DLAD Project 2

Multi-task learning for semantics and depth

Nicolas Hoischen

Student ID: 16-819-880

Dario Bolli

Student ID: 16-832-685

FS 2021

227-0560-00L: Deep Learning for Autonomous Driving

Eidgenössische Technische Hochschule Zürich

Due Date: 13-05-2021

1 Joint architecture

1.1 Hyper-parameter Tuning

1.1.1 The optimizer and LR choice

The learning rate is an important hyperparameter which can range between 0.0 and 1.0. Tiny learning rates gives smaller changes to the weights at each update and require many updates before reaching the minimum point whereas larger learning rates tends to converge quickly but can cause drastic updates leading to divergent behaviours.
This is why we decrease the learning rate with the number of steps. This decrease factor depends on the number of epochs, and goes down to 1e-6 at the minimum. We can indeed observe that the performances on the grader tends to smoothen after a certain number of epochs.

We start by training the DeepLab model with the default parameters for the optimizer and the learning rate, in other words with Stochastic gradient descent (SGD) as optimizer with a learning rate of 0.01. The default batch size is set to 4 and the default number of epochs to 16. This corresponds to a total of 1600 gradient descent steps.
To assess the performance of the SGD optimizer, we tried different learning rate, in order to find the best tuning possible as depicted in the following graph:

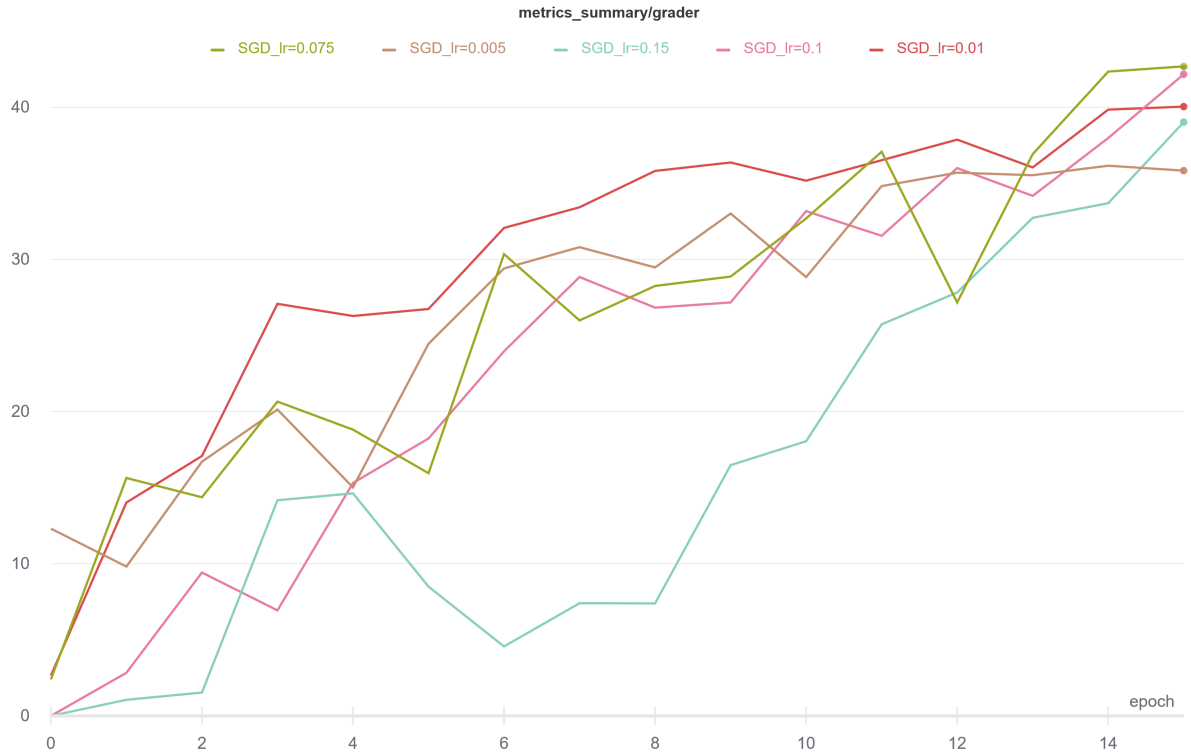

Figure 1: SGD optimizer Grader score with different learning rates

Name	Grader Score	WandB and S3 submission run name
SGD_lr = 0.075	42.688	G5_0408-0855_sgd_0.075_32012/
SGD_lr = 0.1	42.169	G5_0405-1228_SGD_0.1_90eff/
SGD_lr = 0.01	40.044	G5_0404-1832_Default_8df6e/
SGD_lr = 0.15	39.025	G5_0405-1252_SGD_0.15_4a84e/
SGD_lr = 0.005	35.836	G5_0407-0825_sgd_0.005_b8234/

As we can see we obtain the best performance with a learning rate of 0.075. Compared to the baseline learning rate of 0.01 we see than increasing the learning rate up to 0.1 improves performance. However, if we increase it too much, e.g. to 0.15 the grader score decreases. The smallest

learning rate (0.005) gives the worst performance for the SGD optimizer. With this heuristic approach, we found that the best suited learning rate for the SGD optimizer is lr = 0.075 and scores 42.688 on the grader.

- Now, we conduct a similar approach with the adam (Adaptive Moment Estimation) optimizer.

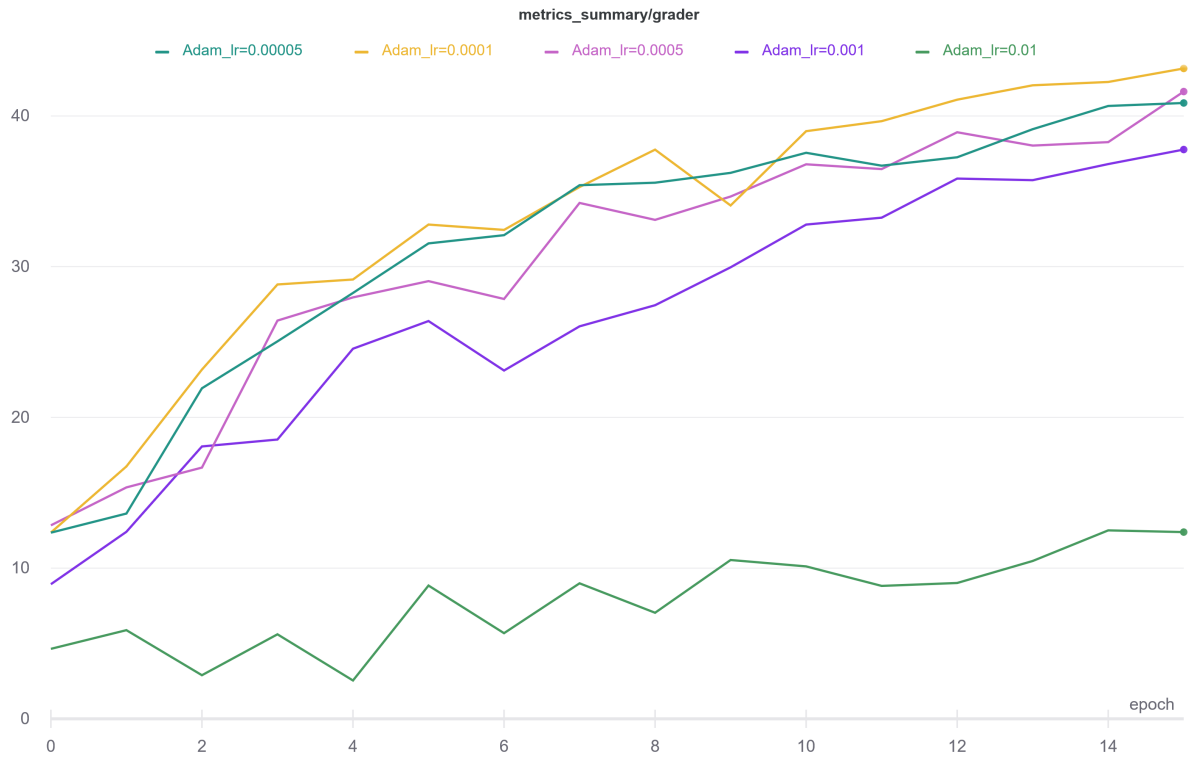

Figure 2: Adam optimizer Grader score with different learning rates

Name	Grader Score	WandB and S3 submission run name
Adam_lr = 0.0001	43.153	G5_0406-1928_adam_0.0001_5bdd2/
Adam_lr = 0.0005	41.62	G5_0405-2026_adam_lr0_is_0.0005_p2_acd07/
Adam_lr = 0.00005	40.863	G5_0407-0822_adam_0.00005_61077/
Adam_lr = 0.001	37.773	G5_0405-1205_adam_lr0_is_0.001_73808/
Adam_lr = 0.01	12.39	G5_0404-2003_Default_7ca88/

The default learning rate of 0.01 is clearly not suited for the adam optimizer. Hence the strategy was to decrease step by step the learning rate to find the best value. The grader score increases the more we decrease the learning rate until the threshold of lr = 0.0005 where it is again lower than for a learning rate of 0.0001. One conclude from this experimentation that the best learning rate found for the Adam optimizer is lr = 0.0001 = 1e-4. Adam combined with a learning rate of 0.0001 achieves a performance of 43.153 for the grader, which is higher than the best score performed by the SGD optimizer.
The following table shows the difference between the two optimizer best runs:

Optimizer	Learning rate	Grader score	Semantics	Depth
Adam	0.0001	43.153	69.77	26.617
SGD	0.075	42.688	69.743	27.055

As we see, Adam performs better in semantics and in Depth estimation. Hence, we decided from this point to pursue the next developments with the **Adam optimizer** and a learning rate of **0.0001** as the new default.

1.1.2 The batch size

By default, the batch size is set to 4 and the numbers of epochs to 16. So there is a factor 4 between these values. In order to keep the same number of gradient descent step when changing the batch size, the number of epochs should always be equal to 4 times the batch size value. An epoch refers to one cycle through the full training dataset. So when the batch size is increased, the number of steps per epoch decreases. Indeed for one epoch we have:

$$\#steps\_per\_epoch = \frac{training\_set\_length}{batch\_size}$$

because the length of the training set is fixed. Hence, the total number of steps is equal to the number of steps per epoch $\times$ the number of epochs.

Therefore, to test the batch size influence, 4 different runs are considered listed in the table below:

Name	WandB and S3 submission run name
Batch=2__Epochs=8	G5_0505-1204_adam_0.0001_2_8_2973e/
Batch=4__Epochs=16	G5_0406-1928_adam_0.0001_5bdd2/
Batch=8__Epochs=32	G5_0409-0814_adam_0.0001_8_32_ca036/
Batch=16__Epochs=64	G5_0410-0930_adam_is_0.0001_16_64_c5452/
Batch=16__Epochs=64 (resumed)	G5_0505-1005_adam_0.0001_16_64_resumed_75e16/

Note: AWS instances were systematically crashing for long running times associated to high batches and epochs number. This is why the run with 64 number of epochs and batch size 4 had to be resumed and is now separated in two parts in the graph.
However, when an instance is resumed, it generally starts in a new AWS S3 bucket, so that the number of steps is reinitialized to zero. For this reason, the grader graphs below are included once with the x-axis in number of steps and once with the number of epochs so that the end of the Batch=16__Epochs=64 run matches with Batch=16__Epochs=64 (resumed).

