Project 2: Multi-task Learning for Semantics and Depth

Abhinav Aggarwal
Department of Computer Science
University of Zurich
aabhinav@student.ethz.ch
18-748-079

Ankush Panwar
Department of Computer Science
University of Zurich
apanwar@student.ethz.ch
19-763-051

June 12, 2020

1 Joint Architecture

In this task, joint architecture as shown in Fig. 1 is used to find the optimal hyper-parameters like type of optimizer, learning rate, batch size and task weights.

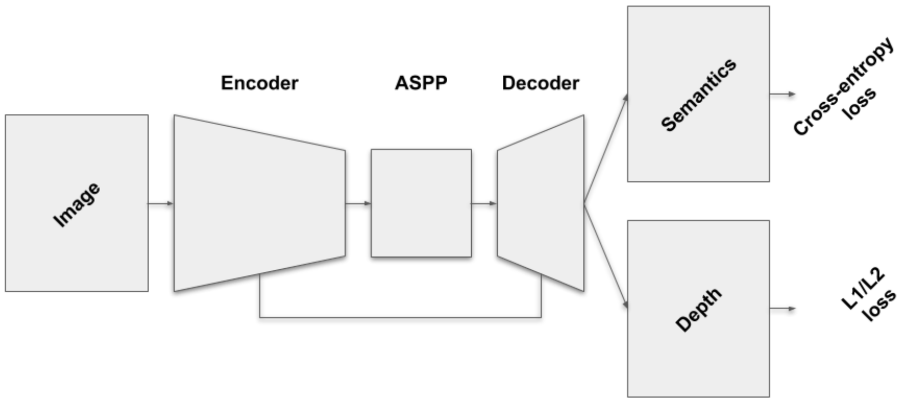


Figure 1: Joint Architecture

1.1 Hyper-parameter tuning

1. **Optimizer Choice:** Initially we run the base model with default hyper-parameters which has SGD as base optimizer. After that we again ran the base model while changing only optimizer to **adam** and optimizer_lr to **0.0001**. Table 1 shows the grader scores corresponding to different optimizers.

Optimizer Name	Muti-task	IOU	SI-logRMSE	Account	TimeStamp
SGD (default lr)	40.36	67.43	27.07	aabhinav	05/06/2020 01:22:49
Adam (lr 0.0001)	43.28	69.84	26.56	apanwar	05/06/2020 14:20:04

Table 1: Effect of Optimizer on model performance

As seen from above table, **Adam** is performing better than SGD on both segmentation task and depth estimation task.

2. **Learning Rate:** In this experiment we tried three different learning rates for Adam optimizer i.e 0.001, 0.0001 and 0.00001. Table 2 shows the grader results for different learning rates:

Learning Rate	Muti-task	IOU	SI-logRMSE	Account	TimeStamp
1e-03	41.02	68.35	27.33	apanwar	05/06/2020 12:35:54
1e-04	43.28	69.84	26.56	apanwar	05/06/2020 14:20:04
1e-05	23.76	56.63	32.86	apanwar	05/06/2020 14:22:55

Table 2: Effect of learning rate on model performance

As seen from the above table learning rate of **1e-04** is performing best in our case. We hypothesize that learning rate greater than 1e-04 might be little higher and thus model did not reach better optima than **1e-04**. Similarly learning rate of 1e-05 might be too low and therefore resulting in slow convergence and worsen the score due to fixed number of epochs.

3. **Batch Size:** In this experiment we run the model with batch size of 4, 8 and 12 with Adam optimizer and learning rate of 1e-04. Table 3 shows the grader scores for different batch sizes

Batch Size	Muti-task	IOU	SI-logRMSE	Account	TimeStamp
4	43.28	69.84	26.56	apanwar	05/06/2020 14:20:04
8	42.20	69.15	26.94	apanwar	05/06/2020 16:50:10
12	41.85	68.80	26.95	aabhinav	05/06/2020 16:45:51

Table 3: Effect of Batch Size on model performance

As seen from table 3, increase in batch size is affecting the model performance on both tasks, hence we decide to use batch size of 4 for our further experiments.

4. **Task weighting:** In this experiment we explored task weight combinations to balance the loss of different tasks. Table 4 shows the grader scores corresponding to different task weight combinations:

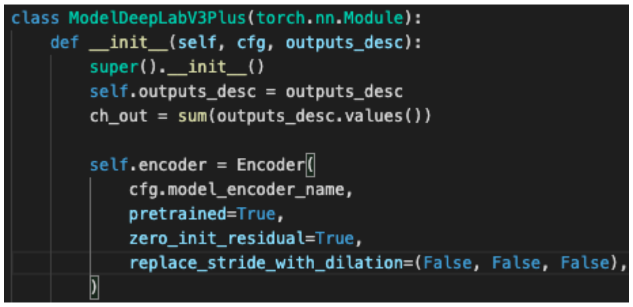
Task weight(Seg,Depth)	Muti-task	IOU	SI-logRMSE	Account	TimeStamp
(0.5,0.5)	43.28	69.84	26.56	apanwar	05/06/2020 14:20:04
(0.67,0.33)	42.82	70.25	27.43	apanwar	05/06/2020 20:46:22
(0.33,0.67)	42.49	68.93	26.44	aabhinav	05/06/2020 20:26:25

Table 4: Effect of different task weight combination on performance

As seen from table 4, equal task weights performed better than other two combinations where one task is given more weight than other.

1.2 Hardcoded hyperparameters

1. **Initialization with ImageNet weights:** As per the provided base code, encoder network is **not** initialized with weights trained on ImageNet classification task. We initialized encoder weights in this `mtl/models/model_deeplab_v3_plus.py` file where `Encoder` class is instantiated. Fig 2 show the code snippet where flag to use pretrained ImageNet weights is set to `True`



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

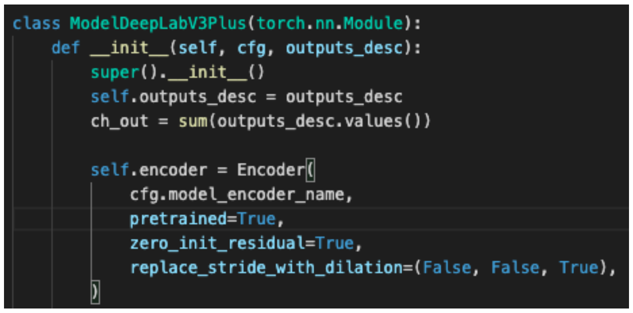
Figure 2: Initialization with ImageNet weights

Model	Muti-task	IOU	SI-logRMSE	Account	TimeStamp
Base Model	43.28	69.84	26.56	apanwar	05/06/2020 14:20:04
ImageNet Pretrained Encoder	47.58	72.90	25.32	apanwar	05/07/2020 02:40:45

Table 5: Effect of using ImageNet Pretrained weights on model performance

Table 5 shows significant improvement in model performance for both segmentation and depth estimation tasks. Please note that both the models are trained using Adam optimizer with learning rate of 1e-04

2. **Dilated Convolutions:** As per the provided base code, dilated convolutions are **not** enabled. After changing the dilation flags to `(False, False, True)`, model performance takes another boost. Table 6 show the impact of using above mentioned dilation combination instead of default False values. Also Fig 3 shows the code snipped where dilation flags are changed.



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

Figure 3: Using (False, False, True) dilation combination

Model	Muti-task	IOU	SI-logRMSE	Account	TimeStamp
ImageNet Pretrained	47.58	72.90	25.32	apanwar	05/07/2020 02:40:45
ImageNet Pretrained + Dilation	58.35	79.88	21.52	apanwar	05/07/2020 02:40:45

Table 6: Effect of using (False, False, True) dilation combination on model performance

As seen in table 6, model performance improves significantly when dilation is used. This is primarily due to reduction in **output stride** from 32 to 16 which helps the model to produce segmentation mask and depth map from higher resolution. Due to this higher resolution, thin or distant objects can be better seen by network, thus resulting in better segmentation masks.

1.3 ASPP Module

In this part we implement ASPP module as shown in Fig 4. Code snippet of implementation of ASPP module is shown in fig 5.

