DEEP LEARNING FOR AUTONOMOUS DRIVING

Multi-Task Learning for Semantics and Depth

TEAM 19: Argenziano Italo Nicholas, Pagani Leonardo

May 13, 2022

1 Joint architecture

1.1 Hyperparameter tuning

(a) *Optimizer and learning rate*

The question requires to analyze the performance of the network using different optimizers and learning rates. It is well known that the learning rate itself is the most important hyperparameter to tune when training deep neural networks. In the gradient descent optimization, large learning rates can lead to overshooting and to the divergence of the algorithm - gradient descent might fail to find a stable global minimum for the weight parameters. Generally speaking, a learning rate that is too large will result in weight updates that will be too large and the performance of the model (such as its loss on the training dataset) will oscillate over training epochs. On the other hand, small learning rates can considerably slow down the training process and might even lead to suboptimal results, as the algorithm may get stuck in local minima. Several methods have been suggested in the literature to find the optimal learning rates ([1]). In this task, however, we were asked to analyze the performance of the network with different learning rates. A simple approach, suggested in the assignment, is to perform a sensitivity analysis of the learning rate for the chosen model, also called the grid-search. This can help to both highlight an order of magnitude where good learning rates may reside, as well as describe the relationship between learning rate and performance. We decided to grid search learning rates on a logarithmic scale from 10^{-1} to 10^{-5}. Below are the results with the classical SGD optimizer.

Analysis of learning rate with SGD			
Learning rate α	Multitask score	Semseg score	Depth score
10^{-1}	**42.947**	69.776	26.829
10^{-2}	**40.294**	67.232	26.938
10^{-3}	**23.338**	55.643	32.305
10^{-4}	**9.487**	25.631	40.513
10^{-5}	**0.9332**	15.321	49.067

Clearly, the most suitable learning rate with the SGD optimizer is $\alpha = 10^{-1}$, as it stands out in the validation performance.

Other experiments were instead carried out using the popular Adam optimzer ([2]). Stochastic gradient descent maintains a single learning rate (termed alpha) for all weight updates and the learning rate does not change during training. In Adam, a learning rate is maintained for each network weight (parameter) and separately adapted as learning unfolds. Overall, Adam is an Adaptive learning rate algorithm, which calculates the exponential moving average of the gradient and the squared gradient (the parameters β_1 and β_2 control the decay rates of these moving averages. When introducing the algorithm, the authors ([2]) list several attractive benefits of using Adam on non-convex optimization problems, such as its efficiency and its little memory requirement. In our quest to determine the proper learning rate, the parameters β_1 and β_2 were set to their default values $\beta_1 = 0.9$, $\beta_2 = 0.999$. Again, a grid search approach was followed to analyze the learning rates.

Analysis of learning rate with Adam			
Learning rate α	Multitask score	Semseg score	Depth score
1	**0**	8.319	81.2
10^{-1}	**0**	10.768	56.999
10^{-2}	**12.491**	40.052	37.509
10^{-3}	**38.121**	66.099	27.978
10^{-4}	**43.171**	69.803	26.632
10^{-5}	**24.038**	56.391	32.353

It is well known that optimal learning rates are usually a couple of orders of magnitude smaller for Adam. In this case we find that the optimal learning rate to be $\alpha = 10^{-4}$.
We have already discussed the structural differences between the two optimizers.
There is a lot of literature performing empirical comparisons of optimizers for deep learning ([3],[4],[5],[6]). Adam is usually considered faster than SGD (although in our task, they both complete the training in approximately 4h 50 minutes). Moreover, adaptive methods (such as Adam) often display faster initial progress on the training set, but their performance quickly plateaus on the validation set. This observation is confirmed in

our experiments where we can observe that the training loss is lower when Adam is used. We notice that the best performance in the validation set is indeed obtained by using the Adam optimizer while setting $\alpha = 10^{-4}$.

