Deep Learning for Autonomous Driving

Project #2 – Report

Yarden As
Florian Mahlknecht

2020-06-12

Contents

1 Methodology

Throughout the development phase we consistently used the *validation* metrics to judge our models. To compete in the challenge, the best models have been uploaded to the codalab server and the test score was obtained. This is justified since test and validation scores have always been in the same range, with a maximal ± 0.1 change.

1.1 Repeatability

Most of the changes introduced in the code base can be steered with command line arguments. The attached source code therefore is able to reproduce the mentioned results from problem 1.4 to problem 4. Furthermore, all our experiment runs are *uniquely* determined by a three digit number. Our naming convention settled down to:

```
000-p-{problem_number}-{special-name}
```

Throughout the report we will indicate the training IDs along within angle brackets. The experiments logs are all available on Amazon S3 as well as on Polybox. Please let us know if there are any issues accessing the Amazon S3 storage in this regard. We will keep all the training runs as long as needed.

2 Problem 1

2.1 Hyperparameter tuning

2.1.1 Optimizer

For evaluating the optimizer performance impact we run the vanilla source code using stochastic gradient descend and the default learning rate 0.01, and for *Adam* we used 0.0001.

More qualitative results are shown in the total loss plot provided in fig. 1: meanwhile Adam generally shows few fluctuations for the equivalent learning rate with respect to SGD, at later iterations spikes may appear. However, overall both optimizers decrease the loss as desired.

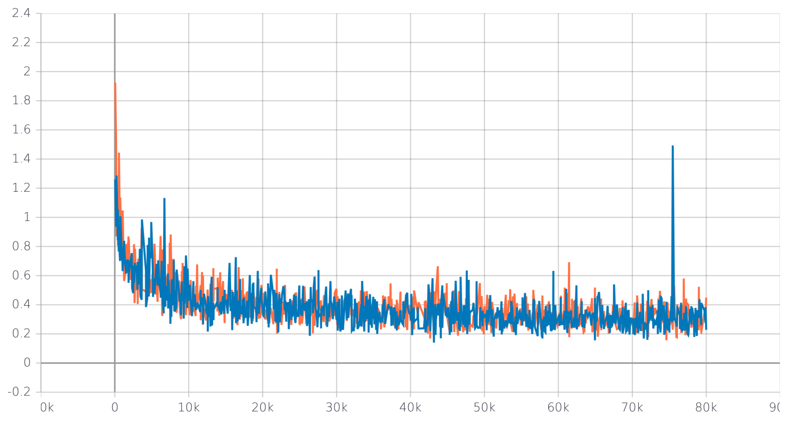
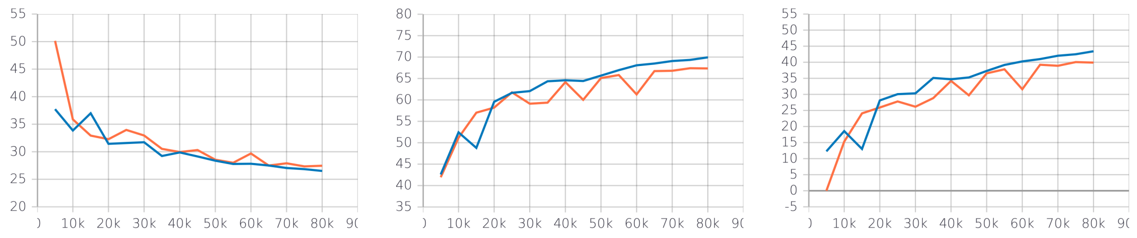


Figure 1: Total loss, Adam (blue) compared to SGD (orange)

Figure 2 provide the validation metrics.



(a) Depth [SI-log RMSE] (b) SemSeg [IoU] (c) Grader metric

Figure 2: Metric results, Adam (blue) compared to SGD (orange)

Adam achieves superior results in all tasks, as well as in the overall grader metric. In table 1 we additionally see that the training time of both optimizer is almost the same, for Adam we only see a minor 5 % increase.

Optimizer	LR	SI-logRMSE	IoU	Grader	Training time
sgd	0.0100	27.34	67.40	40.05	**05:10**
adam	0.0001	**26.51**	**69.94**	**43.42**	05:25

Table 1: Performance results, Adam compared to SGD

Given the above results, Adam will be used by default in the following sections.

2.1.2 Learning rate

From the previous learning rate of 0.0001, we test two new learning rates, one order of magnitude above and one below.

Again starting from a qualitative comparison, fig. 3 unveils that the lowest learning rate gives as expected the slowest convergence, and therefore the largest final offset. The larger learning rate from before, 0.0001 in orange, yields faster convergence and also better results. However increasing again to 0.001, shown in blue, behaves after a good starting point *worse*, with slower convergence and larger final error.

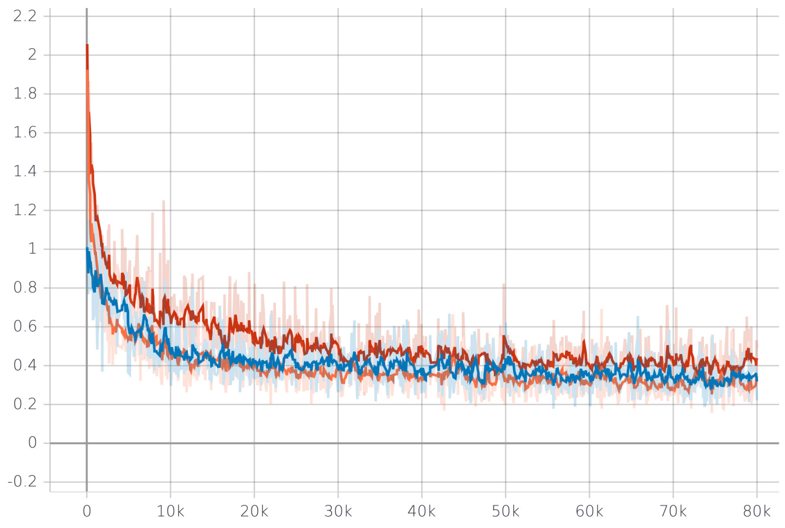
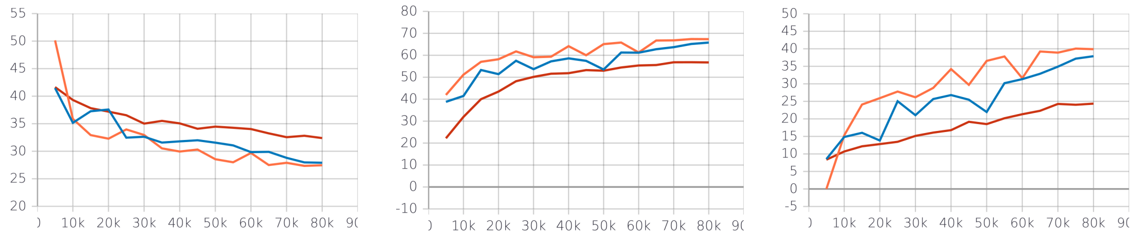


Figure 3: Smoothed total loss, learning rate 0.001 (blue), 0.0001 (orange), 0.00001 (red)

Indeed, the quantitative performance graphs shown in fig. 4 confirm this suspicion.



(a) Depth [SI-log RMSE] (b) SemSeg [IoU] (c) Grader metric

Figure 4: Metric results, learning rate 0.001 (blue), 0.0001 (orange), 0.00001 (red)

Table 2 clearly shows that the initial choice of 0.0001 was the best. As expected on training time was observed.

LR	SI-logRMSE	IoU	Grader	Training time
0.00100	27.91	65.80	37.89	05:26
0.00010	**26.51**	**69.94**	**43.42**	**05:25**
0.00001	32.39	56.74	24.35	05:27

Table 2: Performance results, learning rate

2.1.3 Batch size

Training batch size is another important training parameter that folds the following tradeoff: gradients of smaller batches are faster to compute as they are usually done in a single pass on modern GPUs. Furthermore, the standard error of the estimated mean gradient (of SGD algorithms) is proportional to $1/\sqrt{n}$ [1] (section 8.1.3) (where n are the number of samples). Hence, using large batches does not improve training efficiency a lot. On the contrary, although using small batches can also help regularizing the trained model, too small batches can lead to high variance in gradient estimation and an unstable learning process that requires small learning rates.

