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.

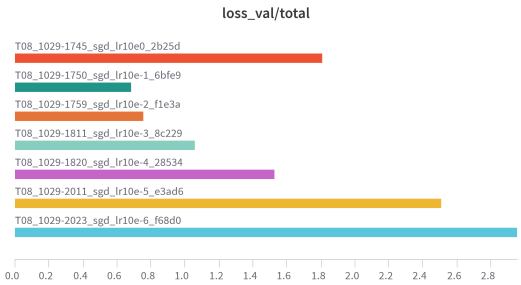

Figure 1: SGD with different learning rates. Minimum: 0.68 with lr = 10e-1.

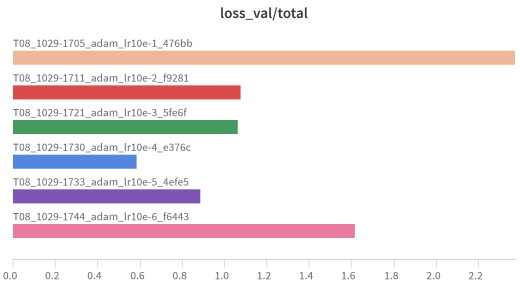

Figure 2: Adam with different learning rates. Minimum: 0.58 with lr = 10e-4.

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 disproportionally larger effect on training time than increasing the batch size. Therefore we kept a batch size of 16 for later experiments as a tradeoff.

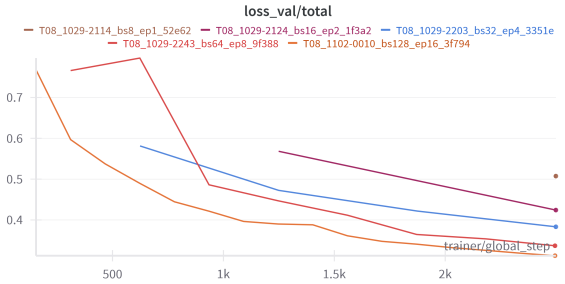

Figure 3: Validation loss for different batch sizes with proportionally adjusted number of epochs.

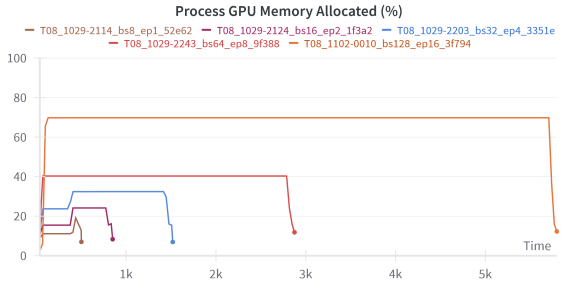

Figure 4: Memory usage for different batch sizes.

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.

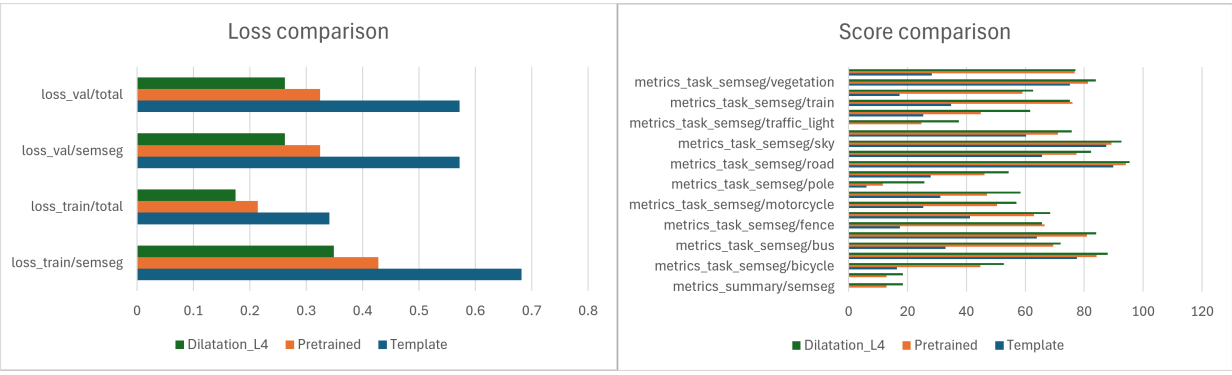

Figure 5: Performance comparison: Not pretrained, pretrained and dilated network.

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, False)`, reaffirming that dilated convolutions are indeed turned off by default.

