

Dr. Christos Sakaridis

# Project 1

Group 08

Yannick Neuffer, yneuffer@ethz.ch

Thorbjörn Höllwarth, hoellwat@ethz.ch

Lukas Nüesch, lnueesch@ethz.ch

17.11.2024

## 1 Semantic Segmentation

### 1.1 Hyper-parameter tuning

#### a) Optimizer and learning rate

In figures 1 and 2, SGD and Adam are tested as optimizers with different learning rates. The results compare the validation loss after one epoch. With the optimal learning rate, Adam shows better performance than SGD, achieving lower validation loss. Since Adam uses both momentum and adaptive learning rates, it enables faster convergence by making more refined updates to each parameter based on gradient history.

The optimal learning rate for Adam was found to be  $10^{-4}$ , which is notably lower than the optimal learning rate for SGD, which was  $10^{-1}$  in our experiment. This difference can be explained by Adam's adaptive mechanism, that adapts the learning rate based on the magnitude of recent gradients, which reduces the need for high learning rates. SGD on the other hand often requires a higher learning rate to make progress.

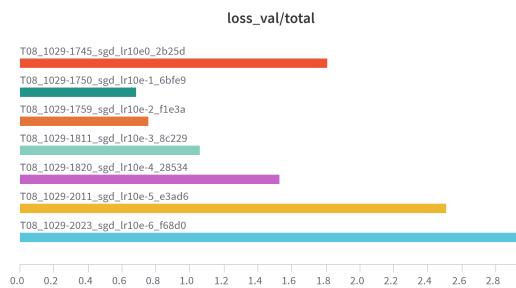


Figure 1: SGD with different learning rates.  
 Minimum: 0.68 with lr =  $10^{-1}$ .

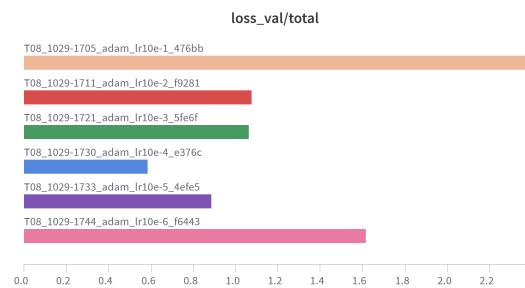


Figure 2: Adam with different learning rates.  
 Minimum: 0.58 with lr =  $10^{-4}$ .

#### b) Batch size

The batch size impacts several aspects. Figure 3 depicts the validation losses for different batch sizes while changing the epochs proportionally. Generally, larger batch sizes will

lead to more stable and reliable gradients and thus more stable convergence. This can be observed in Figure 3 as the validation loss decreases and convergence speed increases as the batch size increases. Further, larger batch sizes require more memory. This is can be observed in Figure 4. Although the number of updates per epoch decreases with larger batch sizes, in practice the total training time increases in our experiments (Figure 4) with proportional epochs. This suggests that increasing the number of epochs has a disproportionately larger effect on training time than increasing the batch size. Therefore we kept a batch size of 16 for later experiments as a tradeoff.

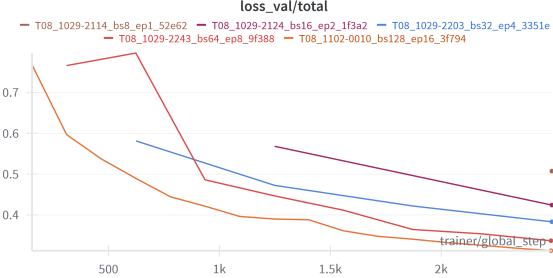


Figure 3: Validation loss for different batch sizes with proportionally adjusted number of epochs.

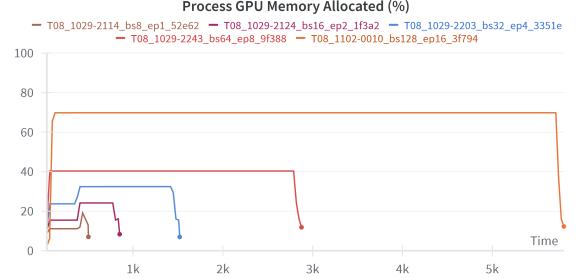


Figure 4: Memory usage for different batch sizes.

## 1.2 Hardcoded hyperparameters

### 1.2.1 Encoder initialization

**Encoder initialization:** How are the weights of the encoder network initialized in the solution template?

The flag "pretrained" is set to False in the template, leading the weights and biases to be initialised randomly.

**What happens when you switch the pretrained flag of the encoder?**

The flag pretrained triggers the weights and biases of the ResNet. If this flag is set to False, they are initialised randomly. If it's set to True, the pretrained values are being used.

**How do both variants compare performance-wise and what might be the reason for that?**

As shown in Figure 5, the Pretrained model consistently outperforms the Template model, achieving lower loss values in both training and validation, which indicates reduced error. Additionally, the Pretrained model achieves higher scores across various segmentation tasks, demonstrating superior accuracy and generalization. Overall, the Pretrained model is clearly more effective than the Template model. This superior performance is due to the Pretrained model having been exposed to additional data and having undergone more extensive training, which enables it to generalize better and achieve lower error rates across various tasks.

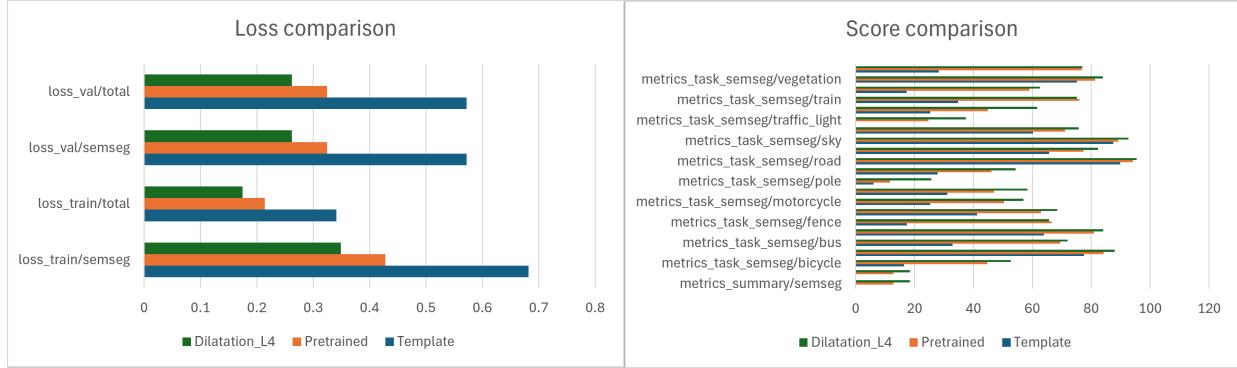


Figure 5: Performance comparison: Not pretrained, pretrained and dilated network.

### 1.2.2 Dilated convolutions

#### Check whether dilated convolutions are enabled.

To verify whether dilated convolutions are enabled in the model, we start by examining the default settings in `model_parts.py`, where the `replace_stride_with_dilation` parameter is defined as `(False, False, False)`. This default configuration indicates that dilated convolutions are disabled. Further inspection in `model_deeplab_v3_plus.py` shows that the calling function explicitly sets `replace_stride_with_dilation` to `(False, False, True)`, reaffirming that dilated convolutions are indeed turned off by default.

Code Snippet 1: Toggle dilated convolutions.

```

1 class ModelDeepLabV3Plus(torch.nn.Module):
2     def __init__(self, cfg, outputs_desc):
3         super().__init__()
4         self.outputs_desc = outputs_desc
5         ch_out = sum(outputs_desc.values())
6
7         self.encoder = Encoder(
8             cfg.model_encoder_name,
9             pretrained=cfg.pretrained,
10            zero_init_residual=True,
11            replace_stride_with_dilation=(False, False, True), #
12            default: F,F,F
13     )

```

The manipulation of these values directly affects the model's receptive field. If the first flag in `replace_stride_with_dilation` is set to `True`, the code modifies the `conv2` layer within `layer2` by adjusting both the padding and dilation parameters to `(2, 2)`. This change effectively expands the receptive field of the convolution without increasing the number of parameters. Similarly, if the second flag is set to `True`, the same padding and dilation adjustments are applied to `conv2` within `layer3`. Lastly, enabling the third flag will apply these adjustments to `conv2` within `layer4`.

