Assignment 4 - Stacked Hourglass Networks for Pose Estimation

Roba Al Majzoub
20020071

Rushali Grandhe
20020076

Zakaria Sebaitre
20020066

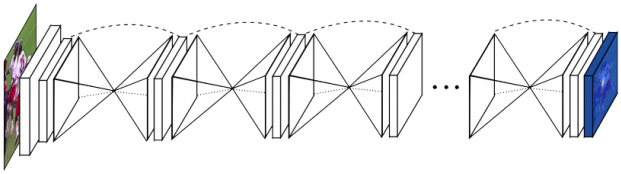


Figure 1. Stacked hourglass Architecture[3]

1 Introduction

Human Pose Estimation (HPE) is one of the most popular problems in computer vision. It involves predicting the posture of the human body within a given picture. The goal is to identify the locations of different joints and body parts from wrists to elbows, knees, head and more. This is so that the model can estimate the pose of people. Many approaches have been adopted for this purpose and one of them was the use of a specific architecture known as the stacked hourglass. In this report, we discuss modifications made to the stacked hourglass model in an attempt to decrease computational time while maintaining a comparable accuracy.

2 Stacked Hourglass Architecture

Stacked Hourglass is a CNN architecture used for HPE. It is formed of a stack of multiple hourglasses, each having encoder-decoder structure (hence the name hourglass)[3]. It starts by extracting information from multiple scales through convolution and maxpooling which increase the receptive field of the network. This increase in the receptive field allows the network to combine local features with global ones for better detection of joints. Each hourglass has 4 downsampling blocks and corresponding upsampling counterparts, which represent the depth of the network Figure 2. The output feature maps or channels of the downsampled blocks are fed to the upsampling blocks of the same dimensions via residual connections. The original stacked hourglass network uses 8 hourglasses along with intermediate supervision. Note that throughout the report, the term channels and feature maps are used interchangeably.

Bottlenecks or residual blocks are heavily used in the network between the downsampling and upsampling blocks and in the residual connections across the hourglass. While the encoder part mainly focuses on the feature extraction through downsampling, the decoder combines features and spatial information (coming from the corresponding residual branch) along with upsampling operation. These residual blocks use 1x1 and 3x3 convolutions with 128 channels. The last layer produces an output of 256 channels using a 1x1 convolutional layer, thus using less parameters while still being able to combine spatial information within the different depths of the hourglass. These bottleneck blocks are used between the encoder and decoder part as well.

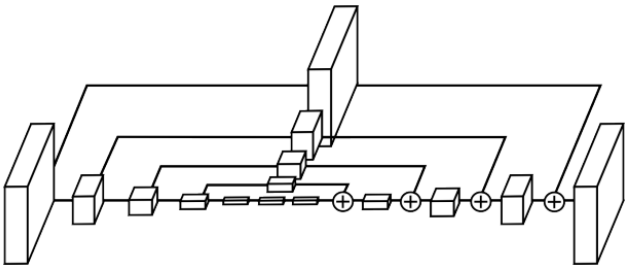


Figure 2. Architecture of one hourglass[3]

during the intermediate supervision stage between hourglasses, the loss is computed between the ground truth and the predicted joints. These losses are added together and are to be minimized. This causes the network to learn better representations as it will be penalized for the losses found through this intermediate supervision.

3 Modifications

Multiple modifications have been performed on the original architecture with the goal of speeding up the network with as little compromise in the accuracy of the model as possible. Experiments were conducted with different changes in the architecture, parameters or a combination of both. Different models were trained for 50 epochs to observe the trend of accuracy and the ones which showed promisingly

high accuracies were run for up to 220 epochs.

Table 1 shows the different experiments performed, the training time required per epoch, the validation mean accuracy along with the inference time.

3.1 Changing number of channels

Each bottleneck layer makes use of 128 channels/output feature maps, which may be computationally heavy. Hence, we decreased the number of channels keeping the remaining architecture fixed.
We used 64 channels as an alternative (Table1,Case 2). This was done by setting self.num_feats variable to 64 in the model.py file.

3.2 Changing depth of hourglasses

The depth of each hourglass is 4 in the original model. We tried to study the effect of different depths for the 8 hourglasses. This was achieved by using different values for the self.depth variable in the model.py file.

In Case 6, we used an increasing depth across the stacks in the order of 2 in the first 3 layers, 3 in the following 3 layers and 4 in the last 2 layers.] (Case 6). We also experimented with decreasing depths in the order of 4 in the first two layers, 3 in the following 4 layers and 2 in the last 2 layers (Case 7). (Table 1,Case 6,7).

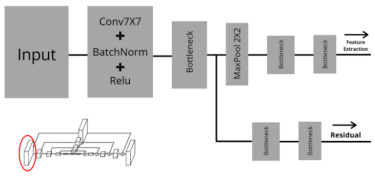
3.3 Changes related to bottleneck modules

Each hourglass of the original architecture has a depth of 4 i.e 4 downsampling and 4 upsampling levels. These levels contain bottleneck modules, 3 for each as shown in Figure 3a and 3b.

