DLAD Homework 2

Yunke Ao, Kaiyue Shen

July 4, 2021

Contents

0 Introduction

In this exercise, we delve into Multi-Task Learning (MTL) architectures for dense prediction tasks. In particular, for semantic segmentation (i.e., the task of associating each pixel of an image with a class label, e.g., person, road, car, etc.) and monocular depth estimation (i.e., the task of estimating the per-pixel depth of a scene from a single image). The dataset we use is "Miniscapes", which is a toy dataset of synthetic scenes in the autonomous driving context, composed of predefined splits with 20000 training images, 2500 validation, and 2500 test images.

As we are doing Multi-Task Learning, we use SI-logRMSE (scale-invariant log root mean squared error) and IoU (intersection-over-union) to separately measure the performance of depth estimation and semantic segmentation, and Grader, which is defined as max (IoU - 50, 0) + max (50 - SI-logRMSE, 0), to measure the final performance. The higher, the better.

The whole report is structured as follows:

In Section 1, we implement and examine the structure of DeepLabv3+ [2, 1], and test the influence of hyper-parameters.

In Section 2, we implement the branched architecture based on joint architecture above and compare their performance.

In Section 3, we add the task distillation module to the branched architecture and do some comparison.

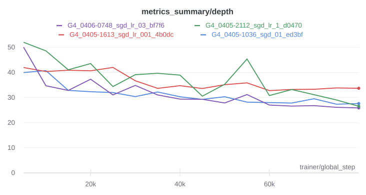
In Section 4, we further improve the model performance with a series of techniques: changing the unit of depth measurement, substituting upsampling with up-convolution, adding skip connection of feature2x, including Squeeze and Excitation layer and other hyper-parameter tuning.

1 Problem 1. Joint architecture

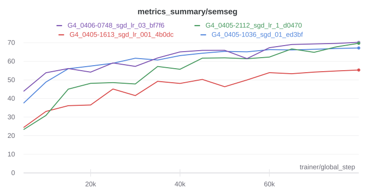
1.1 Hyper-parameter tuning

1.1.1 Optimizer and LR choice

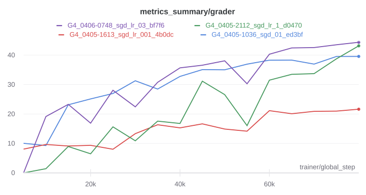
To compare the influence of optimizers and corresponding learning rates, we logarithmically vary the learning rates for both SGD and $Adam$ and the results are shown in Figure 1, 2 and Table 1, where we can see that: 1. As learning rate keeps increasing, the final learning performance first increases, then decreases. 2. The best learning rate for SGD is 0.03, while for $Adam$ is 0.0001, which is a couple of orders of magnitude smaller. We further show the Grader metric of all trials along the training time in Figure 3, we find both the training time and final performance for two optimizers under the best learning rate are similar. Considering that the whole training process of $Adam$ is more stable than that of SGD, we finally choose the combination of $Adam$ and $LR = 0.0001$ for later experiments.
Best run name: G4_0406-1956_adam_lr_0001_40275
Grader: 43.125, semseg: 69.481, depth: 26.356



(a) Depth (SI-logRMSE)

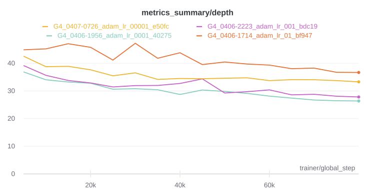


(b) SemSeg (IoU)

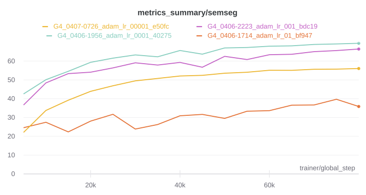


(c) Grader

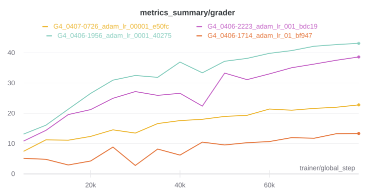
Figure 1: Evaluation metrics results of the *SGD* optimizer with different learning rates. Red: 0.001, Blue: 0.01 (default), Purple: 0.03 (best), Green: 0.1.



(a) Depth (SI-logRMSE)



(b) SemSeg (IoU)



(c) Grader

Figure 2: Evaluation metrics results of the *Adam* optimizer with different learning rates. Yellow: 0.00001, Blue: 0.0001 (best), Purple: 0.001, Orange: 0.01.

1.1.2 Batch size

We set batch size to be 2, 4 and 8 and increase the number of epochs proportionally. From Figure 4a, we see that with larger batch size, the performance increases more along each step. Since larger batch size means more accurate gradient and more stable training process. However, we cannot easily conclude that the larger batch size, the better, taking the computation time into account. From Figure 4b, comparing the trend of blue and purple lines, we see the learning performance with batch size 4 improves more during the same time period. Therefore, we pick the batch size 4.
Best run name: G4_0406-1956_adam_lr_0001_40275
Grader: 43.125, semseg: 69.481, depth: 26.356

1.1.3 Task weighting

In multi-task learning, to avoid the situation that one task loss overwhelms the other, we take different weight combinations and select the best from them. The results are displayed in Figure 5 and more detailed data can be found in Table 2. We see that from left to right, the weight of segmentation loss is increasing. Since we emphasis segmentation more, SI-logRMSE, the metric for Depth estimation task increases (worse) and IoU, the metric for Semantic Segmentation task increases as well (better). Overall, 6:4 balances two losses the best, achieving the highest Grader.
Best run name: G4_0409-0447_adam_lr_0001_weight_sd_64_6fa7f

Table 1: Comparison of models with different optimizer and LR settings

Optimizer	SGD				Adam			
LR	0.001	0.01	0.03	0.1	0.00001	0.0001	0.001	0.01
Depth (SI-logRMSE)	33.68	27.596	**25.587**	26.521	33.372	**26.356**	27.809	36.664
SemSeg (IoU)	55.312	67.154	**70.217**	69.677	56.053	**69.481**	66.42	35.854
Grader	21.632	39.558	**44.36**	43.156	22.781	**43.125**	38.611	13.336