By expanding the receptive field, these modifications allow the model to capture patterns over larger spatial areas without the added computational cost of additional parameters.

This can be especially beneficial in tasks requiring greater contextual awareness, as the model can incorporate information from a broader area of the input image.

### **Relation to the term “output stride”.**

The concept of output stride is closely related to the use of dilated convolutions in the model. Output stride refers to the total down-sampling factor of the input resolution by the encoder before feature extraction. For instance, an output stride of 16 means the encoder has reduced the spatial dimensions of the input by a factor of 16. By applying dilated convolutions, we can maintain a larger output stride while increasing the receptive field of the model without additional down-sampling.

This allows the model to capture larger contextual information, which is especially useful in tasks like semantic segmentation, where preserving spatial resolution is important. Dilated convolutions increase the receptive field by introducing gaps (dilation) between kernel elements, effectively expanding the area covered by each convolution without reducing the feature map resolution. As a result, we achieve a balance between high-level context and spatial detail, which is critical for accurate pixel-level predictions.

### **The mapping of each scale of the feature pyramid to the respective number of channels.**

When we uncomment the print statement, as described in the task, (1:3, 2:64, 4:64, 8:128, 16:512) is printed on a single line. This output represents the scales and channel dimensions of the feature pyramid generated by the encoder. For example, 1:3, 2:64, 4:64, 8:128, 16:512 indicates that the encoder produces feature maps at different down-sampling scales relative to the input resolution, with each scale corresponding to a specific number of channels. Lower scales (e.g., 16:512) capture high-level, contextual information with a larger receptive field, while higher scales (e.g., 1:3 and 2:64) retain finer details with higher spatial resolution. This multi-scale representation is essential for tasks requiring both context and detail, such as segmentation and depth estimation.

### **The largest scale factor in the pyramid.**

In our model’s feature pyramid, the largest scale factor corresponds to the output stride. Based on the output from the print statement (1:3, 2:64, 4:64, 8:128, 16:512), the largest scale factor is 16, meaning the output stride is 16.

### **Set dilation flags to (False, False, True) and train the model.**

See the resulting code in Code Snippet 1.

### **Does the performance improve? If so, why?**

The results indicate that the dilated model outperforms the template model and achieves similar or slightly better results than the pretrained model across most metrics, see figure: 5. In the loss comparison, it exhibits lower training and validation losses, especially in `loss_train/semseg` and `loss_val/semseg`, suggesting better learning and generalization. The score comparison shows that with dilatation in layer 4, the model scores higher on context-dependent classes like `road`, `sky`, and `vegetation`, benefiting from the larger receptive field provided by dilation. However, for smaller objects like `motorcycle` and `pole`, the pretrained model slightly outperforms the dilated model.

## Comparison of example predictions from W&B.

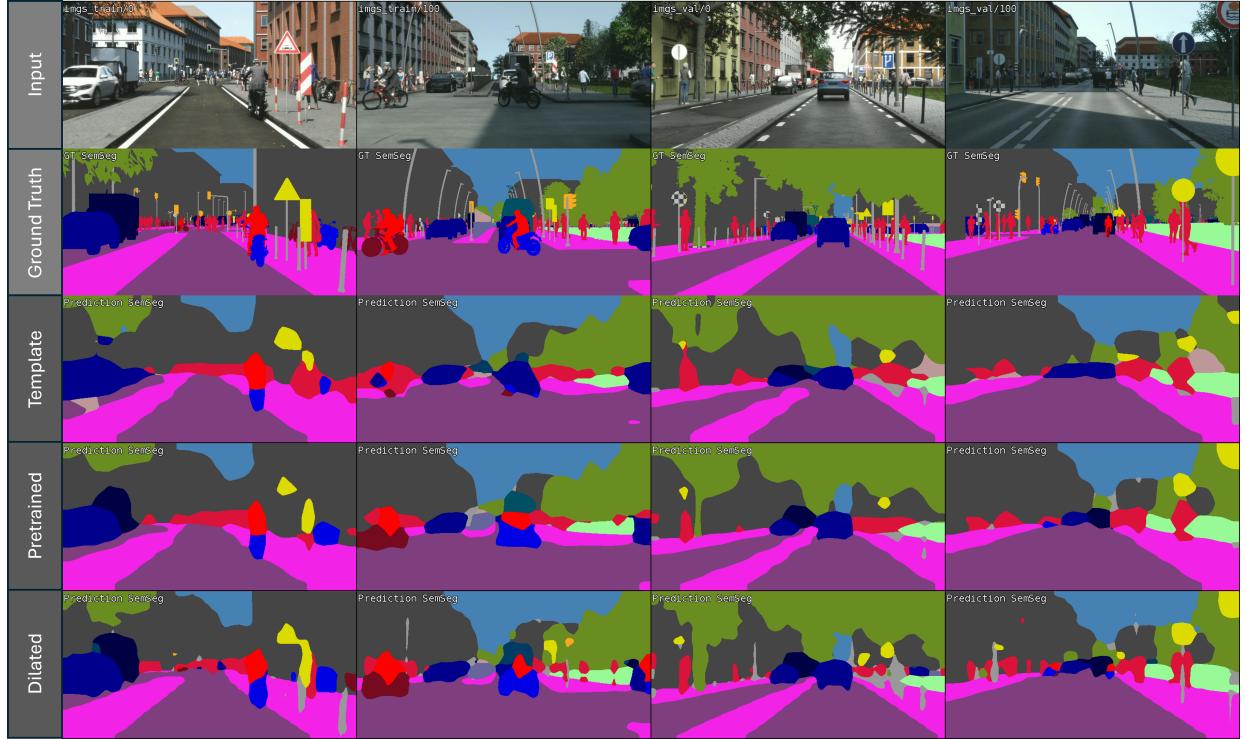


Figure 6: Prediction comparison: Not pretrained, pretrained and dilated network.

An analysis of Figure 6, with a focus on column 3, highlights that the template implementation exhibits limited semantic understanding compared to the alternative approaches. For example, in the template implementation, the stem of the tree on the left side is not detected, whereas the pretrained model successfully identifies this feature. Moreover, the pretrained model demonstrates sharper geometric precision relative to the template. Additionally, the dilated version with pretrained weights further enhances performance. Specifically, the middle cobblestone stripe and the pedestrian walkway show closer alignment to the ground truth. This improvement can be attributed to the dilated architecture of the neural network, which effectively captures geometric features defined over larger spatial regions of the image.

### 1.3 ASPP and Skip Connections for Semantic Segmentation

**Objective** The intention of adding ASPP and skip connections to the DeepLab-based semantic segmentation model is to address two key challenges in segmentation tasks: capturing multi-scale contextual information and preserving spatial details for precise object boundaries.

The Atrous Spatial Pyramid Pooling (ASPP) module enhances the model’s ability to capture contextual information at multiple scales. In complex scenes, objects can vary significantly in size and shape, making it essential for the model to recognize both large-scale structures and fine details. The ASPP module achieves this by employing parallel atrous

(or dilated) convolutions with different dilation rates. Each atrous convolution branch captures features at a unique scale, allowing the model to aggregate information from both local and global contexts simultaneously. This is particularly beneficial for semantic segmentation, where understanding the broader scene context can help correctly identify objects and segment boundaries, especially for classes with high variability or complex structures.

The skip connections are integrated in the decoder to address the loss of spatial resolution that occurs as features are downsampled during encoding. In traditional encoder-decoder architectures, high-resolution information from earlier layers is often discarded, leading to coarse segmentation outputs that may miss fine-grained details. By adding skip connections from early encoder layers to the decoder, we reintroduce these high-resolution features at multiple stages of upsampling in the decoder. Specifically, features from lower layers, which retain spatial details, are concatenated with the upsampled features in the decoder, enriching the decoder’s input with both high-level semantic information (from deeper layers) and spatial detail (from earlier layers). This approach enables the model to produce sharper segmentation boundaries and improves its performance on small or thin objects, which are often challenging for models without skip connections.

**Implementation** Our implementation involved two main architectural changes: constructing a robust ASPP module and adding skip connections to the decoder for enhanced feature fusion.