We conducted 3 trials experimenting with batch sizes of $\{10, 16, 30\}$ while maintaining the same number of gradient steps by changing the number of epochs to $\{40, 64, 120\}$ respectively. Our findings are summarized in table section 2.1.3

Batch size	SI-log RMSE	IoU	Grader	Training time
4	26.51	69.94	43.42	**05:25**
10	25.31	72.2	46.9	10:08
16	**24.81**	**73.11**	**48.3**	14:28
30	-	-	-	-

Table 3: Batch Size Performance Comparison

After experimenting with batches of $\{10, 16\}$ we concluded that the scores would not improve significantly and decided to cancel the last trial with batch size of 30 and save training resources.

2.1.4 Task weighting

We train our model by minimizing a loss for classification (segmentation task) and a loss for regression (depth estimation task), it is possible that one loss might be dominant over the other, biasing the model towards a specific task. To alleviate such problem, it is possible to weight differently the losses, train the models and choose the weights by evaluating the task metrics (IoU for segmentation and SI-log RMSE).

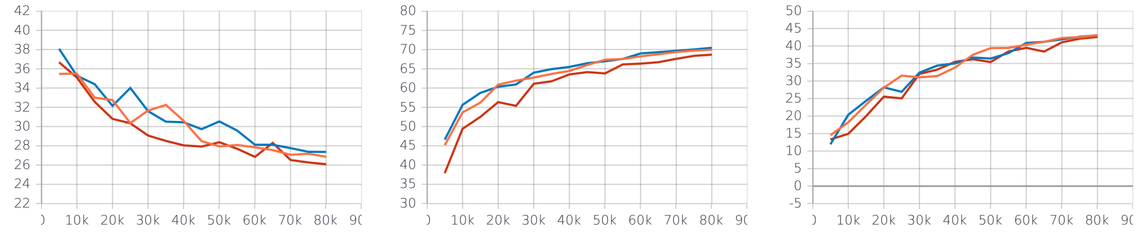
We searched the weight parameters going through the following steps:

Scaling validation losses we first used the validation set loss values at the end of the previous training experiment and scaled them such that after scaling, they will have equal weights. This scaling heuristic gave us weights ratio of 0.58 : 0.42 for semantic segmentation and depth estimation respectively.

Trying to improve by exaggerating since our previous weights, which were not too far apart, did not lead to improvements in the evaluation metrics, we decided to increase the weights ratio and used weights of 0.7 for semantic segmentation and 0.3 for depth estimation.

Somewhat surprisingly even larger ratio lead to only slight change in the validation metrics.

Sanity check we finally decided to experiment weights with an inverse ratio where the weight of depth estimation is larger than the depth estimation weight, in contrast to the previous two experiments. We did this to validate our experimentation process and to verify that we do not get surprising results.



(a) Depth [SI-log RMSE] (b) SemSeg [IoU] (c) Grader metric

Figure 5: Weights 0.58 : 0.42 (orange) `<009>`, 0.7 : 0.3 (blue) `<010>`, 0.3 : 0.7 (red) `<011>`

Weights ratio	SI-logRMSE	IoU	Grader	Training time
0.7 : 0.3	26.86	69.95	43.08	05:26:00
0.58 : 0.42	27.36	70.43	43.07	05:28:00
0.3 : 0.7	26.09	68.65	42.56	05:44:00

Table 4: Task weighting experiment results

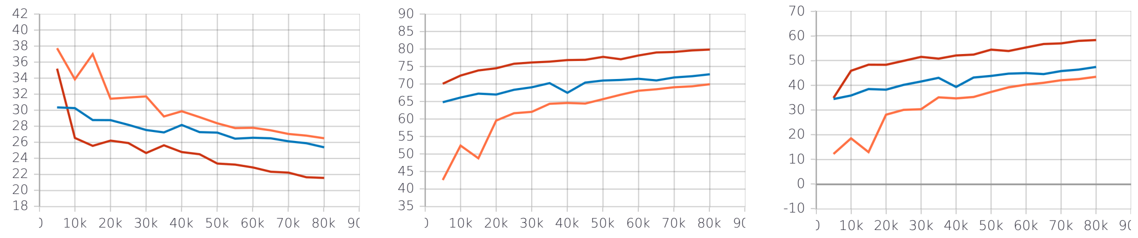
As seen, changing the weights have a small impact and slightly improves only one task at a time but not both tasks together.

2.2 Pretrained Weights and Dilation in Encoder

A common approach in designing model architectures for semantic segmentation and depth estimation from images is to have an encoder that extracts important "high-level" features from the input images, and a decoder that takes these features and transforms them back into the predicted images with transposed convolutional layers. As a consequence, the encoder plays a fundamental role in the prediction pipeline, thus having the most fit encoder for the task is crucial. Another common approach is to use an encoder that was already trained on images taken from a close distribution to our dataset, in example, the distribution of natural looking RGB images from common daily life (e.g. from ImageNet dataset) might be more closely related to our "Miniscapes" dataset than MRI scans of the brain. Using a pretrained encoder on such distribution, will easily capture the important "high-level" features of the images and will only need fine-tuning to match our dataset.

For semantics segmentation and depth estimation tasks, an important requirement is to effectively assign the "high-level" features with their spatial position in the original image. Encoding these features from a larger areas (which are also called "receptive fields") of the original image is beneficial since it accommodates more contextual information from the image. One way to increase the receptive field without increasing the number of parameters per layer is to use "Dilated Convolutions" [2].

In order to verify the aforementioned arguments, we compare our Adam baseline from section 2.1.2 with two pretrained "ResNet34" [3] encoders, one with dilated convolutions and the other with striding on its last block. Figure 6 shows the results.



(a) Depth [SI-log RMSE] (b) SemSeg [IoU] (c) Grader metric

Figure 6: Pretrained weights with striding (blue) `<031>`, baseline (orange) `<001>`, pretrained weights with dilated convolution block (red) `<032>`

	SI-logRMSE	IoU	Grader	Training time
Baseline	26.51	69.94	43.42	05:25
Pretrained + striding	25.39	72.78	47.39	**05:21**
Pretrained + dilation	**21.57**	**79.83**	**58.25**	05:59

Table 5: Different encoder configuration results

As expected, just by using a pretrained encoder, the tasks metrics start off with better scores than the baseline model. Moreover, it takes about $25K$ steps for the baseline model to catch up. Furthermore, the model that uses a dilated block instead of striding achieves significantly better scores than the other two models, as its extracted features are more informative for the decoder.

2.3 Experimenting with Atrous Spatial Pyramid Pooling

One of the most interesting challenges in the task of semantic segmentation is that our model should segment objects successfully, even if they appear in different scales. One powerful approach to answer this challenge was proposed in [4], [5]. In these papers, the authors proposed the Atrous Spatial Pyramid Pooling (ASPP) method. The ASPP method feeds, in parallel, the encoder's output features through multiple dilated convolutional layers. The outputs of these are then concatenated so that the following layers can extract information from different scaling of the same input.

2.3.1 Implementation

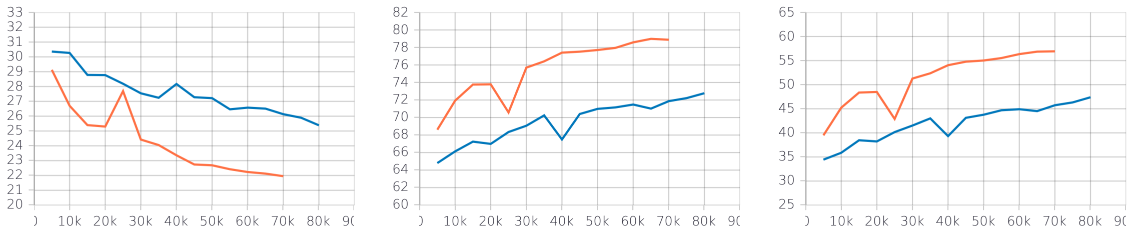
In practice, we implemented the ASPP module as explained in [5]. We used 3 dilated convolutional layers (including batch-normalization layers and ReLU activations) with $6, 12, 18$ dilation rates, as suggested in [5]. Additionally, we used a global average pooling layer and another convolutional layer with $(1, 1)$ sized kernel. All of these are computed in parallel on the encoder's output, concatenated and finally fed through yet another convolution with $(1, 1)$ kernel.

