# Deep Learning for Autonomous Driving: Lab 2

Team 30
Alan Savio Paul
Loïc Houmard

May 13, 2022

## Problem 1. Joint architecture

### 1.1 Hyper-parameter tuning

(a) *Optimizer and Learning Rate.* For this task, we tried to train our base model with the two different optimizers with 4 different learning rates using a base of 10. The best performing combination was Adam with a small learning rate of 0.0001 as we can see in Table 1. For SGD, the optimal learning rate was the biggest one we tried, i.e. 0.01 and the lower the learning rate, the bigger the error was. As we can seen in Figure 1, SGD tends to learn extremely slowly and somehow underfit with low learning rates. For Adam, the optimal learning rate was 0.0001. It tends not to converge very well with big learning rate as we can see in Figure 2 with a learning rate of 0.01. In general, the higher the learning rate, the more our model weights will change and hence our model can converge faster but can sometimes also miss local minima and move around without properly converging. In this experiment, Adam performed slightly better than SGD. However, Adam performed better with smaller learning rates whereas bigger learning rates were needed for SGD.

| Optimizer | Learning rate | Depth | Segmentation | Multitask |
|---|---|---|---|---|
| SGD | 0.01 | 27.422 | 67.401 | 39.979 |
| SGD | 0.001 | 32.69 | 55.333 | 22.643 |
| SGD | 0.0001 | 39.906 | 39.906 | 10.094 |
| SGD | 0.00001 | 50.224 | 15.226 | 0 |
| Adam | 0.01 | 36.576 | 41.426 | 13.424 |
| Adam | 0.001 | 27.651 | 66.368 | 38.718 |
| Adam | 0.0001 | **26.749** | **69.979** | **43.229** |
| Adam | 0.00001 | 32.721 | 55.675 | 22.957 |

Table 1: SGD vs Adam performance metrics with different learning rates

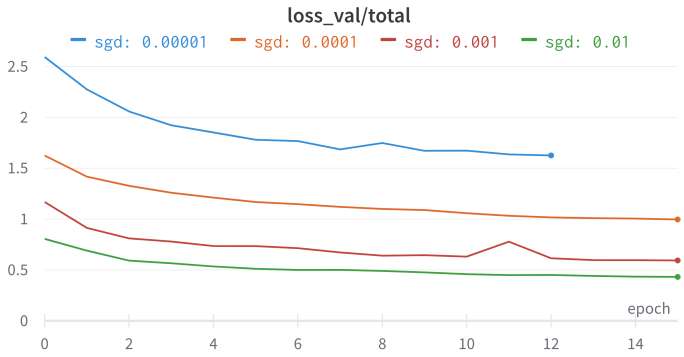


Figure 1: Validation error using SGD with different learning rates

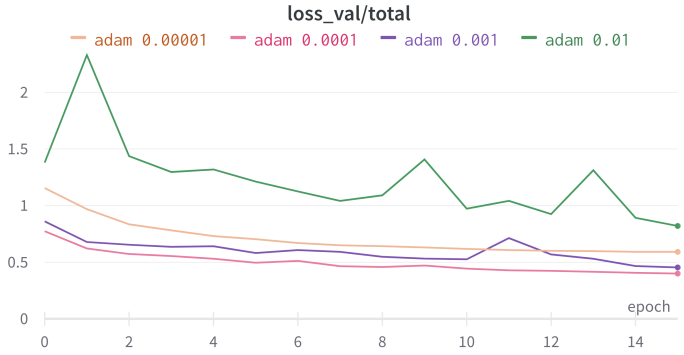


Figure 2: Validation error using Adam with different learning rates

(b) *Batch Size.* For this experiment, we kept the best hyperparameters from the previous experiment, i.e. Adam optimizer and a learning rate of 0.0001. To study the effect of batch size on the performance metrics, we ran experiments using the following batch sizes: 1, 2, 4, 8, 16. We chose batch sizes which are powers of 2 as this allows more efficient usage of the GPU. As we increase the batch size, the number of steps per epoch reduces, and thus we also increased the number of epochs proportionally to ensure that each experiment goes through equal number of optimization steps. Figure 3 shows the validation loss of the models trained with different batch sizes.

Practically, smaller batch sizes mean that we will have noisier gradient steps. This means that we won't necessarily be moving down in the direction of steepest descent, but we get faster computations. This results in lower runtimes as can be seen in Table 3. Larger batch sizes lead to

better gradient steps and as a result they tend to converge in fewer steps but they incur a higher cost per optimization step. Thus, there exists a trade-off between faster training and better convergence.

| Batch Size | Epochs | Depth | Segmentation | Multitask | Runtime |
|---|---|---|---|---|---|
| 1 | 4 | 33.777 | 56.504 | 22.727 | **2h 43m** |
| 2 | 8 | 28.595 | 66.663 | 38.068 | 3h 35m |
| 4 | 16 | 27.006 | 69.754 | 42.748 | 4h 39m |
| 8 | 32 | 25.495 | 71.68 | 46.185 | 8h 2m |
| 16 | 64 | **24.739** | **73.041** | **48.302** | 13h 16m |

Table 2: Effect of Batch Size on Performance Metrics and Runtime

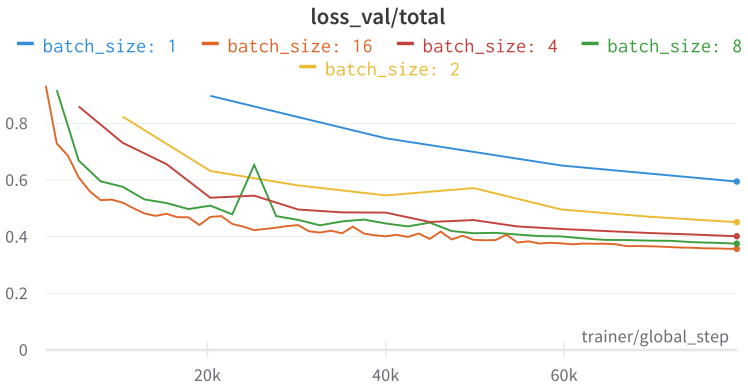


Figure 3: Validation error using different Batch Sizes. All experiments use Adam optimizer with a learning rate of 0.0001.

(c) *Task Weighting.* The two individual losses belonging to the two tasks of the model are part of a weighted sum for the total loss. In this experiment, we try to vary the weight for each individual loss. In particular, we try out weights 0.1, 0.3, 0.5, 0.7 and 0.9. This way we can study whether one loss turns out to be more important than the other. Our experiments results are reported in Figure 4 and Table 4. As we expected, increasing the weight for one individual loss (while decreasing the other) would generally benefit that task while hurting the other, and consequently reduce the multitask score. Since our focus is on multitask learning, we are interested in the loss weights that obtain the highest multitask score. It can be seen that a higher weight for the Segmentation task improves the multitask score. The reason for this could be that the feature representation learned for the semantic segmentation can more easily be used for depth prediction than vice-versa. In other words, focusing more on the semantic segmentation (upto a certain extent) results in better segmentation while

hurting the depth prediction less.

| Depth Weight | SemSeg Weight | Depth | Segmentation | Multitask |
|---|---|---|---|---|
| 0.1 | 0.9 | 29.86 | **70.578** | 40.719 |
| 0.3 | 0.7 | 27.381 | 70.484 | **43.103** |
| 0.5 | 0.5 | 27.006 | 69.754 | 42.748 |
| 0.7 | 0.3 | **26.125** | 68.734 | 42.61 |
| 0.9 | 0.1 | 26.566 | 64.493 | 37.927 |

Table 3: Effect of varying the depth and segmentation loss weight on the performance

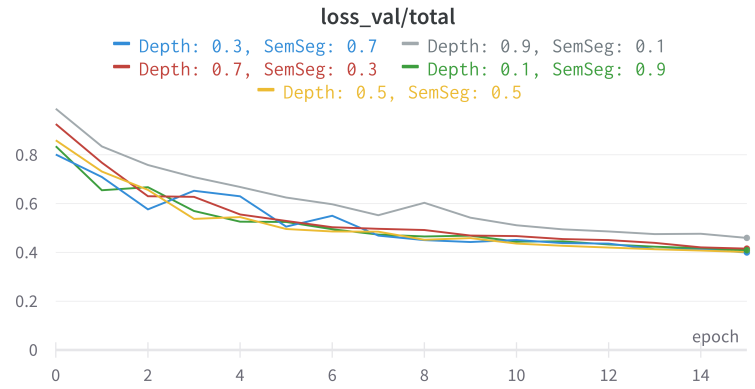


Figure 4: Validation error using different loss weights. All experiments use Adam optimizer with a learning rate of 0.0001.