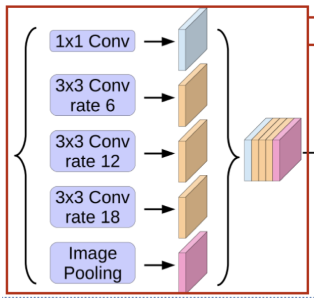
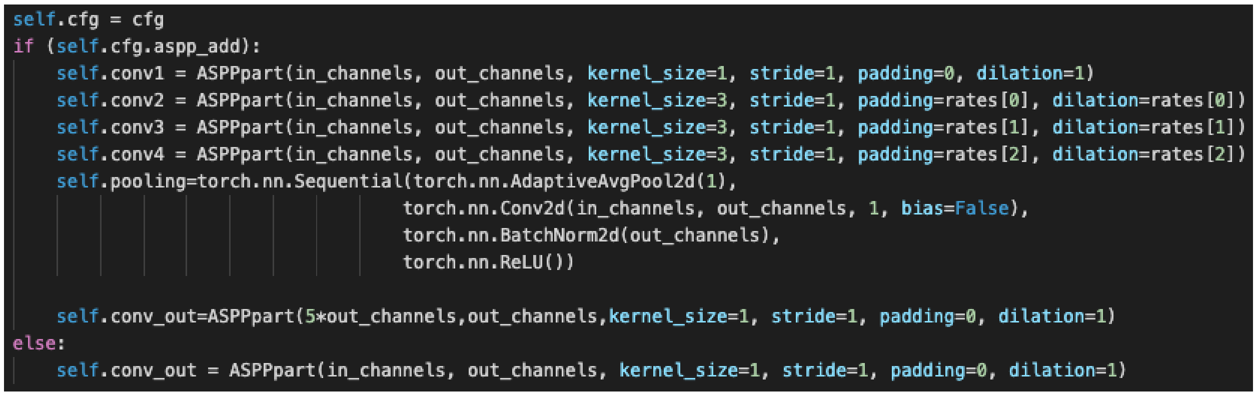


Figure 4: ASPP Architecture

As seen from Table 7, model performance was slightly improved due to ASPP module.



```
self.cfg = cfg
if (self.cfg.aspp_add):
    self.conv1 = ASPPpart(in_channels, out_channels, kernel_size=1, stride=1, padding=0, dilation=1)
    self.conv2 = ASPPpart(in_channels, out_channels, kernel_size=3, stride=1, padding=rates[0], dilation=rates[0])
    self.conv3 = ASPPpart(in_channels, out_channels, kernel_size=3, stride=1, padding=rates[1], dilation=rates[1])
    self.conv4 = ASPPpart(in_channels, out_channels, kernel_size=3, stride=1, padding=rates[2], dilation=rates[2])
    self.pooling=torch.nn.Sequential(torch.nn.AdaptiveAvgPool2d(1),
                                    torch.nn.Conv2d(in_channels, out_channels, 1, bias=False),
                                    torch.nn.BatchNorm2d(out_channels),
                                    torch.nn.ReLU())

    self.conv_out=ASPPpart(5*out_channels,out_channels,kernel_size=1, stride=1, padding=0, dilation=1)
else:
    self.conv_out = ASPPpart(in_channels, out_channels, kernel_size=1, stride=1, padding=0, dilation=1)
```

Figure 5: ASPP implementation Code Snippet

Model	Muti-task	IOU	SI-logRMSE	Account	TimeStamp
Base (Pretrained + Dilation)	58.35	79.88	21.52	apanwar	05/07/2020 02:40:45
Base + ASPP	58.58	80.03	21.46	apanwar	05/07/2020 08:22:29

Table 7: Effect of using ASPP module on model performance

1.4 Skip Connection

In this part we add skip connection between low level encoder features and ASPP output. We first upsample ASPP output to image resolution corresponding to stride 4 i.e upsampled size is equal to image resolution divided by 4. Also lower level features at 4x stride are first passed through 1×1 convolution to reduce the number of channels and are then concatenated with upsampled ASPP output. This concatenated feature map is then passed through another 3×3 conv layer. Then this feature map is passed through 1×1 conv layer to produce output with required number of channels. Final output is recieved by upsampling this feature map to image resolution. This whole process is demostrated in Fig. 6. Code snippet of implementation of skip connection is shown Fig. 7.

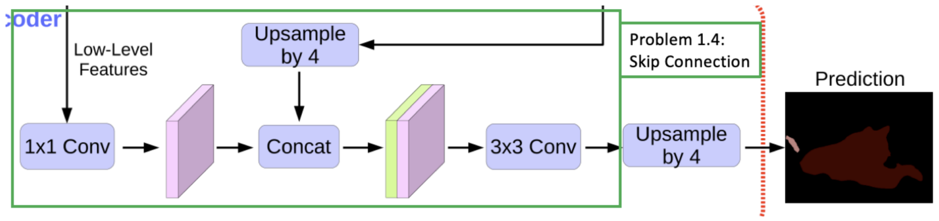


Figure 6: Skip Connection

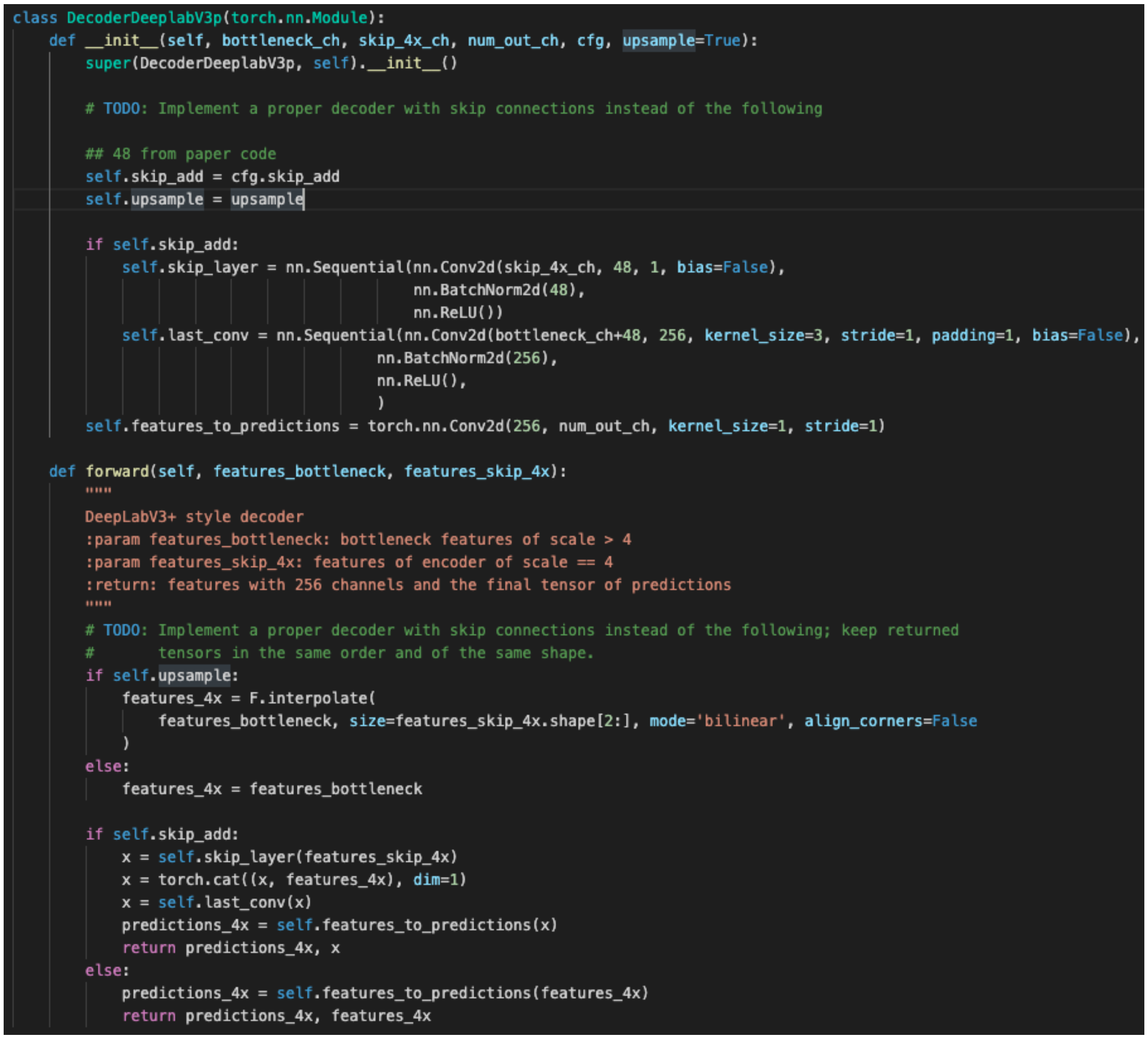
Model	Muti-task	IOU	SI-logRMSE	Account	TimeStamp
Base (Pretrained + Dilation)	58.35	79.88	21.52	apanwar	05/07/2020 02:40:45
Base + ASPP	58.58	80.03	21.46	apanwar	05/07/2020 08:22:29
Base + ASPP + Skip	65.20	84.53	19.33	apanwar	05/07/2020 07:47:01

Table 8: Effect of adding skip connection on model performance

As seen from Table 8, adding skip connection improves model performance significantly. This can be attributed to the fact that having access to low level features from encoder, results in improving segmentation and depth map boundaries. It also creates gradient highway which makes training easier.

