# Deep Learning for Autonomous Driving - Project 2
# Team 36

**Mesut Ceylan**
Legi: 18-748-012
`ceylanm@student.ethz.ch`

**Adrian Hoffmann**
Legi: 15-933-930
`adriahof@student.ethz.ch`

June 12, 2020

We are the team does.not.matter on CodaLab with hellweek being Mesut and pleasure2cu being Adrian.

## 1 Problem 1

### 1.1 Task 1.1

We first trained a few models to get a feel for the setting and then trained 9 models to answer the questions. In all experiments we used the default hyper parameters unless stated otherwise.
Additionally, we will not go in alphabetical order from (a) through (d) to facilitate better story telling.

**(b)** We had good experiences in previous projects (private and at ETH) with learning rate 0.001 on Adam. Consequently, we were more interested in learning rates with SGD as we didn't have so much prior experience with it. In figure 1 (left) you can see the grader scores of three models during training with the learning rates 0.1, 0.01, and 0.001. Each model was trained with a batch-size of 32. We can barely see that the default 0.01 learning rate edges out the 0.1 learning rate. The performance of 0.001 is far behind them.
In the following experiments you will see that we didn't test different learning rates for Adam. This is because we were given the hint that the learning rate for Adam is smaller than for SGD. Hence trying a larger one would be nonsensical. And a smaller learning rate seemed to be a bad idea since the performance hasn't plateaued after the default 16 epochs for all Adam trained models.

**(a)** Models trained with the Adam optimizer always performed better than those trained with SGD. And did so with some margin. All models that were trained with Adam achieved a grader score above 40 while all models trained with SGD stayed below 36.

**(c)** For these experiments we trained three models, all with Adam and a learning rate of 0.001. In figure 1 (middle) we can see that the differences in performance of the tested batch sizes (16, 32, and 64) are razor thin. Their best grader scores are 42.72, 43.37, and 43.47 respectively, giving a large batch size of 64 the narrow victory.

**(d)** We chose a pair of weightings in the following manner: we looked at the best performing model we had at that point (the best one from **(c)**) and looked at the validation losses of the two tasks. Specifically, we looked at the losses after 936 training steps. It showed that the loss for the depth task is only about half that of the semantic segmentation task. Hence, we wanted to even this out and gave the tasks weights 0.65 and 0.35 respectively. We also trained a model with these weights swapped since it might be that the depth task simply "piggy-back rides" on the semantic segmentation task. The models were trained with Adam, learning rate 0.001, and batch-size 64.
We were very surprised to see that neither option improved the grader score. So we went a step further and used a weighting of 0.8 and 0.2 (depth and semantic segmentation respectively). But again no improvement. You can see this in figure 1 (right).

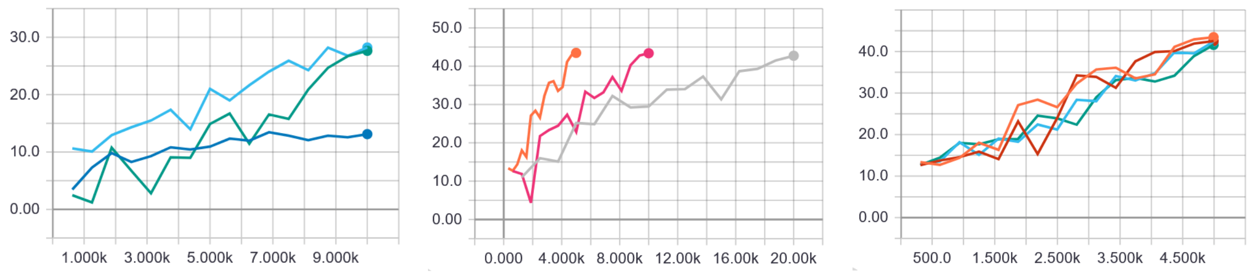


Figure 1: Training graphs of the grader metric from different experiments for problem 1. **(left)** experiment with different learning rates:green is 0.1, light blue is 0.01, and dark blue is 0.001 **(middle)** experiment with different batch sizes: 16 is grey, 32 is pink, and 64 is orange **(right)** experiments with different task weightings for depth-semseg: 0.65-0.35 is red, 0.35-0.65 is light blue, and 0.8-0.2 is green. The orange line is from the best model from task 1.1.(c) for reference