The implementation of the ASPP module is provided in the following code snippet:

```
class ASPP(torch.nn.Module):
 def __init__(self, in_channels, out_channels, rates=(6, 12, 18)):
        super().__init__()
        # ASPP module implementation as par with Figure 2 in handout.
        self.branches = torch.nn.ModuleList([ConvBnRelu(in_channels, out_channels,
        kernel_size=(3, 3),
            stride=1, padding=rate, dilation=rate) for rate in rates])
        self.branches.append(ConvBnRelu(
                in_channels=in_channels,
                out_channels=out_channels,
                kernel_size=(1, 1),
                stride=1,
                padding=0,
                dilation=1))
                [...]
        self.gip_unit = torch.nn.Sequential(
                torch.nn.AdaptiveAvgPool2d((1, 1)),
                ConvBnRelu(in_channels, out_channels, 1, 1, 0, 1))
        self.conv1x1 = ConvBnRelu(
                in_channels=out_channels * 5,
                out_channels=out_channels,
                kernel_size=1,
                stride=1,
                padding=0,
                dilation=1)

 def forward(self, x):
         output = [branch(x) for branch in self.branches]
         gip_output = self.gip_unit(x)
         # expand acts as a bilinear interpolation since the output
         # of the gip_unit is just a single pixel.
         # (we verified same results by using F.interpolate(...))
         gip_output_upsampled = gip_output.expand(
         gip_output.shape[0], gip_output.shape[1],
         output[0].shape[2], output[0].shape[3])
         output.append(gip_output_upsampled)
         return self.conv1x1(
         torch.cat(output, dim=1))
```

Listing 1: ASPP class

2.3.2 Results



(a) Depth [SI-log RMSE] (b) SemSeg [IoU] (c) Grader metric

Figure 7: ASPP module (orange) <033>, without ASPP module (blue) <032>

	SI-logRMSE	IoU	Grader	Training time
Without ASPP	21.57	79.83	58.25	05:59:00
With ASPP	21.38	79.84	58.47	06:19:00

Table 6: Baseline model with and without ASPP module

fig. 7 shows the performance of our previous baseline from section 2.2 compared to the same model and hyperparameters but with the ASPP module. We observe that the ASPP on its own does not significantly improves the performance of the model. As we will see in the next sections, in order to truly shine, the ASPP module needs to be accompanied with an adequate decoder, in contrast to the dummy decoder used in this experiment.

2.4 Decoder – Skip Connection

To successfully exploit the advantages of the ASPP module from the previous section, we need to complete the architecture by implementing the decoder proposed in the Deeplab papers. Through a skip connection from the encoder, we can integrate effectively low level features from the encoder in the decoder stage. This means gaining back valuable spatial context information, improving especially the semantic segmentation task by exploiting these low level features such as edges.

2.4.1 Implementation

As given in the problem statement, we implement the decoder as presented in [6]. Chen, Zhu, Papandreou, *et al.* show in table 1 and table 2 from the paper, that *two* consecutive 3x3 convolutions as a final output stage and reducing the low level skipped features with a 1x1 convolution to just 48 channels give the best result. The channel number in between the losses is 256. The implementation is shown in listing 2.

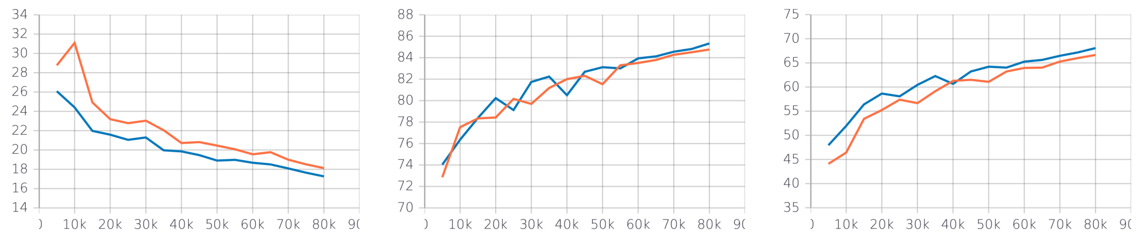
```
class DecoderDeeplabV3p(torch.nn.Module):
  def __init__(self, bottleneck_ch, skip_4x_ch, num_out_ch):
      super(DecoderDeeplabV3p, self).__init__()

      skip_4x_out_channels = 48
      intermediate_ch = 256

      self.skip_4x_conv1 = ConvBnRelu(in_channels=skip_4x_ch,
        out_channels=skip_4x_out_channels, kernel_size=1, padding=0)
      self.final_convolution = torch.nn.Sequential(
          ConvBnRelu(in_channels=skip_4x_out_channels + bottleneck_ch,
            out_channels=intermediate_ch,kernel_size=3, padding=1),
          torch.nn.Conv2d(in_channels=intermediate_ch, out_channels=num_out_ch,
            kernel_size=3, padding=1, bias=True)
      )

  def forward(self, features_bottleneck, features_skip_4x):
      low_level_features = self.skip_4x_conv1(features_skip_4x)
      features_4x = F.interpolate(features_bottleneck, size=features_skip_4x.shape[2:],
        mode='bilinear', align_corners=False)
      features_4x = torch.cat([features_4x, low_level_features], dim=1)
      predictions_4x = self.final_convolution(features_4x)
      return predictions_4x, features_4x
```

Listing 2: Deeplab decoder

It is important to note, that the last module must not contain batch normalization and activation function, as there the final predictions are computed.

2.4.2 Results

The effect on the metric results of the skip decoder module is compared to its previous skeleton code version in fig. 8 and table 7.



(a) Depth [SI-log RMSE] (b) SemSeg [IoU] (c) Grader metric

Figure 8: Metric improvements, with ASPP module (orange <033>), ASPP + depth module (blue <038>)

It is important to point out, that the semantic segmentation metric greatly improves from the newly available low level feature path. However, the depth does not improve significantly, and its validation metric shows a more unstable behavior during training.

Model	SI-logRMSE	IoU	Grader	Training time
ASPP	21.38	79.84	58.47	**06:19**
ASPP with Skip Decoder	**20.09**	**84.42**	**64.34**	07:13

Table 7: Performance results, Deeplab Decoder

The intuition behind is that semantic segmentation could benefit more from edges and low level features as the object boundaries in the classification tasks can effectively be improved, yielding a better IoU. In the depth estimation task on the other hand, the loss is differently designed and might not benefit in the same way from this effect, as the results suggest.

Although this is on a different dataset, we can see that the Deeplab results with mean IoU's above 80 % are possible to reproduce in our current setting.

3 Problem 2: Branched Architecture

In the last parts, the depth estimation and semantic segmentation tasks shared most of the architecture parameters as the only part that was exclusive for each of the tasks was the last convolutional layer. Another approach proposed in [7], [8], suggests that parameters sharing to occur only in the encoder part. One major benefit of this approach is that each branch of the architecture is much more task specific but still uses shared common features from the encoder. On the other hand, having a full inference branch that has its own ASPP module and a decoder requires many more parameters what might cause overfitting and will increase training and inference time.

3.1 Implementation

In the branched architecture, we used the same encoder and ASPP modules as in parts section 2.2 and section 2.3. Furthermore each of the decoders (one per task) was connected to a skip-connection to the encoder and to its respective ASPP module. Since we already had the building block (i.e. `ASPP` module and `DecoderDeeplabV3p`), the implementation of the branched architecture was pretty straightforward:

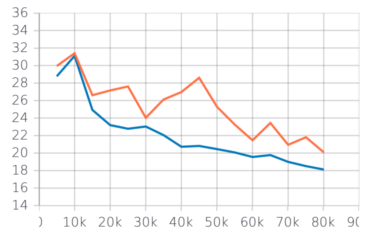
```
class ModelBranchedArchitecture(torch.nn.Module):
    def __init__(self, cfg, outputs_desc):
            super().__init__()
            self.outputs_desc = outputs_desc
            ch_out_depth = outputs_desc[MOD_DEPTH]
            ch_out_semseg = outputs_desc[MOD_SEMSEG]
            self.encoder = Encoder(
            cfg.model_encoder_name,
            pretrained=True,
            zero_init_residual=True,
            replace_stride_with_dilation=(False, False, True),
            )
            ch_out_encoder_bottleneck, ch_out_encoder_4x =
                    get_encoder_channel_counts(cfg.model_encoder_name)
            self.aspp_depth = ASPP(ch_out_encoder_bottleneck, 256)
            self.aspp_semseg = ASPP(ch_out_encoder_bottleneck, 256)
            self.decoder_depth = DecoderDeeplabV3p(256,
                    ch_out_encoder_4x,
                    ch_out_depth)
            self.decoder_semseg = DecoderDeeplabV3p(256,
                    ch_out_encoder_4x,
                    ch_out_semseg)

  def forward(self, x):
      input_resolution = (x.shape[2], x.shape[3])
      features = self.encoder(x)
      lowest_scale = max(features.keys())
      features_lowest = features[lowest_scale]
      features_tasks_depth = self.aspp_depth(features_lowest)
      features_tasks_semseg = self.aspp_semseg(features_lowest)
      predictions_4x_depth, _ = self.decoder_depth(
      features_tasks_depth,
      features[4])
      predictions_4x_semseg, _ = self.decoder_semseg(
      features_tasks_semseg,
      features[4])
      out = {}
      out[MOD_DEPTH] = F.interpolate(
      predictions_4x_depth,
      size=input_resolution, mode='bilinear',
      align_corners=False)
      out[MOD_SEMSEG] = F.interpolate(
      predictions_4x_semseg,
      size=input_resolution, mode='bilinear',
      align_corners=False)
      return out
```