The ASPP module was designed with three parallel branches, each utilizing a dilated convolution with a different dilation rate (3, 6, and 9). These branches captured contextual information at multiple spatial scales, essential for recognizing objects of varying sizes. Additionally, a global average pooling branch was added to capture global context, which aids in understanding larger scene structures. The outputs from these branches were concatenated and refined through a final 1x1 convolution, which reduced the number of channels to the target dimension. This ASPP design, inspired by the DeepLab framework, provided the model with multi-scale contextual capabilities.

In the decoder, we introduced skip connections from the encoder’s earlier layers to retain spatial detail. We applied a 1x1 convolution to the skip features to match the dimensionality of the decoder features. After upsampling the ASPP output, we concatenated it with the processed skip features, effectively merging high-resolution spatial information with deep, multi-scale features from the ASPP module. This combination provided the decoder with rich detail for generating accurate segmentation boundaries.

Code Snippet 2: Implemented ASPP Module

```
1 class ASPP(torch.nn.Module):
2     def __init__(self, in_channels, out_channels, rates=(3, 6, 9)):
3         super().__init__()
4
5         # Define ASPP branches with dilated convolutions
6         self.branches = nn.ModuleList([
7             ASPPpart(in_channels, out_channels, kernel_size=3,
8                     stride=1, padding=rate, dilation=rate)
9         for rate in rates
10     ])
```

```
10 # 1x1 Convolution branch
11 self.branches.append(ASPPpart(in_channels, out_channels,
12     kernel_size=1, stride=1, padding=0, dilation=1))
13
14 # Image pooling branch for global context
15 self.global_avg_pool = nn.AdaptiveAvgPool2d((1, 1))
16 self.global_conv = nn.Conv2d(in_channels, out_channels,
17     kernel_size=1, stride=1)
18
19 # Final 1x1 Convolution to combine all branches
20 self.final_conv = nn.Conv2d(out_channels * (len(rates) + 2),
21     out_channels, kernel_size=1)
22
23 def forward(self, x):
24     # Apply each ASPP branch and collect outputs
25     aspp_outputs = [branch(x) for branch in self.branches]
26
27     # Apply global average pooling branch
28     global_features = self.global_avg_pool(x)
29     global_features = self.global_conv(global_features)
30     global_features = F.interpolate(global_features, size=x.
31         shape[2:], mode='bilinear', align_corners=False)
32
33     # Concatenate all branch outputs
34     aspp_outputs.append(global_features)
35     aspp_out = torch.cat(aspp_outputs, dim=1)
36
37     # Aggregate using final 1x1 convolution
38     out = self.final_conv(aspp_out)
39
40     return out
```

Code Snippet 3: Implemented Decoder with Skip Connections

```
1 class DecoderDeeplabV3p(torch.nn.Module):
2     def __init__(self, bottleneck_ch, skip_4x_ch, num_out_ch):
3         super(DecoderDeeplabV3p, self).__init__()
4
5         # 1x1 Convolution to reduce skip channels
6         self.skip_conv = nn.Conv2d(skip_4x_ch, 48, kernel_size=1,
7             stride=1)
8
9         # 3x3 Convolution for combining bottleneck and skip
10        features
11        self.concat_conv = nn.Sequential(
12            nn.Conv2d(bottleneck_ch + 48, 256, kernel_size=3,
13                  padding=1, bias=False),
14            nn.BatchNorm2d(256),
15            nn.ReLU(inplace=True),
16            nn.Conv2d(256, num_out_ch, kernel_size=1, stride=1)
```

```

14
15
16     def forward(self, features_bottleneck, features_skip_4x):
17         # Apply 1x1 convolution to reduce skip channels
18         features_skip_4x = self.skip_conv(features_skip_4x)
19
20         # Upsample bottleneck features to match skip feature
21         # dimensions
22         features_bottleneck = F.interpolate(
23             features_bottleneck, size=features_skip_4x.shape[2:], mode='bilinear', align_corners=False
24         )
25
26         # Concatenate and process combined features
27         combined_features = torch.cat([features_bottleneck, features_skip_4x], dim=1)
28         predictions = self.concat_conv(combined_features)
29
30     return predictions, combined_features

```

**Experimental Setup** We conducted an experiment with two configurations to evaluate the impact of the ASPP module and skip connections. The experiment included one run with both ASPP and skip connections enabled, and another run with these features disabled, using the default implementation from the template. For both configurations, we applied the set of hyperparameters identified as most effective in previous tasks.

**Results and Analysis** Our experimental results indicate that the addition of ASPP and skip connections led to a significant improvement in segmentation performance compared to the dilated model. After the first epoch, the dilated model achieved a mean IoU of 68.5, while our enhanced model attained 73.5, indicating a strong early performance gain from the added features. By the end of five epochs, the difference remained notable: the template model reached a mean IoU of 76.1, while our model achieved 80.5.

Analyzing class-specific IoU improvements further highlights the advantages of our approach. For example, for the "traffic light" class, our model's IoU reached 56.9, a substantial increase from the reference's 49.6. For "bicycle," our model achieved 63.0 compared to the reference's 59.8, and for "sidewalk," our model reached 91.0 compared to the original 86.0.

The validation loss curves corroborate these results. The dilated model's loss reduction curve shows a gradual improvement but stabilizes at a higher loss than our model. In contrast, the enhanced model's validation loss decreases more rapidly and stabilizes at a lower value, indicating more effective learning and generalization.

These metrics collectively demonstrate that adding ASPP and skip connections not only improved the overall IoU but also enhanced segmentation accuracy on challenging classes, such as smaller or less represented objects like traffic lights and bicycles. This confirms the efficacy of multi-scale contextual information from ASPP and the spatial detail preservation from skip connections.

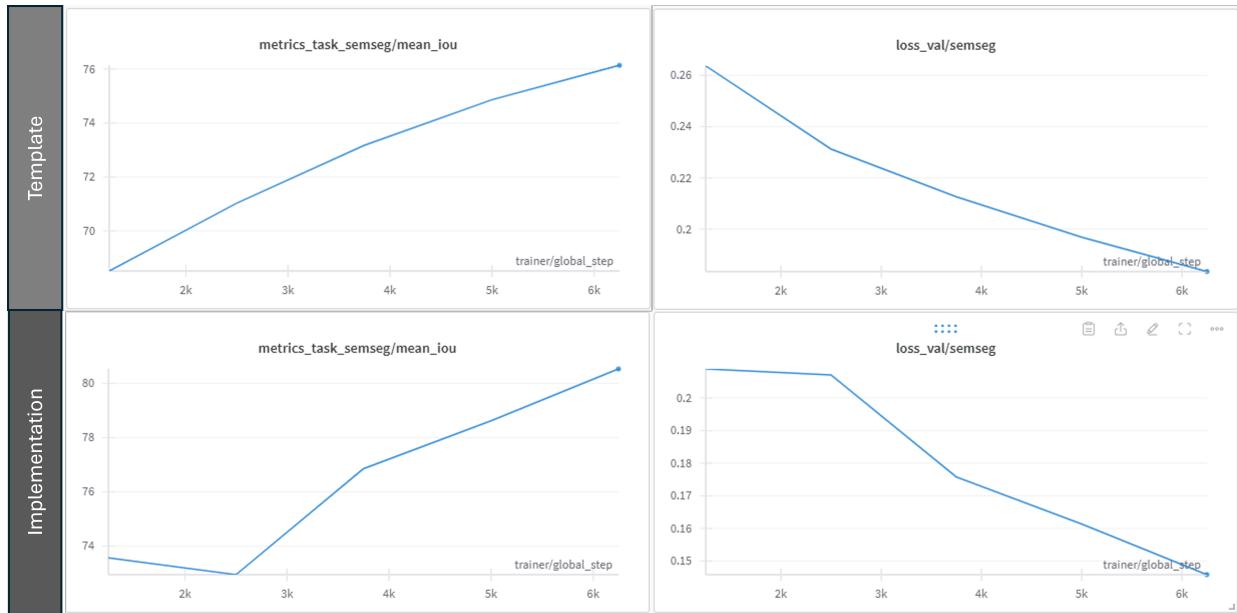


Figure 7: Performance comparison: Dilated Model vs Model with fully implemented ASPP and Skip Connections.