## 1.2 Hardcoded hyperparameters

(a) The weights of the encoder are by default initialized randomly as stated in torch.vision's documentation for resnet. If we switch the pretrained flag to True, it downloads and uses a model pretrained on ImageNet to initialize the weights. As expected, using the pretrained model for weight initialisation gave us better results (see table 4 and figure 5), especially a lower error at the beginning of the training (and hence faster convergence). The reason is that the weights have already been tuned on a lot of data, and even if the data doesn't come from the exact same distribution, it has still learnt some useful information and performs better than random.

(b) By default, the dilated convolutions are disabled and the encoder uses stride instead to downsample. When using dilation in the last layer of the encoder, the performance improves quite a lot (figure 6). As we can

| Initialization | Depth | Segmentation | Multitask |
|---|---|---|---|
| Random | 26.749 | 69.979 | 43.229 |
| Pretrained | **25.22** | **73.099** | **47.88** |

Table 4: Pretrained vs random initialization performance metrics. Both use Adam optimizer with a learning rate of 0.0001

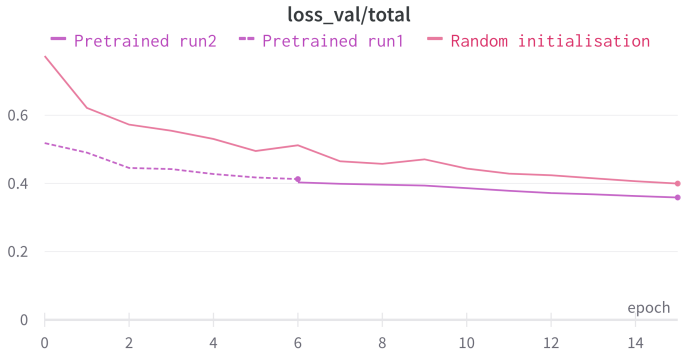


Figure 5: Pretrained vs random initialisation of weights validation error. Both use Adam optimizer with a learning rate of 0.0001

see on the validation images (figure 8 vs figure 7), the model with dilation is especially better on object boundaries. The reason is that dilated convolution can increase the field of view without (or by controlling the amount of) downsampling, whereas strided convolutions only increase the field of view by downsampling the image. Hence, the dilated convolution extract denser feature maps with more detailed information related to object boundaries.

| Model | Depth | Segmentation | Multitask |
|---|---|---|---|
| Without dilation last layer | 25.22 | 73.099 | 47.88 |
| With dilation last layer | **21.57** | **79.93** | **58.36** |

Table 5: With vs without using dilation in the last layer performance metrics. Both use Adam optimizer with a learning rate of 0.0001 and pretrained model.

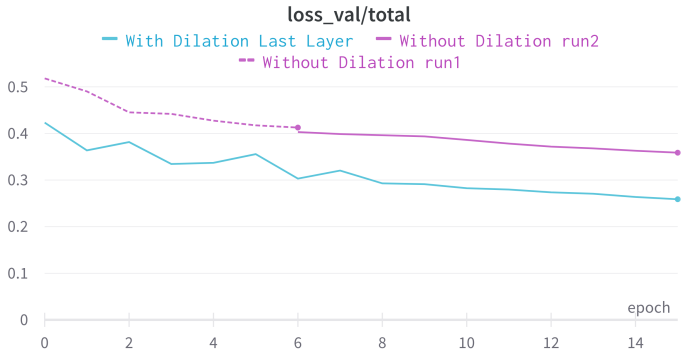


Figure 6: Validation loss using dilation in the last layer or not. Both use Adam optimizer with a learning rate of 0.0001 and pretrained model.

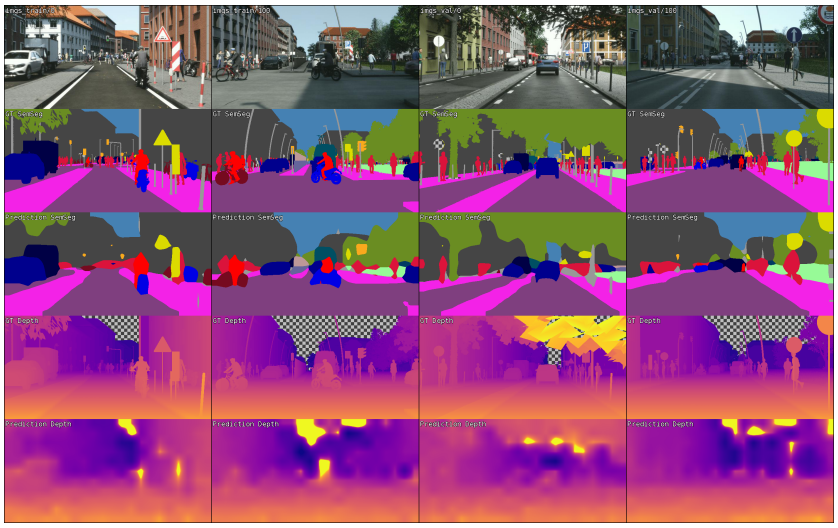

Figure 7: Prediction on validation set without dilation.

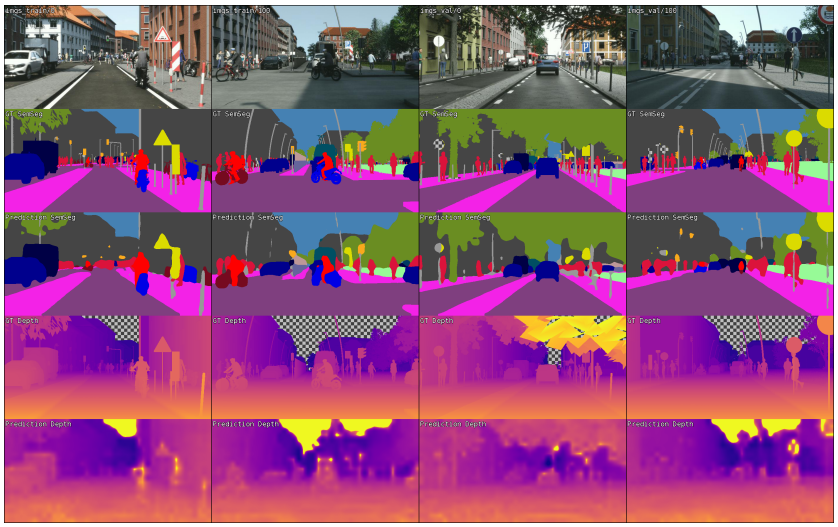

Figure 8: Prediction on validation set using dilation.

### 1.3 ASPP and skip connections

In this section, we implemented the ASPP module and skip connections in the decoder. The ASPP takes the final output of the encoder and uses many different atrous convolutions with different dilation rates (3, 6, 9) in order to extract features at different scales in the image, a 1x1 convolution and a global average pooling to incorporate global context information to the model and then merge them together via a 1x1 final convolution. In the decoder, we use skip connections to merge at the same time the low-level features from the encoder which still contain some raw informations about the image (and hence help to better find object boundaries) and the output of the ASPP module (first bi-linearly upsampled) which contains detailed informations about the image at multi-scale. To merge them, series of convolutions (3 convolutions with batch normalization and ReLU in-between) were applied at the end of the decoder. The choice of using 3 final convolutions (2 256-channel convolutional layers + 1 final convolutional layer) was made because it was the one giving the best results in the papers cited in the handout ([1] and [2]). We also tried to train a model using only 1 final convolution (see table 6, second raw) which, as expected, gave us worse result, probably because 1 layer of convolution is not powerful enough to merge all these features. Adding the ASPP module improved the result quite a lot (see figure 9 and table 6).

| Model | Depth | Segmentation | Multitask |
|---|---|---|---|
| Without ASPP | 21.57 | 79.93 | 58.36 |
| With ASPP, 1 final conv. | 21.046 | 83.343 | 62.297 |
| With ASPP, 3 final conv. | **19.227** | **85.161** | **65.934** |

Table 6: With vs without ASPP performance metrics. Both use Adam optimizer with a learning rate of 0.0001, pretrained model and dilation in the last layer of the encoder.