**Conclusion problem 1.1:** The best performance up to that point were achieved with Adam, learning rate 0.001, batch size 64 and a fifty-fifty weighing of the two task. The model trained in this way gave us a grader score of 43.47, the metric for the depth task went all the way down to 26.36 and for semantic segmentation it achieved 69.83. Consequently we, whenever possible, used these parameters in the remaining experiments.

## 1.2 Task 1.2

In this part of the project, we analyzed the hardcoded hyperparameters namely initiating model with ImageNet weights as well as Dilated convolutions and conducted one-line changes.

**(Initialization with ImageNet weights)** It is well-known fact in the computer vision area that utilization of pre-trained ImageNet weights greatly assists neural network architectures on classification task. In this section of the project, as it is guided, we benefited from ImageNet weights. We made the necessary quick code changes to initialize the model with those weights. After training the model with ImageNet weights, we obtained 46.1548 overall score on grader.

**(Dilated convolutions)** Second subsection of of hardcoded hyperparameter is integration of Dilated Convolutions. As stated in the project guidelines, we set the dilation flags to "(False, False, True)" and trained the model. These convolutions are formed in a way to include varying receptive fields. Compared to standard convolution operation including standard kernel, dilated convolutions allow the model to control field of view of the kernel as stated in [3]. With this special convolution, the model is able to obtain different resolution levels of the features. In our code, by setting stated flags into "(False, False, True)", we replaced stride with dilation. Overall, by implementing dilated convolution, we are able to extract different spatial resolution which greatly helped our model to achieve higher score. According to our submission on codalab, we obtained overall 57.58. Figure 2 depicts the performance difference of our models designed with guidelines on task 1.2.a and 1.2.b.

## 1.3 Task 1.3

In this section of the project, we implemented the ASPP module according to following research papers [3] and [2]. According to the research papers, we completed the model with TODO annotations. With this modul implementation, we overall obtained 57.32 score on codalab competition. Including this module did not help much on our model. We believe that including only this module is not yet complete architecture of the overall framework therefore we expected the results accordingly.

## 1.4 Task 1.4

On top of ASPP module in the encoder part of the framework, we implement skip connection module in the decoder part of the framework. We observed that model performance increased in every performance metric compared to previous tasks.

Figure 3 shows the model performance after implementing encoder and decoder with proper functionalities.

