoders. Throughout Section 3.2 the Stacked hourglass, used for pose estimation, is described. Then, in Section 3.3 the algorithm of Principal Components Analysis is described. In the last section, Section 3.4, K-Means clustering is described.

3.1 Autoencoders (AE)

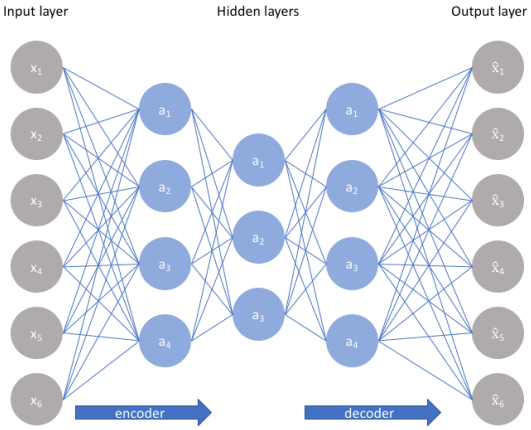


Figure 4: Visualization of an undercomplete autoencoder [13]

Autoencoders (often shortened as *AE*) is a class of neural networks that are trained unsupervised 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 [8].

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 4. 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 [8].

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 [8]. There are various loss functions one could use. In Section 7, where we will be using the autoencoder, we will be using MSE as it fits to our problem.

3.2 Stacked Hourglass (SHG)

When performing the pose estimation in section 5, we will be implementing and using the *Stacked hourglass* (also knowns as *SHG*) by Newell *et al.* [15]. The following description and explanation of the architecture is based on an interpretation of Newell *et al.* [15] and Olsen [18].

3.2.1 Motivation behind using the Stacked Hourglass

MANGLER

3.2.2 The Residual Module

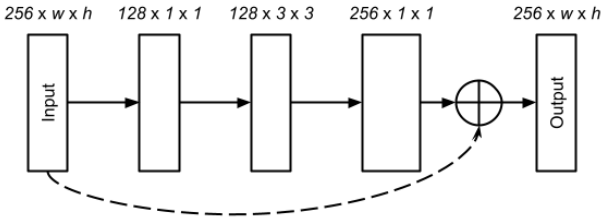


Figure 5: Visualization of the residual module [15]

The Stacked hourglass makes heavily use of so-called *residual modules*, one of which is visualized in Figure 5. The module works by taking an input, which is sent through a 1×1 convolution and a 3×3 convolution, each with 128 channels. Then, the 128 output feature maps are sent through a 1×1 convolution with 256 channels. Lastly, element-wise addition is then used to add the 256 output feature maps 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 feature maps are of the same dimensions as the input featuremeaps.

The residual module makes use of skip-connections to help the model learn. If many residual modules without the skip-connection were stacked, the network could have problems choosing the values of the parameters, making the network perform worse, than in the case of shallower networks. By making use of the skip-connection, the module can easily learn to "skip" unnecessary parameters, essentially making network act as if it was shallower [16].

3.2.3 The Hourglass

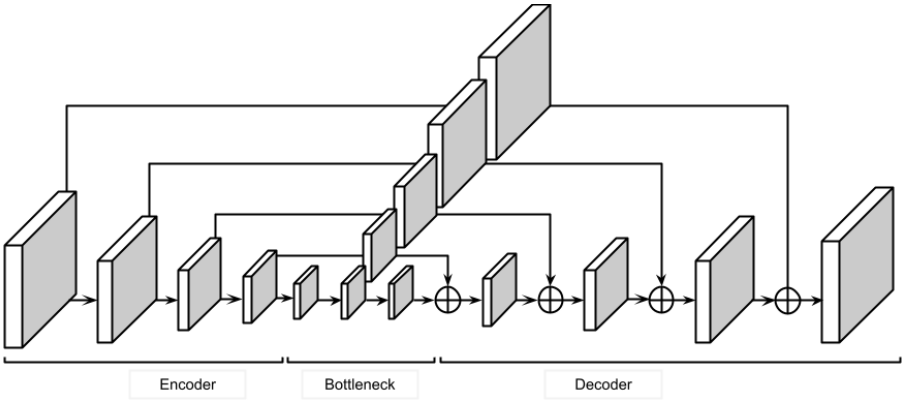


Figure 6: Visualization of a single hourglass [15]

The Stacked hourglass consists of hourglasses, where each hourglass is split into an encoder, where the feature maps is downsampeld, a decoder, where the feature maps are upsampled, and a bottleneck. 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 6 a single hourglass han been visualized, where each box represents 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 feature maps until the feature maps have the same dimensions as the input of the hourglass. Before each max pool in the encoder, the module branches off and applies a residual at different resolutions. 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 module has the bottleneck, where no downsampling or upsampling happens, instead only residuals are processing the feature maps. After the decoder a residual and two 1×1 convolution layers with 256 and n channels, respectievely, are applied to produce the final predictions, where n is the amount of keypoints to be predicted.

3.2.4 The Stacked Hourglass

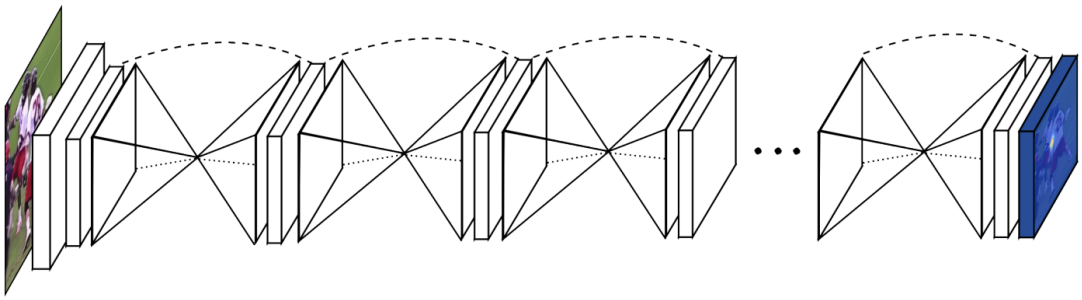


Figure 7: Visualization of the Stacked hourglass [15]

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 7, 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 $256 \times 7 \times 7$ convolution layer with stride 2, followed by a residual module and max pooling to bring the resolution down to the output resolution of 64×64, which is then input to the first hourglass.

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

As the Stacked hourglass performs regression, the MSE will be minimized during training to optimize the performance of the model.

3.3 Principal Components Analysis (PCA)

It is very common, that a given dataset has an enormous amount of dimensions. This quickly becomes 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* [9]. For this reason multiple techniques have been developed for reducing the dimensions of a given dataset. We have already seen in Section 3.1, how autoencoders can be used as non-linear dimensionality reduction. Another very common

Algorithm 6 PCA [22]

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** $\boldsymbol{X}$

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, which the input data is projected onto [22].

The pseudocode of the algorithm is 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. [22]

3.4 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 is 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 [22].

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 [22].

Algorithm 7 K-Means [27]

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

Algorithm 8 Compute Silhouette Score [10]

Require: Clusters $\boldsymbol{C}_0, \boldsymbol{C}_1, ..., \boldsymbol{C}_{k-1}$
for each cluster $\boldsymbol{C}_i$ **do**
    for each sample $\boldsymbol{x} \in \boldsymbol{C}_i$ **do**
        Compute the mean euclidean distance from $\boldsymbol{x}$ to the other samples in the same cluster: $a(\boldsymbol{x}) = \frac{1}{|\boldsymbol{C}_i|-1} \sum_{\boldsymbol{y} \in \boldsymbol{C}_i} D(\boldsymbol{x}, \boldsymbol{y})$
        Compute the mean euclidean distance from $\boldsymbol{x}$ to the nearest other cluster: $\boldsymbol{C}_j$ $b(\boldsymbol{x}) = \frac{1}{|\boldsymbol{C}_j|} \sum_{\boldsymbol{z} \in \boldsymbol{C}_j} D(\boldsymbol{x}, \boldsymbol{z})$
        Compute the Silhouette of $\boldsymbol{x}$: $s(\boldsymbol{x}) = \begin{cases} 1 - \frac{a(\boldsymbol{x})}{b(\boldsymbol{x})} & \text{if } a(\boldsymbol{x}) < b(\boldsymbol{x}) \\ 0 & \text{if } a(\boldsymbol{x}) = b(\boldsymbol{x}) \text{ or } |\boldsymbol{C}_i| = 1 \\ \frac{b(\boldsymbol{x})}{a(\boldsymbol{x})} - 1 & \text{if } a(\boldsymbol{x}) > b(\boldsymbol{x}) \end{cases}$
return mean of the Silhouettes

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 8. 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 the 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 [10].

4 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 4.1, where a brief overview of the dataset is given, and Section 4.2, where the preprocessing of the data is described.