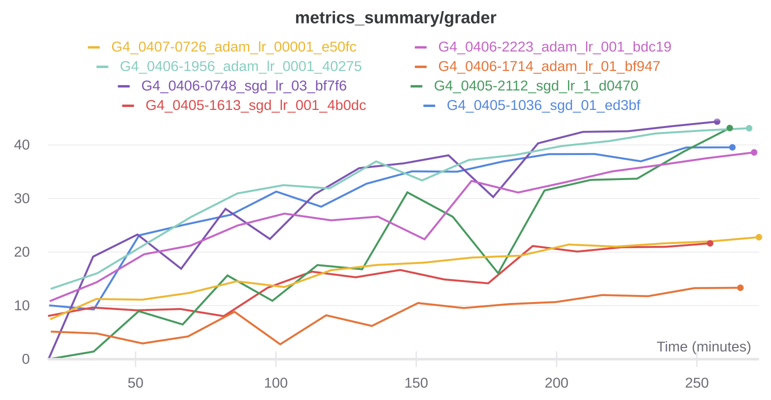


Figure 3: Evaluation metric results along training time for two optimizers

Grader: 43.542, semseg: 70.188, depth: 26.645

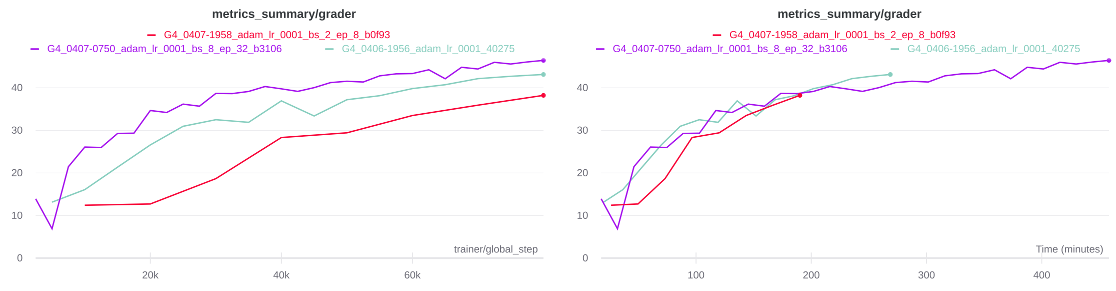
Table 2: Evaluation Metrics with different task weightings

Task weighting (Seg:Depth)	3:7	4:6	5:5	6:4	7:3	8:2	9:1
Depth (SI-logRMSE)	26.3	**26.187**	26.356	26.645	27.112	28.019	29.734
SemSeg (IoU)	68.694	69.504	69.481	70.188	**70.576**	70.313	70.534
Grader	42.394	43.317	43.125	**43.542**	43.464	42.294	40.8

1.2 Hardcoded hyper-parameters

1.2.1 Initialization with ImageNet weights

The encoder, the backbone of the model framework, is used to extract high-level features for later task usage. By default, it is randomly initialized. Now we instead initialize it with the weights pre-trained on the ImageNet dataset. Since ImageNet contains several million images including daily objects which also appear in our ”Miniscapes” dataset, our encoder is able to abstract more useful high-level features from the start of the training. Therefore, the whole model can quickly obtain higher performance, which can be



(a) Along global steps (b) Along training time

Figure 4: Evaluation metrics results with different batch sizes. Red: 2, Blue: 4 (best), Purple: 8

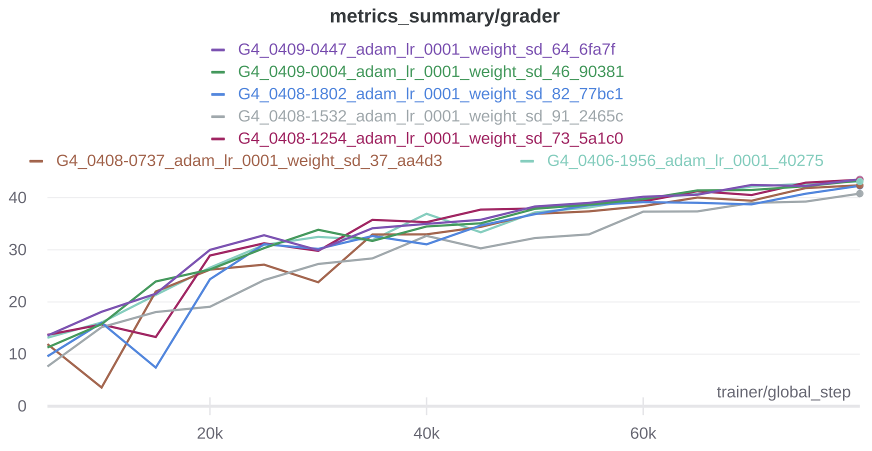


Figure 5: Evaluation metric results with different task weightings

seen from the purple and pink lines in Figure 6. It is obvious that the pink line is above the purple line since the beginning of the training and also gets higher Grader in the end.

Best run name: G4_0409-2042_adam_lr_0001_weight_sd_64_pretrained_76323
Grader: 47.455, semseg: 73.02, depth: 25.564

1.2.2 Dilated convolutions

A convolutional unit only depends on a local region (patch) of the input, i.e., Receptive Field (RF). Dilated convolution enlarges the receptive field, meaning the feature can capture more contextual information, which is crutial for both semantic segmentation and depth estimation task. For example, if we want to predict the boundary of an object, it is important for the model can access to the relevant part of the same object. Compared with pooling layer that can also increase RF, dilated convolution does not decrease the resolution. The benefit of using dilated convolution is shown in Table 3 and

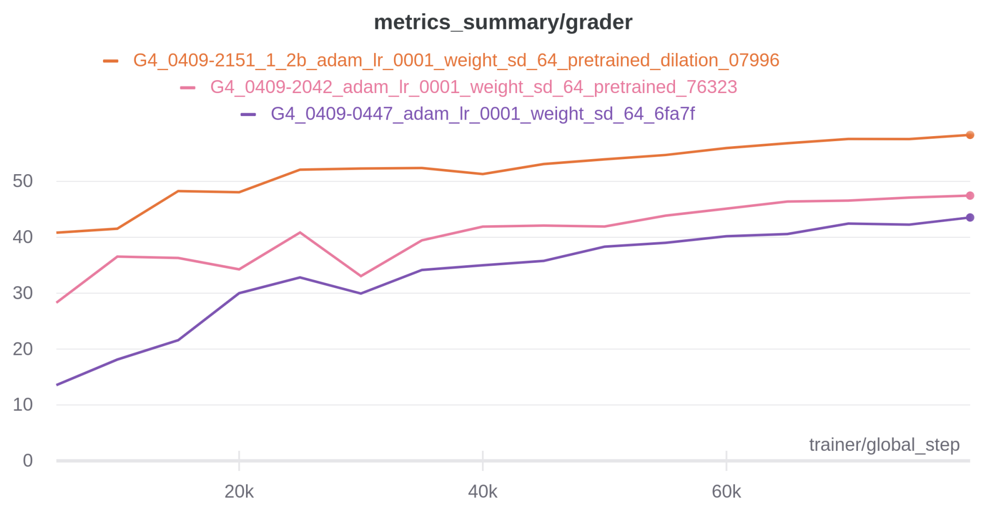


Figure 6: Evaluation metric results with different hardcoded hyper-parameters, Purple: random initialization + no dilation, Pink: encoder pre-trained on ImageNet + no dilation, Orange: encoder pre-trained on ImageNet + dilation.