**Visual Evaluation and Observations** Qualitative evaluation of the segmentation outputs revealed significant improvements in capturing both large and small-scale features. The **boundary precision** was notably enhanced, with sharper transitions between classes in complex areas, such as between “road” and “sidewalk.” Additionally, the multi-scale context provided by the ASPP module improved representation for objects at varying distances, addressing a common limitation in semantic segmentation models. Large objects, like “buildings” and “sky,” were segmented more cohesively, while smaller objects, particularly pedestrians, riders, and traffic signs were segmented in much more detail.

These observations confirmed the success of the ASPP and skip connections in balancing detailed boundary accuracy with broad contextual understanding.

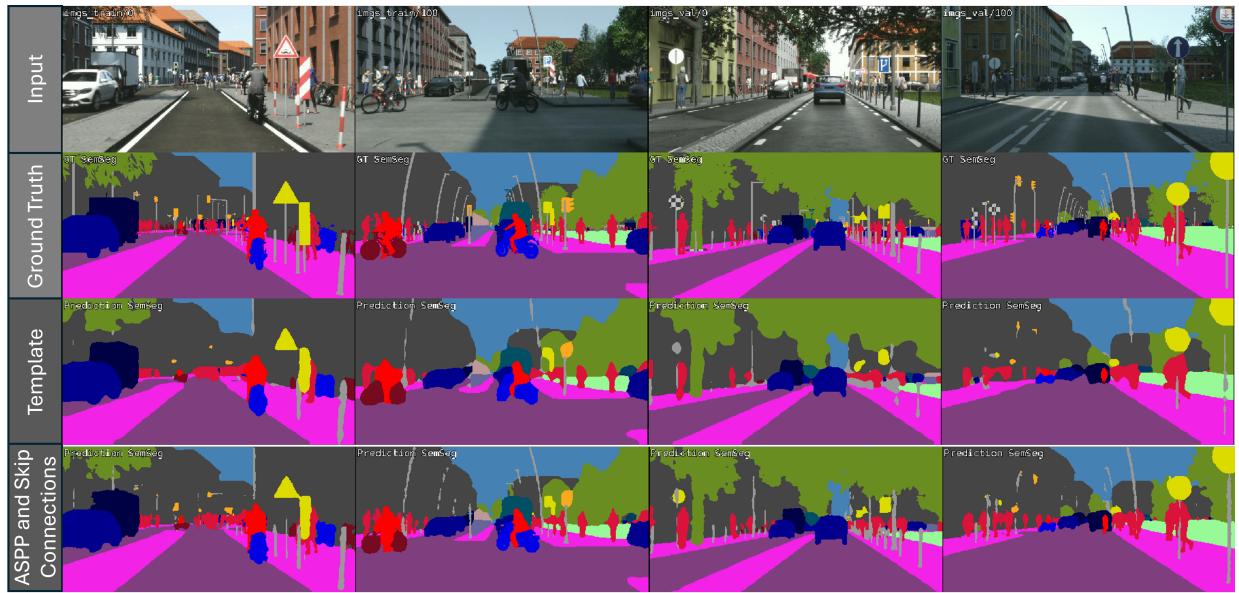


Figure 8: Example Predictions: Dilated Model vs Model with fully implemented ASPP and Skip Connections.

## 2 Depth Estimation

### 2.1 Depth Loss

#### 2.1.1 Scale-Invariant Log (SILog) loss implementation

We apply a mask to `target` to identify valid depth values (non-zero pixels), ensuring that only relevant data contributes to the final loss. For scale invariance, we log-transform `input` and `target`, making the loss sensitive to relative rather than absolute differences, with a small constant (`self.eps`) added to avoid  $\log(0)$  issues.

The difference between `log_input` and `log_target` (`log_diff`) is then used to compute the loss. Using `masked_mean_var`, we calculate the mean and variance of `log_diff`, applying the mask to focus on valid pixels.

The SILog loss is calculated by combining the variance of `log_diff` with a term that scales the squared mean of `log_diff` by `self.gamma`. This setup penalizes large deviations and enforces scale consistency. Finally, the method returns the square root of the mean loss over the batch, producing a single scalar value for training optimization. See full code in Code Snippet 4.

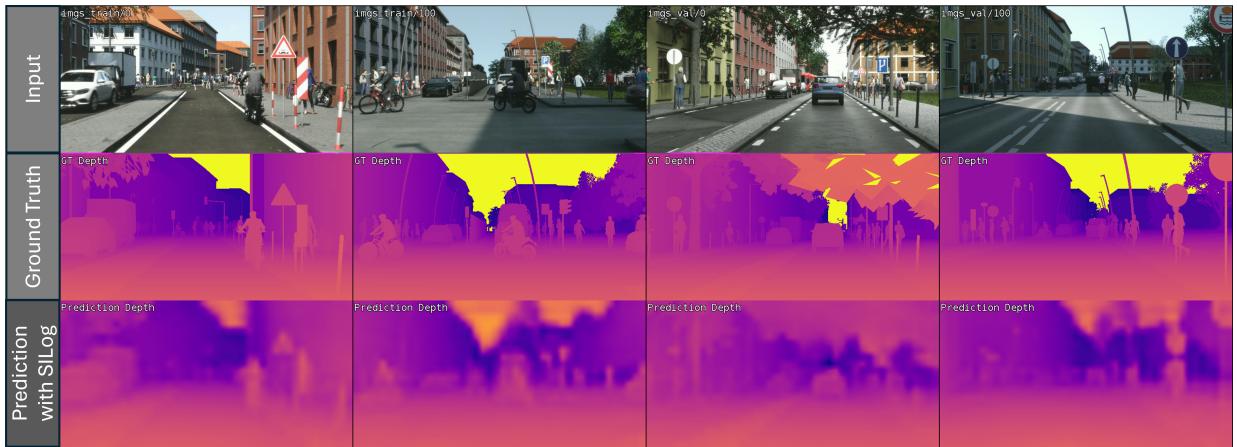


Figure 9: Example Predictions: Depth estimation with SILog implementation.

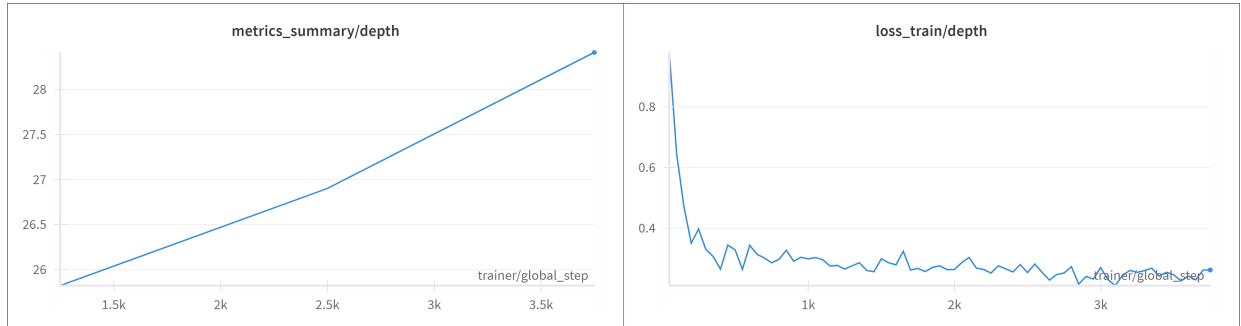


Figure 10: Depth Estimation Performance.