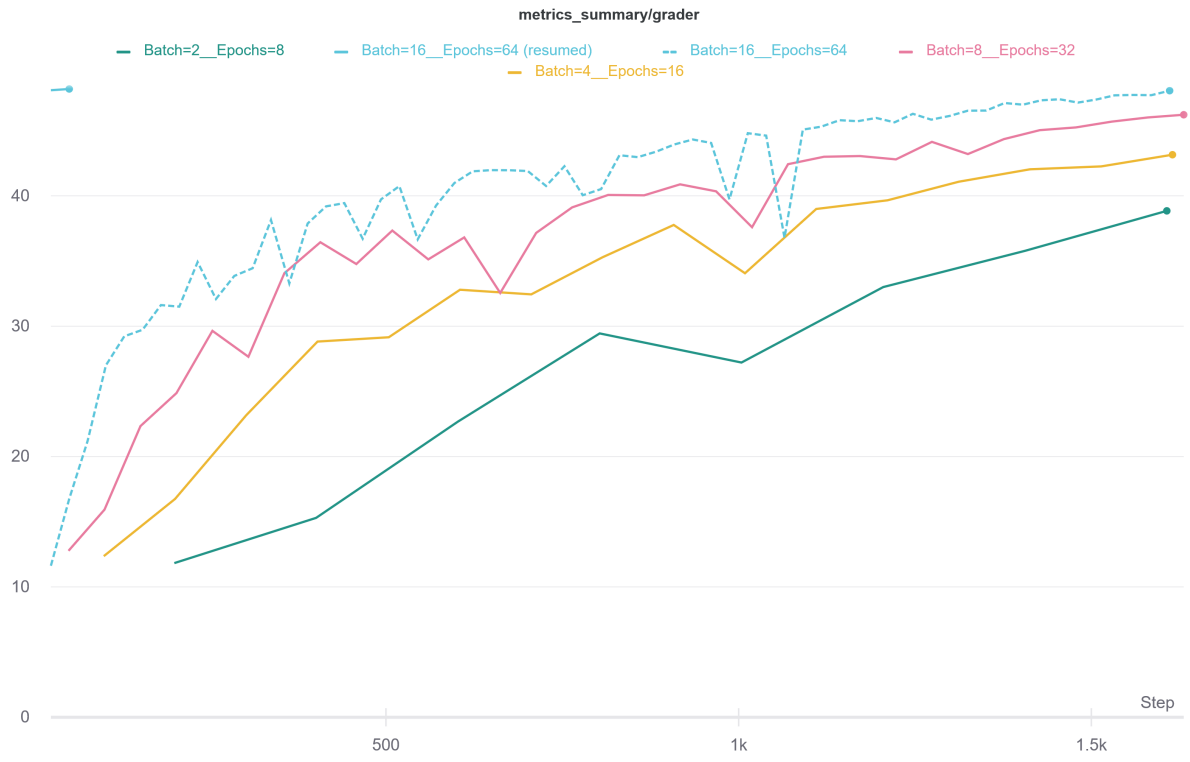

Figure 3: Different batch size and epochs setting: x-axis in step number

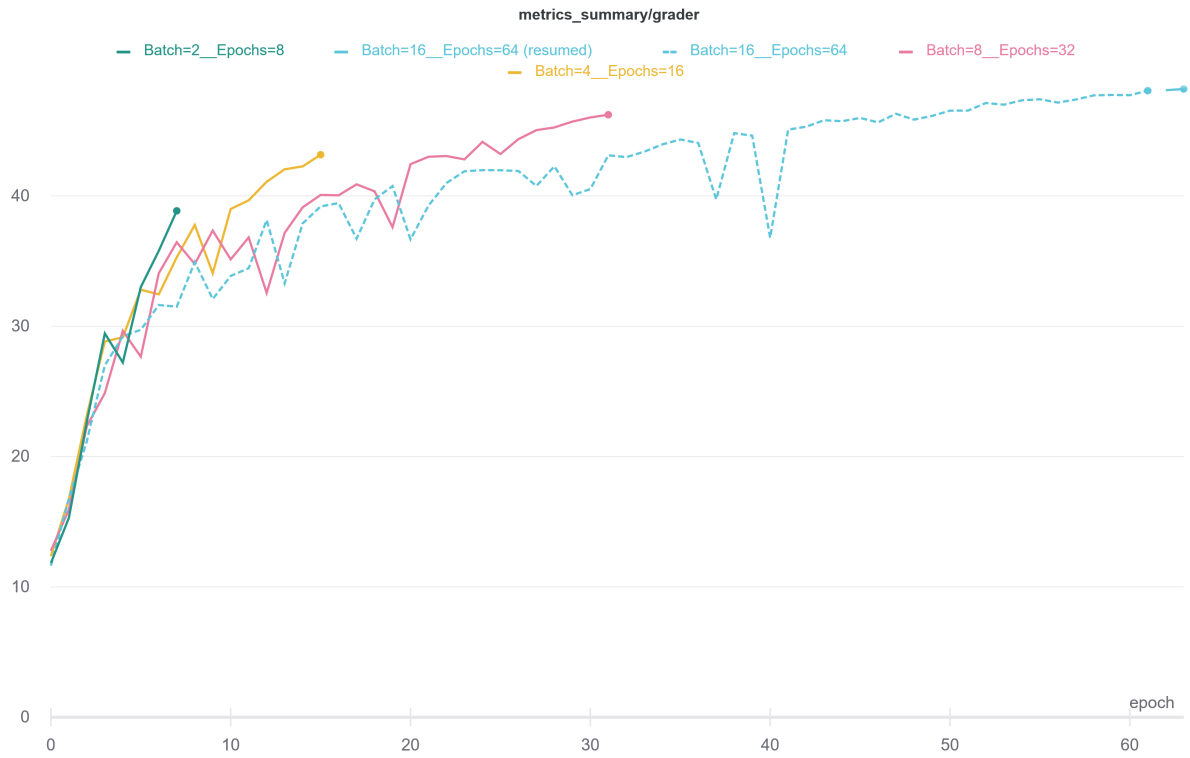

Figure 4: Different batch size and epochs setting: x-axis in epochs

It clearly appears that if we look to the grader score at the end of the training (i.e. @1600 steps), the higher the batch size, the better the performance. To confirm this trend, the metrics are summarized in the following table:

Name	Grader Score	Semantics	Depth Estimation
Batch=2_Epochs=8	38.847	66.852	28.04
Batch=4_Epochs=16	43.153	69.77	26.617
Batch=8_Epochs=32	46.22	71.698	25.478
Batch=16_Epochs=64 (resumed)	48.197	73.02	24.823

We can conclude that a higher batch size (and thus a higher number of epochs) leads to better performance for semantics as well as for depth estimation (lower score is better). Indeed, going from a batch size 2 with 8 epochs to a batch size 16 with 64 epochs is a gain of $G = 48.197 - 38.847 = 9.37$ for the grader. So the best setting appears to be with a batch size of 16 and 64 epochs. Nonetheless, with this solution we noticed that the training time also increased dramatically making our instances crash all the time (AWS: "BidEvictedEvent").

Proper training was therefore not possible with such a batch size, so we adopted a **batch size of 4 and 16 epochs**, although not optimal, to make sure that more runs would finish. (Resuming instances for more than tree times would have been very time-consuming when testing the future models).

1.1.3 Task weighting

Name	WandB and S3 submission run name
semseg=0.65_depth=0.35	G5_0501-2035_4_16_0.35_depth_3a5e1/
semseg=0.35_depth=0.65	G5_0501-2027_4_16_0.65_depth_e02c5/

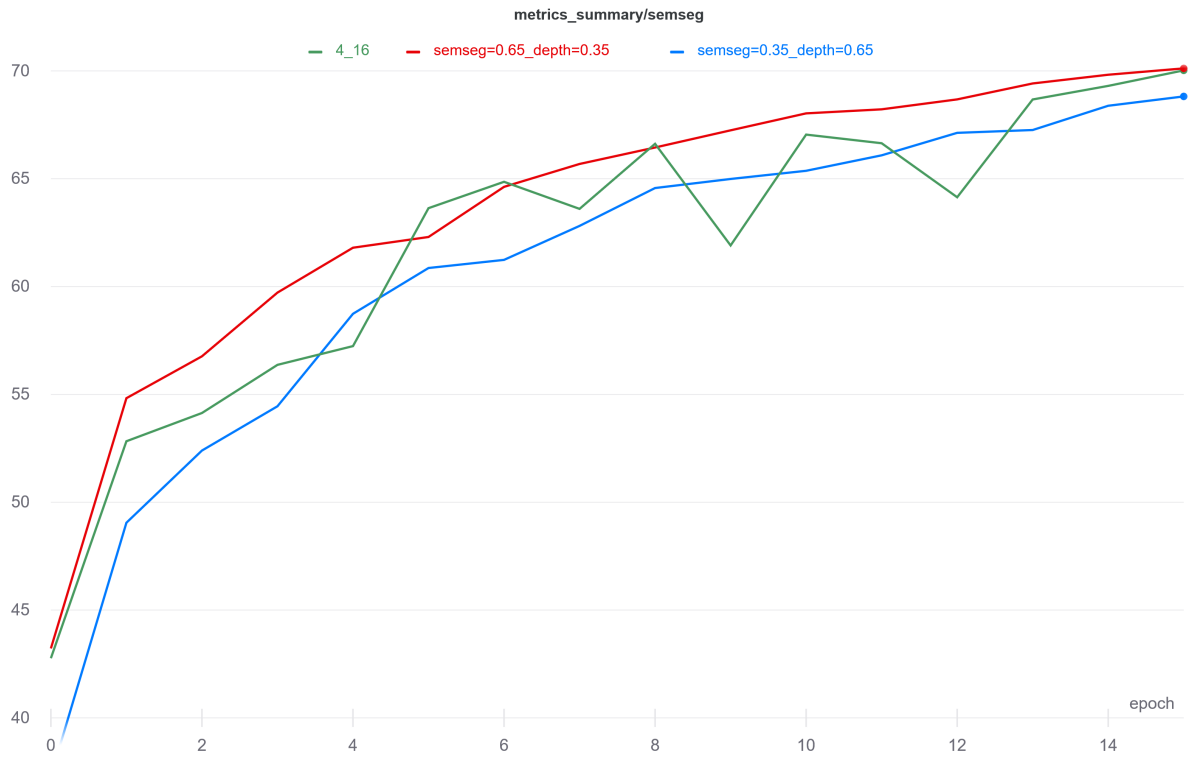

Figure 5: Metrics summary: semantic segmentation

Figure 6: Metrics summary: depth estimation

As expected, when the task weighting is set higher for the semantic segmentation, the model performs better in semantics but less good for depth estimation. Similarly, if the task weighting is larger for depth estimation the model performs better for the depth estimation than for semantic segmentation. The actual model is a joint architecture and has therefore only one shared decoder for both tasks (semantics **and** depth). Losses are however computed separately:
- Semantic segmentation is calculated with cross-entropy loss over the predictions for $n_{classes} = 19$

whereas depth is computed with L1/L2 loss (only one channel). Since the decoder is shared, the weights are updated in function of these two losses, although their gradient might have different direction, hence the trade-off between improving both semantic segmentation and depth simultaneously.

Increasing the semantic segmentation's loss weight, does not result in a big improvement in semantic segmentation, but it does result in an important decrease of the score in depth estimation. On the other hand increasing the depth's loss weight leads to better scores in depth estimation, but we lose too much in semantic segmentation.
So the best trade-off is to keep the loss weights equals.
The table below synthesises the two different task weights results with a comparison to the default setting (semseg = 0.5 and depth = 0.5), which is represented by the run Batch=4_-Epochs=16 (*G5_0406-1928_adam_0.0001_5bdd2/*).