Table 3: Comparison of models with different hardcoded hyper-parameters

Model	Depth (SI-logRMSE)	SemSeg (IoU)	Grader
random initialization + no dilation,	26.645	70.188	43.542
pre-trained + no dilation	25.564	73.020	47.455
pre-trained + dilation	**21.833**	**80.152**	**58.319**

Figure 6, where the orange line is above the pink line a lot during the whole training process, especially in the end, the Grader increases from 47.455 to 58.319, which verifies our statement that dilated convolution can improve the performance in our task by increasing the receptive field of features.

Best run name:
G4_0409-2151_1_2b_adam_lr_0001_weight_sd_64_pretrained_dilation_07996
Grader: 58.319, semseg: 80.152, depth: 21.833

1.3 ASPP and skip connections

We adopt the idea of DeepLabv3+, which combines the ASPP module with the encoder-decoder structure, so that it contains rich semantic information from the encoder module while captures sharper object boundaries using a simple but effective decoder module.

1.3.1 ASPP module

To handle the problem of segment objects at different scales, we use Atrous Spatial Pyramid Pooling (ASPP) module, which resample features at different scales for accurately and efficiently classifying regions of an arbitrary scale. Specifically, four parallel atrous convolutions (same as the dilated convolution we use in Section 1.2.2) with different atrous rates (1, 3, 6, 9) are applied on top of the feature map. Moreover, to incorporate global context info, image-level features are extracted by applying global average pooling on the same feature map. Finally, all five feature maps are concatenated along the channel and passed through a 1x1 convolution.

The detailed implementation can be found in the following code:

```python
class ASPP(torch.nn.Module):
    def __init__(self, in_channels, out_channels, rates=(3, 6, 9)):
        super().__init__()
        modules = []
        modules.append(ASPPpart(in_channels,out_channels,1,stride=1,
                                            padding=0,dilation=1))
        for rate in rates:
            modules.append(ASPPpart(in_channels,out_channels,3,stride=1,
                                            padding=rate,dilation=
                                            rate))
        modules.append(torch.nn.Sequential(
                torch.nn.AdaptiveAvgPool2d((1,1)),
                ASPPpart(in_channels,out_channels,1,stride=1, padding=0,
                                            dilation=1)))
        self.convs = torch.nn.ModuleList(modules)
        self.conv_out = ASPPpart(out_channels*5, out_channels,
                                            kernel_size=1, stride=1,
                                            padding=0, dilation=1)

    def forward(self, x):
        res = []
        for conv in self.convs:
            res.append(conv(x))
        # image-level feature, need bilinearly upsample after the global
                                            average pooling
        res[-1] = F.interpolate(res[-1], size=x.shape[-2:], mode='
                                            bilinear', align_corners=
                                            True)
        res = torch.cat(res, dim=1)

        return self.conv_out(res)
```

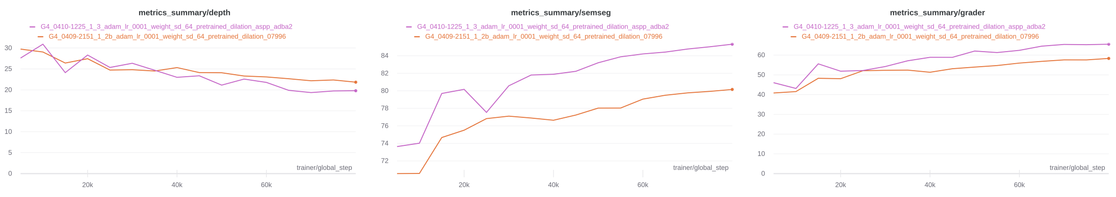
1.3.2 Skip connection

To successfully recover object segmentation details, instead of directly bilinearly upsampling, we use a decoder, the key part of which is the skip connection. To be more specific, the feature output of ASPP module are first bilinearly upsampled by a factor of 4 and then concatenated with the corresponding low-level features from the network backbone that have the same spatial resolution. The low-level features used here has already passed through a 1x1 convolution that reduces number of channels, to prevent the rich encoder features to overwhelm the ASPP feature. After the concatenation, we use two 3x3 convolution to refien the features and one 1x1 convolution together with upsampling to get the final prediction.

The detailed implementation can be found in the following code:

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

        self.features_to_predictions = torch.nn.Conv2d(bottleneck_ch,
                                                     num_out_ch, kernel_size=1,
                                                     stride=1)

        # 1*1 conv according to the paper, 48 channels has best
                                                     performance
        self.skip_to_reduced = torch.nn.Conv2d(skip_4x_ch, 48,
                                                     kernel_size=1, stride=1,
                                                     bias=False)
        self.bn1 = torch.nn.BatchNorm2d(48)
        self.relu = torch.nn.ReLU(inplace=True)
        self.conv3x3_refine_1 = torch.nn.Conv2d(48 + bottleneck_ch,
                                                     bottleneck_ch, kernel_size=3
                                                     , stride=1, padding=1, bias=
                                                     False)
        self.bn2 = torch.nn.BatchNorm2d(bottleneck_ch)
        self.conv3x3_refine_2 = torch.nn.Conv2d(bottleneck_ch,
                                                     bottleneck_ch, kernel_size=3
                                                     , stride=1, padding=1, bias=
                                                     False)
        self.bn3 = torch.nn.BatchNorm2d(bottleneck_ch)

    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
        """
        # 1x1 conv
        features_skip_reduced = self.skip_to_reduced(features_skip_4x)
        features_skip_reduced = self.bn1(features_skip_reduced)
        features_skip_reduced = self.relu(features_skip_reduced)
        # upsampling
        features_bottleneck_4x = F.interpolate(
            features_bottleneck, size=features_skip_4x.shape[2:], mode='
                                                         bilinear', align_corners
                                                         =False
        )
        # concat
        concat_features = torch.cat((features_skip_reduced,
                                              features_bottleneck_4x),dim=
                                              1)
        # refine 1
        features_4x = self.conv3x3_refine_1(concat_features)
        features_4x = self.bn2(features_4x)
        features_4x = self.relu(features_4x)
        # refine 2
        features_4x = self.conv3x3_refine_2(features_4x)
        features_4x = self.bn3(features_4x)
        features_4x = self.relu(features_4x)
        # predict
        predictions_4x = self.features_to_predictions(features_4x)

