1 Experiment

Throughout the following section the results and configuration details of our trained model are described and explained. In the first section, Section 1.1, the configuration details of the model, as well as the training of the model, are described. In the second section, Section 1.2, the results of training the Stacked hourglass is presented, as well as a short discussion of which version of the model which will be used going forward. In the last section, Section 1.3, we give an overview of the technical details behind training the network, which can be followed in case the reader has any technical problems in case of testing for reproducibility.

1.1 Configuration Details

Our model only consists of a single hourglass. The hourglass consists of 4 down- and upsamples. The module uses a single residual between each down- and upsample, as well as 3 residuals in the bottleneck. Newell *et al.* [3] and Olsen [4] experiment with different amount of hourglasses and with different amount of residual modules in each hourglass. They both come to the conclusion, that stacking multiple hourglasses or using hourglasses with multiple residual modules between each down- and upsample increases the performance of the model. However, as the main purpose of this section is not to create a model with state-of-the-art results, but instead to create a model that can be interpreted and explored, we have chosen to reduce the size of the model. For the same reasons, we will not be developing and testing various configurations of the architecture. Likewise, the purpose of this section is neither to improve the model developed by Newell *et. al* [3], hence why we will be making the same configuration choices as Newell *et. al* [3] and Olsen [4], 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 [4] and perform the batch normalization before each convolutional layer in each residual module, after the first convolutional layer of the entire network and before the last convolutional layer of the entire network. For the choice of acitvation function we use ReLU after each batch normalization. Each max pooling and nearest neighbor upsampling uses a kernel size and stride of 2, which halves and doubles the size of the input, respectively. The full network has been visualized in Figure 1.

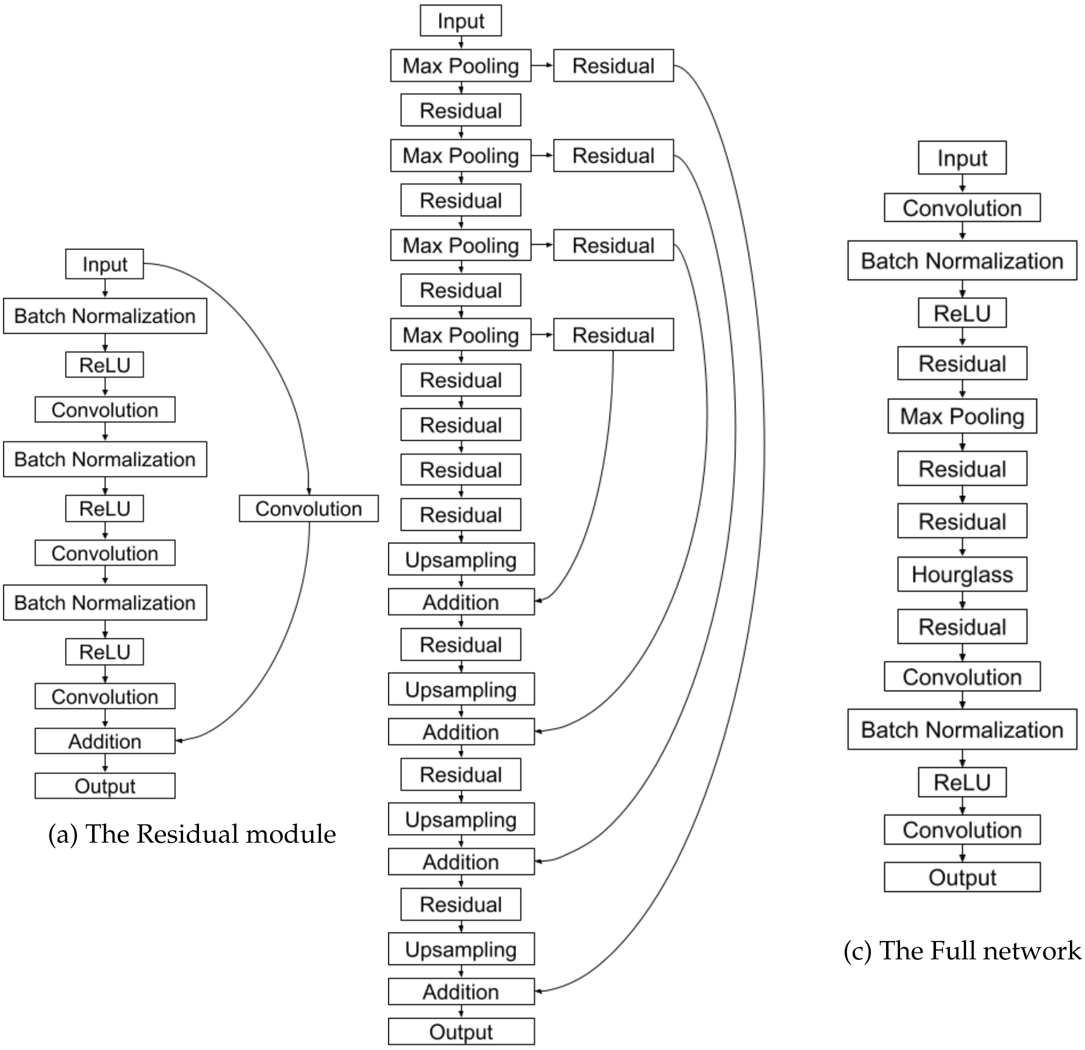
For the initial values of the weights we initialize each weight by sampling from a *Glorot normal distribution* (also known as a *Xavier normal distribution*), described as

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

where fan_{in} is the amount of input connections and fan_{out} is the amount of output connections to the layer of the weight [1]. By doing so we make all layers have the same activation variance and gradient variance, essentially helping the model to converge [2].

