ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CVL Computer Vision Lab

Computer Vision and Artificial Intelligence for Autonomous Cars

Project 1: Semantic Segmentation and Depth Estimation

Authors:

Liam Achenbach 20-940-268
Jérôme Bonvin 20-917-332
Kai Zhang 23-947-583

December 30, 2025

Table of Contents

Problem 1: Semantic Segmentation

1 Hyperparameter Tuning

Hyperparameter tuning is essential in machine learning, as selecting the right parameters can determine whether the model converges effectively within a reasonable time. Problem 1.1 and 1.2 of this project require fine-tuning key parameters of a deep-learning model, including the optimizer, learning rate, and batch size.

a) Optimizer and Learning Rate

The optimizer and learning rate play central roles in a model's convergence toward an optimal solution. In this project, two optimizers were investigated: Stochastic Gradient Descent (SGD)[1] and Adaptive Moment Estimation (Adam)[2]. While SGD performs a standard gradient-based update, Adam adapts the learning rate for each parameter based on its previous gradients, to accelerate learning and adapt to the gradients of the individual parameters. Additionally, Adam uses momentum to smooth out updates and reduce oscillations. SGD updates the model parameters by taking a step in the direction of the gradient to minimize the loss function. The update rule for SGD is defined as follows:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t) \tag{1}$$

In this equation, θ_t represents the model parameters (weights) at step t, while η is the learning rate that controls the step size. A larger learning rate speeds up learning but may overshoot the optimal solution, whereas a smaller learning rate allows finer adjustments but can slow down convergence. The term $\nabla_\theta J(\theta_t)$ is the gradient of the loss function J with respect to θ_t, guiding the model to reduce the error by updating the parameters in the appropriate direction. Each gradient specifies the direction and magnitude of the update, influencing how quickly the model approaches an optimal solution.

This SGD update mechanism contrasts with Adam's adaptive learning rate and momentum. Adam's adaptive approach can accelerate training on data where SGD would take many more steps, for example on data where the gradient is almost 0 but not a local minima. By adjusting different parameter sensitivities, Adam can lead to faster convergence. An illustration of this advantage is presented in Figures 1.a) and 1.b), we can see that a boost in the plateau on the top right of Figure 1.b) reduces the number of steps needed. Additionally, while SGD oscillates through the loss surface, Adam has much smoother and stable steps.

For this project, twelve training sessions of one epoch each were conducted to determine the optimal combination of optimizer (SGD or Adam) and learning rate (ranging from 10^{-1} to 10^{-6}). The Mean Intersection over Union (IoU) metric was used to evaluate segmentation performance and identify the best candidate. As shown in Figure 1.c), the Adam optimizer with a learning rate of 10^{-4} achieved the highest mIoU score of 40.89 after one epoch, making it the best-performing combination. The higher oscillation noise in the loss of SGD compared to the adaptive stepping of Adam wasn't noticeable yet in the first epoch as seen in Figures 1.d), 1.e) and 1.f). However, it is apparent from Figure 1.e) that Adam with a learning rate of 10^{-4} performs larger steps towards a lower loss quicker than SGD with the same learning rate. Therefore, all subsequent experiments in this report were conducted using Adam optimizer with a learning rate of 10^{-4}. The majority of changes for Problem 1 were made by adjusting the hyperparameters in *shell_train.py* and *slurm_train.sh*, as demonstrated in Code Snippet 1. Unless specified otherwise, any further code modifications will involve only these hyperparameters, so we will not repeat these updates as they are straightforward.

shell_train.sh and slurm_train.sh

```
# ... previous code ...
# You can specify the hyperparameters and the experiment name here.
python -m source.scripts.train \
    --log_dir ${SAVEDIR} \
  --dataset_root ${TMPDIR}/miniscapes \
  --name adam_0.0001_lr \
  --model_name semseg \
  --optimizer adam \
  --tasks depth \
  --optimizer_lr 0.0001 \
  --batch_size 16 \
  --num_epochs 3 \
  --workers ${SLURM_CPUS_PER_TASK} \
  --workers_validation ${SLURM_CPUS_PER_TASK} \
  --batch_size_validation 32 \
  --optimizer_float_16 no \
  --pretrained True \
  --model_encoder_name 'resnet18'  \
  #... rest of the code ...
```

Code Snippet 1: Hyperparameter Declaration

b) Batch Size

The batch size in training has a significant impact on performance metrics and various aspects of a multitask network. When increasing the batch size, the number of steps per epoch decreases proportionally since each step processes more data. Increasing the batch size generally stabilizes the gradients because averaging is done over more samples reducing noise in gradient calculation, leading to smoother updates [4, 5]. However, smaller batch sizes may introduce more variability in gradient updates, which can sometimes help the model escape local minima.

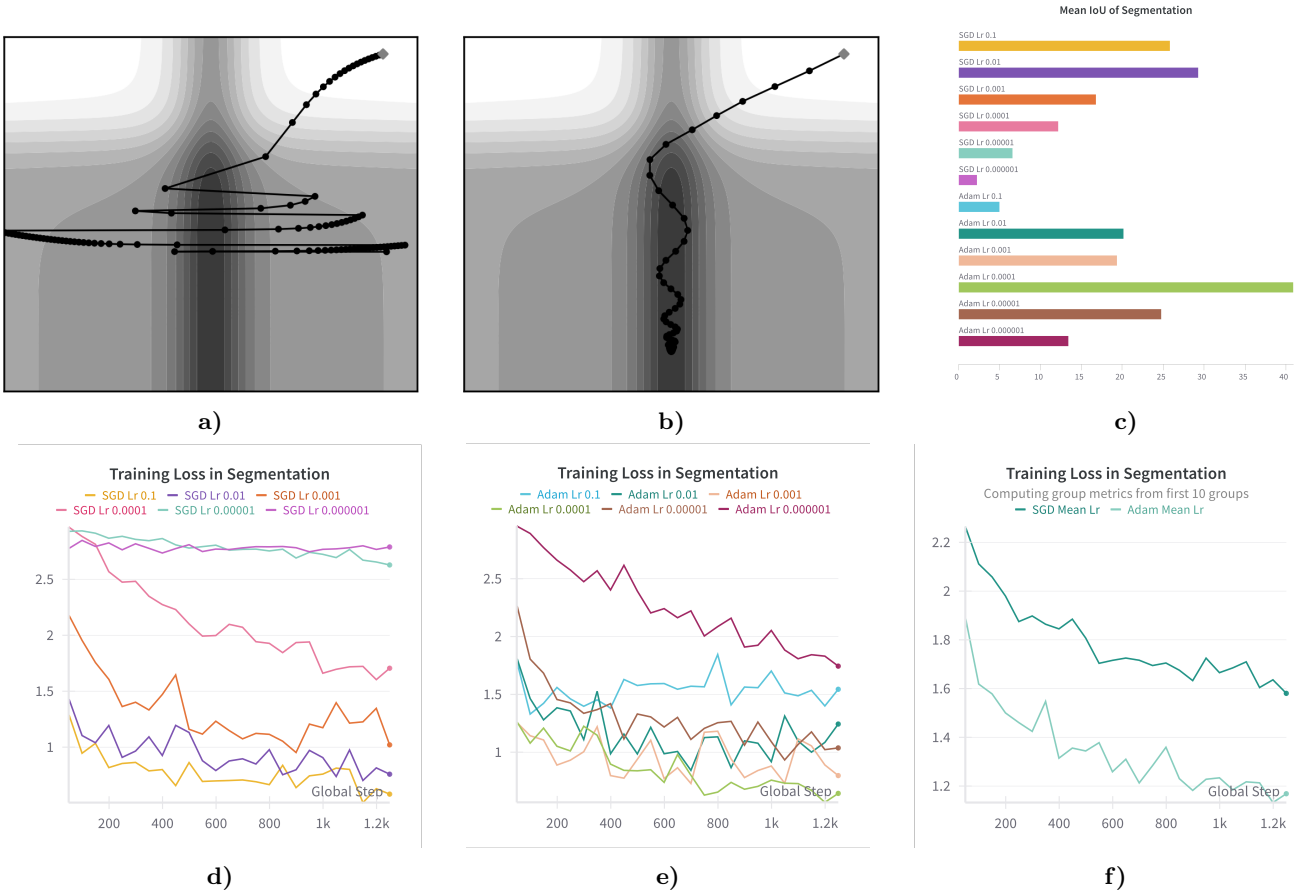


Figure 1: Optimizer and Learning Rate a) Illustration of the convergence of SGD in the loss surface. Darker shaded areas indicate lower loss.[3] b) Illustration of the convergence of Adam in the same loss surface as SGD.[3] c) Mean IoU of segmentation of different optimizers and learning rates d) Training loss for SGD for various learning rates e) Training loss for Adam for various learning rates f) Mean learning rate training loss in SGD vs Adam

Batch size also affects training time and efficiency. As seen in Figure 2.a), smaller batch sizes require more iterations per epoch, resulting in higher total computation time if the number of epochs remains constant. One reason is that larger batch sizes can take advantage of the parallel computing in the cuda kernel, reducing per sample training time, leading to shorter overall training time if the computational resources are sufficient. If the number of epochs increases, so does the time, which is obvious because of more computations and the results are also shown in Figure 2.a).
Having a low batch size and a high number of epochs gives the best results as illustrated in Figure 2.b). We can see that there is a general trend that if you train for longer, the mIoU gets better. Another result that can be extracted is that the number of global steps is linearly proportional to the number of epochs and inversely to the batch size, for example the run with 64 batch size and 2 epochs has the same number of global steps as the run with 32 batch size and 1 epoch. Smaller batch size creates more iterations and variations, which might enable better mIoU score.

2 Hardcoded Hyperparameters

Besides hyperparameter tuning, hardcoded hyperparameters offer a quick and straightforward approach to reduce training time and improve computational efficiency. In this section, we focus on two parameters: Encoder Initialization and Dilated Convolutions.

a) Encoder Initialization

The Encoder is either initialized with pretrained weights or initialized with random weights depending on the "pretrained" flag which is fed as an argument in the *config.py* file. Initializing with pretrained weights, especially weights that have been pretrained on a dataset enables the Encoder to capture rich feature representations quickly. An Encoder with random weights first has to be trained on the current task and thus has additional layers that have to be tuned. This leads to a higher loss and a higher training time for the random initialization. On the contrary, the pretrained model can save a large amount of training time and compute for achieving effective Encoder output.

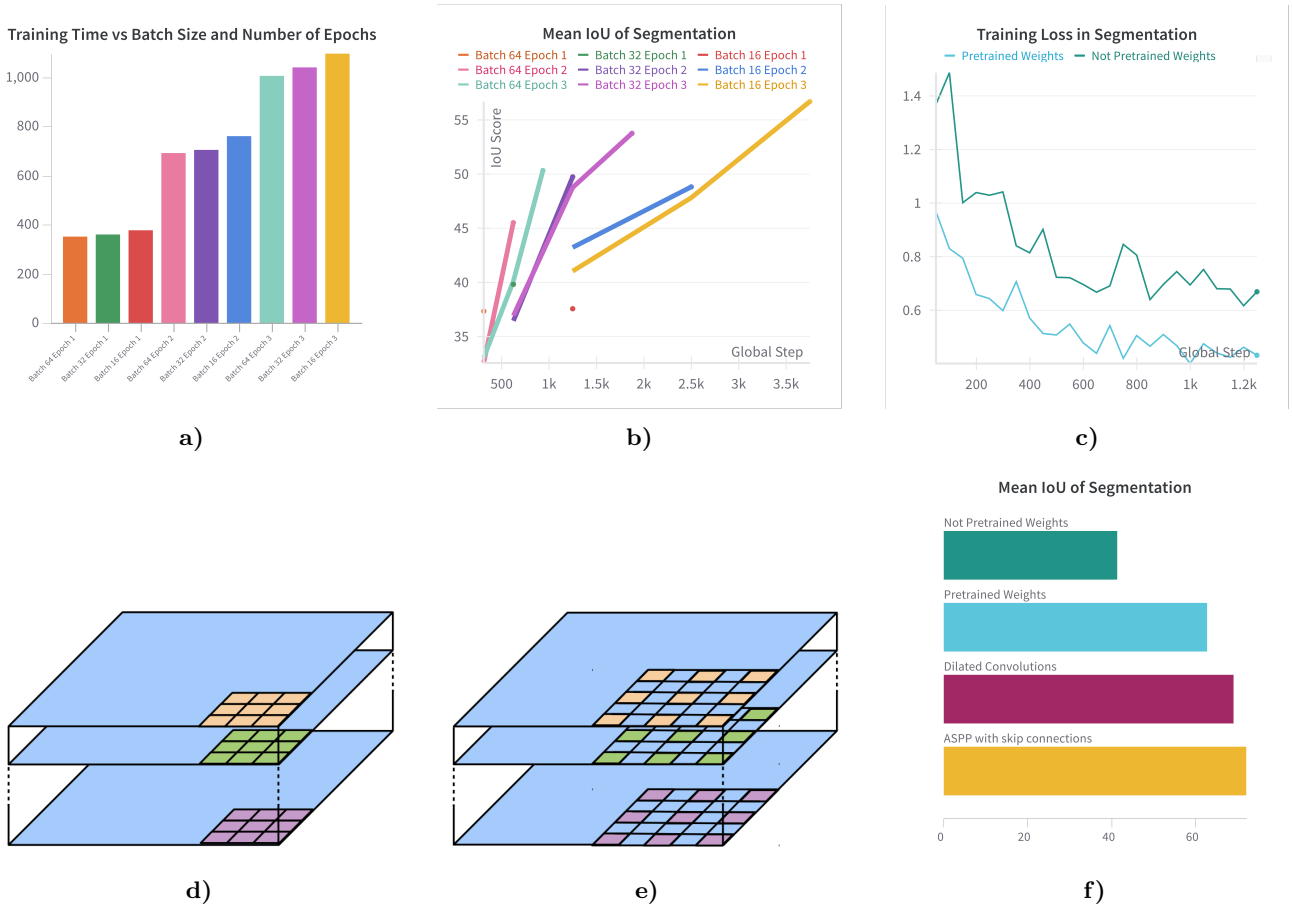


Figure 2: Batch Size and Hardcoded Hyperparameters a) Training time in seconds compared to various batch sizes and number of epochs b) Mean IoU of segmentation of different batch sizes and number of epochs c) Training loss progress of pretrained weights and randomly assigned weights d) Depthwise convolution [6] e) Atrous convolution [6] f) IoU of randomly assigned weights, pretrained weights, dilated convolutions and ASPP with skip connections