4.1 The COCO Dataset

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

Figure 8: Example of an image from the COCO dataset with the keypoints drawn on [14]

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 [14], 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 8.

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.

4.2 Data Preprocessing

4.2.1 Creating the test dataset

To create the testing dataset 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.

4.2.2 Preprocessing the Images

	Amount of images	Percentage
Training set	124.040	$92,450\%$
Validation set	5.064	$3,775\%$
Testing set	5.064	$3,775\%$
Total	**134.168**	**100**$\%$

Table 1: Data distribution

Figure 9: The results of processing the image from Figure 8 with the corresponding labels [14]

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 9. 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 cropped 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`. The corresponding ground truth heatmaps are likewise centered and cropped around the person the model should annotate.

By doing the data preprocessing as described above, we get the distribution of images displayed in Table 1. In Figure 9, the results of processing the image from Figure 8 are shown with the corresponding labels. Lastly, the data is shuffled to help the developed model generalize to unseen data better.

4.2.3 Handling the Keypoints

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

Figure 10: An example of the heatmaps of a single image fused together and put over the original image [4]

For each image our model outputs 17 heatmaps, one for each possible joint in the image. An example of such heatmaps fused together can be seen in Figure 10. 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 the heatmap is resized. Next, a Gaussian filter is used to smear out the heatmap, where the standard deviation 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. In the case of a not-annotated keypoint, that is, when $v = 0$, we do not place a 1 in the heatmap, leaving the heatmap empty. 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 [15].

5 Experiment

Throughout the following section the results and configuration details of our trained model are described and explained. In the first section, Section 5.1, the configuration details of the model, as well as the training of the model, are described. In the second section, Section 5.2, 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 5.3, 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.

5.1 Configuration Details

Our model only consists of a single hourglass. The hourglass consists of 4 down- and upsamples. The module uses a single residual between each down- and upsample, as well as 3 residuals in the bottleneck. Newell *et al.* [15] and Olsen [18] 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* [15], hence why we will be making the same configuration choices as Newell *et. al* [15] and Olsen [18], 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 [18] 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 ReLU after each batch normalization. Each max pooling and nearest neighbor upsampling uses a kernel size and stride of 2, which halves and doubles the size of the input, respectively. The full network has been visualized in Figure 11.

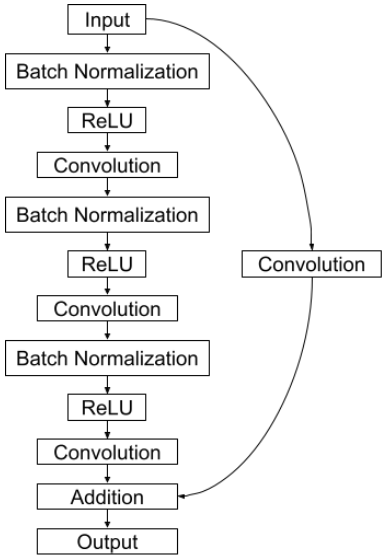
For the initial values of the weights we initialize each weight by sampling from a *Glorot normal distribution* (also known as a *Xavier normal distribution*), described as

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

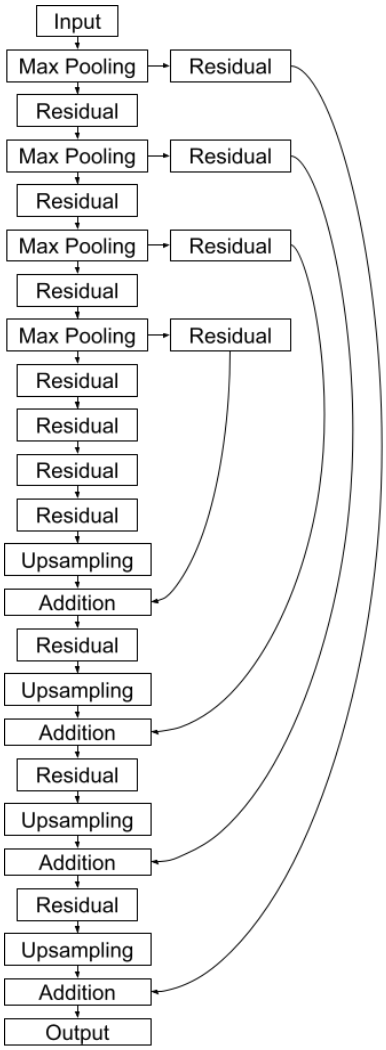
where fan_{in} is the amount of input connections and fan_{out} is the amount of output connections to the layer of the weight [6]. By doing so we make all layers have the same activation variance and gradient variance, essentially helping the model to converge [8].

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* with momentum paramter $\alpha = 0.99$ as our optimizer, as well as use a initial learning rate of $2.5e-4$ as done by [15] and [18]. We decided to use a mini-batch size of 16 as Olsen experienced great results with this mini-batch size [18], as well as it being the biggest mini-batch size that our machine could run.

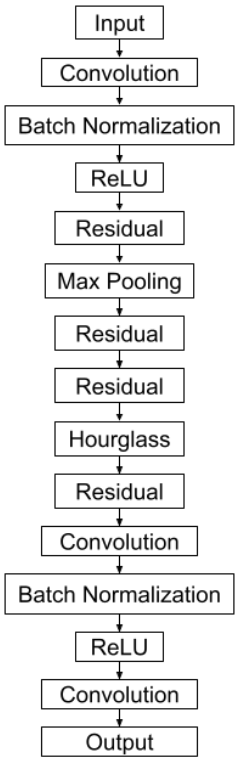
After each epoch we find the validation accuracy of the model by computing the PCK between the predictions and the ground truth of the validation dataset. For the two constants



(a) The Residual module



(b) The Hourglass



(c) The Full network

Figure 11: Overview of the used architecture

for PCK, c (the normalization constant) and r (the threshold radius), we follow Olsen [18] by letting c be one tenth of the heatmap resolution size (that is, $\frac{64}{10} = 6.4$) and r be 0.5.

While training the model the PCK accuracy 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.

We do not make use of automated stopping criterias, such as *automated early stopping*, as these have certain parameters that can be difficult to choose. Instead, we train the network and save a copy of the model after each epoch. While the training is happening, we note the training loss, validation loss as well as the validation PCK accuracy of the model after each epoch. If we notice, that the model does not improve for a while, we stop the training process and use the saved model with the best results.

5.2 Results

In Figure 12, the evolution of the training loss, validation loss and validation PCK accuracy is 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 12 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 6 want to explore a model that performs decently well, we will be using the model with the highest validation accuracy as our model going forward. 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. If we test the model on the held-out testing set from Section 4, we get a PCK score of 0.441, which is based on all annotated keypoints (that is, both for $v = 1$ and $v = 2$)

5.3 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

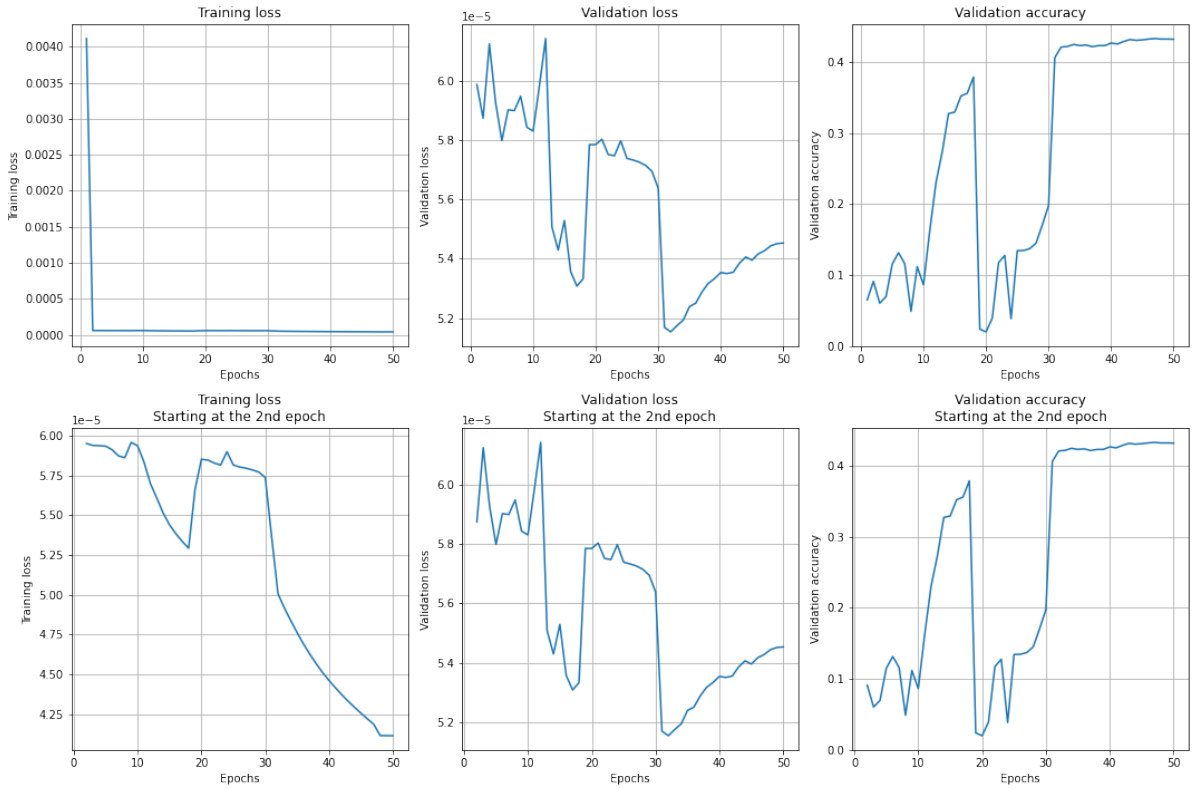


Figure 12: Visualization of the evolution of the training loss, validation loss and validation PCK accuracy of the trained model. Top row shows all of the 50 epochs. Bottom row shows epoch 2 and forward to ease the reading of the training loss