We make use of a mini-batch size of 16 and no data augmentation, since the dataset is already rather large and captures a lot of the variances. To optimize the network we make use of MSE as our loss function, *RMSProp* with momentum paramter $\alpha = 0.99$ as our optimizer, as well as use a initial learning rate of $2.5e-4$ as done by [3] and [4]. We decided to use a mini-batch size of 16 as Olsen experienced great results with this mini-batch size [4], as well as it being the biggest mini-batch size that our machine could run.

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



(a) The Residual module

(b) The Hourglass

(c) The Full network

Figure 1: Overview of the used architecture

for PCK, c (the normalization constant) and r (the threshold radius), we follow Olsen [4] by letting 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 is computed after each epoch, keeping track of the best PCK accuracy. The first time the best PCK accuracy has not improved for 5 continuous epochs, the learning rate is dropped by a factor of 5 permanently, helping the training loss reach a minimum.

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

1.2 Results

In Figure 2, the evolution of the training loss, validation loss and validation PCK accuracy is visualized. The model were initially set to train for 100 epochs, however, we decided to stop the training early, as the model clearly started to overfit after 32 epochs, as seen by comparing the training and validation loss, as well as the PCK validation accuracy seemed to have converged.

The reduction of the learning rate happened after 21 epochs. By looking at the validation accuracy in Figure 2 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 1 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 **??** want to explore a model that performs decently well, we will be using the model with the highest validation accuracy as our model going forward. Thus, our model is the model from epoch 47, which has a training loss of $4.19 \cdot 10^{-5}$, a validation loss of $5.43 \cdot 10^{-5}$ and a validation accuracy of 0.433. If we test the model on the held-out testing set from Section **??**, we get a PCK score of 0.441, which is based on all annotated keypoints (that is, both for $v = 1$ and $v = 2$)

1.3 Training Details

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

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

Table 1: Comparison of the the epochs yielding the best training loss, validation loss and validation accuracy

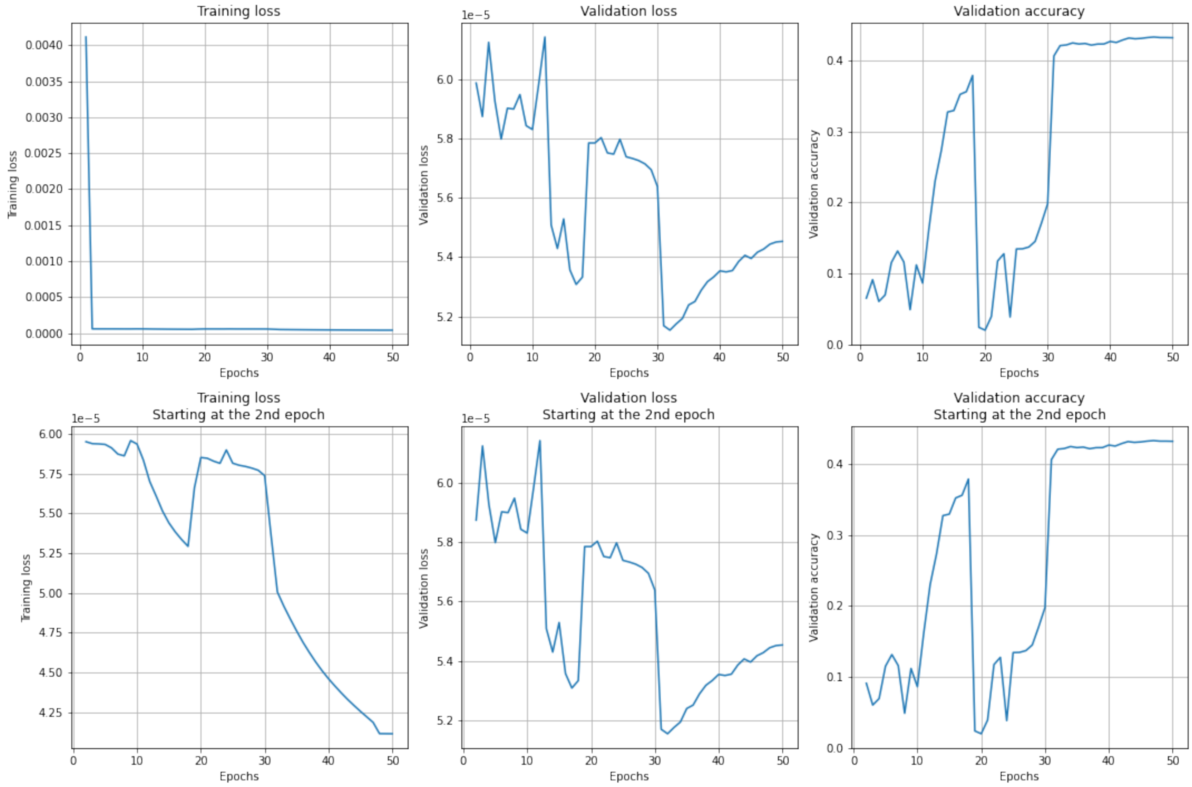


Figure 2: Visualization of the evolution of the training loss, validation loss and validation PCK accuracy of the trained model. Top row shows all of the 50 epochs. Bottom row shows epoch 2 and forward to ease the reading of the training loss