Deep Learning for Autonomous Driving

Multi-Task Learning for Semantics and Depth

Project 2 Report

Marian Kannwischer (mkannwisc@student.ethz.ch)
Fabian Lindlbauer (flindlbauer@student.ethz.ch)

We hereby confirm that we are the sole authors of the written work here enclosed and that we have compiled it in our own words. We have committed none of the forms of plagiarism described in the ETH Zurich 'Citation etiquette' information sheet. We have documented all methods, data and processes truthfully and have not manipulated any data.

Contents

1 Problem 1: Joint Architecture

This section describes the effect of various hyperparameters, the implementation of the Atrous Spatial Pyramid Pooling (ASPP) module and the resulting performance improvements.

1.1 Hyperparameter Tuning

For all hyperparameters we explored multiple values and only stopped when we found a decrease in performance in both directions of adjustment. Also, we found the recorded losses and metrics to show very similar information about performance and stability. Hence, we are only showing the evolution of the metrics over epochs in the following and omit plots of the loss in favor of readability.

1.1.1 Optimizer Choice and Learning Rate

The influence of the learning rate (LR) on the stochastic gradient decent (SGD) method is shown in Figure 1. We ran the given model with learning rates from 1×10^{-1} to 1×10^{-4}. Overall, a learning rate of 1×10^{-2} leads to the best results. One can also see that a higher learning rate causes less stable training. For the semantic segmentation task, a learning rate of 1×10^{-1} performs marginally better, however, this is likely due to the variance rather than a general trend.

Figure 2 illustrates the analogous experiment with the Adam optimizer and learning rates between 1×10^{-2} and 1×10^{-5}. A learning rate of 1×10^{-4} works best with Adam (`depth`: 26.5, `semseg`: 69.8, `grader`: 43.3). Again, one can see how the variance of the performance per epoch is correlated with the learning rate.

A comparison of the best SGD run with the best Adam run is depicted in Figure 3. Adam outperforms SGD in all three performance metrics and also shows a more stable behaviour. Nevertheless, we continued to evaluate most of the following experiments with both Adam and SGD to illustrate their strengths and weaknesses using different architectures.

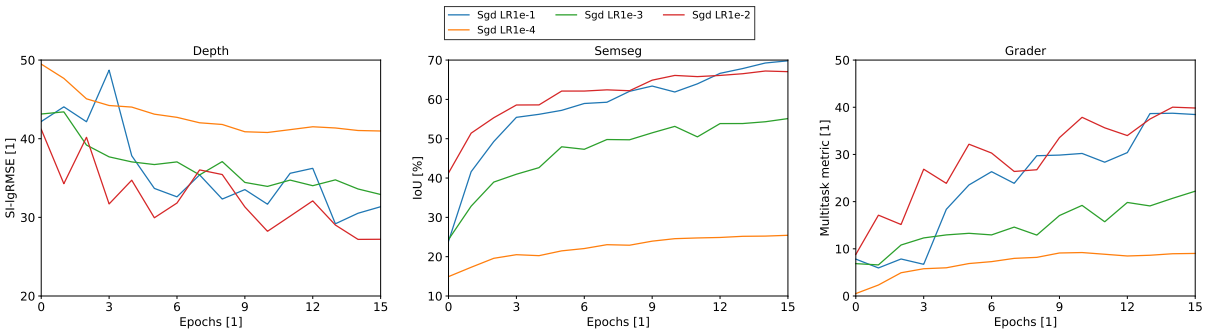

Figure 1: Performance metric comparison of SGD with different learning rates.

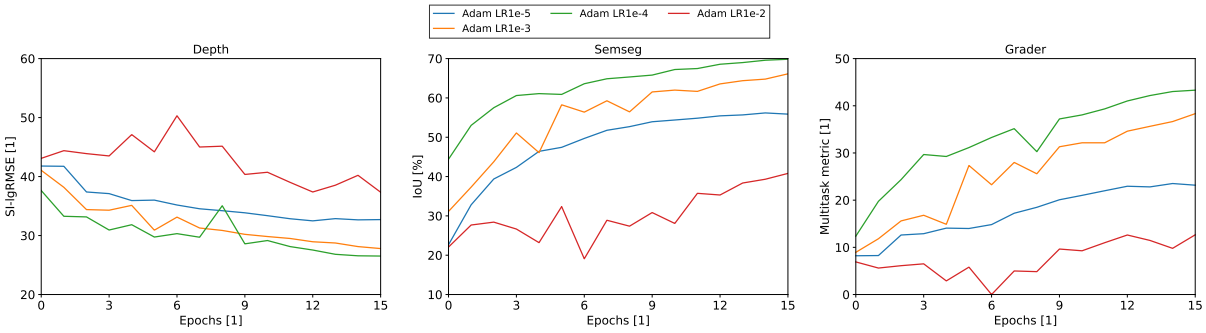

Figure 2: Performance metric comparison of Adam with different learning rates.

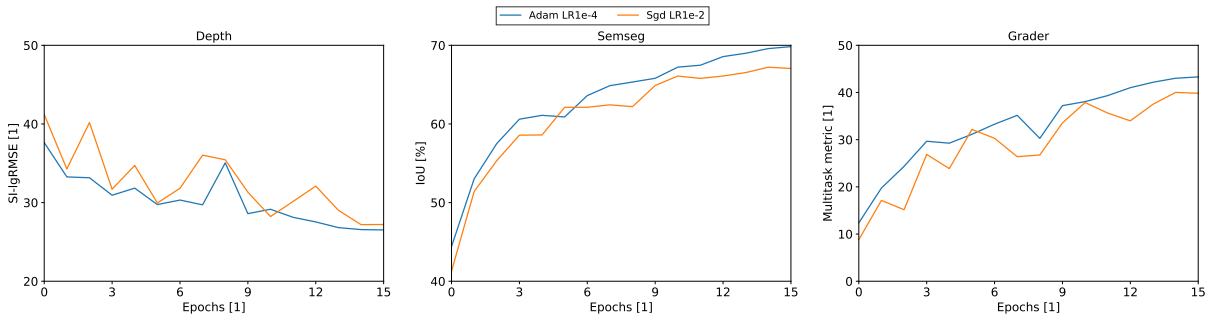

Figure 3: Performance metric of SGD and Adam with tuned learning rate.

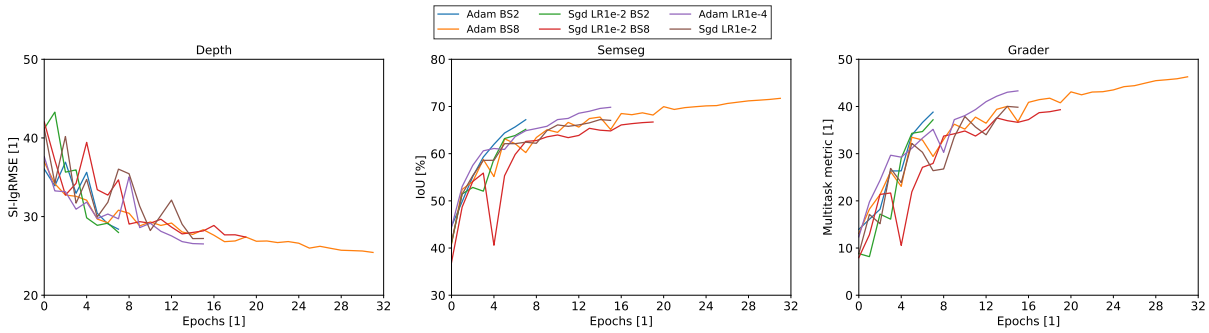

Figure 4: Performance metric analyis of SGD and Adam using different batch sizes.

