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`

May 7, 2021

Source code available at
https://github.com/KeenbitGitHub/bachelor_thesis.git

Supervisor
Kim Steenstrup Pedersen `kimstp@di.ku.dk`

Contents

1 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 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.
- $\nabla_x f$: Gradient of f with respect to 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

2 Machine Learning Theory

Throughout this section the theory of machine learning that will be used in this thesis is described and explained.

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

2.2 Machine Learning Paradigms

Machine learning is usally split into the following three paradigms

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

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

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

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

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 unseen data, which is done by making use of *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 random non-overlapping chunks of equal size. The model is then trained for K rounds on $K-1$ of the chunks, where the last chunk is used for evaluating the model. After each round the parameters of the model is reset to ensure one round does not affect another round. After the K rounds the average loss of the K rounds is the loss of the model [15].

2. *Train-validation-test* where the data is split into 3 random non-overlapping chunks. The training dataset is then used for training the model and the validation dataset is used for evaluating the model as it is being developed - this often means, that the *hyperparameters*, the paramters that are not possible to fit from the data, are being tweaked to yield the best validation loss. Lastly, the testing dataset is used as a final evaluation of the model to yield an unbiased evaluation of the model. Once the testing dataset has been used it can no longer be used for evaluating the data, as this ensure an unbiased evaluation [6].

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 (Loss Functions)

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

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

where y_i is the true value of the ith observation and $\hat{y}_i$ is the estimated value of the ith observation. Thus, MSE measures the average squared difference between the true observation and the estimated observation. The aim of a model is thus to make the MSE as small as possible [7].

2.4 Neural Networks

In recent years *deep learning* and *neural networks* have revolutionized the use of machine learning. In this thesis a neural network will be used for performing the human pose estimation. Throughout subsection 2.4 the theory and mathematics behind neural networks is described and explained.

2.4.1 Feedforward Neural Networks

Algorithm 1 GradientDescent [20]

```
Require: Learning rate $\eta$
Require: Starting position $\boldsymbol{\theta}$
1: while stopping criterion not met do
2:     Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\nabla f(\boldsymbol{\theta})$
3: return $\boldsymbol{\theta}$
```

Overfitting and Regularization

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

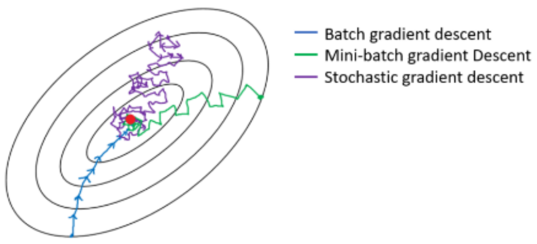


Figure 1: Comparison of batch, mini-batch and stochastic gradient descent [12]

Gradient Descent
The goal of a machine learning model when training is to minimize its loss. There are different methods to do so, however, the most common algorithms are variants of *gradient descent*, whose algorithm is described in Algorithm 1. The algorithm works by taking a learning rate η, a starting position $\boldsymbol{\theta}$ and a function f as input, where f is the function to minimize. It then computes the gradient of f with respect to $\boldsymbol{\theta}$, and subtracts the gradient times η from $\boldsymbol{\theta}$. This is done untill a stopping condition is met, such as when the magnitude of the gradient $|\nabla f(\boldsymbol{\theta})|$ is small or untill a maximum amount of iterations has been reached [20].

Online, Mini-batch and Batch methods
When gradient descent is used in machine learning, computing $\nabla f(\boldsymbol{x})$ is usually done by averaging the gradient of each of the n observations of the trainingset, which is called a *batch gradient method* and is computational inefficient, as the cost is $\mathcal{O}(n)$. It is therefore common to use variants of gradient descents, that reduces the cost of computing the gradient. In *online gradient methods* (also known as *stochastic gradient descent*) 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 [20].
Choosing the right batch size can be difficult, however, there are a few guidelines which one can follow [12] [20]

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

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

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

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

Algorithm 2 RMSProp [4]

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

that it can converge after finding a convex bowl. The algorithm uses a hyperparameter ρ, that controls the length scale of the moving average [4].