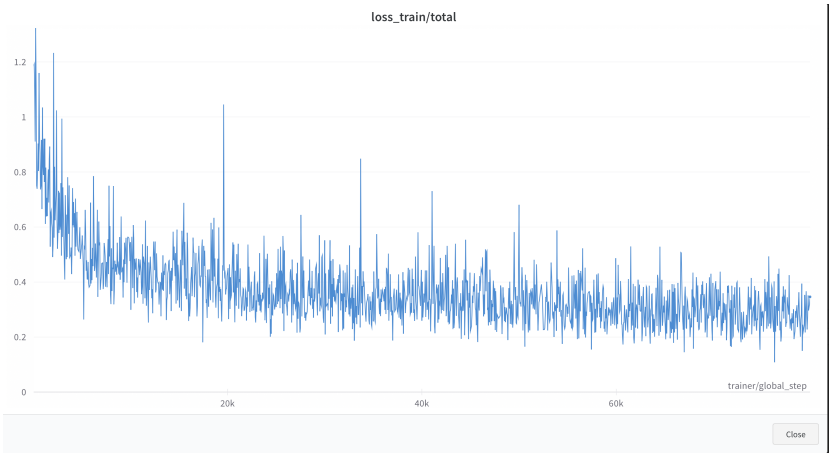


Figure 1: Training loss with Adam optimizer and $\alpha = 10^{-4}$.

Figure 2: Training loss with Adam optimizer and $\alpha = 10^{-1}$.

In the figures above we can effectively visualize the effect the learning rate. In *Figure 1*, we observe a correct behaviour of the training loss (when setting the best performing learning rate). In *Figure 2*, the learning rate is too large and does not guarantee any convergence of the training loss.

In *Figure 3* we can analyze the behaviour of the training loss whenever

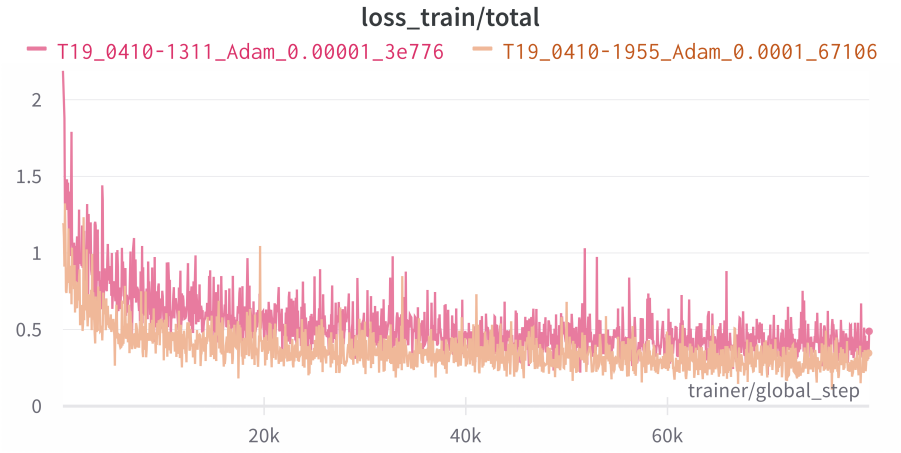


Figure 3: Training loss with Adam optimizer and $\alpha = 10^{-4}$ (brown) and $\alpha = 10^{-5}$ (pink).

the learning rate is too small. In this case, the loss is slow to converge to the optimal value. As suggested in the handout, from now on we use the Adam optimizer with a learning rate of 10^{-4}.

(b) *Batch size*

To understand the results of this task it is first necessary to make some clarifictions on the meaning of **epoch** and **batch size**. In the framework of training a neural network, the batch size is the number of samples processed before the model is updated. The number of epochs, instead, is the number of complete passes through the training dataset. The size of a batch must be more than or equal to one and less than or equal to the number of samples in the training dataset. The number of epochs can be set to an integer value between one and infinity. Batch size and epochs are hyperparameters of the training model and therefore have to be chosen by analyzing the validation loss. It is a common choice, whenever using a GPU, to set the batch size to a power of 2 to guarantee better runtime. We now provide an intuition on why it is necessary to increase the number of epochs proportionally to the batch size. As outlined before, the batch size is the number of samples after which an update of the weights is performed. That means that for every epoch e, the number of updates $\mathrm{U_e}$ will be:

$$U_e = \frac{|\mathbf{D}|}{\mathbf{B}}$$

where $|\mathbf{D}|$ is the number of samples in the training dataset D and B represents the batch size. It is clear that the number of updates U during an epoch decreases as the batch increases. Let us now denote with E the number of training epochs. The total number of updates U_{tot}, across all the epochs can be computed as:

$$U_{\mathrm{tot}} = \sum_e U_e = E * \frac{|\mathbf{D}|}{\mathbf{B}}$$

Therefore, since the cardinality of the dataset D does not change, in order to maintain the number of total updates constant, the number of epochs E and the batch size have to be tuned proportionally. Let us now evaluate the validation performance of our model for different values of the epochs and of the batch size.