2 Branched Architecture

In this part, we implement Branched Architecture as shown in Fig. 8. Here instead of one ASPP module and one decoder shared between both segmentation and depth estimation tasks, we have task specific ASPP modules and decoders. Code snippets for implementation of Branched Architecture is shown in Fig. 9, 10. We have created new



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

        # TODO: Implement a proper decoder with skip connections instead of the following

        ## 48 from paper code
        self.skip_add = cfg.skip_add
        self.upsample = upsample

        if self.skip_add:
            self.skip_layer = nn.Sequential(nn.Conv2d(skip_4x_ch, 48, 1, bias=False),
                                            nn.BatchNorm2d(48),
                                            nn.ReLU())
            self.last_conv = nn.Sequential(nn.Conv2d(bottleneck_ch+48, 256, kernel_size=3, stride=1, padding=1, bias=False),
                                       nn.BatchNorm2d(256),
                                       nn.ReLU(),
                                       )
        self.features_to_predictions = torch.nn.Conv2d(256, num_out_ch, kernel_size=1, stride=1)

    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.
        if self.upsample:
            features_4x = F.interpolate(
                features_bottleneck, size=features_skip_4x.shape[2:], mode='bilinear', align_corners=False
            )
        else:
            features_4x = features_bottleneck

        if self.skip_add:
            x = self.skip_layer(features_skip_4x)
            x = torch.cat((x, features_4x), dim=1)
            x = self.last_conv(x)
            predictions_4x = self.features_to_predictions(x)
            return predictions_4x, x
        else:
            predictions_4x = self.features_to_predictions(features_4x)
            return predictions_4x, features_4x
```

Figure 7: Skip Connection Code Snippet

file in `mtl/models/model_branched_arch.py` and defined new class naming `ModelBranchedArch` fro branched architecture.

Model	Muti-task	IOU	SI-logRMSE	ModelSize (Millions)	Computations (GMACs)	Account	TimeStamp
Base + ASPP + Skip	65.20	84.53	19.33	26	11.24	apanwar	05/07/2020 07:47:01
Branched Network	66.97	84.83	17.85	31	15.15	aabhinav	05/07/2020 09:05:21

Table 9: Effect of Branched Architecture on model performance

As seen from table 9, replacing joint architecture with branched architecture results in 19% increase in model size as well as 35% increase in computations whereas grader score improves by 1.8%.

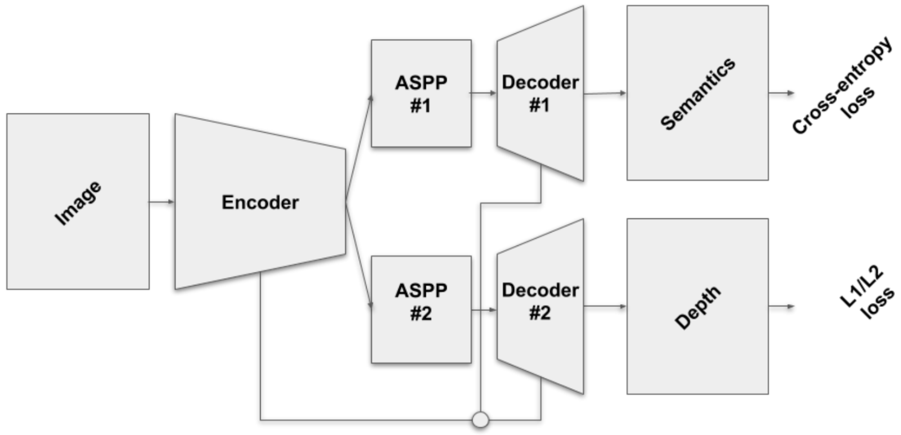
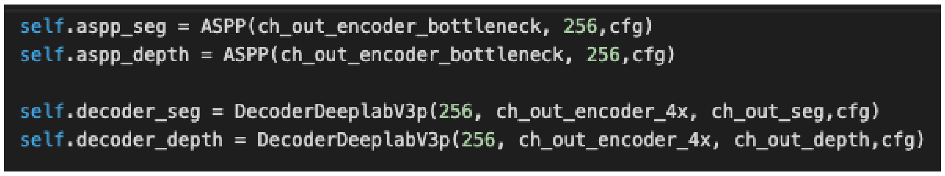
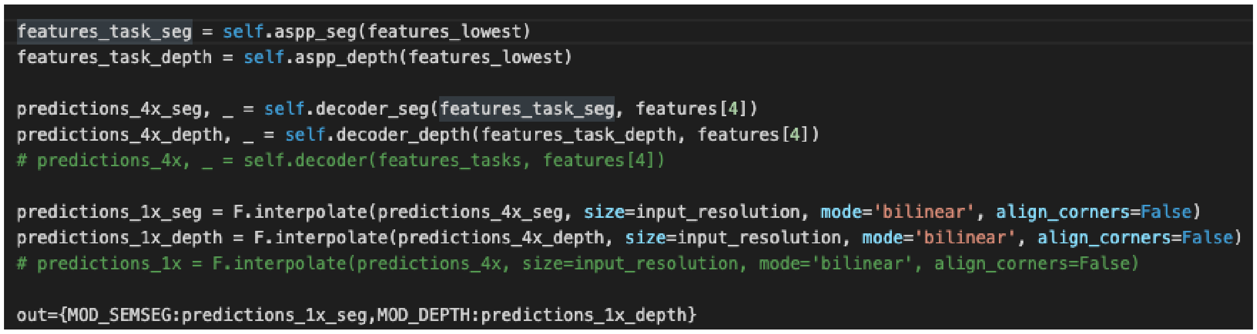


Figure 8: Branched Architecture



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

self.decoder_seg = DecoderDeeplabV3p(256, ch_out_encoder_4x, ch_out_seg,cfg)
self.decoder_depth = DecoderDeeplabV3p(256, ch_out_encoder_4x, ch_out_depth,cfg)
```

Figure 9: Initializing separate ASPP and Decoder module for depth and segmentation task



```
features_task_seg = self.aspp_seg(features_lowest)
features_task_depth = self.aspp_depth(features_lowest)

predictions_4x_seg, _ = self.decoder_seg(features_task_seg, features[4])
predictions_4x_depth, _ = self.decoder_depth(features_task_depth, features[4])
# predictions_4x, _ = self.decoder(features_tasks, 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)
# predictions_1x = F.interpolate(predictions_4x, size=input_resolution, mode='bilinear', align_corners=False)

out={MOD_SEMSEG:predictions_1x_seg,MOD_DEPTH:predictions_1x_depth}
```

Figure 10: Branched Architecture

3 Task Distillation

In this task, we implement task distillation architecture where information from initial task predictions is used to distill information across tasks using spatial attention module. Task distillation architecture is show in Fig. 11. Code snippets showing implementation of Task distillation architecture is shown in Fig. 12, 13 and 14

Model	Muti-task	IOU	SI-logRMSE	Account	TimeStamp
Branched Network	66.97	84.83	17.85	aabhinav	05/07/2020 09:05:21
Task Distillation	68.35	85.37	17.02	aabhinav	05/07/2020 08:58:25

Table 10: Effect of Branched Architecture with task distillation on model performance

As seen in table 10, there is further improvement in grader score from 66.97 to 68.35 by using task distillation with branched architecture.

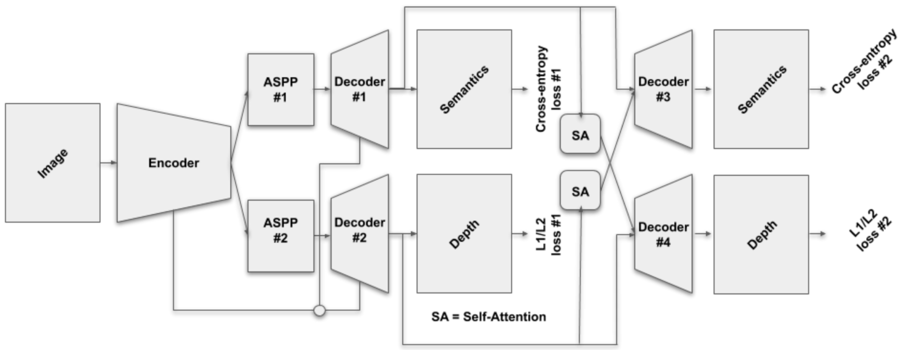
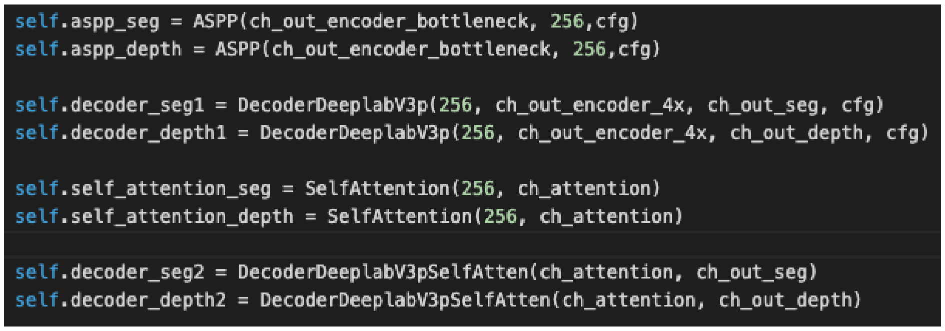
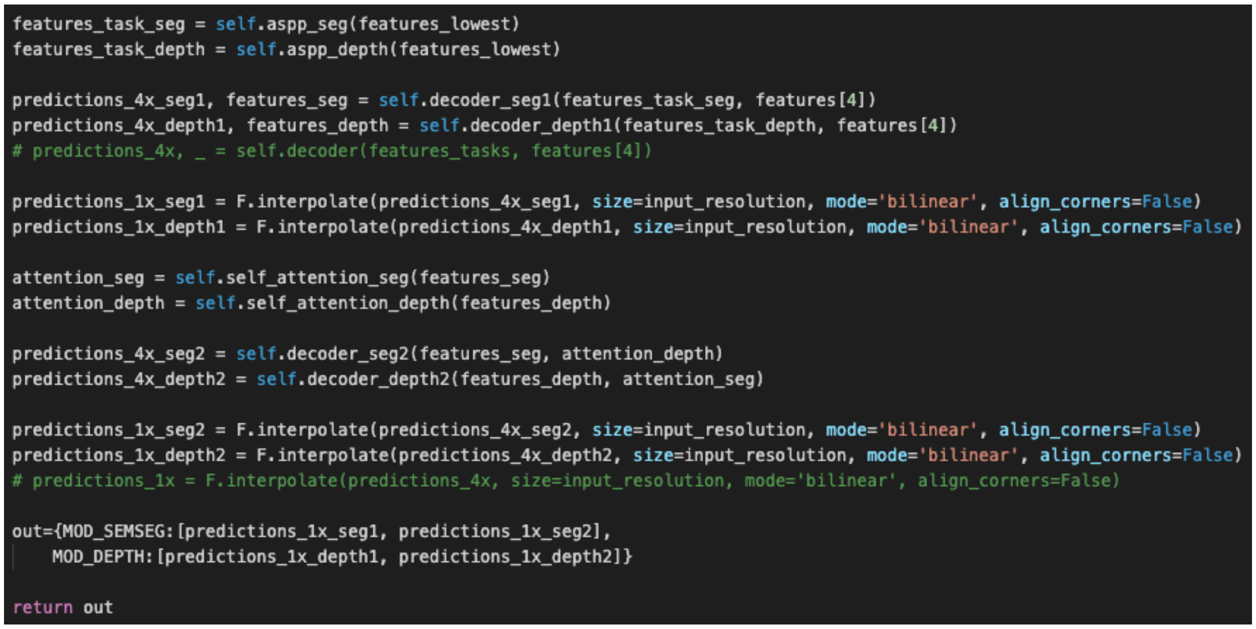