In this task, we initialized specific layers with pretrained weights from a pretrained network which is called *resnet18* [7]. In the code, the PyTorch implementation of *resnet18* is used, which is pretrained on the ImageNet-1K dataset with RGB images of size 256x256. This dataset is diverse and large with 1.2M different images and 1000 object classes. This is especially valuable since the train dataset in our task is limited in size and amount of classes, so a model pretrained on a larger dataset will be more robust. It is reasonable to say that the pretrained encoder already has a general understanding for object classes that are in the ImageNet dataset and can adapt in some sense from this to our task, which is a main benefit of pretraining. To demonstrate the impact of pretrained weights, the model was trained for one epoch in two versions: one with pretrained weights and the other with randomly initialized weights. As shown in Figure 2.f), the pretrained model outperforms the randomly initialized model in early performance metrics. Additionally, Figure 2.c) illustrates that while both models follow similar learning curves, the pretrained network effectively "jumps ahead," suggesting an early performance advantage. The pretrained configuration will be retained for further experiments to maximize performance.

b) Dilated Convolutions

Dilated, or atrous, convolutions expand the receptive field of a convolutional filter by introducing gaps within the filter's grid, allowing it to cover a larger portion of the input without increasing the number of parameters or computational demands [6]. A visual representation of the differences are given in Figures 2.d) and 2.e). This approach enables the model to capture more contextual information from a wider area, which is crucial in applications like semantic segmentation where both fine detail and broad context matter.
Traditional convolutions or pooling layers reduce the resolution of feature maps to achieve a broader view of the input, often resulting in a loss of detailed information. Dilated convolutions, however, preserve the original res-

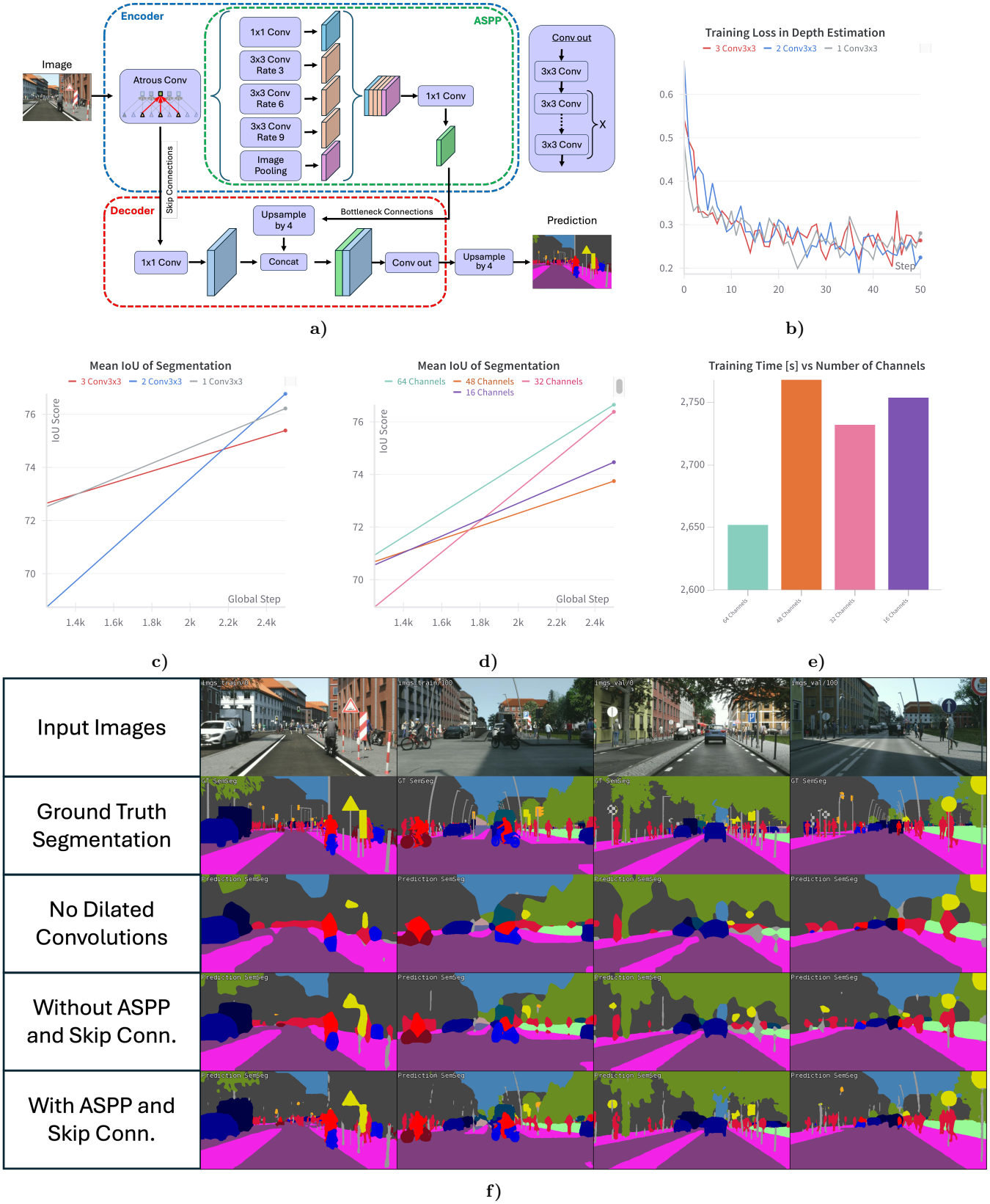


Figure 3: ASPP and Skip Connections a) Structure of the model [6] b) Effect of the amount of 3x3 convolutions on the loss c) Effect of 3x3 convolutions on the mIoU d) Effect of the number of channels in the skip connections on the mIoU after 4 epochs of training e) Effect of the number of channels in the skip connections on the training time f) Comparison of not dilated convolutions, with dilated convolutions but without ASPP and ASPP predictions predictions after 1 epoch of training

olution, ensuring that fine-grained features, like object boundaries, remain sharp while still benefiting from an expanded receptive field. Additionally, when different dilation rates are applied simultaneously, the model can recognize features across multiple scales, which improves its ability to generalize across objects of varying sizes within the same image.

To sum it up, dilated convolutions offer a way to achieve both precise localization and broader context, making them ideal for dense prediction tasks that require a balance between detail and scope. Unlike pooling, which sacrifices spatial resolution, dilated convolutions maintain spatial detail while still expanding the receptive field.

In our task we opt for the (False, False, True) setting which replaces the fourth layer of the ResNet encoder with a dilation layer as seen in Code Snippet 2. This avoids a further downsampling of the encoded features. Our model can keep the reception field of the original ResNet while having a larger feature map. This keeps more fine-grained details and achieves better IoU scores. The results of this choice on the segmentation output can be seen in Figure 3.f).

model_deeplab_v3_plus.py

```python
import torch
import torch.nn.functional as F
from source.models.model_parts import Encoder, get_encoder_channel_counts, ASPP, DecoderDeeplabV3p
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,
            #changed (False,False,False) to (False,False,True)
            replace_stride_with_dilation=(False, False, True),
        )
        #... rest of code ...
```

Code Snippet 2: Code for the DeepLabV3+ Model

3 ASPP and Skip Connections

To elaborate on atrous convolutions, we introduce the Atrous Spatial Pyramid Pooling (ASPP) module. ASPP enables the network to detect both small and large objects within the same scene by integrating spatial information from both nearby and distant regions of the feature map, utilizing the previously discussed atrous convolutions. This capability is particularly valuable for dense prediction tasks, where accurately labeling each pixel is essential [6]. Adding an ASPP module helps the model retain finer details from higher features into lower features by increasing the amount of channels.

The model structure was modified to incorporate ASPP in addition to the existing architecture. Figure 3.a) illustrates these structural adjustments. After training the model, a slight improvement in mean IoU is observed, as shown in Figure 2.f). Although the increase in mean IoU appears minor, Figure 3.f) demonstrates a notable enhancement in edge sharpness and shape definition. This result indicates that while future progress may yield smaller IoU gains, it will likely lead to more refined object boundaries and clearer shapes. This seems to be in line with our expectation of ASPP improving the models all-round understanding of the scene.

To fine tune a bit more the ASPP, we wanted to investigate how many number of out channels after the 1x1 convolution of the skip connections were optimal. 16, 32, 48 and 64 channels were chosen to be investigated since the skip connections have a total of 64 channels. In theory, we want to reduce the number of channels since we want the bottleneck connections to be the main focus of the computation and not confuse too much with the low level features added from the skip connections part. However, as seen in Figure 3.d), 64 channels out performed the other number of channels and on top of that, it needed less time as illustrated in Figure 3.e). Therefore, we adapt 64 channels for the rest of our implementation.

Another parameter that we can fine tune is the number of convolution layers that we put in the *Conv out* module that is present in Figure 3.a). We trained the model 3 times with X being 0, 1 or 2 (X represents the number of additional convolutions that we add to *Conv out*. This is in line with the implementation in the original paper which mentions adding two layers [6]. Figure 3.c) shows the mIoU of achieved in the first two epochs. It is evident that two 3x3 convolutions achieve an initial lower score than one or three. However, we decided to use two convolutions for the rest of the implementation due to the sharp upwards trend that can be observed for the mIoU and the stably decreasing loss as seen in Figure 3.b). We again re-evaluate this decision in the multitask problem to ensure we still made an optimal choice. The relevant changes made for the implementation of the ASPP module and the skip connections are displayed in Code Snippet 3 and Code Snippet 4.

model_parts.py

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

class ASPP(torch.nn.Module):
    def __init__(self, in_channels, out_channels, rates=(3, 6, 9)):
        super().__init__()
        # TODO: Implement ASPP properly instead of the following
        #self.conv_out = ASPPpart(in_channels, out_channels, kernel_size=1, stride=1, padding=0, dilation=1)
```

```
        #### START OF OUR IMPLEMENTATION ####
        self.aspp_blocks = torch.nn.ModuleList([
            ASPPpart(in_channels, out_channels, kernel_size=1, stride=1, padding=0, dilation=1)
        ] + [
            ASPPpart(in_channels, out_channels, kernel_size=3, stride=1, padding=rate, dilation=rate) for rate in rates
        ])
        # Pooling layer
        self.global_avg_pool = torch.nn.Sequential(
            torch.nn.AdaptiveAvgPool2d((1, 1)),
            torch.nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=1, bias=False),
            torch.nn.BatchNorm2d(out_channels),
            torch.nn.ReLU()
        )
        total_out_channels = out_channels * (len(rates) + 2) # +1 for 1x1 +1 for global pooling
        self.conv_out = torch.nn.Conv2d(total_out_channels, out_channels, kernel_size=1, stride=1)
        #### END OF OUR IMPLEMENTATION ####

    def forward(self, x):
        # TODO: Implement ASPP properly instead of the following
        #### START OF OUR IMPLEMENTATION ####
        aspp_outs = [block(x) for block in self.aspp_blocks]
        # Apply global average pooling
        pooled = self.global_avg_pool(x)
        pooled = F.interpolate(pooled, size=x.shape[2:], mode='bilinear', align_corners=False)
        aspp_outs.append(pooled)
        out = torch.cat(aspp_outs, dim=1)
        #### END OF OUR IMPLEMENTATION ####
        out = self.conv_out(out)
        return out
# ... rest of code ....
```

Code Snippet 3: Code for the ASPP module

model_parts.py

```
# ... previous code ....
class DecoderDeeplabV3p(torch.nn.Module):
    def __init__(self, bottleneck_ch, skip_4x_ch, num_out_ch):
        super(DecoderDeeplabV3p, self).__init__()
        # TODO: Implement a proper decoder with skip connections instead of the following
        #### START OF OUR IMPLEMENTATION ####
        self.skip_channels = 64 # 64 better before rest better if more training
        # Convolution to reduce the number of channels in the skip connection
        self.conv_skip = torch.nn.Conv2d(skip_4x_ch, self.skip_channels, kernel_size=1, stride=1)
        # Convolution to combine the bottleneck features and the skip connection
        self.conv1_student = torch.nn.Sequential(
            torch.nn.Conv2d(bottleneck_ch + self.skip_channels, 256, kernel_size=3, stride=1, padding=1),
            torch.nn.BatchNorm2d(256),
            torch.nn.ReLU()
        )
        self.conv2_student = torch.nn.Sequential(
            torch.nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            torch.nn.BatchNorm2d(256),
            torch.nn.ReLU()
        )
        self.conv3_student = torch.nn.Sequential(
            torch.nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            torch.nn.BatchNorm2d(256),
            torch.nn.ReLU()
        )
        # Final convolution to get the output predictions
        self.features_to_predictions = torch.nn.Conv2d(256, num_out_ch, kernel_size=1, stride=1)
        #### END OF OUR IMPLEMENTATION ####