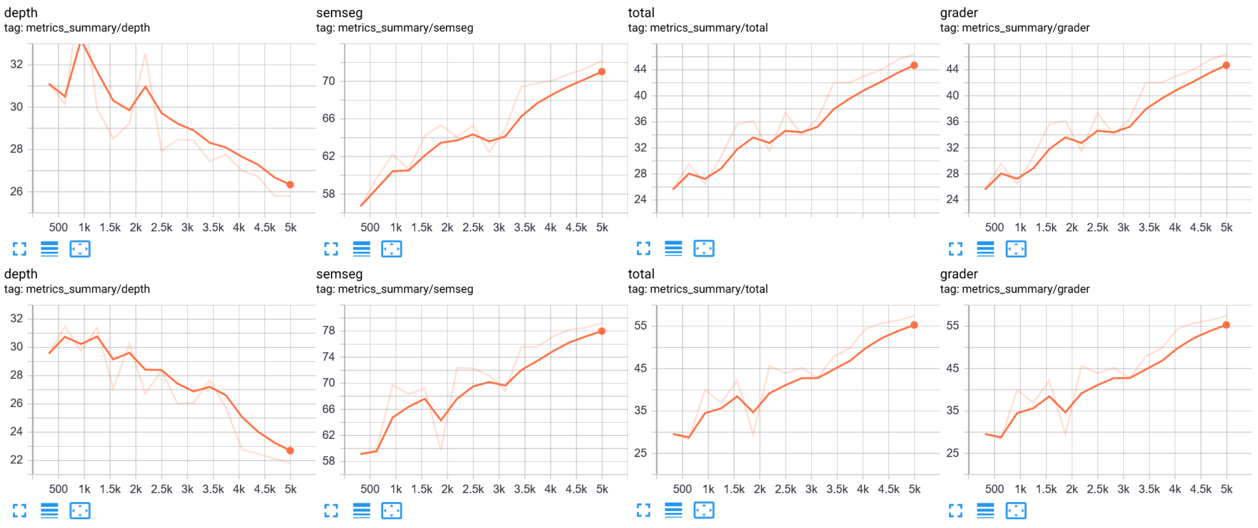


Figure 2: Tensorboard graphs of the Task 1.2.a and 1.2.b. **(Above)** Trained model with only ImageNet Weights, achieving metric scores following: 25.08 on depth, 72.20 on semantic segmentation, 46.40 on total and on 46.40 grader. **(Below)** Trained model with ImageNet Weights as well as dilated convolution achieving metric scores following: 21.82 on depth, 79.23 on semantic segmentation, 57.42 on total and 57.42 on grader.

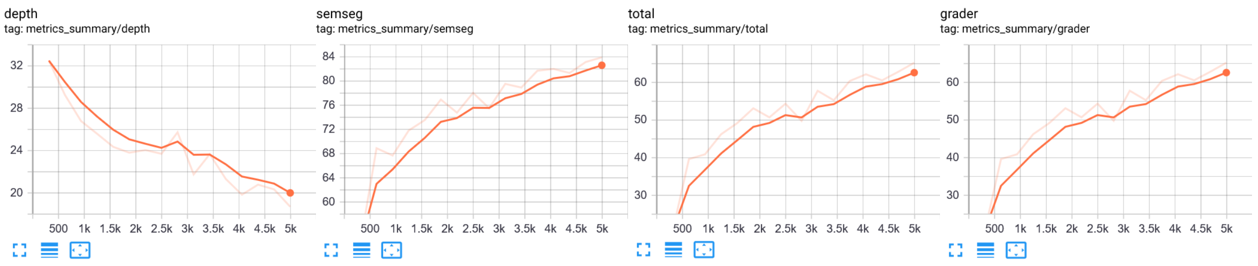


Figure 3: Tensorboard graphs of the Task 1.4. Tensorboard screenshots of our trained model ASPP and Skip Connection module included. We achieved the following performance metric scores: 18.7 on depth, 83.92 on semantic segmentation, 65.22 on total and grader.

Completion of ASPP module with Skipping Connection, we finally obtained proper encoder and decoder that fuse different features of the input. We also observed practical performance boost of complete model after implementing task 1.4. Our model reached the best performance given previous task. It reached 65.22 overall on grader metric.

## 2 Problem 2

The changes needed in the code for this problem are rather easy. All we needed to do was add another ASPP and Decoder instance so that each task has their own ASPP and Decoder. Additionally, we had to choose the appropriate number of output channels for each task in the decoders. You can find the code in *dlad_ex2_multitask/mtl/models/model_problem_2.py* on CodaLab. The according submission has description "problem 2" and it was submitted by pleasure2cu.
For training we used Adam, a learning rate of 0.001, and batch size 32. We had to decrease the batch size due to memory constraints. The model achieved the following results on the metrics on the test set (i.e. on CodaLab): 64.71 grader, 19.28 depth, and 83.99 semantic segmentation. Which is about the same as the best performance in problem 1 but with double the training steps (due to the halved batch size).
The number of trainable parameters increased to approximately 32 million from about 27 million in the architecture from problem 1.4 - so a nearly 20% increase. Additionally, all added 5 million parameters are not pre-trained. In this context it is understandable that the larger architecture needed more update steps to get about the same performance as the smaller architecture.
Interestingly, the training time actually decreased. The training time in problem 1.4 was 10 hours and 17 minutes against a time of 9 hours and 33 minutes in this problem. We hypothesis that the smaller batch size allowed the hardware of