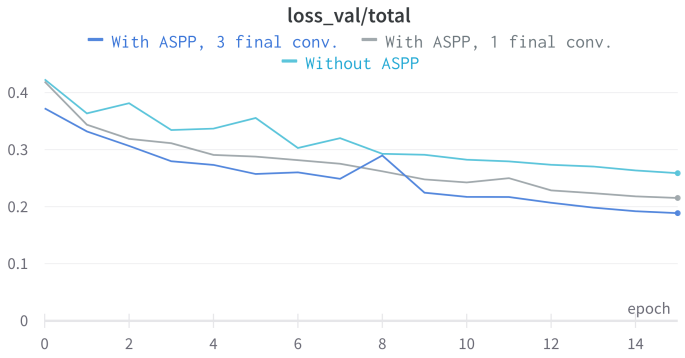


Figure 9: With vs without ASPP validation loss

As we can see on the predicted image (figure 10), a lot of more fine-grained informations are detected by our model with ASPP which were not before using it (figure 8), like body parts. The object boundaries are also sharper than before due to the skip connections in the decoder. However, the depth is still not very smooth, which could be improved.

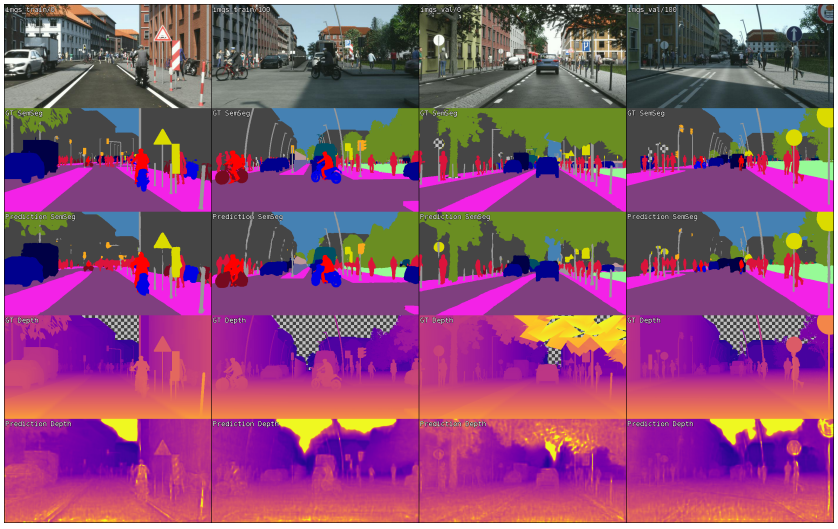

Figure 10: Prediction on validation set using ASPP and 3 final convolutions

Here is our code for this part:

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


        ENCODER_OUTPUT_CHANNELS = 48 #In paper, gives the best result
        self.encoder_convolution = torch.nn.Conv2d(skip_4x_ch,
                                        ENCODER_OUTPUT_CHANNELS,
                                        kernel_size=1, stride=1,
                                        padding=0, dilation=1)
        self.features_to_predictions = torch.nn.Sequential(
            torch.nn.Conv2d(bottleneck_ch+ENCODER_OUTPUT_CHANNELS, 256,
            kernel_size=3, stride=1, padding=1, dilation=1, bias=False),
            torch.nn.BatchNorm2d(256),
            torch.nn.ReLU(),
            torch.nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1,
            dilation=1, bias=False),
            torch.nn.BatchNorm2d(256),
            torch.nn.ReLU(),
```

```
            torch.nn.Conv2d(256, num_out_ch, kernel_size=3, stride=1,
            padding=1, dilation=1, bias=True),
        )

    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
        """
        features_4x = F.interpolate(
            features_bottleneck, size=features_skip_4x.shape[2:],
            mode='bilinear', align_corners=False
        )
        features_encoder = self.encoder_convolution(features_skip_4x)

        concatenation = torch.cat([features_4x, features_encoder], dim=1)

        predictions_4x = self.features_to_predictions(concatenation)

        return predictions_4x, features_4x

class ASPP(torch.nn.Module):
    def __init__(self, in_channels, out_channels, rates=(3, 6, 9)):
        super().__init__()
        self.first = ASPPpart(in_channels, out_channels, kernel_size=1,
                              stride=1, padding=0, dilation=1)
        self.second = ASPPpart(in_channels, out_channels, kernel_size=3,
                              stride=1, padding=rates[0], dilation=rates[0])
        self.third = ASPPpart(in_channels, out_channels, kernel_size=3,
                              stride=1, padding=rates[1], dilation=rates[1])
        self.fourth = ASPPpart(in_channels, out_channels, kernel_size=3,
                              stride=1, padding=rates[2], dilation=rates[2])

        self.globalAveragePooling = torch.nn.AdaptiveAvgPool2d(1)
        self.conv_after_average = ASPPpart(in_channels, out_channels,
                              kernel_size=1, stride=1, padding=0, dilation=1)
        self.out_conv = ASPPpart(5*out_channels, out_channels,
                              kernel_size=1, stride=1, padding=0, dilation=1)

    def forward(self, x):

        first = self.first(x)
        second = self.second(x)
```

```
third = self.third(x)
fourth = self.fourth(x)

#Used to upsample average pooling to right size
_, _, Hout, Wout = first.shape

fifth = F.interpolate(
            self.conv_after_average(self.globalAveragePooling(x)),
            size=(Hout, Wout), mode='bilinear', align_corners=False
)
concatenation = torch.cat([first, second, third, fourth, fifth], dim=1)
out = self.out_conv(concatenation)
return out
```

## Problem 2. Branched architecture

For this task, we implemented the branched architecture using the same building blocks as in the previous question (encoder, decoder, ASPP) but using two different ASPP modules and two different decoders to extract different features for both tasks. The branched architecture performed slightly better overall, especially for the depth estimation as we can see in table 7. This is expected, since in this architecture, our model can learn different useful features for both tasks and is not restricted to learn everything together, which forces a trade-off between the two losses in the representation. Since the depth improves way more than the segmentation, it might mean that our previous model mainly learnt features used for the segmentation which, as already discussed in part 1.1.c, can also be used for the depth. However, this model uses more parameters (approx. 5.5 millions (20%) more) since different parameters must be learnt for both decoders and ASPP modules and hence it takes longer to train as we can see in table 8. We can also derive from this table that most parameters must be in the encoder since the difference between the two models represents only 20% of the parameters. The GPU memory allocated to both models was almost the same for both models (figure 12) and almost constant during the training. We were surprised, as we would have expected the model having more parameters to use more GPU memory.

| Model | Depth | Segmentation | Multitask |
|---|---|---|---|
| With ASPP, 3 final conv. | 19.227 | 85.161 | 65.934 |
| Branched architecture | **17.542** | **85.345** | **67.803** |

Table 7: Branched architecture vs previous architecture with ASPP performance metrics

| Model | Runtime | Number Parameters | GPU memory allocated (%) |
|---|---|---|---|
| With ASPP, 3 final conv. | **7h 18m** | **26'756'996** | 35.07 |
| Branched architecture | 10h 17m | 32'183'220 | **34.44** |

Table 8: Branched architecture vs previous architecture with ASPP resources utilization. The last column gives the mode of GPU memory allocated, which was almost constant during training (see figure 12)

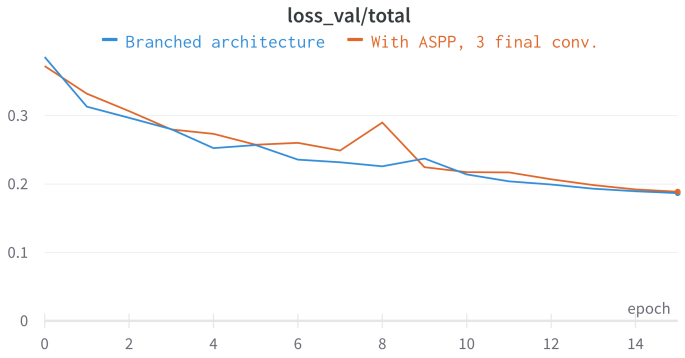


Figure 11: Branched architecture vs ASPP validation loss

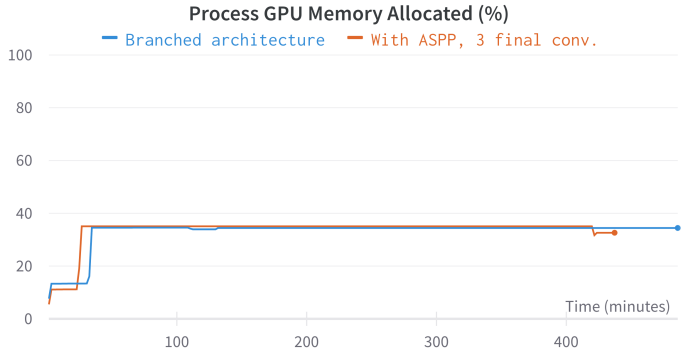


Figure 12: Branched architecture vs ASPP GPU memory allocation

Here is the code for this part. Note that we have not included the small changes in config.py and in helpers.py since they are only useful to launch the training, but not to understand the model.