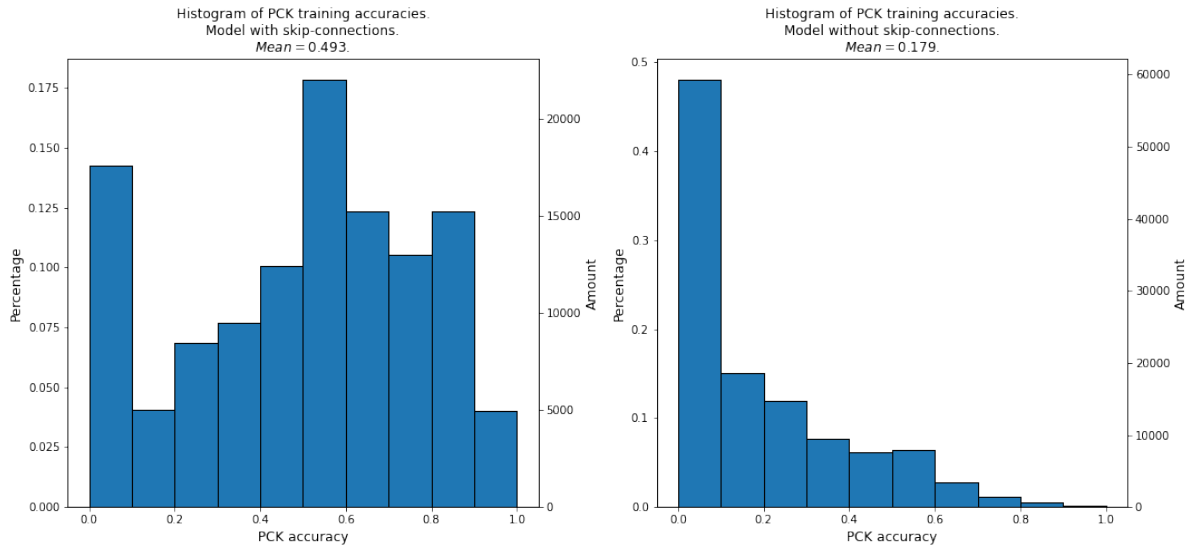


Figure 13: Histogram of PCK training accuracies of the model with the skip-connections enabled (left) and the model with the skip connections disabled (right).

6 Interpreting the Model

In the following section we will be interpreting the model developed in section 5, 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 6.1 an overview of the motivation behind interpreting the model is given. Section 6.2 then evaluates the effects of the skip-connections of the model. Then, Section 6.3 explores the latent space of the model with respect to getting an understanding of the principal components of the latent space. Section 6.4 also explores the latent space of the model, however, instead it uses clustering to seperate the latent space.

6.1 Motivation

MANGLER Throughout section 6 we will be exploring and understanding what the model, developed in section 5, learned during training and what the different parts of the model are used for, leading towards improving the performance of the model easier.

6.2 Verifying the Effects of Skip-Connections

Olsen [18] and Newell [15] claims, that the skip-connections are used in order to recreate details that are lost during the encoder-phase. Throughout subsection 6.2 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 5
2. The trained Stacked Hourglass from section 5, 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 13 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 14 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 is visualized.

By looking at Figure 14 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 [18] and Newell's [15] claims, that the skip-connections are used for recreating details lost in the encoder.

6.3 Dimensionality Analysis of the Latent Space

In subsection 3.2.1 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 $256 \times 4 \times 4$ 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.

We start off by finding the 4.096 principal components and the corresponding explained variance ratio of the data by using PCA. 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 real data 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 real data point is found and stored. By comparing the ground truth heatmaps of these real data points, we can see the transition of the heatmaps along the principal component, which will give us an idea of what effects the component has.

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

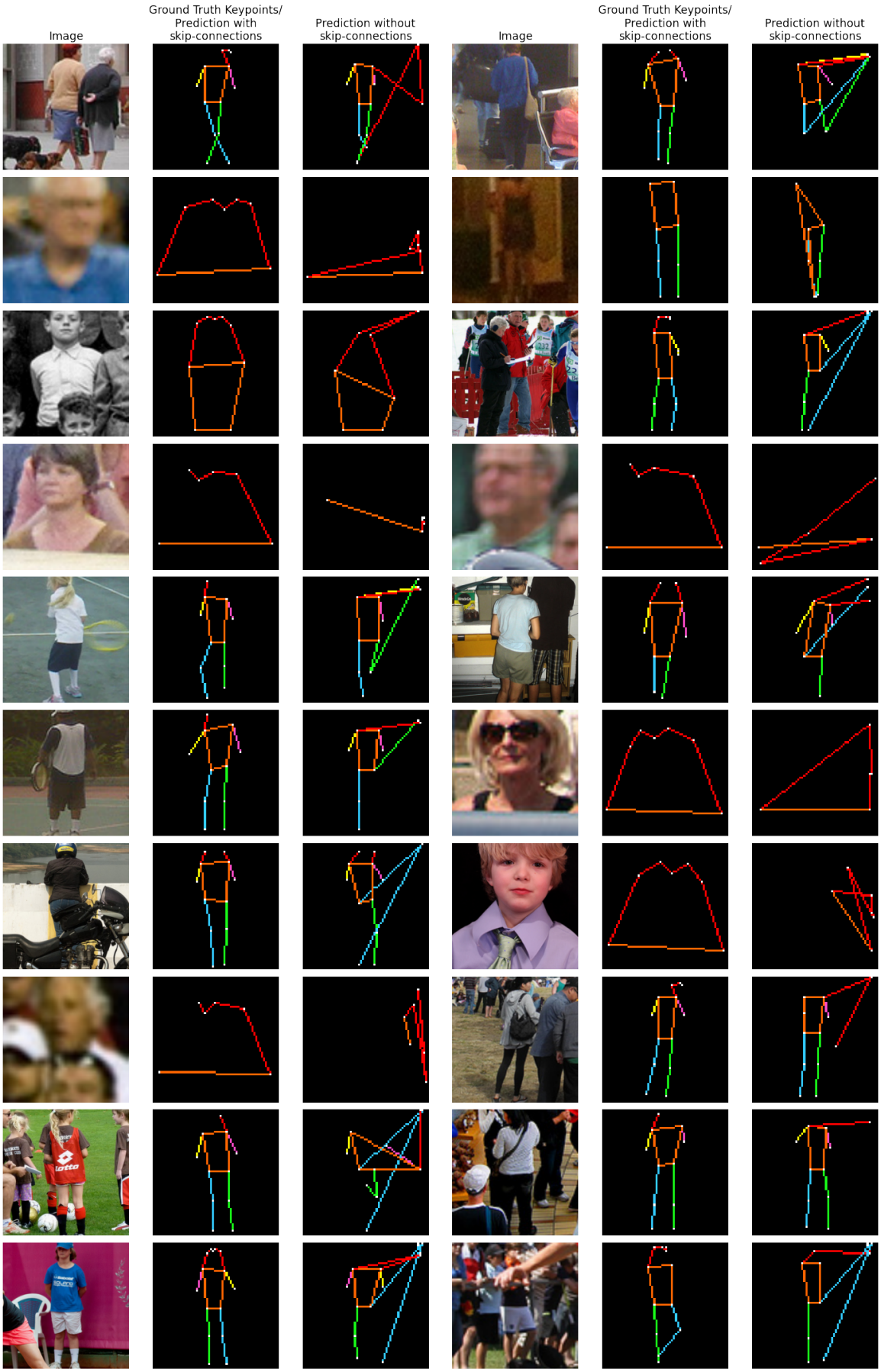


Figure 14: 20 samples of images correctly predicted by the model with skip-connections enabled, the corresponding ground truth heatmap and predictions by the model with skip-connections disabled.

we ensure, that we are not walking too far, ending up with misleading visualizations due to long distance between the end point to the nearest observation. Some of the results have been visualized in Figure 15.