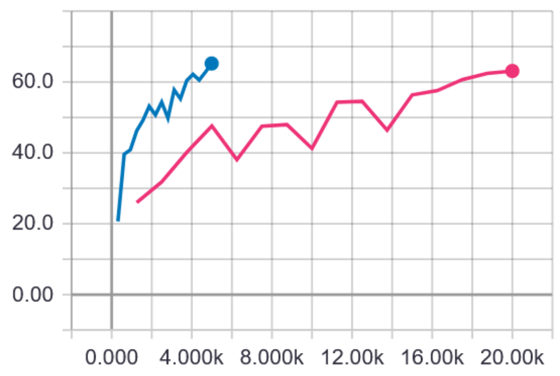


Figure 4: The pink curve is the grader metric during training of the model from problem 3. In blue is the same curve for the model from problem 1.4 for reference. Note: the latter has a four times larger the batch-size than the former.

AWS to store more of the weights and losses in higher levels of the memory hierarchy making the whole computation faster despite performing more update steps.

## 3 Problem 3

The code changes for this problem were trickier than the changes in the previous problem. We identified the following issues:

1. implement a decoder model-part for decoders #3 and #4 from the given architecture. We could not reuse the given decoder since it has two inputs (the second one is for the skip connection)[1]
2. instantiating the new decoders and self-attention units with the correct dimensions
3. wiring all parts up in the correct manner

Regarding the first issue we went with the design from [1] where two deconvolution layers increase the resolution to the required size and a final convolution layer produces the desired number of channels (19 for segmentation and 1 for depth). You can find our decoder (named SkinnyDecoder) in appendix A. The second issue turned out to be rather easy since the self-attention layer needs to retain the dimensions of its input so that its output can be added to the tensor from the other task. Finally, the wiring up of everything can be found in the appendix B. The full code is part of the submission with description "problem 3" made by pleasure2cu on CodaLab. The changes are in the files *model_problem_3.py* and *model_parts.py* in the folder *dlad_ex2_multitask/mtl/models/*.
We proceeded to train the model with the Adam optimizer, a learning rate of 0.001, and batch size 16 (batch size 32 gave us a memory error). The metrics at the end of the default 16 epochs were 63.25 (grader), 19.22 (depth), and 82.47 (semantic segmentation) on the test data (i.e. on CodaLab). These are about the same as for the model in problem 2.

## 4 Problem 4

At this section of the project, we analyzed different methods and metrics to increase our model performance. First of all, we set our model batch size to 16. Secondly, we introduced geometric data augmentations as stated DeepLab paper [4]. As paper states, randomly scaling input helped their performance. Thus, we also include this data augmentation in our project. We decided to scale the input with min scale parameter 0.5 and max scale parameter as 1.5. In addition to this data augmentation, we also tried to include horizontal flipping in our model as [3] paper included and benefited from this data augmentation strategy.

We also experimented activating all data augmentation options such as wiggling and tilting but those data augmentation methods degraded our model performance. Finally, we kept crop size 256 due to results of experimentation. We learnt the rational behind cropping size is to feed the model with object view directly to make it focus into more main objects than extracting information from other less important vision elements. Figure 5 depicts model performance differences