    def forward(self, features_bottleneck, features_skip_4x):
        """
        DeepLabV3+ style decoder
        :param features_bottleneck: bottleneck features of scale > 4
        :param features_skip_4x: features of encoder of scale == 4
        :return: features with 256 channels and the final tensor of predictions
        """
        # TODO: Implement a proper decoder with skip connections instead of the following; keep returned
        #       tensors in the same order and of the same shape.
        features_4x = F.interpolate(features_bottleneck, size=features_skip_4x.shape[2:], mode='bilinear', align_corners=False)
        ### START OF OUR IMPLEMENTATION ####
        #Reduce the number of channels in the skip connection
        features_skip_4x = self.conv_skip(features_skip_4x) #TODO directly from the encoder side
        # Concatenate the upsampled bottleneck features and the skip connection features
        combined_features = torch.cat([features_4x, features_skip_4x], dim=1)
        # Apply the combined convolution layers
        combined_features = self.conv1_student(combined_features) #just need one 3 by 3 convolution
        combined_features = self.conv2_student(combined_features)
        combined_features = self.conv3_student(combined_features)
        # Get the final predictions
        predictions_4x = self.features_to_predictions(combined_features)
        ### END OF OUR IMPLEMENTATION ####
        return predictions_4x, combined_features
# ... rest of code ....
```

Code Snippet 4: Code for the DecoderDeepLabV3+

Problem 2: Introducing a Second Task: Depth Estimation

In addition to segmenting an image, depth estimation provides valuable information about the environment around a car. This can help prioritize objects closer to the car over those farther away. For instance, a pedestrian 5 meters away is more likely to influence the car's path than one at 50 meters, highlighting the importance of depth estimation.

1 Depth Loss

To introduce depth estimation, the number of output channels from the decoder is increased to include depth information, while the overall architecture remains the same. The structure is depicted in Figure 4.a). Training requires defining a suitable loss function, for which the Scale-Invariant Log (SILog) Loss is used. The goal is to minimize the variance of this error:

$$\begin{aligned} \text{Minimize } V(\epsilon), \quad & \epsilon = \log(y^*) - \log(\hat{y}), \\ \text{subject to:} \quad & E[\epsilon]^2 \leq \mu \end{aligned}$$

The $E[\epsilon]^2 \leq \mu$ constraint is essential to prevent large depth deviations while allowing the model to focus on depth structure in a way that produces both accurate and robust depth outputs across the entire scene. Essentially, this constraint prevents the model to only minimize the variance at the cost of high bias. This prevents the model from overfitting during the training process and thus enables better generalization to unseen data. To enforce the constraint $E[\epsilon]^2 \leq \mu$, a Lagrange multiplier λ is applied, reformulating the objective function as:

$$\mathcal{L} = V(\epsilon) + \lambda \left(E[\epsilon]^2 - \mu \right)$$

Substituting $\epsilon = \log(y^*) - \log(\hat{y})$ and removing the $-\lambda \cdot \mu$ term since it's just a constant, the Lagrangian becomes:

$$\mathcal{L} = V(\log(y^*) - \log(\hat{y})) + \lambda E[(\log(y^*) - \log(\hat{y}))]^2$$

This form, with λ as a regularization parameter, controls scale invariance by penalizing the variance of the log error. A function $f(\cdot)$ is scale-invariant if $f(a \cdot x) = f(x)$. To verify that the SILog loss is scale-invariant, here is the demonstration:

$$\begin{aligned} \epsilon' &= \log(ay^*) - \log(a\hat{y}) \\ &= (\log(a) + \log(y^*)) - (\log(a) + \log(\hat{y})) \\ &= \log(y^*) - \log(\hat{y}) = \epsilon \end{aligned}$$

Since the error stays unchanged, the variance also remains unchanged. This shows that scaling the values does not affect the loss, achieving the intended scale invariance. This reduces the absolute difference error and emphasizes the relative difference. This approach naturally penalizes errors at closer depths more than those farther away, creating the desired scale-invariant property.

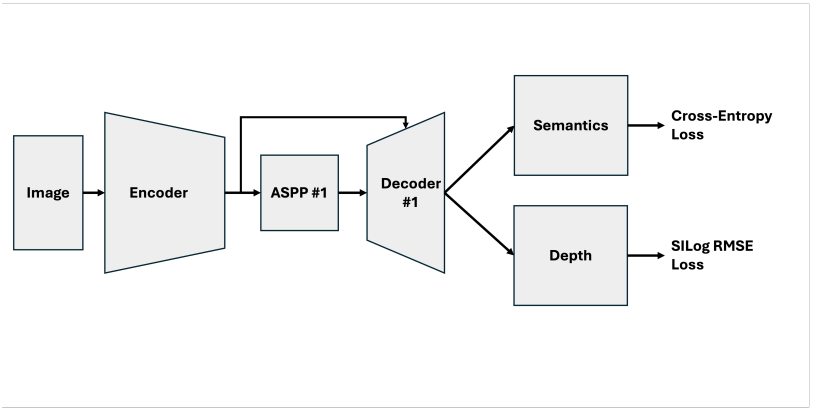
2 Task weighting

Since 2 losses are at play at the same time, the general loss is a weighted sum of the individual losses: $\mathcal{L}_{TOTAL} = (1 - w) \cdot \mathcal{L}_{CROSS} + w \cdot \mathcal{L}_{SILOG}$. Up until now w was set to 0.5 so that both losses have the same importance. Therefore, to obtain the optimal weight w, 3 tests were conducted: $w = 0.25$, 0.5 and 0.75. The results of the individual losses are presented in Figures 4.b) and 4.d). A conclusion from these previously cited graphs is that if a loss is attributed more weight than the other (e.g. 0.25 to cross entropy loss and 0.75 to SILog loss), it will perform better in that task than the other. This is a totally logical response since it should comprise a bigger part of the total loss. Another result that can be taken is that the symmetrical weights achieve good losses in both cases, it achieves almost as good as the best scores of the 3 tests. For the rest of the report, w will be assigned to 0.5 due to this observation. In Figure 5.a), we can see which pixels were predicted compared to the ground truth. This is insightful since we can see that some progress can be made in differentiating between trees, buildings, posts and pedestrians.

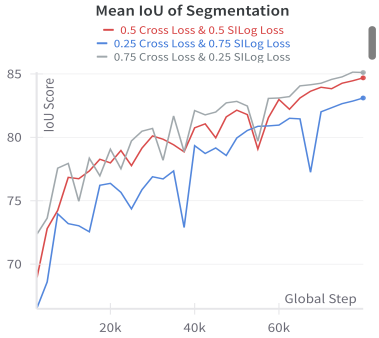
si_log.py

```python
# ... previous code ....
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
        ### START OF OUR IMPLEMENTATION ###
        valid_mask = target > 0
        input_clamped = input.clamp(min=self.eps)
        target_clamped = target.clamp(min=self.eps)
        log_input = torch.log(input_clamped)
        log_target = torch.log(target_clamped)
        error = log_target - log_input
        mean, scale_invariance = masked_mean_var(error, valid_mask, [0,1,2])
        mse = torch.pow(mean,2.0)
        loss_sq = scale_invariance + self.gamma * mse
        loss_sq = loss_sq.squeeze(1).squeeze(1)
        ### END OF OUR IMPLEMENTATION ###
        return torch.sqrt(loss_sq + self.eps)
```

Code Snippet 5: Code for the forward loop of the SIlog RMSE Loss



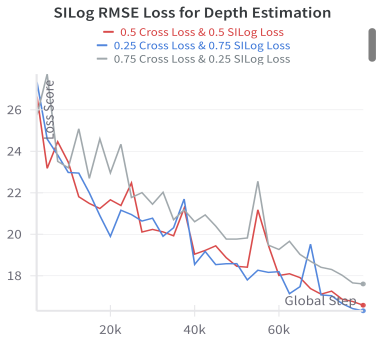
a)



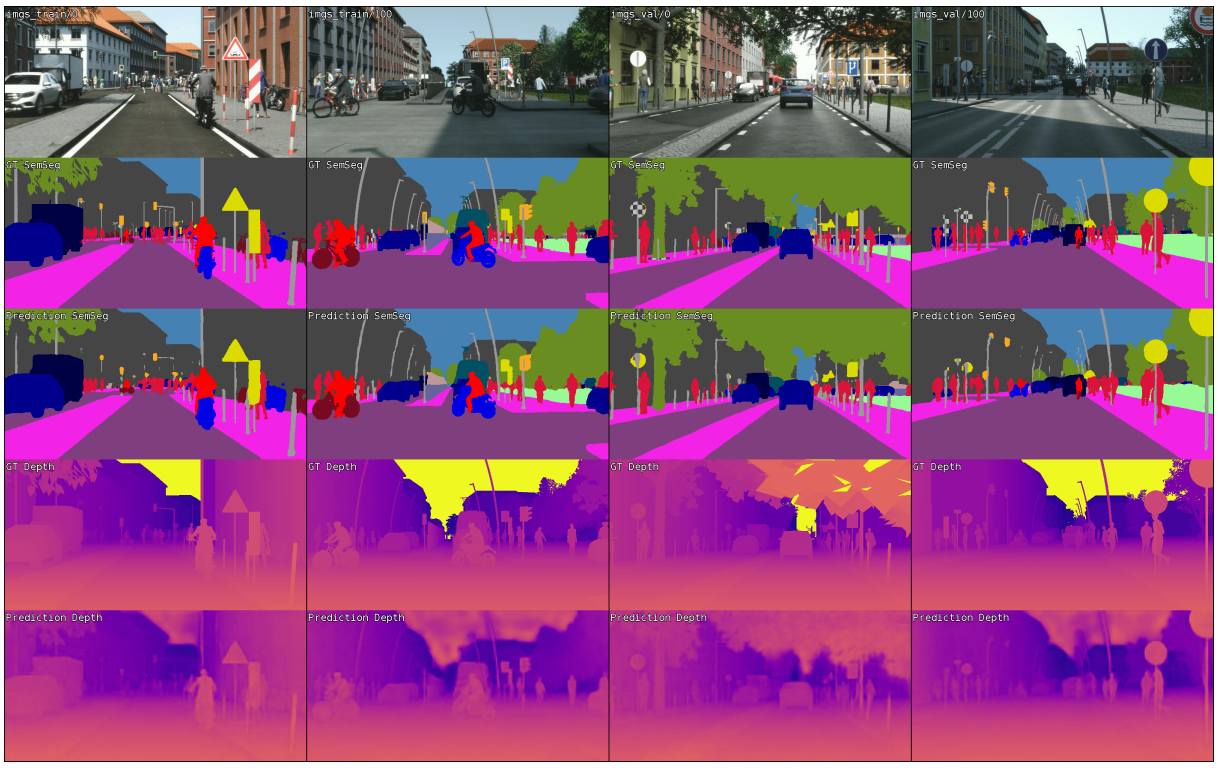
b)

Weighted Loss Models	Segmentation IoU	SILog Loss
0.25 Cross / 0.75 SILog	83.1%	16.3
0.5 Cross / 0.5 SILog	84.7%	16.6
0.75 Cross / 0.25 SILog	85.1%	17.6

c)

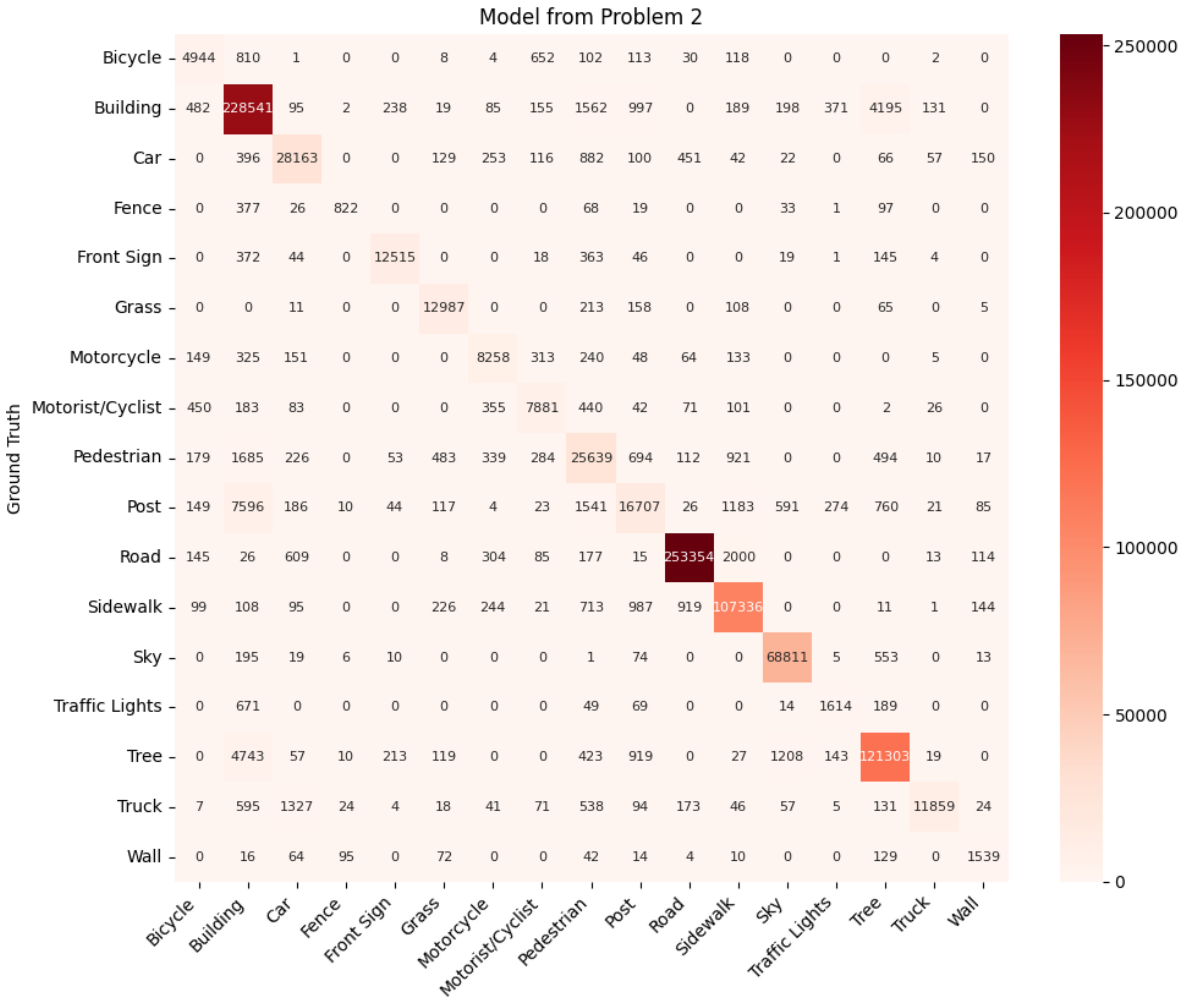


d)



e)

Figure 4: Depth Estimation a) Structure Depth Loss b) IoU of Segmentation with varying weights for the losses c) Segmentation IoU and SILog Loss values for different weighted models d) Depth estimation loss with varying weights for the losses e) Segmentation and Depth Predictions (with weights of 0.5 and 0.5) of different images



a)

Figure 5: Depth Estimation Continued a) Confusion Matrix for Predicted labels and groud truth. As shown in the matrix, our model performs most poorly in classes of trees, buildings, posts and pedestrains.

Problem 3: MultiTask Learning

1 Multitask Model

To improve depth estimations, separating segmentation and depth predictions immediately after the encoder may be more effective than making this separation only at the final convolutional layer. This approach involves creating two ASPP modules and two decoders, each dedicated to either segmentation or depth prediction, rather than using a single decoder for both tasks. The revised architecture is shown in Figure 6.a).
This design enables the allocation of additional computational resources specifically for depth estimation and segmentation. As illustrated in Figures 6.b) and 6.c), this modified model achieves comparable results in half the epochs (16 epochs for the new model compared to 32 epochs for the joint architecture), demonstrating a marked improvement in design efficiency. To elaborate a bit more on its improvement, we computed again the confusion matrix just like in Problem 2 but for the new model. The confusion matrix is shown in 7.a). Additionally, to further show the improvement, we subtract the confusion matrix in Figure 7.a) with the confusion matrix in Figure 5.a), this gives us a good overview of which classes have improved the most.

2 GPU Performance

However, this benefit is offset by significantly increased training time, as shown in Figure 6.d). Specifically, training the model with one ASPP and one decoder took 4 hours and 46 minutes for 32 epochs, whereas training with two ASPPs and two decoders took 8 hours and 57 minutes for just 16 epochs. This means the modified model requires approximately four times as much time to train for the same number of epochs.
Examining the number of parameters in each model provides some insight into this time difference. As detailed in Tables 1 and 2, the one-ASPP model has 16 million parameters, while the two-ASPP model has 22 million—a 1.33x increase in parameters. The code to retrieve these numbers of parameters is given in Code Snippet 6. A similar increase can be observed in GPU memory usage, as shown in Figure 6.f). However, this increase alone does not account for the 4x increase in training time.

Module	Parameters
Encoder	11,176,512
ASPP	4,131,584
Decoder	1,300,036
Total	16,608,132

Table 1: Number of parameters for previous (shared) model

Module	Parameters
Encoder	11,176,512
ASPP_Seg	4,131,584
ASPP_Depth	4,131,584
Decoder_Seg	1,299,779
Decoder_Depth	1,295,153
Total	22,034,612

Table 2: Number of parameters for multitask model

train.py