Code Snippet 4: SiLog implementation.

```
1 # decorator, to use half-precision (force-disabled) /
2 # Using half precision might lead to numerical instability
3 @torch.cuda.amp.autocast(enabled=False)
4 def forward(self, input: torch.Tensor, target: torch.Tensor) ->
5     torch.Tensor:
6     # Avoid dummy channel dimension
7     if input.ndim == 4:
8         input = input.squeeze(1)
9     if target.ndim == 4:
10        target = target.squeeze(1)
11
12     # TODO: Implement a proper silog loss and taking into
13     # consideration
14     # the invalid pixels (target is 0).
15
16     # mask to ignore invalid pixels, bzw. use non-zero entries.
17     # Epsilon for computational tractability
18     target_mask = (target > self.eps).float()
19
20     # Log-transform --> scale-invariance, then difference. Epsilon
21     # for computational tractability
22     log_input = torch.log(torch.clamp(input, min=self.eps))
23     log_target = torch.log(torch.clamp(target, min=self.eps))
24     log_diff = log_input - log_target
25
26     # mean and variance of log_diff using our mask
27     mean, var = masked_mean_var(log_diff, target_mask, dim=[1,
28
29     2])
30
31     # loss computation
32     loss = var + self.gamma * mean.pow(2)    # (Don't use numpy here
33     , as it breaks the autograd graph)
34
35     # Sum over the batch dimension
36     return torch.sqrt(loss.mean()) # return scalar. Sqrt as in
37     task description.
```

### 2.1.2 Invariant property of log error variance

The Scale-Invariant Log loss is designed to measure the difference between the logarithms of the ground truth depth  $y^*$  and the predicted depth  $\hat{y}$ . Specifically, the error  $\varepsilon$  is defined as:

$$\varepsilon = \log(y^*) - \log(\hat{y}) = \log\left(\frac{y^*}{\hat{y}}\right)$$

Let's consider scaling the predictions  $\hat{y}$  by a positive constant factor  $a > 0$ , leading to new predictions  $\hat{y}' = a \cdot \hat{y}$ . The new error  $\varepsilon'$  becomes:

$$\begin{aligned}\varepsilon' &= \log(y^*) - \log(\hat{y}') \\ &= \log(y^*) - \log(a \cdot \hat{y}) \\ &= \log(y^*) - [\log(a) + \log(\hat{y})] \\ &= [\log(y^*) - \log(\hat{y})] - \log(a) \\ &= \varepsilon - \log(a)\end{aligned}$$

Here,  $\log(a)$  is a constant shift introduced by scaling the predictions. The key observation is that subtracting a constant from a random variable (in this case,  $\varepsilon$ ) does not affect its variance. Therefore, the variance of the new error  $\varepsilon'$  remains the same as the original variance:

$$\text{Var}(\varepsilon') = \text{Var}(\varepsilon - \log(a)) = \text{Var}(\varepsilon)$$

The loss function effectively measures the relative error between  $y^*$  and  $\hat{y}$ , disregarding any consistent scaling in the predicted depths.

### 2.1.3 Role of the Constraint $\mathbb{E}[\varepsilon]^2 \leq \mu$

The optimization problem seeks to minimize the variance of this log error  $V(\varepsilon)$ , subject to the constraint  $\mathbb{E}[\varepsilon]^2 \leq \mu$ , where  $\mathbb{E}[\varepsilon]$  is the mean of  $\varepsilon$  and  $\mu$  is a small positive constant acting as threshold.

The constraint  $\mathbb{E}[\varepsilon]^2 \leq \mu$  is essential because minimizing the variance alone does not guarantee that the predicted depths have the correct **absolute scale**. While minimizing  $V(\varepsilon)$  reduces the variability of the errors, it does not address systematic biases where all predictions might consistently overestimate or underestimate the true depths by a constant factor. This consistent scaling error means that the model captures relative depth differences but fails to align with the true metric scale of the scene.

By enforcing the constraint on the squared mean log error, we ensure that the average discrepancy between predicted and actual depths remains within an acceptable threshold. The term  $\mathbb{E}[\varepsilon]$  represents the average scaling difference; controlling it prevents the model from developing a systematic bias in depth estimation.

## 2.2 Task weighting

In multi-task learning, the total loss is a weighted sum of the individual task losses:

$$\text{Total Loss} = w_1 \cdot \text{Loss}_1 + w_2 \cdot \text{Loss}_2,$$

where  $w_1$  and  $w_2$  are the weights for each task.

Increasing the loss weight for one task emphasizes it during training, causing the model to focus more on minimizing that task's error. Decreasing the weight for the other task reduces its influence, potentially leading to poorer performance on that task due to less contribution to the total loss and gradient updates.

Task weights should be chosen to balance the learning of all tasks effectively. They should prevent any single task from dominating the training due to larger loss magnitudes and reflect the relative importance of each task for the application. Adaptive weighting strategies can also be employed to adjust weights during training based on factors like task difficulty or convergence.

### 3 Multitask Learning

In this task, we explored the implementation of a branched architecture for multi-task learning (MTL) to address the challenges of semantic segmentation and monocular depth estimation. The primary objective was to compare the branched model with the joint architecture and evaluate its performance in terms of segmentation quality, depth accuracy, GPU memory consumption, number of parameters, and training time.

We began by implementing a straightforward branched architecture, where task-specific ASPP modules and decoders were added for each task. Following this, we introduced more specialized configurations for each task to further enhance performance, including separate encoders, fine-tuned ASPP parameters, and tailored decoders. These steps allowed us to systematically evaluate the impact of task-specific architectural elements on performance and efficiency.

#### 3.1 Initial Implementation: Basic Branched Model

The first implementation maintained a shared encoder for both tasks while introducing separate ASPP modules and decoders for semantic segmentation and depth estimation. This design retained simplicity, allowing us to assess the value of even modest task-specific specialization without significantly increasing model complexity. The ASPP modules and decoders for both tasks were identical, ensuring that any observed differences in performance could be attributed to the separation of responsibilities rather than architectural nuances.

The basic branched model performed slightly better than the non-branched architecture, particularly for semantic segmentation. The introduction of task-specific ASPP modules allowed each task to process features from the encoder in a way more suited to its needs, yielding improved segmentation accuracy. However, the improvement was relatively modest for the depth estimation task, suggesting that shared encoder features may still have been suboptimal for both tasks.

In terms of computational requirements, the basic branched model mirrored the non-branched implementation, requiring approximately 10–15 minutes per epoch during training and utilizing less than 40% of the GPU memory for most of the training process. The similarity in resource consumption is expected, given the minimal changes in architecture beyond the task-specific ASPP and decoders. Below is the code for this implementation:

Code Snippet 5: Basic Branched Architecture Implementation.

```
1 class ModelDeepLabV3PlusMultiTask(nn.Module):
2     def __init__(self, cfg, outputs_desc):
3         super().__init__()
4         self.outputs_desc = outputs_desc
5
6         # Initialize encoder
7         self.encoder = Encoder(
8             cfg.model_encoder_name,
9             pretrained=cfg.pretrained,
10            zero_init_residual=True,
```

```

12         replace_stride_with_dilation=(False, False, True)
13     )
14
15     # Get encoder channel dimensions
16     ch_out_encoder_bottleneck, ch_out_encoder_4x =
17         get_encoder_channel_counts(cfg.model_encoder_name)
18
19     # Task-specific ASPP and Decoder for each output type
20     self.aspps = nn.ModuleDict({
21         task: ASPP(ch_out_encoder_bottleneck, 256)
22         for task in outputs_desc
23     })
24
25     self.decoders = nn.ModuleDict({
26         task: DecoderDeeplabV3p(256, ch_out_encoder_4x, num_ch)
27         for task, num_ch in outputs_desc.items()
28     })
29
30     def forward(self, x):
31         input_resolution = (x.shape[2], x.shape[3])
32         features = self.encoder(x)
33         lowest_scale = max(features.keys())
34         features_lowest = features[lowest_scale]
35
36         out = {}
37
38         for task, num_ch in self.outputs_desc.items():
39             features_aspp = self.aspps[task](features_lowest)
40             predictions_4x, _ = self.decoders[task](features_aspp,
41                 features[4])
42             predictions_1x = F.interpolate(predictions_4x, size=
43                 input_resolution, mode='bilinear', align_corners=
44                 False)
45             out[task] = predictions_1x.exp().clamp(0.1, 300.0) if
46                 task == "depth" else predictions_1x
47
48         return out

```

## 3.2 Enhanced Branched Architecture: Task-Specific Modules

To push the limits of what the branched architecture could achieve, we introduced a more sophisticated implementation featuring task-specific encoders, ASPP modules, and decoders. Each task was given its own encoder, tailored to its specific requirements. For semantic segmentation, the encoder’s configuration emphasized a larger receptive field by enabling dilation earlier in the network, enabling better context capture. For depth estimation, the encoder preserved spatial resolution by delaying dilation to deeper layers, prioritizing the retention of fine-grained details.

The ASPP modules were also customized for each task. Semantic segmentation received

larger dilation rates to emphasize broader context, while depth estimation utilized smaller dilation rates to focus on finer details. Similarly, the decoders were designed with task-specific considerations, with the depth decoder featuring additional convolutional layers to enhance spatial detail recovery.