1.1.2 Batch Size

The runs in the previous Section 1.1.1 were conducted with the default parameters of a batch size (BS) of 4 and 16 epochs. To explore the influence of the batch size we explored additionally batch sizes of 2 and 8. The handout suggested to vary the number of epochs proportionally to the batch size, thus we trained with a batch size of 2 for 8 epochs and with a batch size of 8 for 32 epochs. The outcome is shown in Figure 4. The best performance is achieved by the Adam optimizer with a batch size of 8 and 32 epochs (`depth`: 25.4, `semseg`: 21.7, `grader`: 46.3). Unfortunately, the run with `BS` $= 8$ for SGD terminated prematurely on epoch 20. However, the general trend is already visible and it is very unlikely that it would have outperformed Adam. Also, both runs with `BS` $= 2$ show significant performance gains on the last epoch, so training them for more than 8 epochs would further increase performance. Even though a higher batch size reduces the training time per epoch, the overall training times still varies considerably, as listed in Table 1. As `BS` $= 8$ and 32 epochs are prone to early termination, we chose BS $= 4$ for further experiments.

Parameters	BS $= 2$, epochs $= 8$	BS $= 4$, epochs $= 16$	BS $= 8$, epochs $= 32$
SGD run time	3h 11m	4h 28m	4h 49m*
Adam run time	3h 31m	4h 31m	7h 55m

Table 1: Run times of different batch sizes. The run marked with '*' terminated early.

1.1.3 Task Weighting

Since the problem at hand is a multi-task learning problem we studied the influence of task weighting next. The previous runs were weighted with 50 % on semantic segmentation and 50 % on depth estimation (in the following we express this as '50/50'). Additionally, runs with 30/70, 40/60, 60/40 and 30/70 were conducted. Figure 5 and 6 show the results. As expected, a higher weighting of the semantic segmentation task results in better performance for the respective metric for both Adam and SGD. This hypothesis holds true for the depth estimation task as well. Basically, it can be stated that an increase of performance in one task generally comes at the cost of decreased performance in the other. Considering the performance metric results at epoch 15 using Adam, the 50/50 weighting shows the best results for both semantic

segmentation and depth estimation, and thus, yields the highest overall score. Hence, we continue with this setting.

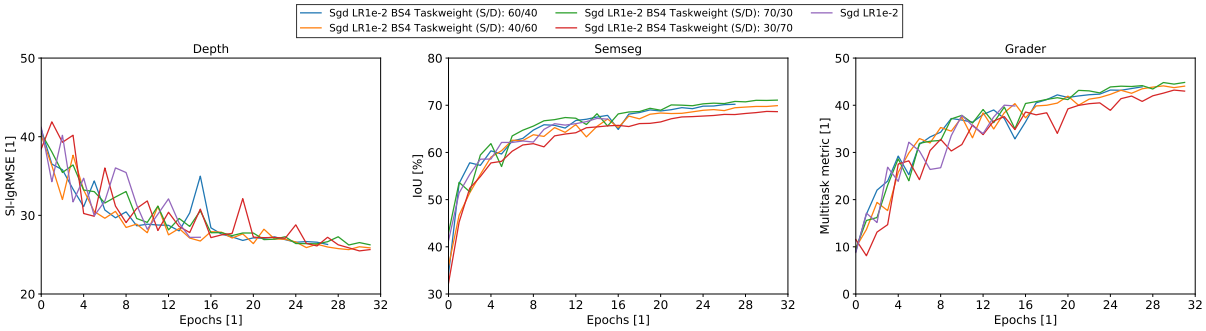

Figure 5: Performance metric analysis of SGD with different task loss weights.

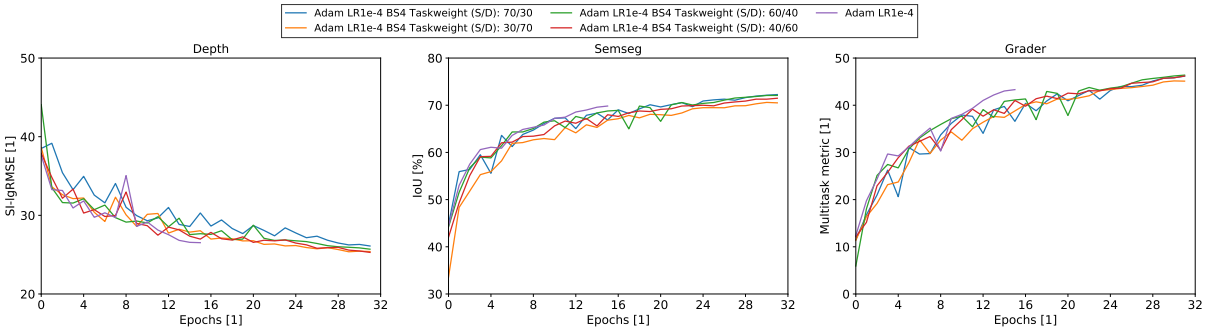

Figure 6: Performance metric analysis of Adam with different task loss weights.

1.2 Hardcoded Hyperparameters: Initialization with ImageNet Weights and Dilated Convolutions

In the default model, the encoder network is not initialized with any pretrained weights. Using initial weights from training on the ImageNet dataset boosts the performance in the first epochs significantly, especially for semantic segmentation, which is shown in Figure 7. The gap narrows as the training progresses. At epoch 15 the advantage of the pretrained nets concerning depth estimation is almost gone compared to the metrics with plain Adam LR $= 10^{-4}$. Furthermore, the performance difference in semantic segmentation of 1.48 % for SGD and 0.81 % for Adam confirms the assumption that an increased training time will eventually allow the untrained networks to converge to similar performances. Additionally, utilizing dilated convolution, which had not been enabled yet, improves performance across all epochs for semantic segmentation and depth estimation for both Adam and SGD. This results in a gain of 10.34 % for SGD and 7.75 % for Adam in the combined grader metric. The final scores are listed in Table 4 for a concise overview of the improvements.
The performance improvements in semantic segmentation using dilated convolution comes from its usage of stride sizes greater than 1. A stride size of 1 is referred to as normal convolution, whereas convolution operations with a stride size greater than 1 are so-called dilated convolutions. By increasing the stride the receptive field of the convolution operation is increased while keeping the number of parameters constant. Due to the increased receptive field the dilated convolution operation can extract more spatial information which helps boost semantic segmentation performance.

1.3 ASPP and Skip Connections

For this task we were asked to implement the ASPP module with a decoder containing skip connections [1, 2]. Our implementation of the ASPP module is shown in Listing 1. The code template provides a module called `ASPPpart` which combines a 2D convolution with batch normalization and a ReLu activation function. According to the referenced paper we created 4 convolutional layers based on the

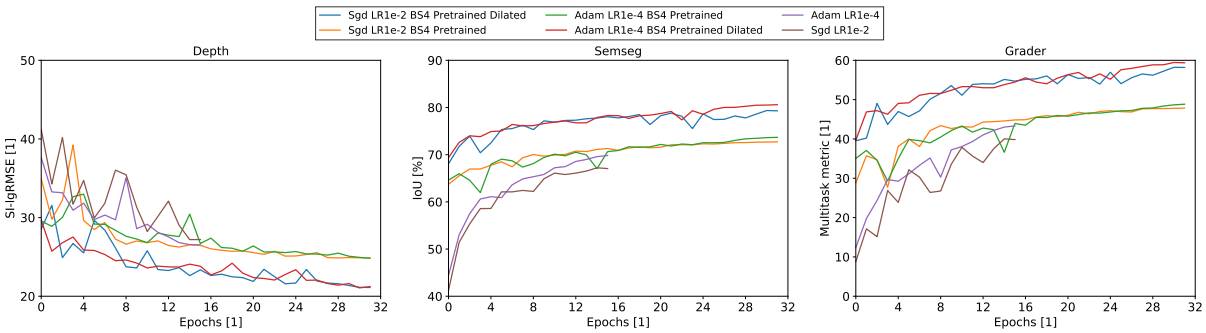

Figure 7: Performance metric analysis of SGD and Adam with dilated convolution and/or pretrained ResNet weights. Performances of SGD and Adam without dilated convolution and pretrained weights are displayed for comparison.