Semseg weighting	Depth weighting	Semantics Score	Depth Score	Grader score
0.35	0.65	68.817	26.275	42.543
0.65	0.35	70.118	27.1	43.018
0.5	0.5	69.77	26.617	43.153

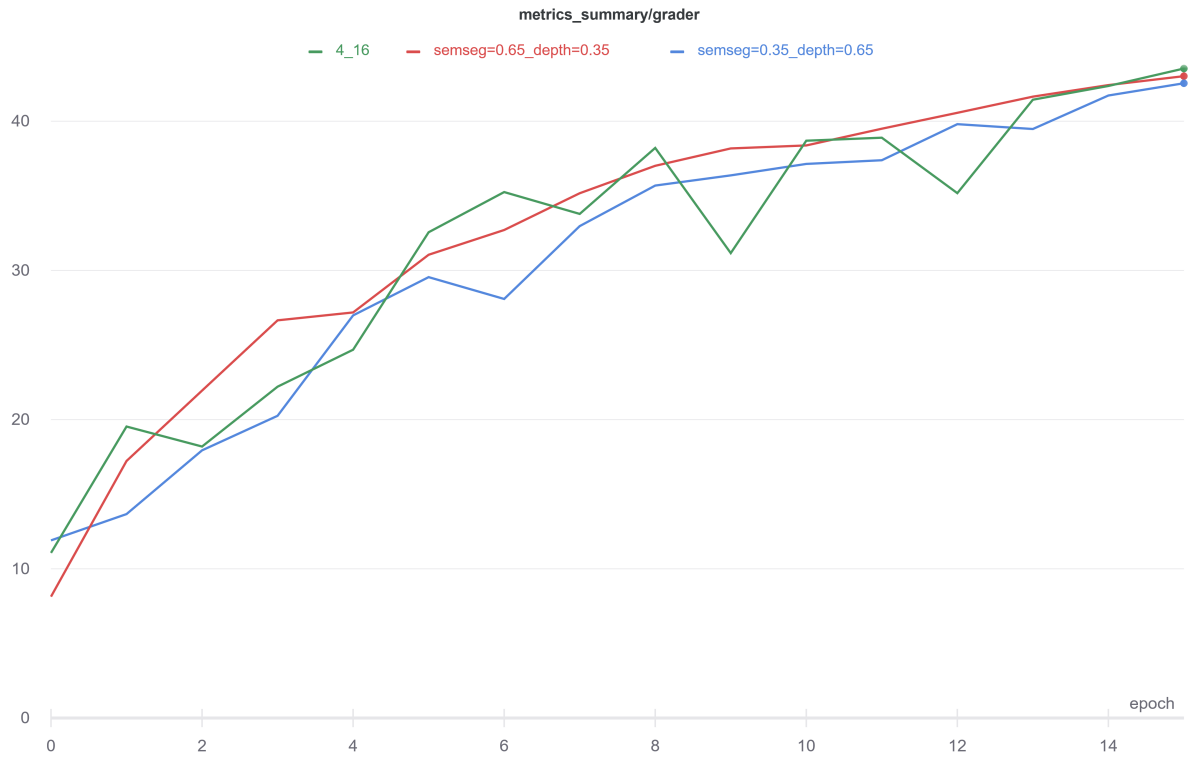

Figure 7: Metrics summary: Grader

Clearly, the best score for semantic segmentation is achieved by setting the task weighting to 0.65 for semseg, whereas the best score for depth estimation is achieved by setting the depth task weighting to 0.65. For the final grader however, the equally weighted task model scores the best.

Therefore, we keep the initial setting (semseg = 0.5 and depth = 0.5) for the rest of the exercise.

Note: This argument is based on our analysis with batch size 4 and epochs 16. We observed that increasing the semantic task weight performs better than increasing the depth task weight. We also did runs with batch size 8 and 32 epochs with different task weights which gave the following results

- semseg = 0.3, depth = 0.7, Grader score = 45.287

- semseg = 0.7, depth = 0.3, Grader score = 46.178

- semseg = 0.5, depth = 0.5, Grader score = 46.22

Similarly, the equally weighted tasks run performs slightly better, which confirms our choice to stay with 0.5 loss coefficient for both tasks. As mentioned earlier, we did not persist with a batch size of 8, because the instance run time was increased leading to higher risks of being closed by AWS.

1.2 Hardcoded hyperparameters

1.2.1 Initialization with ImageNet weights

So we continue with batch size 4 and a number of epochs equal to 16, with equal loss-weights for the semantic and depth tasks. The initialization with ImageNet weights improves the overall performance. Indeed, the model directly starts higher at the first epoch (grader score = 33.829 vs 12.36 for the non-pretrained model) and performs better (final grader score is 47.669)

1.2.2 Dilated convolutions

Dilation flags are set to (false, false, true). This setting really improves the overall performance in both depth and semantics, as the grader score increases from 47.669 without the true dilation flag to 58.48 with the true dilation flag activated. Hence, the gain for the grader score between the pretrained model and the pretrained model with dilation is $58.48 - 47.669 = 10.811$.

This improvement can be explained by the fact that convolution with stride in the 4th layer of the Resnet34 model are replaced by dilated convolutions. Dilated convolutions allow to have a larger receptive field with reasonable complexity, in other words, they allow to catch more information from a sparse area with the same number of parameters. The implemented model without dilation uses stride for convolution, which results in loses in-between features.
So the observed improvement in performance is explained by the increase of field of view (without decreasing the spatial dimensions) induced by the dilation convolutions. Dilation convolutions lead to feature map with higher resolution than with standard convolution and result in better predictions, as we observe here.

The different performance of the modified hardcoded hyperparameters vs the baseline (batch 4, #epochs = 16) are summarized in the following table.

Name	Semantics	Depth	Grader	WandB/AWS S3 submission
4_16_pretrained_with_dilation	79.895	21.416	58.48	G5_0502-0844_4_16_pretrained_dilation_b4308/
4_16_pretrained	72.891	25.222	47.669	G5_0502-0833_4_16_pretrained_6529b/
4_16	70.025	26.496	43.53	G5_0508-0651_4_16_dae2c/

Note: 4_16_pretrained_with_dilation and 4_16_pretrained were in another WandB project folder than the original model with batch size 4 used in the previous questions, so that it would have been impossible to show the original model and the new models with hardcoded hyperparameters on the same graph. This is why we used another instance (4_16) in the same folder as 4_16_pretrained_with_dilation and 4_16_pretrained, hence the small discrepancy in the final result of both runs with batch size 4 (Grader score is 43.53 vs 43.153 previously).

The following graph resumes well the overall improvement gained by initializing the model with ImageNet weights and activating the dilated convolutions:

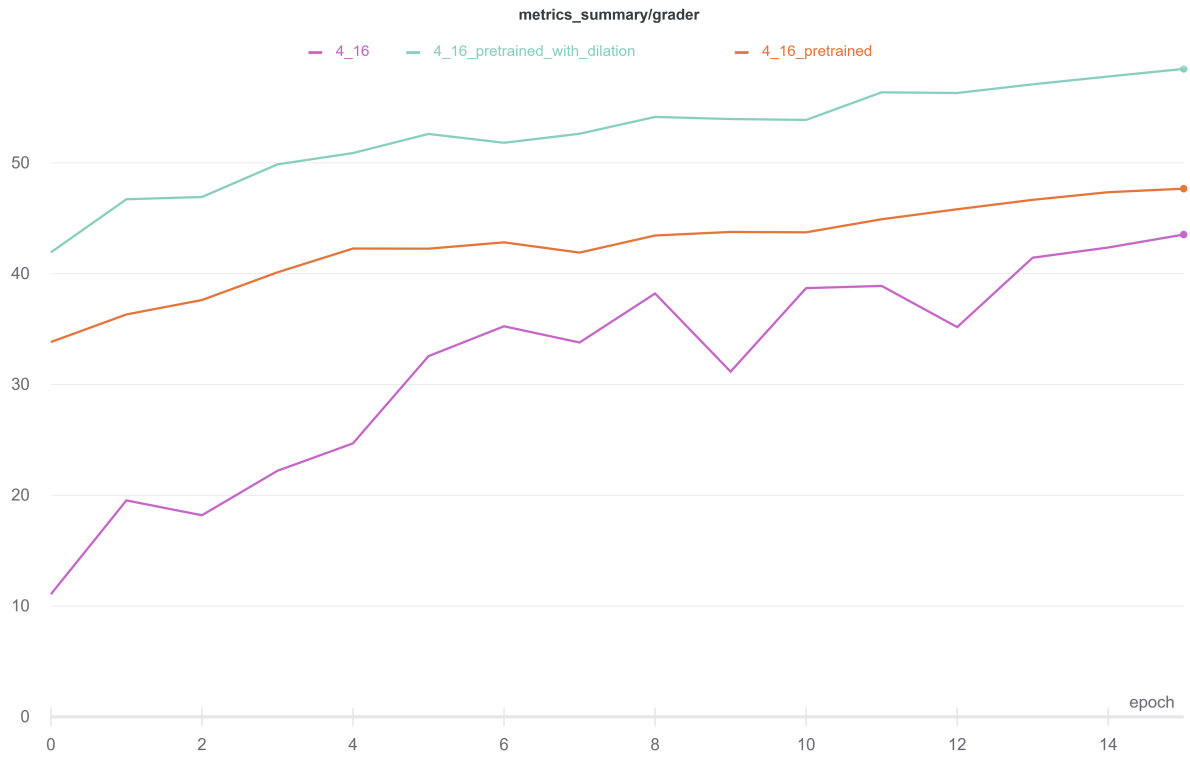

Figure 8: Effect of the initialization with ImageNet weights and dilated convolutions

Thus, we keep the initialization of the weights with ImageNet as well as the last dilation flag set to true, since it is clear from the graph and table, that *4_16_pretrained_with_dilation* performs 10.81 grader points above *4_16_pretrained* and 14.95 grader points above *4_16*.

1.3 ASPP and skip connections

We now want to implement the following ASPP module on top of our existing model.
The ASPP module consists in the concatenation of 4 different convolutional layer and one pooling layer applied on the output of the encoder.

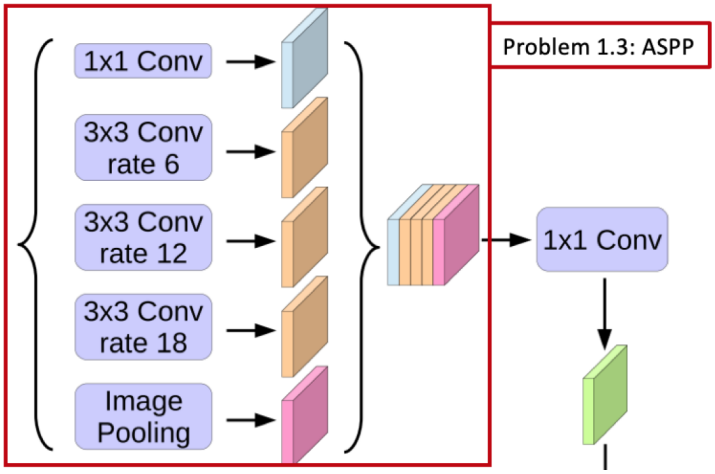

Figure 9: ASPP architecture

For the pooling layer, we chose to do some Global Pooling (according to: [1]), which consists in an average over all the pixels of one channel (one feature map), applied separately on each channel. The output dimension of this operation is (1,1,num_channels).
Thus, we upsample it back to the original spatial dimension i.e. to the same dimensions as the original feature maps to be able to apply a concatenation with the other ASPP layers.

For the four convolutions, the different rates of the Atrous convolutions allows to have several model's field-of-view, and thus enables features encoding at multiple scales. The rates have been chosen to be 6, 12 and 18 according to the figure in the exercice intructions and according to [1]. The 5 parallel layers are then concatenated along the channel dimension axis.
After the concatenation, we apply a 1x1 convolutional layer, to reduce the number of channels $5 * 256$(or 5*out_channels) down to 256 which is the number of out_channels of the ASPP module. This combination of spatial pyramid pooling with atrous convolutions exploits the multi-scale information and should perform better than the original model.

The code below depicts the implementation of the four atrous convolution, the global pooling, the concatenation of these five parallel layers and the 1x1 convolution to reduce the number of channels to 256.