Code Snippet 1: Toggle dilated convolutions.

```
class ModelDeepLabV3Plus(torch.nn.Module):
    def __init__(self, cfg, outputs_desc):
        super().__init__()
        self.outputs_desc = outputs_desc
        ch_out = sum(outputs_desc.values())

        self.encoder = Encoder(
            cfg.model_encoder_name,
            pretrained=cfg.pretrained,
            zero_init_residual=True,
            replace_stride_with_dilation=(False, False, True), #
                default: F,F,F
        )
```

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.

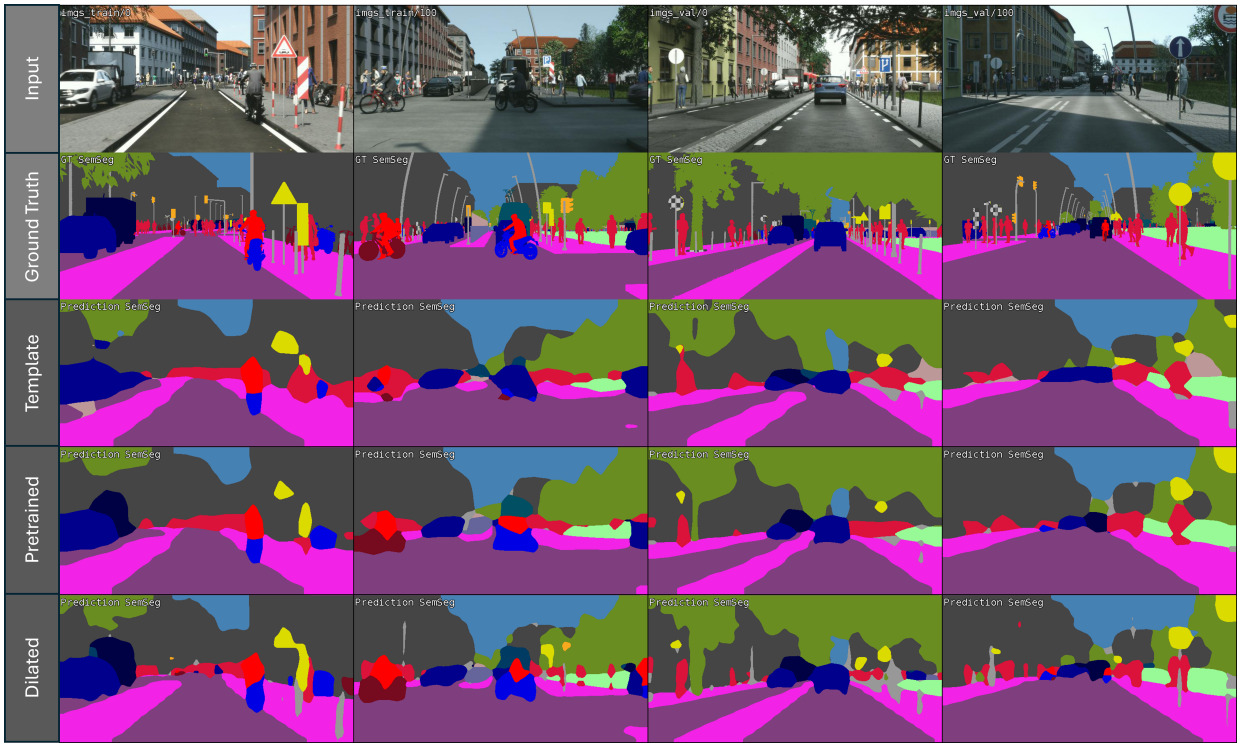

Figure 6: Prediction comparison: Not pretrained, pretrained and dilated network.