        return predictions_4x, features_4x
```

1.3.3 Results



(a) Depth (SI-logRMSE) (b) SemSeg (IoU) (c) Grader

Figure 7: Evaluation metrics results before and after adding ASPP module and skip connection to the decoder. Orange: before, Purple: after

Table 4: Comparison of models with and without ASPP module and skip connection

Model	Depth (SI-logRMSE)	SemSeg (IoU)	Grader	training time
w/o ASPP+skip	21.833	80.152	58.319	**4:50**
with ASPP+skip	**19.787**	**85.313**	**65.527**	6:45

From Figure 7 and Table 4, we can clearly see that after using the ASPP module and skip connection structured decoder, although the training time increases, the Grader increases from 58.319 to 65.527, which shows the effectiveness of ASPP module and skip connection functioning.

Best run name:
G4_0410-1225_1_3_adam_lr_0001_weight_sd_64_pretrained_dilation_aspp_adba2
Grader: 65.527, semseg: 85.313, depth: 19.787

2 Problem 2. Branched architecture

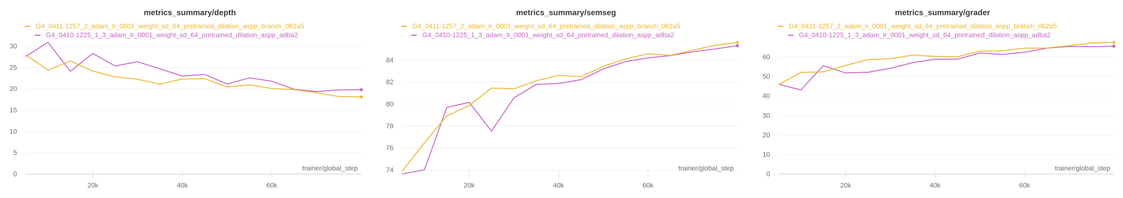
2.1 Implementation

Different from the joint architecture in Section 1, that shares all network components except the last convolutional layer – to learn both tasks, the branched architecture [4, 6] in this section only shares the encoder. Task-specific ASPP modules and decoders are implemented for semantic segmentation and monocular depth estimation, respectively. As two tasks are related, shared encoder can extract common features while task-specific ASPP and decoders allow for more targeted training.

The detailed implementation can be found in the following code:

```python
class ModelDeepLabV3PlusBranch(torch.nn.Module):
    def __init__(self, cfg, outputs_desc):
        super().__init__()
        self.outputs_desc = outputs_desc
        # get output channels for segmentation and depth map
        ch_out_seg = outputs_desc[MOD_SEMSEG]
        ch_out_depth = outputs_desc[MOD_DEPTH]

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

        ch_out_encoder_bottleneck, ch_out_encoder_4x =
                                             get_encoder_channel_counts(
                                             cfg.model_encoder_name)

        # create task-specific ASPP and decoders
        self.aspp_seg = ASPP(ch_out_encoder_bottleneck, 256)
        self.aspp_depth = ASPP(ch_out_encoder_bottleneck, 256)
        self.decoder_seg = DecoderDeeplabV3p(256, ch_out_encoder_4x,
                                             ch_out_seg)
        self.decoder_depth = DecoderDeeplabV3p(256, ch_out_encoder_4x,
                                             ch_out_depth)

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

        features = self.encoder(x)

        lowest_scale = max(features.keys())

        features_lowest = features[lowest_scale]

        # task-specific features and predictions
        features_seg = self.aspp_seg(features_lowest)
        features_depth = self.aspp_depth(features_lowest)

        predictions_4x_seg, _= self.decoder_seg(features_seg,features[4])
        predictions_4x_depth, _= self.decoder_depth(features_depth,
                                                    features[4])

        predictions_1x_seg = F.interpolate(predictions_4x_seg, size=
                                           input_resolution, mode='
                                           bilinear', align_corners=
                                           False)
        predictions_1x_depth = F.interpolate(predictions_4x_depth, size=
                                           input_resolution, mode='
                                           bilinear', align_corners=
                                           False)

        out = {}
        out[MOD_SEMSEG] = predictions_1x_seg
        out[MOD_DEPTH] = predictions_1x_depth

        return out
```

2.2 Results



(a) Depth (SI-logRMSE) (b) SemSeg (IoU) (c) Grader

Figure 8: Evaluation metrics results of joint architecture and branched architecture. Purple: joint architecture, Yellow: branched architecture

Table 5: Comparison of joint architecture and branched architecture

Model	Depth (SI-logRMSE)	SemSeg (IoU)	Grader	#params	training time
Joint	19.787	85.313	65.527	**26749396**	**6:45**
Branched	**18.099**	**85.623**	**67.524**	32142356	9:00

From Figure 8 and Table 5, we can see after changing the model from joint architecture to branched architecture, the Grader slightly increases from 65.527 to 67.524, at the cost of expanded model size and required computations.

Best run name:
G4_0411-1257_2_adam_lr_0001_weight_sd_64_pretrained_dilation_aspp_branch_062a5
Grader: 67.524, semseg: 85.623, depth: 18.099

3 Problem 3. Task distillation

3.1 Implementation

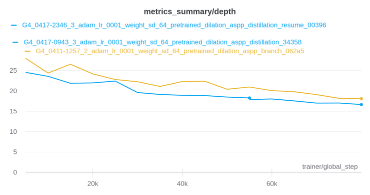
Based on the branched MTL architecture in Section 2, related work [5, 7, 8] proposed to leverage the initial task predictions to distill information across tasks. In our case, the first part of the new model is the same as branched architecture. However, instead of directly using the decoder output as the prediction, we make use of the feature before the last convolutional layer. For each task, we first get such a feature map from its own decoder. Then we get the feature map corresponding to the other task and apply self-attention to it. Finally, we sum up two feature maps as the input of another task-specific decoder, where we can get the final prediction. In summary, we have four decoders, two for each task. The idea of such an apparently complex operation is that since two tasks are closely related, the final feature before the prediction for one task may contain some useful information for another task. Task distillation can be thought as a mutually motivated interaction.

The self-attention module has been given. Indeed, it generates an attention map/-mask and new feature map from the feature map, and element-wise multiply them together. The attention map determines how important per feature unit is.

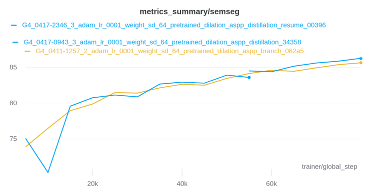
The detailed implementation can be found in the following code:

```
class ModelDeepLabV3PlusBranchAttention(torch.nn.Module):
    def __init__(self, cfg, outputs_desc):
        super().__init__()
        self.outputs_desc = outputs_desc
        # get output channels for segmentation and depth map
        ch_out_seg = outputs_desc[MOD_SEMSEG]
        ch_out_depth = outputs_desc[MOD_DEPTH]