By looking at Figure 15 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 because 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 component 50, which explains 0.165% of the variance in the data, and the following principal components 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 necessary.

6.4 Using Clustering to Seperate the Latent Space

Similar to subsection 6.3, 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 6.3, 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 16. The plot only explains about 37% of the variance of the original data, however, we 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 the model seperates the data in the latent space, we will be using K-Means. 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 is visualized in Figure 17a. By looking at Figure 17a we can clearly see, how the optimal K for the model is when $K = 2$.

The results of running the K-Means model with $K = 2$ is visualized in Figure 18. The K-Means model was run on the latent space in all of the 4.096 dimensions and only each cluster

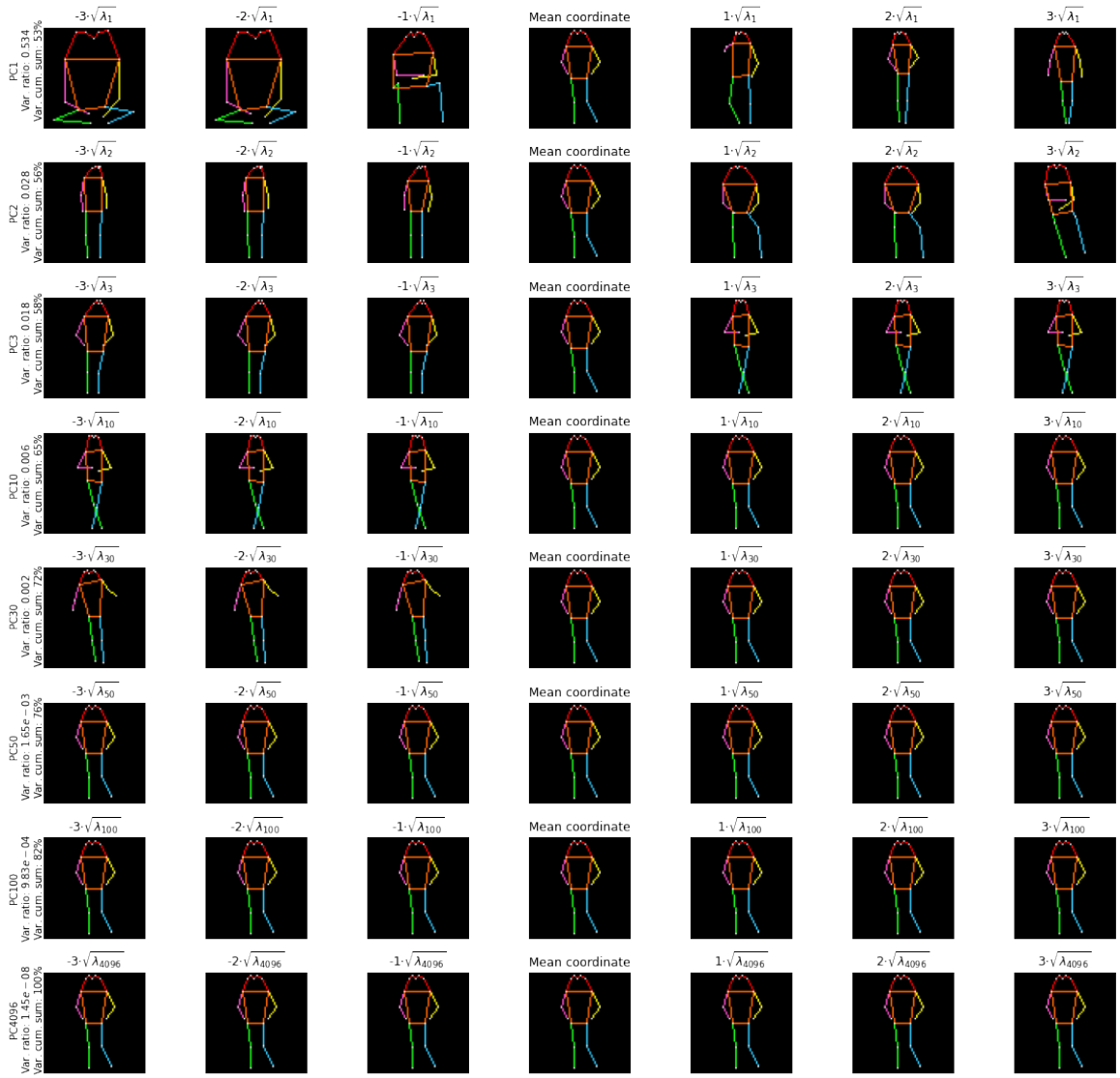


Figure 15: Nearest observations to the end point of "walking" along various principal components with varying step sizes.

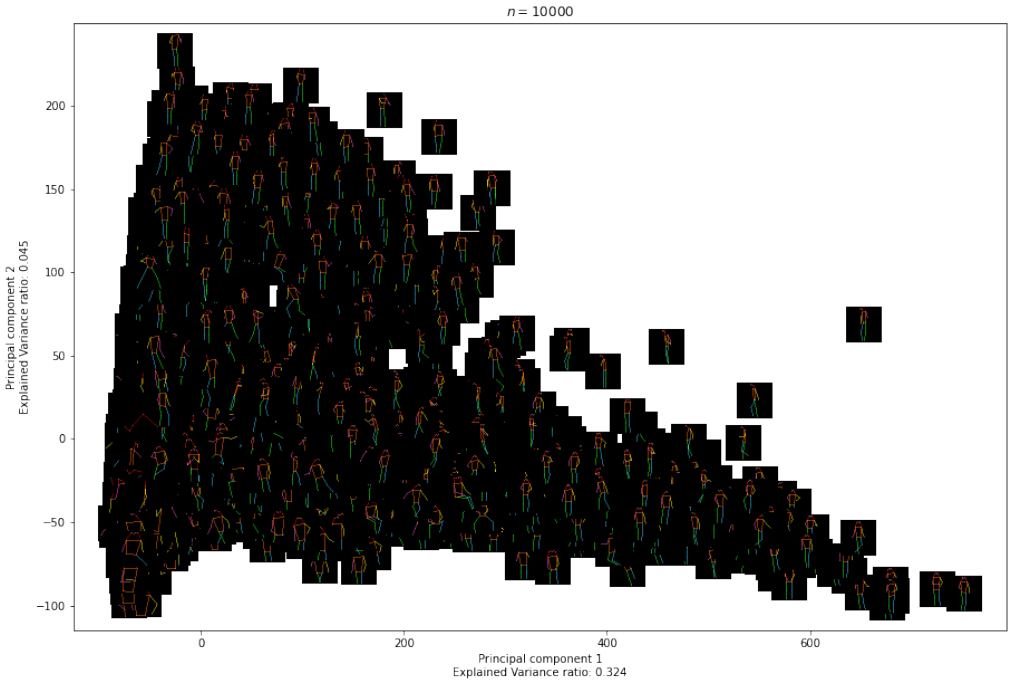
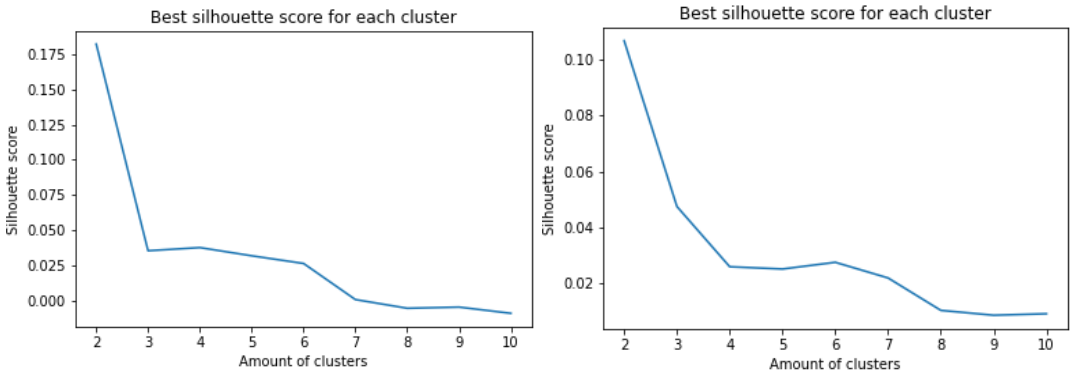


Figure 16: Plot of 10.000 samples of the latent space of the trained model, with the corresponding ground truth heatmaps



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

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

Figure 17: Silhouette score of running various K-Means models on different data from the latent space.