```
import torch
import torch.nn.functional as F
from mtl.datasets.definitions import MOD_DEPTH, MOD_SEMSEG
```

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

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


        num_classes_semseg = outputs_desc[MOD_SEMSEG]
        num_classes_depth = outputs_desc[MOD_DEPTH]

        self.encoder = Encoder(
            cfg.model_encoder_name,
            pretrained=True,
            zero_init_residual=True,
            replace_stride_with_dilation=(False, False, True),
        )
        ch_out_encoder_bottleneck, ch_out_encoder_4x = get_encoder_channel_counts(
                                                            cfg.model_encoder_name)
        self.aspp_depth = ASPP(ch_out_encoder_bottleneck, 256)
        self.aspp_semseg = ASPP(ch_out_encoder_bottleneck, 256)

        self.decoder_semseg = DecoderDeeplabV3p(256, ch_out_encoder_4x,
                                                num_classes_semseg)
        self.decoder_depth = DecoderDeeplabV3p(256, ch_out_encoder_4x,
                                                num_classes_depth)

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

        lowest_scale = max(features.keys())
        features_lowest = features[lowest_scale]
        features_tasks_semseg = self.aspp_semseg(features_lowest)
        features_tasks_depth = self.aspp_depth(features_lowest)

        predictions_semseg_4x, _ = self.decoder_semseg(features_tasks_semseg,
                                                       features[4])
        predictions_depth_4x, _ = self.decoder_depth(features_tasks_depth,
                                                       features[4])

        prediction_semseg_1x = F.interpolate(predictions_semseg_4x,
                                    size=input_resolution, mode='bilinear',
                                    align_corners=False
```

```
)
prediction_depth_1x = F.interpolate(predictions_depth_4x,
                                    size=input_resolution, mode='bilinear',
                                    align_corners=False
)

out = {}

out[MOD_SEMSEG] = prediction_semseg_1x
out[MOD_DEPTH] = prediction_depth_1x

return out
```

## Problem 3. Task Distillation

In this task, we modified the Branched Architecture from Problem 2 to include Task Distillation by following the instructions in the handout. We used the already-implemented Self Attention module to select the relevant features from one task which can be used for the other. We used the features before the last convolutional layer in the first set of decoders (Decoder 1 and 2)for the self-attention, as mentioned in the handout. The features of each task are summed with the result of self-attention applied on the other task. However, one difference in our implementation is that for the final decoder set (Decoder 3 and 4), we modified the decoders from our previous task (having 3 convolutional layers) instead of using the decoder given to us in the skeleton code (having 1 convolutional layer). The reason for this decision is that we misunderstood the instruction and had already ran our experiments. Therefore our results are probably slightly better than what we would have get using only 1 final convolution.

The results from this experiment are given in Table 9 and Figure 13. Table 10 and Figure 14 compare the resources used by this model compared to the previous ones. The task distillation successfully improves the multitask score as well as each of the individual task scores. The individual scores are improved because the self-attention module exploits the relation between semantic segmentation and depth, and boosts each task by selecting the most important features from the other task. In terms of resources, the task distillation has many more parameters (approx. 4.8 million (15%) more) than the previous best. Optimizing many more parameters takes much longer and this is why this model takes a few hours longer than the previous one for training. The GPU memory requirement also increases for this model, which is natural for deeper networks because of their increased number of parameters.

| Model | Depth | Segmentation | Multitask |
|---|---|---|---|
| With ASPP, 3 final conv. | 19.227 | 85.161 | 65.934 |
| Branched architecture | 17.542 | 85.345 | 67.803 |
| Task Distillation | **16.595** | **85.833** | **69.238** |

Table 9: Task Distillation (Problem 3) vs Branched architecture (Problem 2) vs ASPP architecture (Problem 1) performance metrics

| Model | Runtime | Number Parameters | GPU memory allocated (%) |
|---|---|---|---|
| With ASPP, 3 final conv. | **7h 18m** | **26'756'996** | 35.07 |
| Branched architecture | 10h 17m | 32'183'220 | **34.44** |
| Task Distillation | 18h 19m* | 36'949'960 | 37.56 |

Table 10: Task Distillation vs Branched architecture vs ASPP architecture resources utilization. The last column gives the mode of GPU memory allocated, which was almost constant during training (see figure 14) * This experiment crashed 3 times, so the run time is a little more than it should have been with no crashes. This is because each resumption would redo several optimization steps.

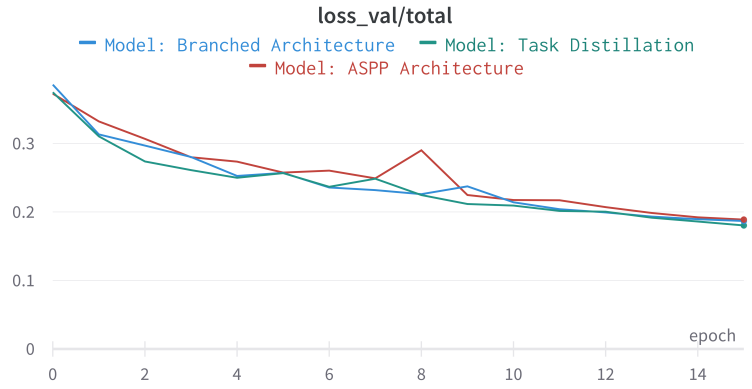


Figure 13: Task Distillation vs Branched architecture vs ASPP validation loss

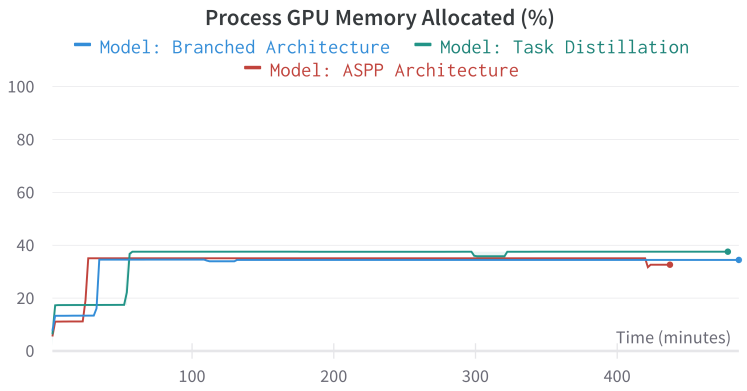


Figure 14: Task Distillation vs Branched architecture vs ASPP GPU memory allocation