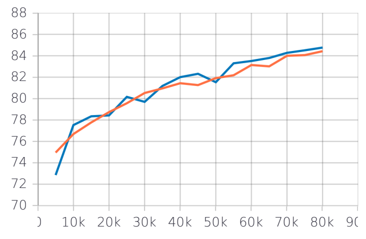
Listing 3: `ModelBranchedArchitecture` class

3.2 Results

A metrics comparison between the branched model and the DeepLabV3 model from section 2.4 is reported in fig. 9.

yardas@student.ethz.ch
fmahlknecht@student.ethz.ch


(a) Depth [SI-log RMSE]


(b) SemSeg [IoU]

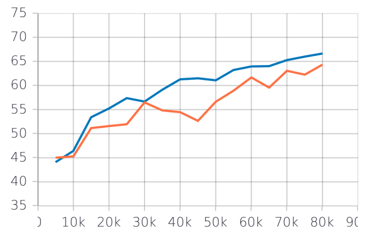

(c) Grader metric

Figure 9: Branched architecture (Light blue) `<039>`, DeeplabV3 Architecture (orange) `<038>`

	SI-logRMSE	IoU	Grader	#Parameters	Training time
DeeplabV3 architecture	20.09	84.42	64.34	26166708	**07:13**
Branched architecture	**18.12**	**84.76**	**66.64**	31002644	**08:24**

Table 8: Performance results, Deeplab Decoder

As seen, the branched architecture scores slightly better than the previous model since each branch is more specialized in its own task. While the branched architecture performs better than the DeeplabV3 with single branch, it also takes slightly more time to train, due to larger number of trainable parameters. An interesting insight is that most of the parameters are located neither in the ASPP nor the decoder, but actually in the encoder.

4 Problem 3: Distillation Architecture

We now exploit the provided self attention module to implement a task distillation architecture. The previously branched architecture is extended by basically cross combining its high level features from the decoders into a new final output decoder stage. All of the losses are used for training and backpropagation, while only the last ones are used for the final predictions.

The cross combination is implemented already in the self attention modules and basically creates first an attention mask for the cross mapping through a convolution followed by a sigmoid activation. This mask then maps the 'foreign' features to the own feature space in which they are consecutively *summed* to the others.

4.1 Implementation

The final decoder modules do not use the skip connection anymore, which is why we adopted the decoders from section 2.4. These simpler decoders contain just the final two 3x3 convolutions and are implemented as `DistillationDecoder`. The architecture becomes then straight forward to implement, as shown in listing 4.

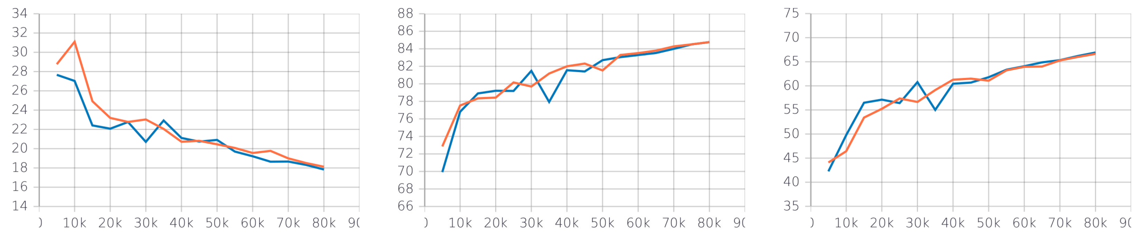
```
class ModelTaskDistillation(torch.nn.Module):
  def __init__(self, cfg, outputs_desc):
    super().__init__()

    # [...]

    skip_4x_out_channels = 48
    self.decoder_depth = DecoderDeeplabV3p(256, ch_out_encoder_4x,
                                  ch_out_depth, skip_4x_out_channels)
    self.decoder_semseg = DecoderDeeplabV3p(256, ch_out_encoder_4x,
                                  ch_out_semseg, skip_4x_out_channels)

    sa_feature_channels = 256

    self.sa_depth = SelfAttention(sa_feature_channels, sa_feature_channels)
    self.sa_semseg = SelfAttention(sa_feature_channels, sa_feature_channels)

    self.decoder_depth_final = DistillationDecoder(sa_feature_channels, ch_out_depth)
    self.decoder_semseg_final = DistillationDecoder(sa_feature_channels, ch_out_semseg)

  def forward(self, x):
    # [...]

    predictions_4x_depth, features_4x_depth = self.decoder_depth(features_tasks_depth, fea
    predictions_4x_semseg, features_4x_semseg = self.decoder_semseg(features_tasks_semseg,

    sa_depth_prediction = self.sa_depth(features_4x_depth)
    sa_semseg_prediction = self.sa_semseg(features_4x_semseg)

    sum_depth = torch.add(sa_depth_prediction, features_4x_semseg)
    sum_semseg = torch.add(sa_semseg_prediction, features_4x_depth)

    depth_prediction = self.decoder_depth_final(sum_depth)
    semseg_prediction = self.decoder_semseg_final(sum_semseg)


    out = {}
    out[MOD_DEPTH] = [F.interpolate(predictions_4x_depth, size=input_resolution, mode='bil
                                align_corners=False),
                     F.interpolate(depth_prediction, size=input_resolution, mode='bilinea
                                align_corners=False)]
    out[MOD_SEMSEG] = [F.interpolate(predictions_4x_semseg, size=input_resolution, mode='b
                                 align_corners=False),
                      F.interpolate(semseg_prediction, size=input_resolution, mode='bilin
                                 , align_corners=False)]
```

Listing 4: Distillation Architecture

It is crucial to use the second return values from the first decoders, which contain the *features* before the final prediction stages. These features are then passed to the self attention modules, and then *added* to generate the input for the final decoders. Please note that the provided skeleton code was already able to handle multiple outputs and uses the last one as the final predictions and both of them to compute the losses for backpropagation as expected. Therefore the implementation is again quite straight forward, by passing a *list* of outputs for each task.

4.2 Results

Figure 10 and table 9 present the results.



(a) Depth [SI-log RMSE] (b) SemSeg [IoU] (c) Grader metric

Figure 10: Metric improvements, from branched architecture (orange `<039>`) to distillation architecture (blue `<075>`)

The distillation architecture yields approximately one point improvement in depth and semantic segmentation respectively. However, the training time increases significantly by more than 50 %. The increase in parameters with respect to the branched architecture by approximately 3 million is smaller than the increase witnessed by approximately 5 million from DeepLab to branched architecture. It is interesting to note, that the training time before however just increased by approximately 15 %. A possible explanation is better parallelization of the branched architecture, whereas adding sequentially the SA modules, impacts training efficiency more drastically.

Architecture	SI-logRMSE	IoU	Grader	# Parameters	Training time
Branched	18.12	84.76	66.64	31002644	**08:24**
Distillation	**17.26**	**85.33**	**68.07**	34588712	13:02

Table 9: Performance results, Deeplab Decoder

5 Problem 4: Codalab Challenge

5.1 Workflow improvements

From a practical point of view, we engineered quite a few improvements through various scripts and enhancement, allowing us to keep the overview for many different training runs and fully exploiting the SageMaker possibilities.

5.1.1 Sync Script

Arguably one of the best decisions was to unify the naming of training runs and use this convention to *automatically* synchronize the results from S3 storage to a shared polybox folder.

The trailing identifier was a manually increased 3-digit number and could be used to effectively identify training runs on S3 and our local storage. Although the manual increase might seem a limitation, it was straight forward as the training launches were anyhow carried out manually from the notebook instance, and the last identifier usually persisted there.