`ASPPpart` module. The first one features `kernel_size` $= 1$ and `stride` $= 1$. The other three are all 3×3 convolutions with strides of 3, 6, and 9, respectively. Additionally, there is an image pooling layer that we constructed using PyTorch's `AdaptiveAvgPool2d()` function with a subsequent 1×1 convolution and batch normalization. The 4 convolutions and the image pooling are separately applied on the output of the ResNet-34 [3]. The 5 resulting tensors are concatenated and a final 3×3 convolution is applied on this tensor.

```
class ImagePooling(torch.nn.Sequential):
    def __init__(self, in_channels, out_channels):
        super().__init__(
            torch.nn.AdaptiveAvgPool2d(1),
            torch.nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=1, padding=0, bias=False),
            torch.nn.BatchNorm2d(out_channels),
        )

class ASPP(torch.nn.Module):

    def __init__(self, in_channels, out_channels, rates=(3, 6, 9)):
        super().__init__()
        # 1x1 convolution.
        self.conv1 = ASPPpart(in_channels, out_channels, kernel_size=1, stride=1, padding=0, dilation=1)
        # Dilated convolutions.
        self.conv_r1 = ASPPpart(in_channels, out_channels, kernel_size=3, stride=1, padding=3, dilation=3)
        self.conv_r2 = ASPPpart(in_channels, out_channels, kernel_size=3, stride=1, padding=6, dilation=6)
        self.conv_r3 = ASPPpart(in_channels, out_channels, kernel_size=3, stride=1, padding=9, dilation=9)
        # Image pooling.
        self.im_pool = ImagePooling(in_channels=in_channels, out_channels=out_channels)
        # 1x1 convolution before hand-over to the decoder (is fed the concatenated tensor from the ASPP module).
        self.conv_out = ASPPpart(out_channels*5, out_channels, kernel_size=1, stride=1, padding=0, dilation=1)

    def forward(self, x):
        # 1x1 convolution.
        c1 = self.conv1(x)
        # Dilated convolutions.
        cr1 = self.conv_r1(x)
        cr2 = self.conv_r2(x)
        cr3 = self.conv_r3(x)
        # Image pooling.
        h = x.shape[2]
        w = x.shape[3]
        ip = F.interpolate(self.im_pool(x), size=(h, w), mode='bilinear', align_corners=False)
        # Concatenate individual tensors.
        concat_tensor = torch.cat((c1, cr1, cr2, cr3, ip), 1)
        # Convolve all ASPP tensors.
```

```
38          out = self.conv_out(concat_tensor)
39          return out
```

Listing 1: ASPP module implementation

Listing 2 presents the decoder implementation. The decoder directly takes low-level features from the encoder module and applies a 1×1 convolution on those channels. The resulting tensor is concatenated with the upsampled (by a factor of 4) output of the ASPP module. Finally, a 3×3 convolution is applied on the concatenated tensor.

```
1  class DecoderDeeplabV3p(torch.nn.Module):
2      def __init__(self, bottleneck_ch, skip_4x_ch, num_out_ch):
3          super(DecoderDeeplabV3p, self).__init__()
4          # 3x3 on concatenated feature maps.
5          self.conv_3x3 = torch.nn.Conv2d(bottleneck_ch+skip_4x_ch, num_out_ch, kernel_size
       =3, stride=1, padding=1)
6
7      def forward(self, features_bottleneck, features_skip_4x):
8          """
9          DeepLabV3+ style decoder
10         :param features_bottleneck: bottleneck features of scale > 4
11         :param features_skip_4x: features of encoder of scale == 4
12         :return: features with 256 channels and the final tensor of predictions
13         """
14         # Upsample ASPP output 4x.
15         features_4x = F.interpolate(features_bottleneck, size=features_skip_4x.shape[2:],
        mode='bilinear', align_corners=False)
16         # Tensor concatination.
17         concat_tensor = torch.cat((features_4x, features_skip_4x), 1)
18         # 3x3 Convolution
19         predictions_4x = self.conv_3x3(concat_tensor)
20         return predictions_4x, features_4x
```

Listing 2: Decoder implementation

The effects of the ASPP module for both optimizers are depicted in Figure 8. On the one hand, training with Adam and the ASPP implementation exhibits a significant improvement of 5.31 % in performance for the semantic segmentation task. On the other hand, the depth estimation performance deteriorated quite substantially. The obtained results clearly reveal the strength of the ASPP module in semantic segmentation. In order to utilize the advantages gained by the ASPP module and eliminate the unstable behaviour in depth estimation a branched network architecture, as discussed in the following, was implemented.

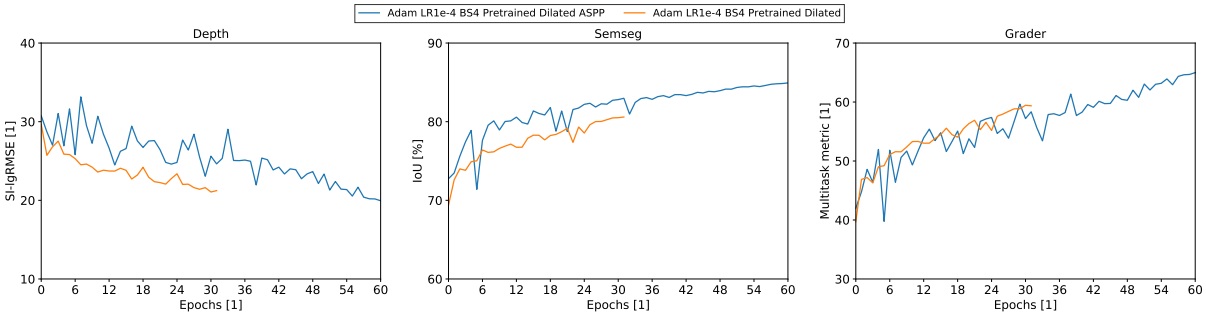

Figure 8: Performance metric analysis of Adam with and without the correct implementation of the ASPP and decoder modules.

2 Problem 2: Branched Architecture

A common approach to tackle multi task learning problems is to adopt a branched network architecture, which makes use of one shared encoder module, but two decoders, using one for each task. The branched network architecture was implemented in the file `model_deeplab_v3_plus_branched.py` and is shown below. The code for the shared encoder is identical to all previous tasks. However, the encoder output is then fed separately into the two decoders, where one is trained on semantic segmentation whereas the

other one is trained on depth estimation. Implementation details are provided in Listings 3 and 4.

```python
class ModelDeepLabV3PlusBranched(torch.nn.Module):

    def __init__(self, cfg, outputs_desc):
        super().__init__()

        self.outputs_desc = outputs_desc
        ch_out_segm = outputs_desc['semseg']
        ch_out_depth = outputs_desc['depth']

        self.encoder = Encoder(
            cfg.model_encoder_name,
            # Option to use pretrained ResNet labels.
            pretrained=True,
            zero_init_residual=True,
            # Option to use dilated conv.
            # ResNet layers (2, 3, 4).
            replace_stride_with_dilation=(False, False, True),
        )

        ch_out_encoder_bottleneck, ch_out_encoder_4x = get_encoder_channel_counts(cfg.model_encoder_name)

        # Semantic segmentation head.
        self.aspp_segm = ASPP(ch_out_encoder_bottleneck, 256)
        self.decoder_segm = DecoderDeeplabV3p(256, ch_out_encoder_4x, ch_out_segm)

        # Depth estimation head.
        self.aspp_depth = ASPP(ch_out_encoder_bottleneck, 256)
        self.decoder_depth = DecoderDeeplabV3p(256, ch_out_encoder_4x, ch_out_depth)
```

Listing 3: `__init__()` function in `model_deeplab_v3_plus_branched.py`.