Figure 11: Branched Architecture with task distillation



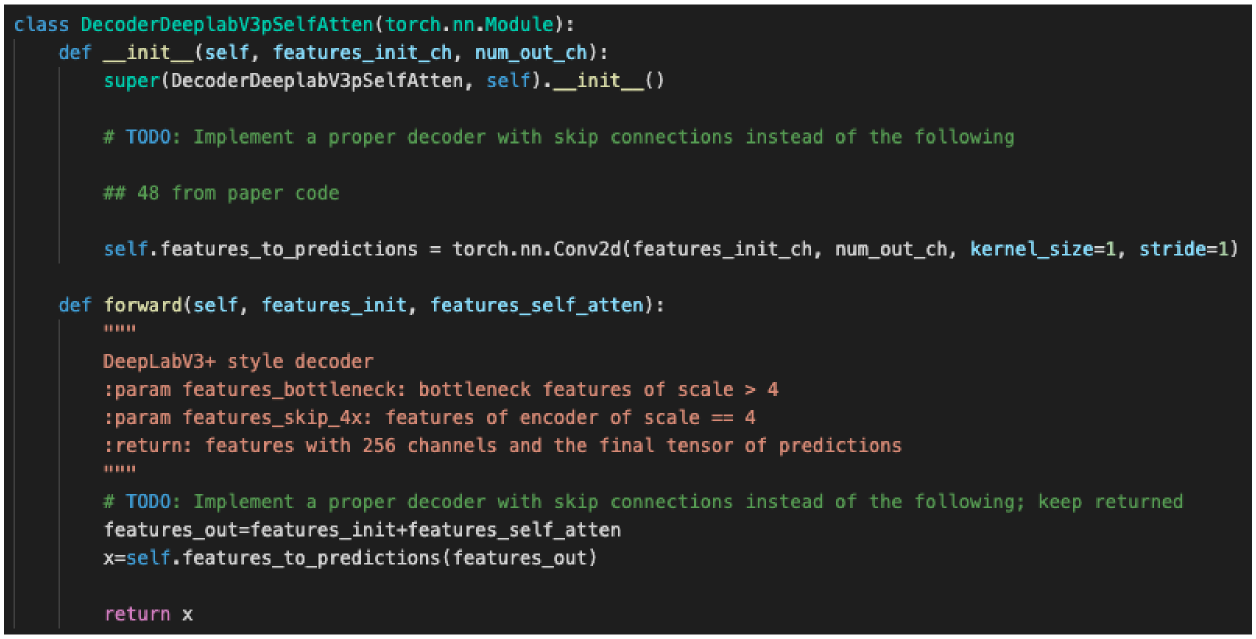
```
self.aspp_seg = ASPP(ch_out_encoder_bottleneck, 256,cfg)
self.aspp_depth = ASPP(ch_out_encoder_bottleneck, 256,cfg)

self.decoder_seg1 = DecoderDeeplabV3p(256, ch_out_encoder_4x, ch_out_seg, cfg)
self.decoder_depth1 = DecoderDeeplabV3p(256, ch_out_encoder_4x, ch_out_depth, cfg)

self.self_attention_seg = SelfAttention(256, ch_attention)
self.self_attention_depth = SelfAttention(256, ch_attention)

self.decoder_seg2 = DecoderDeeplabV3pSelfAtten(ch_attention, ch_out_seg)
self.decoder_depth2 = DecoderDeeplabV3pSelfAtten(ch_attention, ch_out_depth)
```

Figure 12: Task Distillation: Initializing different decoder and self attention module for task distillation network



```
features_task_seg = self.aspp_seg(features_lowest)
features_task_depth = self.aspp_depth(features_lowest)

predictions_4x_seg1, features_seg = self.decoder_seg1(features_task_seg, features[4])
predictions_4x_depth1, features_depth = self.decoder_depth1(features_task_depth, features[4])
# predictions_4x, _ = self.decoder(features_tasks, features[4])

predictions_1x_seg1 = F.interpolate(predictions_4x_seg1, size=input_resolution, mode='bilinear', align_corners=False)
predictions_1x_depth1 = F.interpolate(predictions_4x_depth1, size=input_resolution, mode='bilinear', align_corners=False)

attention_seg = self.self_attention_seg(features_seg)
attention_depth = self.self_attention_depth(features_depth)

predictions_4x_seg2 = self.decoder_seg2(features_seg, attention_depth)
predictions_4x_depth2 = self.decoder_depth2(features_depth, attention_seg)

predictions_1x_seg2 = F.interpolate(predictions_4x_seg2, size=input_resolution, mode='bilinear', align_corners=False)
predictions_1x_depth2 = F.interpolate(predictions_4x_depth2, size=input_resolution, mode='bilinear', align_corners=False)
# predictions_1x = F.interpolate(predictions_4x, size=input_resolution, mode='bilinear', align_corners=False)

out={MOD_SEMSEG:[predictions_1x_seg1, predictions_1x_seg2],
    MOD_DEPTH:[predictions_1x_depth1, predictions_1x_depth2]}

return out
```

Figure 13: Task Distillation: Forward function



```python
class DecoderDeeplabV3pSelfAtten(torch.nn.Module):
    def __init__(self, features_init_ch, num_out_ch):
        super(DecoderDeeplabV3pSelfAtten, self).__init__()

        # TODO: Implement a proper decoder with skip connections instead of the following

        ## 48 from paper code

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

    def forward(self, features_init, features_self_atten):
        """
        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
        features_out=features_init+features_self_atten
        x=self.features_to_predictions(features_out)

        return x
```

Figure 14: Task Distillation: Decoder 3 and Decoder 4 classes

4 Winning the Challenge

In this section, we try different ideas to further improve the performance of multi-task network. Following are the few ideas we explored resulting in 7^{th} **place in final Codalab leaderboard**.

4.1 SqueezeAndExcitation Module

Taking inspiration from [1], we use SqueezeAndExcitation Module in our ResNet34 encoder block i.e `BasicBlockWithDilation` class. Code snippet in fig. 15 shows our implementation. SqueezeAndExcitation block adaptively recalibrates channel-wise feature responses by explicitly modelling inter-dependencies between channels.

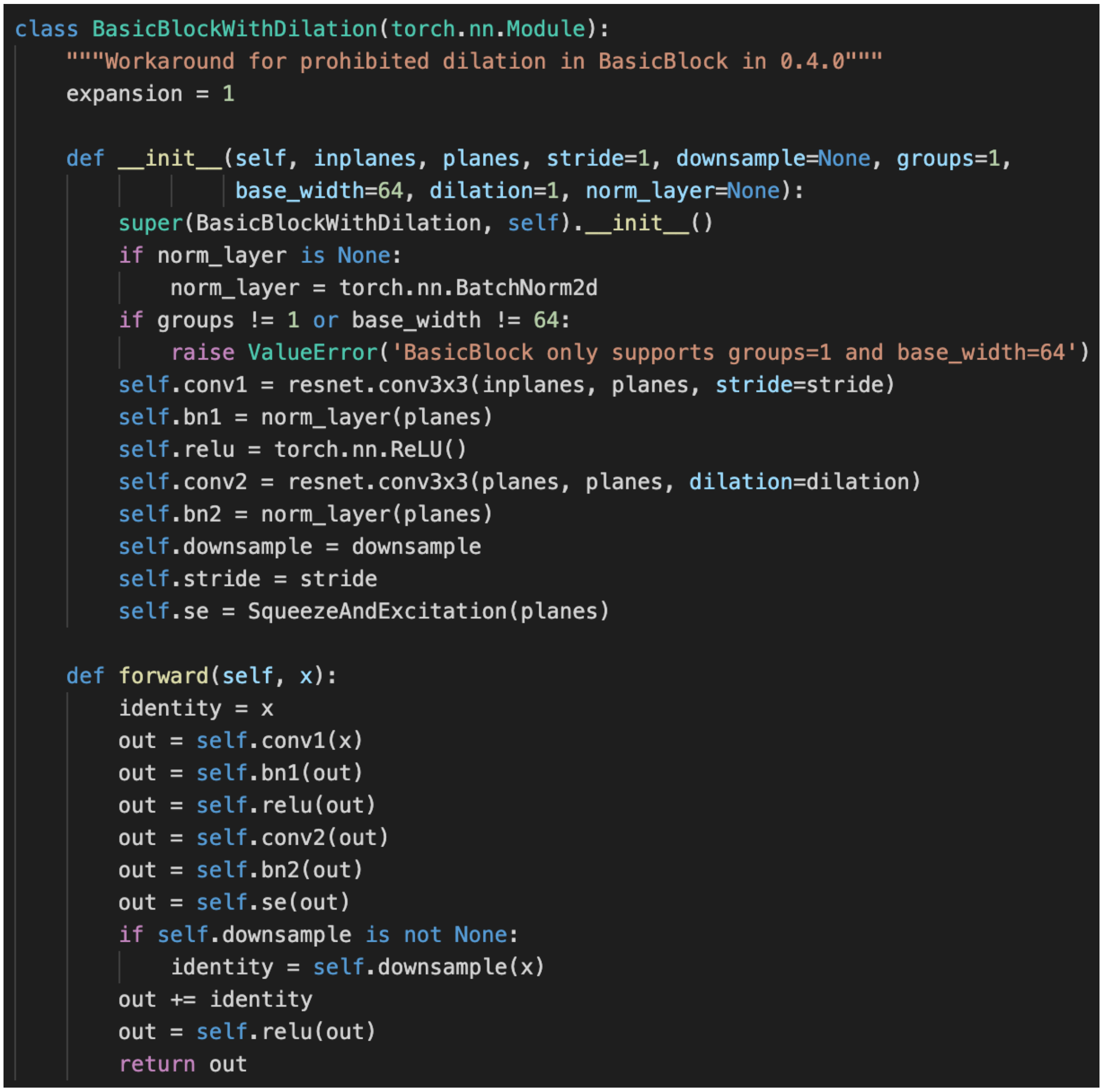
As expected adding SqueezeAndExcitation module in encoder block results in significant improvement in model performance and helps in better generalization as well. Table 11 shows improvement in model performance after using SqueezeAndExcitation module for both segmentation and depth estimation task.

