FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

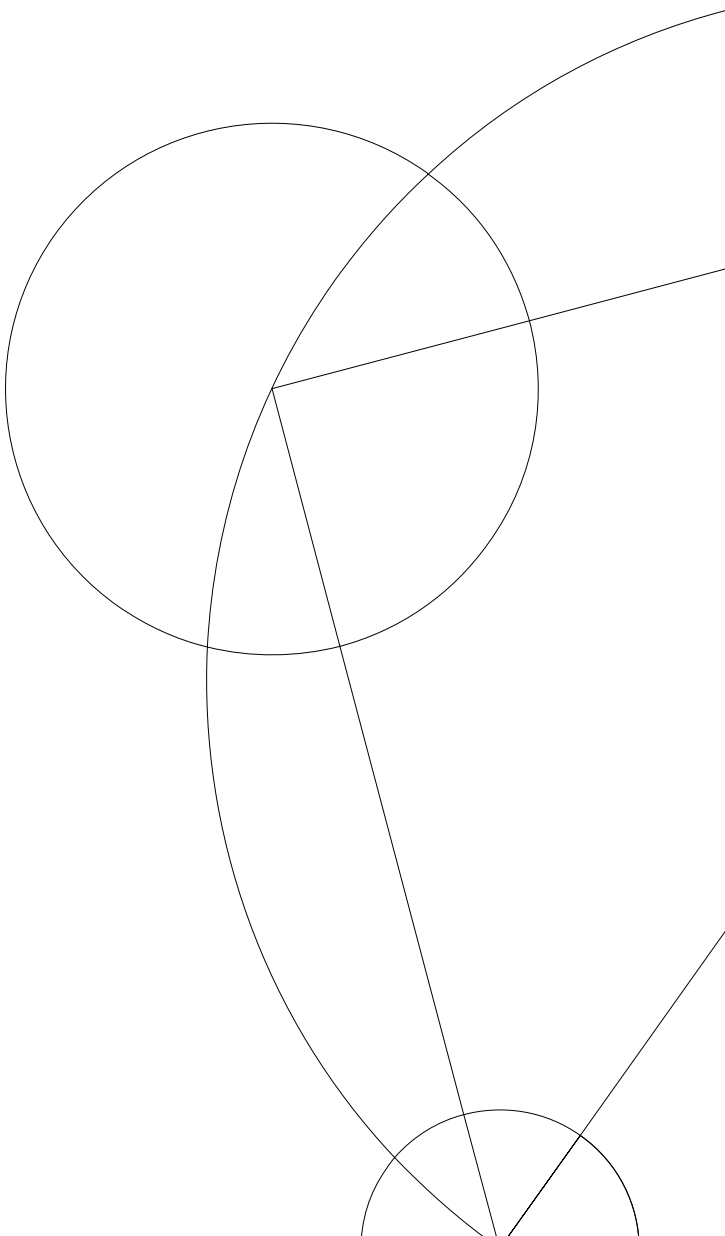
Bachelor Thesis

2D Articulated Human Pose Estimation

Using Explainable Artificial Intelligence

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

April 7, 2021



Supervisor
Kim Steenstrup Pedersen kimstp@di.ku.dk

Contents

1 Machine Learning Theory

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

1.1 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.

1.2 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 [4]. 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 [4].

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.

1.3 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*.

1.3.1 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 [8].

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 [5].

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.

1.3.2 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 [6].

1.4 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 1.4 the theory and mathematics behind neural networks is described and explained.

1.4.1 The Mathematics Behind Neural Networks

Algorithm 1 GradientDescent [12]

Require: Learning rate η
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* [3].

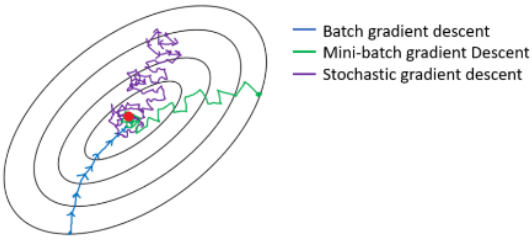


Figure 1: Comparison of batch, mini-batch and stochastic gradient descent [7]

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 [12].