The following code illustrates the enhanced branched architecture:

Code Snippet 6: Enhanced Branched Architecture with Task-Specific Modules.

```

1 import torch
2 import torch.nn.functional as F
3 from torch import nn
4
5 from source.models.model_parts import Encoder,
6     get_encoder_channel_counts, ASPP, DecoderDeeplabV3p
7
8 class ModelDeepLabV3PlusMultiTask(nn.Module):
9     def __init__(self, cfg, outputs_desc):
10         super().__init__()
11         self.outputs_desc = outputs_desc
12
13         # Task-specific configurations for encoders
14         encoder_configs = {
15             'semseg': {
16                 'replace_stride_with_dilation': (False, True, True)
17
18                 ,
19                 'pretrained': cfg.pretrained,
20                 'zero_init_residual': True
21             },
22             'depth': {
23                 'replace_stride_with_dilation': (False, False, True)
24                 ,
25                 'pretrained': cfg.pretrained,
26                 'zero_init_residual': True
27             }
28         }
29
30         # Initialize separate encoders for each task
31         self.encoders = nn.ModuleDict({
32             task: Encoder(
33                 cfg.model_encoder_name,
34                 **encoder_configs[task]
35             ) for task in outputs_desc
36         })
37
38         # Task-specific ASPP configurations
39         aspp_configs = {
40             'semseg': {
41                 'rates': (6, 12, 18, 24),
42                 'out_channels': 256
43             },
44             'depth': {
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
617
618
619
619
620
621
622
623
624
625
626
627
627
628
629
629
630
631
632
633
634
635
635
636
637
637
638
639
639
640
641
642
642
643
643
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
```

```

41         'rates': (3, 6, 9),
42         'out_channels': 256
43     }
44 }
45
46 self.aspps = nn.ModuleDict({
47     task: ASPP(
48         ch_out_encoder_bottleneck,
49         aspp_configs[task]['out_channels'],
50         rates=aspp_configs[task]['rates']
51     ) for task in outputs_desc
52 })
53
54 self.decoders = nn.ModuleDict({
55     task: DecoderDeeplabV3p(
56         aspp_configs[task]['out_channels'],
57         ch_out_encoder_4x,
58         num_ch
59     ) for task, num_ch in outputs_desc.items()
60 })
61
62 def forward(self, x):
63     input_resolution = x.shape[2:]
64     out = {}
65     for task, num_ch in self.outputs_desc.items():
66         features = self.encoders[task](x)
67         features_aspp = self.aspps[task](features[highest_scale]
68             [])
69         predictions_1x = F.interpolate(
70             self.decoders[task](features_aspp, features[4])[0],
71             size=input_resolution,
72             mode='bilinear',
73             align_corners=False
74         )
75         out[task] = predictions_1x
76
77 return out

```

This enhanced architecture showed clear performance improvements for both tasks, especially after three epochs of training. However, the increased complexity resulted in significantly longer training times (35–50 minutes per epoch) and higher GPU memory consumption (approximately 60%). The trade-off between performance and computational cost highlights the benefits of task-specific specialization but also underscores the need for efficiency in resource-constrained scenarios. Additionally, the heavy computational requirements limited the amount of different configurations that could be tested, as it would take several hours to see whether small changes were successful or not.

**Network Parameters Analysis.** The number of trainable parameters provides insight into the computational complexity and capacity of each model. For the non-branched architecture, which employs a shared encoder, ASPP module, and decoder, the total pa-

parameter count is approximately 16.1 million. This relatively lightweight architecture is efficient and enables faster training. The branched architecture with a shared encoder, however, includes task-specific ASPP modules and decoders, increasing the total parameter count to 20.6 million. This additional capacity comes from the specialization in the task-specific modules, which improve performance, particularly for segmentation. Finally, the fully branched model, which employs separate encoders, ASPP modules, and decoders for each task, significantly increases the parameter count to 32.3 million. This architecture provides the highest potential for task-specific optimization but incurs a substantial cost in terms of computational resources and memory usage. These differences in parameter counts align with the observed trade-offs between performance and resource requirements across the models.

**Improved Results and Trade-offs.** The improved results achieved by the enhanced architecture can be attributed to its greater degree of task-specific adaptation, which allowed for more effective handling of the unique requirements of semantic segmentation and depth estimation. For semantic segmentation, the model benefited from the inclusion of a larger receptive field, made possible by using higher dilation rates in the Atrous Spatial Pyramid Pooling (ASPP) module. This enhancement enabled the model to capture broader contextual information, crucial for distinguishing between complex or overlapping objects in the scene. Meanwhile, the depth estimation task leveraged the preservation of spatial resolution, achieved through modifications to the encoder, and the inclusion of additional convolutional layers in the decoder. These changes improved the model’s ability to retain fine-grained spatial details, leading to more accurate depth predictions, particularly in regions with abrupt depth changes or intricate structures.

However, the increased resource demands of the enhanced architecture highlight the trade-offs involved in achieving such specialization. The addition of task-specific modules and the use of separate encoders for each task significantly increased the computational complexity, as reflected in the longer training times and higher memory usage. These resource requirements emphasize the need to balance architectural complexity with practical considerations, particularly in resource-constrained environments. Despite this, the improvements in performance demonstrate that such task-specific enhancements are effective in addressing the unique challenges posed by each task in multitask learning.

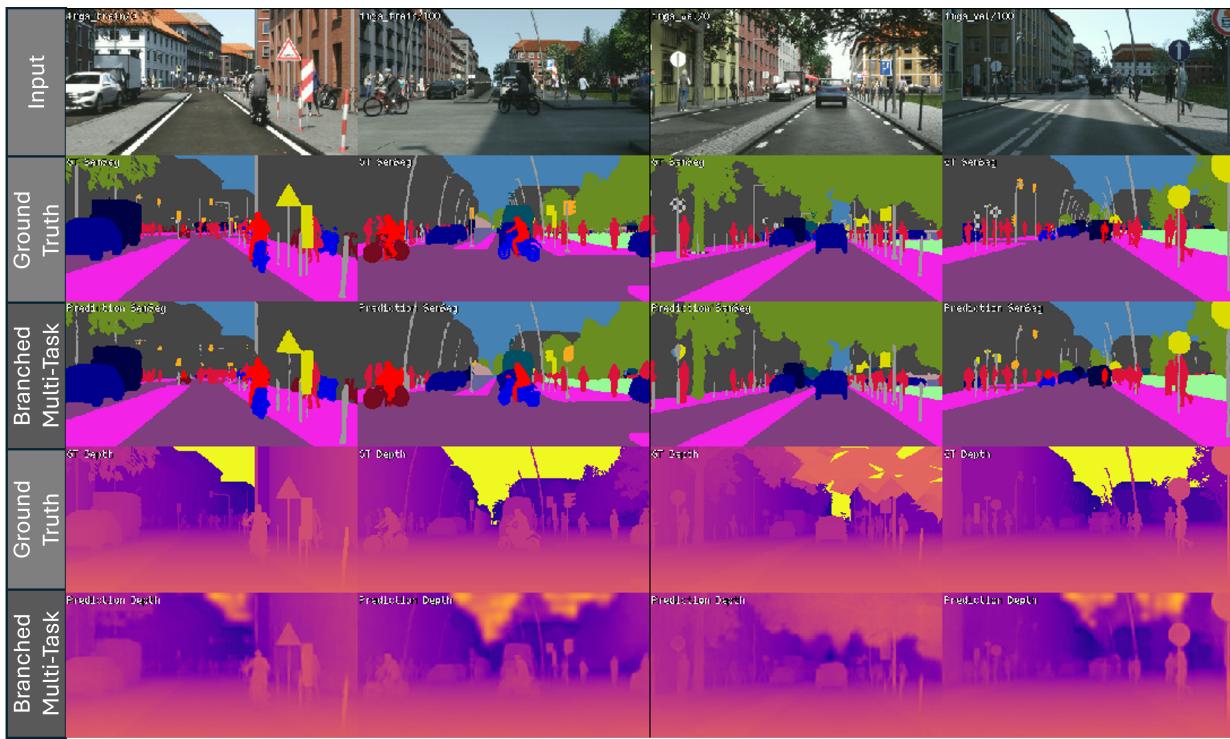


Figure 11: Example Predictions: Branched Multitask Model.