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

class ASPP(torch.nn.Module):
    def __init__(self, in_channels, out_channels, rates=(3, 6, 9)):
        super().__init__()
        modules = []
        rates = [2*x for x in rates]
        modules.append(ASPPpart(in_channels, out_channels, kernel_size=1,
                                stride=1, padding=0, dilation=1))
        for rate in rates:
            modules.append(ASPPpart(in_channels, out_channels, kernel_size=3,
                                    stride=1, padding=rate, dilation=rate))

        global_avg = torch.nn.Sequential(torch.nn.AdaptiveAvgPool2d(1),
                                         torch.nn.Conv2d(in_channels,
                                                         out_channels,
                                                         kernel_size = 1,
                                                         bias = False),
                                         torch.nn.BatchNorm2d(out_channels),
                                         torch.nn.ReLU())
        modules.append(global_avg)
        self.aspp_convs = torch.nn.ModuleList(modules)
        self.conv_1x1 = torch.nn.Sequential(torch.nn.Conv2d(
                                            out_channels*len(self.aspp_convs),
                                            out_channels, kernel_size = 1, bias = False),
                                            torch.nn.BatchNorm2d(out_channels),
                                            torch.nn.ReLU())
    def forward(self, x):
        aspp_net = []
        resolution_h_w = (x.shape[2], x.shape[3]) # height and width of feature map
        for layer in self.aspp_convs:
            aspp_net.append(layer(x))
        #aspp_net[4] is the output of the average pooling but has h= 1, w = 1 hence the
    upsample
        aspp_net[4] = F.interpolate(aspp_net[4],resolution_h_w, mode = 'bilinear',
                                    align_corners=False)
        aspp_net = torch.cat(aspp_net, dim = 1)
        return self.conv_1x1(aspp_net)
```

Listing 1: Implementation of the ASPP class

Then, we had to implement the Decoder with skip connections such that it takes the ASPP's output and combines it with the features from the encoder.

Some of the design choices for the decoder:
According to [2], we use a 1x1 convolution to decrease the number of channels in the skip connection (from the encoder) from 64 to 48. Indeed, the authors in [2] have shown that reducing the channels of the low-level feature map from the encoder module to 48 channels have led to the best results.

These skip connection feature maps are then concatenated with the bilinearly upsampled by 4 output of the ASPP module (so that they have the same spatial dimension).
Finally, we apply the last convolutional layers on the concatenated feature maps.
Below is the implementation in Pytorch:

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

        self.features_to_concatenation = torch.nn.Sequential(torch.nn.Conv2d(
                                                             skip_4x_ch, 48,
                                                             kernel_size=1,
                                                             stride=1),
                                                             torch.nn.BatchNorm2d(48),
                                                             torch.nn.ReLU())

        self.conv3x3 = torch.nn.Sequential(torch.nn.Conv2d(bottleneck_ch+48, 256,
                                                           kernel_size=3, padding=1,
                                                           bias = False),
                                           torch.nn.BatchNorm2d(256),
                                           torch.nn.ReLU())
        self.concatenation_to_predictions = torch.nn.Conv2d(256, num_out_ch, kernel_size=1,
                                                            stride=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 and the final tensor of predictions
        """
        #upsample ASPP features by 4
        features_ASPP = F.interpolate(
                        features_bottleneck, size=features_skip_4x.shape[2:],
                        mode='bilinear', align_corners=False)
        #1x1 conv2d on lowest feature (skip)
        features_skip = self.features_to_concatenation(features_skip_4x)
        #concatenation of lowest feature and upsampled output of ASPP
        features_cat = self.conv3x3(torch.cat([features_skip,features_ASPP], dim=1))
        #1x1 conv to get predictions
        predictions = self.concatenation_to_predictions(features_cat)
        return predictions, features_cat
```

Listing 2: Implementation of the DecoderDeeplabV3p class

Inspired by [2], we tested several architectures for the last convolutional layers on the concatenated features.

- **One convolution with kernel size 3 followed by one convolution of kernel size 1**. First 3x3 convolution with channel reduction from 256+48 = 304 to 256 and second 1x1 convolution with channel reduction from 256 to output channels (19 for semantic segmentation and 1 for depth).
- **Two convolutions with kernel size 3**. The first one with channel reduction from 304 to 256 and the second one with channel reduction from 256 to output channels.
- **Two convolutions with kernel size 3 followed by one convolution of kernel size 1**. First 3x3 convolution with channel reduction from 304 to 256, the second 3x3 convolution without channel reduction. Last 1x1 with channel reduction from 256 to output channels.

Another thing we can observe, is that the model's performance still increases after 16 epochs. From now on, as the model gets more and more parameters, it is interesting to make it run longer, so we will run the model on num_epochs 25. To have a fair comparison of performances, one run with batch size 4 and 25 epochs, pretrained and with the last dilation flag enabled has been made. At 16 epochs, *4_16_pretrained_with_dilation* performs better than *4_16_pretrained_with_dilation*, however *4_25_pretrained_dilation* performs better at the end of its training time i.e. at the end of the 25 epochs.

In the table below, we clearly see the improvement brought by the ASPP module and the skip connection decoder from the first epoch the the 25th epoch. This meets our expectations in terms of improvement to baseline.

Name	Semantics	Depth	Grader	WandB/AWS S3 submission
aspp_4_25_2_3x3_1x1	85.827	18.433	67.394	G5_0428-1049_aspp_4_25_2_3x3_1x1_08a84/
aspp_4_25_3x3_1x1	85.168	18.685	66.483	G5_0508-1348_aspp_4_25_1_3x3_1x1_146a6/
aspp_4_25_2_3x3	85.243	19.004	66.239	G5_0428-2010_aspp_4_25_2_3x3_1de09/
4_25_pretrained_dilation	80.462	21.029	59.433	G5_0514-0747_4_25_pretrained_dilation_e2297
4_16_pretrained_with_dilation	79.895	21.416	58.48	G5_0502-0844_4_16_pretrained_dilation_b4308/

Table 1: ASPP performance comparison

We can see that the model with two convolutions with kernel size 3 followed by one convolution with kernel size 1 performs best overall. It takes 1 point on the two other architectures on the grader.

For the semantic segmentation, we have a gain of $G = 85.827 - 80.462 = 5.365$ (compared to the pretrained, with dilation model).
In depth estimation, the score improves by $G = 21.029 - 18.433 = 2.596$.
This leads to a gain $G = 67.394 - 59.433 = 7.961$ on the grader.

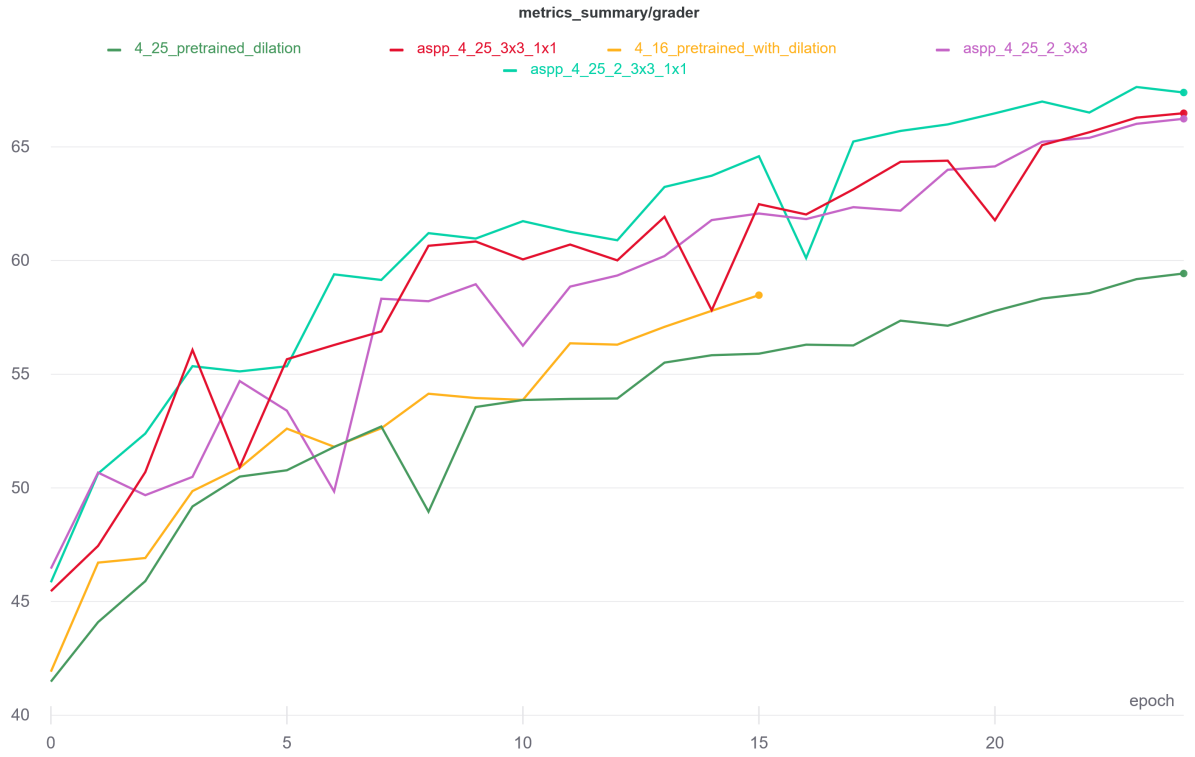

Figure 10: ASSP with different decoder implementation compared to initial model, and pretrained+dilation model

Concerning the tunning of the hyperparameters of the model with ASPP, it is rather limited: The initialization of the encoder with ImageNet weights and the dilated convolution improve the model's performances, so we will keep them.
As the architecture of the model has not changed a lot, the depth and semantic segmentation are still sharing the same decoder, the task weighting shouldn't change a lot. The tunning that

provided the best results was with loss_semseg=0.5 and loss_depth=0.5, so we'll keep those values. We are left with the batch size(and num_epochs), and the learning rate to tune.
For the batch size, we have been limited in our testings by AWS, as the instances all crashed when increasing the batch size and the number of epochs.
We deduced from our previous experiments on batch size tunning on the basic model, that we could win around 4-5 points on the grader with a batch size 16 and num_epochs 64. We decided to adopt a batch size of 4 and num_epochs of 16 to avoid the crashes. As explained above, the number of epochs has been extended to 25, since the model has more parameters to train and it can be beneficial to train it longer.
As we are obtaining good performances with the current learning rate at 0.0001, we won't tune it any further.

The expected performances for this task are 62.3 ± 4.0 for the grader, 83.6 ± 2.0 in semantic segmentation, and 20.5 ± 2.0 in depth estimation.
Our two best ASPP models have the following results, see Tab 1:

- semantic segmentation = 85.827, depth estimation = 18.433, Grader score = 67.394
  Architecture: two convolutions with kernel size 3 followed by one convolution of kernel size 1

- semantic segmentation = 85.168, depth estimation = 18.685, Grader score = 66.483
  Architecture: one convolution with kernel size 3 followed by one convolution of kernel size 1

Our model seems more than satisfying, so we will stop the tunning here. Although it is clear that two convolutions with kernel size 3 followed by one convolution of kernel size 1 performs the best, it takes also more time to train (more parameters), this is why we also consider the second best architecture with only one convolution 3x3 followed by one convolution 1x1 which trains faster and only loses approximately one point for the grader.