Batch Normalization

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

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

where

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

and

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

which makes the minibatch have 0 mean and unit variance. $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are then used to make the mini-batch have an arbitrary mean and standard deviation and are two parameters that needs to be learned when the network is being fitted. This helps the network to converge, as the batch normalization keeps centering the mean and standard deviation of the mini-batches [20].

Epoch

An *epoch* is an iteration through the whole dataset during fitting of the network. Multiple epochs are often needed to reach the minimum of the loss function [20].

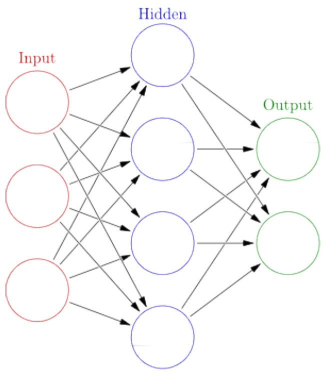


Figure 2: Visualization of a feedforward neural network with a single hidden layer [19]

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 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 an *activation function*, that is typically applied element-wise [4] [17]. One often used activation function is the *rectified linear activation function* (or *ReLU* for short) defined by

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

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

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

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

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

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

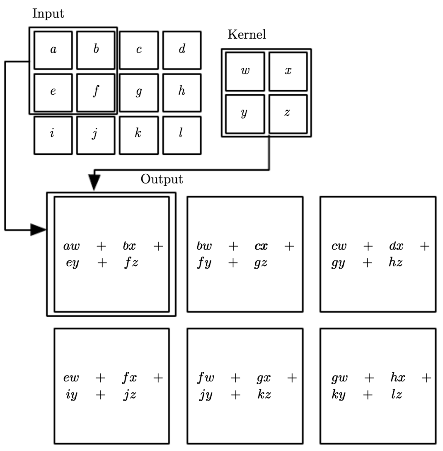
and the partial derivative of each bias is

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

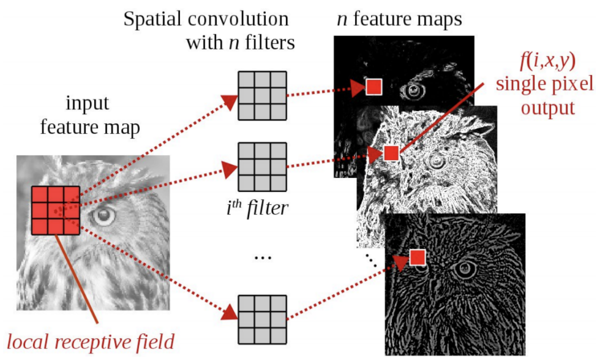
where for both cases

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

if $\boldsymbol{a}^{(i)}$ is not the output-layer. Once the partial derivative of all weights and biases has been found, the gradient vector can be formed and an optimization method can be used to optimize the parameters of the model [16].



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



(b) Example of a convolution with multiple filters [13]

Figure 3: Convolutions visualized

2.4.2 Convolutional Neural Networks

Feedforward neural networks introduced in 2.4.1 can be used for pattern recognition within images, however, they are usually not used for this task. Consider a colored input image of dimension 64×64. If we were to use a feedforward neural network on this image, each neuron in the first hidden layer would be connected to the input layer through 12.288 weights. Not only would this use a lot of computational power and time to train, however, a network of this size would also be prone to overfitting. Instead, *convolutional neural networks* (also known as CNN's) are usually used. A CNN usually consists of *convolutional layers*, *pooling layers* and *fully-connected layers*, where the fully-connected layers are analogous to the layers in a feedforward neural network [10].

Convolutional layers
A convolutional layer is composed of a set of *kernels* (also known as *filters*), which are matrices of weights of dimension $k \times k$, where k usually is 5, 3 or 1, and each weight is a parameter for the model to be learned [13]. Each kernel is used on the input to produce a *feature map*. The kernels are applied to the input by "sliding" over the input (where the step size is called *stride* and is usually by default equal to 1). Each $k \times k$ grid of the input (called the *local receptive field*) is then used to compute the dot-product between the grid and each kernel, which is then placed in the corresponding feature map of each kernel, as visualized in Figure 3. When all of the feature maps have been computed, the feature maps are stacked together, forming a tensor, which is then returned by the layer and a activation function can be applied.
As described previously, by using a convolutional layer we can dramatically decrease the amount of weights used by the layer. If we were to use a 3×3 kernel on a colored image, we would reduce the amount of weights on each neuron in the following layer from 12.288 down to just 27, reducing both the training time and making the network less prone to overfitting [10].