## 4 Adaptive Depth Estimation

### 4.1 Attention

#### a) Implementation

For the implementation of the attention mechanism we first define 3 projections to later obtain the query, key and value matrix. We also define a LayerNorm operation.

Code Snippet 7: Query, key and value projections and a layernorm.

```
1 self.proj_q = nn.Linear(dim, dim)
2 self.proj_k = nn.Linear(dim, dim)
3 self.proj_v = nn.Linear(dim, dim)
4 self.norm = nn.LayerNorm(dim)
```

In the forward method we first apply the layer norm and then project the input onto the query, key and value matrices. A positional encoding is added to the query vectors if present. To incorporate multiple attention heads we reshape the matrices by unpacking the C channels among the heads. We then compute the similarity and take the softmax to obtain the attention weights. A final product with the value matrix together with a reshape back to (B, N, C) gives us the final output.

Code Snippet 8: Attention mechanism.

```
1 def forward(self, x, pos_embed):
2
3     B, N, C = x.shape
4
5     x = self.norm(x)
6     q = self.proj_q(x)
7     k = self.proj_k(x)
8     v = self.proj_v(x)
9
10    if pos_embed is not None:
11        q += pos_embed
12
13    q = q.reshape(B, N, self.num_heads, C // self.num_heads).
14        transpose(1, 2)
15    k = k.reshape(B, N, self.num_heads, C // self.num_heads).
16        transpose(1, 2)
17    v = v.reshape(B, N, self.num_heads, C // self.num_heads).
18        transpose(1, 2)
19
20    s = q.matmul(k.transpose(2, 3)) / ((C // self.temperature)
21        **0.5)
22    weights = torch.softmax(s, dim=3)
23    weights = self.dropout(weights)
24
25    a = weights.matmul(v)
26    a = a.transpose(1, 2).reshape(B, N, C)
```

```

24     x = self.out(a)
25     return x

```

## b) Theoretical Questions

### b.i)

The attention mechanism computes the query ( $Q$ ), key ( $K$ ), and value ( $V$ ) matrix. This is done by projecting the input matrix  $X \in \mathbb{R}^{N \times D}$ :  $Q = XW^Q$ ,  $K = XW^K$ ,  $V = XW^V$ . Here  $W^Q \in \mathbb{R}^{D \times D}$ ,  $W^K \in \mathbb{R}^{D \times D}$ ,  $W^V \in \mathbb{R}^{D \times D}$  are the weight matrices (Given the implementation, we can say that  $C = D$ ). These matrix multiplications are in the order of  $O(B \cdot N \cdot D^2)$ .

The addition of heads does not change the complexity, since each of the  $H$  heads processes  $1/H$  of the computations.

The attention weights are computed by the scaled dot-product  $QK^T$   $Q \in \mathbb{R}^{N \times D}$ ,  $K \in \mathbb{R}^{N \times D}$ . Therefore, the complexity for the attention weights is  $O(B \cdot N^2 \cdot D)$ . The attention weights are then multiplied with the value matrix  $V \in \mathbb{R}^{N \times D}$  to get the output. This is done in the order of  $O(B \cdot N \cdot D^2)$ . The softmax and layernorm are both of lower order.

Therefore, the overall complexity is  $O(B \cdot N \cdot D^2 + B \cdot N^2 \cdot D)$ .

### b.ii)

By default the attention mechanism has a memory requirement of  $O(N^2)$  with  $N$  number of tokens. To alleviate the high memory requirements induced by the attention mechanism 3 solutions have been presented in class:

1. Local attention: This method limits the attention to a small window around each query which in turn reduces the memory requirements.  $O(N \times W)$ ,  $W$  window size
2. Approximate attention: This doesn't compute the full attention matrix but tries to approximate it using low-rank approximations.  $O(N \times C)$ ,  $C$  approximation rank
3. Sparse attention: This method only pays attention to a subset of tokens. This is more flexible than local attention since the subset doesn't have to be local.  $O(N \times S)$ ,  $S$  sparsity level

### b.iii)

The nature of the attention mechanism requires full pairwise interactions ( $O(N^2)$  operations), a non-linear softmax and dot products. This makes it hard to factorize without loosing expressiveness.

## 4.2 Adaptive Depth

First of all we define the different modules that are needed to achieve dynamic depth bins. This includes the LatentsExtractor and 3 transformer layers. Further we add a Linear layer that projects the latents down from ch\_out\_encoder\_bottleneck to the size of the queries vector which is 256. This is needed to align the dimensions for the queries and latents vectors to compute the dot product later on. Finally we add a projection that maps the latents to the depth values by reducing the number of features to 1.

Code Snippet 9: adaptive depth network: layer definitions.

```

1  self.latents_extractor = LatentsExtractor(num_latents=cfg.
2      num_bins)
2  self.transformer_layers = nn.ModuleList([TransformerBlock(dim=
3      ch_out_encoder_bottleneck, num_heads=cfg.num_heads,
4      expansion=cfg.expansion)for _ in range(cfg.
5      num_transformer_layers)])
3  self.latents_proj = nn.Linear(ch_out_encoder_bottleneck, 256)
4  self.depth_value_proj = nn.Linear(ch_out_encoder_bottleneck, 1)
```

In the forward method of the adaptive depth module we first initialize the latents using the latents\_extractor. We then process the latents with 3 transformer layers. The depth\_value\_projection then gives us the division of the bins. Using the queries we then compute the similarity between the queries and latents. Taking the softmax then gives us the depth-bin-probabilities for each input. Using the dynamically computed depth\_values we can then obtain a depth map. Upsampling gives the final output.

Code Snippet 10: Adaptive layer network: forward.

```

1  latents = self.latents_extractor(features_lowest)
2
3  for layer in self.transformer_layers:
4      latents = layer(latents)
5
6  depth_values = self.depth_value_proj(latents).unsqueeze(-1)
7
8  latents = self.latents_proj(latents)
9
10 similarity = torch.einsum("bcij,bnc->bnij", queries, latents)
11
12 probabilities = F.softmax(similarity, dim=1)
13 predictions_4x = (probabilities * depth_values).sum(dim=1,
14     keepdim=True)
15
16 predictions_1x = F.interpolate(predictions_4x, size=
17     input_resolution, mode='bilinear', align_corners=False)
```

After one epoch of training we achieve a SI\_log\_RMSE of 27.69 and after 5 epochs a SI\_log\_RMSE of 21.38.

## 5 Open Challenge

### 5.1 Test Time Augmentation

We experimented with test time augmentation methods. To that effect we changed the definition of the inference\_step method to evaluate the depth and segmentation on several different sizes of the image and then aggregate them. We tried different scales and different aggregation methods. The relevant code looks like this:

Code Snippet 11: Test time augmentation with max.

```
1 def inference_step(self, batch):
2
3     ...
4
5     scales = [0.5, 0.75, 1.0, 1.25, 1.5]
6
7     ...
8
9     if aggregated_y_hat_semseg is None:
10         aggregated_y_hat_semseg = y_hat_semseg_resized
11         aggregated_y_hat_depth = y_hat_depth_resized
12     else:
13         aggregated_y_hat_semseg = torch.max(aggregated_y_hat_semseg,
14                                             y_hat_semseg_resized)
14         aggregated_y_hat_depth = torch.max(aggregated_y_hat_depth,
15                                             y_hat_depth_resized)
15
16     y_hat_semseg_lbl = aggregated_y_hat_semseg.argmax(dim=1)
17
18     ...
```

For scales we tested with 5 or 3 different scales and spacing the the scales differently. Firstly all we tested different numbers of scale values, i.e. scales = [0.75, 1, 1.25] and scales = [0.5, 0.75, 1, 1.25, 1.5]. Further we tested both average (Snippet 12) and max aggregation (Snippet 11) methods. We trained the model for 1 epoch.

Our baseline for the experiments had following metric scores: Total: 23.85, Semseg: 23.14, Depth: 24.57 . The results show no significant improvements for the semseg and depth metrics aswell as the total metric (Tables 2, 3, 1). What we can observe is that especially for the depth prediction task max aggregation performs significantly worse. This suggests that averaging is more suited to the task. Possible reasons for the lack of improvement could be an insufficiently explored parameter space for the scales or further improvements could only show up for models that are trained for more than 1 epoch.