```python
# ... previous code ....
def count_parameters(model):
    print(f"{'Modules':<40} {'Parameters'}")
    print("-" * 60)
    total_params = 0
    for name, parameter in model.named_parameters():
        if not parameter.requires_grad:
            continue
        params = parameter.numel()
        print(f"{name:<40} {params}")
        total_params += params
    print("-" * 60)
    print(f"Total Trainable Params: {total_params}")
    return total_params
# ... rest of code ....
```

Code Snippet 6: Code for retrieving the number of parameters in models

Additional analysis of GPU power usage provides further insights. As shown in Figure 6.e), the original model (Problem 2) consumes more power during training than the multitask model, with an average of 207 watts compared to 130 watts for the multitask model. This indicates that the multitask model's architecture is less efficient in backpropagation, likely due to dependencies on previously computed gradients, which limit parallelization and increase training time.
Overall, this analysis highlights the trade-offs involved in optimizing architecture for both computational focus and training efficiency.

model_deeplab_v3_plus_multitask.py

```python
# ... previous code ....
class ModelDeepLabV3PlusMultiTask(torch.nn.Module):
```

```python
    def __init__(self, cfg, outputs_desc):
        super().__init__()
        self.outputs_desc = outputs_desc
        self.encoder = Encoder(
            cfg.model_encoder_name,
            pretrained=cfg.pretrained,
            zero_init_residual=True,
            replace_stride_with_dilation=(False, False, True),
        )
        ch_out_encoder_bottleneck, ch_out_encoder_4x = get_encoder_channel_counts(cfg.model_encoder_name)
        aspps = {}
        decoders = {}
        # TODO: Implement aspps and decoders for each task as a ModuleDict.
        ### MY IMPLEMENTATION ####
        self.aspps = nn.ModuleDict({
            'aspp_seg': ASPP(ch_out_encoder_bottleneck, 256),
            'aspp_depth': ASPP(ch_out_encoder_bottleneck, 256)
        })

        self.decoders = nn.ModuleDict({
            'decoder_seg': DecoderDeeplabV3p(256, ch_out_encoder_4x, self.outputs_desc['semseg']),  # Segmentation decoder
            'decoder_depth': DecoderDeeplabV3p(256, ch_out_encoder_4x, self.outputs_desc['depth'])
        })

        ### MY IMPLEMENTATION ####

    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 = {}
        # TODO: implement the forward pass for each task as in the previous exercise.
        # However, now task's aspp and decoder need to be forwarded separately.
        ### START OF OUR IMPLEMENTATION ###
        # Segmentation Task
        seg_features_tasks = self.aspps['aspp_seg'](features_lowest)
        seg_predictions_4x, _ = self.decoders['decoder_seg'](seg_features_tasks, features[4])
        seg_predictions_1x = F.interpolate(seg_predictions_4x, size=input_resolution, mode='bilinear', align_corners=False)

        # Depth Estimation Task
        dep_features_tasks = self.aspps['aspp_depth'](features_lowest)
        dep_predictions_4x, _ = self.decoders['decoder_depth'](dep_features_tasks, features[4])
        dep_predictions_1x = F.interpolate(dep_predictions_4x, size=input_resolution, mode='bilinear', align_corners=False)
        # Construct output dictionary for each task
        offset = 0
        # Adjusting the forward pass:
        for task, num_ch in self.outputs_desc.items():
            if task == 'semseg':
                current_prediction = seg_predictions_1x  # Shape: [16, 19, 256, 256]
            elif task == 'depth':
                current_prediction = dep_predictions_1x  # Shape: [16, 1, 256, 256]
                # Ensure depth predictions are clamped to realistic values
                current_prediction = current_prediction.exp().clamp(0.1, 300.0)
            else:
                raise ValueError(f"Unknown task: {task}")
            out[task] = current_prediction
        ### END OF OUR IMPLEMENTATION ###
        return out
```

Code Snippet 7: Code for the DeeplabV3+ Multitask Model

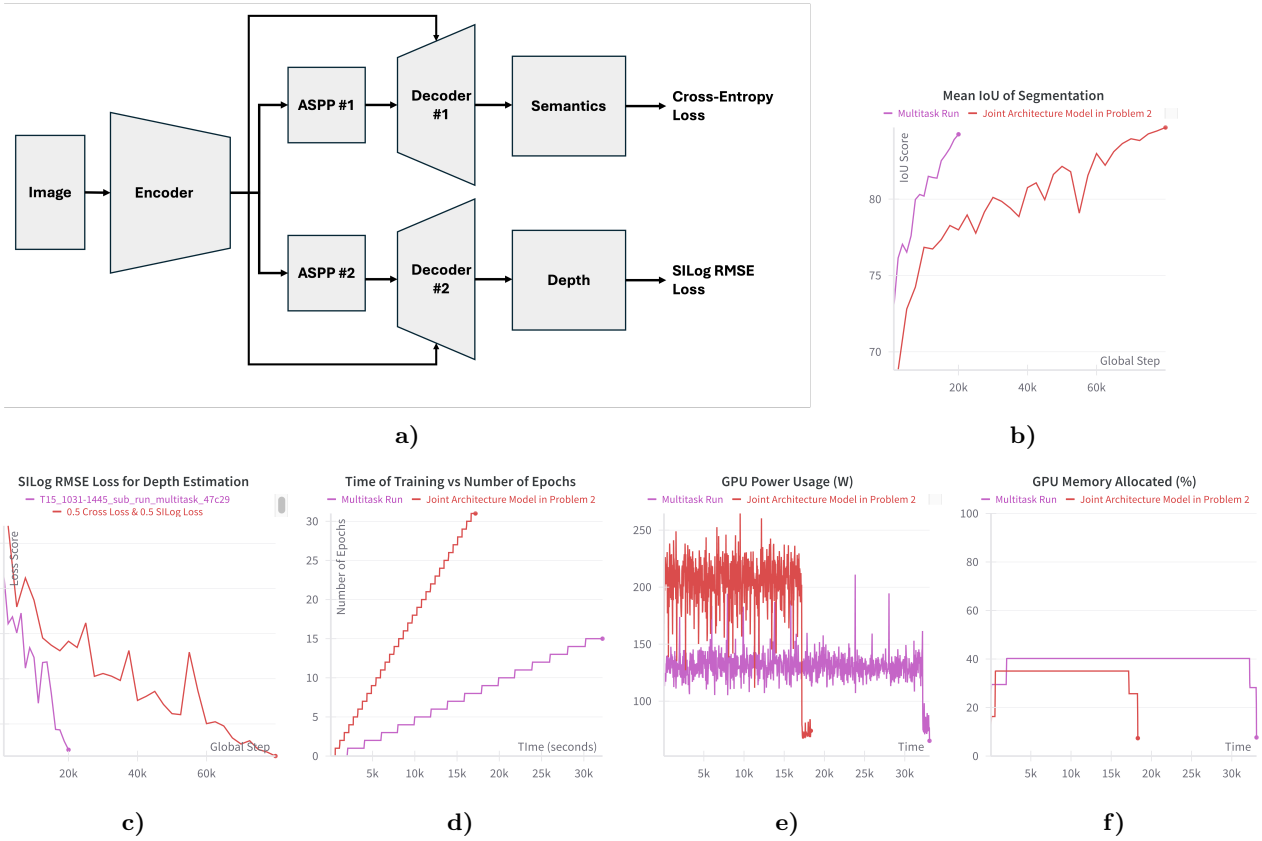
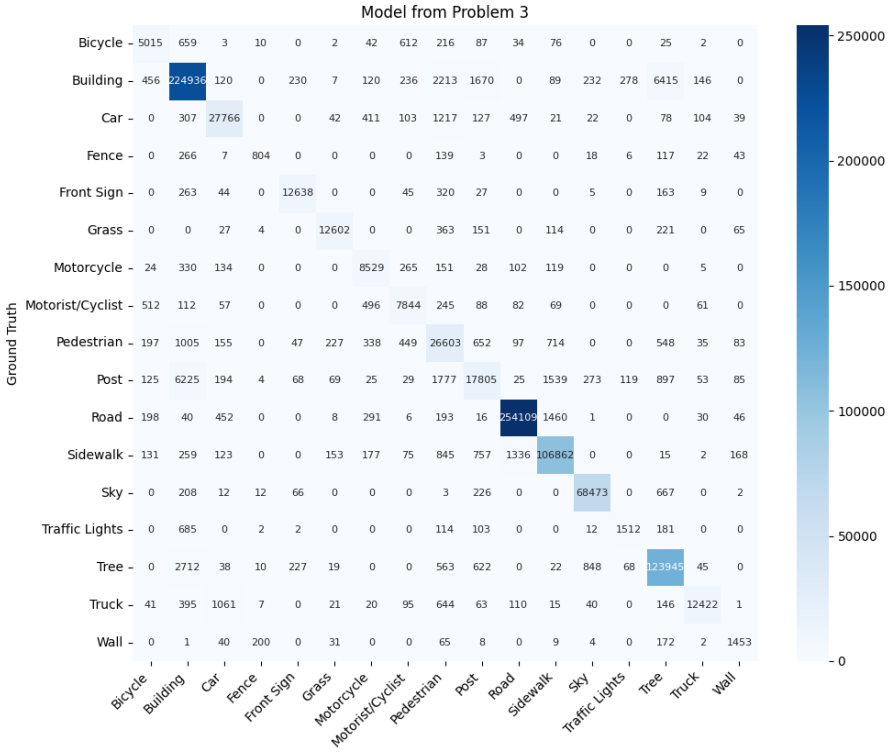
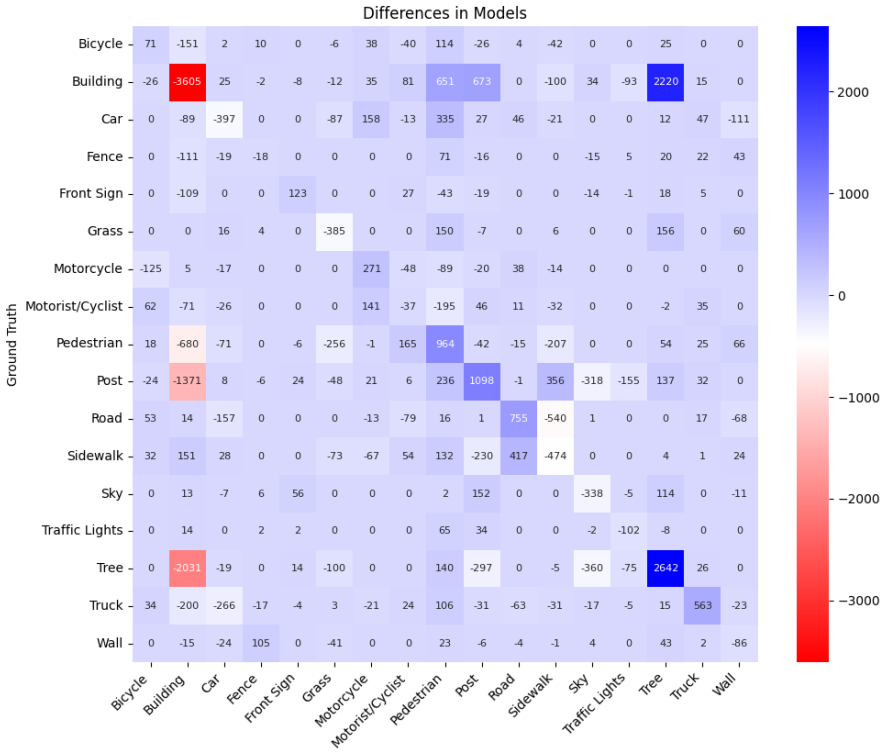


Figure 6: Branched Multi-Task Learning Architecture and Comparisons. a) Structure of the branched Multi-Task model. b)-f) Comparisons of joint and branched models with respect to the IoU metric, SILog metric, training time, GPU power comparison and GPU memory needed.



a)



b)

Figure 7: Branched Multi-Task Learning Comparisons Continued a) Confusion Matrix for Predicted labels and groud truth for multitask learning. b) Subtraction of confusion matrices of Figure 5.a) and 7.a). Significant improvements were made to the tree and post classes, though this came at a tradeoff with the building class.

Problem 4: Adaptive Depth Estimation

1 Attention Mechanism

a) Self Attention

i.) General Implementation To implement Self Attention we largely follow the instructions given in the task description and in the "TODO" comments within the code. The *SelfAttention* class code is given in Code Snippet 8 to have a better understanding of it. First, we normalize the input into the self attention class. This is also aligned with recent findings that suggest that the normalization layer before the multi-head attention is important for the query to attend to all keys equally [8]. Then we use three linear projections to $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$ each. Since we're using Multi-head attention we then reshape the three obtained tensors to include the number of heads H and the rescaled dimension $d = \frac{C}{H}$. The resulting shape of each tensor is $[B, H, N, d]$ with the goal of facilitating the parallel computation across multiple attention heads. Afterwards we calculate the similarity between the queries and keys and scale it by the normalization constant d_k and since by using multi-head attention we have each head treating a subpart of the input embeddings, we also need to rescale the normalization constant to $d_k = \frac{C}{H}$. We then retrieve the combination weights by multiplying the softmax of the similarity with the value tensor and apply a dropout layer. We then concatenate the output from all attention heads by using a permutation and reshaping operation [9]. We then project the result back to dimensionality D with a linear layer and return it from the SelfAttention class. The residual connection and the expansion are both handled in the provided TransformerBlock class.

ii.) Positional embedding We try two variants of positional embedding. In one we only add the positional embedding to the queries vector and in one we add it to the whole of the input vector x. The TODO asks us to add the positional embeddings only to the query vector. Adding it to the whole input vector seems to be more in line with approaches for transformers in general so we wanted to investigate this as well [10]. We initialize a *self.latents* learnable parameter inside the model with the size being the same as the channel dimensions chosen within the transformer layers. We then feed the parameter into the transformer layers as a positional embedding. We compare the two possibilities of adding these kind of positional embeddings in Figure 9.a) and Figure 9.b). Unfortunately, with this embedding scheme we don't find any visible benefit in the metrics or the loss as seen in the Figure. Additionally, adding this style of embedding to x before the linear projection layers reduces performance compared to adding the positional embedding to queries after the projection.

iii.) Dropout As a small addition we investigate the differences between using a dropout layer and not using one after computing the attention weights. Generally, dropout prevents over-fitting behavior and produces better learnability as well as more stable gradients [9]. As proof for our implementation we also use results from task 2. As seen in Figure 8.f) we produce better results using a dropout layer than without it.

model_parts.py