In the `forward(x)` function we first pass the image through the encoder as before. The resulting low-level features are then fed to each ASPP module, whose output is further used as input for the decoders together with the low-level features extracted from the encoder. Finally, the output of each decoder is upsampled by a factor of 4 to obtain the final predictions. The code changes that accomplish this behavior are given in Listing 4.

```python
    def forward(self, x):
        input_resolution = (x.shape[2], x.shape[3]) # Height, width

        # Encoder.
        features = self.encoder(x)

        # Get features from encoder module.
        lowest_scale = max(features.keys())
        features_lowest = features[lowest_scale]

        # Branched network decoders.

        # Segmentation head.
        features_segm = self.aspp_segm(features_lowest)
        predictions_4x_segm, _ = self.decoder_segm(features_segm, features[4])
        predictions_1x_segm = F.interpolate(predictions_4x_segm, size=input_resolution, mode='bilinear', align_corners=False)

        # Depth estimation head.
        features_depth = self.aspp_depth(features_lowest)
        predictions_4x_depth, _ = self.decoder_depth(features_depth, features[4])
        predictions_1x_depth = F.interpolate(predictions_4x_depth, size=input_resolution, mode='bilinear', align_corners=False)

        # Return predictions.
        out = {
            'semseg': predictions_1x_segm,
            'depth': predictions_1x_depth,
        }
        return out
```

Listing 4: `forward(x)` function in `model_deeplab_v3_plus_branched.py`.

The performance of the branched architecture can be examined in Figure 9. Evidently, the branched architecture offers significant performance gains for depth estimation and ultimately manages to get rid of the drawbacks of the ASPP module concerning this task. Therefore, the improvements in depth estimation help boost the overal metric performance. Training time increased slightly for the branched architecture by 7.82 % (1476 min vs. 1369 min for 64 epochs) due to the separate network branches and the additional number of parameters that need to be trained. Regarding the network parameters, the branched architecture has twice the number of ASPP module and decoder parameters compared to the single branch module. The number of encoder parameters remains the same. The final performance of the branched architecture is: `grader`: 65.82, `semseg`: 85.28, `depth`: 19.46.

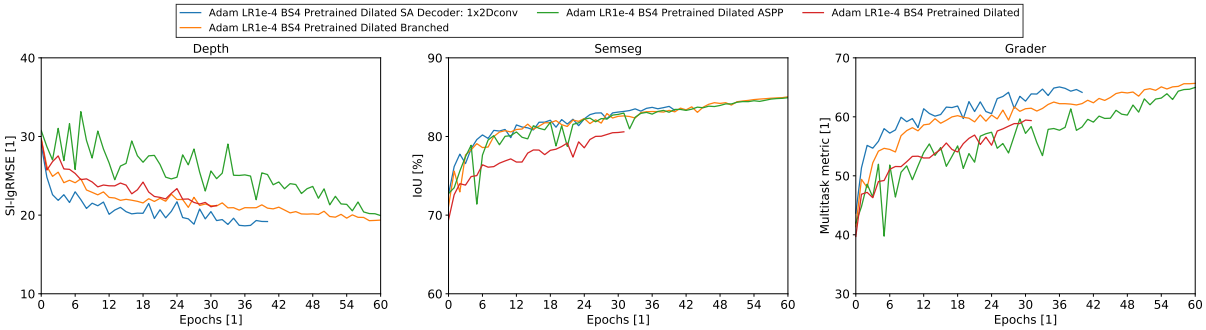

Figure 9: Performance metric analysis using Adam. The branched network architecture is compared with the combined decoder architecture using the 'untouched' provided ASPP and decoder modules as well as the correctly implemented ones.

3 Problem 3: Task Distillation

Inspired by the results in [4], we further extended the model described in Section 2 with a distillation stage based on the provided self attention (SA) module. The principle idea is that knowledge about depth can be relevant for semantic segmentation and vice versa. Hence, we use the semantic segmentation estimation of the original decoder #1 and combine it with features based on the depth prediction of the original decoder #2. However, not all the depth data is important so the depth feature maps are first passed through the SA module. Afterwards, an element-wise addition of the semantic segmentation decoder (decoder #1) output tensor with the SA module output tensor for depth estimation is performed.
The self attention module consists of two 3×3 convolutional layers that are independently applied on the input feature maps. A sigmoid function is applied on only one of the two convolution output tensors to obtain values between 0 and 1 for every single feature value. Finally, an element-wise multiplication of the two tensors is performed. The tensor with values between 0 and 1 acts as a mask weighting important features of the other tensor stronger than less important ones. This weighting is at the core of the "self-attention" as one output tensor takes care of which feature values of the depth estimation network branch are indeed valuable to feed forward through the network and combine with the semantic segmentation branch. The combined feature maps (depth estimation features fed through the SA module and semantic segmentation features resulting from decoder #1) are then fed into another decoder #3, which has the same architecture as the original ones, in order to predict the pixel-wise labelling. For depth estimation an analogue procedure is applied. The semantic segmentation data of decoder #1 is passed through a SA module and is then element-wise summed with the output features of the first decoder of the depth estimation branch (decoder #2). This tensor is then forward propagated through decoder #4 to obtain the depth estimation results. To evaluate network performance the losses of the original network architecture (loss #1 and #2) are combined with the losses of the self attention network heads (loss #3 and #4). Specifically, the mean of loss #1 and #2 is computed as final semantic segmentation loss, whereas the mean of loss #3 and #4 serves as final evaluation metric for depth estimation. The code for this implementation is shown in Listings 5 and 6, the discussed network architecture is further provided in Figure 10.

In Figure 9 one can see the performance of the extended self-attention architecture using Adam in comparison with previous runs. It can be observed that performance on semantic segmentation has indeed reached its maximum with the sole implementation of the ASPP module. Neither the branched network

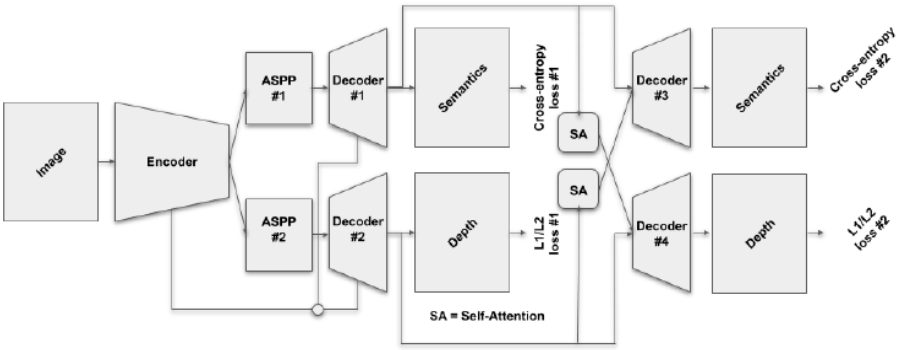

Figure 10: Branched architecture with with distillation task.

architecture nor the distilled architecture were able to push semantic segmentation scores significantly further. In contrast, a noteworthy improvement regarding depth estimation has been achieved: 19.18 vs. 20.89 at epoch 40. Unfortunately, the training crashed at epoch 40 so that the performances after 64 epochs cannot be contrasted. Nonetheless, the strengths of the SA module are already visible. By extracting features from the semantic segmentation branch it is possible to boost the depth estimation performance even further.
Another point that is worth mentioning concerning the training of the different network architectures is train time. As can be seen from Table 2, the implementation of the ASPP module with its parallel convolution layers as described in Section 1 shows no run time penalty compared to the initial architecture provided by the instructors. Moreover, it can be stated that a branched architecture with its clear performance gains in depth estimation comes with a slight run time penalty of 12.12 %. Whereas the run time of the plain architecture (provided by the instructors), the implementation of the ASPP module, and the branched network show similar run times, making use of the distilled architecture reveals a clear drawback as far as train time is concerned. At 64 % of the epochs the distilled architecture surpasses the train time of the branched implementation already by 11 %. This is as expected due to the introduction of additional network parameters by implementing two SA modules and two additional decoders (decoder #3 and #4), where each decoder has the same number of parameters as decoder #1 respectively decoder #2.

Architecture	Plain	ASPP	Branched	Distilled
Epochs	32	64	64	41*
Run time	13h 42m	22h 50m	25h 36m	28h 25m

Table 2: Run times of the different tested network architectures using Adam. The run marked with '*' terminated early.