Through the use of AWS command line interface[1], we could then automatically download and unpack the folders. Some highlights are shown in listing 5.

```
echo $missing_ids | while read i
do
  prefix=`printf "%03d" $i`
  fullname=`aws s3 ls "sagemaker-us-east-1-081019717444/$prefix" | grep -oP [...]`
  destname=${fullname: : -20} # remove timestamp
  echo "Processing $destname"

  status=`aws sagemaker describe-training-job --training-job-name $fullname | [...] `

  # [...]

  mkdir $destname
  cd $destname
  echo "Downloading model.tar.gz..."
  cmd="aws s3 cp s3://sagemaker-[...]/${fullname}/output/model.tar.gz model.tar.gz"
  eval $cmd
  # [...]

  echo "Extracting model.tar.gz..."
  tar xfz model.tar.gz
  mv log/* .
  rm -rf log checkpoints predictions source.zip model.tar.gz
  cd ..
  echo "Successfully downloaded $destname"
done
```

Listing 5: Synchronization scripts highlights

5.1.2 Result Parsing

The synchronized folder structure from the previous section can now be used to parse all the *results* with a python script. It automatically extracts the metrics from `submission.zip` and can conveniently generate markdown tables as well as latex tables, by using the Pandas[2] library. Some parts of the code are shown in listing 6. Key functionality is provided by ZipFile, allowing to read only specific parts from archives.

[1]see `https://aws.amazon.com/cli/`
[2]see `https://pandas.pydata.org/`

```python
# [...]
for filename in file_list:
    folder = filename.split('/')[-2]
    sys.stderr.write(F"Parsing {folder}...\n")
    info = {}
    experiment_num = int(folder[0:3])
    info['id'] = experiment_num
    info['Experiment Name'] = folder[4:]
    with zipfile.ZipFile(filename, 'r') as archive:
        with archive.open('source.zip') as source_zip:
            with zipfile.ZipFile(source_zip, 'r') as source_unzip:
                with source_unzip.open('cmd.txt') as cmd_file:
                    cmd = TextIOWrapper(cmd_file, 'utf-8').read()
                    cmd_lookup = parse_command(cmd)
                    info['Epochs'] = cmd_lookup['num_epochs']
                    info['Optimizer'] = cmd_lookup['optimizer']

                    # [...]

            df = pd.read_csv(csv_file)

            # [...]

            metrics_grader = df['metrics_summary/grader'].dropna().to_numpy()

            # [...]

            info['Best Epoch ID'] = best_index
            info['Grader'] = metrics_grader[best_index]
            info['SI-logRMSE'] = metrics_si_log_rmse[best_index]
            info['IoU'] = metrics_iou[best_index]
    infos.append(info)

assert len(infos) > 0

df = pd.DataFrame(infos)

# [..]

print(df.to_markdown(showindex=False)
```

Listing 6: Parse results script

A sample output snippet is shown below:

```
## Architecture Ranking TOP 10
**generated by parse_results.py on Fri Jun 12 14:35:06 2020**


|  ID | Experiment Name                         | Model        | SI-logRMSE |   IoU | Grader |      Time |
|----:|:----------------------------------------|:-------------|-----------:|------:|-------:|----------:|
| 058 | p-4-ultimate-log-inv-bugfix             | branched     |      15.29 | 86.98 |  71.7  |  19h 24m |
| 059 | p-4-ultimate-log-inv-bugfix-augmented   | branched     |      15.79 | 86.53 |  70.74 |  24h 20m |
| 048 | p-4-depth-aspp-semseg-new-vortex        | branched     |      16.96 | 86.02 |  69.06 |  17h 09m |
| 044 | p-2-branched-o-stride-8-rates-12-24-36  | branched     |      17.09 | 86.04 |  68.96 |  16h 06m |
| 043 | p-2-branched-output-stride-8            | branched     |      17.19 | 86.08 |  68.89 |  16h 01m |
| 041 | p-4-vortex-pooling                      | branched     |      18.47 | 86.04 |  67.57 |  16h 20m |
| 040 | p-3-final-distillation-bias-padding     | distillation |      17.84 | 84.76 |  66.93 |  15h 16m |
| 039 | p-2-final-branched-bias-padding         | branched      |      18.12 | 84.76 |  66.64 |   8h 24m |
| 046 | p-4-erf-transform-deeplab-w-11-0-5      | deeplabv3p    |      18.85 | 84.58 |  65.73 |   6h 54m |
| 038 | p-1-4-final-deeplab-bias-padding        | deeplabv3p    |      20.09 | 84.42 |  64.34 |   7h 13m |
```

It is needless to say, that having such an immediate way of comparing results is a big advantage, especially when more and more training runs are carried out. Even most of the latex tables presented in this report have been generated by this script.

5.1.3 Resume training on AWS

A major advantage for the final competition was to allow training runs to be resumed on AWS. The key aspect is to add a new input channel with an existing training log folder from AWS, which is then uploaded to the training instance. Listing 7 shows the snippet.

```python
input = {'training': 's3://dlad-miniscapes/miniscapes.zip'}
if checkpoint:
    input['checkpoint'] = F"s3://sagemaker-us-east-1-081019717444/{checkpoint}/"

estimator.fit(
    input,
    job_name=experiment_name + datetime.datetime.now().strftime("-%Y-%m-%d-%H-%M-%S"),
)
```

Listing 7: Checkpoint channel on Sagemaker

On the training instance the previous `model.tar.gz` can then be extracted into the new training folder, by a few lines of code, as listing 8 shows.

```python
checkpoint_dir = ''
if 'SM_CHANNEL_CHECKPOINT' in os.environ.keys():
    checkpoint_dir = os.environ['SM_CHANNEL_CHECKPOINT']

# [...]


if __name__=='__main__':
    # [...]

    if not checkpoint_dir:
        print("No previous checkpoint given")
    else:
        tarfile = os.path.join(checkpoint_dir, 'output/model.tar.gz')
        target_folder = os.path.abspath(os.path.join(log_dir, '..'))
        print(F"Unpacking checkpoint {tarfile}")
        shutil.unpack_archive(tarfile, target_folder)
```

Listing 8: Resume training on sagemaker [train_sagemaker.py]

Unfortunately in the used `lightning` version, the API for resuming is not that explicit, such that we had a few issues. The major problem is related to the LR scheduler, since the implemented version relates the learning rate to the total number of epochs:

$$LR = \left(1 - \frac{n_{epoch}}{N_{epochs}}\right)^{0.9}$$

Resuming a run, which completed for a given number of total epochs, using a higher number of epochs causes jumps in the learning rate. This problem was mitigated by introducing a constant learning rate scheduler. Furthermore, the LR schedulers are *restored* by the lightning framework, which again causes some issues. We therefore added in the lighting module another hook, which could successfully overwrite the restored schedulers with the newly configured ones. Additionally a code change from restored version resulted in an exception, which we changed into a warning. For details we refer to the files `helpers.py:50`, `rules.py:57` and `experiment_semseg_with_depth.py:87`.

5.1.4 Parameter Tuning on AWS

To fully exploit the AWS architecture we explored also the build-in hyperparameter tuning framework on Sagemaker.

The first step was to provide the metrics to Sagemaker, by printing them on the console after each validation step (see `experiment_semseg_with_depth:220`). Additional parameters in the Trainer instantiation allow to pass metric definition as regular expressions, as shown in listing 9. This additionally improves the output on AWS for training jobs, as all the metrics are conveniently displayed even with out the use of ngrok.

```
metric_definitions = [
    {"Name": "metric_grader", "Regex": "metric_grader=(\d*\.?\d*)"},
    {"Name": "metric_semseg", "Regex": "metric_semseg=(\d*\.?\d*)"},
    # [...]
]
```