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 [12].
Choosing the right batch size can be difficult, however, there are a few guidelines which one can follow [7] [12]

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 power of 2 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 discard knowledge from the past, so

Algorithm 2 RMSProp [3]

Require: Learning rate η
Require: Decay rate ρ
Require: Starting position $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}
1: Initialize accumulation variables: $\boldsymbol{r} \leftarrow \mathbf{0}$
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 [3].

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

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

where

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

and

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

which makes the minibatch have 0 mean and unit variance. $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are then used to make the mini-batch have an arbitrary mean and standard deviation and are two parameters that needs to be learned when the network is being fitted. This helps the network to converge, as the batch normalization keeps centering the mean and standard deviation of the mini-batches [12].

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 [12].

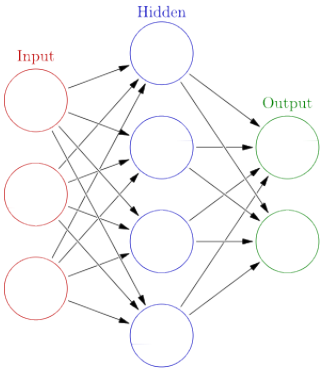


Figure 2: Visualization of a feedforward neural network with a single hidden layer [11]

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 [3] [10]. 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* that 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 [3].

Backpropagation

Backpropagation is an algorithm used to compute the gradient of each parameter in 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

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

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

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

and the partial derivative of each bias is

$$\frac{\partial L}{\partial b_j^{(i)}} = \frac{\partial z_j^{(i)}}{\partial b_j^{(i)}} \frac{\partial a_j^{(i)}}{\partial z_j^{(i)}} \frac{\partial L}{\partial a_j^{(i)}} = g^{(i+1)} \left(z_j^{(i+1)} \right) \frac{\partial L}{\partial a_j^{(i)}}$$

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 the 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 [9].

1.4.2 Convolutional Neural Networks

NN Upsampling
Maxpooling
Convolution

1.4.3 Stacked Hourglass

Reasoning behind using the Stached Hourglass
The Residual Modules
The Hourglass
The Stacked Hourglass

1.5 Principal Components Analysis and K-means Clustering

1.5.1 Principal Components Analysis (PCA)

1.5.2 K-means Clustering

2 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.

2.1 The COCO Dataset

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

Figure 3: Example of an image from the COCO dataset with the keypoints drawn on **[COCOarticle]**

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 **[COCOarticle]**, 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 3.
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, telling 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.

2.2 Data Preprocessing

2.2.1 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.

2.2.2 Preprocessing the images

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

Table 1: Data distribution

Figure 4: The results of processing the image from Figure 3 with the corresponding labels **[COCOarticle]**

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.
An issue can happen, where the bounding box still contains multiple people, which will confuse our model, since it does not know which person it should annotate. An example of this can be seen in the first image of Figure 4. 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. This 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 current bounding boxes would 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 lie 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, has their rgb-values centered by subtracting the mean rgb of all of the images from the training set, and saved. Doing all of these steps results in the distribution of images displayed in Table 1. In Figure 4, the results of processing the image from Figure 3 are shown with the corresponding labels. Lastly, the data is shuffled to help the developed model generalize the data better.

2.2.3 Handling the labels

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

Figure 5: An example of the heatmaps of a single image fused together and put over the original image [2]

For each image of a single person, our model outputs 17 heatmaps, one for each possible joint in the image, which tells the probability of the joint being in each pixel. An example of the heatmaps fused toegether can be seen in Figure 5.
The ground truth heatmap of a single joint is created firstly by initializing an all-zero 2D array with size 256×256 for each of the 17 heatmaps. Next, in the ith 2D array at position (x_i, y_i), corresponding to the position of the ith joint, a 1 is placed - this 1 now corresponds to where the ith joint is placed in the image according to the keypoint annotation of the image. Next, a Gaussian filter is used to smear out the image, where the standard deviation depends on the visibility of the joint: if the joint is visible, then the standard deviation is 0.5, whereas the standard deviation is 1 if the joint is not visible, as there are more uncertainty with the labeling of such keypoints. Lastly, as the model outputs 17 64×64 heatmaps, our heatmaps are resizes from a dimension of 256×256 to a dimension of 64×64.
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.

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

[4] 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).

[5] 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).

[6] 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).

[7] 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).

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

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

[10] 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).

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

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