Here is the code for the modified Decoder (Decoder 3 and 4), followed by another code block with the architecture of the Task Distillation network.

```
import torch

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

        self.features_to_predictions = torch.nn.Sequential(
            torch.nn.Conv2d(num_in_ch, 256, kernel_size=3,
                            stride=1, padding=1,
                            dilation=1, bias=False),
            torch.nn.BatchNorm2d(256),
            torch.nn.ReLU(),
            torch.nn.Conv2d(256, 256, kernel_size=3,
                            stride=1, padding=1,
                            dilation=1, bias=False),
            torch.nn.BatchNorm2d(256),
            torch.nn.ReLU(),
            torch.nn.Conv2d(256, num_out_ch, kernel_size=3,
                            stride=1, padding=1,
                            dilation=1, bias=True),
        )


    def forward(self, features):
        """
        DeepLabV3+ style decoder
        :param features: sum of features coming from previous
                         decoder and features after SelfAttention
                         module, 4 times downsample
        """

        predictions_downsampled = self.features_to_predictions(features)

        return predictions_downsampled
```

Below is the code for the architecture of this task. As before, we have not included changes in config.py and helpers.py.

```
import torch
import torch.nn.functional as F
from mtl.datasets.definitions import MOD_DEPTH, MOD_SEMSEG

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

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

        num_classes_semseg = outputs_desc[MOD_SEMSEG]
        num_classes_depth = outputs_desc[MOD_DEPTH]

        self.encoder = Encoder(
            cfg.model_encoder_name,
            pretrained=True,
            zero_init_residual=True,
            replace_stride_with_dilation=(False, False, True),
        )

        ch_out_encoder_bottleneck, ch_out_encoder_4x = \
                                get_encoder_channel_counts(
                                cfg.model_encoder_name
                                )

        self.aspp_depth = ASPP(ch_out_encoder_bottleneck, 256)
        self.aspp_semseg = ASPP(ch_out_encoder_bottleneck, 256)

        self.first_decoder_semseg = DecoderDeeplabV3p(256,
                                                      ch_out_encoder_4x,
                                                      num_classes_semseg)
        self.first_decoder_depth = DecoderDeeplabV3p(256,
                                                     ch_out_encoder_4x,
                                                     num_classes_depth)

        self.attention_semseg = SelfAttention(256, 256)
        self.attention_depth = SelfAttention(256, 256)

        self.second_decoder_semseg = DecoderNoSkipConnection(256, num_classes_semseg)
        self.second_decoder_depth = DecoderNoSkipConnection(256, num_classes_depth)


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

        features = self.encoder(x)

        lowest_scale = max(features.keys())

        features_lowest = features[lowest_scale]
```

```
features_tasks_semseg = self.aspp_semseg(features_lowest)
features_tasks_depth = self.aspp_depth(features_lowest)

# Initial prediction after decoder #1 and #2
initial_predictions_semseg_4x, _, initial_penultimate_semseg_4x = \
                                    self.first_decoder_semseg(
                                          features_tasks_semseg,
                                          features[4]
                                    )
initial_predictions_depth_4x, _, initial_penultimate_depth_4x = \
                                    self.first_decoder_depth(
                                            features_tasks_depth,
                                            features[4]
                                        )
initial_prediction_semseg_1x = F.interpolate(initial_predictions_semseg_4x,
                                             size=input_resolution,
                                             mode='bilinear',
                                             align_corners=False)
initial_prediction_depth_1x = F.interpolate(initial_predictions_depth_4x,
                                            size=input_resolution,
                                            mode='bilinear',
                                            align_corners=False)

after_attention_semseg = self.attention_semseg(initial_penultimate_semseg_4x)
after_attention_depth = self.attention_depth(initial_penultimate_depth_4x)

sum_semseg = torch.add(initial_penultimate_semseg_4x, after_attention_depth)
sum_depth = torch.add(initial_penultimate_depth_4x, after_attention_semseg)

# Final prediction after decoder #3 and #4
final_prediction_semseg_4x = self.second_decoder_semseg(sum_semseg)
final_prediction_depth_4x = self.second_decoder_depth(sum_depth)
final_prediction_semseg_1x = F.interpolate(final_prediction_semseg_4x,
                                           size=input_resolution,
                                           mode='bilinear',
                                           align_corners=False)
final_prediction_depth_1x = F.interpolate(final_prediction_depth_4x,
                                          size=input_resolution,
                                          mode='bilinear',
                                          align_corners=False)


out = {}

out[MOD_SEMSEG] = [initial_prediction_semseg_1x, final_prediction_semseg_1x]
```

```
        out[MOD_DEPTH] = [initial_prediction_depth_1x, final_prediction_depth_1x]

        return out
```

# Problem 4. Open Challenge

For the open challenge, we decided to try to augment our task-distillation model (baseline) to learn contour and normal by adding two losses (one for each of them) and then using the learnt representation via self-attention to improve our final depth and segmentation result. Our hope was that firstly, our encoder would learn more general features useful for both tasks and secondly that, thanks to the attention module, some useful features learnt for the new tasks would help our final prediction. Our approach keeps most of the element of our previous architecture for task-distillation and augments it using the ideas proposed in the paper on Pad-Net [4], module c. The overall architecture we wanted to implement looks as shown in figure 15. We started by augmenting our model using the contours only (see section 4.1), then used the normals only (section 4.2) and since our results using the normals only was worse than our previous results with task-distillation, we finally didn't implement the full architecture (figure 15) as we wanted at first.

## Problem 4.1. Segmentation contour

In this part, we augmented our task-distillation model by adding the contour of the segmentation image only (see figure 16).

To implement it, we first extracted the contour based on the segmentation ground truth. This is simply done by checking whenever two neighboring pixels have different semantic labels:

```
def compute_contour_given_labels(y_semseg_lbl):
    """
    Compute the edges given the semantic labels of each pixels

    Parameters
    ----------
    y_semseg_lbl : torch.tensor(B, H, W)
        The semantic labels for each pixels for each image of the batch

    Return
    ----------
    edges : torch.tensor(B, H, W)
        The edges of the semantic images with 1 for the pixels being edges
        (change between 2 different classes, the one being to the right
        or below the other is considered as an edge) or 0 otherwise.
    """
```