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

```
class ASPP(torch.nn.Module):
    def __init__(self, in_channels, out_channels, rates=(3, 6, 9)):
        super().__init__()

        # Define ASPP branches with dilated convolutions
        self.branches = nn.ModuleList([
            ASPPpart(in_channels, out_channels, kernel_size=3,
                stride=1, padding=rate, dilation=rate)
            for rate in rates
        ])
```

```
        # 1x1 Convolution branch
        self.branches.append(ASPPpart(in_channels, out_channels,
            kernel_size=1, stride=1, padding=0, dilation=1))

        # Image pooling branch for global context
        self.global_avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.global_conv = nn.Conv2d(in_channels, out_channels,
            kernel_size=1, stride=1)

        # Final 1x1 Convolution to combine all branches
        self.final_conv = nn.Conv2d(out_channels * (len(rates) + 2)
            , out_channels, kernel_size=1)

    def forward(self, x):
        # Apply each ASPP branch and collect outputs
        aspp_outputs = [branch(x) for branch in self.branches]

        # Apply global average pooling branch
        global_features = self.global_avg_pool(x)
        global_features = self.global_conv(global_features)
        global_features = F.interpolate(global_features, size=x.
            shape[2:], mode='bilinear', align_corners=False)

        # Concatenate all branch outputs
        aspp_outputs.append(global_features)
        aspp_out = torch.cat(aspp_outputs, dim=1)

        # Aggregate using final 1x1 convolution
        out = self.final_conv(aspp_out)
        return out
```

Code Snippet 3: Implemented Decoder with Skip Connections

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

        # 1x1 Convolution to reduce skip channels
        self.skip_conv = nn.Conv2d(skip_4x_ch, 48, kernel_size=1,
            stride=1)

        # 3x3 Convolution for combining bottleneck and skip
            features
        self.concat_conv = nn.Sequential(
            nn.Conv2d(bottleneck_ch + 48, 256, kernel_size=3,
                padding=1, bias=False),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, num_out_ch, kernel_size=1, stride=1)
```

```
        )

    def forward(self, features_bottleneck, features_skip_4x):
        # Apply 1x1 convolution to reduce skip channels
        features_skip_4x = self.skip_conv(features_skip_4x)

        # Upsample bottleneck features to match skip feature dimensions
        features_bottleneck = F.interpolate(
            features_bottleneck, size=features_skip_4x.shape[2:], mode='bilinear', align_corners=False
        )

        # Concatenate and process combined features
        combined_features = torch.cat([features_bottleneck, features_skip_4x], dim=1)
        predictions = self.concat_conv(combined_features)

        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.

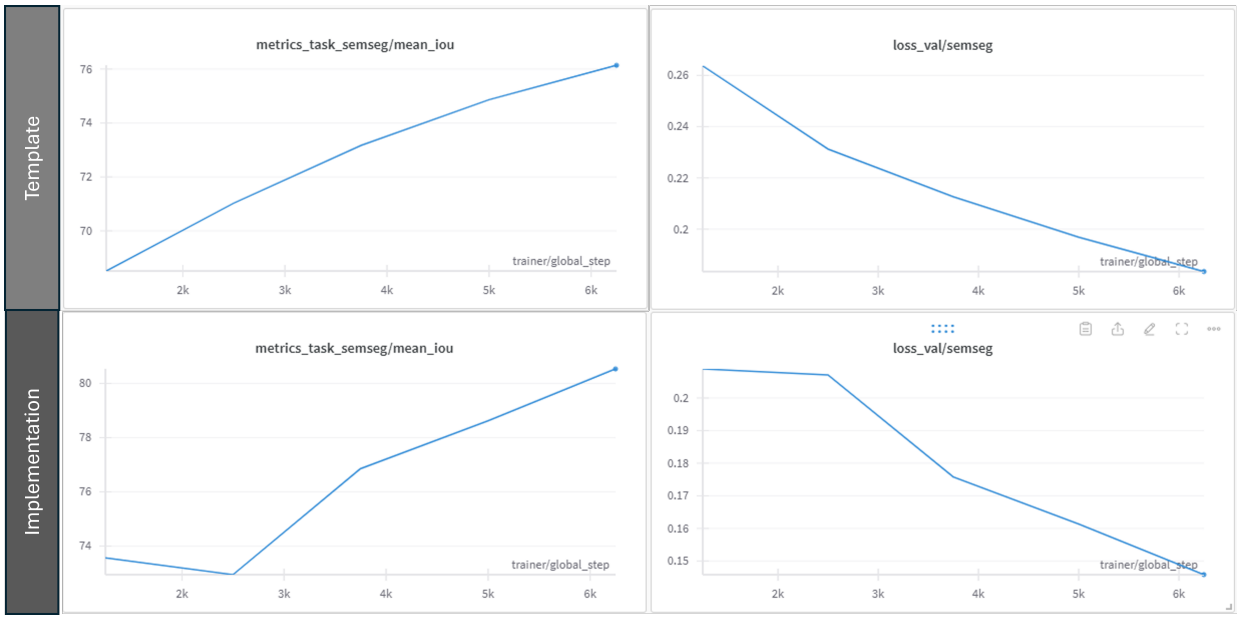

Figure 7: Performance comparison: Dilated Model vs Model with fully implemented ASPP and Skip Connections.