The following Listings 5, 6, and 7 show the Python implementation of task 3.

```
class ModelDeepLabV3PlusBranchedSA(torch.nn.Module):

    def __init__(self, cfg, outputs_desc):
        super().__init__()

        self.outputs_desc = outputs_desc
        ch_out_segm = outputs_desc['semseg']
        ch_out_depth = outputs_desc['depth']

        self.encoder = Encoder(
            cfg.model_encoder_name,
            # Option to use pretrained ResNet labels.
            pretrained=True,
            zero_init_residual=True,
            # Option to use dilated conv.
            # ResNet layers (2, 3, 4).
            replace_stride_with_dilation=(False, False, True),
```

```
        )

        ch_out_encoder_bottleneck, ch_out_encoder_4x = get_encoder_channel_counts(cfg.model_encoder_name)

        # Semantic segmentation head.
        self.aspp_segm = ASPP(ch_out_encoder_bottleneck, 256)
        # Decoder of the first network part, combine ASPP output with feature map from shared encoder.
        self.decoder_1_segm = DecoderDeeplabV3pSA(256, ch_out_encoder_4x, ch_out_segm)
        # Self-attention (SA) module.
        self.sa_segm = SelfAttention(256+ch_out_encoder_4x, 256+ch_out_encoder_4x)
        # Decoder of the second network part, combine SA depth output with decoder_skip_segm output.
        self.decoder_2_segm = DecoderDeeplabSA(256+ch_out_encoder_4x, ch_out_segm)

        # Depth estimation head.
        self.aspp_depth = ASPP(ch_out_encoder_bottleneck, 256)
        # Decoder of the first network part, combine ASPP output with feature map from shared encoder.
        self.decoder_1_depth = DecoderDeeplabV3pSA(256, ch_out_encoder_4x, ch_out_depth)
        # Self-attention (SA) module.
        self.sa_depth = SelfAttention(256+ch_out_encoder_4x, 256+ch_out_encoder_4x)
        # Decoder of the second network part, combine SA segm output with decoder_skip_depth output.
        self.decoder_2_depth = DecoderDeeplabSA(256+ch_out_encoder_4x, ch_out_depth)
```

Listing 5: `__init__()` function in `model_deeplab_v3_plus_sa.py`.

```
    def forward(self, x):
        input_resolution = (x.shape[2], x.shape[3]) # Height, width

        # Encoder.
        features = self.encoder(x)

        # Get features from encoder module.
        lowest_scale = max(features.keys())
        features_lowest = features[lowest_scale]

        # Branched network decoders.

        # First network part: shared encoder, and split ASPP, decoder, and SA modules.
        # Segmentation head.
        features_segm = self.aspp_segm(features_lowest)
        predictions_4x_segm_1, features_4x_segm = self.decoder_1_segm(features_segm, features[4])
        predictions_1x_segm_1 = F.interpolate(predictions_4x_segm_1, size=input_resolution, mode='bilinear', align_corners=False)
        sa_features_segm = self.sa_segm(features_4x_segm)
        # Depth estimation head.
        features_depth = self.aspp_depth(features_lowest)
        predictions_4x_depth_1, features_4x_depth = self.decoder_1_depth(features_depth, features[4])
        predictions_1x_depth_1 = F.interpolate(predictions_4x_depth_1, size=input_resolution, mode='bilinear', align_corners=False)
        sa_features_depth = self.sa_depth(features_4x_depth)

        # Second network part: Decoder (combines SA features + first network part features) and final prediction.
        # Segmentation head.
        predictions_4x_segm_2 = self.decoder_2_segm(features_4x_segm, sa_features_depth)
        predictions_1x_segm_2 = F.interpolate(predictions_4x_segm_2, size=input_resolution, mode='bilinear', align_corners=False)
        # Depth estimation head.
        predictions_4x_depth_2 = self.decoder_2_depth(features_4x_depth, sa_features_segm)
        predictions_1x_depth_2 = F.interpolate(predictions_4x_depth_2, size=input_resolution, mode='bilinear', align_corners=False)

        # Return predictions.
        # Values are lists of both network part predictions. The loss is computed for both predictions individually.
        # The losses are then summed and averaged for a final loss value.
```

```
        # See file experiment_semseg_with_depth.py, function training_step(self, batch, batch_nb).
        out = {
            'semseg': [predictions_1x_segm_1, predictions_1x_segm_2],
            'depth': [predictions_1x_depth_1, predictions_1x_depth_2],
        }
        return out
```

Listing 6: `forward(x)` function in `model_deeplab_v3_plus_sa.py`.

```
""" Task 3. First decoder of the SA module. """

class DecoderDeeplabV3pSA(torch.nn.Module):
    def __init__(self, bottleneck_ch, skip_4x_ch, num_out_ch):
        super(DecoderDeeplabV3pSA, self).__init__()
        # DCNN 1x1 convolution (only necessary if the no. of channels should be increased from 64 to 256).
        # self.dcnn_conv = torch.nn.Conv2d(64, 256, kernel_size=1, stride=1)
        # 3x3 on concatenated feature maps.
        self.conv_3x3 = torch.nn.Conv2d(bottleneck_ch+skip_4x_ch, num_out_ch, kernel_size=3, stride=1, padding=1)

    def forward(self, features_bottleneck, features_skip_4x):
        """
        DeepLabV3+ style decoder
        :param features_bottleneck: bottleneck features of scale > 4
        :param features_skip_4x: features of encoder of scale == 4
        :return: features with 256 channels, output for the SA module (concat tensor)
        """
        # Upsample ASPP output 4x.
        features_4x = F.interpolate(features_bottleneck, size=features_skip_4x.shape[2:], mode='bilinear', align_corners=False)
        # DCNN 1x1 convolution (only necessary if the no. of channels should be increased from 64 to 256).
        # features_dcnn = self.dcnn_conv(features_skip_4x)
        # Tensor concatination.
        concat_tensor = torch.cat((features_4x, features_skip_4x), 1)
        # 3x3 Convolution
        predictions_4x = self.conv_3x3(concat_tensor)

        return predictions_4x, concat_tensor


""" Task 3. Second decoder of the SA module, simple 3x3 conv. """

class DecoderDeeplabSA(torch.nn.Module):
    def __init__(self, num_in_ch, num_out_ch):
        super(DecoderDeeplabSA, self).__init__()
        # 3x3 on concatenated feature maps.
        self.conv_3x3 = torch.nn.Conv2d(num_in_ch, num_out_ch, kernel_size=3, stride=1, padding=1)

    def forward(self, features_4x_decoder, features_4x_sa):
        """
        DeepLabV3+ style decoder
        :param features_4x_decoder: features of encoder of scale == 4
        :param features_4x_sa: features of the sa module, same resolution as features_4x_decoder
        :return: final tensor of predictions
        """
        features_4x_decoder_sa = features_4x_decoder + features_4x_sa
        return self.conv_3x3(features_4x_decoder_sa)
```

Listing 7: Decoder models for the first stage (#1 and #2) and second stage (#3 and #4) of the distilled network architecture in `model_parts.py`.

4 Problem 4: Winning the Leaderboard

Based on [4], we tried further modifications to the second stage of decoder modules. The architecture in the previous Section 3 utilizes a sole 2D convolution - as recommended in the instructions sheet. We created a new variant that implements three 2D convolutions. After both the first and the second convolution the feature map is up-sampled by a factor of 2, whereas the depth is decreased by a factor of 2. The implementation details can be seen in Listing 8. For the sake of a quicker comparison, we decided to start with BS=2 to evaluate whether this deeper decoder architecture leads to a performance increase. Performance using BS=4 with the simple 2D convolution is provided to show the influence of the batch size. The performance metrics are plotted in Figure 11 for both Adam and SGD. On the one hand, one can see a performance increase for both depth and semantic segmentation when using SGD as optimizer. On the other hand, there is no noteworthy performance gain using Adam. Nevertheless, the runs with Adam achieve overall the better performance for BS=2. Another vital point to stress here is that using BS=4 with Adam as optimizer clearly outperforms runs conducted with BS=2. From the plots it can be further inferred that Adam is clearly the better optimizer choice for the distilled architecture with BS=4.