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 [10].

Algorithm 3 Nearest Neighbour Upsampling [14]

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

Nearest Neighbour Upsampling

Sometimes we want to increase the size of an image. This is 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 3. The algorithm starts off by taking an image, X, of size $m \times n$, as input, which we wish to upsample to size $xm \times yn$. The algorithm then loops over each pixel, p, in X, finds the corresponding index, i, j, of p in X, and places p at index (xi, yj) in the output image O of size $xm \times yn$. When this is done, it assigns each of the empty pixel in O the value of their nearest neighbour, making each pixel in O have a value, and then returns O [14].

3 Stacked Hourglass, PCA and K-Means

In this section the various algorithms and architectures used throughout this thesis is described and explained in details.

3.1 Stacked Hourglass

When performing the pose estimation in section **REFERENCE MANGLER**, we will be implementing and using the *stacked hourglass* described by Newell *et al.* [9]. The following description and explanation of the architecture is based on an interpretation of Newell *et al.* [9] and Camilla Olsen [11].

3.1.1 Reasoning behind using the Stacked Hourglass

We have decided to make use of the Stacked hourglass described by Newell *et al.*, as it is an architecture that has shown state-of-the-art results. At the same time the architecture of the network is similar to the architecture of *autoencoders*, making the architecture useful for encoding the data into a lower dimension, which can be useful in section **REFERENCE MANGLER**, when we will be doing the interpretation of the model.

3.1.2 The Residual Module

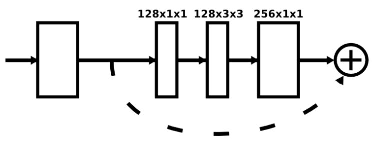


Figure 4: Visualization of the residual module [9]

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

3.1.3 The Hourglass

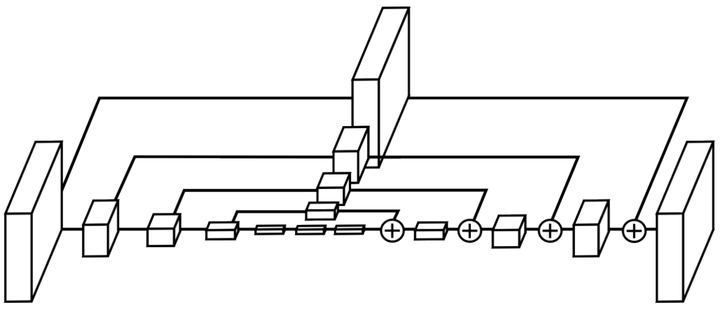


Figure 5: Visualization of a single hourglass [9]

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

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

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

3.1.4 The Stacked Hourglass

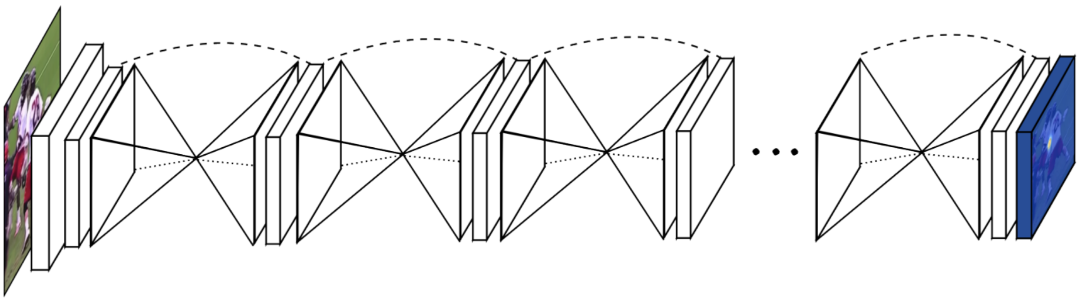


Figure 6: Visualization of the Stacked hourglass [9]