4.2 Adding skip connection at every scale of feature pyramid

In the original Deeplabv3+ paper [2], output from ASPP module is upsampled 4 times and is then concatenated with low level features at 4x scale. These features are again upsampled 4 times to get original image resolution. In our implementation, we upsample ASPP module output by the factor of two and then these features are concatenated with low level encoder features at 8x stride to get feature map f_{8x}. Again f_{8x} features are upsampled 2 times and concatenated with encoder features at 4x stride to generate feature map f_{4x}. Finally features f_{4x} are upsampled 2 times and concatenated with encoder features at 2x stride to get f_{2x}. Finally f_{2x} features are sent to decoder2 where information from other tasks are distilled using spacial attention and further upsampled 2 times to produce final output. Our complete architecture can be seen in fig. 16. Please note that all these changes are done in decoders before task distillation.

The key motivation to add residual connection is to get both high level semantic information from features generated by ASPP module as well as boundary information from low level encoder features. This also provides gradient highway between encoder and decoder networks making training easier. Thus providing residual connections at every scale provides boundary information from low level features.

We implement this model in separate file naming `model_task_distill_all_connect.py` in `models` folder. This model can be run using `model_name` as `taskdistillallcon` in `config.py` file. Fig. 17, 18, 19, 20 and 21 shows the code snippets of our implementation of all-skip connection idea described above.



```python
class BasicBlockWithDilation(torch.nn.Module):
    """Workaround for prohibited dilation in BasicBlock in 0.4.0"""
    expansion = 1

    def __init__(self, inplanes, planes, stride=1, downsample=None, groups=1,
                 base_width=64, dilation=1, norm_layer=None):
        super(BasicBlockWithDilation, self).__init__()
        if norm_layer is None:
            norm_layer = torch.nn.BatchNorm2d
        if groups != 1 or base_width != 64:
            raise ValueError('BasicBlock only supports groups=1 and base_width=64')
        self.conv1 = resnet.conv3x3(inplanes, planes, stride=stride)
        self.bn1 = norm_layer(planes)
        self.relu = torch.nn.ReLU()
        self.conv2 = resnet.conv3x3(planes, planes, dilation=dilation)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride
        self.se = SqueezeAndExcitation(planes)

    def forward(self, x):
        identity = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        out = self.se(out)
        if self.downsample is not None:
            identity = self.downsample(x)
        out += identity
        out = self.relu(out)
        return out
```

Figure 15: SqueezeAndExcitation Module Code Snippet

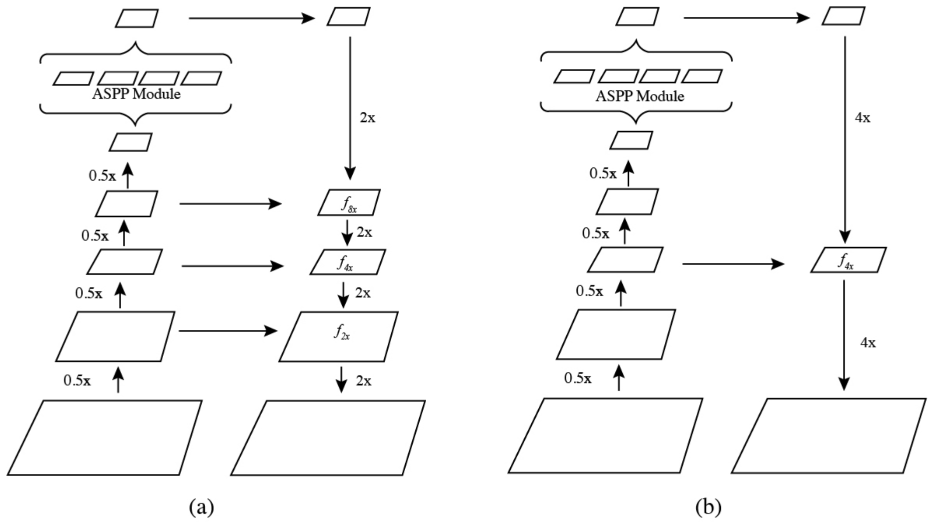
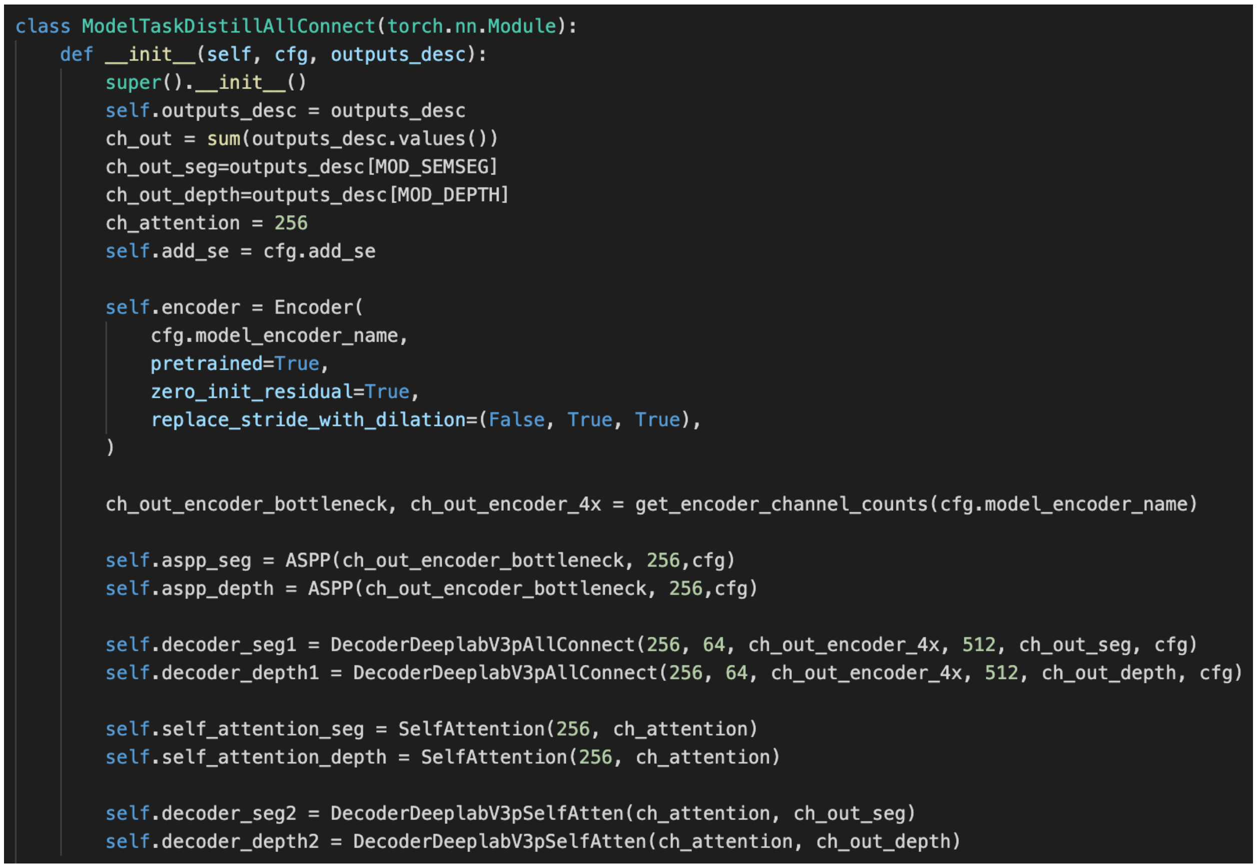
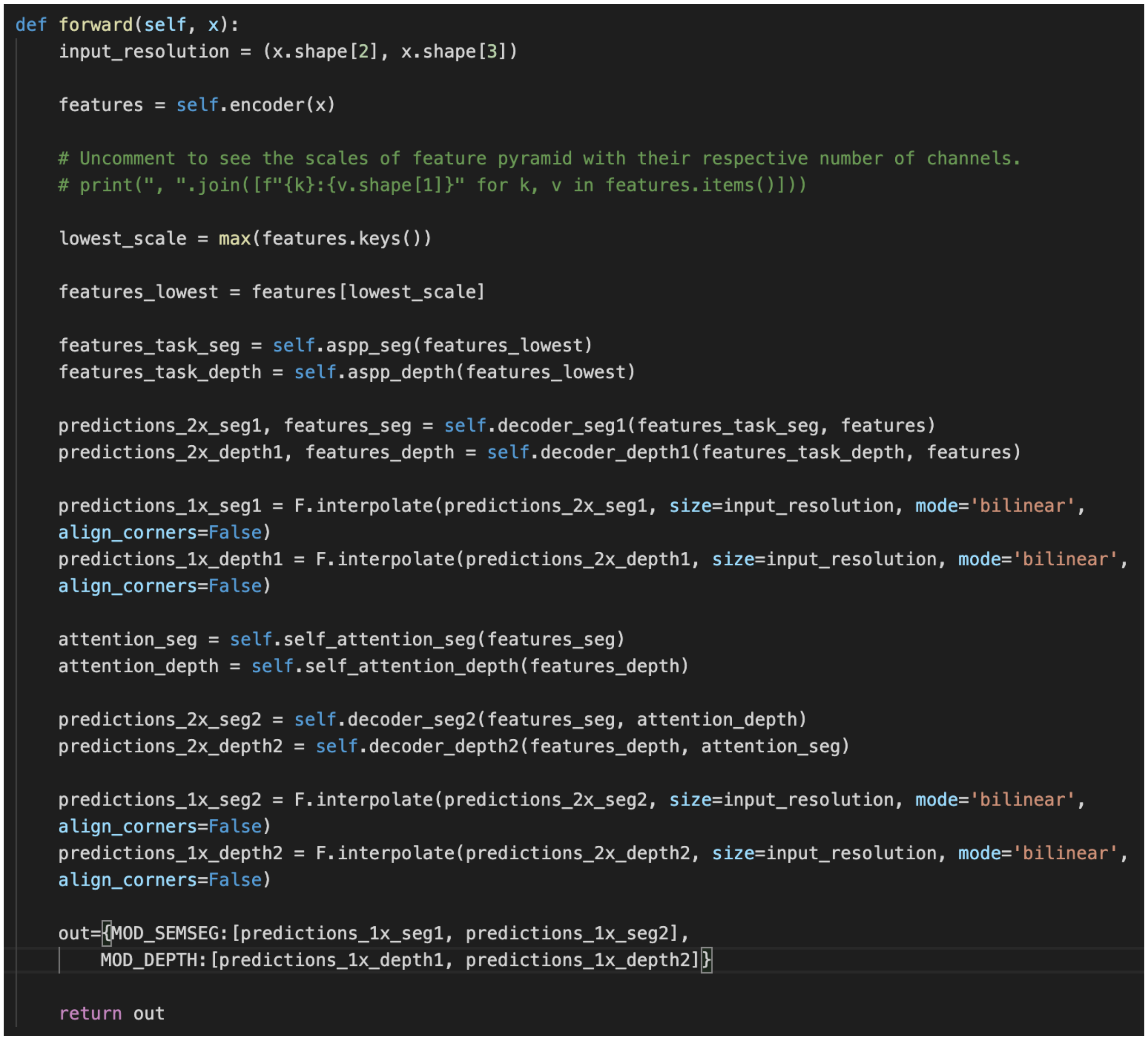