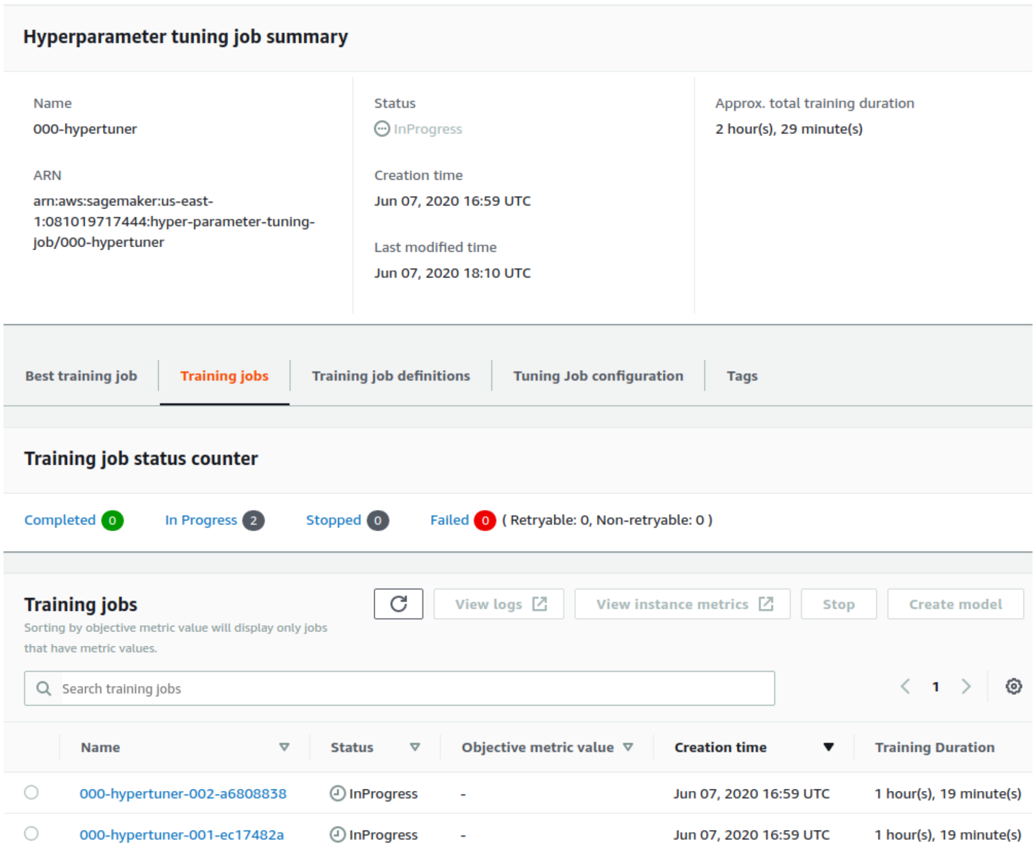
Listing 9: Sagemaker Metric Definitions

Then the tuning instantiation becomes straight forward as given in listing 10. By default a Bayesian approach is chosen, which should effectively improve convergence speed from a pure grid space approach. Integer and categorical parameters can be specified as well, we limited ourselves to the learning rate.

```
my_tuner = HyperparameterTuner(estimator=estimator,
                                 objective_metric_name='metric_grader',
                                 hyperparameter_ranges={'optimizer_lr': ContinuousParameter(
                                 metric_definitions=metric_definitions,
                                 max_jobs=25,
                                 max_parallel_jobs=2,
                                early_stopping_type='Auto')


my_tuner.fit(input,
             job_name='000-hypertuner')
```

Listing 10: Sagemaker Metric Hyperparameter Tuner

Once the notebook is executed, the tuning job is properly created and provides the interface shown in fig. 11. Training jobs are automatically instanced with different hyperparameter choices, in the degrees of freedom specified before. The identifiers are automatically picked.



Hyperparameter tuning job summary

Name: 000-hypertuner	Status: InProgress	Approx. total training duration: 2 hour(s), 29 minute(s)
ARN: arn:aws:sagemaker:us-east-1:081019717444:hyper-parameter-tuning-job/000-hypertuner	Creation time: Jun 07, 2020 16:59 UTC	
	Last modified time: Jun 07, 2020 18:10 UTC	

Best training job | Training jobs | Training job definitions | Tuning Job configuration | Tags

Training job status counter

Completed 0 In Progress 2 Stopped 0 Failed 0 (Retryable: 0, Non-retryable: 0)

Training jobs

Sorting by objective metric value will display only jobs that have metric values.

View logs | View instance metrics | Stop | Create model

Search training jobs

Name	Status	Objective metric value	Creation time	Training Duration
000-hypertuner-002-a6808838	InProgress	-	Jun 07, 2020 16:59 UTC	1 hour(s), 19 minute(s)
000-hypertuner-001-ec17482a	InProgress	-	Jun 07, 2020 16:59 UTC	1 hour(s), 19 minute(s)

Figure 11: Sagemaker Hyperparmetertuner

Unfortunately we did not have the necessary time once we had our architecture running, since the model is quite large and takes several days to compete in the challenge. However, in a production environment with one month of time, in our opinion this tool could greatly reduce the hassle involved in hyperparameter tunings. Once started, this is truly autonomous, and on many instances in parallel (without the limit of 2 instances as it was the case for us) it will most likely find the best combination in a reasonable amount of time.

5.2 Architecture Improvements

5.2.1 Vortex Module

After skimming through some papers about recent advances in semantic segmentation, we decided to implement the Vortex Pooling Pyramid proposed in [9]. It is an alternative to the ASPP module and improves the *utilization* factor for the encoder bottleneck features.

Overall, it is the same concept as the ASPP module, with just alternative convolutions and rates, i.e. final concatenation and parallel branch concept remains the same. It is however important to note, that they work with a bottleneck output stride of 8, so we need to change both final layers from the encoder backend to dilated versions, keeping the resolutions higher (i.e. flag configuration `False, True, True`).

In the paper they present various versions, and the module C is claimed to be the most efficient one, as it reuses previously computed average pools in a geometric series fashion. Figure 12 shows the module, without the same global average pooling layer, which works in the same

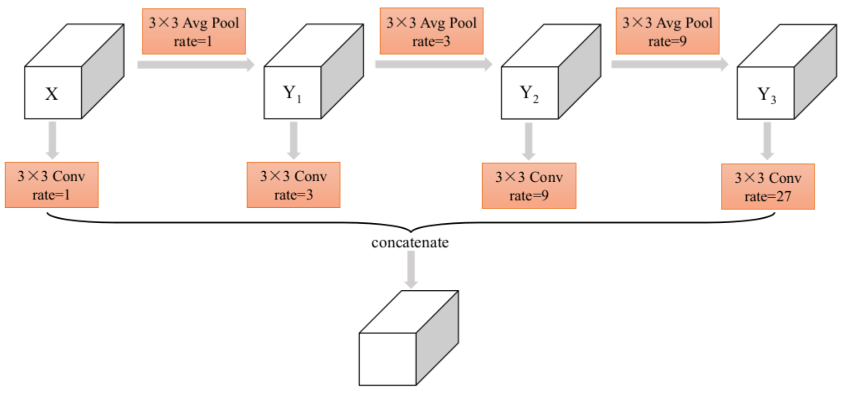


Figure 12: Vortex Pooling layer [9]

Implementation The main issue we encountered was the missing dilation parameter in PyTorch for the average pooling convolutions, see `https://github.com/pytorch/pytorch/issues/14907`.

To overcome the issue, we implemented this building block out from the plain convolution in PyTorch as follows in listing 11

```python
class DilatedAvgPoolingConvolution(torch.nn.Module):
  def __init__(self, num_ch, kernel_size, dilation):
    super(DilatedAvgPoolingConvolution, self).__init__()
    assert isinstance(kernel_size, int)

    weights = torch.tensor([[[[1 / kernel_size ** 2]]]]).expand(num_ch, 1,
                                                     kernel_size, kernel_size)

    self.weights = torch.nn.Parameter(weights, requires_grad=False)
    self.num_channels = num_ch
    self.dilation = dilation

  def forward(self, x):
    return F.conv2d(x, self.weights, padding=self.dilation,
      groups=self.num_channels, dilation=self.dilation)
```

Listing 11: Dilated average pooling

With this building block the implementation becomes straight forward and analogous to the ASPP module. Figure 12 provides thereby the necessary guidance. The most important implementation parts follow in listing 12.

```python
class VortexPoolingPyramid(torch.nn.Module):
    def __init__(self, in_channels, out_channels, rates=(6, 12, 18)):
        super().__init__()
        self.convolutions = {rate: ConvBnRelu(in_channels=in_channels,
          out_channels=out_channels, kernel_size=3, stride=1, padding=rate,
          dilation=rate) for rate in (1, 3, 9, 27)}

        self.avg_poolings = {rate: DilatedAvgPoolingConvolution(in_channels,
         3, dilation=rate) for rate in (1, 3, 9)}

        # [...]

    def forward(self, x):
        y_1 = self.avg_poolings[1](x)
        y_2 = self.avg_poolings[3](y_1)
        y_3 = self.avg_poolings[9](y_2)

        output = [self.convolutions[1](x),
                  self.convolutions[3](y_1),
                  self.convolutions[9](y_2),
                  self.convolutions[27](y_3),
                  ]

        gip_output = self.gip_unit(x)
        gip_output_upsampled = gip_output.expand(gip_output.shape[0],
          gip_output.shape[1], output[0].shape[2], output[0].shape[3])
        output.append(gip_output_upsampled)
        return self.conv1x1(
            torch.cat(output, dim=1)
        )
```

Listing 12: Vortex Pooling Pyramid

Results In order to get a fair comparison with the ASPP module, we adopt the output stride 8 to the latter, and run it with the suggested rates in the Deeplab paper [6], i.e. 12, 24, 36.