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

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

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

3.2 Principal Components Analysis (PCA)

It is very common, that a given dataset has an enormous amount of dimensions. This is quickly become a problem, as it can be difficult to visualize or it can lead to other problems, such as models becoming too complex, thus being prone to overfitting. This is a common phenomen called the *Curse of Dimensionality* [5]. For this reason multiple techniques have been developed

Algorithm 4 PCA [15]

```
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 $\boldsymbol{X}$
```

for reducing the dimensions of a given dataset. One of the most common techniques 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. [15]

The pseudocode of the algorithm has been visualized in Algorithm 4. 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. [15]

3.3 K-Means Clustering

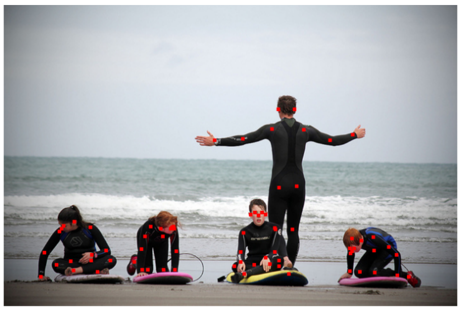
Often we want to find patterns in data that has not been labelled. One common group of techniques for this purpose is the *clustering algorithms*, that are used to group observations into clusters, such that observations from the same cluster are more similar, than observations from different clusters.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

K-Means is a unsupervised method used for clustering observations into K groups of similar observations, such that no observation occurs in multiple clusters. The algorithm uses distance as a measure of similarity, such that observations closer to each other are more likely to being grouped to the same cluster, than two observations far appart. In the middle of each cluster is a synthetic observation (that is, not a real observation), called the *centroid*, which is defined as the mean of the cluster. The pseudocode of the algorithm has been visualized in Algorithm 5. 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 [15].

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 [15].

Algorithm 5 K-Means [15]

```
Require: Input data $\boldsymbol{X}$
Require: Amount of clusters $K$
1: Let $\boldsymbol{\mu}_k$ be a matrix of the coordinates of the centroids of the $k$ clusters. Usually intially set with random values
2: Let $\boldsymbol{Z}_{nk}$ be a matrix of binary indicator variables that is $1$ if object $n$ is assigned to cluster $k$ and $0$ otherwise
3: while $\boldsymbol{Z}_{nk}$ is changed between each iteration do
4:     for each observations $\boldsymbol{x}_n \in \boldsymbol{X}$ do
5:         Find the centroid, $k$, that $\boldsymbol{x}_n$ is closest to
6:         Let $\boldsymbol{Z}_{nk} \leftarrow 1$ and $\boldsymbol{Z}_{nj} \leftarrow 0$ for all $j \neq k$
7:     Update centroids: $\boldsymbol{\mu}_k \leftarrow \frac{\sum_n \boldsymbol{Z}_{nk}\boldsymbol{x}_n}{\sum_n \boldsymbol{Z}_{nk}}$
```



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

Figure 7: Example of an image from the COCO dataset with the keypoints drawn on [8]

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 preprocessing are described.

4.1 The COCO Dataset

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 [8], which we will be using. The dataset contains annotations for different purposes, however, for our pose-estimation-task, only the keypoint annotations of human bodies are needed. An example of such a picture with the keypoints labeled can be seen in Figure 7.

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

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

The training and validation sets contains a total of about 123.000 various images. As we only need the images that contain humans, we will be discarding the images without any humans, leaving us with a total of about 66.808 images of humans doing various tasks, with a total of 149.813 humans annotated with keypoints. Each image can contain multiple people, which we need to handle before training our model, as we will be focusing on single-human pose estimation. Besides this, each image also has different resolution and aspect ratio, which we also need to handle, as our model requires the images to have a fixed resolution. Lastly, we should also do some handling of the labels before training the model, as there could have been some inaccuracies, when the joints were labeled. This especially applies when $v = 1$, that is, when the joint is labeled but not visible, as there are more inaccuracies or uncertainty when