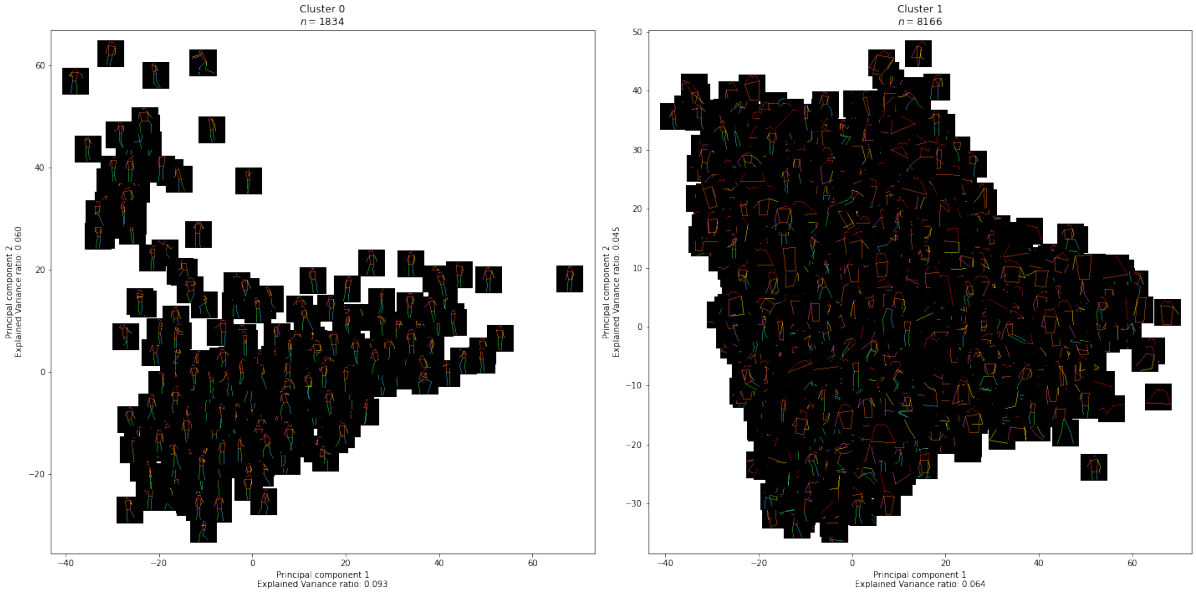


Figure 18: The resulting clusters of running a K-Means model with $K = 2$ on the latent space consisting of 10.000 random training samples

were projected down to 2 dimensions for the purpose of visualization. By looking at Figure 18 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 19. 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.

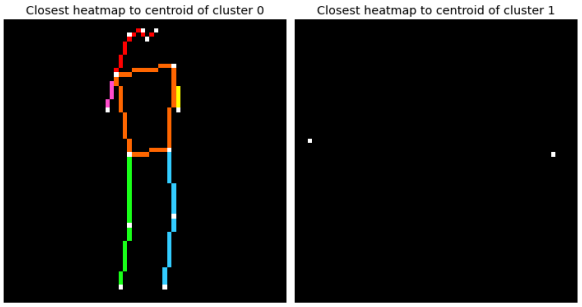


Figure 19: Closest points to the centroids of the two clusters from running K-Means on the latent space consisting of 10.000 random training samples

Although there are differences in the two clusters in Figure 18, there are still quite a lot of misclassified 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 17b, 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 20 and the corresponding closest ground truth heatmaps for the samples closest to the

centroids have been visualized in Figure 21. 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 21, 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 18. 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.

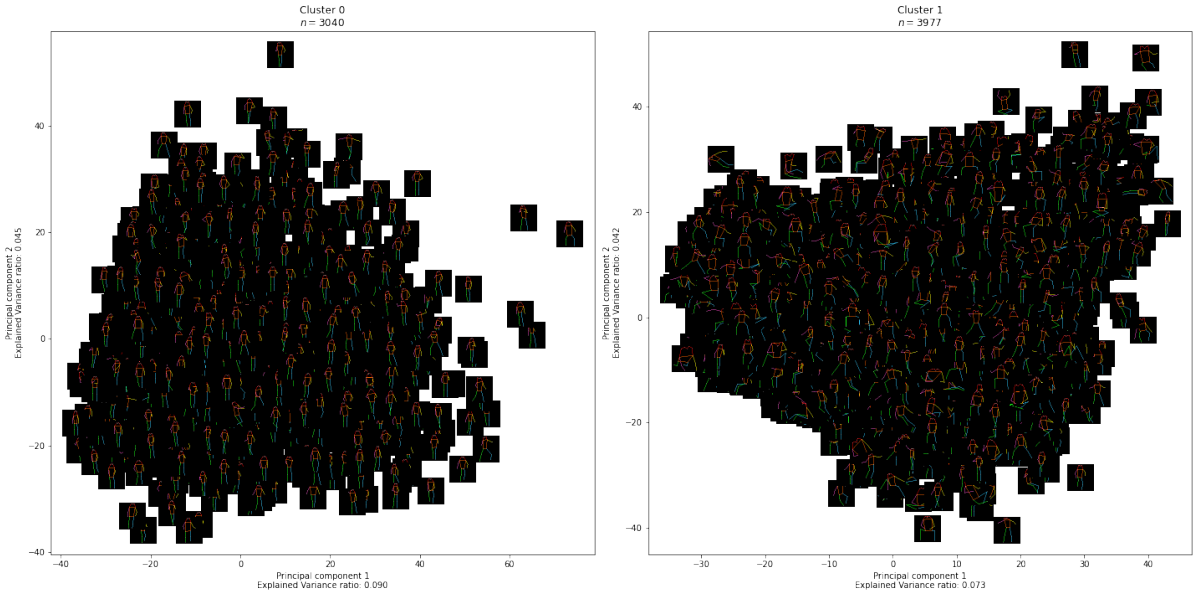


Figure 20: The resulting clusters of running a K-Means model with $K = 2$ on the latent space consisting of 7.017 fully-annotated training samples

All in all we have learned, that the model, during training, has learned to seperate almost fully-annotated people and not fully-annotated people, as well as learned to seperate stationary people and moving people. However, the seperations are not perfect, as there are some missclassifications, hinting towards the inaccuracies of the model.

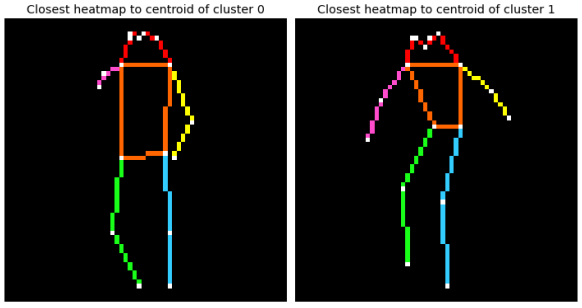


Figure 21: Closest points to the centroids of the two clusters from running K-Means on the latent space consisting of 7.017 fully-annotated training samples

7 Improving the Model

In the following section we will use our knowledge about the developed model to modify and improve the performance of the Stacked hourglass. We will start off in Section 7.1, where we will be giving a brief overview of how the model will be improved. Then, in Section 7.2 the configuration details of the model will be explained and argued for. Lastly, in Section 7.4 the various training details will be described.

7.1 Motivation

In Section 6 we explored the developed model from Section 5. 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 act 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.

7.2 Configuration Details

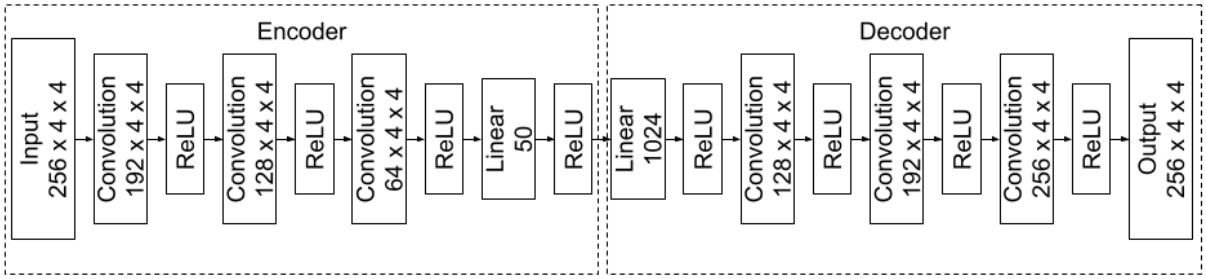


Figure 22: Visualization of the architecture of the developed autoencoder. The numbers in each box represents the dimensions of the output of the corresponding layer.

The developed autoencoder is visualized in Figure 22. The autoencoder makes use of convolutional layers and a linear layer to downsample the input down to only 50 dimensions. We chose 50 as the dimensions of the bottleneck, as we saw in Section 6.3, that principal component 50 and above act as noise, indicating that we need 50 or less dimensions. The autoencoder uses an decoder that is identical to the encoder, however flipped, to produce the final predictions.