```python
# ... previous code ....
class SelfAttention(torch.nn.Module):
    def __init__(self, dim, num_heads=4, dropout=0.0):
        super().__init__()
        self.num_heads = num_heads
        self.out = nn.Linear(dim, dim)
        self.temperature = 1.0
        self.dropout = nn.Dropout(dropout)
        # TODO: Implement self attention, you need a projection and the normalization layer
        ### START OF OUR IMPLEMENTATION ###
        self.layer_norm = nn.LayerNorm(dim)
        # Projection layers for query, key, value
        self.query_proj = nn.Linear(dim, dim)
        self.key_proj = nn.Linear(dim, dim)
        self.value_proj = nn.Linear(dim, dim)
        ### END OF OUR IMPLEMENTATION ###

    def forward(self, x, pos_embed):
        """
        Pre-normalziation style self-attetion
        :param x: batched tokens with dimesion dim (B,N,C),
        :param pos_embed: batched positional with shape (B, N, C)
        :return: tensor processed with output shape same as input
        """
        B, N, C = x.shape
        # TODO: Implement self attention
        ### START OF OUR IMPLEMENTATION ###
        # Normalize input
        x = self.layer_norm(x)
        # Compute Q, K, V
        Q = self.query_proj(x)
        K = self.key_proj(x)
        V = self.value_proj(x)
        # Reshape for Multihead attention
        Q = Q.reshape(B, N, self.num_heads, C // self.num_heads)
        K = K.reshape(B, N, self.num_heads, C // self.num_heads)
        V = V.reshape(B, N, self.num_heads, C // self.num_heads)
        # Rearrange
        Q = Q.permute(0, 2, 1, 3) # B H N C/H
        K = K.permute(0, 2, 1, 3)
        V = V.permute(0, 2, 1, 3)
        if pos_embed is not None:
            pos_embed = pos_embed.expand(B, -1, -1)
            pos_embed = pos_embed.reshape(B, N, self.num_heads, C // self.num_heads)
            pos_embed = pos_embed.permute(0, 2, 1, 3) # B, H, N, C/H
            Q = Q + pos_embed
        # Dot product computation
```

```
        S = torch.matmul(Q, K.transpose(-2, -1))
        dk = C / self.num_heads
        S = S / (dk ** 0.5)
        # Apply Softmax
        attention = F.softmax(S, dim=-1)
        attention = self.dropout(attention) # TODO
        attention_output = torch.matmul(attention, V)
        attention_output = attention_output.permute(0, 2, 1, 3).reshape(B, N, C)
        # Final pass
        projected_output = self.out(attention_output)
        ### END OF OUR IMPLEMENTATION ###
        return projected_output
# ... rest of code ...
```

Code Snippet 8: Code for Self Attention

b) Theoretical Questions

i.) Scaling of Parameters: The overall time complexity of the attention model is determined by four key parameters: batch size (B), number of tokens in each sample input (N), dimensionality of each token (D), and the number of heads (H). Batch size refers to processing multiple input samples simultaneously, which allows the model to leverage parallel computation capabilities of GPUs effectively. Since each sample in the batch undergoes the same sequence of operations independently, the overall computational complexity scales linearly with the batch size. For simplicity, we will consider a single sample $\mathbf{X} \in \mathbb{R}^{N \times D}$ for a single head and analyze step be step. The initial LayerNorm computes mean and variance across each dimension and normalize the value, leads to complexity of $O(N \cdot D)$. Then the input is projected into $\boldsymbol{Q}$, $\boldsymbol{K}$, $\boldsymbol{V}$ through matrix multiplication, leading to complexity of $O(N \cdot D^2)$. The similarity is calculated through the product of $\boldsymbol{Q}$ and $\boldsymbol{K}$, which has a complexity of $O(N^2 \cdot D)$. The combination weight is the softmax of the result, which takes $O(N^2)$. The convex combination takes $O(N^2 \cdot D)$. The final projection and residual addition takes complexity of $O(N \cdot D^2 + N \cdot D)$ Overall, the complexity for a single input with a single head is $O(N \cdot D + N \cdot D^2 + N^2 \cdot D)$, which is $O(N^2 \cdot D)$ in practice ($N >> D$). Having a batch size B makes complexity $O(B \cdot N^2 \cdot D)$. Adding multiple heads is similar to decrease dimensionality and increase batch size ($\mathbb{R}^{B \times N \times D} \rightarrow \mathbb{R}^{(B \times H) \times N \times \frac{D}{H}}$), and the final computational complexity is $O((B \cdot H) \cdot N^2 \cdot \frac{D}{H}) = O(B \cdot N^2 \cdot D)$.

ii.) High Memory Requirements: To combat against the high memory requirements of the attention model, multiple solutions are possible. First of all, we can see that the high memory is primarily affected by the batch size as seen in Figure 8.e). Therefore, by lowering the batch size would drastically reduce the memory requirements but at the cost of a worse generalization and potentially worse convergence, increased training time as we also see in examples throughout our report. A second approach could be to use sparse attention [11], this method saves memory by computing attention only nearby or relevant tokens instead of computing attention over all tokens. This is naturally powerful in tasks where mostly local context is needed. A third method, is the linear attention [12], this approach replaces the softmax with a feature map-based method, enabling attention calculations to be performed independently across pairs of queries and keys [12]. This reduces the quadratic complexity of the similarity calculation to linear complexity. A fourth approach would be to scale up and/or out the hardware to fulfill these requirements if possible.

Module	Parameters
Encoder	11,176,512
Decoder	2,579,008
Transformer Blocks	2,369,280
Conv3x3	590,080
Bin Projection	257
Total	16,715,137

Table 3: Number of parameters for each model component

iii.) Factorized Attention Mechanism: The attention mechanism, particularly the computation of combination weights via the dot-product similarity and softmax, is challenging to factorize into more efficient operations due to its global dependency structure. Each token's attention weight depends on its relationship with every other token, creating an $N \times N$ matrix of interactions that captures all pairwise dependencies. This calculation inherently involves full matrix multiplications that are computationally and memory-intensive. Consequently, the dense, pairwise nature of attention computation limits factorization and demands high computational resources, especially as sequence length N grows [13].

2 Adaptive Depth

Including the Self Attention into an adaptive model structure to predict depth bins is the next logical step. For our implementation, we initially extract the latents using the provided LatentExtractor class. This class performs a bilinear downsampling and flattening operation on the input data. As input, we try three different features: the lowest features coming directly from the ResNet encoder, the predictions tensor coming from the decoder and the ASPP features coming from the ASPP. The features differ in distance to the final output and therefore the abstraction from the output task which is depth in this case. Taking features closer from the final output might be better suited to contain depth, while features further from the output such as the lowest features we expect to be better suited to contain global context.

a) Lowest Encoder Features

Initially, we work with the lowest encoder features contained within the output of the ResNet like encoder with a dilation in the last layer, as implemented in task a). The shape of the lowest features returned from the encoder is $[B, C, H, W]$ and the LatentExtractor reshapes it to $[B, N, C]$. The channel dimension is $C = 512$. We decide to either use the rest of the transformer architecture with 512 channels or downsize the channel dimension to 256. For the downsizing we consider a linear projection layer and a 1d convolutional layer. We emprically conclude that the linear projection leads to stable training and simplifies our implementation. If 512 channels are used within the transformer layers we need to perform a reprojection to 256 layers to compute the similarity between the queries and the learnable bins. We also use a linear layer for this reprojection. We implement the linear projection after we project the depth values because we want to use the most possible channels for a projection.

i.) Channel dimension The returned channel dimension for the lowest features from the latent extractor is 512. We investigate whether it is beneficial to train with this dimension or downsize to 256. One important factor in this comparison is the scaling factor d_k. Because we increase the channel dimension if we go from 256 to 512, the scaling factor $d_k = \frac{C}{H}$ increases for a constant number of heads. This leads to different rescaling of the similarity inside the Self Attention and thus influences the distribution returned by the Softmax in the model adaptive class. We compare three configurations:

1. 256 Channels with 4 heads, $d_k = 64$, $\sqrt{d_k} = 8$
2. 512 Channels with 4 heads, $d_k = 128$, $\sqrt{d_k} = 11.314$
3. 512 Channels with 8 heads, $d_k = 64$, $\sqrt{d_k} = 8$

We select 2. because of the same number of heads and 3. because of the same relative scaling with d_k. The results are listed in Figures 8.c) and 8.d). We investigate these for a batch size of 48 to compare quickly since we are interested in relative comparison. We deduct that 256 channels lead to better initial results, which is why we adapt it from here on. Additionally, it is apparent that 2. converges well in the second epoch as well, we however are interested in reducing the initial SILog-RMSE penalty as much as possible and therefore opt for 256. Also, achieving a similar result with a smaller hidden dimension is computationally preferred.

ii.) Position Embedding Now we investigate the effect of adding positional embeddings to the features that are fed into the transformer layers. We initialize a *self.latents* learnable parameter inside the model with the size being the same as the channel dimensions chosen within the transformer layers. We then feed the parameter into the transformer layers as a positional embedding. However, we don't see any added benefits from this as also seen in 9.a). We therefore continue on without additional position embedding. We recognize that adding positional embeddings can support the models spatial understanding due to the nature of transformer layers but for now we don't see any benefit in implementation. We assume that this style of positional embedding might interfere with features already encoded in the feature maps or our implementation might be incorrect. From here on we proceed without positional embeddings.

iii.) Hyperparameter tuning We further investigate the effect of modifying the hyperparameters of the adaptive depth model with lowest features. We investigate the effect of changing the number of heads in the attention mechanism and the expansion parameter. The effect can be seen in Figures 9.e) and 9.f). Using a higher expansion of 8 leads to worse outcomes unless the number of heads is also increased as visible in the Figures. However, we do not find a distinctive benefit of using a higher expansion and therefore stay with an expansion of 4. Additionally, it is important to mention that we experienced higher training stability with a batch size of 48 conducting the above experiments. The additional benefit of shortened training time proved important during troubleshooting as well. This follows our investigations of the effects of batch size in Problem 1 and is the reason most runs are conducted using a batch size of 48.

b) Decoder Features

Now we investigate another set of input features for extracting the latents. As mentioned we now use the predictions returned from the decoder. This set of features is very close to the output depth space. The shape of the predictions is $[B, C, H, W]$ and the Latent Extractor returns $[B, N, C]$ where $N = 256$ and $C = 256$. Preserving this channel dimension of 256 we run a comparison with the same hyperparameters to the lowest encoder features.

i.) Comparison to lowest features The comparison between using the lowest features and the predictions can be seen in Figure 9.c). We conclude that the lowest level features still provide the best SILog-RMSE score compared to the other features used as input. We further investigate the ASPP bottleneck features attempting to utilize their semantic scene understanding.

c) ASPP bottleneck features

i.) Comparison to lowest features Using the ASPP features does not lead to improvements over using

lowest features or decoder features for depth prediction after one epoch as seen in Figure 9.c). We hypothesize that while ASPP features are rich in semantic understanding, they may lack the spatial detail necessary for accurate depth prediction in early training stages. We try a different combination of hyperparameters to improve the performance.

ii.) ASPP hyperparameter adjustments We increase the number of heads to 8, we increase the number of layers to 6, do a combination of both and try with a batch size of 16. We plot the results in Figure 9.g). We can deduct that a smaller batch size improves the performance. Additionally, we can see that increasing the number of heads to 8 improves the performance. We hypothesize thath increasing the number of heads likely allows the model to capture finer-grained spatial relationships in the ASPP features, which are more abstract and semantically rich. Increasing the expansion size while also adding more heads which did deliver acceptable performance for the lowest features does not increase performance here. Adding more transformer layers additionally increases performance, such that we see the best performance for a run with 6 transformer blocks and 8 heads for the aspp features. We conclude that this is the best configuration for using the aspp featuers and use them from here on.

d) Adaptive Depth Conclusion

We conclude that both the lowest features from the encoder as well as the ASPP bottleneck features from the ASPP blocks provide a reasonable input to the transformer blocks. Using the ASPP features offers more sematic understanding of the scene but it adds computational complexity due to adding the ASPP blocks and their trainable parameters. We don't see any improvements with positional embeddings but acknowledge that we would expect them given the models need for spatial understanding. If time permitted we would have liked to explore more positional embedding options. For scenarios where computational resources are limited, lowest encoder features offer a strong balance of performance and efficiency. However, when computational cost is not a constraint, ASPP features with 8 heads and 6 layers provide superior basis for depth prediction tasks.

model_adaptive_depth.py

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from source.models.model_parts import Encoder, get_encoder_channel_counts, DecoderDeeplabV3p, TransformerBlock, LatentsExtractor, ASPP
torch.autograd.set_detect_anomaly(True)

class ModelAdaptiveDepth(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),
        )
        ch_out_encoder_bottleneck, ch_out_encoder_4x = get_encoder_channel_counts(cfg.model_encoder_name)
        self.decoder = DecoderDeeplabV3p(ch_out_encoder_bottleneck, ch_out_encoder_4x, 256)
        self.conv3x3 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)
        self.dimension = 256
        self.pos_embed = False
        self.latents = nn.Parameter(torch.randn(1, cfg.num_bins, self.dimension) * 0.02, requires_grad=True)
        ### START OUR IMPLEMENTATION ###
        self.latents_extractor = LatentsExtractor(num_latents=cfg.num_bins)
        self.transformer_blocks = nn.ModuleList(
            [TransformerBlock(dim=self.dimension, num_heads=cfg.num_heads, expansion=cfg.expansion) for _ in range(cfg.num_transformer_layers)]
        )
        self.bin_proj = nn.Linear(self.dimension, 1)
        self.base_features = "features_lowest"
        if self.base_features == "ASPP":
            self.aspp =  ASPP(ch_out_encoder_bottleneck, 256)
        self.channel_reducer_ind = "Linear"
        if self.channel_reducer_ind == "Linear":
            self.channel_reducer = nn.Linear(512, self.dimension)
        elif self.channel_reducer_ind == "Conv":
            self.channel_reducer = nn.Conv1d(in_channels=512, out_channels=self.dimension, kernel_size=1, stride=1)
        if self.dimension == 512:
            self.latent_proj_after_transformer = nn.Linear(512, 256)
        ### END OUR IMPLEMENTATION ###

    def forward(self, x):
        B, _, H, W = x.shape
        input_resolution = (H, W)
        features = self.encoder(x)
        lowest_scale = max(features.keys())
        features_lowest = features[lowest_scale]
        features_4x, _ = self.decoder(features_lowest, features[4])
        queries = self.conv3x3(features_4x)
        ### START OUR IMPLEMENTATION ###
        if self.base_features == "predictions":
            learnable_bins = self.latents_extractor(features_4x)
        if self.base_features == "ASPP":
            aspp_features = self.aspp(features_lowest)
            learnable_bins = self.latents_extractor(aspp_features)
        if self.base_features == "features_lowest":
```

```
            if self.channel_reducer_ind == "Linear":
                learnable_bins = self.latents_extractor(features_lowest)
                learnable_bins = self.channel_reducer(learnable_bins)
            elif self.channel_reducer_ind == "Conv":
                learnable_bins = self.latents_extractor(features_lowest)
                learnable_bins = learnable_bins.permute(0, 2, 1)
                learnable_bins_reduced = self.channel_reducer(learnable_bins)
                learnable_bins_reduced = learnable_bins_reduced.permute(0, 2, 1)
                learnable_bins = learnable_bins_reduced
            elif self.channel_reducer_ind == "No":
                learnable_bins = self.latents_extractor(features_lowest)
        if self.pos_embed:
            for idx, transformer_block in enumerate(self.transformer_blocks):
                learnable_bins = transformer_block(learnable_bins, self.latents)
        else:
            for idx, transformer_block in enumerate(self.transformer_blocks):
                learnable_bins = transformer_block(learnable_bins)
        projected_d = self.bin_proj(learnable_bins) # torch.Size([16, 256, 1]) B N 1
        projected_d = projected_d.squeeze(-1)
        if self.base_features == "features_lowest":
            if self.dimension == 512:
                learnable_bins = self.latent_proj_after_transformer(learnable_bins)
        similarity = torch.einsum("b c h w, b n c -> b n h w", queries, learnable_bins)
        probabilities = F.softmax(similarity, dim=1)
        projected_d = projected_d[:, :, None, None]
        final_depth = (probabilities * projected_d).sum(dim=1, keepdim=True)
        predictions_1x = F.interpolate(final_depth, size=input_resolution, mode='bilinear', align_corners=False)
        ### END OUR IMPLEMENTATION ###
        out = {}
        offset = 0
        for task, num_ch in self.outputs_desc.items():
            current_precition = predictions_1x[:, offset:offset+num_ch, :, :]
            out[task] = current_precition
            offset += num_ch
        return out
```