2 Branched architecture

Until now, our model was sharing one decoder for the semantic segmentation task and the depth estimation task. We now want to implement one decoder for each task, with the following architecture:

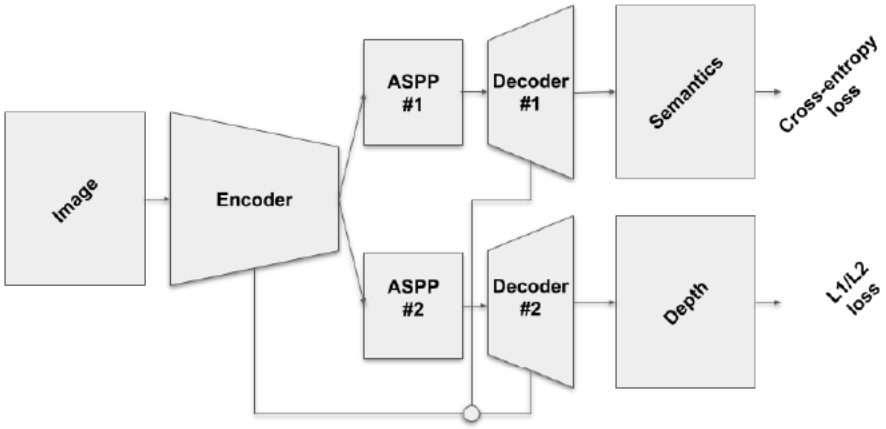

Figure 11: branched model architecture

For the implementation of the branched model, we build on the two best architectures of the previous ASPP model.
We created a new class, ModelBranched, which defines two similar DecoderDeeplabV3p, one for the semantic segmentation task, the other for the depth estimation task. It allows us to have more parameters dedicated to estimate the depth, and more for the semantic segmentation.

```
#ASPP and decoder for semantic segmentation task
self.aspp_semseg = ASPP(ch_out_encoder_bottleneck, 256)

self.decoder_semseg = DecoderDeeplabV3p(256, ch_out_encoder_4x, ch_out-1)

#ASPP and decoder for depth estimation task
self.aspp_depth = ASPP(ch_out_encoder_bottleneck, 256)

self.decoder_depth = DecoderDeeplabV3p(256, ch_out_encoder_4x, 1)
```

Listing 3: Initialization of the two ASPP and decoder modules in class ModelBranched

```
    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]

        #Semantic Segmentation
        features_tasks_semseg = self.aspp_semseg(features_lowest)
        predictions_4x_semseg,_ = self.decoder_semseg(features_tasks_semseg, features[4])

        predictions_1x_semseg = F.interpolate(predictions_4x_semseg, size=input_resolution,
     mode='bilinear', align_corners=False)

        #Depth estimation
        features_tasks_depth = self.aspp_depth(features_lowest)
        predictions_4x_depth,_ = self.decoder_depth(features_tasks_depth, features[4])

        predictions_1x_depth = F.interpolate(predictions_4x_depth, size=input_resolution,
    mode='bilinear', align_corners=False)
```

Listing 4: Features flow in the forward method of the class ModelBranched

See 11 for the entire ModelBranched class

Name	Semantics	Depth	Grader	WandB/AWS S3 submission
resume2_branched_2_3x3_1x1	86.14	16.746	69.394	G5_0505-2025_resume_branched_4_25_9d545/
resume_branched_2_3x3_1x1	86.14	16.746	69.394	G5_0505-2025_resume_branched_4_25_9d545/
branched_2_3x3_1x1	86.14	16.746	69.394	G5_0429-1233_branched_4_25_56e93/
branched_3x3_1x1	85.238	17.363	67.875	G5_0509-0959_branched_3x3_1x1_aa9fc/
aspp_4_25_2_3x3_1x1	85.827	18.433	67.394	G5_0428-1049_aspp_4_25_2_3x3_1x1_08a84/
aspp_4_25_3x3_1x1	85.168	18.685	66.483	G5_0508-1348_aspp_4_25_1_3x3_1x1_146a6/

Figure 12: branched model with different decoder implementation compared to ASPP model with similar decoder architecture

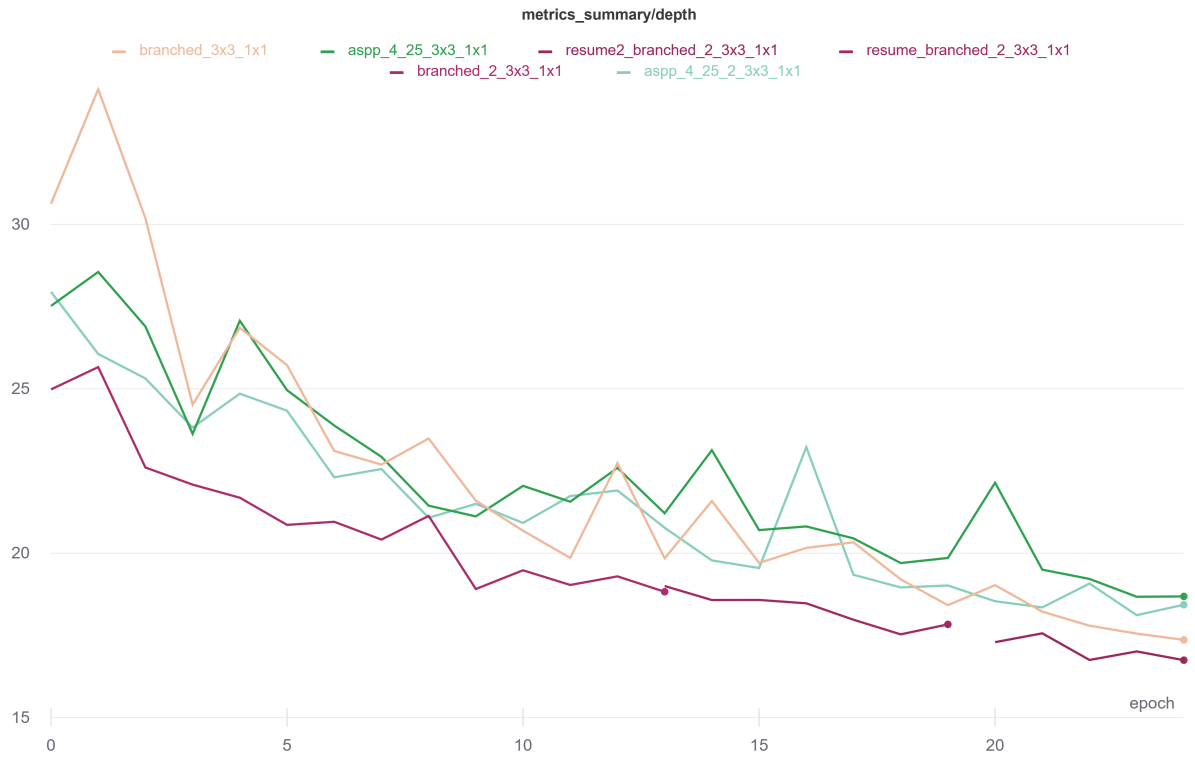

Figure 13: Depth estimation score of branched model with different decoder implementation compared to ASPP model with similar decoder architecture

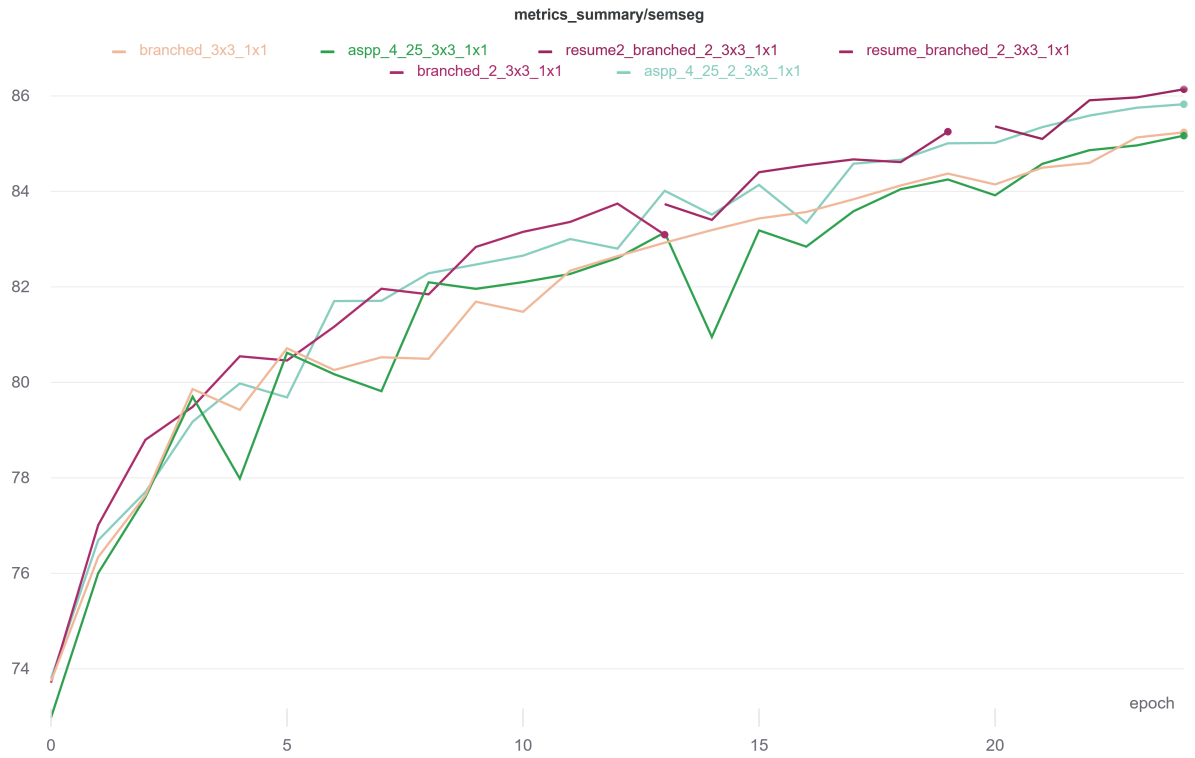

Figure 14: Semantic Segmentation score of branched model with different decoder implementation compared to ASPP model with similar decoder architecture.

The expected performances for this task are 65.3 ± 4.0 for the grader, 84.5 ± 2.0 in semantic segmentation, and 19.2 ± 2.0 in depth estimation.
The two best branched models have the following results:

- semantic segmentation = 86.14, depth estimation = 16.746, Grader score = 69.394
- semantic segmentation = 85.238, depth estimation = 17.363, Grader score = 67.875

The branched model uses two separate decoders, one for the semantic segmentation and one for the depth estimation. This allows the model to better estimate both tasks. For the semantic segmentation, we observe a slight improvement (around 0.2). But the main improvement is in **depth estimation**:

- For the 2 3x3 followed by 1x1 (convolutional layers), we have a gain of G = 18.433-16.746 = 1.687 in the depth estimation score.
- For the 3x3 followed by 1x1 (convolutional layers), we have a gain of G = 18.685-17.363 = 1.322 in the depth estimation score.

We have a clear improvement of the depth estimation with our branched model. This confirms the utility of the branched architecture with two separate decoders.

The required computation time for our ASPP model (*aspp_4_25_2_3x3_1x1*) is 10-11h, whereas our branched model requires around 15 hours.

Both of our branched models are satisfying, however, as the model gets more and more complex, it takes more time to train, and due to some issues with AWS, we are not able to run our models for a long time without crashing.
Thus, for the following task, we will select the second model, whose architecture has less convolutional layers.

3 Task distillation

In the third task, we are asked to implement a model that distills information over tasks. On top of our branched model. To do this, we will get back the output features from the first decoders(of our branched model), and feed them to two self attention modules (one for each task).
We will then combine the output of each of the self-attention modules, with the output features from the first decoder of the other task (that is where we distill the informations between the semantic segmentation and depth estimation task).