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.

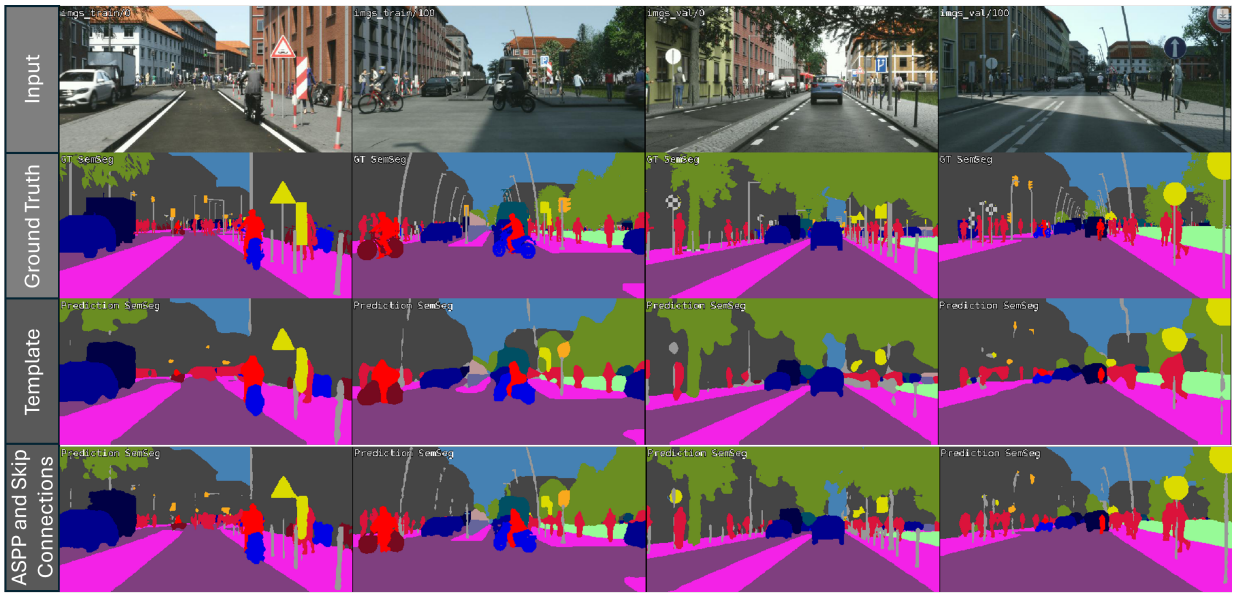

Figure 8: Example Predictions: Dilated Model vs Model with fully implemented ASPP and Skip Connections.

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.

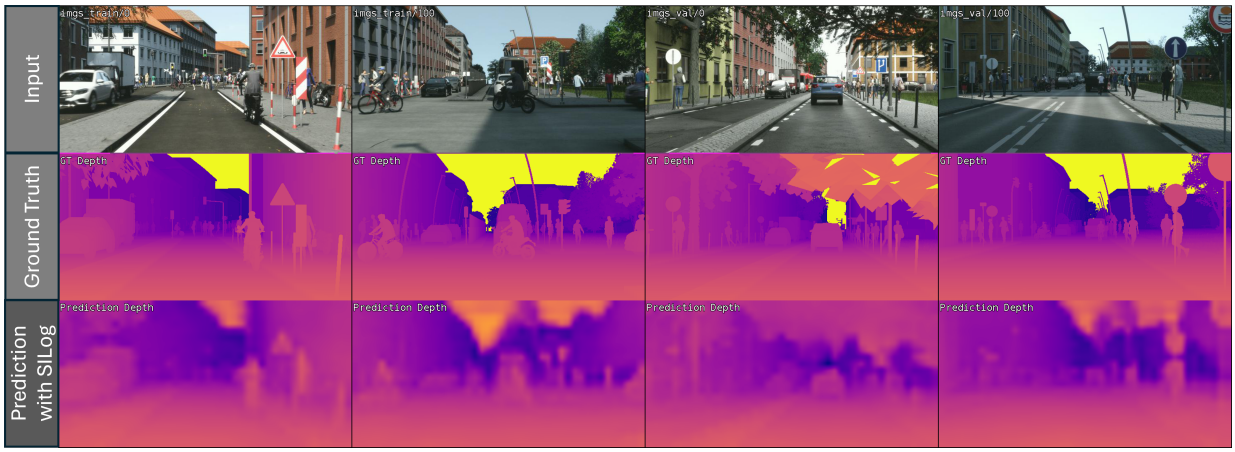

Figure 9: Example Predictions: Depth estimation with SILog implementation.