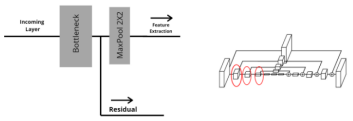
We experimented to see if we could compensate smaller number of stacks (6, rather than 8) with more bottleneck modules in each depth of the hourglass (Table 1, Case 4). This was done by modifying the _make_hour_glass() function in the model.py file.

Furthermore, each of the residual modules contains 3 blocks of Conv, ReLU and Batchnorm. We tried adding 1 such block for all the bottleneck modules (Table 1, Case 8). So, we attempted adding one entire block along with the subsequent layers that constitute the module. The goal was to verify whether adding more bottleneck modules is supposed to make the model faster as the theory suggests. Since these bottlenecks use smaller kernel sizes but in multitudes instead of larger singular kernels, they are supposed to decrease the computational cost during both training and inference stages. This addition appeared to be a decent substitute since it is significantly faster while preserving information to a larger extent as they propagate across the network. In practicality, we added another block consisting of convolutional layer, a ReLU and a batchnorm layer in the Bottleneck class of the model.py file. This experiment was conducted in comparison with the other modifications as per(Table 1, Case 8). The same model was traind for 220 epochs (Table 2, Case D).

BatchNorm was also used in the bottleneck module. In batch normalization, input values of the same neuron for all the data in the mini-batch are normalized. We tried to analyse the effect of replacing it with LayerNorm, in which input values for all neurons in the same layer are normalized for each data sample(Table 1, Case 5). This was done by replacing nn.BatchNorm(planes) with nn.GroupNorm(1,planes) in the Bottleneck class of the model.py file.



(a) First downsampling layer[1]



(b) Other 3 downsampling layers[1]

3.4 Changing number of hourglass stacks

The original architecture uses 8 stacks of the hourglass module. We (Table 1,Case 1)considered that to be our baseline for comparison. Next we tried to experiment with lower number of stacks. Using 6 stacks (num_stacks=6 in the model.py file) improved the training and inference time, while the accuracy decreased slightly (Table 1,Case 3). The model was also trained for 220 epochs(Table 2, Case C)

3.5 Using Adam optimizer

We tried using the Adam optimizer instead of RMSProp, since Adam or Adaptive Moment Optimization algorithms combines the heuristics of both Momentum and RMSProp[2]. We used RMSProp with momentum for the experiments.

Model	Training time(min/epoch)	Validation Mean accuracy	Inference time(min)
1) original	18	83.18	1.2
2) 64 channels	14.3	80.48	1.03
3) 6 stacks	14.4	83.13	1.05
4) 6 stacks 64 channels 4 residual blocks	14.3	80.73	1.08
5) 8 stacks 64 channels, layer norm	14.7	78.83	1.1
6) 8 stacks, depth [2x3,3x3,4x2] 64 channels	14.3	80.68	1.1
7) 8 stacks, depth [4x2,3x4,2x2] 64 channels	14.8	81.47	1.15
8) 4 blocks in each bottleneck module	14.6	84.23	1.06

Table 1. Results of different modifications for 50 epochs with batch size 20

Model	Batch size	Training time(min/epoch)	Validation M. Accuracy	Inference time(min)
A) original	6	18	86.7	1.2
B) original	20	14.8	86.4	1.1
C) 6 stacks	20	14.4	85.8	1.08
D) 4 blocks per bottleneck module	20	14.6	86.38	1.06

Table 2. Results of different modifications for upto 220 epochs

3.6 Changing Batch size

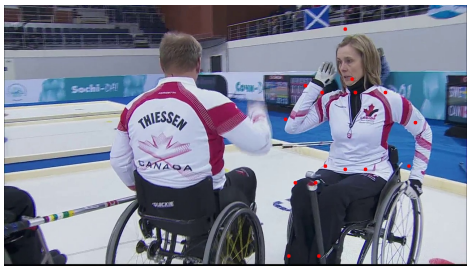
Another modification was to change the batch size of the data trained per loop. Since the original batch size was 6, We tried increasing it from 6 to 20.

4 Results

Table 1 shows the results for different architectures obtained after training for 50 epochs with a batch size of 20. Table 2 shows our best architectures along with the original model. We use the metric PCKh for joint detection, where the prediction is considered correct if the predicted joint location is within the range of 0.5 of head bone length from the true joint location. We use this metric to show validation accuracies for the body joints in Table 3 and Table 4.

5 Qualitative results

To visualize the predictions obtained through the various changes made to the model, we drew the positions of the predicted joints on the same image using locations of joints as red dots to compare between different models, as can be seen in (Figure 4). More results are added to the appendix each corresponding to one of the changes mentioned in Tables 1.



