1 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 1.1, where we will be giving a brief introduction to autoencoders. Throughout Section 1.2 the Stacked hourglass is described. Then, in Section 1.3 the algorithm of Principal Components Analysis is described. In the last section, Section 1.4, K-Means clustering is described.

1.1 Autoencoders (AE)

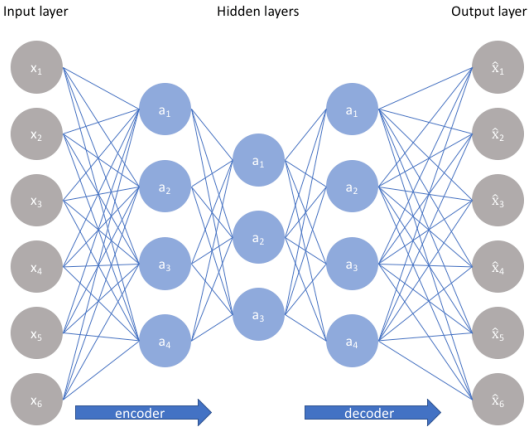

Figure 1: Visualization of an undercomplete autoencoder [4]

Autoencoders (often shortened as *AE*) is a class of neural networks that is trained unsupervised to output the input of the model. An autoencoder consists of two parts: the *encoder* and the *decoder*. When data is fed to the autoencoder, the data is passed through the encoder which processes the data and passes the data to the decoder, where the data is again processed and finally returned [1].

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 1. 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 [1].

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

1.2 Stacked Hourglass (SHG)

When performing the pose estimation in section **??**, we will be implementing and using the *Stacked hourglass* (also knowns as *SHG*) by Newell *et al.* [5]. The following description and explanation of the architecture is based on an interpretation of Newell *et al.* [5] and Olsen [7].

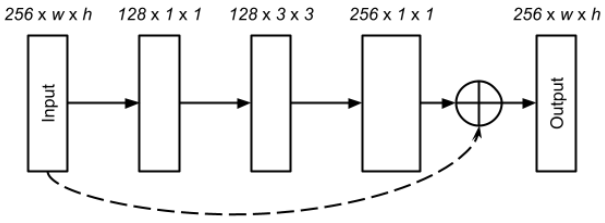

Figure 2: Visualization of the residual module [5]

1.2.1 The Residual Module

The Stacked hourglass makes heavily use of so-called *residual modules*, one of which is visualized in Figure 2. 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 [6].

1.2.2 The Hourglass

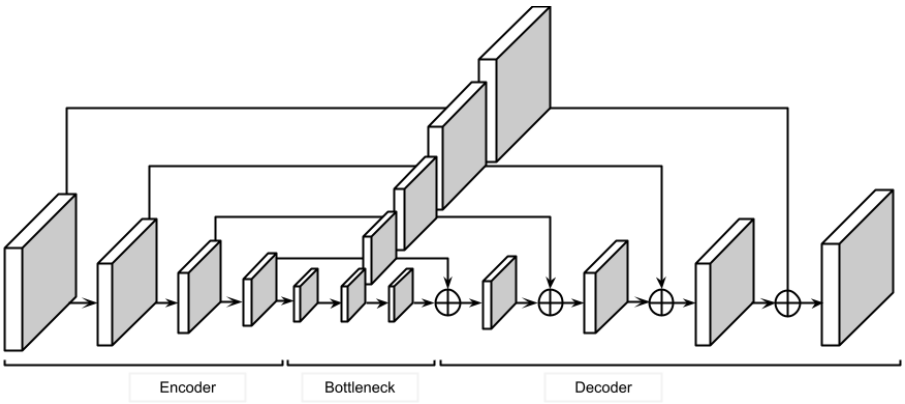

Figure 3: Visualization of a single hourglass [5]

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 3 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 out-

put 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.

1.2.3 The Stacked Hourglass

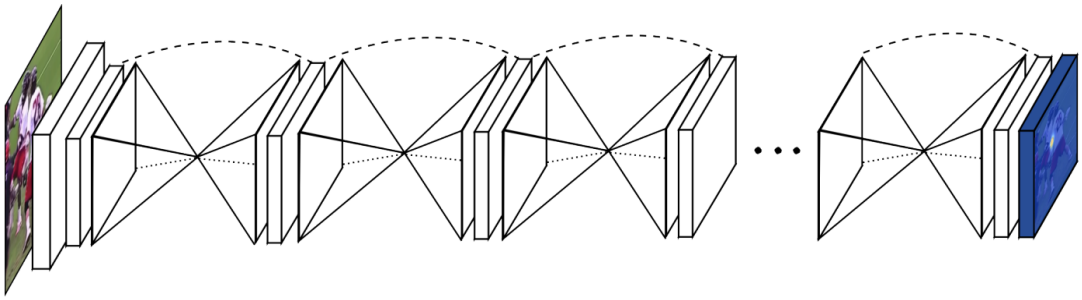

Figure 4: Visualization of the Stacked hourglass [5]

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 4, which makes each hourglass reevaluate estimates. To evaluate each hourglass, intermediate supervision is used by applying a loss to the intermediate predictions of each hourglass.

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.

1.3 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* [2]. For this reason multiple techniques have been developed for reducing the dimensions of a given dataset. We have already seen in Section 1.1, how autoencoders can be used for non-linear dimensionality reduction. Another very common technique for reducing the dimension of a dataset is *Principal Components Analysis* (*PCA*).

PCA is an unsupervised linear projection method used for reducing the dimension of a dataset from D down to k dimensions. The algorithm works by finding the k orthogonal vectors that

maximizes the variance of the input data, which the input data is then projected onto [8].

The pseudocode of the algorithm is visualized in Algorithm 1. 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. [8]

Algorithm 1 PCA [8]

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** $\boldsymbol{X}$

1.4 K-Means Clustering

Algorithm 2 K-Means [9]

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

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 an 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 2. 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 [8].

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 [8].

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 3. 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 [3].

Algorithm 3 Compute Silhouette Score [3]

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

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