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

        ch_out_encoder_bottleneck, ch_out_encoder_4x, ch_out_encoder_2x =
                                              get_encoder_channel_counts(
                                             cfg.model_encoder_name)

        # create task-specific ASPP and decoders
        self.aspp_seg = ASPP(ch_out_encoder_bottleneck, 256)
        self.aspp_depth = ASPP(ch_out_encoder_bottleneck, 256)
        self.decoder_seg_1 = DecoderDeeplabV3p(256, ch_out_encoder_4x,
                                             ch_out_encoder_2x,
                                             ch_out_seg)
        self.decoder_depth_1 = DecoderDeeplabV3p(256, ch_out_encoder_4x,
                                             ch_out_encoder_2x,
                                             ch_out_depth)
        # for info distillation across tasks
        self.SA_seg = SelfAttention(128, 128)
        self.SA_depth = SelfAttention(128, 128)
        self.decoder_seg_2 = DecoderDeeplabV3p(128, ch_out_encoder_4x,
                                             ch_out_encoder_2x,
                                             ch_out_seg)
        self.decoder_depth_2 = DecoderDeeplabV3p(128, ch_out_encoder_4x,
                                             ch_out_encoder_2x,
                                             ch_out_depth)

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

        features = self.encoder(x)

        lowest_scale = max(features.keys())

        features_lowest = features[lowest_scale]

        # task-specific features and predictions
        features_seg = self.aspp_seg(features_lowest)
        features_depth = self.aspp_depth(features_lowest)

        predictions_4x_seg_1, final_features_seg_1 = self.decoder_seg_1(
                                             features_seg, features[4])
        predictions_4x_depth_1, final_features_depth_1 = self.
                                             decoder_depth_1(
                                             features_depth, features[4])

        # prediction 1
        predictions_1x_seg_1 = F.interpolate(predictions_4x_seg_1, size=
                                             input_resolution, mode='
                                             bilinear', align_corners=
                                             False)
        predictions_1x_depth_1 = F.interpolate(predictions_4x_depth_1,
                                             size=input_resolution, mode=
                                             'bilinear', align_corners=
                                             False)

        # self attention
        sa_features_seg = self.SA_seg(final_features_seg_1)
        sa_features_depth = self.SA_depth(final_features_depth_1)

        # distillation
        features_input_seg_2 = final_features_seg_1 + sa_features_depth
        features_input_depth_2 = final_features_depth_1 + sa_features_seg
        predictions_4x_seg_2, _ = self.decoder_seg_2(features_input_seg_2
                                             , features[4])
        predictions_4x_depth_2, _ = self.decoder_depth_2(
                                             features_input_depth_2,
                                             features[4])

        # prediction 2
        predictions_1x_seg_2 = F.interpolate(predictions_4x_seg_2, size=
                                    input_resolution, mode='
                                    bilinear', align_corners=
                                    False)
        predictions_1x_depth_2 = F.interpolate(predictions_4x_depth_2,
                                    size=input_resolution, mode=
                                    'bilinear', align_corners=
                                    False)

        out = {}
        out[MOD_SEMSEG] = [predictions_1x_seg_1, predictions_1x_seg_2]
        out[MOD_DEPTH] = [predictions_1x_depth_1, predictions_1x_depth_2]