Figure 16: (a) Skip Connection at every scale (Ours) (b) Original Deeplabv3+ Architecture



```
class ModelTaskDistillAllConnect(torch.nn.Module):
    def __init__(self, cfg, outputs_desc):
        super().__init__()
        self.outputs_desc = outputs_desc
        ch_out = sum(outputs_desc.values())
        ch_out_seg=outputs_desc[MOD_SEMSEG]
        ch_out_depth=outputs_desc[MOD_DEPTH]
        ch_attention = 256
        self.add_se = cfg.add_se

        self.encoder = Encoder(
            cfg.model_encoder_name,
            pretrained=True,
            zero_init_residual=True,
            replace_stride_with_dilation=(False, True, 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,cfg)
        self.aspp_depth = ASPP(ch_out_encoder_bottleneck, 256,cfg)

        self.decoder_seg1 = DecoderDeeplabV3pAllConnect(256, 64, ch_out_encoder_4x, 512, ch_out_seg, cfg)
        self.decoder_depth1 = DecoderDeeplabV3pAllConnect(256, 64, ch_out_encoder_4x, 512, ch_out_depth, cfg)

        self.self_attention_seg = SelfAttention(256, ch_attention)
        self.self_attention_depth = SelfAttention(256, ch_attention)

        self.decoder_seg2 = DecoderDeeplabV3pSelfAtten(ch_attention, ch_out_seg)
        self.decoder_depth2 = DecoderDeeplabV3pSelfAtten(ch_attention, ch_out_depth)
```

Figure 17: All skip-connect model initialization Code Snippet



```
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]

    features_task_seg = self.aspp_seg(features_lowest)
    features_task_depth = self.aspp_depth(features_lowest)

    predictions_2x_seg1, features_seg = self.decoder_seg1(features_task_seg, features)
    predictions_2x_depth1, features_depth = self.decoder_depth1(features_task_depth, features)

    predictions_1x_seg1 = F.interpolate(predictions_2x_seg1, size=input_resolution, mode='bilinear',
    align_corners=False)
    predictions_1x_depth1 = F.interpolate(predictions_2x_depth1, size=input_resolution, mode='bilinear',
    align_corners=False)

    attention_seg = self.self_attention_seg(features_seg)
    attention_depth = self.self_attention_depth(features_depth)

    predictions_2x_seg2 = self.decoder_seg2(features_seg, attention_depth)
    predictions_2x_depth2 = self.decoder_depth2(features_depth, attention_seg)

    predictions_1x_seg2 = F.interpolate(predictions_2x_seg2, size=input_resolution, mode='bilinear',
    align_corners=False)
    predictions_1x_depth2 = F.interpolate(predictions_2x_depth2, size=input_resolution, mode='bilinear',
    align_corners=False)

    out={MOD_SEMSEG:[predictions_1x_seg1, predictions_1x_seg2],
        MOD_DEPTH:[predictions_1x_depth1, predictions_1x_depth2]}

    return out
```

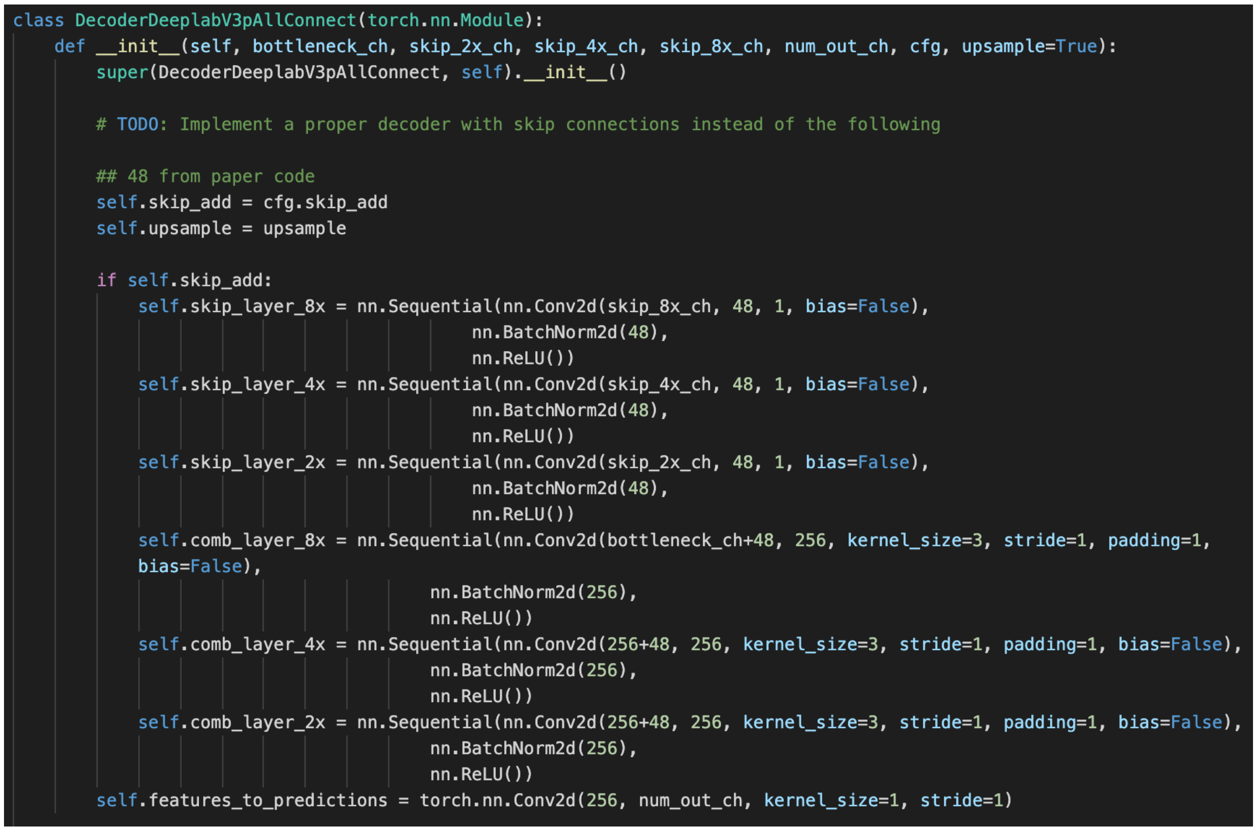
Figure 18: All skip-connect model forward Code Snippet

Table 11 shows the improvement and validates our hypothesis about adding residual connections at all scales.