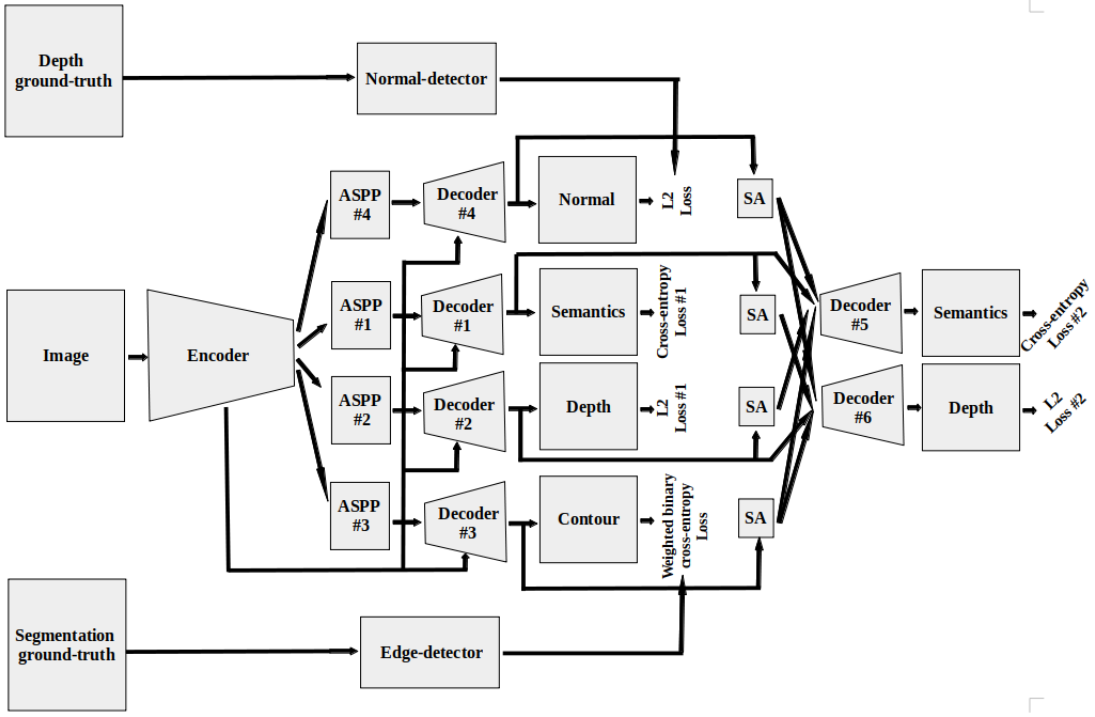


Figure 15: Full architecture using contour and normal

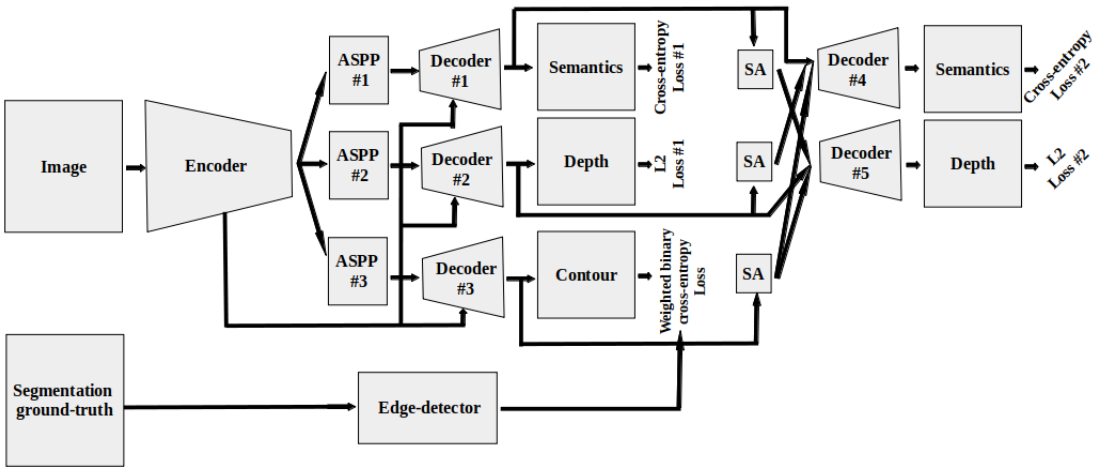


Figure 16: Architecture using contour only

```
    B, H, W = y_semseg_lbl.shape

    edges = torch.zeros((B, H, W), device="cuda" if torch.cuda.is_available() else "cpu")

    edges[:, 1:, :] = torch.ne(y_semseg_lbl[:, 1:, :], y_semseg_lbl[:, :H-1, :])
    edges[:, :, 1:] = edges[:, :, 1:] + torch.ne(
                                        y_semseg_lbl[:, :, 1:],
                                        y_semseg_lbl[:, :, :W-1]
                                        ) > 0

    return edges.to(torch.float)
```

We then used these contours as ground truth and created a new branch in our architecture to predict them. To train it, we first used a normal binary cross-entropy loss, but our performance was not improving and we realized, from Figure 2 and Section 3.1 in [3], that it was due to the high imbalance between non-contour (black values) and contour (white values) where most of the pixels are non-contour, and hence our model was not able to learn the contours successfully. To overcome that, we switched the loss to a weighted binary cross-entropy as also done in paper [4]. Our code for the loss is the following:

```
class WeightedBCE(torch.nn.Module):
    @staticmethod
    def forward_one_image(prediction, label):

        label = label.long()
        mask = label.float()
        num_positive = torch.sum((mask == 1).float()).float()
        num_negative = torch.sum((mask == 0).float()).float()

        mask[mask == 1] = 1.0 * num_negative / (num_positive + num_negative)
        mask[mask == 0] = 1.1 * num_positive / (num_positive + num_negative)
        mask[mask == 2] = 0
        cost = torch.nn.functional.binary_cross_entropy(prediction.float(),
        label.float(), weight=mask)

        if label.numel() == 0:
            return None, False
        return cost, True

    def forward(self, y_hat, y):
        assert torch.is_tensor(y_hat) and torch.is_tensor(y)
        assert y.dim() == 3 or y.dim() == 4 and y.shape[1] == 1
        if y.dim() == 4:
            y = y.squeeze(1)
```

```
assert y_hat.dim() == 3 or y_hat.dim() == 4 and y_hat.shape[1] == 1
if y_hat.dim() == 4:
    y_hat = y_hat.squeeze(1)
assert y_hat.shape == y.shape
N, H, W = y_hat.shape

out_loss = torch.tensor(0, device=y.device, dtype=y.dtype,
                        requires_grad=True)
out_cnt = 0
for i in range(N):
    loss, have_loss = self.forward_one_image(y_hat[i], y[i])
    if have_loss:
        out_loss = out_loss + loss
        out_cnt += 1

return out_loss / max(out_cnt, 1)
```

We then used the features before the last convolutional layer of our decoder 3, passed it through a self-attention module and added it to the corresponding features for the semantics and depth. To train our overall model, we used a smaller weight for this task as for the two other main tasks. We first tried to put the weight slightly below the other two weights, and it slightly improved our baseline, by improving the segmentation a bit, which was expected since the contour was extracted from the ground-truth of the segmentation and hence, we expected the segmentation to improve more than the depth. However, since the improvement wasn't very significant, it's hard to say whether our model really learnt to extract the contour and use them or if it was due to randomness or simply to the fact that our model has slightly more parameters (table 12). We then realized that in the paper [4], they use 0.8 for this loss and 1 for the four other losses and we wanted to use the same ratio ($\frac{0.8}{4.8} = 0.16$), but we made a mistake in our run (used the wrong weight) and only discovered it while writing the report, hence we were not able to try this setting on time. It might probably have improved the results slightly, but we think that it wouldn't have improved them significantly.

The results from our experiment with the contours added are given in Table 11 and Figure 17.

| Model | Weights<br>Contour/ Seg./ Depth | Depth | Segmentation | Multitask |
|---|---|---|---|---|
| Task Distillation | -/ 0.5 / 0.5 | **16.595** | 85.833 | 69.238 |
| Task Distillation + Contours | 0.28/ 0.36/ 0.36 | 16.662 | **86.183** | **69.521** |

Table 11: Performance metrics of Task Distillation with Contours vs without Contours

Our code for the Task Distillation architecture with Contours is shown below.

| Model | Runtime | Number Parameters | GPU memory allocated (%) |
|---|---|---|---|
| Task Distillation | **18h 19m*** | **36'949'960** | **37.56** |
| Task Distillation + Contours | 19h 5m | 43'558'137 | 40.07 |

Table 12: Task Distillation with Contours vs without Contours resources utilization. The last column gives the mode of GPU memory allocated, which was almost constant during training * This experiment crashed 3 times, so the run time is a little more than it should have been with no crashes. This is because each resumption would redo several optimization steps.