Code Snippet 9: Code for Adaptive Depth Model

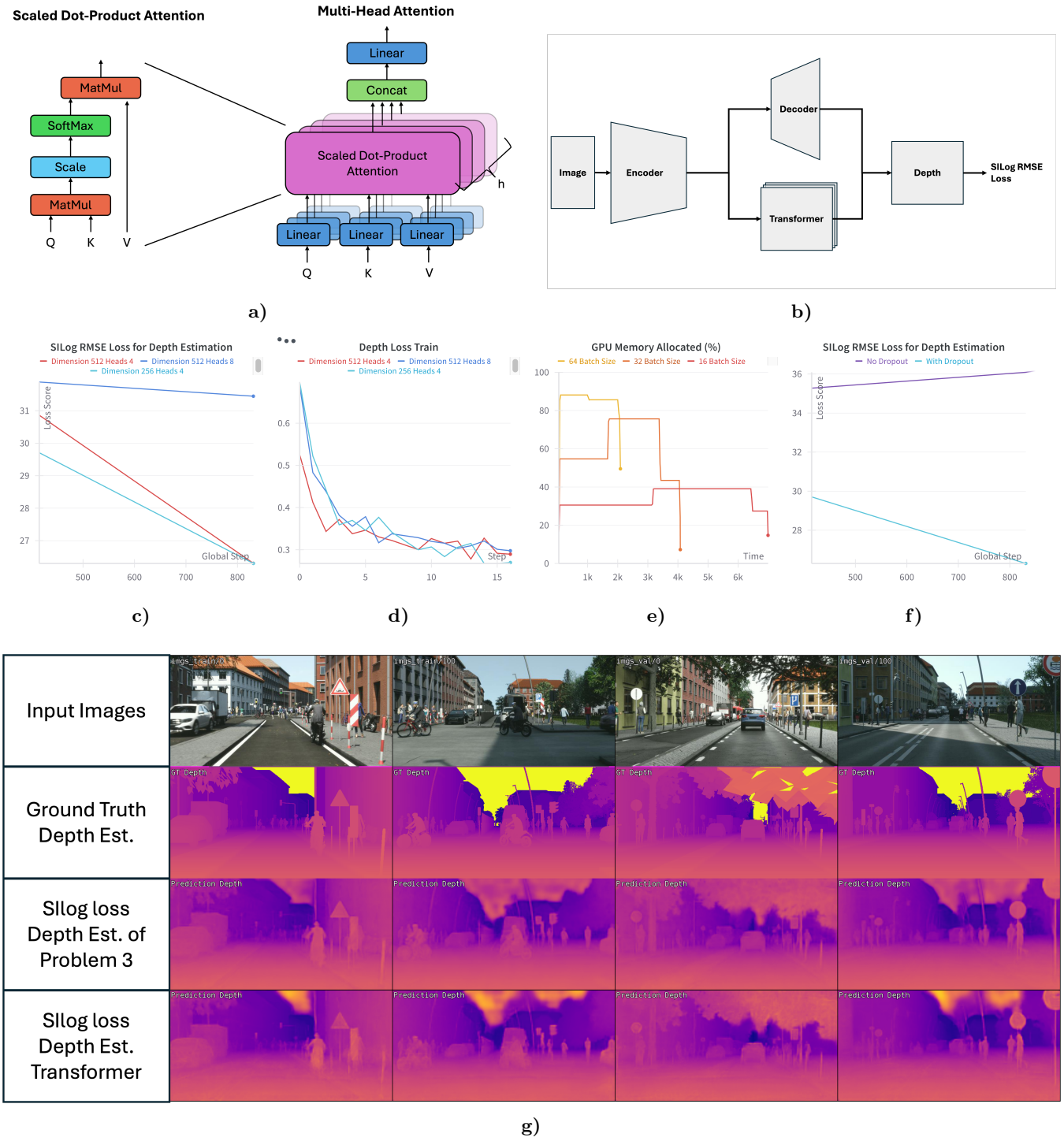
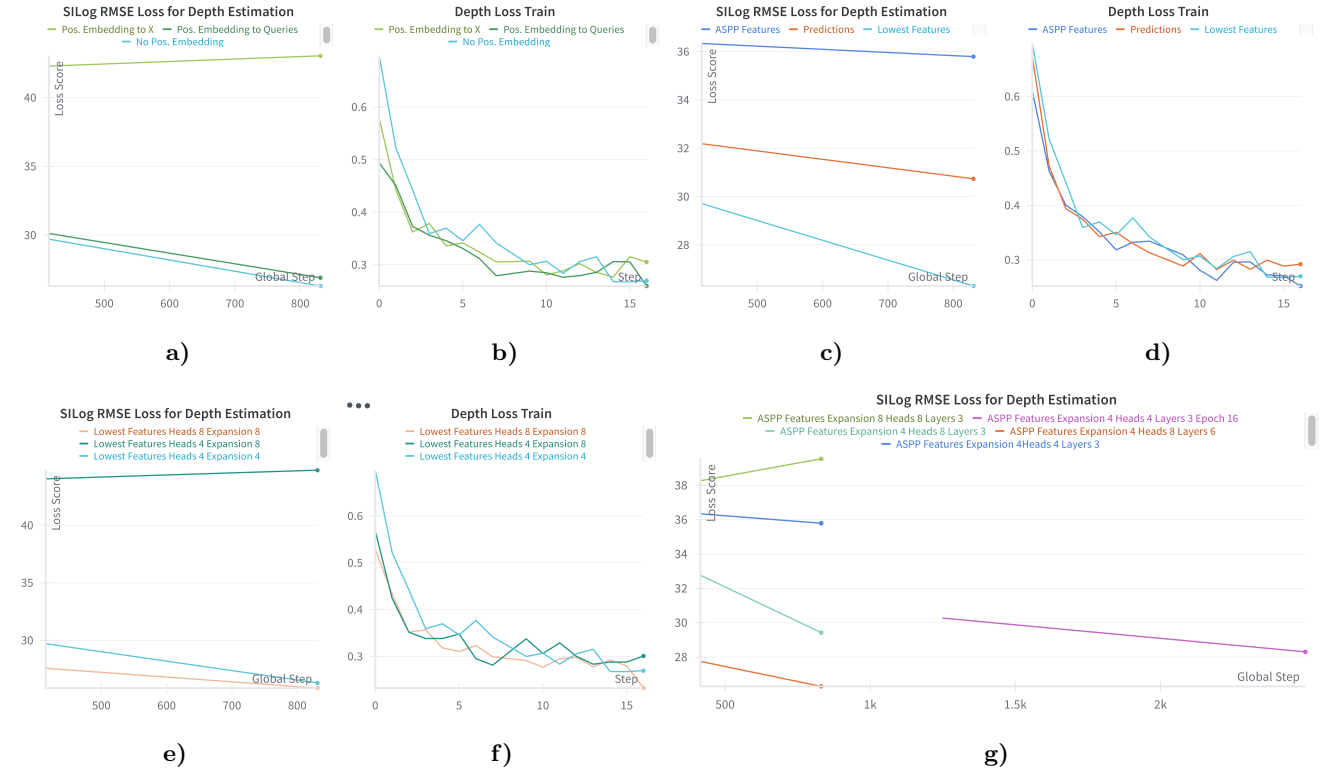


Figure 8: Adaptive Depth Estimation a) The structure of Multi-Head Attention mechanism. b) The structure of adaptive depth estimator model. c)-e) Comparisons of different number of heads and output dimensions with respect to SILog RMSE metric, traing depth loss and GPU memory needed. f) Comparison of model performances between with no dropout layer and with dropout models g) Comparisons of Depth Prediction results on different images.



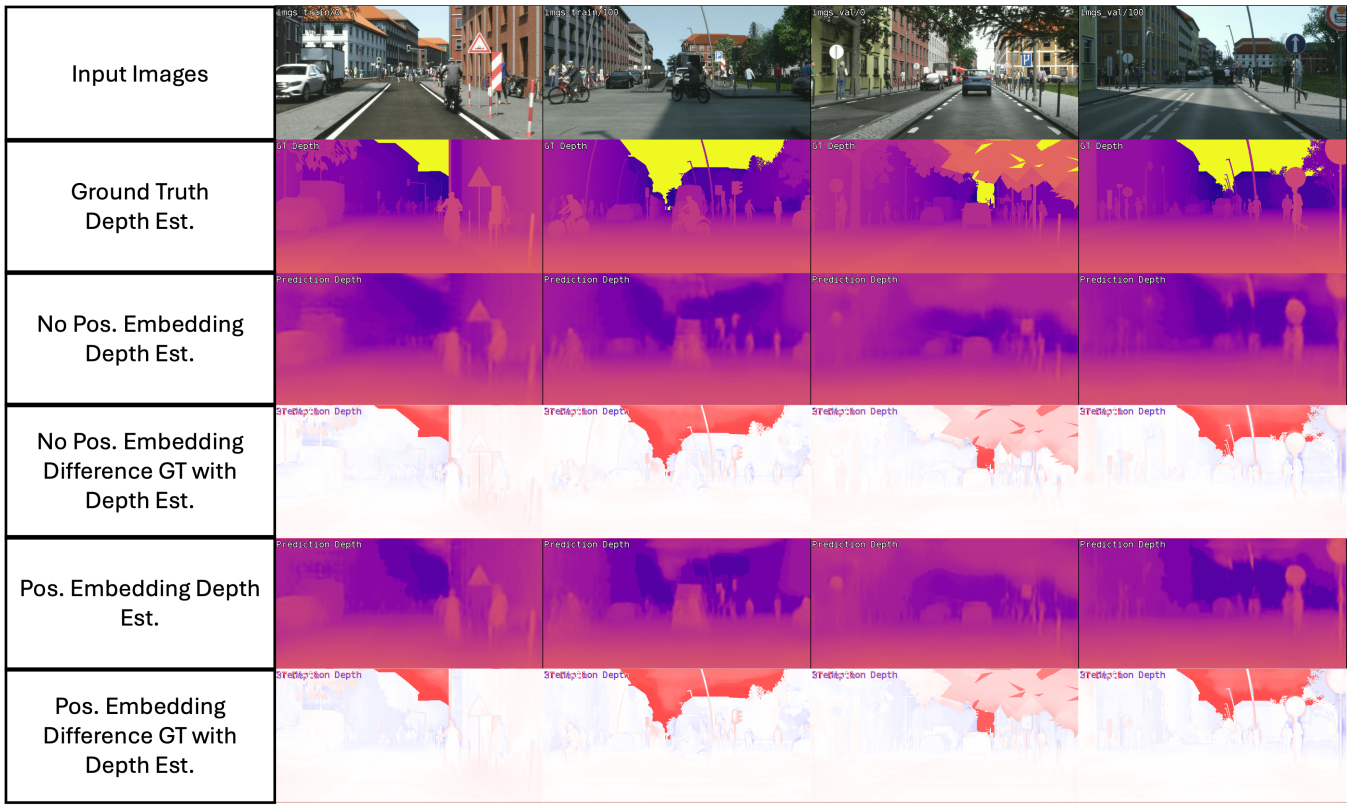


Figure 9: Adaptive Depth Estimation Continued a)-f) Comparisons of SILog RMSE metric and training depth loss for different positional embedding methods, features and hyperparameters. g) SILog RMSE metric for different combinations. h) Comparisons of Depth Prediction results for different embeddings.

Problem 5: Open Challenge

1 Other improvements

Throughout the implementation of the other tasks we explored various ways of improving the architectures of other networks. Since we're unsure about the scale of what is required in task 5, we would like to mention these improvements here as well to indicate our efforts to improve networks. One improvement we investigated are the amount of additional 3x3 convolutional layers in the decoder in ASPP and Skip Connections. Additionally, we also investigated the amount of channels ideally added in the skip connections part in ASPP and Skip Connections. These small changes improve the performance of the multitask model. In addition, we investigated various input features to the self attention mechanism in 2 and discuss the performance.