[1]see also post 81 on Piazza

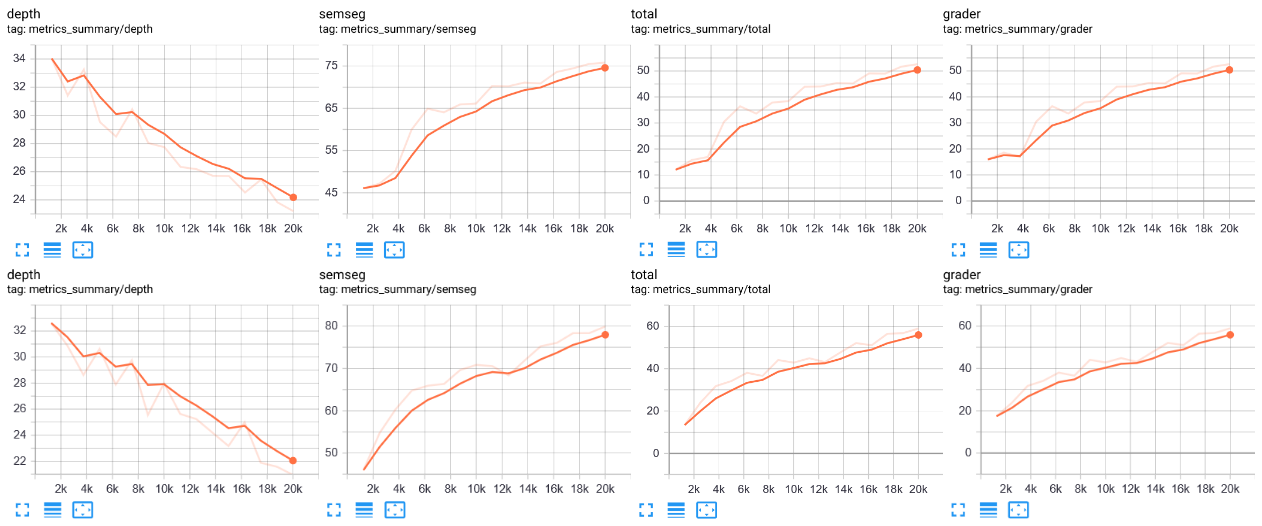


Figure 5: Tensorboard graphs of the Task 4 **(Above)** Trained model with all data augmentation (wiggle, scale, crop, horizontal flip) with reasonable metrics. We obtained metric scores as following: 23.2 on depth, 75.81 on semantic segmentation, 52.61 on total and grader. **(Below)** Trained model with crop size 256, min scaling 0.5, max scaling 1.5 data augmentation methods.We obtained the following metric scores: 20.94 on depth, 79.92 on semantic segmentation, 58.97 on total and grader.

between differing data augmentation methods. We observed that best to include scale, crop and horizontal flip.

As a group, we have strict time constraints during project timeline, we unfortunately were not able to test SqueezeAndExcitation module and implementation of more than one skipping connections on AWS.

When we submitted and tested all of our models, we observed that the best performing model was model from task 1.4 after completing fusion of ASPP and Skipping Connection. That model achieved 65.32 on Multitask, 84.09 on IoU and 18.77 on SI-logRMSE CodaLab competition page. With these scores, our placement is 29th in the competition.

## References

[1] Xu, D., Ouyang, W., Wang, X., Sebe, N.: Pad-net: Multi-tasks guided prediction-and-distillation network for simultaneous depth estimation and scene parsing. In: Proceedingsof the IEEE Conference on Computer Vision and Pattern Recognition. pp. 675–684 (2018)

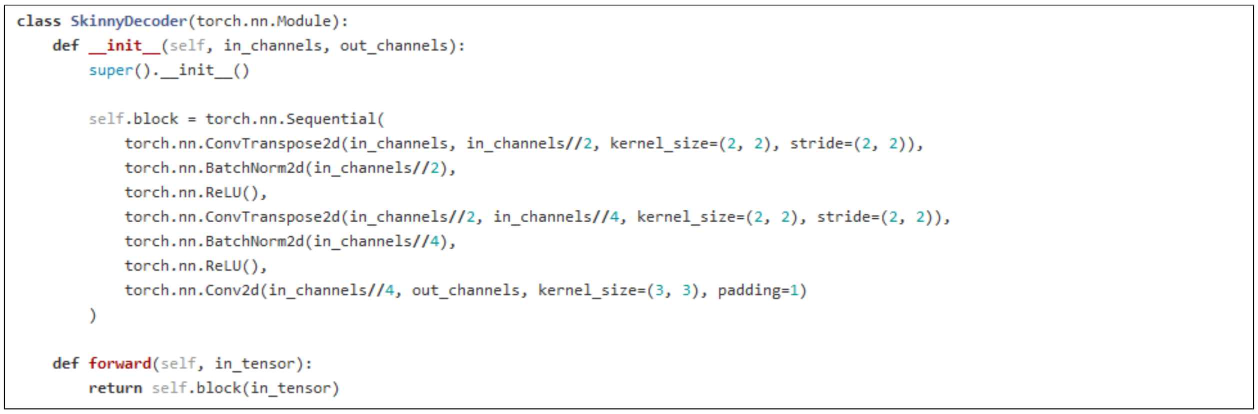
[2] Chen, L.C., Papandreou, G., Schroff, F., Adam, H.: Rethinking atrous convolution forsemantic image segmentation. arXiv preprint arXiv:1706.05587 (2017)