        return out
```

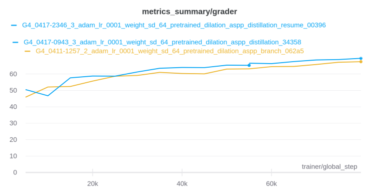
3.2 Results



(a) Depth (SI-logRMSE)



(b) SemSeg (IoU)



(c) Grader

Figure 9: Evaluation metrics results of models with and without task distillation. Yellow: without, Blue: with

Table 6: Comparison of models with and without task distillation

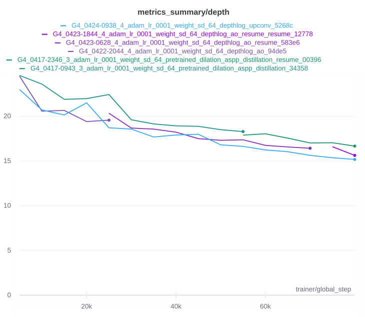
Model	Depth (SI-logRMSE)	SemSeg (IoU)	Grader	#params	training time
w/o distillation	18.099	85.623	67.524	**32142356**	**9:00**
with distillation	**16.654**	**86.246**	**69.592**	37095656	17:06

From Figure 9 and Table 6, we can after adding the task distillation, the Grader further increases from 67.524 to 69.592, also at the cost of expanded model size. One thing need to notice is that the training time almost doubles after this change.

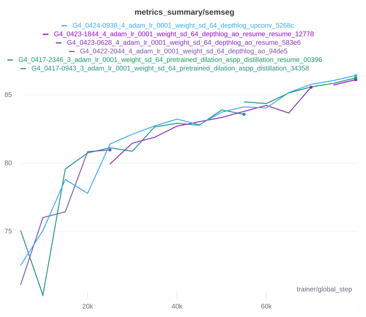
Best run name:
G4_0417-2346_3_adam_lr_0001_weight_sd_64_pretrained_dilation_aspp_distillation_resume_00396
Grader: 69.592, semseg: 86.246, depth: 16.654

4 Problem 4. Further improvement

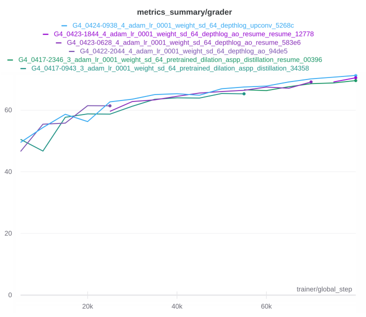
4.1 Learn depth values in log meters



(a) Depth (SI-logRMSE)



(b) SemSeg (IoU)



(c) Grader

Figure 10: Evaluation metrics results of the model of task 1.3, the model using log meters as units and the model added up-convolution layers in the decoder.

In general, depth values in meters in 2D images do not follow a normal distribution. Therefore in existing works of monocular depth estimation, log value of depth is widely used to compute the loss or as target of learning. In this task, we first utilize log meter as the depth unit of the learning target and modify the normalizing process of data. The mean and variance of depth compute with this new unit is computed as 2.8936 and 0.8661 respectively. The resulting ground truth distribution of depth in image is predicted much better compared with the previous unit as is shown in figure 11 and 12. As is shown in Figure 10(a), after this modification (purple line), the test performance of depth estimation is largely improved compared with using meter as unit (green). Therefore the final total grade is also improved.
Best run name:
G4_0423-1844_4_adam_lr_0001_weight_sd_64_depthlog_ao_resume_resume_12778
Grader: 70.504, semseg: 86.118, depth: 15.614

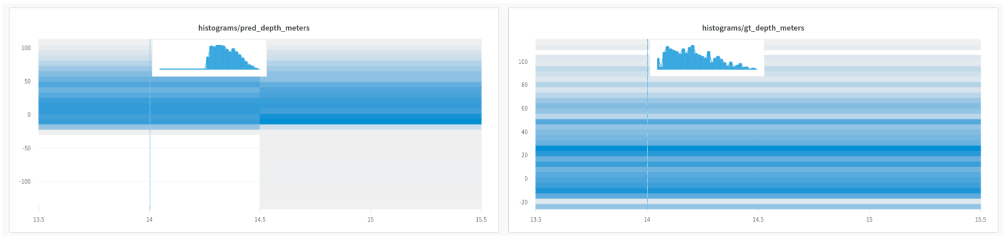


Figure 11: Distribution of predicted depth with log meter as unit

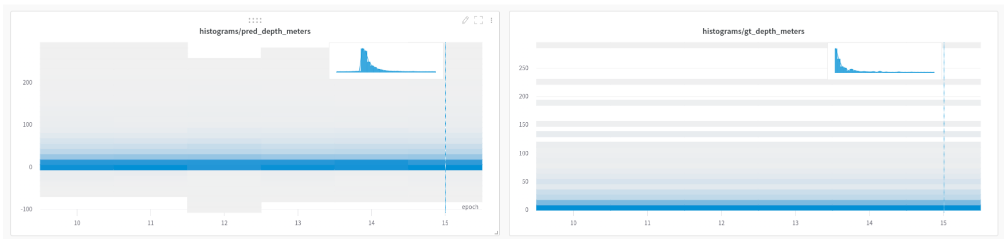


Figure 12: Distribution of predicted depth with meter as unit

4.2 Substitute up-sampling with up-convolution and add skip of feature 2x

In [3], up-convolution is applied to decode smaller features to features of larger size in order to reduce the information loss of direct up-sampling. In this task we also modify the decoder architectures to include up-convolution layers. The up-convolution layer use 4 different convolutions to up project the features with double sizes. The detailed architecture of the decoder is shown in Figure 13. We only upsample the features to the 1/2 size of the original image because more up convolution will result in much larger model size and more computational cost. With this modification, we improved both the depth estimation and segmentation performance as is shown in Figure 10 (blue line).
Best run name:
G4_0424-0938_4_adam_lr_0001_weight_sd_64_depthlog_upconv_5268c
Grader: 71.265, semseg: 86.428, depth: 15.163

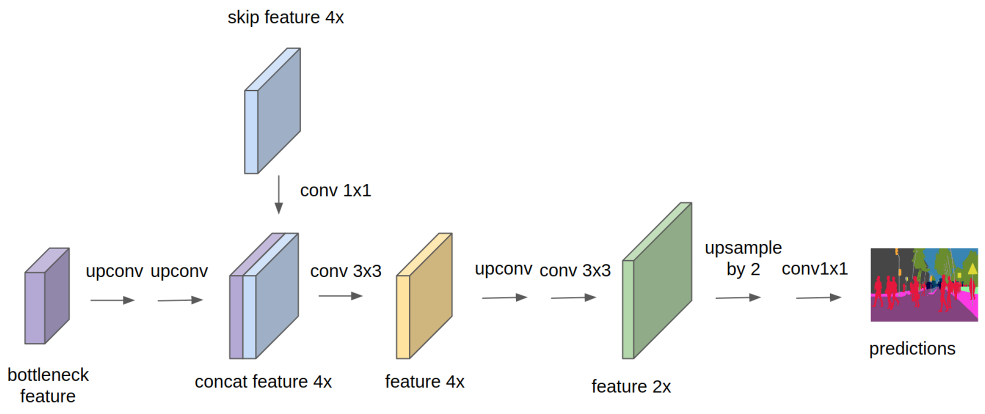


Figure 13: Decoder with up-convolution layers and skip connection from feature 4x.

The detailed code of the up-projection layer is shown below.

```
class UpProject(torch.nn.Module):

    def __init__(self, in_channel, out_channel):
        super(UpProject, self).__init__()

        self.conv1_ = torch.nn.Sequential(collections.OrderedDict([
            ('conv1', torch.nn.Conv2d(in_channel, out_channel,
                                               kernel_size=3)),
            ('bn1', torch.nn.BatchNorm2d(out_channel)),
        ]))

        self.conv2_ = torch.nn.Sequential(collections.OrderedDict([
            ('conv1', torch.nn.Conv2d(in_channel, out_channel,
                                               kernel_size=(2, 3))),
            ('bn1', torch.nn.BatchNorm2d(out_channel)),
        ]))

        self.conv3_ = torch.nn.Sequential(collections.OrderedDict([
            ('conv1', torch.nn.Conv2d(in_channel, out_channel,
                                               kernel_size=(3, 2))),
            ('bn1', torch.nn.BatchNorm2d(out_channel)),
        ]))

        self.conv4_ = torch.nn.Sequential(collections.OrderedDict([
            ('conv1', torch.nn.Conv2d(in_channel, out_channel,
                                               kernel_size=2)),
            ('bn1', torch.nn.BatchNorm2d(out_channel)),
        ]))

        self.ps = torch.nn.PixelShuffle(2)
        self.relu = torch.nn.ReLU(inplace=True)

    def forward(self, x):
        # print('Upmodule x size = ', x.size())
        x1 = self.conv1_(torch.nn.functional.pad(x, (1, 1, 1, 1)))
        x2 = self.conv2_(torch.nn.functional.pad(x, (1, 1, 0, 1)))
        x3 = self.conv3_(torch.nn.functional.pad(x, (0, 1, 1, 1)))
        x4 = self.conv4_(torch.nn.functional.pad(x, (0, 1, 0, 1)))

        x = torch.cat((x1, x2, x3, x4), dim=1)

        output = self.ps(x)
        output = self.relu(output)