Although Xie, Zhou, and Wu claim to get an improvement of one percentage point, on our (different) dataset we could not reproduce those results, as table 10 shows.

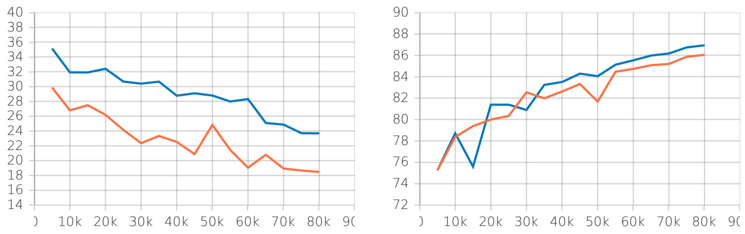
ID	Model	SI-logRMSE	IoU	Grader	Training time
<044>	Branched, output stride = 8	**17.09**	**86.04**	68.96	**16:06**
<041>	Branched Vortex Pooling Pyramid	18.47	**86.04**	67.57	16:20

Table 10: Performance results, Vortex Pooling Pyramid

Clearly, the performance for depth estimation *decreased*, whereas the semantic segmentation performance remained exactly the same. Despite not being able to reproduce the performance enhancements, wee decide to stick with the Vortex Module for the segmentation task, as it surely does not worsen performance. Furthermore a more specialized module for the segmentation task might yield positive side effects.

5.2.2 Double Skip Decoder

Aside from using the original skip connections from section 2.4, we added another skip connection from the encoder's features with output stride 2 (i.e., the spatial resolution of these features is smaller than the original image resolution by a factor of 2) and implemented the `DoubleSkipDecoder` class as follows. For each of the feature maps from the encoder, denoted by $4x$ and $2x$, we used a convolutional layer with batch normalization and ReLU activation. The outputs of these blocks where concatenated and fed into another convolutional layer (with batch normalization and ReLU), followed by another convolutional layer with a linear activation and without batch normalization. This final layer outputs class probabilities for each pixel.



(a) Depth [SI-log RMSE] (b) SemSeg [IoU] (c) Grader metric

Figure 13: With double skip connection (blue) `<042>`, without double skip connection (orange) `<041>`

Architecture	SI-logRMSE	IoU	Grader	Training time
Without double skip	**18.47**	86.04	**67.57**	**16:20**
With double skip	23.69	**86.93**	63.25	25:06

Table 11: Double skip connection results

Even though the general score is decreased, we can see that this is due to performance reduction only in depth estimation. Contrary to that, the double skip connection improves the semantic segmentation score, thus in the final architecture it was used only for the semantic segmentation task.

5.2.3 Final architecture

For the final architecture we combine the former approaches from sections 5.2.1 and 5.2.2 to finally combine them in a specialized branched architecture.

As discussed for instance also in section 2.4.2, depth estimation barely improves from low level features, which is why we decide to add the double skip connection in the segmentation branch, together with Vortex Pooling Pyramid. The depth estimation branch on the other hand remains coherent with the standard Deeplab architecture.

To keep the source code at this stage clean, we add a new model named `branched_vortex`, implementing the branched architecture in the described way. For details please refer to `model_branched_vortex.py`, as the code is straight forward and in line with the previously shown architectures.

Note that through the new model, the previous results remain untouched and reproducible with the attached source code.

The final model contains 32070180 parameters, i.e. 1 million more than the branched architecture, but still 2 million below the distillation architecture.

5.3 Depth Space Transformations

As it was suggested in the report, we investigate the depth distribution of the depth data of our training split.

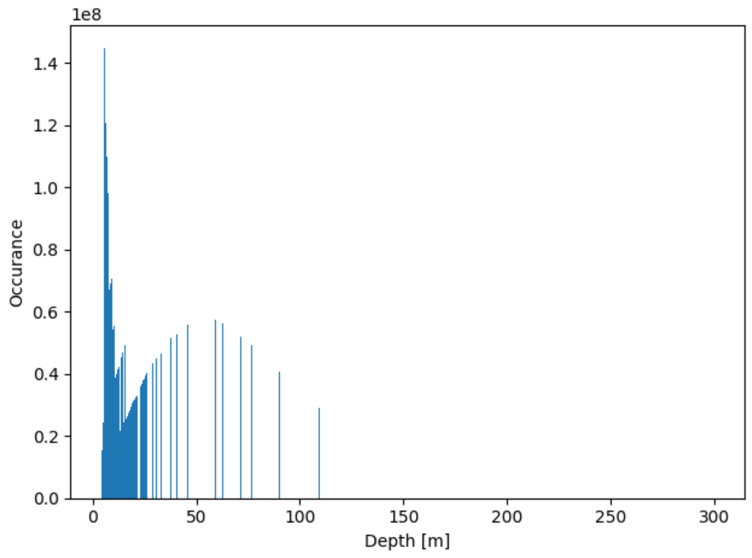


Figure 14: Metric Depth Histogram

Figure 14 shows the result of the plain depth data provided. It is interesting to note that there are clearly some artifacts from a log like discretization visible in the dataset. Overall the distribution could either be interpreted as a lognormal or a one-sided Gaussian. For both cases we examined the best transformations that could be applied to map them back to a ordinary Gaussian distribution or a uniform distribution respectively.