[3] Chen, L.C., Zhu, Y., Papandreou, G., Schroff, F., Adam, H.: Encoder-decoder with atrousseparable convolution for semantic image segmentation. In: Proceedings of the Europeanconference on computer vision (ECCV). pp. 801–818 (2018)

[4] Chen, L.C., Papandreou, G., Kokkinos, I., Murphy, K., Yuille, A.L.: Deeplab: Semanticimage segmentation with deep convolutional nets, atrous convolution, and fully connectedcrfs. IEEE transactions on pattern analysis and machine intelligence40(4), 834–848 (2017)

# Appendices

## A Skinny Decoder Code



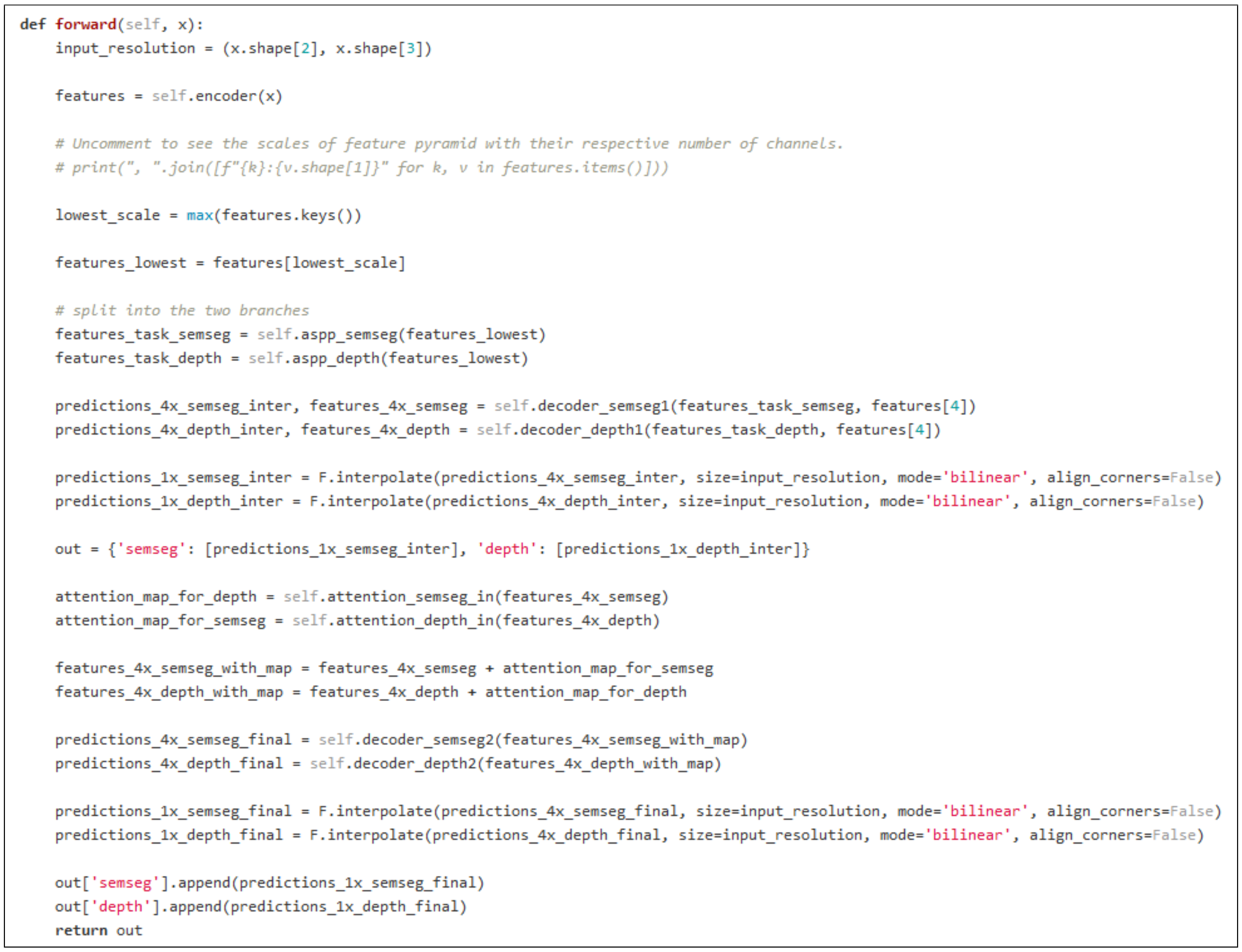
```python
class SkinnyDecoder(torch.nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()

        self.block = torch.nn.Sequential(
            torch.nn.ConvTranspose2d(in_channels, in_channels//2, kernel_size=(2, 2), stride=(2, 2)),
            torch.nn.BatchNorm2d(in_channels//2),
            torch.nn.ReLU(),
            torch.nn.ConvTranspose2d(in_channels//2, in_channels//4, kernel_size=(2, 2), stride=(2, 2)),
            torch.nn.BatchNorm2d(in_channels//4),
            torch.nn.ReLU(),
            torch.nn.Conv2d(in_channels//4, out_channels, kernel_size=(3, 3), padding=1)
        )

    def forward(self, in_tensor):
        return self.block(in_tensor)
```