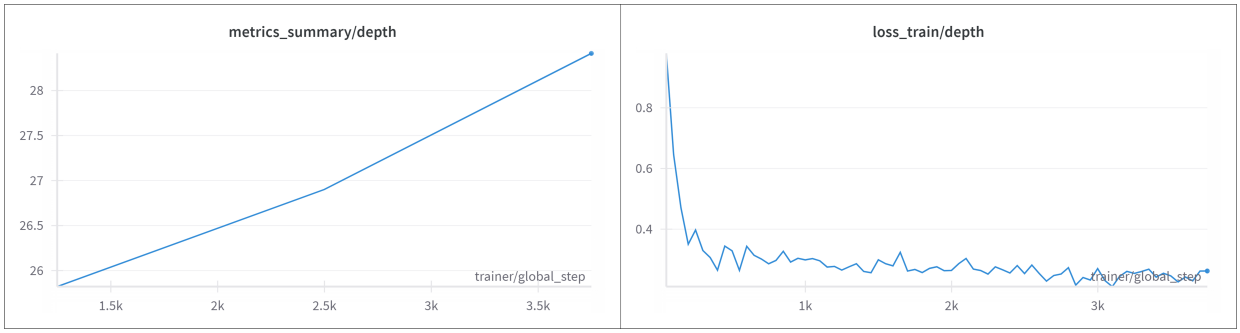

Figure 10: Depth Estimation Performance.

Code Snippet 4: SiLog implementation.

```python
# decorator, to use half-precision (force-disabled) /
# Using half precision might lead to numerical instability
@torch.cuda.amp.autocast(enabled=False)
def forward(self, input: torch.Tensor, target: torch.Tensor) ->
    torch.Tensor:
    # Avoid dummy channel dimension
    if input.ndim == 4:
        input  = input.squeeze(1)
    if target.ndim == 4:
        target = target.squeeze(1)

    # TODO: Implement a proper silog loss and taking into
       consideration
    # the invalid pixels (target is 0).

    # mask to ignore invalid pixels, bzw. use non-zero entries.
       Epsilon for computational tractability
    target_mask = (target > self.eps).float()

    # Log-transform --> scale-invariance, then difference. Epsilon
       for computational tractability
    log_input   = torch.log(torch.clamp(input, min=self.eps))
    log_target  = torch.log(torch.clamp(target, min=self.eps))
    log_diff    = log_input - log_target

    # mean and variance of log_diff using our mask
    mean, var   = masked_mean_var(log_diff, target_mask, dim=[1,
       2])

    # loss computation
    loss = var + self.gamma * mean.pow(2)   # (Don't use numpy here
       , as it breaks the autograd graph)

    # Sum over the batch dimension
    return torch.sqrt(loss.mean())  # return scalar. Sqrt as in
       task descrition.
```

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 ε 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 ε' 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, ε) does not affect its variance. Therefore, the variance of the new error ε' remains the same as the original variance:

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

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

```
            replace_stride_with_dilation=(False, False, True)
        )

        # Get encoder channel dimensions
        ch_out_encoder_bottleneck, ch_out_encoder_4x =
            get_encoder_channel_counts(cfg.model_encoder_name)

        # Task-specific ASPP and Decoder for each output type
        self.aspps = nn.ModuleDict({
            task: ASPP(ch_out_encoder_bottleneck, 256)
            for task in outputs_desc
        })

        self.decoders = nn.ModuleDict({
            task: DecoderDeeplabV3p(256, ch_out_encoder_4x, num_ch)
            for task, num_ch in outputs_desc.items()
        })

    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]

        out = {}

        for task, num_ch in self.outputs_desc.items():
            features_aspp = self.aspps[task](features_lowest)
            predictions_4x, _ = self.decoders[task](features_aspp,
                features[4])
            predictions_1x = F.interpolate(predictions_4x, size=
                input_resolution, mode='bilinear', align_corners=
                False)
            out[task] = predictions_1x.exp().clamp(0.1, 300.0) if
                task == "depth" else predictions_1x

        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.

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

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

class ModelDeepLabV3PlusMultiTask(nn.Module):
    def __init__(self, cfg, outputs_desc):
        super().__init__()
        self.outputs_desc = outputs_desc

        # Task-specific configurations for encoders
        encoder_configs = {
            'semseg': {
                'replace_stride_with_dilation': (False, True, True)
                   ,
                'pretrained': cfg.pretrained,
                'zero_init_residual': True
            },
            'depth': {
                'replace_stride_with_dilation': (False, False, True
                   ),
                'pretrained': cfg.pretrained,
                'zero_init_residual': True
            }
        }

        # Initialize separate encoders for each task
        self.encoders = nn.ModuleDict({
            task: Encoder(
                cfg.model_encoder_name,
                **encoder_configs[task]
            ) for task in outputs_desc
        })

        # Task-specific ASPP configurations
        aspp_configs = {
            'semseg': {
                'rates': (6, 12, 18, 24),
                'out_channels': 256
            },
            'depth': {
```

```
                'rates': (3, 6, 9),
                'out_channels': 256
            }
        }

        self.aspps = nn.ModuleDict({
            task: ASPP(
                ch_out_encoder_bottleneck,
                aspp_configs[task]['out_channels'],
                rates=aspp_configs[task]['rates']
            ) for task in outputs_desc
        })

        self.decoders = nn.ModuleDict({
            task: DecoderDeeplabV3p(
                aspp_configs[task]['out_channels'],
                ch_out_encoder_4x,
                num_ch
            ) for task, num_ch in outputs_desc.items()
        })

    def forward(self, x):
        input_resolution = x.shape[2:]
        out = {}
        for task, num_ch in self.outputs_desc.items():
            features = self.encoders[task](x)
            features_aspp = self.aspps[task](features[highest_scale
                ])
            predictions_1x = F.interpolate(
                self.decoders[task](features_aspp, features[4])[0],
                size=input_resolution,
                mode='bilinear',
                align_corners=False
            )
            out[task] = predictions_1x
        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-

rameter 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.

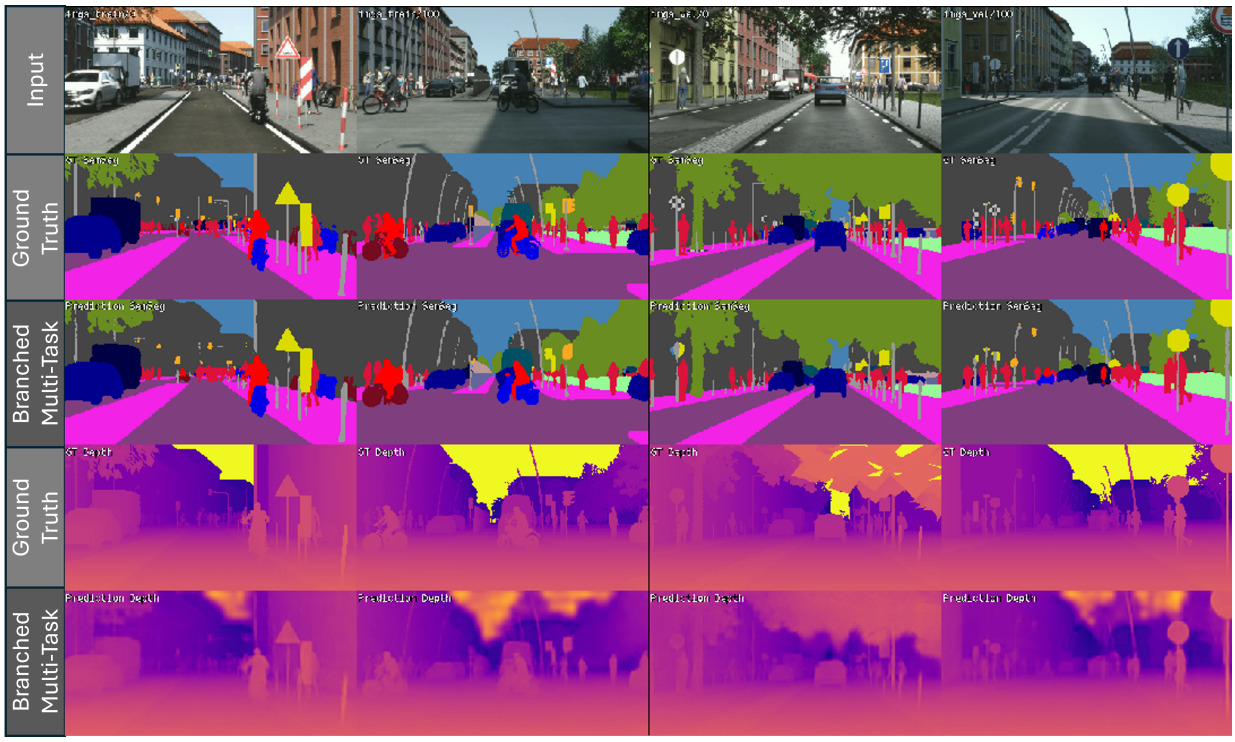

Figure 11: Example Predictions: Branched Multitask Model.

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.