Analysis of epochs and batch size		
Epochs - Batch Size	Multitask score	Training time
$8-2$	37.751	3h 31m 5s
$16-4$	43.171	4h 53m 56s
$32-8$	46.287	7h 58m 25
$64-16$	48.193	14h 2m 1s

It seems here that increasing the number of epochs and the batch size is beneficial for the validation loss.
Although there are not rigorous theoretical reasons to choose a certain

batch size over another, in most deep learning models the usual batch size oscillates between 32,64,128 and 256 ([7]). Therefore, in our experiments we have always opted for a batch size that is generally considered quite small. We decided not to experiment bigger batch sizes (and more epochs) due to training time issues. We have previously explained that the number of updates during the training is kept constant with our choice of the hyperparameters. However, it seems that the performance have improved consistently by increasing the epochs and the batch size. The improvement of the validation metrics might be justified by the fact that a too small batch size yields very noisy and stochastic estimates of the gradient, leading the training to converge to unreliable models. Small batch sizes may lead to slow convergence of the learning algorithm. Training samples are randomly drawn from the training set every step and therefore the resulting gradients are noisy estimates based on partial data. The fewer samples we use in a single batch, the noisier and less accurate the gradient estimates will be. That is, the smaller the batch, the bigger impact a single sample has on the applied variable updates. Larger batch sizes instead provide better approximations of the true gradient.

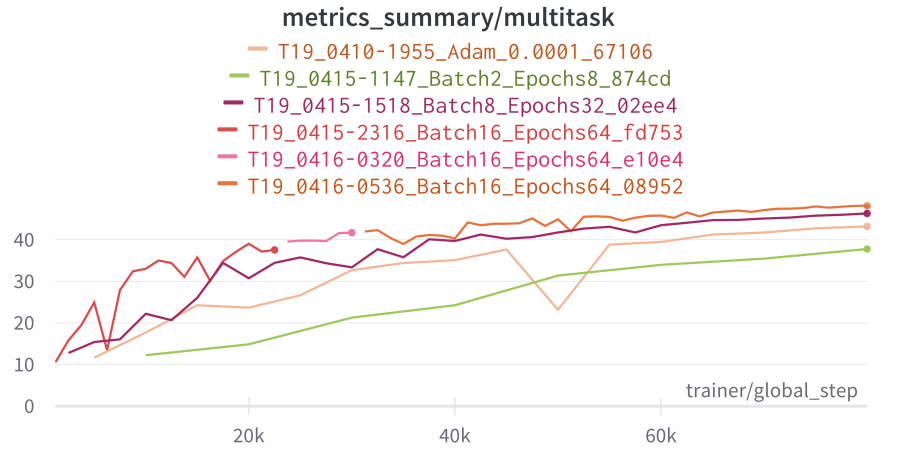


Figure 4: Comparison of validation metrics with different epochs and batch sizes. On the x-axis there are the training steps.

In *Figure 5* we can observe how it is necessary to increase the batch size as we increase the number of epochs. We see that if the number of epochs remains the same, the better performances are achieved by the model with lower batch sizes as they are able to perform more training steps (gradient updates).