The input of the autoencoder has the dimensions of $256 \times 4 \times 4$. As the width and height of the input already has a size of only 4×4, it does not make sense to use max pooling or similar methods to reduce this. Instead, the autoencoder only reduces the 256 layers of the input. This is done by making use of convolutional layers, that should learn the best way to reduce the amount of layers in the data. The innermost layers use linear layers instead of convolutional layers, as they make it easier to control the dimensions of the latent space, compared to the case with convolutional layers. We decided to use ReLU as the acitvation function of the autoencoder, as ReLU has some nice properties, which we described in Section 2.4, as well as it makes the autoencoder be in the same range as 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 23.

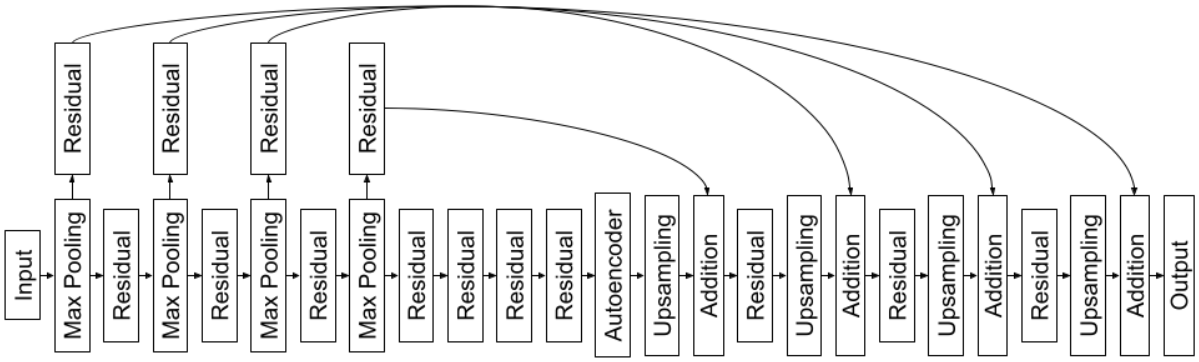


Figure 23: Visualization of the proposed combination of the hourglass and autoencoder for the Stacked hourglass.

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 23, 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, a learning rate of $5e-4$, which is halved every 25th epoch and a mini-batch size of 16. To increase the robustness of the autoencoder, we add noise sampled from

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

to each input training sample, where x is the value of the training sample. To help the model converge, we initialize all parameters by sampling from a Glorot normal distribution, like in the case with the Stacked hourglass.

After the autoencoder has been trained it is placed in the hourglass of the developed Stacked hourglass. The modified Stacked hourglass is then trained by following Newell *et al.* [15] as described in Section 5.1.

7.3 Results

By training the autoencoder isolated, we get the evolution of the training- and validation loss visualized in Figure 24. 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 Stacked hourglass with the autoencoder is visualized in Figure 25. 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 in a 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 model for compairson, we evaluate our modified Stacked hourglass on the held-out testing set from Section 4. By doing so we get a PCK score of 0.474, which is based on all annotated keypoints (that is, both for $v = 1$ and $v = 2$). Compared to the testing PCK accuracy of the standard Stacked hourglass, developed in Section **??**, the Stacked hourglass gets a performance increase of 7.48% or 0.033, by making use of an autoencoder.

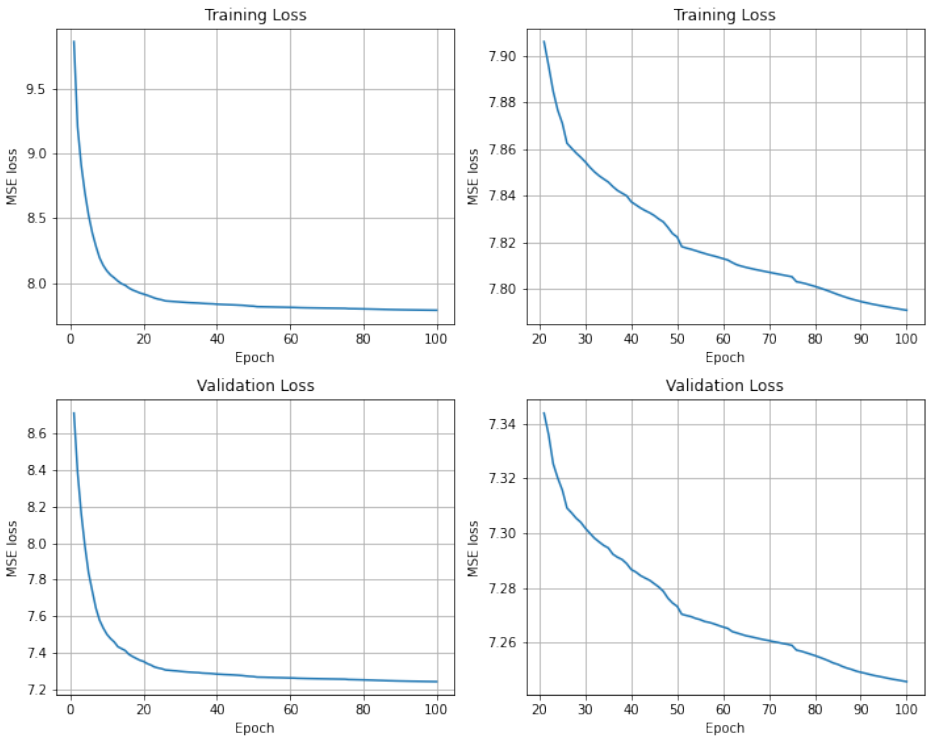


Figure 24: Visualization of the evolution of the training- and validation loss of the autoencoder during training. The left column shows all of the 100 epochs. Right column shows epoch 21 and foward

7.4 Training Details

Training of the autoencoder, as well as the modified Stacked hourglass, follow the training details described in Section 5.3. Training the autoencoder takes about 2 minutes per epoch, whereas the modified Stacked hourglass takes about 70 minuttes per epoch.

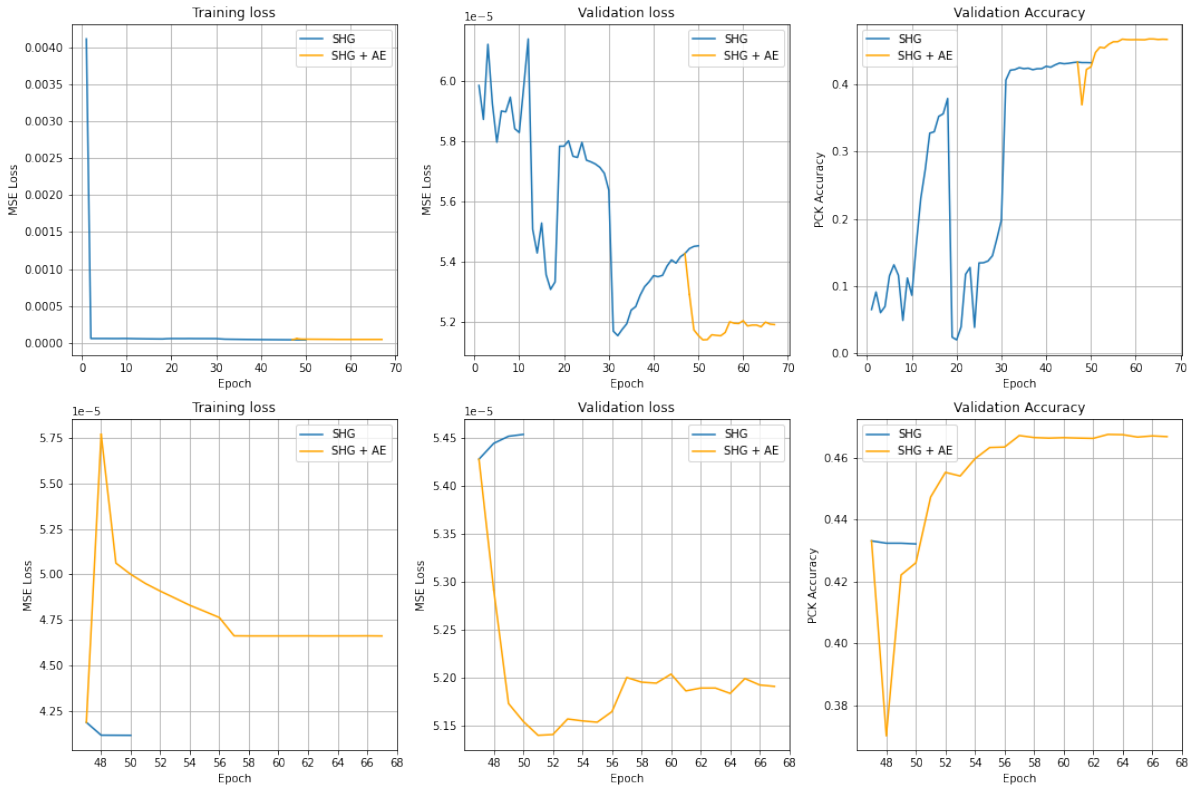


Figure 25: Visualization of the evolution of the training- and validation loss, as well as the PCK validation accuracy of the combination of the Stacked hourglass and autoencoder, compared with the evolution of training the orignial Stacked hourglass. The top row is of all of the 57 epochs. The bottom row shows epoch 47 and forward.