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

Table 1: Data distribution

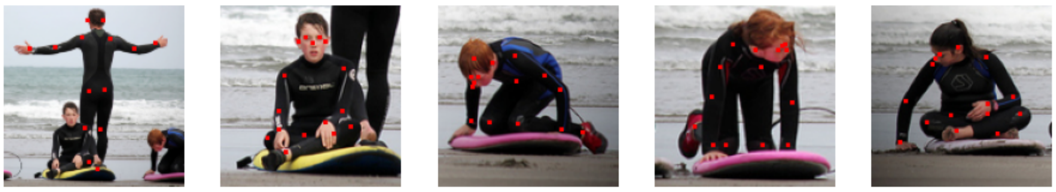


Figure 8: The results of processing the image from Figure 7 with the corresponding labels [8]

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

4.2.2 Preprocessing the images

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

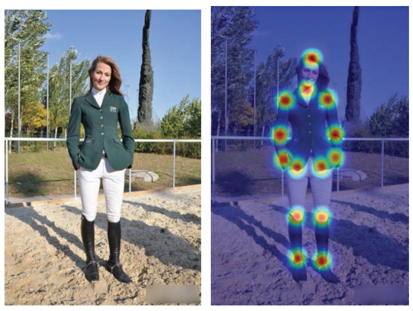
It is possible for each bounding box to contain multiple people. This is a problem, as it will confuse our model, since it will not know which person to annotate. An example of this can be seen in the first image of Figure 8. To fix this we center the bounding box around the person it should annotate,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

Since each keypoint does not necessarily lie on the edge of the person, the bounding boxes could result in not all of the pixels of the corresponding person being in the bounding box. For this reason, each bounding box is expanded with 10% in the height and width. If, however, the image cannot contain the expanded bounding box, the bounding box is then expanded as much as possible, while still being a square. If it is the case, that one of the corners of the bounding box lies outside of the image, then the bounding box is moved either up or down, making the corner of the bounding box be inside the image and keeping the annotated person centered along the x-axis.

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

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

4.2.3 Handling the labels



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

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

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

5 Experiment

Throughout the following section the results and configuration details of our trained model are described and explained.

5.1 Configuration Details

Our model only consists of a single hourglass. The hourglass consists of 4 down- and upsamples, with 1 residual module between each down- and upsample. In each skip-connection a single residual module is used, and in the bottleneck 3 residual modules are used. Newell *et al.* [9] and Olsen [11] 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 thesis 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 is neither to improve the model developed by Newell *et. al* [9], hence why we will be making the same configuration choices as Newell *et. al* [9] and Olsen [11], 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 [11] and perform the batch normalization before each convolutional layer in each residual module, after the first convolutional layer of the entire network and before the last convolutional layer of the entire network. For the choice of acitvation function we use the $ReLU$-function after each batch normalization. Each max pooling and nearest neighbor upsampling uses a kernel size of 2, which halves and doubles the size of the input, respectively. The full network has been visualized in Figure 10.

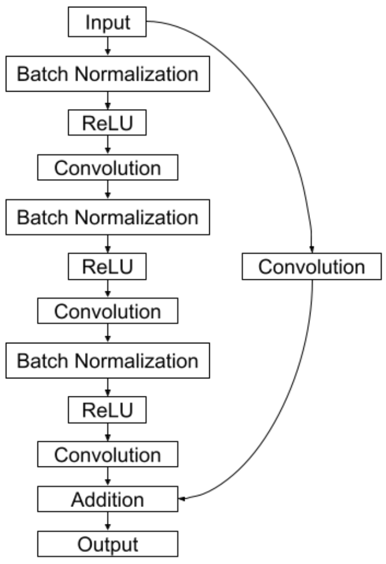
For the initial values of the weights we initialize each weight by sampling from a *Glorot normal distribution* (also known as a *Xavier normal distribution*), described as