Figure 6: Code for the decoders #3 and #4 from problem 3.

## B Problem 3 model forward function



```python
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]

    # split into the two branches
    features_task_semseg = self.aspp_semseg(features_lowest)
    features_task_depth = self.aspp_depth(features_lowest)

    predictions_4x_semseg_inter, features_4x_semseg = self.decoder_semseg1(features_task_semseg, features[4])
    predictions_4x_depth_inter, features_4x_depth = self.decoder_depth1(features_task_depth, features[4])

    predictions_1x_semseg_inter = F.interpolate(predictions_4x_semseg_inter, size=input_resolution, mode='bilinear', align_corners=False)
    predictions_1x_depth_inter = F.interpolate(predictions_4x_depth_inter, size=input_resolution, mode='bilinear', align_corners=False)

    out = {'semseg': [predictions_1x_semseg_inter], 'depth': [predictions_1x_depth_inter]}

    attention_map_for_depth = self.attention_semseg_in(features_4x_semseg)
    attention_map_for_semseg = self.attention_depth_in(features_4x_depth)

    features_4x_semseg_with_map = features_4x_semseg + attention_map_for_semseg
    features_4x_depth_with_map = features_4x_depth + attention_map_for_depth

    predictions_4x_semseg_final = self.decoder_semseg2(features_4x_semseg_with_map)
    predictions_4x_depth_final = self.decoder_depth2(features_4x_depth_with_map)

    predictions_1x_semseg_final = F.interpolate(predictions_4x_semseg_final, size=input_resolution, mode='bilinear', align_corners=False)
    predictions_1x_depth_final = F.interpolate(predictions_4x_depth_final, size=input_resolution, mode='bilinear', align_corners=False)

    out['semseg'].append(predictions_1x_semseg_final)
    out['depth'].append(predictions_1x_depth_final)
    return out
```

Figure 7: Wiring of the model for problem 4.