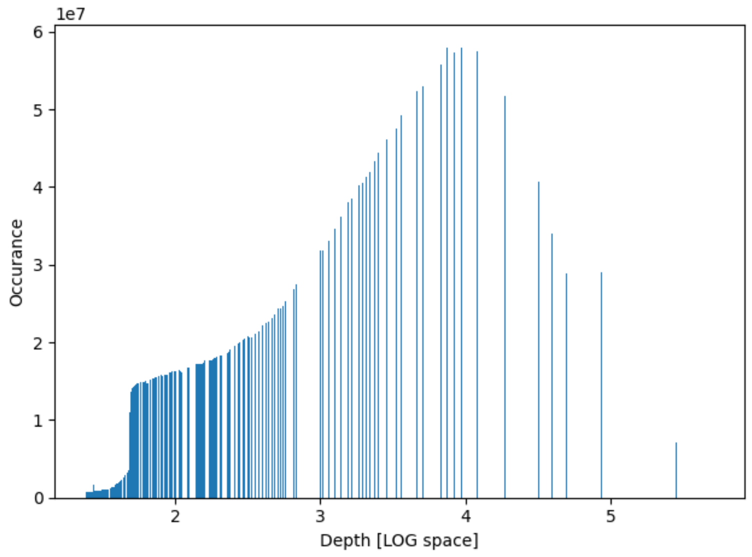


Figure 15: Logarithmic Depth Histogram

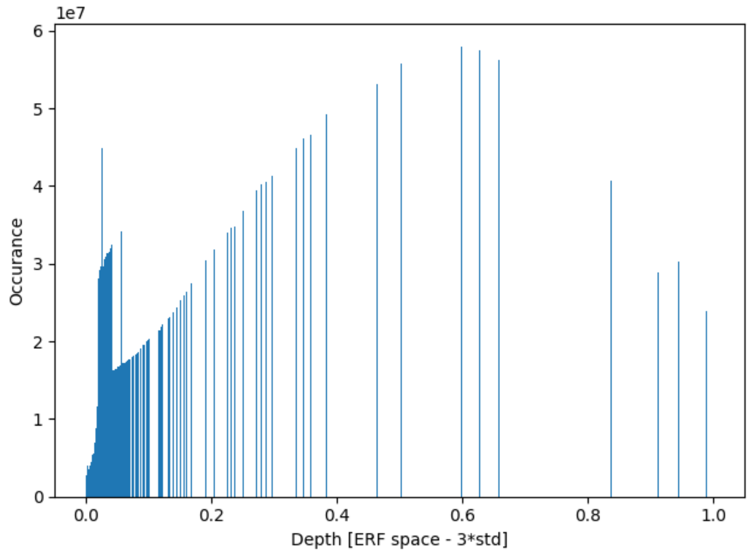


Figure 16: ERF transformed depth histogram

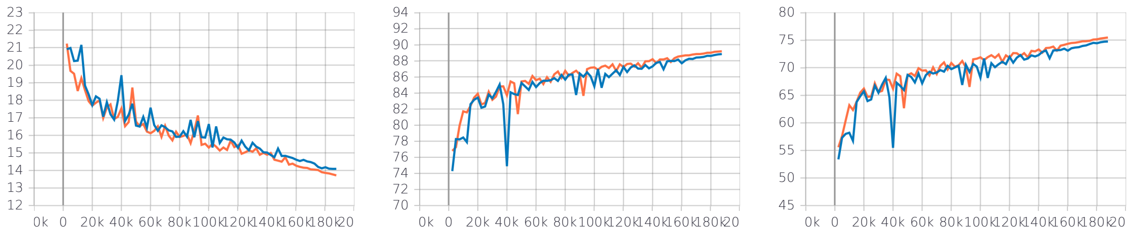
Figures 15 and 16 show that the LOG transform remains the best choice for getting a Gaussian similar distribution. For implementation details please refer to `compute_statistics.py` and `plot_statistics.py`.

Again, the transformation is optional and can be activated by using the flag `predict_log_space_depth`.

5.4 Data Augmentation

Applying different image transformations (e.g. rotations, scaling and so forth) is an easy way to make the dataset more diverse thus helping to combat overfitting and to make the model more robust. To compare results, we ran the same model architecture and hyperparameters with and without data augmentation. Our data augmentation included scaling with a maximum ratio of

1.25, minimum ratio of 1.0 and crop size of 288. We used these values so that no image data should be padded with zeros.



(a) Depth [SI-log RMSE] (b) SemSeg [IoU] (c) Grader metric

Figure 17: Without data augmentation (orange) `<060>`, with data augmentation (blue) `<061>`

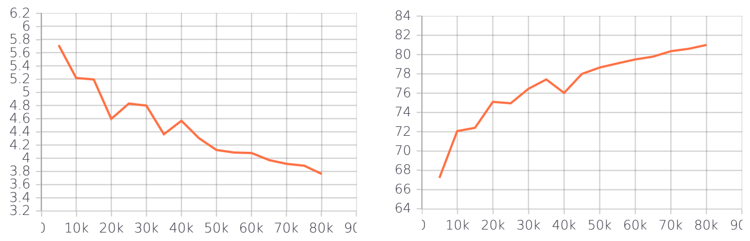
	SI-logRMSE	IoU	Grader	Training time
Without data augmentation	**13.72**	**89.18**	**75.46**	**80:33**
With data augmentation	14.09	88.83	74.74	93:30

Table 12: Data augmentation comparison

Although the experiment with data augmentation did not yield an improvement in results, we believe that using different hyperparameters for data augmentation, as well as including more types of augmentation, might increase our model's performance.

5.5 Pitfalls

As discussed in section 5.3, we applied a logarithm function to the depth data in order to improve performance. In order to validate our results, following the skeleton code, the transformations are applied to the ground truth. To compute the metrics, the *inverse* transformation is applied again to the transformed GT, to get the GT again. A bug in the *inverse* transform caused the depth metrics calculations to be erroneously computed in the logarithm space. The resulted depth metric was surprisingly good as shown in figure fig. 18.



(a) Depth [SI-log RMSE] (b) Grader metric

Figure 18: Pitfall in depth estimation validation scores `<047>`

The SI-log RMSE reaches to values well below 10 and the grader metric "achieved" a score of 81. There are two important lessons learned from this experiment:

1. the importance of a test set that is completely separate of the validation set

2. the importance of keeping the test *and* validation sets untouched: transformations should only occur on the outputs of the model and compared to the bare GT

Listing 13 points out the issue.

```
class ExperimentSemsegDepth(pl.LightningModule):
    def __init__(self, cfg):

        # [...]

        self.datasets[SPLIT_TRAIN].set_transforms(get_transforms(
            # [...]
            depth_meters_mean=self.depth_meters_mean,
            depth_meters_stddev=self.depth_meters_stddev,
            depth_meters_minimum=self.depth_meters_min,
            use_log=self.cfg.predict_log_space_depth,
        ))

        # ISSUE: same normalization applied to validation set
        self.transforms_val_test = get_transforms(
            # [...]
            depth_meters_mean=self.depth_meters_mean,
            depth_meters_stddev=self.depth_meters_stddev,
            depth_meters_minimum=self.depth_meters_min,
            use_log=self.cfg.predict_log_space_depth,
        )

    # [...]

    def validation_step(self, batch, batch_nb):
        y_hat_semseg, y_hat_semseg_lbl, y_hat_depth, y_hat_depth_meters = self.inference_s

        # [...]

        # ISSUE: Getting back the GT through inverse transforms
        if self.cfg.predict_log_space_depth:
            y_depth_meters = LogNormalization.inverse(y_depth)
        else:
            y_depth_meters = y_depth * self.depth_meters_stddev + self.depth_meters_mean

        # [...]
        self.metrics_depth.update_batch(y_hat_depth_meters, y_depth_meters) # ISSUE: this

        # [...]
```

Listing 13: Validation metrics issue

We are however aware, that this might be a desire not easily implementable in given frameworks.

5.6 Conclusion and final results

In the final competition, our longest training run <072> (training resumed two times) got a final test score of 76.51, i.e. 5th place overall and 89.72 IoU, i.e. 3rd place in segmentation. As we could expect we did not perform well in depth estimation.

- Although different tasks have a completely different measuring units, it is a common practice to add up the losses of these tasks. We think that an interesting research branch in multi-task learning would be to design more mathematically sound loss functions.
- In many machine learning tasks, it is easy to fall in the trap of "crunching" the data with some deep neural network without understanding the underlying characteristics of it. We found it very insightful to examine our data from different perspectives as a complementary step to hyperparameters tuning. In example, understanding that our depth ground truth data is very skewed, and that it is not evenly distributed in its domain, helped us treat it differently and apply the log transformation.

5.6.1 Future improvements

Our notable improvement of performance in the semantic segmentation task can be contributed mainly to the use of modules that are specialized for this task. In example, using the ASPP or the Vortex pooling layers improved feature extracting at different scales, which is an major aspect of the semantic segmentation problem. We think that designing modules that are more task specific to depth estimation (e.g. [10]) can improve performance in this task as well.

In the development process of our different model architecture, we kept on using the same hyperparameters from section 2. More specifically, we did not change the batch size from 4 to 16, trading off reduced performance with a faster results feedback from our training runs. Furthermore, we did not re-tuned the hyperparameters for our final architecture in section 5.2.3 so we are certain that there is still some room for improvement there.

Definitely train on batch size 16. Further improve resume training (additional parameter for total wanted number of epochs, such that LR scheduler is right throughout the whole experiment).

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016, `http://www.deeplearningbook.org`.

[2] F. Yu and V. Koltun, *Multi-scale context aggregation by dilated convolutions*, 2015. arXiv: `1511.07122 [cs.CV]`.

[3] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, 2015. arXiv: `1512.03385 [cs.CV]`.

[4] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, *Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs*, 2016. arXiv: `1606.00915 [cs.CV]`.

[5] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, *Rethinking atrous convolution for semantic image segmentation*, 2017. arXiv: `1706.05587 [cs.CV]`.

[6] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, *Encoder-decoder with atrous separable convolution for semantic image segmentation*, 2018. arXiv: `1802.02611 [cs.CV]`.

[7] D. Neven, B. D. Brabandere, S. Georgoulis, M. Proesmans, and L. V. Gool, *Fast scene understanding for autonomous driving*, 2017. arXiv: `1708.02550 [cs.CV]`.

[8] S. Vandenhende, S. Georgoulis, B. D. Brabandere, and L. V. Gool, *Branched multi-task networks: Deciding what layers to share*, 2019. arXiv: `1904.02920 [cs.CV]`.

[9] C.-W. Xie, H.-Y. Zhou, and J. Wu, *Vortex pooling: Improving context representation in semantic segmentation*, 2018. arXiv: `1804.06242 [cs.CV]`.

[10] A. Mousavian, D. Anguelov, J. Flynn, and J. Kosecka, *3d bounding box estimation using deep learning and geometry*, 2016. arXiv: `1612.00496 [cs.CV]`.