        return output
```

4.3 Skip connection from feature 2x

Based on the up-projection, it is also intuitive to further add skip connection of the feature2x from the encoder inspired by the U-Net architecture ronneberger2015u, as is shown in Figure 14. The feature2x is concatenate with the up-projected feature 4x and further processed with a 3×3 convolution layer to get the final feature 2x. This skip connection improve the performance for both depth estimation and semantic segmentation, as is shown in figure 15
Best run name:
G4_0430-0629_adam_lr_0001_weight_sd_640_pretrained_dilation_ aspp_distill_depthlog_

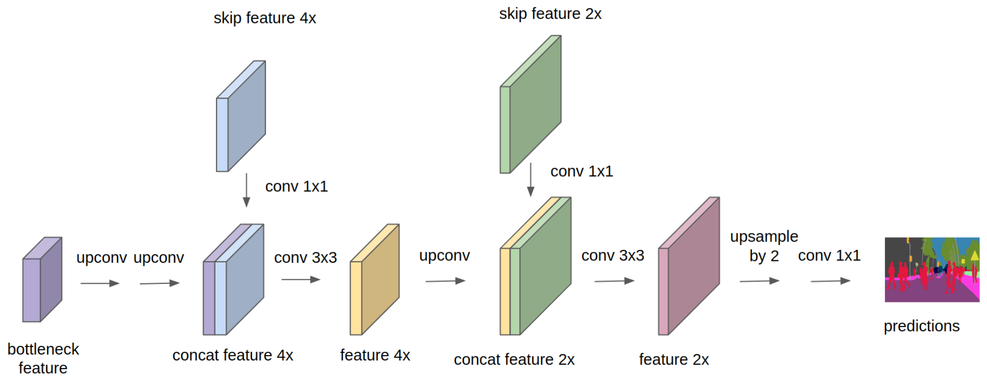
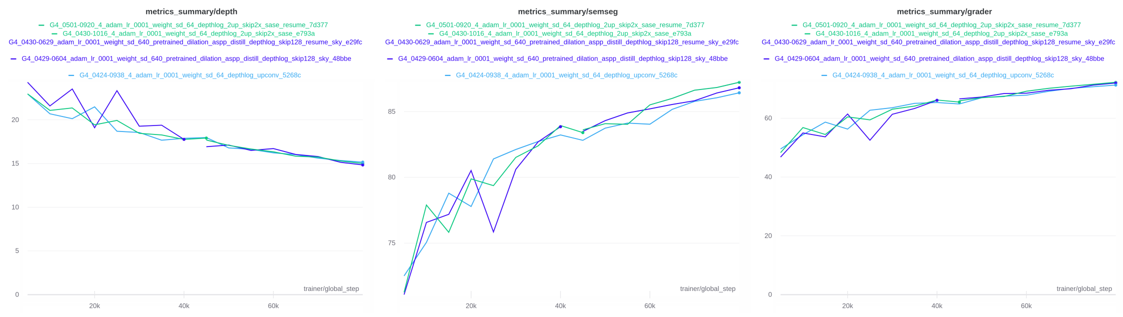


Figure 14: Decoder with up-convolution layers as well as skip connections from feature 4x and feature 2x.



(a) Depth (SI-logRMSE) (b) SemSeg (IoU) (c) Grader

Figure 15: Evaluation metrics results of the model added up-convolution layers in the decoder, the model further added skip connection from feature2x and the model further added SE layer for task distillation.

skip128_resume_sky_e29fc
Grader: 71.971, semseg: 86.81, depth: 14.939

4.4 Multi-task distillation with Squeeze And Excitation layer

In [8], a class of methods mentioned for multi-task distillation is to learn the Gate functions that filter the final features from other tasks before feeding them to the target task. We also extend the current task distillation framework by:

1. The inputs to SA layers are both the concatenation of the final features of depth estimation and semantic segmentation. The output channel is the same as the feature input of the second decoder of each task.

2. The feature output by the SA layer is further processed by a SE residual block to select useful channels.

As is shown in Figure 15, we achieved further improvement of the performance mainly for semantic segmentation.
Best run name:
G4_0501-0920_4_adam_lr_0001_weight_sd_64_depthlog_2up_skip2x_sase_resume_7d377
Grader: 72.215, semseg: 87.227, depth: 15.013

The entire code for the new decoder is shown below:

```
class ModelDeepLabV3PlusUpConvSkipSE(torch.nn.Module):
    def __init__(self, cfg, outputs_desc):
        super().__init__()
        self.outputs_desc = outputs_desc
        # get output channels for depth and seg
        ch_out_seg = outputs_desc[MOD_SEMSEG]
        ch_out_depth = outputs_desc[MOD_DEPTH]

        self.encoder = Encoder(
            cfg.model_encoder_name,
            pretrained=True, # modified to be true
            zero_init_residual=True,
            replace_stride_with_dilation=(False, False, True),
        )

        ch_out_encoder_bottleneck, ch_out_encoder_4x =
                                             get_encoder_channel_counts(
                                             cfg.model_encoder_name)

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

        self.decoder_seg_1 = DecoderDeeplabV3pConv2xSkip(256,
                                             ch_out_encoder_4x,
                                             ch_out_encoder_4x,
                                             ch_out_seg, 128, 1)
        self.decoder_depth_1 = DecoderDeeplabV3pConv2xSkip(256,
                                             ch_out_encoder_4x,
                                             ch_out_encoder_4x,
                                             ch_out_depth, 128, 1)

        self.SA_seg = SelfAttention(256, 128)
        self.SA_depth = SelfAttention(256, 128)
        self.SE_seg = SqueezeAndExcitation(128)
        self.SE_depth = SqueezeAndExcitation(128)

        self.decoder_seg_2 = DecoderDeeplabV3pConv2xSkip(128,
                                             ch_out_encoder_4x,
                                             ch_out_encoder_4x,
                                             ch_out_seg, 128, 0)
        self.decoder_depth_2 = DecoderDeeplabV3pConv2xSkip(128,
                                             ch_out_encoder_4x,
                                             ch_out_encoder_4x,
                                             ch_out_depth, 64, 0)
```

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

    features = self.encoder(x)

    # Uncomment to see the scales of feature pyramid with their
                                    respective number of
                                    channels.
    print(", ".join([f"{k}:{v.shape[1]}" for k, v in features.items()
                                    ]))

    lowest_scale = max(features.keys())

    features_lowest = features[lowest_scale]

    # aspp
    features_seg = self.aspp_seg(features_lowest)

    features_depth = self.aspp_depth(features_lowest)

    # decoder 1
    predictions_2x_seg_1, final_features_seg_1 = self.decoder_seg_1(
                                    features_seg, features[4],
                                    features[2])

    predictions_2x_depth_1, final_features_depth_1 = self.
                                    decoder_depth_1(
                                    features_depth, features[4],
                                     features[2])

    # prediction 1
    predictions_1x_seg_1 = F.interpolate(predictions_2x_seg_1, size=
                                    input_resolution, mode='
                                    bilinear', align_corners=
                                    False)

    predictions_1x_depth_1 = F.interpolate(predictions_2x_depth_1,
                                    size=input_resolution, mode=
                                    'bilinear', align_corners=
                                    False)

    # self attention
    sa_features_seg = self.SA_seg(torch.cat([final_features_seg_1,
                                    final_features_depth_1], dim
                                    =1))
    sa_features_depth = self.SA_depth(torch.cat([final_features_seg_1
                                    ,final_features_depth_1],
                                    dim=1))
    sase_features_seg = self.SE_seg(sa_features_seg) +
                                    sa_features_seg
    sase_features_depth = self.SE_depth(sa_features_depth) +
                                    sa_features_depth
```

```
        # distillation
        features_input_seg_2 = self.relu(final_features_seg_1 +
                                    sase_features_seg)
        features_input_depth_2 = self.relu(final_features_depth_1 +
                                    sase_features_depth)

        # decoder 2
        predictions_2x_seg_2, _ = self.decoder_seg_2(features_input_seg_2
                                    , features[4], features[2])
        predictions_2x_depth_2, _ = self.decoder_depth_2(
                                    features_input_depth_2,
                                    features[4], features[2])

        # prediction 2
        predictions_1x_seg_2 = F.interpolate(predictions_2x_seg_2, size=
                                    input_resolution, mode='
                                    bilinear', align_corners=
                                    False)

        predictions_1x_depth_2 = F.interpolate(predictions_2x_depth_2,
                                    size=input_resolution, mode=
                                    'bilinear', align_corners=
                                    False)

        out = {}

        out[MOD_SEMSEG] = [predictions_1x_seg_1, predictions_1x_seg_2]
        out[MOD_DEPTH] = [predictions_1x_depth_1, predictions_1x_depth_2]

        return out
```