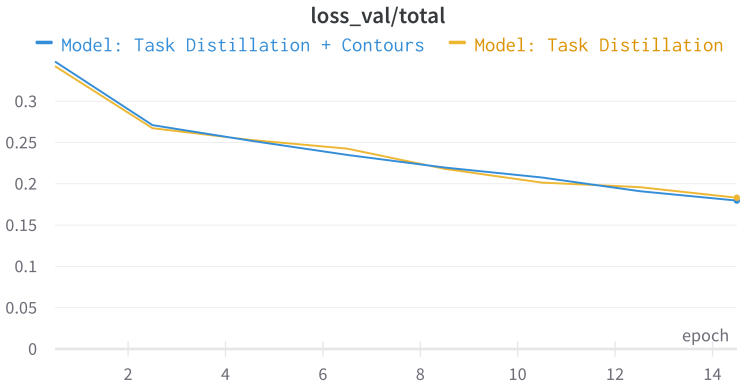


Figure 17: Comparing Validation loss of Task Distillation with and without Contours

Note that we haven't included the small changes in experiment_semseg_with_depth.py, helpers.py and config.py since we don't think that they are very informative and they were rather straight forward:

```
import torch
import torch.nn.functional as F
from mtl.datasets.definitions import MOD_DEPTH, MOD_SEMSEG, MOD_CONTOUR

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

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

        num_classes_semseg = outputs_desc[MOD_SEMSEG]
        num_classes_depth = outputs_desc[MOD_DEPTH]
```

```
        num_classes_contour = outputs_desc[MOD_CONTOUR]

        self.encoder = Encoder(
            cfg.model_encoder_name,
            pretrained=True,
            zero_init_residual=True,
            replace_stride_with_dilation=(False, False, True),
        )

        ch_out_encoder_bottleneck, ch_out_encoder_4x =
            get_encoder_channel_counts(cfg.model_encoder_name)

        self.aspp_depth = ASPP(ch_out_encoder_bottleneck, 256)
        self.aspp_semseg = ASPP(ch_out_encoder_bottleneck, 256)
        self.aspp_contour = ASPP(ch_out_encoder_bottleneck, 256)

        self.first_decoder_semseg = DecoderDeeplabV3p(256, ch_out_encoder_4x,
                                                      num_classes_semseg)
        self.first_decoder_depth = DecoderDeeplabV3p(256, ch_out_encoder_4x,
                                                     num_classes_depth)
        self.first_decoder_contour = DecoderDeeplabV3p(256, ch_out_encoder_4x,
                                                       num_classes_contour)
        self.output_decoder_contour_to_probability = torch.nn.Sigmoid() # Because
        # each pixel must be between 0 (not an edge) and 1 (an edge) for BCE

        self.attention_semseg = SelfAttention(256, 256)
        self.attention_depth = SelfAttention(256, 256)
        self.attention_contour = SelfAttention(256, 256)

        self.second_decoder_semseg = DecoderNoSkipConnection(256, num_classes_semseg)
        self.second_decoder_depth = DecoderNoSkipConnection(256, num_classes_depth)


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

        features = self.encoder(x)

        lowest_scale = max(features.keys())

        features_lowest = features[lowest_scale]

        features_tasks_semseg = self.aspp_semseg(features_lowest)
        features_tasks_depth = self.aspp_depth(features_lowest)
        features_tasks_contour = self.aspp_contour(features_lowest)
```

```
# Initial prediction after decoder #1, #2 and #3
initial_predictions_semseg_4x, _, initial_penultimate_semseg_4x =
    self.first_decoder_semseg(features_tasks_semseg, features[4])
initial_predictions_depth_4x, _, initial_penultimate_depth_4x =
    self.first_decoder_depth(features_tasks_depth, features[4])
initial_predictions_contour_4x, _, initial_penultimate_contour_4x =
    self.first_decoder_contour(features_tasks_contour, features[4])

initial_prediction_semseg_1x = F.interpolate(initial_predictions_semseg_4x,
                                    size=input_resolution, mode='bilinear',
                                    align_corners=False)
initial_prediction_depth_1x = F.interpolate(initial_predictions_depth_4x,
                                    size=input_resolution, mode='bilinear',
                                    align_corners=False)
initial_prediction_contour_1x = F.interpolate(initial_predictions_contour_4x,
                                    size=input_resolution, mode='bilinear',
                                    align_corners=False)
initial_probabilities_contour_1x =
    self.output_decoder_contour_to_probability(initial_prediction_contour_1x)

after_attention_semseg = self.attention_semseg(initial_penultimate_semseg_4x)
after_attention_depth = self.attention_depth(initial_penultimate_depth_4x)
after_attention_contour = self.attention_contour(initial_penultimate_contour_4x)

sum_semseg = torch.add(torch.add(initial_penultimate_semseg_4x,
                                 after_attention_depth),
                                 after_attention_contour)
sum_depth = torch.add(torch.add(initial_penultimate_depth_4x,
                                after_attention_semseg),
                                after_attention_contour)

# Final prediction after decoder #4 and #5
final_prediction_semseg_4x = self.second_decoder_semseg(sum_semseg)
final_prediction_depth_4x = self.second_decoder_depth(sum_depth)
final_prediction_semseg_1x = F.interpolate(final_prediction_semseg_4x,
                                           size=input_resolution,
                                           mode='bilinear',
                                           align_corners=False)
final_prediction_depth_1x = F.interpolate(final_prediction_depth_4x,
                                          size=input_resolution,
                                          mode='bilinear',
                                          align_corners=False)


out = {}
```

```
        out[MOD_SEMSEG] = [initial_prediction_semseg_1x, final_prediction_semseg_1x]
        out[MOD_DEPTH] = [initial_prediction_depth_1x, final_prediction_depth_1x]
        out[MOD_CONTOUR] = initial_probabilities_contour_1x

        return out
```

Another modification to our contour detection which could possibly improve the multitask results is to use the edges from the RGB image (by using the canny edge detector, for example) as ground truth instead of using the edges from semantic images. This brings more information about edges within object boundaries that the semantic segmentation ground truth does not have and this information could potentially boost the depth prediction more.

## Problem 4.2. Surface Normals

In this part, we did something very similar to the previous part, except that we computed the normals instead of the contour (figure 18).

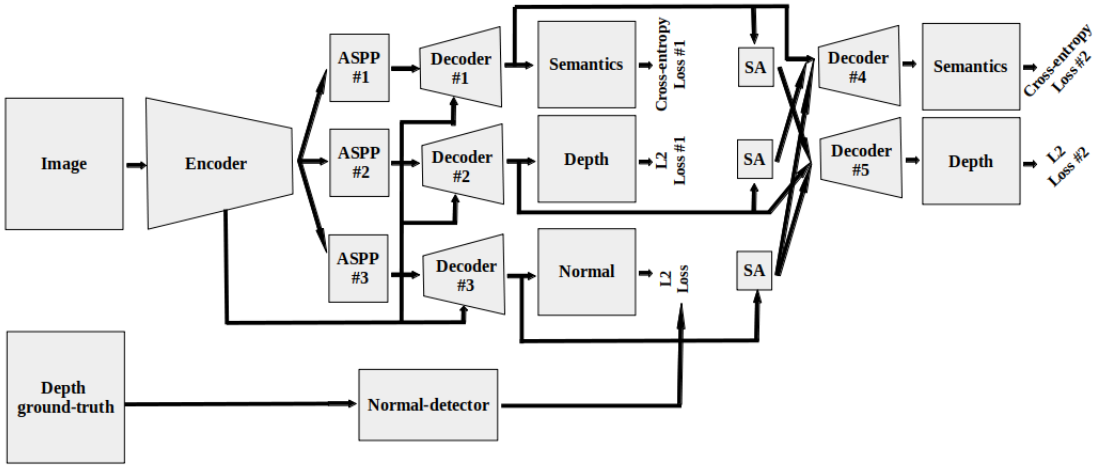


Figure 18: Architecture using normal only

The normals are 3D vectors perpendicular to the tangent plane at every point normalized to have length 1. The rest of the architecture is the same, except that we use 3 channels at the end of the decoder (one for each dimension) and then a tanh activation function instead of the sigmoid to have values in the [-1, 1] range which are the only possible values for normalized vectors. Since the code is similar as before except for the aforementioned point, we are not showing it here. For the loss, we use the L2 loss just like for the depth. We slightly changed the code so that it fits our structure. The code for the normal computation (which was adapted to python without for loops