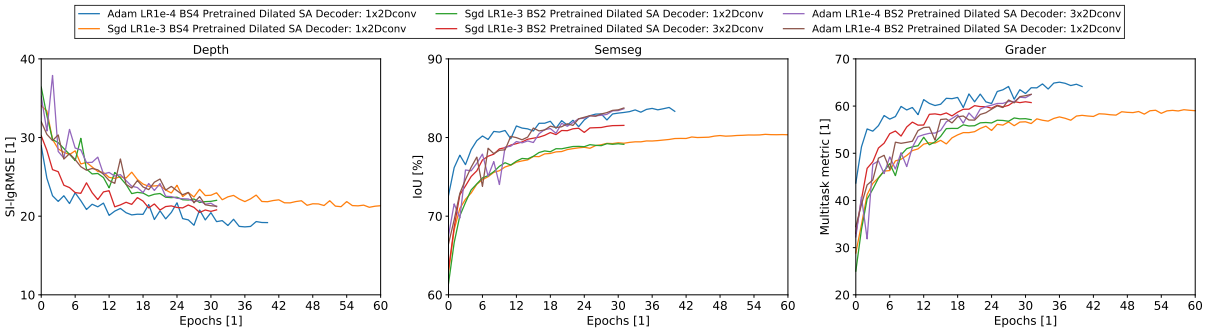

Figure 11: Performance metric analysis of SGD and Adam using two different variants for the decoder module. One decoder architecture consists of one sole 2D convolution - as provided in the template code. The second variant implements three 2D convolutions. After both the first and second convolution the feature map is upsampled by a factor of 2 and channel depth is decreased by a factor of 2. This implementation is similar to the one proposed in [4].

Considering the train times of the different second stage decoder architectures it should be noted that the deeper decoder architecture increased train time for SGD by 75 %. Due to the increased number of parameters this can be considered as an expected behaviour. Interestingly, the train time with Adam as optimizer increased only by 20 min. This behaviour was unexpected and another run with the same settings was run to confirm the outcome. This second run took a similar amount of time (see table below).

Decoder Architecture	1x2D	3x2D	1x2D	3x2D	3x2D (confirmation)
Optimizer	SGD	SGD	Adam	Adam	Adam
Run time	22h 6m	38h 34m	15h 34m	15h 54m	15h 41m

Table 3: Run times of the different tested network architectures using Adam and SGD with BS=2 and Epochs=32.

```
""" Task 3. Second decoder of the SA module, deeper version with upscaling """

class DecoderConvUpscaleLayer(torch.nn.Sequential):
    def __init__(self, in_channels, out_channels, kernel_size, stride, padding):
        super().__init__(
            torch.nn.Conv2d(in_channels, out_channels, kernel_size=kernel_size, stride=
    stride, padding=padding, bias=False),
            torch.nn.BatchNorm2d(out_channels),
            torch.nn.ReLU(),
        )

class DecoderDeeplabSADeeper(torch.nn.Module):
    def __init__(self, num_in_ch, num_out_ch):
        super(DecoderDeeplabSADeeper, self).__init__()
```

```
        # 3x3 conv. with upscaling and channel reduction.
        self.conv_upscale1 = DecoderConvUpscaleLayer(num_in_ch, int(num_in_ch/2),
    kernel_size=3, stride=1, padding=1)
        self.conv_upscale2 = DecoderConvUpscaleLayer(int(num_in_ch/2), int(num_in_ch/4),
    kernel_size=3, stride=1, padding=1)
        # 3x3 final conv.
        self.conv_3x3 = torch.nn.Conv2d(int(num_in_ch/4), num_out_ch, kernel_size=3,
    stride=1, padding=1)

    def forward(self, features_4x_decoder, features_4x_sa, output_resolution):
        """
        DeepLabV3+ style decoder modified according the pad-net paper.
        The input layer is run through a 3x3 conv. and then upscaled by a factor of 2
    while
        reducing the channel size by a factor of 2. This is done twice. Therefore, the
    output
        resolution is reached.
        Finally, a 3x3 conv. is run decreasing the channel output size to the target
    output size.
        :param features_4x_decoder: features of encoder of scale == 4
        :param features_4x_sa: features of the sa module, same resolution as
    features_4x_decoder
        :output_resolution: 2 tuple of ints (height, width)
        :return: final tensor of predictions
        """
        # Add the inputs from the first decoder and self attention module element-wise.
        features_4x_decoder_sa = features_4x_decoder + features_4x_sa
        # Resolution for the first 2x upscaling.
        intermediate_resolution = (int(output_resolution[0]/2), int(output_resolution
    [1]/2))
        # 3x3 conv and 2x upscaling.
        out = self.conv_upscale1(features_4x_decoder_sa)
        out = F.interpolate(out, size=intermediate_resolution, mode='bilinear',
    align_corners=False)
        out = self.conv_upscale2(out)
        out = F.interpolate(out, size=output_resolution, mode='bilinear', align_corners=
    False)
        # Final 3x3 conv. layer (note: no ReLu and Batchnorm anymore -> final predictions
    ).
        out = self.conv_3x3(out)
        return out
```

Listing 8: Decoder with three 2D conv layers implementation in `model_parts.py`.

5 Summary

Lastly, in Table 4 we provide an overview of the performances of the different architectures achieved on the validation set. Table 5 displays the results obtained on Codalab. We further included sample images in Figure 12 to 16 to visualize the actual input, ground truth, and predictions of all the different architectures implemented in this project.

Optimizer	Epochs	Pretrained	Dilation	ASPP	Branched	Distilled	Grader	Semseg	Depth
SGD	16						39.84	67.05	27.23
Adam	16						43.31	69.83	26.55
SGD	32	✓					47.85	72.71	24.86
Adam	32	✓					48.84	73.67	24.83
SGD	32	✓	✓				58.18	76.30	21.11
Adam	32	✓	✓				59.36	80.59	21.23
Adam	64	✓	✓	✓			65.21	85.15	19.94
Adam	64	✓	✓	✓	✓		65.82	85.28	19.46
SGD	64	✓	✓	✓	✓	✓	59.15	80.37	21.23
Adam	41	✓	✓	✓	✓	✓	64.15	83.83	19.18

Table 4: Comparison of the different benefits of each architecture with BS=4 and a 50/50 weighting of tasks.

Optimizer	BS	Weight	Epochs	Pretrained	Dillated	ASPP	Branched	Distilled	Grader
Task 1_1a: G46_0406-0948_adam_lr_1e-4_4f5fb									
Adam	4	50/50	16						43.14
Task 1_1b: G46_0426-2155_adam_bs_8_epochs_32_bf927									
Adam	8	50/50	32						46.07
Task 1_1c: G46_0514-1934_adam_lr_1e-4_bs_4_epochs_32_wsegdep_60_40_fa7f9									
Adam	4	60/40	32						46.34
Task 1_2a: G46_0516-2246_adam_lr_1e-4_bs_4_pretr_Tr_epochs_32_005cb									
Adam	4	50/50	32	✓					48.69
Task 1_2b: G46_0511-0821_adam_lr_1e-4_bs_4_pretr_Tr_dilconv_Tr_epochs_32_0da56									
Adam	4	50/50	32	✓	✓				59.40
Task 1_3: G46_0511-0827_adam_lr_1e-4_bs_4_pretr_Tr_dilconv_Tr_aspp_decode_epochs_64_564af									
Adam	4	50/50	64	✓	✓	✓			65.19
Task 2: G46_0512-0808_adam_lr_1e-4_bs_4_pretr_Tr_dilconv_Tr_branched_epochs_64_255c1									
Adam	4	50/50	64	✓	✓	✓	✓		65.76
Task 3: G46_0519-1559_adam_lr_1e-4_bs_4_pretr_Tr_dilconv_Tr_sa_loss_avg_dcd_3conv_epochs_64_8886e									
Adam	4	50/50	64	✓	✓	✓	✓	✓	66.76

Table 5: CodaLab performance metrics for the 8 different problems.