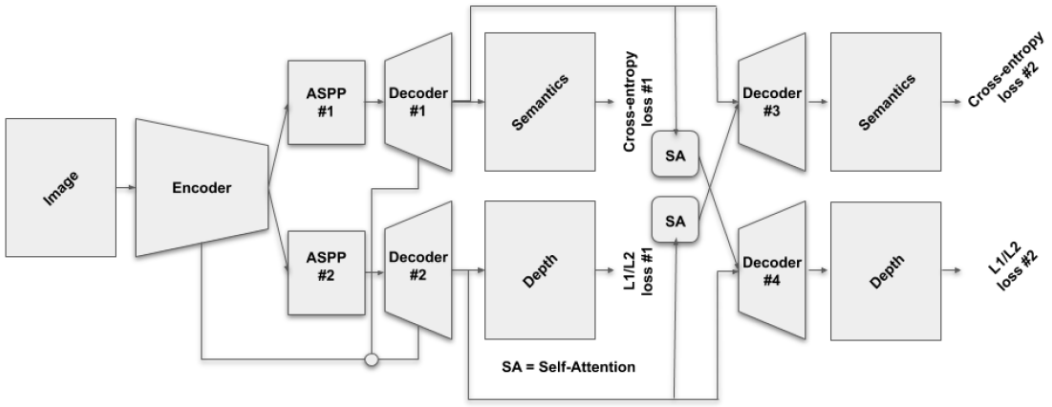

Figure 15: branched model followed by task specific distillation architecture

To implement this, we have created a new class ModelDistillation, based on the class Model-Branched of the previous architecture.
But now, we also extract the features from decoder 1 and 2 (see Fig.15), and pass them to the self-attention modules. We then feed the output of the self-attention modules, and the output of decoder 1 and 2, to decoder 3 and 4 (see Listing 7 for self-attention implementation and Listing 8 for decoder 3 and 4).

```
1
2  class ModelDistillation(torch.nn.Module):
3      def __init__(self, cfg, outputs_desc):
4          super().__init__()
5
6          num_features = 256
7          #Self-Attention
8          self.attention_semseg = SelfAttention(num_features, num_features)
9          self.attention_depth = SelfAttention(num_features, num_features)
10
11         #Decoder 3 and 4
12         self.decoder3 = Decoder(num_features,num_features,ch_out-1)
13         self.decoder4 = Decoder(num_features,num_features,1)
```

Listing 5: Changes in the init method of the class ModelDistillation

```
1      def forward(self, x):
2          #Semantic Segmentation
3          features_task_semseg = self.aspp_semseg(features_lowest)
4          predictions_4x_semseg,features_dec1 = self.decoder_semseg(features_task_semseg,
       features[4])
5          predictions_1x_semseg = F.interpolate(predictions_4x_semseg, size=input_resolution,
        mode='bilinear', align_corners=False)
6
7          #Depth estimation
8          features_task_depth = self.aspp_depth(features_lowest)
9          predictions_4x_depth,features_dec2 = self.decoder_depth(features_task_depth,
       features[4])
10         predictions_1x_depth = F.interpolate(predictions_4x_depth, size=input_resolution,
       mode='bilinear', align_corners=False)
11         #Self-Attention
12         features_attention_semseg = self.attention_semseg(features_dec1)
```

```
        features_attention_depth = self.attention_depth(features_dec2)

        #Decoder 3 and 4
        out_semseg_4x,_ = self.decoder3(features_attention_depth, features_dec1)
        out_semseg_1x = F.interpolate(out_semseg_4x, size=input_resolution, mode='bilinear'
    , align_corners=False)
        out_depth_4x,_ = self.decoder4(features_attention_semseg, features_dec2)
        out_depth_1x = F.interpolate(out_depth_4x, size=input_resolution, mode='bilinear',
    align_corners=False)

        out = {}
        offset = 0

        for task, num_ch in self.outputs_desc.items():
            if task == 'semseg':
                out[task] = [out_semseg_1x[:, offset:offset+num_ch, :, :],
    predictions_1x_semseg[:, offset:offset+num_ch, :, :]]
                offset += num_ch
            elif task == 'depth':
                out[task] = [out_depth_1x[:,:, :, :],predictions_1x_depth[:,:, :, :]]
                offset += num_ch

            else:
                print("mod erreur, should be either MOD_SEMSEG or MOD_DEPTH")
        return out
```

Listing 6: Changes in the forward method of the class ModelDistillation

The entire ModelDistillation class can be found in Listing 12

```
class SelfAttention(torch.nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv = torch.nn.Conv2d(in_channels, out_channels, 3, padding=1, bias=False)
        self.attention = torch.nn.Conv2d(in_channels, out_channels, kernel_size=3, padding
    =1, bias=False)
        with torch.no_grad():
            self.attention.weight.copy_(torch.zeros_like(self.attention.weight))

    def forward(self, x):
        features = self.conv(x)
        attention_mask = torch.sigmoid(self.attention(x))
        return features * attention_mask
```

Listing 7: Implementation of the self-attention module

```
class Decoder(torch.nn.Module):
    def __init__(self, attention_ch, out_DeepLab_ch, num_out_ch):
        super(Decoder, self).__init__()

        self.conv3x3 = torch.nn.Sequential(torch.nn.Conv2d(out_DeepLab_ch, 256,
                                           kernel_size=3, padding=1, bias = False),
                                           torch.nn.BatchNorm2d(256),
                                           torch.nn.ReLU())
        self.concatenation_to_predictions = torch.nn.Conv2d(256, num_out_ch, kernel_size=1,
     stride=1)

    def forward(self, features_attention, features_out_decoder):
        """
        DeepLabV3+ style decoder
        :param features_attention: features out of self-attention module
        :param features_out_decoder: features out of decoder 1 or 2
        :return: features with 256 channels and the final tensor of predictions
        """

        #sum of self-attention features and output of first decoder features
        summed_features = features_attention
        for i in range(features_out_decoder.shape[1]):
            summed_features[:,i,:,:] =  features_attention[:,i,:,:]+features_out_decoder[:,
    i,:,:]
        features_intermediate = self.conv3x3(summed_features)
        #1x1 conv2d for final predictions
        predictions = self.concatenation_to_predictions(features_intermediate)
        return predictions, features_intermediate
```

Listing 8: Implementation of the final decoder

As our distillation model did not improve the grader score as we expected (i.e to score at least one point better than the branched model), we decided to tune the model.
Batch size changes were rather limited, although we would have liked to increase it to 8 or 16. The pretrained and dilation flags could only enhance the model's performance. So we were left with the tasks weighting and the learning rate tuning.
We tried to change the loss-coefficients, setting 0.6 for the depth, 0.4 for the semantic segmentation and vice versa for the distillation model. However improvements were not observed compared to the equally weighted coefficients, so we did not let these model run to the end of training since the amount of instances was limited. This is coherent with the tuning observations concluded in task 1.

We also tried several learning rates for our distillation model. The learning rates higher than 0.0001 (e.g. 0.0002) were performing pretty bad, so we did not relaunch them once they crashed. The smaller learning rate however, brought a slight improvement:

Name	Semantics	Depth	Grader	WandB/AWS S3 submission
resume_distillation_lr=0.00008	85.218	17.341	67.877	G5_0510-2025_resume_full3x3and1x1_distillation_4_25_lr=0.00008_3ccde/
distillation_lr=0.00008	85.218	17.341	67.877	G5_0510-0822_full3x3and1x1_distillation_4_25_lr=0.00008_e74d1/
branched_3x3_1x1	85.238	17.363	67.875	G5_0509-0959_branched_3x3_1x1_aa9fc/
resume_distillation_lr=0.00006	85.165	17.573	67.592	G5_0512-0646_resume_full3x3and1x1_distillation_4_25_lr=0.00006_49ef3/
distillation_lr=0.00006	85.165	17.573	67.592	G5_0510-2035_full3x3and1x1_distillation_4_25_lr=0.00006_feead/
distillation_lr=0.0001	84.959	17.861	67.097	G5_0508-1339_full3x3and1x1_distillation_4_25_86d84

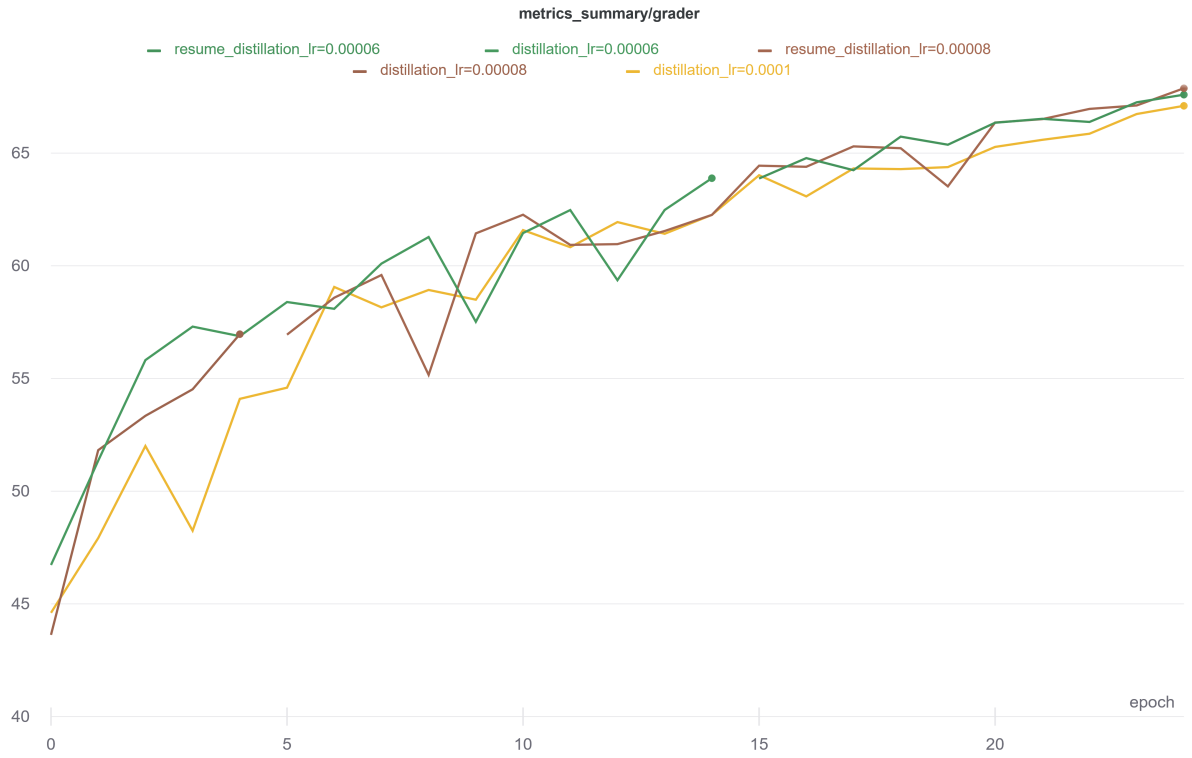

Figure 16: tunning of the learning rates for the distillation model

We observe the optimal learning rate at 0.00008, with a gain $G = 85.218 - 84.959 = 0.259$ compared to the model with our initial learning rate.
The expected performances for the third task are 67.3 ± 4.0 for the grader, $84{:}5 \pm 2.0$ in semantic segmentation, and 17.3 ± 2.0 in depth estimation.
Our best distillation model has the following results:

- semantic segmentation = 85.218, depth estimation = 17.341, Grader score = 67.877

So we see that we are within the range expected for the distillation model, however the improvements brought by the distillation architecture are a bit disappointing compared to the branched model of task 2, considering that the training time increased a lot. Indeed the required computation time is now around 2 days and 5 hours.
For this reason, we tried to improve our model, referring to [3] in paragraph 3.2 and 3.3, by weighting the self attention features with a **scale parameter** α and then perform the element-wise sum. As the authors suggested, we initialized α to zero and defined it as a parameter of our model, so that it gradually learns to assign more weight.

```
self.alpha = torch.nn.parameter.Parameter(torch.zeros(1))
```

Listing 9: Declaration of alpha as a model parameter initialized to zero

```
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    summed_features = torch.zeros(features_attention.shape).to(device)
    features_attention = self.alpha.to(device)*features_attention.to(device)
    features_out_decoder = features_out_decoder.to(device)
    for i in range(features_out_decoder.shape[1]):
        summed_features[:,i,:,:] =  features_attention[:,i,:,:]+
                                    features_out_decoder[:,i,:,:]
    ...
```

Listing 10: Implementation of the final decoder's forward method with the alpha parameter