4.3 Normalizing depth distribution using log transformation

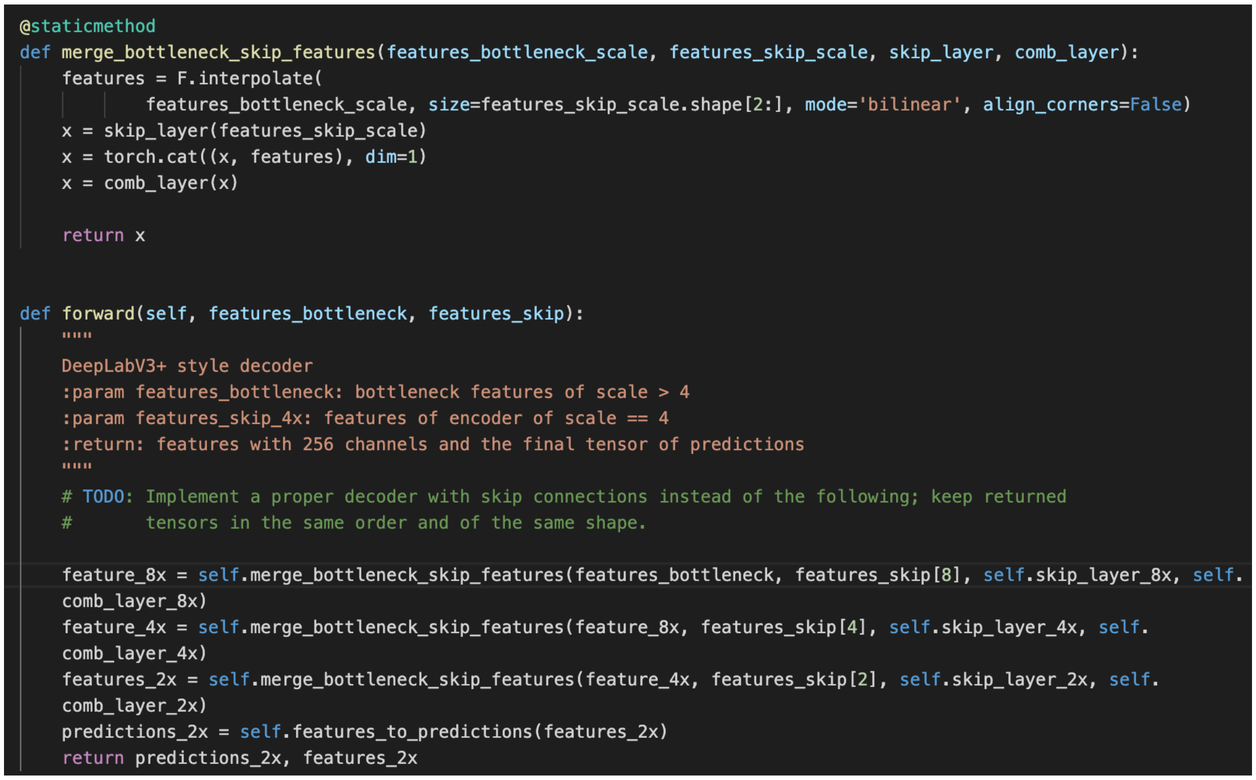
This modification is inspired by the project problem sheet suggestion. As observed in histogram section of Tensorboard, depth values are not in normal distribution, thus directly normalizing leads to skewed distribution as shown in fig. 22a. Since most of the unnormalized values are skewed towards zero, hence we first transform this distribution using **log** transformation and then normalize it using mean and standard deviation of log-transformed distribution. Our log transformation code snippet can be seen in fig. 23. This transformation can be seen in `mtl/utils/transforms.py` file. Due to the behaviour of log function, resultant distribution is much closer to the normal distribution as shown in fig. 22b making depth estimation training much easier for the network.

This modification makes training much easier and results in reduction of depth error. Table 11 show slight improvement especially in depth metrics.



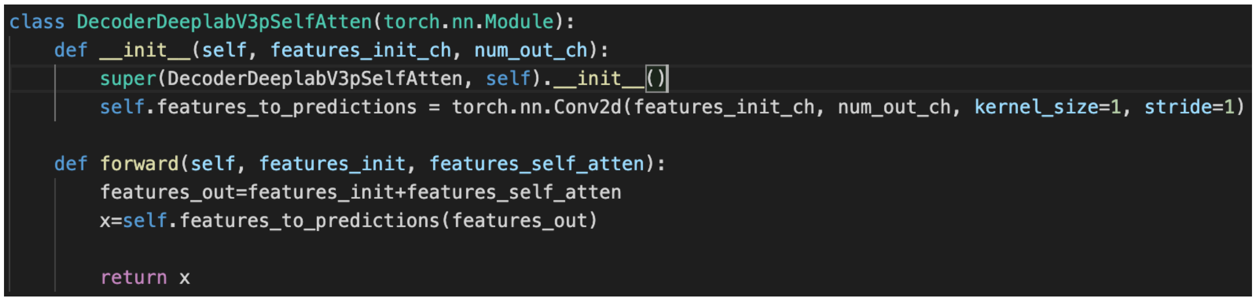
```
class DecoderDeeplabV3pAllConnect(torch.nn.Module):
    def __init__(self, bottleneck_ch, skip_2x_ch, skip_4x_ch, skip_8x_ch, num_out_ch, cfg, upsample=True):
        super(DecoderDeeplabV3pAllConnect, self).__init__()

        # TODO: Implement a proper decoder with skip connections instead of the following

        ## 48 from paper code
        self.skip_add = cfg.skip_add
        self.upsample = upsample

        if self.skip_add:
            self.skip_layer_8x = nn.Sequential(nn.Conv2d(skip_8x_ch, 48, 1, bias=False),
                                                nn.BatchNorm2d(48),
                                                nn.ReLU())
            self.skip_layer_4x = nn.Sequential(nn.Conv2d(skip_4x_ch, 48, 1, bias=False),
                                                nn.BatchNorm2d(48),
                                                nn.ReLU())
            self.skip_layer_2x = nn.Sequential(nn.Conv2d(skip_2x_ch, 48, 1, bias=False),
                                                nn.BatchNorm2d(48),
                                                nn.ReLU())
            self.comb_layer_8x = nn.Sequential(nn.Conv2d(bottleneck_ch+48, 256, kernel_size=3, stride=1, padding=1,
            bias=False),
                                            nn.BatchNorm2d(256),
                                            nn.ReLU())
            self.comb_layer_4x = nn.Sequential(nn.Conv2d(256+48, 256, kernel_size=3, stride=1, padding=1, bias=False),
                                            nn.BatchNorm2d(256),
                                            nn.ReLU())
            self.comb_layer_2x = nn.Sequential(nn.Conv2d(256+48, 256, kernel_size=3, stride=1, padding=1, bias=False),
                                            nn.BatchNorm2d(256),
                                            nn.ReLU())
        self.features_to_predictions = torch.nn.Conv2d(256, num_out_ch, kernel_size=1, stride=1)
```

Figure 19: All skip-connect Decoder #1 initialization Code Snippet



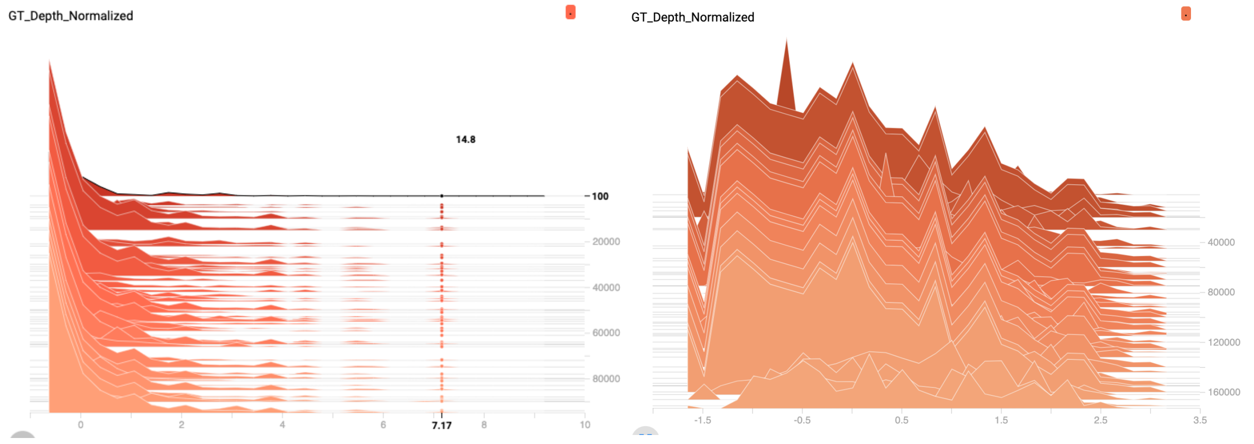
```
@staticmethod
def merge_bottleneck_skip_features(features_bottleneck_scale, features_skip_scale, skip_layer, comb_layer):
    features = F.interpolate(
            features_bottleneck_scale, size=features_skip_scale.shape[2:], mode='bilinear', align_corners=False)
    x = skip_layer(features_skip_scale)
    x = torch.cat((x, features), dim=1)
    x = comb_layer(x)

    return x


def forward(self, features_bottleneck, features_skip):
    """
    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.

    feature_8x = self.merge_bottleneck_skip_features(features_bottleneck, features_skip[8], self.skip_layer_8x, self.
    comb_layer_8x)
    feature_4x = self.merge_bottleneck_skip_features(feature_8x, features_skip[4], self.skip_layer_4x, self.
    comb_layer_4x)
    features_2x = self.merge_bottleneck_skip_features(feature_4x, features_skip[2], self.skip_layer_2x, self.
    comb_layer_2x)
    predictions_2x = self.features_to_predictions(features_2x)
    return predictions_2x, features_2x
```