Appendix

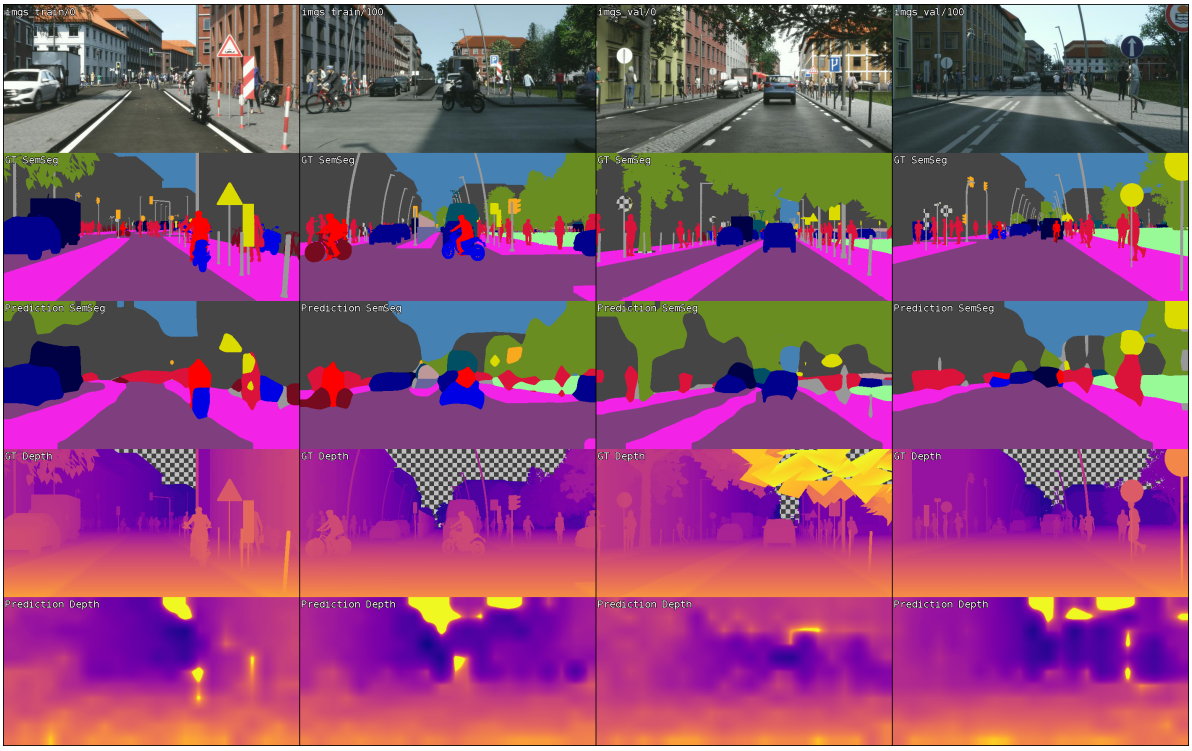

Figure 12: Input, ground truths and predictions for both tasks for 4 exemplary images. The predictions were generated from the default architecture trained with Adam and LR $= 10^{-4}$ described in Subsection 1.1.1.

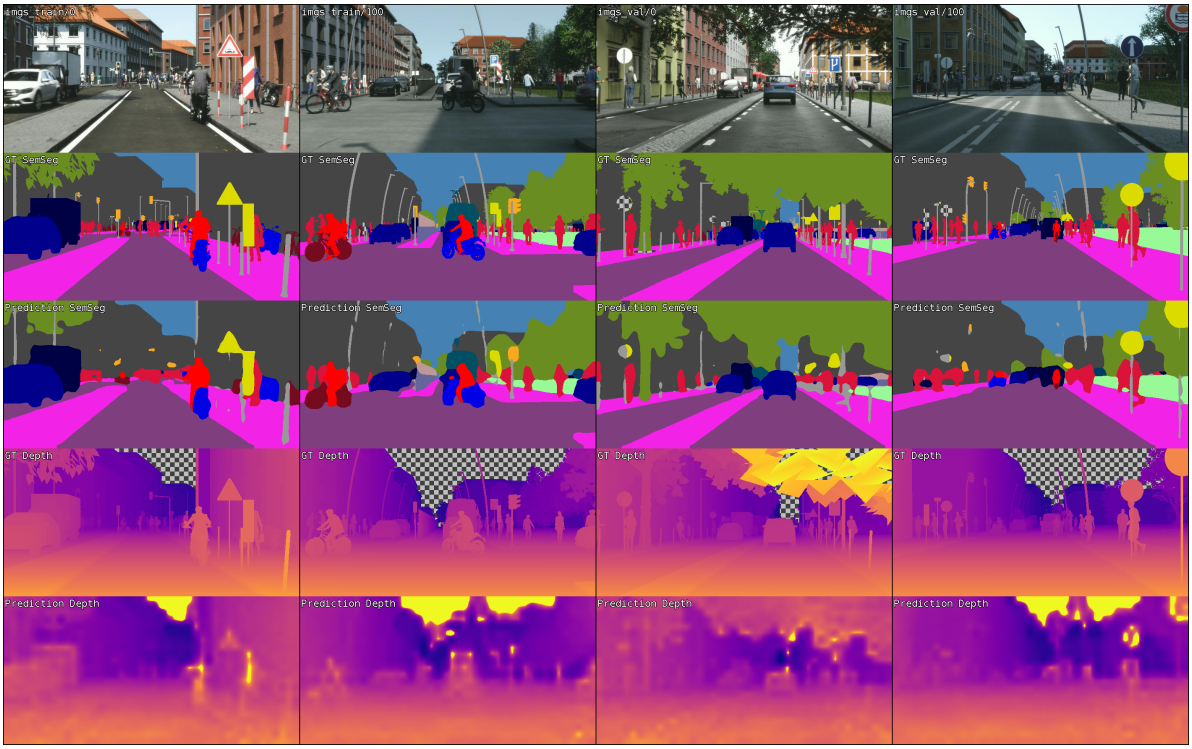

Figure 13: Input, ground truths and predictions for both tasks for 4 exemplary images. The predictions were generated from the default architecture with pre-trained weights and dilated convolutions described in Subsection 1.2. Optimizer is Adam and LR $= 10^{-4}$.