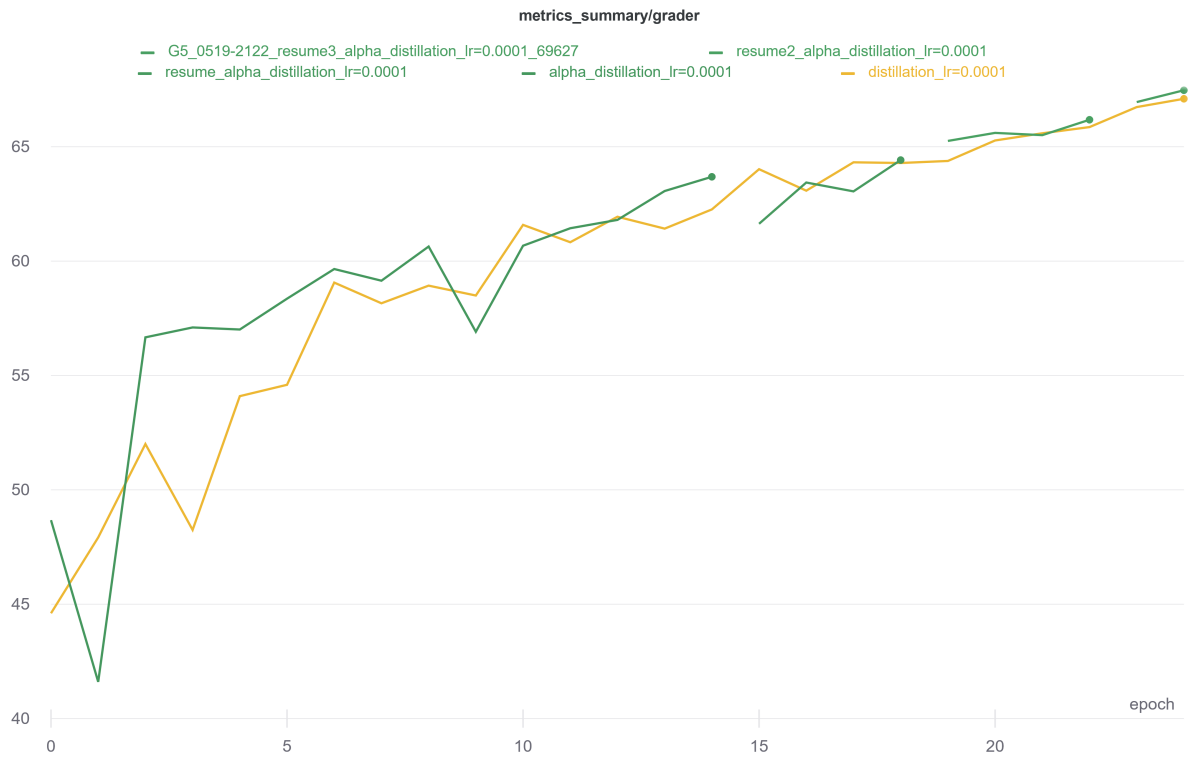

Figure 17: Grader: Distillation model with alpha parameter

Name	Semantics	Depth	Grader	WandB/AWS S3 submission
resume2_alpha_distillation_lr=0.0001	85.052	17.583	67.47	G5_0519-0658_resume2_alpha_distillation_lr=0.0001_948a8e/
resume_alpha_distillation_lr=0.0001	85.052	17.583	67.47	G5_0518-0744_resume_alpha_distillation_lr=0.0001_c6584
resume_alpha_distillation_lr=0.0001	85.052	17.583	67.47	G5_0516-1506_alpha_distillation_lr=0.0001_f6204
distillation_lr=0.0001	84.959	17.861	67.097	G5_0508-1339_full3x3and1x1_distillation_4_25_86d84

Comparing the model with the alpha parameter to the original distillation model and keeping the same learning rate (0.0001) we gain 0.373 points on the grader. We had to resume this model

three times and after the first crash it seems that we lost some points, but it is hard to determine if this had an influence on the final grader performance. Although this new implementation improves a bit the performance, we decided not to continue with this architecture since the improvement was not significant enough. Indeed, the best performance was reached by setting the learning rate to 0.0008 with the distillation model as depicted on the table above.
Finally, we conclude that the tuning of the distillation model did not bring much improvement. Possibly, a higher batch size would have performed better.

Moreover, we also realized that our ASPP model already scored slightly higher than the expected grader performances (around 66.4 vs 62.3 $\pm$ 4), which could explain why we did not witness drastic changes in the grader score between the model at end of task 1 and the distillation model in task 3.

To resume, we can see the evolution of our model, and the improvements through the different steps.

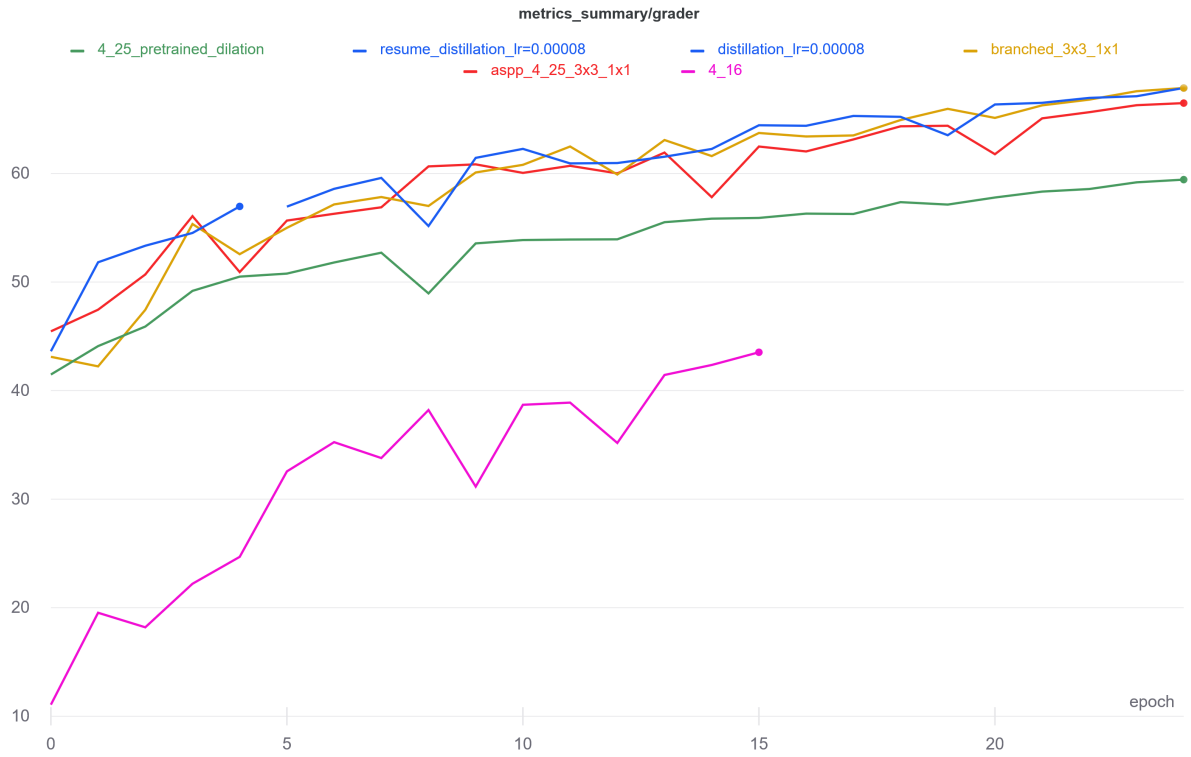

Figure 18: Grader: Model evolution through the tasks

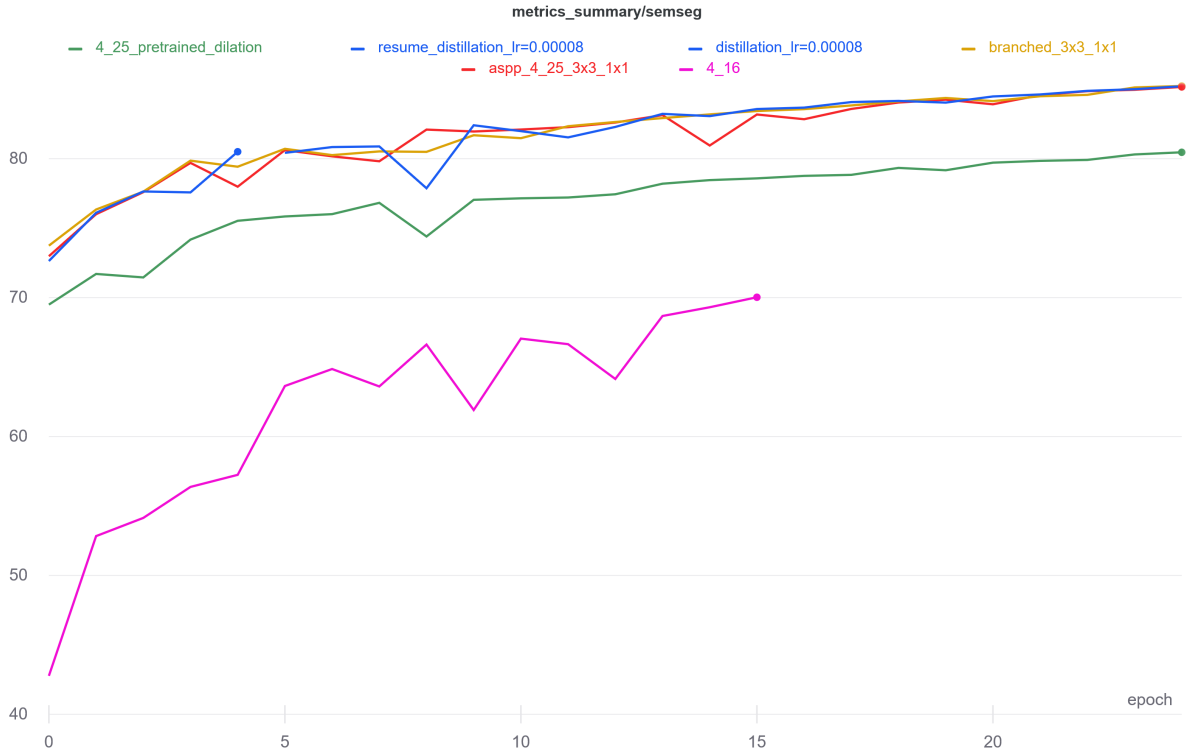

Figure 19: Semantic Segmentation: Model evolution through the tasks

Figure 20: Depth Estimation: Model evolution through the tasks

Final Discussion:
It can be seen that the overall performance improves through the tasks. Obviously, initializing the network with pretrained weights and the last dilation flag enabled gives the network a boost directly at the start and reduces the loss. We define the *4_25_pretrained_dilation* as **baseline**, because the architecture is the original one, but great improvements have been made thanks to the tuning of hyperparameters. The ASPP and decoder with skip connection implementation

aspp_4_25_3x3_1x1 really improves the semantic segmentation from the start to the end of the training: it always performs higher than the baseline.
The depth estimation of the ASPP model fluctuates more during the training (high peaks indicates that the gradient descent takes large steps in opposite direction) but manages to score lower than the baseline (lower is better). With both semantic segmentation and depth estimation progress, the grader score also increased by 7.05 points compared to baseline.

The branched architecture *branched_3x3_1x1* uses two separate decoders for semantic segmentation and depth estimation. It performs a bit better than the ASPP model in semantics, but not really significantly. However the branched architecture depth estimation score is significantly lower (1.322 points lower than the ASPP model at the end of training), and always stays lower than the ASPP model after 13 epochs. This confirms the utility of the branched architecture employing two decoders (Multi-task networks) even though the training time increases.
Finally, the distillation model with self-attention (*distillation_lr=0.00008*) grader score is slightly higher than with the branched architecture (67.877 vs 67.875) but not sufficient to conclude that this enhancement is brought by the distillation architecture. We indeed expected better performance for the distillation architecture implemented in task 3 (with self-attention),in other words to be at least one point above the branched architecture model implemented in task 2 (on the grader) .
Nonetheless, having a look at the depth estimation graph shows that generally over all epochs, the depth estimation of the distillation model tends to be better (lower) than the branched architecture. The final depth estimation score also reflects this (17.341 instead of 17.363).