2 MultiTask Transformer Model

However, these changes are rather incremental so we explore a symmetrical multitask model in which both tasks, segmentation and depth prediction are split after the encoder, equivalent to problem 4 but both use attention mechanisms. A model architecture is shown in Figure 10.a). We wrote a new file called *model_transformer_multitask.py* which we included as callable parameter in the *config.py* and the *__init__.py*. We employ various approaches to producing effective multitask outputs.

a) Self Attention

Due to the nature of the semantic segmentation task self attention requires high memory because it scales quadratically as we discussed in 4.1.b) and semantic segmenation is a dense prediction task where we want to predict the class of each pixel. As in task 2 we have to downsample our input features. We achieve this using the same LatentExtractor class since its operations work for both and we also stay with 256 as number of latents. As we have previously seen, increasing the number of latents beyond this leads to computational issues without adding too many benefits. We perform the same depth calculation using the aspp features as input. We choose the aspp features due to their semantic understanding which are important for semantic segmentation. The aspp features are now also fed through a number of transformer blocks with self attention and then converted into a segmentation map using the previously implemented decoder. Using the pre-implemented decoder here still ensures that our approach is Transformer based but utilizes the strengths of convolutional layers after the transformer layers like many other implementations do [14]. The implementation of our new model can be seen in Code Snippet 10. With a batch size of 48 and after two epochs we achieve a mIoU of 70.96 and a SILog-RMSE of 25.23. While this is lower than the initial scores of the Multitask architectures of Problem 3 and 4, this has a lower number of global steps. This result shows that our model is functional and reaches slightly worse performance for a lower number of global steps. We show this in Figures 10.b) and 10.c). We observe a very slight mIoU improvement for one Self attention run over the two baseline runs for the non-Transformer based models in the Multitask setting, for the same number of global steps. However, something went wrong during this run with our SILog-RMSE loss which continues to be high. Also due to the batch size difference the transformer based run reports scores after fewer global steps than the other two runs. Changing to a batch size of 16 does decrease performance here potentially due to instability in the gradient updates. We unfortunately ran out of time in pinpointing this issue. We hypothesize that the higher SILog-RMSE could result from instability in attention weights during training, compounded by the smaller batch size which is more prone to instability during gradient updates.

model_transformer_multitask.py

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np
import os
from source.models.model_parts import Encoder, get_encoder_channel_counts, DecoderDeeplabV3p, TransformerBlock, LatentsExtractor, ASPP, CrossTransformerBlock
torch.autograd.set_detect_anomaly(True)
class ModelAdaptiveMultitask(torch.nn.Module):
    def __init__(self, cfg, outputs_desc):
        super().__init__()
        self.outputs_desc = outputs_desc
        ch_out = sum(outputs_desc.values())
        self.encoder_shared = Encoder(
            cfg.model_encoder_name,
            pretrained=cfg.pretrained,
            zero_init_residual=True,
            replace_stride_with_dilation=(False, False, True),
        )
        ch_out_encoder_bottleneck, ch_out_encoder_4x = get_encoder_channel_counts(cfg.model_encoder_name)
        self.aspps = nn.ModuleDict({
            'aspp_seg': ASPP(ch_out_encoder_bottleneck, 256),
            'aspp_depth': ASPP(ch_out_encoder_bottleneck, 256)
        })
        self.decoders = nn.ModuleDict({
            'decoder_depth': DecoderDeeplabV3p(ch_out_encoder_bottleneck, ch_out_encoder_4x, 256),
            'decoder_seg': DecoderDeeplabV3p(256, ch_out_encoder_4x, self.outputs_desc['semseg'])
```

```
        })
        self.dimension_seg = 256
        num_classes = self.outputs_desc['semseg']
        #self.class_tokens =
        ↪  nn.Parameter(torch.randn(num_classes,
        ↪  self.semseg_dimension))
        self.transformer_blocks_seg = nn.ModuleList(
            [TransformerBlock(dim=self.dimension_seg,
            ↪  num_heads=cfg.num_heads,
            ↪  expansion=cfg.expansion) for _ in
            ↪  range(cfg.num_transformer_layers)]
        )
        self.dimension_depth = 256
        self.conv3x3_depth = nn.Conv2d(256, 256,
        ↪  kernel_size=3, stride=1, padding=1)
        self.latents_extractor =
        ↪  LatentsExtractor(num_latents=cfg.num_bins)
        self.transformer_blocks_depth = nn.ModuleList(
            [TransformerBlock(dim=self.dimension_depth,
            ↪  num_heads=cfg.num_heads,
            ↪  expansion=cfg.expansion) for _ in
            ↪  range(cfg.num_transformer_layers)]
        )
        self.bin_proj_depth =
        ↪  nn.Linear(self.dimension_depth, 1)
        self.base_features = "ASPP" # Other choices:
        ↪  features from features_lowest, predictions,
        ↪  ASPP
        if self.base_features == "features_lowest":
            self.channel_reducer = nn.Linear(512,
            ↪  self.dimension_depth)

    def forward(self, x):
        # Output things
        out = {}
        offset = 0
        B, _, H, W = x.shape
        input_resolution = (H, W)
        features = self.encoder_shared(x)
        lowest_scale = max(features.keys())
        features_lowest = features[lowest_scale]
        features_4x, _ =
        ↪  self.decoders['decoder_depth'](features_lowest,
        ↪  features[4])
        queries = self.conv3x3_depth(features_4x)
        if self.base_features == "ASPP":
            aspp_features =
            ↪  self.aspps['aspp_depth'](features_lowest)
            learnable_bins =
            ↪  self.latents_extractor(aspp_features)
        if self.base_features == "features_lowest":
            learnable_bins =
            ↪  self.latents_extractor(features_lowest)
            ↪  # [B, 256, 512]
            learnable_bins =
            ↪  self.channel_reducer(learnable_bins)
        for idx, transformer_block in
        ↪  enumerate(self.transformer_blocks_depth):
            learnable_bins =
            ↪  transformer_block(learnable_bins)
        projected_d = self.bin_proj_depth(learnable_bins)
        ↪  # torch.Size([16, 256, 1]) B N 1
        projected_d = projected_d.squeeze(-1) #
        ↪  torch.Size([16, 256]) B N
        similarity = torch.einsum("b c h w, b n c -> b n
        ↪  h w", queries, learnable_bins)
        probabilities = F.softmax(similarity, dim=1)
        projected_d = projected_d[:, :, None, None]
        final_depth = (probabilities *
        ↪  projected_d).sum(dim=1, keepdim=True)
        depth_predictions_1x = F.interpolate(final_depth,
        ↪  size=input_resolution, mode='bilinear',
        ↪  align_corners=False)
        seg_features =
        ↪  self.aspps['aspp_seg'](features_lowest)
        B, C, H, W = seg_features.shape
        seg_features_flat =
        ↪  self.latents_extractor(seg_features)
        for transformer_block in
        ↪  self.transformer_blocks_seg:
            seg_features_flat =
            ↪  transformer_block(seg_features_flat)
        edge = self.latents_extractor.edge
        seg_features_transformed =
        ↪  seg_features_flat.view(B, edge, edge,
        ↪  C).permute(0, 3, 1, 2)  # Shape: [B, C, edge,
        ↪  edge]
        decoder_input_size = features[4].shape[2:] #TODO:
        ↪  Check the sizes here
        seg_features_upsampled =
        ↪  F.interpolate(seg_features_transformed,
        ↪  size=decoder_input_size, mode='bilinear',
        ↪  align_corners=False)
        seg_predictions_4x, _ =
        ↪  self.decoders['decoder_seg'](seg_features_upsampled,
        ↪  features[4])
        seg_predictions_1x =
        ↪  F.interpolate(seg_predictions_4x,
        ↪  size=input_resolution, mode='bilinear',
        ↪  align_corners=False)
        for task, num_ch in self.outputs_desc.items():
            if task == 'semseg':
                current_prediction = seg_predictions_1x
                ↪  # [B, num_classes, H, W]
            elif task == 'depth':
                current_prediction = depth_predictions_1x
                ↪  # [B, 1, H, W]
            else:
                raise ValueError(f"Unknown task: {task}")
            out[task] = current_prediction
        return out
```

Code Snippet 10: Code for Self Attention Model

b) Cross Attention in Segmentation

We also explore a cross attention architecture to combine the rich global understanding we have from the lowest features and the rich semantic information we have because of the ASPP modules. We simply adapt from the Self Attention implementation we did in Task 2 and some information gathered online [15] and our adaption can be seen in Code Snippet 11. The main difference to the self attention is naturally that the queries don't attend to themselves but rather to another input. We therefore also employ a separate normalization layer for keys and queries. We also include the dropout layer here that we introduced in the Self Attention class and that improved our implementation as we show in 2. Our general understanding is to use a higher spatial resolution feature map for the queries in combination with a high semantic resolution feature map for the keys and values. We try to use a randomly initialized learnable parameter *class tokens* with shape $[num_{class}, 256]$ as queries and the combined features that are returned from the decoder after the aspp as keys and values. While this leads to mediocre mIoU performance of 43 after 3 Epochs, the output semantic maps are still quite pixelated likely due to the upscaling and potential model inaccuracies as seen in Figure 10.d). We also tried to increase the channel dimension and thus the dimension of the class tokens to 512 to investigate possible improvements. For this architecture we project the ASPP combined features to channel dimension 512 with a linear layer. However, we ran out of memory here by 1.42 GiB for 3 transformer layers. We decrease to 2 transformer layers and a smaller batch size of 32 and run the model again. It is apparent

to us here that this strategy is also extremely time and space inefficient and we abandon it for the self attention strategy. We believe using cross attention can lead to good results for segmentation tasks but we were unable to achieve a good model architecture in time. We therefore further pursue the self attention for multitask as mentioned above and just include this to show the avenues explored [16].

c) Conclusion Problem 5

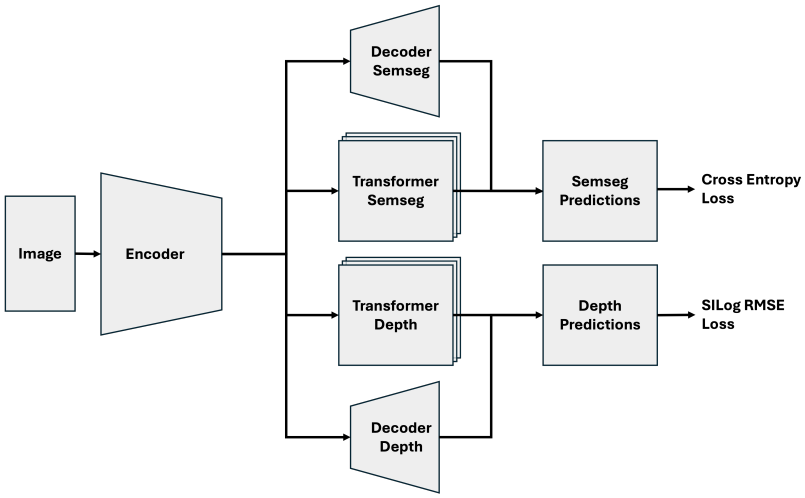
We implement a symmetric model as in Problem 4 and use self attention for the semantic segmentation task as well. Unfortunately, this does barely lead to an improvement in mIoU score and the joint performance is worse as seen in the Figures 10.b) and 10.c). We also try cross attention with the idea of combining the strengths of different feature maps through this attention mechanism. Unfortunately, this shows poor performance and we did not have the time to fully solve all issues related to this model architecture. However, we still think that we offer interesting changes to the model architectures and show strong efforts in improving over the baselines. Running the self attention model through the grader achieves 51.36 overall, 74.03 on segmentation and 22.67 on SILog-RMSE.

model_transformer_multitask_cross.py (old)

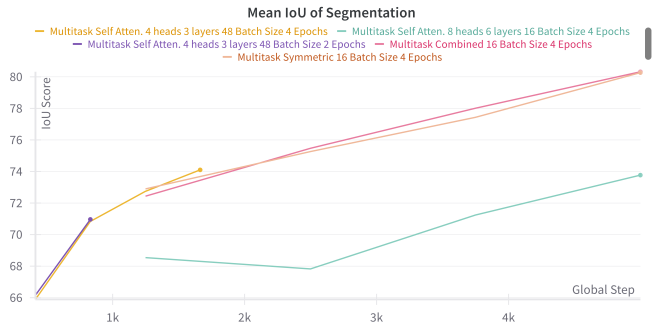
```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np
import os
from source.models.model_parts import Encoder, get_encoder_channel_counts, DecoderDeeplabV3p, TransformerBlock, LatentsExtractor, ASPP, CrossTransformerBlock
torch.autograd.set_detect_anomaly(True)
def save_heatmap(data, title="Heatmap", cmap='viridis', output_dir="heatmaps"):
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)
    plt.figure(figsize=(10, 10))
    plt.imshow(data, cmap=cmap)
    plt.title(title)
    plt.colorbar()
    file_path = os.path.join(output_dir, f"{title.replace(' ', '_')}.png")
    plt.savefig(file_path)
    plt.close()
class ModelAdaptiveMultitask(torch.nn.Module):
    def __init__(self, cfg, outputs_desc):
        super().__init__()
        self.outputs_desc = outputs_desc
        ch_out = sum(outputs_desc.values())
        self.encoder_shared = Encoder(
            cfg.model_encoder_name,
            pretrained=cfg.pretrained,
            zero_init_residual=True,
            replace_stride_with_dilation=(False, False, True),
        )
        ch_out_encoder_bottleneck, ch_out_encoder_4x = get_encoder_channel_counts(cfg.model_encoder_name)
        self.aspps = nn.ModuleDict({
            'aspp_seg': ASPP(ch_out_encoder_bottleneck, 256)
        })
        self.decoders = nn.ModuleDict({
            'decoder_depth': DecoderDeeplabV3p(ch_out_encoder_bottleneck, ch_out_encoder_4x, 256),
            'decoder_seg': DecoderDeeplabV3p(256, ch_out_encoder_4x, self.outputs_desc['semseg'])
        })
        self.semseg_dimension = 256
        num_classes = self.outputs_desc['semseg']
        self.class_tokens = nn.Parameter(torch.randn(num_classes, self.semseg_dimension))  # Initialize class tokens
        self.transformer_blocks_seg = nn.ModuleList(

            [CrossTransformerBlock(dim=self.semseg_dimension, num_heads=cfg.num_heads, expansion=cfg.expansion)
             for _ in range(cfg.num_transformer_layers)]
        )
        self.dimension_depth = 256
        self.conv3x3_depth = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1)
        self.latents_extractor = LatentsExtractor(num_latents=cfg.num_bins)
        self.transformer_blocks_depth = nn.ModuleList(
            [TransformerBlock(dim=self.dimension_depth, num_heads=cfg.num_heads, expansion=cfg.expansion) for _ in range(cfg.num_transformer_layers)]
        )
        self.bin_proj_depth = nn.Linear(self.dimension_depth, 1)
        self.channel_reducer = nn.Linear(512, self.dimension_depth)

```