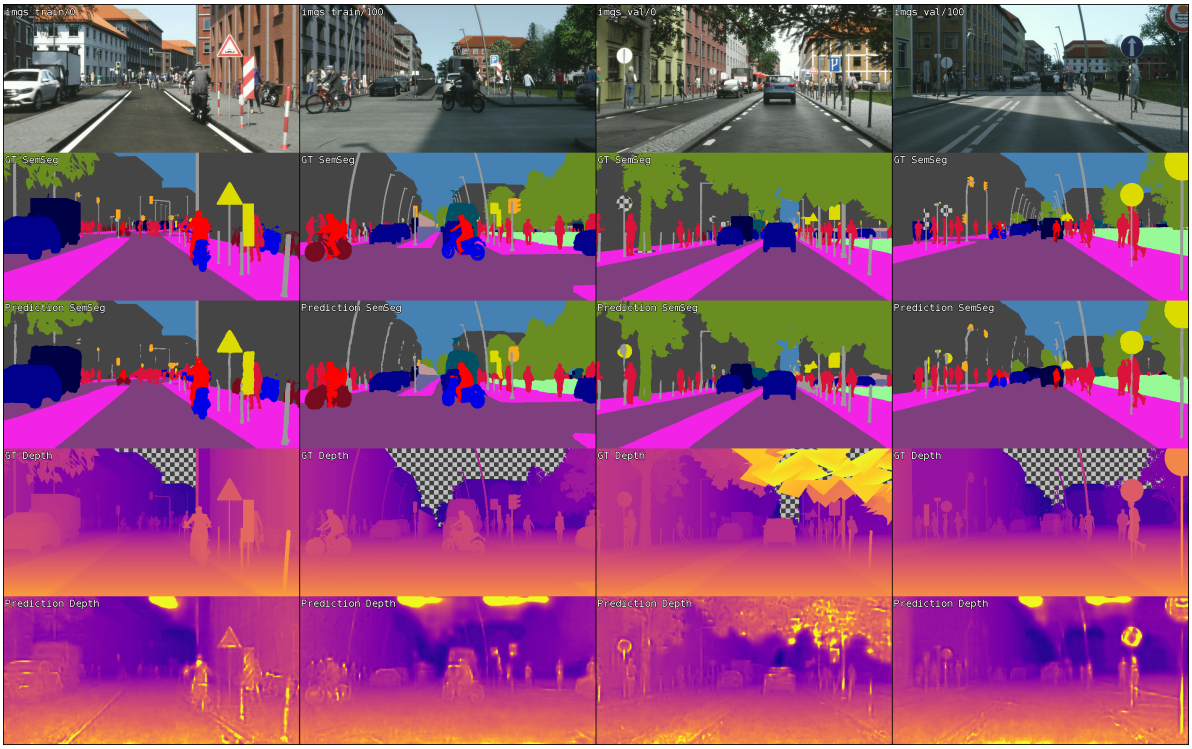

Figure 14: Input, ground truths and predictions for both tasks for 4 exemplary images. The predictions were generated from the ASPP architecture described in Subsection 1.3. Optimizer is Adam and LR $= 10^{-4}$.

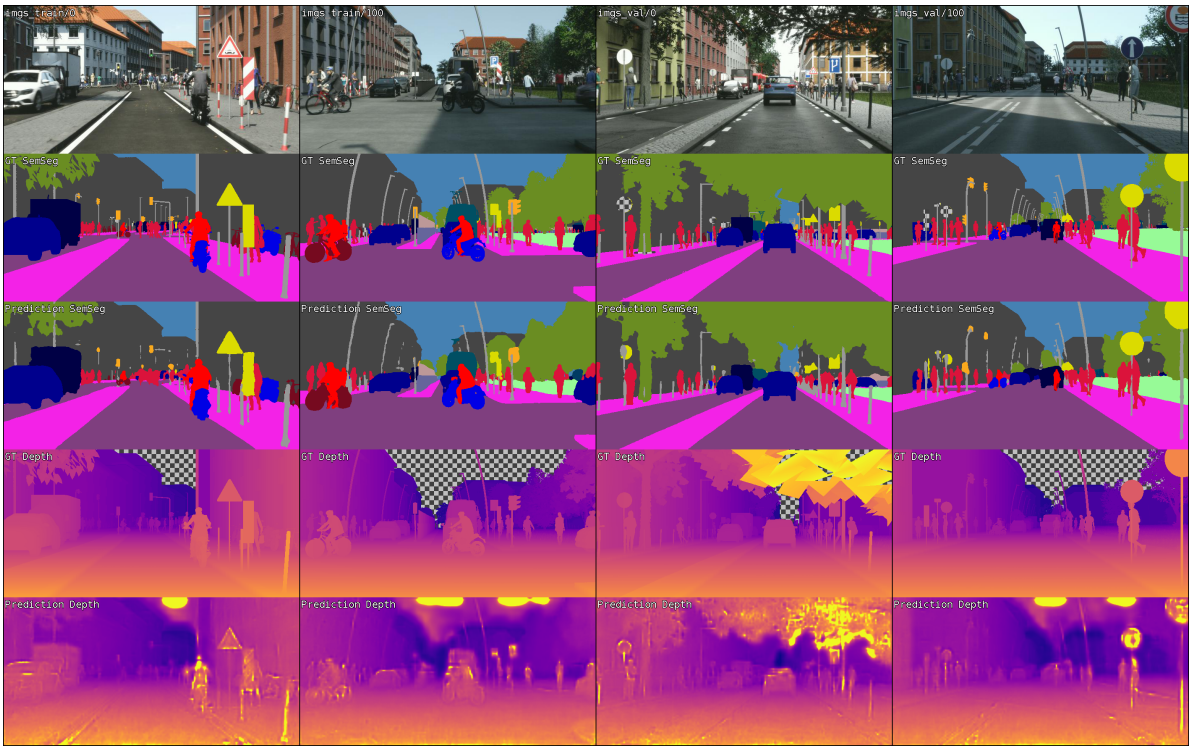

Figure 15: Input, ground truths and predictions for both tasks for 4 exemplary images. The predictions were generated from the branched architecture described in Section 2. Optimizer is Adam and LR = 10^{-4}.

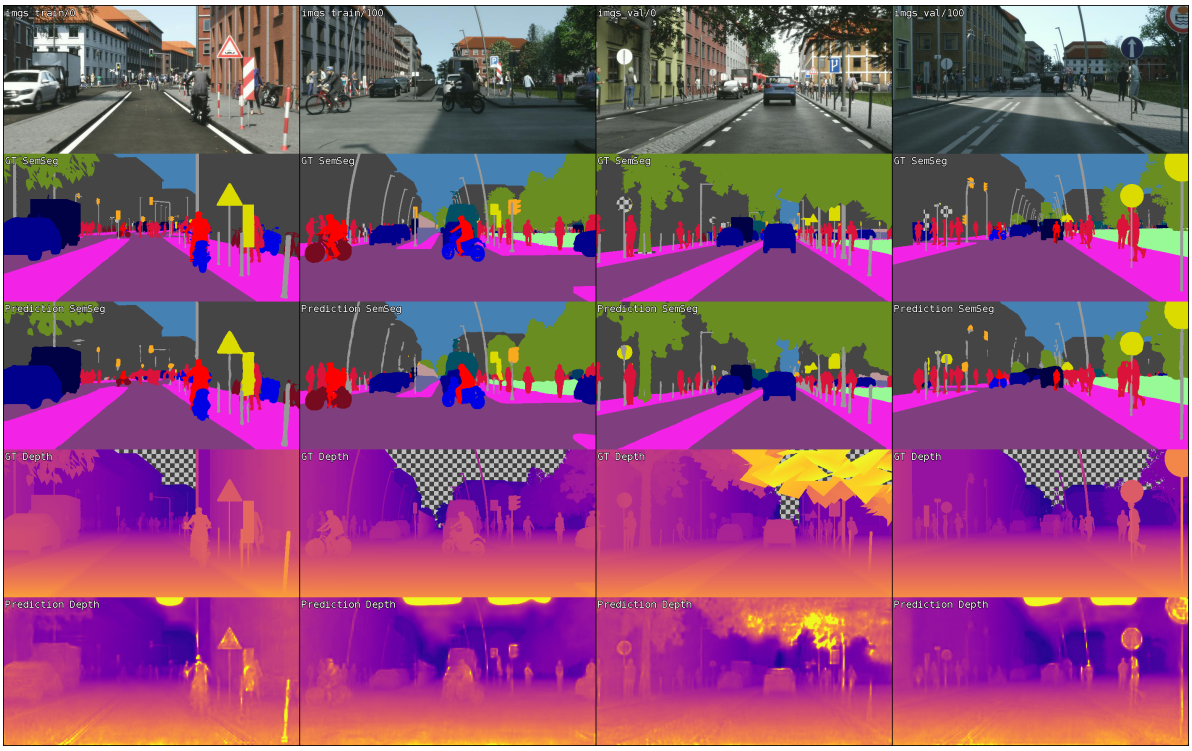

Figure 16: Input, ground truths and predictions for both tasks for 4 exemplary images. The predictions were generated from the distilled architecture described in Section 3. Optimizer is Adam and LR = 10^{-4}.

References

[1] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *CoRR*, abs/1706.05587, 2017.

[2] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. *CoRR*, abs/1802.02611, 2018.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[4] Dan Xu, Wanli Ouyang, Xiaogang Wang, and Nicu Sebe. Pad-net: Multi-tasks guided prediction-and-distillation network for simultaneous depth estimation and scene parsing. *CoRR*, abs/1805.04409, 2018.