Code Snippet 12: Test time augmentation with avg.

```
1 ...
2
3     if aggregated_y_hat_semseg is None:
4         aggregated_y_hat_semseg = y_hat_semseg_resized
```

```

5     aggregated_y_hat_depth = y_hat_depth_resized
6 else:
7     aggregated_y_hat_semseg += y_hat_semseg_resized
8     aggregated_y_hat_depth += y_hat_depth_resized
9
10    aggregated_y_hat_semseg /= len(scales)
11    aggregated_y_hat_depth /= len(scales)
12
13    ...

```

Scale/Method	avg	max
<b>3 Scales</b>	24.15 Avg	20.66
<b>5 Scales</b>	23.19	22.99

Table 1: Total metric: Comparing different scales and aggregation methods, Baseline: 23.85

Scale/Method	avg	max
<b>3 Scales</b>	22.20	21.37
<b>5 Scales</b>	21.57	23.15

Table 2: Semseg metric: Comparing different scales and aggregation methods, Baseline: 23.14

Scale/Method	avg	max
<b>3 Scales</b>	26.08	19.95
<b>5 Scales</b>	24.80	22.83

Table 3: Depth metric: Comparing different scales and aggregation methods, Baseline: 24.57

## 5.2 Additional Skip Connections and Attention Mechanisms for Multitask Learning

As a second approach to improve on the performance of the Multi-Task Model, we focus on enhancing the skip connection framework. Our goal was to build upon the existing branched architecture by introducing a more sophisticated mechanism for multi scale feature integration, aimed at improving performance while maintaining computational efficiency as much as possible.

The DeepLab-based architecture traditionally incorporates a single skip connection from the encoder at the 4x scale to supply high-resolution spatial features to the decoder. While effective, this approach potentially underutilizes the rich feature hierarchy encoded at other scales. Our hypothesis was that leveraging additional skip connections from the 8x and 16x scales could further improve the model’s ability to integrate fine-grained details with broader contextual information. Each scale in the encoder captures features at a unique level of abstraction, and incorporating these into the decoder has the potential to refine predictions more effectively for both tasks.

To achieve this, we designed an attention-based fusion mechanism to dynamically integrate features from multiple scales. This module allowed the model to weigh the importance of each skip connection based on the task and input data. For instance, finer details from the 4x scale might be prioritized for depth estimation, while broader contextual features from the 16x scale could be more valuable for semantic segmentation. The attention mechanism learned to aggregate features in a data-driven manner, ensuring that the most relevant information was utilized for each task.

Code Snippet 13: Attention-Based Skip Fusion for Multiscale Integration.

```
1 class AttentionFusion(nn.Module):
2     def __init__(self, skip_scales):
3         super().__init__()
4         self.attention_layers = nn.ModuleDict({
5             scale: nn.Sequential(
6                 nn.Conv2d(channels, 1, kernel_size=1),
7                 nn.Sigmoid()
8             ) for scale, channels in skip_scales.items()
9         })
10
11    def forward(self, skip_features):
12        # Compute attention weights for each scale
13        attention_weights = {scale: self.attention_layers[scale](
14            feat) for scale, feat in skip_features.items()}
15
16        # Apply attention weights to the skip features
17        weighted_features = {scale: attention_weights[scale] * feat
18            for scale, feat in skip_features.items()}
19
19        # Concatenate all attended features
20        return torch.cat(list(weighted_features.values()), dim=1)
```

The multiscale skip connection framework enabled the model to extract features from 4x, 8x, and 16x scales, fusing them through the attention module before feeding them into task-specific decoders. This approach allowed the decoder to process a richer and more diverse set of features, which we hypothesized would enhance its ability to refine outputs for both tasks.

In our experiments, the addition of multiscale skip connections and attention-based fusion resulted in a slight improvement in the performance metrics for depth estimation, particularly in capturing finer details, and achieving a si-log-rmse of 18.33 after 5 epochs, compared to the baseline of 19.88. However, there was no major change in performance for the semantic segmentation task. The increased skip connections also introduced additional resource demands, with GPU memory usage increasing by approximately 10 percentage points compared to the baseline model. Training time per epoch was also slightly longer, but not prohibitively so. Despite these changes, the improvements were not as substantial as we had anticipated.

Several factors could explain the limited performance gains. First, while the added skip connections provided more multiscale features, the complexity of integrating these features effectively may not have been fully addressed by the current attention-based fusion module. The design of this fusion mechanism might not have been optimal for balancing the contributions of features at different scales, potentially leading to suboptimal use of the additional information. Moreover, the added complexity might have inadvertently introduced noise or redundancy into the decoding process, diluting the benefits of the richer feature set.

If more time had been available, several improvements could have been explored. One possibility would have been to refine the attention mechanism for feature fusion, potentially adopting a more advanced design such as a transformer-based module that could better model the relationships between features at different scales. Another avenue could have been to experiment with selective use of skip connections, incorporating only the most relevant scales for each task to reduce redundancy and noise. Additionally, further hyperparameter optimization, including learning rates and regularization strategies, might have helped the model achieve better performance.

## Related Works

The approach of incorporating multiscale skip connections to enhance feature integration in multitask learning aligns with trends observed in several notable works in the field. A key example is the UNet++ architecture, which redefines skip connections by introducing dense connections between encoder and decoder sub-networks at varying levels. This methodology aggregates multiscale features more effectively, facilitating a flexible and precise feature fusion scheme. Like our approach, UNet++ emphasizes capturing both local details and global context, albeit primarily for segmentation tasks. However, unlike our work, UNet++ does not explicitly target multitask learning scenarios involving depth estimation, focusing solely on single-task improvements in segmentation performance.

Another closely related work is UCTransNet, which integrates transformer modules into the skip connection paradigm. This model addresses the semantic gap between encoder and decoder features by employing attention mechanisms to better align and fuse features

at different scales. The use of transformers for multiscale fusion is a key difference from our approach, where we utilized a more straightforward attention-based mechanism. While UCTransNet demonstrated marked improvements in segmentation tasks, its reliance on transformer modules comes with a higher computational cost, a trade-off we sought to avoid in our design.

Both architectures underscore the importance of multiscale feature integration, yet they differ from our work in critical ways. Our model uniquely focuses on the dual-task setting of semantic segmentation and depth estimation, leveraging multiscale skip connections specifically tailored to enhance the performance of both tasks. Additionally, our attention-based fusion mechanism, while simpler than the transformer-based approaches seen in UCTransNet, aimed to maintain computational efficiency. However, the limited gains in performance observed in our experiments suggest that these more sophisticated fusion strategies might indeed be necessary to fully exploit the potential of multiscale skip connections in a multitask context.

These related works highlight the potential and challenges of utilizing multiscale information through advanced skip connection designs. While our approach shared similarities in its reliance on multiscale features, the differences in fusion strategies and task-specific adaptations point to opportunities for further refinement and optimization in future studies.

## APPENDIX: Performance

Step size = 0.0012 $\overline{5+1=0.0002}$

Task	W&B Name	W&B Notes	Validation Multitask Metric	Validation SemSeg Metric	Validation Depth Metric	Validation Grader Multitask Metric	Validation Grader SemSeg Metric	Validation Grader Depth Metric
1.2 a	T08_1102-1404_Default_d25c7	P1.2a Pretrained	N/A	12.90537	N/A	0	62.8632	0
1.2 b	T08_1105-1814_Thor_P1.2b_dilatedConv_45601	P1.2b Dilated	N/A	18.01743	N/A	0	68.234	0
1.3	T08_1113-0236_YES-ASPP-f21b7	Problem 1.3 - ASPP Module Active	30.5	80.53	N/A	0	80.66	0
2	T08_1114-0119_Thor_P2.1a_SILog_V3.3E_f04e6	P2.1 SILog	N/A	24.15515	28.41326	41.53447	68.39174	26.85726
3	T08_1115-0614_task3_20_halfHR_alphaic	Problem 3 - Multitask active - Reduced LR and larger batch size - best leaderboard result	33.6	83.2	16.03	67.42	83.39	15.97
4	T08_1113-1928_adaptive_depth.test_6aa05	Problem 4.2 - 5 epochs - best run	N/A	N/A	28.61188	0	0	21.11205

Table 4: Performance sheet.