8 Discussion

In the following we will be discussing our procedure as well as our results. We will start off in Section 8.1, where we will be giving a brief summary of our obtained results from Section 5, 6 and 7. We will then in Section 8.2 compare our results with the results of Newell *et al.* [15] and Olsen [18], as well as discuss the differences in the results. In Section 8.3 we will argue and discuss why the autoencoder improved our initial model. Lasty, we will in Section 8.4 comment on potential future work in relation to this project.

8.1 Summary of the Obtained Results

In Section 5, we succefully implemented and trained a Stacked hourglass, consisting of a single hourglass. We did this by following the configuration details described by Newell *et al.* [15] and Olsen [18]. The developed model has a validation PCK accuracy of 0.433 and a test PCK accuracy of 0.441.

In Section 6 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 claimed by Newell *et al.* [15] and Olsen [18]. 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 first principal component of the latent space is used for deciding if a given person is sitting down or standing up,, as well as possibly discovering some redundancy in the model, as principal componet 50 and above seemed to act as noise. Lastly, we clustered the latent space 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 groupings are not always correct.

In Section 7, 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. We decided to make use of an autoencoder, as we knew from the previous section, that the latent space of the Stacked hourglass contained a lot of noise, as well as some missclassifications, which the autoencoder possibly could help on. By doing so, both the validation and test PCK accuracy increased to 0.467 and 0.473, respectively.

8.2 Comparison of Models

Description	# stacks	Testing accuracy
Olsen - $2A$	2	0.72
Olsen - $2B$	2	0.81
Olsen - $2M$	2	0.83
Our SHG	1	0.469
Our modified SHG	**1**	**0.576**

Table 3: Comparison of our developled models with Olsens developed model [18]. The testing accuracy is only based on fully-visible joints (that is, where $v = 2$)

To further understand the performance of our developed models, we have in Table 3 compared our models with the models developed by Olsen [18], which consisted of 2 stacks.

By looking at Table 3 we see, that the performances of the models developed by Olsen [18] exceeds the performances of our developed models. However, we also see, that all of Olsen's [18] models use more stacks, than our models, which could explain the better performance. This is also argued by Newell *et al.*, as they describe how stacking multiple hourglasses, increases the performance of the model [15].

Another reasoning behind the loss of performance compared with the models developed by Olsen [18] is, that Olsen's models were only trained on fully-visible joints [18], whereas our models were trained on both fully-visible joints and on non-visible joints. This could make Olsen's models [18] perform better on fully-visible joints, as this probably makes the model more focused on learning fully-visible joints during training. We would thus argue, that if we were to retrain our models on only fully-visible joints, the gap between the performance of our models and Olsen's models [18] would become smaller, and even very small in the case of the modified Stacked hourglass. This is however not possible to predict and thus only purely speculation, however, our thought experiment does support this claim.

Olsen's models and data are more similar to ours, than the ones used by Newell *et al.* [15], making the comparison more accurate. However, we can still compare our results with the ones obtained by Newell *et al.* [15] to get a further performance evaluation of our models.

Like in our case, Newelll *et al.* do also develop a model consisting of only a single hourglass. However, Newell *et al.* does not report the testing accuracy of this model. Instead, they report the evolution of the validation accuracy during training, for joints that are not associated with the head and torso. The validation accuracy seems to top at around 0.65 [15]. If we test our models on the joints from our testing data, that are not associated with the head or torso, we get a PCK accuracy of 0.32 and 0.38 for the Stacked hourglass and the modified Stacked hourglass, respectively.

There are a number of reasons that could explain the poor performance of our models compared with the models developed by Newell *et al.* [15].

1. We use a different dataset than Newell *et al.* [15]: Newell *et al.* uses the FLIC [25] and MPII Human Pose [1] datasets, whereas we use the 2017 Microsoft COCO dataset [14]. These datasets could potentially be different from the dataset that we use, making it an unfair comparison to compare our results with Newell *et al.*'s results [15]. For instance, the FLIC [25] and MPII Human Pose [1] could contain less occlusion than the Microsoft COCO dataset [14], which would decrease the performance of our model, as argued by Newell *et al.* [15].

2. Our preprocessing of not-visible joints was different than the preprocessing done by Newell *et al.* [15]: when we preprocessed the data, we made use of a Guassian filter to represent uncertainty in the annotation. For not-visible joints we used a standard deviation of 0.5, whereas we for visible joints used a standard deviation of 1. Newell *et al.* describes, how they for all joints used a standard deviation of 1 [15]. We decided to use a standard deviation of 0.5 for not-visible joints, as the uncertainty of the annotation of these joints is greater, than for visible joints. This could however also possibly have confused the model, as it thus implicitly also learns to predict whether a given joint is visible or not, making the complexity of the model greater, possibly decreasing the performance of the model.

3. We use a different PCK accuracy metric: Newell *et al.* state, that they use different PCK accuracy metrics - that is PCK accuracy metrics, where the normalization constant is different - at different times [15], however, they do not state which PCK accuracy metric

they used for validation during training. We chose a normalization constant equal to one tenth of the heatmap size, however, if Newell *et al.* [15] used a greater normalization constant, the predictions would have a bigger probability of being accepted as correct, making the accuracy of the developed model seem greater than it actually is. Likewise, Newell *et al.* [15] do not state the threshold radius they use. If they were to use a threshold radius greater than the one we use, their estimations have a greater probability of being counted as correct, than our estimations.

4. Newell *et al.* describes that they use batch normalizations to prevent the model from overfitting, however, they do not describe where they used these batch normalizations [15]. If we were to place our batch normalizations at places different than where Newell *et al.* [15] place theirs, we could make our model overfit much quicker than what Newell *et al.* [15] experiences, resulting in worse performance. This, however, is much less likely, as we decided to follow Olsen [18] for the placement of batch normalization and they achieved great results, hinting towards their, and thus also our, placements of the batch normalizations being correct.

5. We use different configurations: Newell *et al.* [15] does not state the decay rate ρ for RMSProp. We decided to use a decay rate of 0.99, as this is the default value in PyTorch, resulting in a slow decay of the accumulated squared gradient $\boldsymbol{r}$. If Newell *et al.* [15] use a different value for the decay rate, $\boldsymbol{r}$ would most likely decay differently, potentially resulting in different results. Likewise, Newell *et al.* [15] does not state how they initialize the parameters of their model, hence why, the differences in performance could be explained by a potentiel difference in how they initialize the parameters in their model and in how we initialize the parameters in our model. This is however, not likely to have a big impact on the results, as Olsen [18] performed the same method for initializing the parameters of their model as we did and received great results, making us believe, that our method for initialization should not be a problem.

Overall, the comparison between our developed model and the model developed by Newell *et al.* [15] is difficult, as they have a lot of differences. The comparison between our model and Olsen's model [18] is thus more accurate, however, still inaccurate, as Olsen's models use 2 stacks instead of just 1 as in our case [18].

8.3 Why did the Autoencoder Improve the Stacked hourglass?

In Section 7.1 we argued the following two reasons for why using the autoencoder could improve the performance of the model: (1) inconsistency of the placement of the training observations, and (2) having principal components that acts as noise. In the following we will be analysing and discussing our results of using the autoencoder, leading to if these two reasons actually had an effect on the performance of the model.

8.3.1 Clustering the Latent Space

In Figure 26, we have visualized the results of performing K-Means on the latent space of the autoencoder with $K = 2$. Due to memory limitations only 10.000 training samples were used.

In Figure 18 we visualized equivalent results of performing the same procedure on the latent space of the Stacked hourglass. In correlation to Figure 18 we argued, that the two clusters contained some missclassifications, which could explain the poor performance of the model. More specifically, *Cluster 1*, the cluster containing samples with a lot of keypoints missing, contained a lot of samples that should ideally have been in *Cluster 0*.