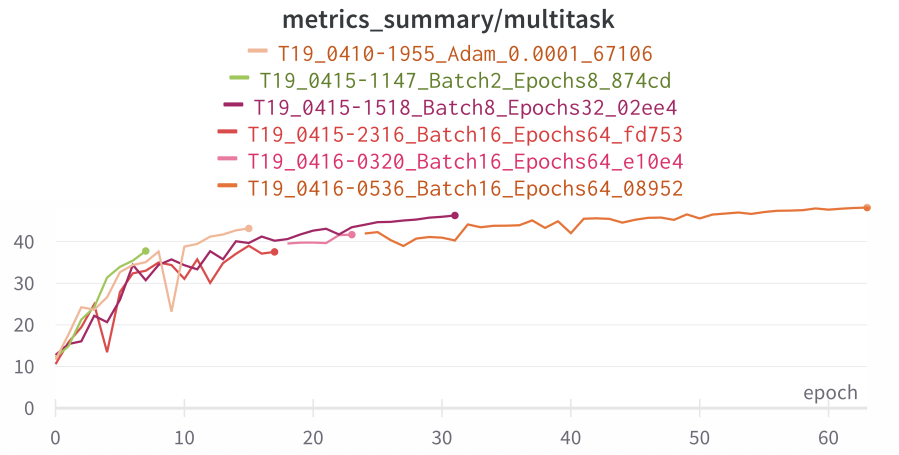


Figure 5: Comparison of validation metrics with different epochs and batch sizes. On the x-axis there are the epochs.

Another relevant aspect to keep in consideration when setting these hyperparameters might be the **training time**. As outlined in the previous table, the training time increases with the batch size (provided that the number of training steps remains the same). Theoretically, we would expect the computation of the gradient to be roughly linear in the batch size ($\mathcal{O}(\mathbf{B})$). However, from our table, it looks like that the dependence between the training time and the batch size is actually slightly sublinear. This is probably due to linear algebra libraries using use vectorization for vector and matrix operations to speed them up, at the expense of using more memory.

Finally the last aspect to evaluate when tuning the batch size is **memory allocation** ([]8). The larger the batch size, the more samples are being propagated through the neural network in the forward pass. This results in larger intermediate calculations (e.g. layer activation outputs) that need to be stored in GPU memory.It is now clearly noticeable that increasing the batch size will directly result in increasing the required GPU memory. In many cases, not having enough GPU memory prevents us from increasing the batch size. In *Figure 6*, the GPU memory allocated with different batch sizes across time is shown.

As suggested in the handout, from now on we set the number of epochs to 16 and the batch size to 4.

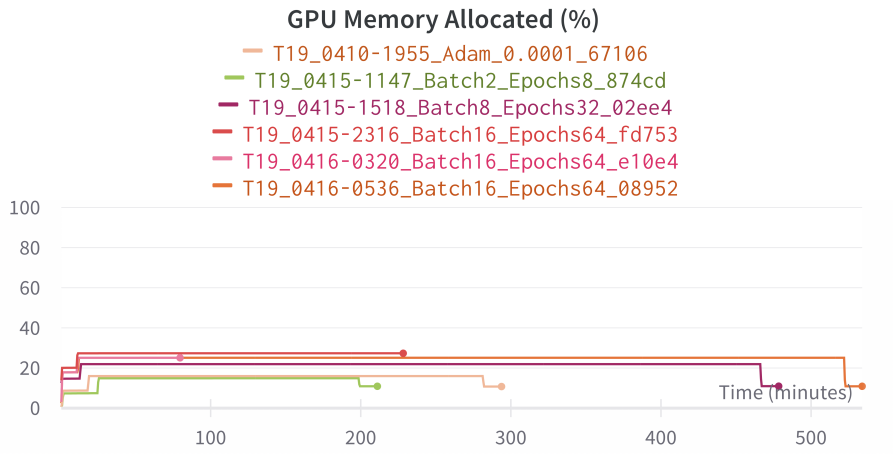


Figure 6: Process GPU Memory Allocated (%)

(c) *Task weighting*

The total training loss when performing training is computed as:

$$Training\ loss = \mathrm{W}_{\mathrm{seg}} * Loss_{\mathrm{seg}} + \mathrm{W}_{\mathrm{depth}} * Loss_{\mathrm{depth}}$$

By default, $Loss_{\mathrm{seg}}$ is set as the *Cross-Entropy Loss*, while $Loss_{\mathrm{depth}}$ equals the standard *L2* regression loss. Changing the two weights $\mathrm{W}_{\mathrm{seg}}$ and $\mathrm{W}_{\mathrm{depth}}$ has significant implications on the performance of the model.