4.5 Further hyper-parameter tuning

All the previous results are runned with the initially tuned hyper-parameters for 16 epochs Given the final architecture, we further tuned the hyperparameters including learning rate, batch size and increase number of epochs. Noticing that given same training time, batch size 4 can give best final performance, we still use 4 as the batch size instead of increasing it to 8 or 16. We also discovered that smaller learning rate 0.00002 and 0.00005 could give us better performance compared with 0.0001. For data augmentation, we found that adding random scale and geometric augmentation will result in much more time cost while achieving worse performance in wthin the same time, so we elliviate them in this task. For the same reason, we do not substitute more pooling layers with dilation layers. Several long term trained results are shown in Figure 16. Among them the best result is trained with learning rate 0.00005 for 24 epochs. It only took 32-33 hours and we believe the performance could further improve if we trained it for full 36 hours.
Best run name:
G4_0504-1135_4_adam_lr_00005_weight_sd_64_depthlog_2up_skip2x_sase_full_2a550
Grader: 74.378, semseg: 88.321, depth: 13.943

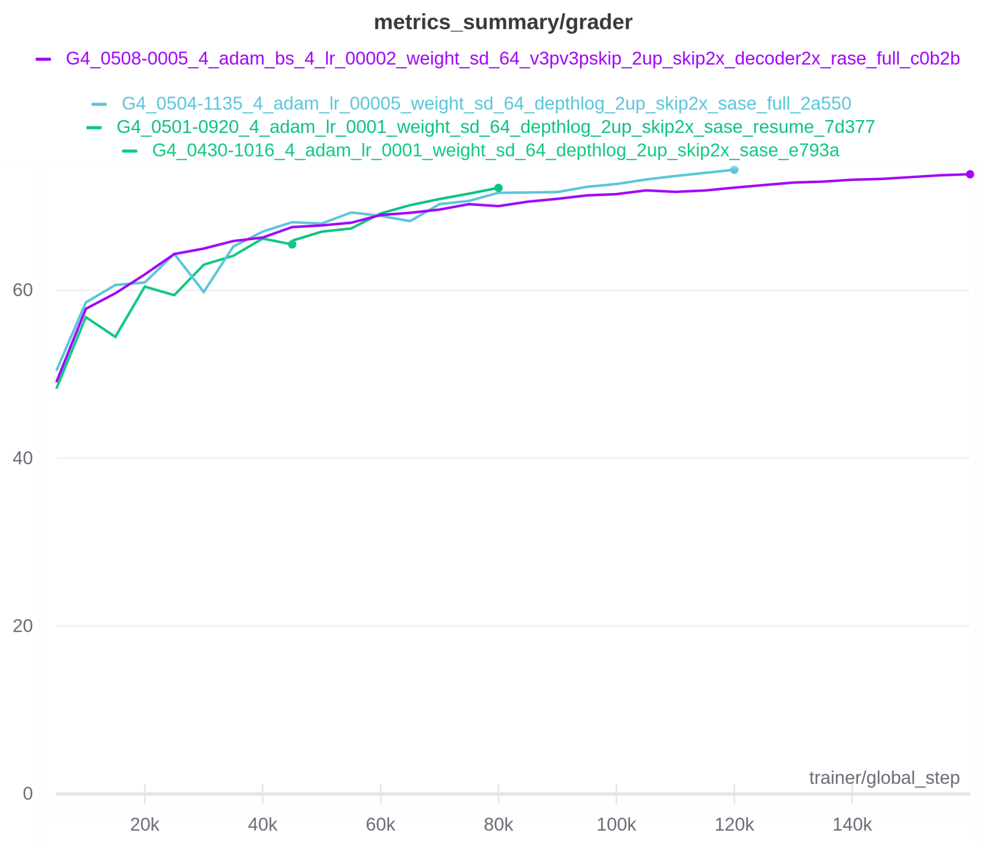


Figure 16: Performance curve of the final model with different hyperparameters.

References

[1] Liang-Chieh Chen et al. “Encoder-decoder with atrous separable convolution for semantic image segmentation”. In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 801–818.

[2] Liang-Chieh Chen et al. “Rethinking atrous convolution for semantic image segmentation”. In: *arXiv preprint arXiv:1706.05587* (2017).

[3] Iro Laina et al. “Deeper depth prediction with fully convolutional residual networks”. In: *2016 Fourth international conference on 3D vision (3DV)*. IEEE. 2016, pp. 239–248.

[4] Davy Neven et al. “Fast scene understanding for autonomous driving”. In: *arXiv preprint arXiv:1708.02550* (2017).

[5] Simon Vandenhende, Stamatios Georgoulis, and Luc Van Gool. “Mti-net: Multi-scale task interaction networks for multi-task learning”. In: *European Conference on Computer Vision*. Springer. 2020, pp. 527–543.

[6] Simon Vandenhende et al. “Branched multi-task networks: deciding what layers to share”. In: *arXiv preprint arXiv:1904.02920* (2019).

[7] Dan Xu et al. “Pad-net: Multi-tasks guided prediction-and-distillation network for simultaneous depth estimation and scene parsing”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2018, pp. 675–684.

[8] Zhenyu Zhang et al. “Pattern-affinitive propagation across depth, surface normal and semantic segmentation”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 2019, pp. 4106–4115.