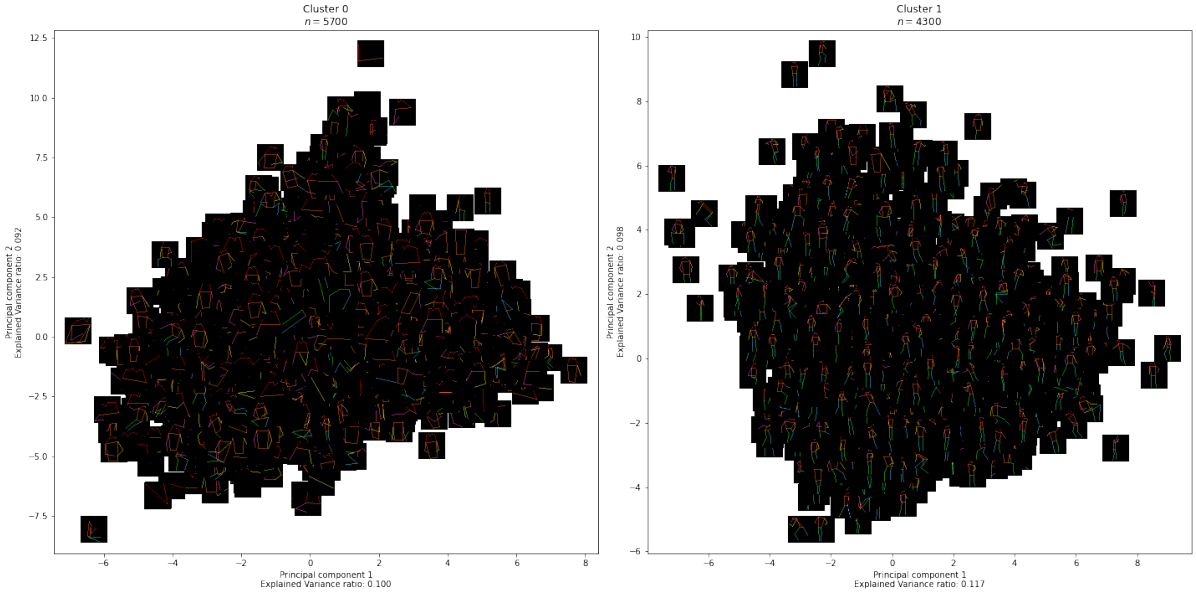
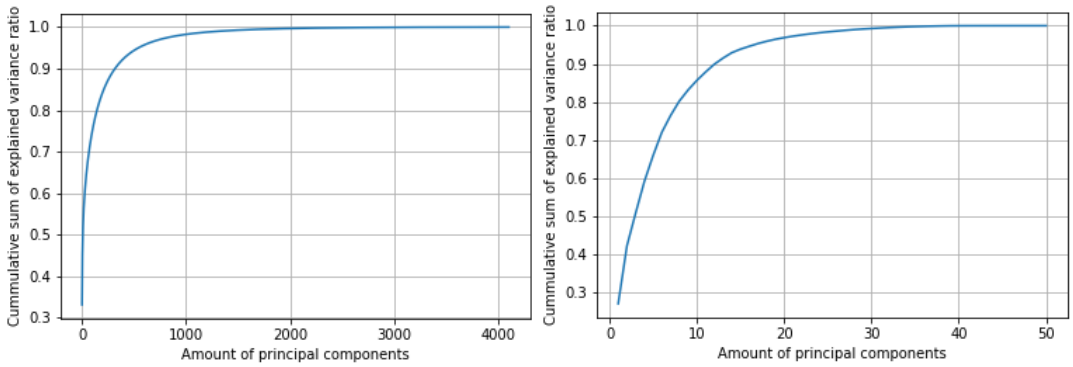


Figure 26: Results of performing K-Means clustering on 10.000 random samples of the latent space of the autoencoder

If we compare Figure 26 with Figure 18 we can see, that the clusters in Figure 26 contains fewer missclassifications than what was the case in 18. More specifically we can see, that *Cluster 0* in 26, the cluster containing samples with a lot of keypoints missing, contains fewer samples that should have been in *Cluster 1*, than what was the case earlier. Likewise, the distribution of the amount of samples in each cluster is also more balanced, leading to believe that fewer missclassifications are indeed happening.

On the other hand, the 10.000 samples used in Figure 26 and 18 are not the same. Both figures contains 10.000 random samples out of the total 124.040 samples, meaning the two figures most likely are not alike, however, probably have some overlap. This leads to us believe, that the two figures are not exactly comparable, however, will still give a decent insight into the clustering of the two latent spaces.

8.3.2 Removing Noisy Features from the Latent Space



(a) Stacked hourglass (b) Autoencoder

Figure 27: Cummulative sum of the explained variance ratio per principal component of the latent space of the Stacked hourglass and autoencoder, respectively

In Section 6.3 we argued, that the latent space of the developed Stacked hourglass contained a lot of principal components that acted as noise. By using an autoencoder this noise should be removed, since the autoencoder learns the most important features of the input data, as the output of the autoencoder is only an approximation of the input.

In Figure 27 we have visualized the cummulative sum of the explained variance ratio per principal component of the latent space of the Stacked hourglass and the autoencoder for the training data. We can clearly see how the autoencoder needs a lot less principal components to represent the data, than the Stacked hourglass. If we assume, that 5% of variance of the data acts as noise, the autoencoder needs 16 principal components to explain the remaining 95% of the variance of the data, where the Stacked hourglass needs a total of 541. This means, that by using the autoencoder we have removed a lot of redundancy in the form of noise, essentially denoising the data, which could explain the improvement of the performance.

To sum up, it seems, that we by using the autoencoder in the Stacked hourglass, have improved the structure of the latent space of the model, as well as removed the noise of the latent space, which seems to have improved the performance, as argued in Section 7.1.

8.4 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 [15], 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.* [15] achieves with 4 hourglasses. In correlation to this, it would also be ideal to examine if any redundancy is added when multiple modified hourglasses are stacked. For instance, if we were to stack 2 modified hourglasses, maybe only 25 filters are needed in the second hourglass.

Secondly, one could check for redundancy in the other parts of the model. In Section 6 and Section 7, we found redundancy in the bottleneck of the model, however, we did not search for any redundancy in the other parts of the model, such as the encoder, decoder and skip-connections, leaving some potential future work. In addition to this, one could retrain a new Stacked hourglass without skip-connections, unilke our procedure in Section 6.2, where we simply disabled the skip-connections of an already trained network. This would give less biased results, as well as better insight into the effects of the skip-connects, in comparison to our procedure.

Lastly, it would be interesting to get an understanding of how much of a performance increase one could obtain by using the autoencoder in the hourglass. We described in Section 8.3, how the latent space of the autoencoder still has some missclassifications, hinting towards some potential increase of performance. If one could correct these missclassifications, how much of an increase in performance would this result in?

9 Conclusion

We have succefully implemented and trained a Stacked hourglass, developed by Newell *et al.* [15], consisting of a single hourglass. The network was trained, validated and tested on the 2017 Microsoft COCO dataset [14]. 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 inaccuracies of the model. Lastly, we used our obtained knowledge about the model to succefully improve the performance of the developed model, by modifying the model to make use of an autoencoder.

10 References

[1] Mykhaylo Andriluka1, Leonid Pishchulin, Peter Gehler, and Bernt Schiele. *2D Human Pose Estimation: New Benchmark and State of the Art Analysis*. Tech. rep. Max Planck Institute for Informatics, Max Planck Institute for Intelligent Systems, and Stanford University, 2014.

[2] Zhe Cao, gines Hidalgo, Tomas Simon, Shih-En Wei, and Yaser Sheikh. *OpenPose: Realtime Multi-Person 2D Pose Estimation using Part Affinity Fields*. Tech. rep. IEEE, 2019.

[3] João Carreira, Pulkit Agrawal, Katerina Fragkiadaki, and Jitendra Malik. *Human Pose Estimation with Iterative Error Feedback*. Tech. rep. UC Berkeley, 2016.

[4] 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.

[5] Dumitru Erhan, Yoshua Bengio, Aaron Courville, and Pascal Vincent. *Visualizing Higher-Layer Features of a Deep Network*. Tech. rep. Département d'Informatique et Recherche Opérationelle, 2009.

[6] 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).

[7] Pedro Felzenszwalb, David McAllester, and Deva Ramanan. *A Discriminatively Trained, Multiscale, Deformable Part Model*. Tech. rep. University of Chicago, Toyota Technological Institute at Chicago and UC Irvine, 2008.

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

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

[10] 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).

[11] 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).

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

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

[14] 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.

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

[16] Andrew Ng. *C4W2L04 Why ResNets Work*. DeepLearningAI. URL: https://www.youtube.com/watch?v=RYth6EbBUqM.

[17] 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.

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

[19] 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).

[20] 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.

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

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

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

[24] 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).

[25] Ben Sapp and Ben Taskar. *MODEC: Multimodal Decomposable Models for Human Pose Estimation*. Tech. rep. Google Inc and University of Washington, 2013.

[26] 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.

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

[28] Alexander Toshev and Christian Szegedy. *DeepPose: Human Pose Estimation via Deep Neural Networks*. Tech. rep. Google, 2014.

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

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

[31] Matthew D. Zeiler and Rob Fergus. *Visualizing and Understanding Convolutional Networks*. Tech. rep. Department of Computer Science, New York University, 2014.

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

[33] 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.