Analysis of the task weights			
Task weights $Loss_{\mathrm{seg}}$ - $Loss_{\mathrm{depth}}$	Multitask score	Semseg score	Depth score
$0.2 - 0.8$	**41.097**	67.283	26.186
$0.4 - 0.6$	**43.158**	69.421	26.263
$0.5 - 0.5$	**43.171**	69.803	26.632
$0.6 - 0.4$	**43.54**	70.293	26.753
$0.8 - 0.2$	**42.324**	70.439	28.115

We have decided to try different combinations of the values of $\mathrm{W}_{\mathrm{seg}}$ and $\mathrm{W}_{\mathrm{depth}}$. As expected, by increasing the weight related to one task, the score for such task improves, while the score associated to the other task drops. Overall, it appears that the best multitask score is obtained by giving slightly more contribution to the classification cross entropy loss, with a task weight of 0.6, while keeping the weight of the L2 regression loss at 0.4. Furthermore, we notice that if we make the total loss unbalanced

towards one of the two task, the overall performance rapidly deteriorates. We did not even try the combination of weights 1-0 / 0-1 as that would mean to completely disregard one task. The choice of the task weights depends of course on our application and therefore on the metrics we choose to adopt to evaluate the performances. In our case the combined evaluation metric is given by:

$$\textit{Multitask-score} = max(IoU - 50, 0) + max(50 - SIlogrmse, 0)$$

where IoU and *SIlogrmse* are respectively the metrics associated with the semantic segmentation and depth estimation tasks. These two metrics evenly contribute to the total evaluation metric and it is therefore a wise choice to keep the weights $\mathrm{W}_{\mathrm{seg}}$ and $\mathrm{W}_{\mathrm{depth}}$ both at 0.5. Again, we can think as the task weights as hyperparameters of the models, thus to be tuned with the validation set. As suggested in the handout, from now on $\mathrm{W}_{\mathrm{seg}}$ and $\mathrm{W}_{\mathrm{depth}}$ will be both kept to 0.5.

1.2 Hardcoded hyperparameters

(a) *Encoder initialization*

The encoder of our architecture is built with the provided off the shelf module by `torchvision.models.resnet`. Basically, our encoder follows the structure of the *Resnet34* architecture as outlined in ([9]). It consists in a concatenation of residual blocks as shown in *Figure 7*.

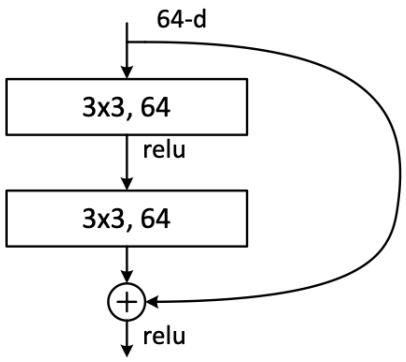


Figure 7: Building residual block of ResNet34

This block is built by two layers of 3 x 3 convolutions, each followed by a batch normalization module. At the end of the block, a ReLU function is used as the activation function. The two convolutional layers, as well as the batch normalization layers have learnable parameters which must be initialized to start the training. The 2D convolutional layers are initialized by using **Kaiming initialization**. In his paper ([10]), Kaiming et al propose their own initialization scheme that is tailored for deep neural nets that use these kinds of asymmetric, non-linear activations. In particular the weights' tensors are randomly initialized from a zero mean Gausssian distribution $\mathcal{N}(0, \frac{2}{n_l})$. Here n_l denotes the number of incoming connections into layer l. Bias tensors instead are initialized to zero. Kaiming initialization aims at cautiously modeling non - linearity of ReLUs, which makes deep models (> 30 layers) converge. By choosing Kaiming initialization we are able to avoid both problems of vanishing gradient and exploding gradient.