```
    def forward(self, x):
        out = {}
        offset = 0

        # Shared encoder
        B, _, H, W = x.shape
        input_resolution = (H, W)
        features = self.encoder_shared(x)
        lowest_scale = max(features.keys())
        features_lowest = features[lowest_scale]
        features_4x, _ = self.decoders['decoder_depth'](features_lowest, features[4])
        queries = self.conv3x3_depth(features_4x)
        learnable_bins = self.latents_extractor(features_lowest) # x_down_flat shape:  torch.Size([16, 256, 512]) B HW=N C
        learnable_bins = self.channel_reducer(learnable_bins)
        for idx, transformer_block in enumerate(self.transformer_blocks_depth):
            learnable_bins = transformer_block(learnable_bins)
        projected_d = self.bin_proj_depth(learnable_bins) # torch.Size([16, 256, 1]) B N 1
        projected_d = projected_d.squeeze(-1) # torch.Size([16, 256]) B N
        similarity = torch.einsum("b c h w, b n c -> b n h w", queries, learnable_bins)
        probabilities = F.softmax(similarity, dim=1)
        projected_d = projected_d[:, :, None, None]
        final_depth = (probabilities * projected_d).sum(dim=1, keepdim=True)
        depth_predictions_1x = F.interpolate(final_depth, size=input_resolution, mode='bilinear', align_corners=False)
        seg_features = self.aspps['aspp_seg'](features_lowest)
        seg_predictions_4x, seg_features_queries = self.decoders['decoder_seg'](seg_features, features[4])
        B, C, H, W = seg_features_queries.shape
        N_kv = H * W
        seg_features_flat = seg_features_queries.view(B, C, N_kv).permute(0, 2, 1)  # [B, N_kv, C]
        class_tokens = self.class_tokens.unsqueeze(0).expand(B, -1, -1)  # [B, num_classes, C]
        #print("class_tokens.shape", class_tokens.shape)
        N_q = class_tokens.shape[1]
        for transformer_block in self.transformer_blocks_seg:
            class_tokens = transformer_block(class_tokens, seg_features_flat, seg_features_flat)
        class_tokens_out = class_tokens
        print("class_tokens_out shape: ", class_tokens_out.shape)
        class_tokens_expanded = class_tokens_out.unsqueeze(-1).unsqueeze(-1) # [B, num_classes, C, 1, 1]
        similarity = torch.einsum("bkc,bchw->bkhw", class_tokens_out, seg_features_queries)  # [B, num_classes, H, W]
        seg_logits = similarity
        seg_probs = F.softmax(seg_logits, dim=1)
        with torch.no_grad():
            sample_similarity_seg = similarity[0].cpu().numpy()
            for i in range(min(5, sample_similarity_seg.shape[0])):
                save_heatmap(sample_similarity_seg[i], title=f"Segmentation_Similarity_Map_Class_{i}")
            sample_probs_seg = seg_probs[0].cpu().numpy()
            for i in range(min(5, sample_probs_seg.shape[0])):
                save_heatmap(sample_probs_seg[i], title=f"Segmentation_Probability_Distribution_Cla
        seg_predictions_1x = F.interpolate(seg_probs, size=input_resolution, mode='bilinear', align_corners=False)
        for task, num_ch in self.outputs_desc.items():
            if task == 'semseg':
                current_prediction = seg_predictions_1x # [B, num_classes, H, W]
            elif task == 'depth':
                current_prediction = depth_predictions_1x  # [B, 1, H, W]
                # TODO: Clamping yes no?
            else:
                raise ValueError(f"Unknown task: {task}")
            out[task] = current_prediction
        return out
```

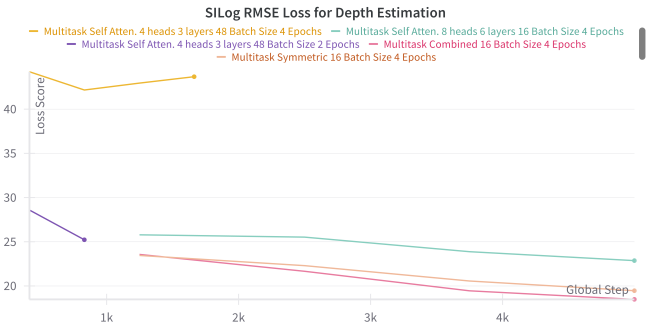
Code Snippet 11: Code for Cross Attention



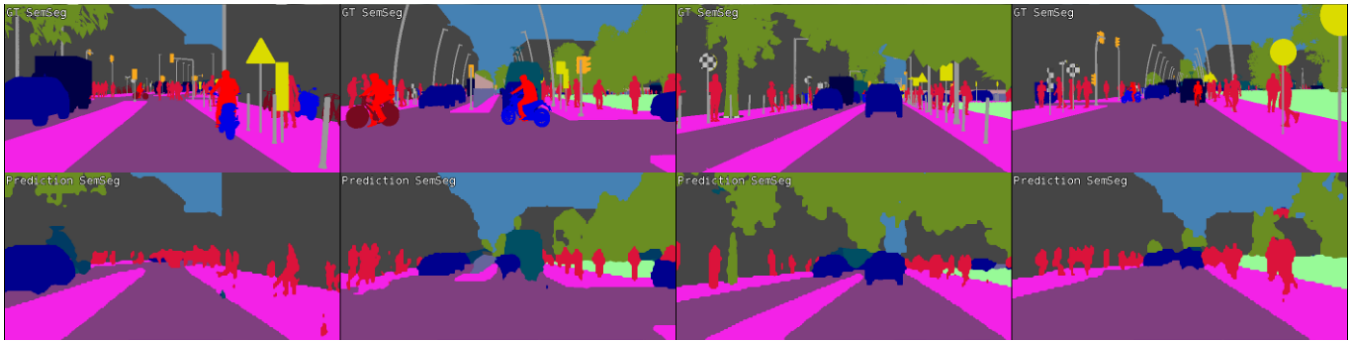
a)



b)



c)



d)

Figure 10: Open Challenge a) Structure of the branched MultiTask transformer model. b) Mean IoU metrics of different self-attention setups. c) SILog RMSE metrics of different self-attention setups. d) Predictions of the Cross Attention model

Bibliography

[1] Sebastian Ruder. An overview of gradient descent optimization algorithms, June 2017. arXiv:1609.04747.

[2] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, January 2017. arXiv:1412.6980.

[3] Introduction to Machine Learning (2023) | Learning & Adaptive Systems Group.

[4] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, March 2015. arXiv:1502.03167.

[5] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How Does Batch Normalization Help Optimization?, April 2019. arXiv:1805.11604.

[6] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation, August 2018. arXiv:1802.02611.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, December 2015. arXiv:1512.03385.

[8] Shaked Brody, Uri Alon, and Eran Yahav. On the Expressivity Role of LayerNorm in Transformers' Attention, May 2023. arXiv:2305.02582.

[9] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors, July 2012. arXiv:1207.0580.

[10] Xin Zhou, Zhaohui Ren, Shihua Zhou, Zeyu Jiang, TianZhuang Yu, and Hengfa Luo. Rethinking Position Embedding Methods in the Transformer Architecture. *Neural Processing Letters*, 56(2):41, February 2024.

[11] Chao Lou, Zixia Jia, Zilong Zheng, and Kewei Tu. Sparser is Faster and Less is More: Efficient Sparse Attention for Long-Range Transformers, June 2024. arXiv:2406.16747.

[12] Biao Zhang, Ivan Titov, and Rico Sennrich. Sparse Attention with Linear Units. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6507–6520, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[13] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers, 2019.

[14] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows, 2021.

[15] Chun-Fu Chen, Quanfu Fan, and Rameswar Panda. Crossvit: Cross-attention multi-scale vision transformer for image classification, 2021.

[16] Mengyu Liu and Hujun Yin. Cross Attention Network for Semantic Segmentation, July 2019. arXiv:1907.10958.

Appendix

classescompare.py

```python
import os
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from sklearn.metrics import confusion_matrix
import seaborn as sns
color_labels = {
    "Tree": (106, 141, 34),
    "Motorist/Cyclist": (254, 0, 0),
    "Building": (69, 69, 69),
    "Minivan": (0, 79, 99),
    "Traffic Lights": (249, 169, 29),
    "Wall": (101, 101, 155),
    "Sidewalk": (243, 34, 231),
    "Text (Ignore)": (254, 254, 254),
    "Back Sign": (63, 63, 63),
    "Car": (0, 0, 141),
    "Front Sign": (219, 219, 0),
    "Minivan 2": (0, 59, 99),
    "Road": (127, 63, 127),
    "Motorcycle": (0, 0, 229),
    "Text 2 (Ignore)": (0, 0, 0),
    "Post": (152, 152, 152),
    "Bicycle": (118, 10, 31),
    "Grass": (151, 250, 151),
    "Sky": (69, 129, 179),
    "Fence": (189, 152, 152),
    "Back Sign 2": (191, 191, 191),
    "Pedestrian": (219, 19, 59),
    "Truck": (0, 0, 69)
}
def compare_image_sections(image_path):
    image = Image.open(image_path)
    image = image.convert("RGB")
    arr = np.array(image)
    height = arr.shape[0]
    half_height = height // 2
    top_half = arr[:half_height, :]
    bottom_half = arr[half_height:, :]
    return top_half, bottom_half

def create_color_confusion_matrix(ground_truth, predicted):
    ground_truth_flat = ground_truth.reshape(-1, 3)
    predicted_flat = predicted.reshape(-1, 3)
    ground_truth_colors = [tuple(color) for color in ground_truth_flat]
    predicted_colors = [tuple(color) for color in predicted_flat]
    unique_colors = list(set(ground_truth_colors + predicted_colors))
    color_to_index = {color: idx for idx, color in enumerate(unique_colors)}
    ground_truth_indices = [color_to_index[color] for color in ground_truth_colors]
    predicted_indices = [color_to_index[color] for color in predicted_colors]
    cm = confusion_matrix(ground_truth_indices, predicted_indices, labels=range(len(unique_colors)))
    filtered_color_labels = {label: color for label, color in color_labels.items() if "(Ignore)" not in label}
    filtered_unique_colors = [color for color in unique_colors if color in filtered_color_labels.values()]
    combined_labels = {
        "Minivan": ["Minivan", "Minivan 2"],
        "Back Sign": ["Back Sign", "Back Sign 2"]
    }
    for combined_label, labels_to_combine in combined_labels.items():
        combined_color = tuple(sum(color_labels[label][i] for label in labels_to_combine) for i in range(3))
        filtered_color_labels[combined_label] = combined_color
        for label in labels_to_combine:
            if label in filtered_color_labels:
                del filtered_color_labels[label]
    for combined_label, labels_to_combine in combined_labels.items():
        for i, color in enumerate(ground_truth_colors):
            if color in [color_labels[label] for label in labels_to_combine]:
                ground_truth_colors[i] = color_labels[combined_label]
        for i, color in enumerate(predicted_colors):
            if color in [color_labels[label] for label in labels_to_combine]:
                predicted_colors[i] = color_labels[combined_label]
    filtered_unique_colors = [color for color in unique_colors if color in filtered_color_labels.values()]
    filtered_indices = [unique_colors.index(color) for color in filtered_unique_colors]
    filtered_cm = cm[np.ix_(filtered_indices, filtered_indices)]
    return filtered_cm, filtered_color_labels, filtered_unique_colors

def reorder_confusion_matrix(cm, color_labels, unique_colors):
    sorted_labels = sorted(color_labels.keys())
    index_mapping = {i: sorted_labels.index(label) for i, label in enumerate(color_labels.keys()) if label in sorted_labels}
    reordered_cm = np.zeros_like(cm)
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            reordered_cm[index_mapping[i], index_mapping[j]] = cm[i, j]
    reordered_unique_colors = [color_labels[label] for label in sorted_labels]
    return reordered_cm, sorted_labels, reordered_unique_colors

def subtract_confusion_matrices(cm1, cm2):
    return cm1 - cm2

def plot_confusion_matrix(cm, labels, unique_colors):
    plt.figure(figsize=(12, 10))
    ax = sns.heatmap(cm, annot=True, fmt='d', cmap='viridis',
                     xticklabels=labels,
                     yticklabels=labels,
                     annot_kws={"size": 8})
    plt.xticks(rotation=45, ha='right')
    plt.xlabel('Predicted')
    plt.ylabel('Ground Truth')
    plt.title('Classes Confusion Matrix')
    plt.show()
def main():
    image_path = 'symmetric_confusion.png'
    ground_truth, predicted = compare_image_sections(image_path)
    cm1, color_labels1, unique_colors1 = create_color_confusion_matrix(ground_truth, predicted)
    cm2, color_labels2, unique_colors2 = create_color_confusion_matrix(ground_truth, predicted)
    cm_diff = subtract_confusion_matrices(cm1, cm2)
```

```
    cm1_reordered, sorted_labels1, reordered_unique_colors1 = reorder_confusion_matrix(cm1, color_labels1, unique_colors1)
    cm2_reordered, sorted_labels2, reordered_unique_colors2 = reorder_confusion_matrix(cm2, color_labels2, unique_colors2)
    cm_diff_reordered, sorted_labels_diff, reordered_unique_colors_diff = reorder_confusion_matrix(cm_diff, color_labels1, unique_colors1)
    plot_confusion_matrix(cm1_reordered, sorted_labels1, reordered_unique_colors1)
    plot_confusion_matrix(cm2_reordered, sorted_labels2, reordered_unique_colors2)
    plot_confusion_matrix(cm_diff_reordered, sorted_labels_diff, reordered_unique_colors_diff)
if __name__ == "__main__":
    main()
```

Code Snippet 12: Code for the Confusion Matrices

gtcompare.py

```
import os
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap
from PIL import Image

def compare_image_sections(image):
    image = image.convert("L")
    arr = np.array(image)
    height = arr.shape[0]
    third_height = height // 3
    second_third = arr[third_height:2*third_height, :]
    third_third = arr[2*third_height:, :]
    min_height = min(second_third.shape[0], third_third.shape[0])
    second_third = second_third[:min_height, :]
    third_third = third_third[:min_height, :]
    result_arr = second_third.astype(np.int16) - third_third.astype(np.int16)
    print(np.min(result_arr), np.max(result_arr), np.mean(result_arr), np.std(result_arr))
    max_abs_val = max(abs(np.min(result_arr)), abs(np.max(result_arr)))
    norm_result_arr = result_arr / max_abs_val
    cmap = LinearSegmentedColormap.from_list('blue_black_red', ['blue', 'white', 'red'], N=256)
    colored_result_arr = cmap((norm_result_arr + 1) / 2)
    result_img = Image.fromarray((colored_result_arr[:, :, :3] * 255).astype('uint8'))
    fig, ax = plt.subplots()
    cax = ax.imshow(colored_result_arr, vmin=0, vmax=1)
    ax.set_title('Comparison of Second and Third Thirds')
    ax.axis('off')
    sm = plt.cm.ScalarMappable(cmap=cmap, norm=plt.Normalize(vmin=-1, vmax=1))
    sm.set_array([])
    fig.colorbar(sm, ax=ax, orientation='vertical')
    plt.show()
    return result_img

def main():
    png_files = [f for f in os.listdir('.') if f.endswith('.png') and f.startswith('media')]
    print(png_files)
    if len(png_files) < 1:
        print("No images to perform comparison.")
        return
    with Image.open(png_files[1]) as img:
        result_img = compare_image_sections(img)

        result_img.save(f"compared_{os.path.basename(png_files[1])}")

if __name__ == "__main__":
    main()
```

Code Snippet 13: Code for Subtraction with Ground Truth