from the following post https://answers.opencv.org/question/82453/calculate-surface-normals-from-depth-image-using-neighboring-pixels-cross-product/) and our custom L2 loss are shown here:

```
def compute_normals(depth_image):
    """
    Compute the normals given the depth of each pixel using cross-product of
    neighboring pixels. Set first row and column to same normal as neighbors
    since it cannot be computed for them.

    Parameters
    ----------
    depth_image : torch.tensor(B, H, W)
        The depth for each pixels for each image of the batch. Might contain
        Nan values for pixels not to consider.

    Return
    ----------
    output : torch.tensor(B, 3, H, W)
        The normal (3D) at each pixel for the whole batch. Might contain Nan
        values if the depth contains Nan values for pixels not to consider.
    """
    B, H, W = depth_image.shape

    output = torch.zeros((B, 3, H, W), dtype=torch.float, device="cuda"
                                if torch.cuda.is_available() else "cpu")

    xvalues = torch.arange(0, W, device="cuda" if torch.cuda.is_available()
                                    else "cpu")
    yvalues = torch.arange(0, H, device="cuda" if torch.cuda.is_available()
                                    else "cpu")

    """
    /* * * * *
    * * t * *
    * l c * *
    * * * * */
    """

    tyy, txx = torch.meshgrid(yvalues[:H-1], xvalues[1:])
    t = torch.cat([torch.stack([txx, tyy], dim=2).unsqueeze(0).repeat(B, 1, 1, 1)
        .to(torch.float), depth_image[:, :H-1, 1:].unsqueeze(3)], dim=3)

    lyy, lxx = torch.meshgrid(yvalues[1:], xvalues[:W-1])
    l = torch.cat([torch.stack([lxx, lyy], dim=2).unsqueeze(0).repeat(B, 1, 1, 1)
        .to(torch.float), depth_image[:, 1:, :W-1].unsqueeze(3)], dim=3)
```

```
        cyy, cxx = torch.meshgrid(yvalues[1:], xvalues[1:])
        c = torch.cat([torch.stack([cxx, cyy], dim=2).unsqueeze(0).repeat(B, 1, 1, 1)
            .to(torch.float), depth_image[:, 1:, 1:].unsqueeze(3)], dim=3)


        d = torch.cross(torch.sub(l, c), torch.sub(t, c), dim=3)
        n = torch.nn.functional.normalize(d, dim=3).transpose(2,3).transpose(1,2)

        output[:, :, 1:, 1:] = n
        output[:, :, 0, :] = output[:, :, 1, :]
        output[:, :, :, 0] = output[:, :, :, 1]

        return output

    class LossNormal(torch.nn.Module):
        @staticmethod
        def forward_one_image(y_hat, y):
            valid_mask = y == y  # filter out NaN values
            y_hat = y_hat[valid_mask]
            y = y[valid_mask]

            if y.numel() == 0:
                return None, False

            # L1 penalty
            #loss = (y_hat - y).abs().mean()
            # L2 penalty
            loss = (y_hat - y).pow(2).mean()

            return loss, True

        def forward(self, y_hat, y):
            assert torch.is_tensor(y_hat) and torch.is_tensor(y)
            assert y.shape[1] == 3 and y_hat.shape[1] == 3 and y_hat.shape == y.shape
            B, C, H, W = y_hat.shape

            out_loss = torch.tensor(0, device=y.device, dtype=y.dtype, requires_grad=True)
            out_cnt = 0
            for i in range(B):
                loss, have_loss = self.forward_one_image(y_hat[i], y[i])
                if have_loss:
                    out_loss = out_loss + loss
                    out_cnt += 1

            return out_loss / max(out_cnt, 1)
```

Since the normal didn't improve our code, we decided not to try the full architecture using normals and contours (figure 15) which would probably have been worse than the one using contours only and would have wasted GPU utilization.

| Model | Weights Normal/ Seg./ Depth | Depth | Segmentation | Multitask |
|---|---|---|---|---|
| Task Distillation | -/ 0.5 / 0.5 | **16.595** | **85.833** | **69.238** |
| Task Distillation + Normals | 0.16/ 0.42/ 0.42 | 17.284 | 85.573 | 68.288 |
| Task Distillation + Normals | 0.28/ 0.36/ 0.36 | 16.678 | 85.792 | 69.114 |

Table 13: Performance metrics of Task Distillation with normals vs without normals

| Model | Runtime | Number Parameters | GPU memory allocated (%) |
|---|---|---|---|
| Task Distillation | **18h 19m*** | **36'949'960** | **37.56** |
| Task Distillation + Normals | 20h 8m | 43'562'747 | 40.12 |

Table 14: Task Distillation with normals vs without normals resources utilization. The last column gives the mode of GPU memory allocated, which was almost constant during training * This experiment crashed 3 times, so the run time is a little more than it should have been with no crashes. This is because each resumption would redo several optimization steps.

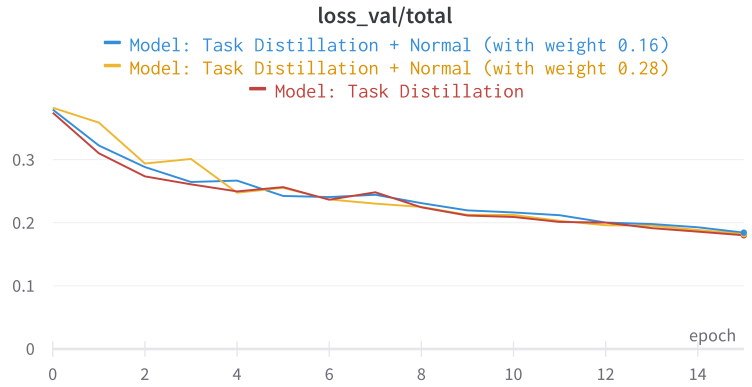


Figure 19: Comparing Validation loss of Task Distillation with and without Normals

### Problem 4.3. Trying to improve edge boundaries

We tried a very last small experiment which didn't improve our results either, but we thought was worth mentioning. We realized that our architecture still had some problems to detect sharp object boundaries, even when the contour was used. We think that one of the main reason for that is that the last operation of our last decoders is an upsampling, which doesn't allow our model to be very precise. Hence, we tried to use a simple canny edge detector on the input image to detect the edges of the images and to concatenate the result with our previous final prediction and use a last convolution on that to try to learn how to use this information to refine our predictions. It didn't improve our model, but we still believe that there is room for improvement along that way. We believe that using only 1 convolution was not enough and that maybe some smarter architecture might be needed instead of simply applying a canny edge detector and using a concatenation. Here is the code that we added at the end of the forward function in our task-distillation model:

```
self.final_convolution_depth = torch.nn.Conv2d(num_classes_depth+1,
                            num_classes_depth, kernel_size=5, stride=1, padding=2)
self.final_convolution_semseg = torch.nn.Conv2d(num_classes_semseg+1,
                            num_classes_semseg, kernel_size=5, stride=1, padding=2)

B, C, H, W = x.shape
contours = torch.empty(0, H, W)
for rgb_image in x:
    # We want channels as last dimension for Canny to work
    rgb_image = rgb_image.transpose(0, 1).transpose(1, 2)

    contour = torch.from_numpy(cv.Canny(rgb_image.cpu().numpy().astype(np.uint8),
                            100, 200)).float()/255 # /255 to have either 0 or 1
    contours = torch.cat([contours, contour.unsqueeze(0)])
contours = contours.unsqueeze(1) #To have 1 channel

if torch.cuda.is_available():
    contours = contours.cuda()

semseg_concat_with_contour = torch.cat([final_prediction_semseg_1x, contours], dim=1)
depth_concat_with_contour = torch.cat([final_prediction_depth_1x, contours], dim=1)

final_prediction_semseg_1x = self.final_convolution_semseg(semseg_concat_with_contour)
final_prediction_depth_1x = self.final_convolution_depth(depth_concat_with_contour)
```

# References


[1] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint*

*arXiv: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. In *Proceedings of the European conference on computer vision (ECCV)*, pages 801–818, 2018.

[3] Ruoxi Deng, Chunhua Shen, Shengjun Liu, Huibing Wang, and Xinru Liu. Learning to predict crisp boundaries. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 562–578, 2018.

[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. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 675–684, 2018.

# Appendix

To compute the scores of all our experiments shown in the tables, we used the following metrics:

- IoU (intersection-over-union) is a metric of performance of the semantic segmentation task. Its values lie in the range [0,100]. Higher values are better. It was used for the segmentation evaluation.
- SI-logRMSE (scale-invariant log root mean squared error) is a metric of performance of the monocular depth prediction task. Its values are positive. Lower values are better. It was used for the depth evaluation.
- The Multitask metric is a simple product of the aforementioned task-specific metrics, computed as max(iou−50, 0) + max(50−SIlogRMSE, 0). Its values lie in the range [0,100]. Higher values are better.