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

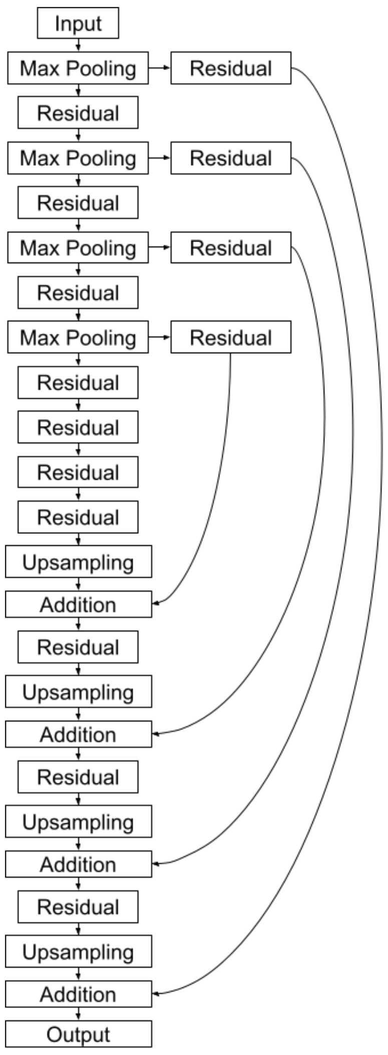
where fan_{in} is the amount of input connections and fan_{out} is the amount of output connections to the layer of the weight [3]. By doing so we make all layers have the same activation variance and gradient variance, essentially helping the model to converge [4].

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

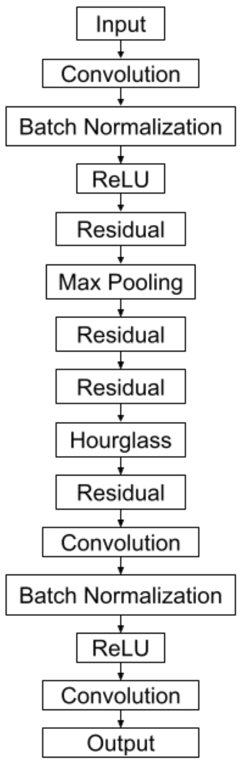
After each epoch we find the validation accuracy of the model by computing the *PCK* between the predictions and the ground truth of the validation dataset, as described by Olsen [11] and the source code of the Stacked Hourglass Network [18], where it is called *heatmap accuracy*. The pseudocode of PCK has been visualized in Algorithm 6. 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



(a) The Residual module



(b) The Hourglass



(c) The Full network

Figure 10: Overview of the used architecture

Algorithm 6 PCK [11][18]

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

truth heatmaps and the corresponding predicted heatmaps [11] [18]. The aim is thus to maximize the PCK accuracy. To produce the final PCK accuracy of the model, the PCK accuracy is computed for each image in the validation dataset. The mean PCK accuracy is then used as the PCK accuracy of the model. For the two constants, c and r, we let c be one tenth of the heatmap resolution size $\left(\text{that is, } \frac{64}{10} = 6.4\right)$ and r be 0.5.

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

5.2 Results

In Figure 11 the evolution of the training loss, validation loss and validation PCK accuracy has been visualized. The model were initially set to train for 100 epochs, however, we decided to stop the training early, as the model clearly started to overfit after 30 epochs, as seen by comparing the training and validation loss.

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

Comparing the training loss, validation loss and the validation accuracy from Table 2 we see, that the there is not an overlap between the models yielding the best training loss, validation loss and validation accuracy. As we in section **REFERENCE MANGLER** 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.