```
    self.proj_q = nn.Linear(dim, dim)
    self.proj_k = nn.Linear(dim, dim)
    self.proj_v = nn.Linear(dim, dim)
    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.

```
    def forward(self, x, pos_embed):

        B, N, C = x.shape

        x = self.norm(x)
        q = self.proj_q(x)
        k = self.proj_k(x)
        v = self.proj_v(x)

        if pos_embed is not None:
            q += pos_embed

        q = q.reshape(B, N, self.num_heads, C // self.num_heads).
            transpose(1, 2)
        k = k.reshape(B, N, self.num_heads, C // self.num_heads).
            transpose(1, 2)
        v = v.reshape(B, N, self.num_heads, C // self.num_heads).
            transpose(1, 2)

        s = q.matmul(k.transpose(2, 3)) / ((C // self.temperature)
            **0.5)
        weights = torch.softmax(s, dim=3)
        weights = self.dropout(weights)

        a = weights.matmul(v)
        a = a.transpose(1, 2).reshape(B, N, C)

```

```
        x = self.out(a)
        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 witch 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.
          num_bins)
2     self.transformer_layers = nn.ModuleList([TransformerBlock(dim=
          ch_out_encoder_bottleneck, num_heads=cfg.num_heads,
          expansion=cfg.expansion)for _ in range(cfg.
          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,
          keepdim=True)
14
15    predictions_1x = F.interpolate(predictions_4x, size=
          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.

```
def inference_step(self, batch):

    ...

    scales = [0.5, 0.75, 1.0, 1.25, 1.5]

    ...

    if aggregated_y_hat_semseg is None:
        aggregated_y_hat_semseg = y_hat_semseg_resized
        aggregated_y_hat_depth = y_hat_depth_resized
    else:
        aggregated_y_hat_semseg = torch.max(aggregated_y_hat_semseg
            , y_hat_semseg_resized)
        aggregated_y_hat_depth = torch.max(aggregated_y_hat_depth,
            y_hat_depth_resized)

    y_hat_semseg_lbl = aggregated_y_hat_semseg.argmax(dim=1)

    ...
```

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 signifcant 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.

```
    ...

        if aggregated_y_hat_semseg is None:
            aggregated_y_hat_semseg = y_hat_semseg_resized
```

```
            aggregated_y_hat_depth = y_hat_depth_resized
        else:
            aggregated_y_hat_semseg += y_hat_semseg_resized
            aggregated_y_hat_depth += y_hat_depth_resized

    aggregated_y_hat_semseg /= len(scales)
    aggregated_y_hat_depth /= len(scales)

    ...
```

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

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

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

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

Scale/Method	avg	max
3 Scales	26.08	19.95
5 Scales	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.

```
class AttentionFusion(nn.Module):
    def __init__(self, skip_scales):
        super().__init__()
        self.attention_layers = nn.ModuleDict({
            scale: nn.Sequential(
                nn.Conv2d(channels, 1, kernel_size=1),
                nn.Sigmoid()
            ) for scale, channels in skip_scales.items()
        })

    def forward(self, skip_features):
        # Compute attention weights for each scale
        attention_weights = {scale: self.attention_layers[scale](
           feat) for scale, feat in skip_features.items()}

        # Apply attention weights to the skip features
        weighted_features = {scale: attention_weights[scale] * feat
            for scale, feat in skip_features.items()}

        # Concatenate all attended features
        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	Grader Multitask Metric	Grader SemSeg Metric	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_halfLR_a1e1c	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.