Note: By ASPP model we always mean the ASPP module + the decoder with skip connections, as done in the end of task 1 (corresponding run is: *aspp_4_25_3x3_1x1*)

Note: For the ASPP model and the branched architecture model, we did not take our best runs into account (i.e. with two convolutions with kernel size 3 followed by one convolution of kernel size 1 for the decoder), because the distillation model would have been to slow to train with such a decoder model. In the interest of relevance, since the decoders in the distillation model are implemented with only one convolution with kernel size 3 followed by one convolution of kernel size 1, we use the same architecture for the branched and ASPP model.

Name	Semantics	Depth	Grader	WandB/AWS S3 submission
resume_distillation_lr=0.00008	85.218	17.341	67.877	G5_0510-2025_resume_full3x3and1x1_distillation_4_25_lr=0.00008_3ccde/
distillation_lr=0.00008	85.218	17.341	67.877	G5_0510-0822_full3x3and1x1_distillation_4_25_lr=0.00008_e74d1/
branched_3x3_1x1	85.238	17.363	67.875	G5_0509-0959_branched_3x3_1x1_aa9fc/
aspp_4_25_3x3_1x1	85.168	18.685	66.483	G5_0508-1348_aspp_4_25_1_3x3_1x1_146a6/
4_25_pretrained_dilation	80.462	21.029	59.433	G5_0514-0747_4_25_pretrained_dilation_e2297
4_16	70.025	26.496	43.53	G5_0508-0651_4_16_dae2c/

Note: The grader score is gently increased for the submission in Codalab. See below the table with the Codalab scores for the submitted runs:

Name	Grader	Codalab submission
resume_distillation_lr=0.00008	68.01	G5_0510-2025_resume_full3x3and1x1_distillation_4_25_lr=0.00008_3ccde/
branched_3x3_1x1	67.97	G5_0509-0959_branched_3x3_1x1_aa9fc/
aspp_4_25_3x3_1x1	66.53	G5_0508-1348_aspp_4_25_1_3x3_1x1_146a6/
4_25_pretrained_dilation	59.48	G5_0514-0747_4_25_pretrained_dilation_e2297
4_16	42.88	G5_0508-0651_4_16_dae2c/

We can see the semantic segmentation and the depth prediction with our best performing model (*branched_2_3x3_1x1*) from Task 2, which scored 69.394 at the grader. Although we expected the distillation model to perform the best, which could have been the case if we kept the decoder architecture with two 3x3 convolution followed by a 1x1 convolution, the training time increased a lot. Overall the best compromise between training time and performance was achieved by

the following run: *branched_2_3x3_1x1*. Hence we provide the results of this model on semantic segmentation and depth estimation for the Cityscapes dataset.

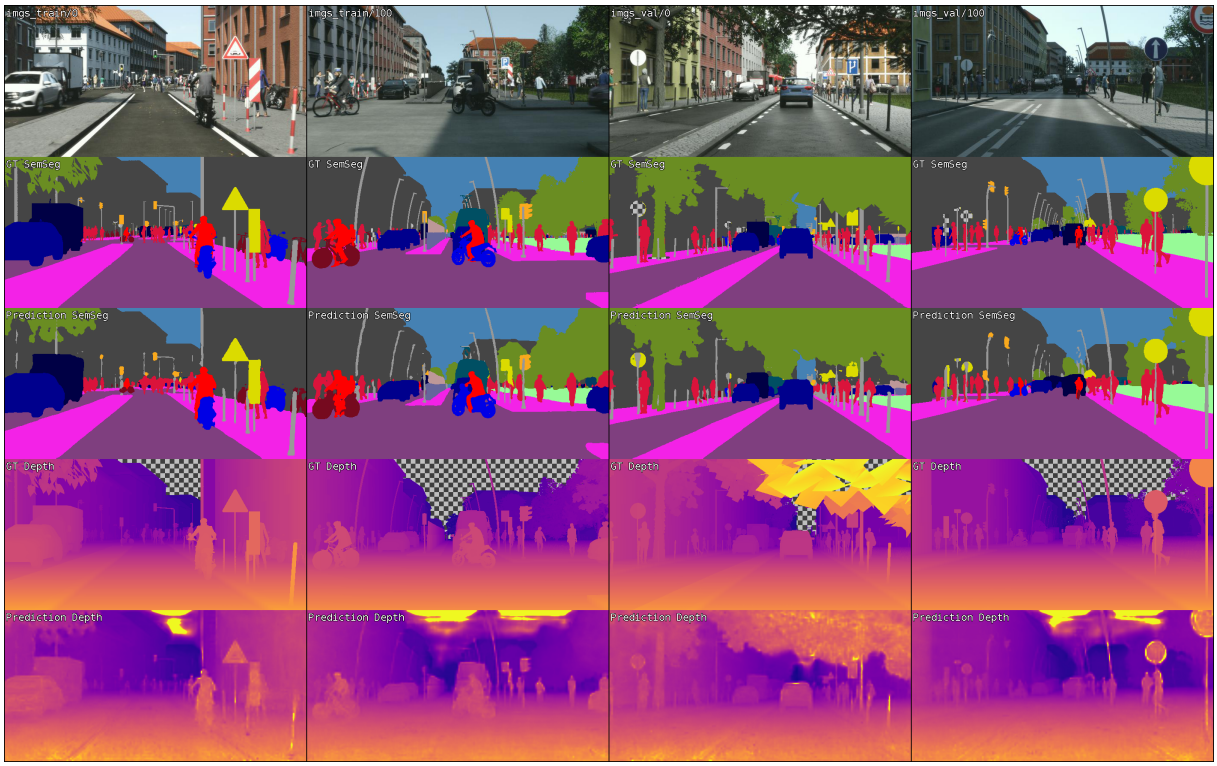

Figure 21: observed samples

4 Appendix

model_branched.py

```python
import torch
import torch.nn.functional as F

from mtl.models.model_parts import Encoder, get_encoder_channel_counts, ASPP, DecoderDeeplabV3p

class ModelBranched(torch.nn.Module):
    def __init__(self, cfg, outputs_desc):
        super().__init__()
        self.outputs_desc = outputs_desc
        #1 channel for depth and n channels for n semantic segmentation classes
        ch_out = sum(outputs_desc.values())

        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)

        #ASPP and decoder for semantic segmentation task
        self.aspp_semseg = ASPP(ch_out_encoder_bottleneck, 256)

        self.decoder_semseg = DecoderDeeplabV3p(256, ch_out_encoder_4x, ch_out-1)

        #ASPP and decoder for depth estimation task
        self.aspp_depth = ASPP(ch_out_encoder_bottleneck, 256)

        self.decoder_depth = DecoderDeeplabV3p(256, ch_out_encoder_4x, 1)

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

        features = self.encoder(x)

        # Uncomment to see the scales of feature pyramid with their respective number of channels.
        #print(", ".join([f"{k}:{v.shape[1]}" for k, v in features.items()]))
        lowest_scale = max(features.keys())

        features_lowest = features[lowest_scale]

        #Semantic Segmentation
        features_tasks_semseg = self.aspp_semseg(features_lowest)
        predictions_4x_semseg, _ = self.decoder_semseg(features_tasks_semseg, features[4])

        predictions_1x_semseg = F.interpolate(predictions_4x_semseg, size=input_resolution, mode='bilinear', align_corners=False)

        #Depth estimation
        features_tasks_depth = self.aspp_depth(features_lowest)
        predictions_4x_depth, _ = self.decoder_depth(features_tasks_depth, features[4])

        predictions_1x_depth = F.interpolate(predictions_4x_depth, size=input_resolution, mode='bilinear', align_corners=False)
        out = {}
        offset = 0

        for task, num_ch in self.outputs_desc.items():
            if task == 'semseg':
                out[task] = predictions_1x_semseg[:, offset:offset+num_ch, :, :]
                offset += num_ch
            elif task == 'depth':
                out[task] = predictions_1x_depth[:,:, :, :]
                offset += num_ch

            else:
                print("mod erreur, should be either MOD_SEMSEG or MOD_DEPTH")
```

```
67          return out
```

Listing 11: Implementation of the class ModelBranched

model_distillation.py

```
import torch
import torch.nn.functional as F

from mtl.models.model_parts import Encoder, get_encoder_channel_counts, ASPP,
    DecoderDeeplabV3p,SelfAttention, Decoder


class ModelDistillation(torch.nn.Module):
    def __init__(self, cfg, outputs_desc):
        super().__init__()
        self.outputs_desc = outputs_desc

        ch_out = sum(outputs_desc.values())

        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)

        #ASPP and decoder for semantic segmentation task
        self.aspp_semseg = ASPP(ch_out_encoder_bottleneck, 256)

        self.decoder_semseg = DecoderDeeplabV3p(256, ch_out_encoder_4x, ch_out-1)

        #ASPP and decoder for depth estimation task
        self.aspp_depth = ASPP(ch_out_encoder_bottleneck, 256)

        self.decoder_depth = DecoderDeeplabV3p(256, ch_out_encoder_4x, 1)

        num_features = 256
        #Self-Attention
        self.attention_semseg = SelfAttention(num_features, num_features)
        self.attention_depth = SelfAttention(num_features, num_features)

        #Decoder 3 and 4
        self.decoder3 = Decoder(num_features,num_features,ch_out-1)
        self.decoder4 = Decoder(num_features,num_features,1)

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

        features = self.encoder(x)

        # Uncomment to see the scales of feature pyramid with their respective number of
    channels.
        #print(", ".join([f"{k}:{v.shape[1]}" for k, v in features.items()]))
        lowest_scale = max(features.keys())

        features_lowest = features[lowest_scale]

        #Semantic Segmentation
        features_task_semseg = self.aspp_semseg(features_lowest)
        predictions_4x_semseg,features_dec1 = self.decoder_semseg(features_task_semseg,
    features[4])
        predictions_1x_semseg = F.interpolate(predictions_4x_semseg, size=input_resolution,
     mode='bilinear', align_corners=False)

        #Depth estimation
        features_task_depth = self.aspp_depth(features_lowest)
        predictions_4x_depth,features_dec2 = self.decoder_depth(features_task_depth,
    features[4])
        predictions_1x_depth = F.interpolate(predictions_4x_depth, size=input_resolution,
    mode='bilinear', align_corners=False)
        #Self-Attention
        features_attention_semseg = self.attention_semseg(features_dec1)
```

```
        features_attention_depth = self.attention_depth(features_dec2)

        #Decoder 3 and 4
        out_semseg_4x,_ = self.decoder3(features_attention_depth, features_dec1)
        out_semseg_1x = F.interpolate(out_semseg_4x, size=input_resolution, mode='bilinear'
    , align_corners=False)
        out_depth_4x,_ = self.decoder4(features_attention_semseg, features_dec2)
        out_depth_1x = F.interpolate(out_depth_4x, size=input_resolution, mode='bilinear',
    align_corners=False)

        out = {}
        offset = 0

        for task, num_ch in self.outputs_desc.items():
            if task == 'semseg':
                out[task] = [out_semseg_1x[:, offset:offset+num_ch, :, :],
    predictions_1x_semseg[:, offset:offset+num_ch, :, :]]
                offset += num_ch
            elif task == 'depth':
                out[task] = [out_depth_1x[:,:, :, :],predictions_1x_depth[:,:, :, :]]
                offset += num_ch

            else:
                print("mod erreur, should be either MOD_SEMSEG or MOD_DEPTH")
        return out
```

Listing 12: Implementation of the class ModelDistillation

References

[1] L.C. Chen, G. Papandreou, F. Schroff, H. Adam *Rethinking atrous convolution for semantic image segmentation.* 5 Dec 2017; url: https://arxiv.org/abs/1706.05587

[2] L.C. Chen, Y. Zhu, G. Papandreou, F. Schroff, H. Adam *Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation.* 22 Aug 2018 ; url: https://arxiv.org/abs/1802.02611

[3] Jun Fu, Jing Liu, Haijie Tian, Yong Li, Yongjun Bao, Zhiwei Fang, Hanqing Lu *Dual Attention Network for Scene Segmentation* 21 Apr 2019; url: https://arxiv.org/abs/1809.02983