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

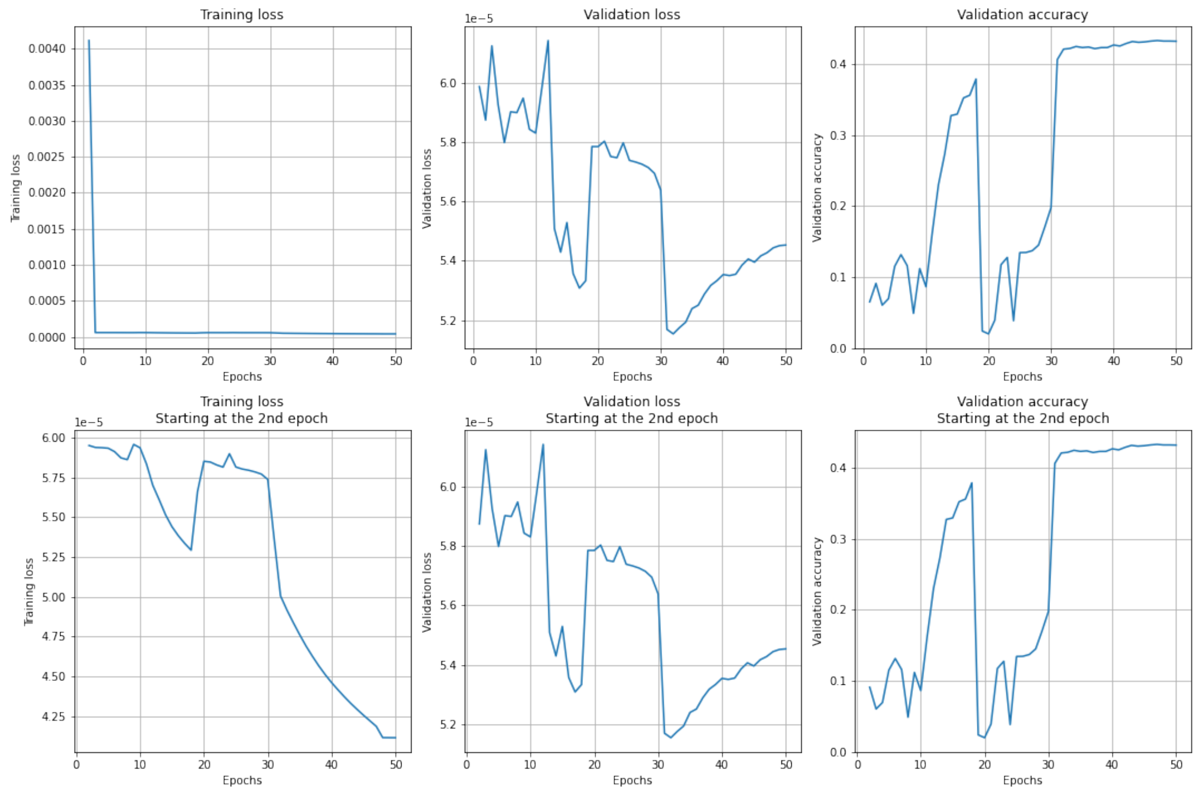


Figure 11: Visualization of the evolution of the training loss, validation loss and validation PCK accuracy of the trained model. Top row shows all of the 50 epochs. Bottom row shows epoch 2 and forward to ease the reading of the training loss

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

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.

6 Discussion

- PCK: Scaler ikke med input størrelse
- Glorot Initialisering - kan være dårligt, se EML forelæsning om optimization.
- Acc. var stadig stigende - måske skulle jeg lade den træne længere
- Måske skulle jeg have gjort brug af gap statistics til at finde optimals k
- Curse of dimensionality ved clustering

References

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Ed. by B. Schölkopf M. Jordan J. Kleinberg. URL: https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf. (accessed: 10.3.2021).

[2] Qi Dang, Jianqin Yin, Bin Wang, and Wenqing Zheng. "Deep Learning Based 2D Human Pose Estimation: A Survey". In: 6.24 (December 2019).

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

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

[5] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *Elements of Statistical Learning*. URL: https://web.stanford.edu/~hastie/ElemStatLearn/printings/ESLII_print12_toc.pdf. (accessed: 10.3.2021).

[6] 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).

[7] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning with Applications in R*. URL: https://static1.squarespace.com/static/5ff2adbe3fe4fe33db902812/t/601cc86d7f828c4792e0bcae/1612499080032/ISLR+Seventh+Printing.pdf. (accessed: 17.3.2021).

[8] 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". In: (2014).

[9] Alejandro Newell, Kaiyu Yang, and Jia Deng. "Stacked Hourglass Networks for Human Pose Estimation". In: (2016).

[10] 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.

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

[12] 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).

[13] 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.

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

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

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

[17] 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).

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

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

[20] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. https://d2l.ai. 2020.