In batch normalization modules, the data distribution is normalized so that the subsequent layers have a solid ground to continue. It is inappropriate, however, to have zero mean and unit variance for every layer as that

would limit the expressive power of the network. Each *Batch Norm* module has two learnable parameters, controlling the mean and the variance of the distribution. Let us define them as γ and β. Batch normalization layers work as follows:

$$y = \frac{x - \mathbf{E}[x]}{\sqrt{\mathbf{Var}[x] + \epsilon}} * \gamma + \beta$$

Figure 8: Batch normalizing transform, applied over a minibatch x

The parameters γ and β are respectively known as the *weight* and the *bias* of the batch normalization module. In the *ResNet34* encoder architecture they are respectively initialized with $\gamma = 1$ and $\beta = 0$ for all batch normalization layers. Below is the code snippet from `torchvision.models.resnet` providing details on parameters' initialization.

```
for m in self.modules():
    if isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(m.weight, mode = 'fan out',
        ↪  nonlinearity = 'relu')
    elif isinstance (m, (nn.BatchNorm2d, nn.GroupNorm)):
    nn.init.constant_(m.weight,1)
    nn.init.constant_(m.bias,0)
```

A common solution to enhance the training for a certain task is to initialize the weights of certain layers with weights trained for another task. When switching the *pre-trained* flag to *True*, we are de facto performing **transfer learning**. Instead of starting training from scratch, the network is initialized with pre-trained weights and it is therefore able to accelerate and improve the convergence. In particular, by setting the *pretrained* flag to *True*, `torchvision.models.resnet` returns a model pre-trained on the *ImageNet* database. Let us now analyze how the performances change when switching the *pre-trained* flag to *True*.

Analysis of the effect of pre-training			
Pretrained flag	Multitask score	Semseg score	Depth score
False	**43.171**	69.803	26.632
True	**47.206**	72.762	25.556

Clearly, the performances of the network improve considerably when the *pre-trained* flag is set to *True* for the other experiments. In particular,the

network does a significantly better job in the semantic segmentation task. It seems that the encoder parameters (pre-trained on the *ImageNet*) have helped improved the performance. The low level features of the images of the two datasets clearly have something in common as the same set of weights seem to correctly adapt to both of them. In *Figure 9* we show how the validation loss in the pretrained model is already very low from the start even when the model has not even been trained on the *Miniscapes* dataset. The pretrained model just starts from an advanced stage of the training and during the 16 training epochs is basically **fine-tuning** the parameters to better adapt to our MTL framework. It is also evident how the difference between the two models is shrinking as the training goes on. It looks like the two models will eventually converge to the same asymptote. However, the not-pretrained model (orange) would probably need a few more fine-tuning epochs to reach the validation loss of the pretrained model. That is the reason why tranfer learning is also useful for saving time during training. For the remaining experiments the *pre-trained* flag is set to *True*.

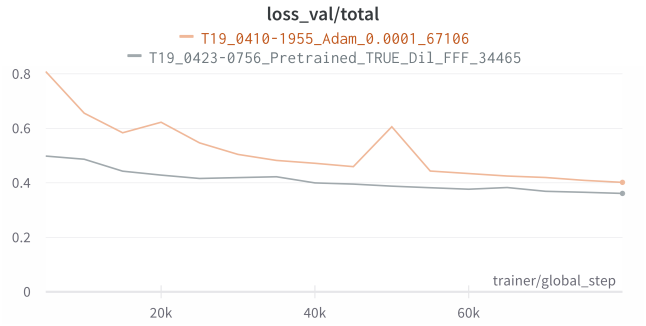
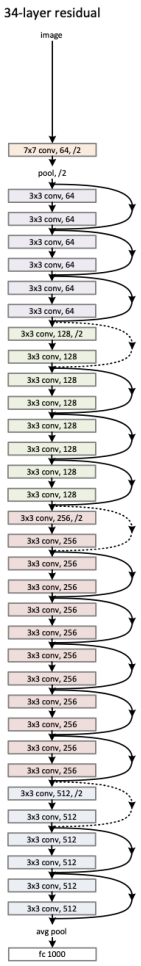


Figure 9: Total validation loss of the pretrained model (grey) against the not-pretrained model (pink)