Figure 20: All skip-connect Decoder #1 forward Code Snippet



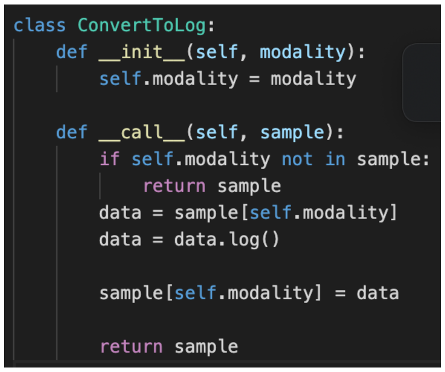
```
class DecoderDeeplabV3pSelfAtten(torch.nn.Module):
    def __init__(self, features_init_ch, num_out_ch):
        super(DecoderDeeplabV3pSelfAtten, self).__init__()
        self.features_to_predictions = torch.nn.Conv2d(features_init_ch, num_out_ch, kernel_size=1, stride=1)

    def forward(self, features_init, features_self_atten):
        features_out=features_init+features_self_atten
        x=self.features_to_predictions(features_out)

        return x
```

Figure 21: All skip-connect Decoder #2 Code Snippett



(a) Raw Normalized depth distribution (b) Log transformed Normalized depth distribution (ours)

Figure 22: Effect of Log transformation on Depth distribution



```
class ConvertToLog:
    def __init__(self, modality):
        self.modality = modality

    def __call__(self, sample):
        if self.modality not in sample:
            return sample
        data = sample[self.modality]
        data = data.log()

        sample[self.modality] = data

        return sample
```

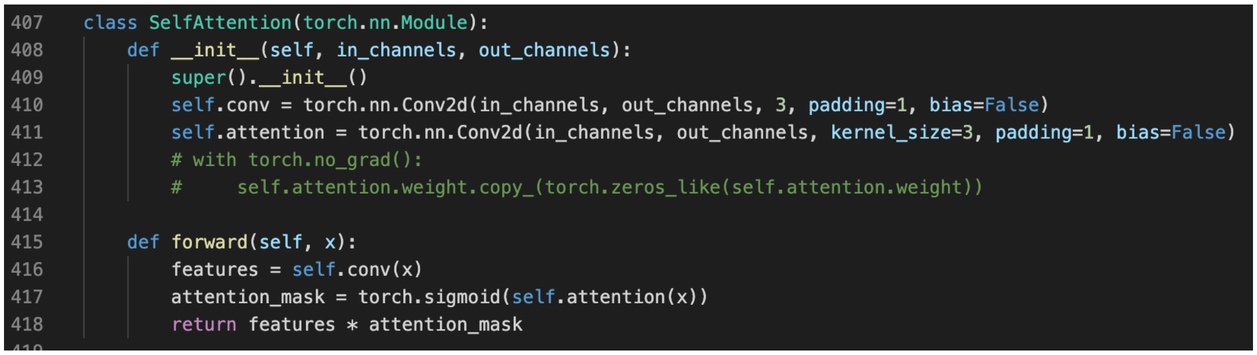
Figure 23: Log transformation code snippet

4.4 L1 loss instead of L2 loss for depth

The key issue with L2 loss is that it does not penalize points closer to zero whereas it penalizes points much farther from origin. In our case due to log transformation, small error in log scale can be significant in actual scale. Thus the motivation of uniform penalization of all points inspire us to use L1 loss instead of L2 loss. Also as per [3], L1 loss works better in low data regime for depth estimation.

4.5 Attention Correction

One important observation we made when reviewing provided template code specially `SelfAttention` class is that weights of attention layer are fixed to zero values and are not getting learned during training. As seen in line 412 and 413 of fig. 24, weights of self attention layer are forced to zero tensor in `torch.no_grad` environment removing it from computation graph. We comment these two lines to make self attention layer parameters learnable.



```
407  class SelfAttention(torch.nn.Module):
408      def __init__(self, in_channels, out_channels):
409          super().__init__()
410          self.conv = torch.nn.Conv2d(in_channels, out_channels, 3, padding=1, bias=False)
411          self.attention = torch.nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1, bias=False)
412          # with torch.no_grad():
413          #     self.attention.weight.copy_(torch.zeros_like(self.attention.weight))
414
415      def forward(self, x):
416          features = self.conv(x)
417          attention_mask = torch.sigmoid(self.attention(x))
418          return features * attention_mask
```

Figure 24: Correction in SelfAttention Module

4.6 Replacing stride with dilation for both 3rd and 4th layer

Replacing-stride-with-dilation configuration in above models was set to `(False,False,True)` which gives 16x down-sampled features to our ASPP module. This configuration provides low resolution features to our ASPP module and model was not able to recognise thin and distant objects such as poles. We added dilation to both 3rd and 4th layer of encoder by changing the configuration to `(False,True,True)` and thus providing 8x down-sampled features (improved resolution) to ASPP module. This helped our model to learn objects which were thin and it improved our segmentation and depth prediction at object boundaries. Fig. 25 shows the modification in network architecture due to change in Replacing-stride-with-dilation configuration. Making this modification results in improving our **grader score from 73.75 to 75.56**.

Model	Muti-task	IOU	SI-logRMSE	Account	TimeStamp
Task Distillation (Base)	69.45	85.92	16.47	apanwar	05/07/2020 18:21:22
Base + SEnet	71.52	87.28	15.75	aabhinav	05/31/2020 06:25:34
Base + SEnet + AllConnect	72.77	87.94	15.17	apanwar	06/01/2020 22:03:46
Base + SEnet + AllConnect + Log normalized depth + L1 loss + Attention Correction	73.76	87.86	14.10	aabhinav	06/05/2020 20:06:21
Base + SEnet + AllConnect + Log normalized depth + L1 loss + Attention Correction + Dilation	**75.56**	**88.98**	**13.42**	apanwar	06/11/2020 21:07:26

Table 11: Impact of model performance due to our novel idea implementations. All these experiments are done for 30 epochs with initial learning rate of 5×10^{-5} (Best Model "Rank 7" is highlighted in Bold)

References

[1] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," *CoRR*, vol. abs/1709.01507, 2017. [Online]. Available: http://arxiv.org/abs/1709.01507

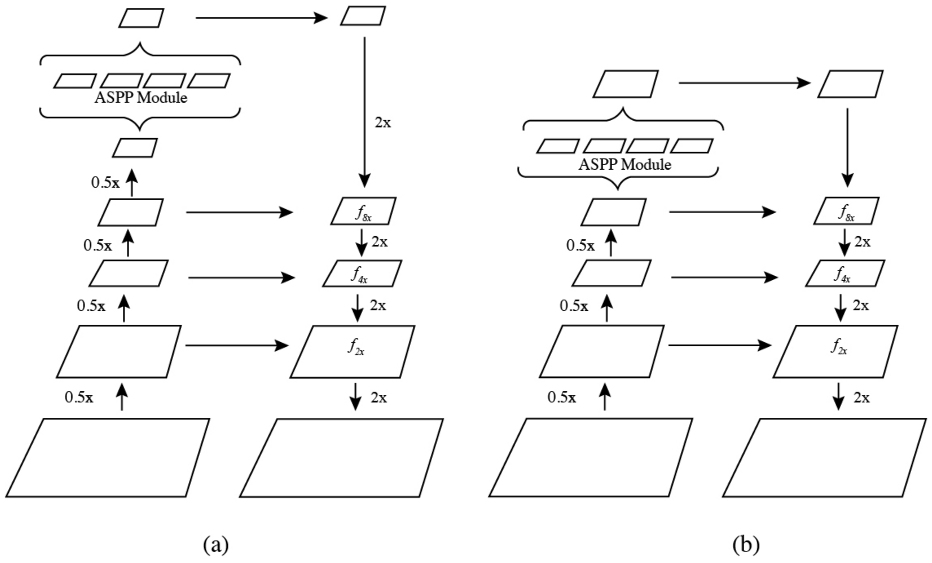


Figure 25: (a) Decoder architecture with output stride of 16 by replacing stride with dilation for only 4^{th} layer (b) Decoder architecture with output stride of 8 by replacing stride with dilation for both 3^{rd} and 4^{th} layer (Ours)

[2] L. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation," *CoRR*, vol. abs/1802.02611, 2018. [Online]. Available: http://arxiv.org/abs/1802.02611

[3] M. Carvalho, B. L. Saux, P. Trouvé-Peloux, A. Almansa, and F. Champagnat, "On regression losses for deep depth estimation," *ICIP*, 2018. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01925321/document