(a) Case A

(b) Case C

(c) Case D

Figure 4. Qualitative results of different architectures

6 Discussion

We observed that decreasing the number of channels, although decreased the training and inference time (due to the smaller structure), lead to a drop in the accuracy as well, indicating that more feature maps help in capturing more useful information thus leading to better accuracy. Similarly, reducing the number of stacked hourglasses from 8 to 6 led to a slight drop in accuracy but is still comparable to that of

Head	Shoulder	Elbow	wrist	Hip	Knee	Ankle
1) 93.89	92.05	85.04	79.66	81.27	77.27	74.92
2)93.62	90.66	82.09	75.95	77.06	73.83	69.04
3)93.93	91.86	84.73	79.46	81.79	76.53	72.22
4) 93.32	90.83	82.44	75.66	79.52	73.38	68.42
5) 92.70	89.83	81.42	74.10	75.42	70.84	65.64
6) 93.76	90.69	82.34	76.01	78.81	73.95	67.62
7) 93.79	90.69	83.13	75.38	80.56	75.80	69.49
8) 93.55	92.88	85.96	80.16	83.09	78.52	72.69

Table 3. Validation accuracies - PCKh in % on different joints for each implemented case as in Table1

Head	Shoulder	Elbow	wrist	Hip	Knee	Ankle
A) 95.63	94.17	88.36	83.57	84.92	82.14	77.49
B) 94.98	93.35	87.68	82.37	84.44	81.97	78.64
C) 94.95	93.16	87.17	82.35	83.99	80.46	77.47
D) 95.02	93.41	87.76	82.51	84.63	82.05	77.21

Table 4. Validation accuracies - PCKh in % on different joints for each implemented case as in Table2

the original model's accuracy. Moreover, it has a better training and inference time due to its smaller size where the training time per epoch dropped from 18 to 14.4 min/epoch. Increasing the number of blocks proved to enhance the computational efficiency of the network where the time required per epoch decreased with this change in architecture along with a larger batch size from 18 to 14.6 min/epoch. We tried adding 1 block to the residual modules. Results comparable to the original model were obtained with 4 blocks in the residual modules. Moreover, the training and inference time were less than those of the original model. However with 2 blocks, the accuracy was lower. Batchnorm seemed to produce better results than LayerNorm as well.

Using hourglass depths in decreasing order [4x2,3x4,2x2] seemed to perform better than the ascending values of depths [2x3,3x3,4x2]. This may indicate that having larger depths in the initial hourglasses than in the later hourglasses helps in extracting more features that help in joint prediction. Nonetheless, changing depths lead to a decrease in accuracy in both cases without much improvement in training or inference time.

The original code used RMSProp with momentum. This turned out to be better than Adam in terms of the time required for training the model but Adam seemed to be doing better with bigger batch sizes, thus reducing fluctuations in the loss. A batch size of 6 was used in the original model, but a bigger batch size of 20 lead to better and faster results (Table 2) where the training time per epoch dropped from 18 minutes per epoch to 14.8 on the same architecture.

7 Conclusion

Stacked hourglass module was the first to introduce the encoder-decoder stack architecture, which proved to be very efficient at the time and was adopted in many similar methods in the years that followed. And like any architecture, there is always room for enhancements and changes to increase accuracy along with reducing the computational complexity of the architecture. Reducing time complexity was the main goal of our experiments.
All in all, we tried experimenting with different parameters like the depth of hourglasses, number of channels, number of hourglasses, residual modules, batch sizes and optimizers. Finally, we tabulated the best results in Table 2 and Table 4. The 6 stacked model and the 4 blocks in each residual module change showed comparable accuracy to the original model along with improvements in training and inference times.

References

[1] Nushaine Ferdinande. Using hourglass networks to understand human poses. https://bit.ly/2S05uzz, May 2020.

[2] Ayoosh Kathuria. Intro to optimization in deep learning: Momentum, rmsprop and adam. https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/, May 2018.

[3] Alejandro Newell, Kaiyu Yang, and Jia Deng. Stacked hourglass networks for human pose estimation, 2016.

Appendices

Multiple other experiments were conducted but their results were not good enough to include. One experiment was to use mixed precision for the data during training and backpropagation. The system automatically chooses the precision of the numbers based on the operation and may reduce the precision from 32 to 16 bits, which may actually enhance the computational speed of the system. Another was to use dropout methods but the validation stage was also dependent on the validation results or the first 20 epochs to determine which experiments were worth going forward with.

Figure 5. Case 1

Figure 6. Case 2

[1]NOTE: In the zipped code folder, we have included a note.txt file for some of the instructions related to run our modifications. Each modification has been implemented as a class in the model.py file

Figure 7. Case 3

Figure 8. Case 4

Figure 9. Case 5

Figure 10. Case 6

Figure 11. Case 7

Figure 12. Case 8