(b) *Dilated convolutions*



As outlined before, our encoder is built using the provided module from `torchvision.models.resnet`. In particular, a *ResNet34* is implemented. The *Resnet34* encoder is constructed by 4 layers of residual blocks with 3x3 convolutions. Each of these 4 layers contain respectively 3,4,6, and 3 building residual blocks (*Figure 7*). In the figure on the left, we can see the reconstruction of the entire encoder, as outlined in ([11]). By setting the dilations flag to (False, False, True) we replace the strided convolution with a dilated one with rate 2 in all the building blocks of the last layer. If had set all the flags to True, we would have applied a dilation with rate 2 to layer 2,3 and 4. The performance of the network significantly improves after this adjustment. By the dilated convolution, the receptive field of the encoder is expanded. The network is now able to retrieve much better spatial information, as it can infer the context from a wider neighborhood. The features extracted by the encoder are indeed more meaningful and valuable. By passing them to the decoder, we are able to perform better both for the semantic segmentation and for the depth estimation task.

Analysis of the dilation flags' effects			
Dilation flags	Multitask score	Semseg score	Depth score
False False False	**47.206**	72.762	25.556
False False True	**58.578**	79.964	21.386

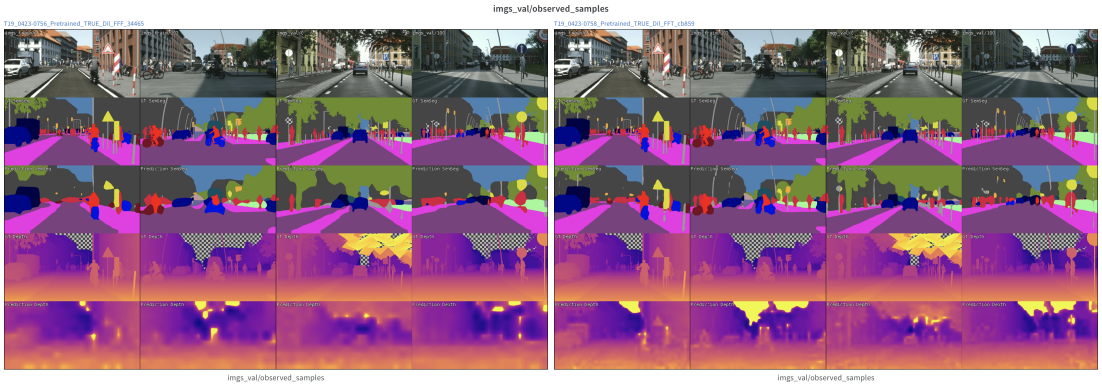


Figure 10: Validation predictions before(left)and after(right) setting the last dilation flasg to True.

From *Figure 10*, we notice that after the activation of the dilated convolution in the last layer of the encoder, the predictions of the network for the validation set are much more refined. In particular, for the semantic segmentation task, we can observe that the boundaries of all the objects become much more distinguishable. This is true specially for the smallest and most peculiar objects in the images (fence, truck,pole,..). Let us recall that the *Mean Intersection Over Union* score for semantic segmentation is a quite severe metric, as it assigns the same weight to predictions onto rare and common objects. Therefore, achieving much better IoU scores for every object is decisive to improve the *mIoU* score. Even the predictions for the depths are more detailed and refined, whereas the results of the previous model (no dilation) are coarser. From now on, the last dilation flag will set to *True* as suggested in the handout.

1.3 ASPP and skip connections

The ASPP module consists of a pyramid of convolutional filters applied to the feature map output by the encoder. Specifically, the aforementioned output is processed in parallel (i.e., not sequentially) by a classical 1x1 convolution, three 3x3 dilated convolutions, and a module defined Image Pooling. As mentioned before, the goal of dilated convolutions is to ensure that the convolutional operation has a larger receptive field (thus capturing more relevant spatial information) without excessively reducing the image resolution through several convolutions. The dilation rate can be considered as a hyparameter that regulates the scale of such an operation. Performing dilated convolutions with different rates allows to merge spatial information acquired at different scales, making the feature map more spatially meaningful. Additionally, the Image Pooling block is used to incorporate global context information.
The Image Pooling module, introduced in ([17]), is structured as follows: a Global Average Pooling layer performs a channel-wise average, preserving the number of channels and reducing the width and height dimensions to a single scalar. The resulting image-level features are then fed into a 1x1 convolution, which performs a linear combination across channels, and then bilinearly upsampled back to the pre-ASPP resolution.
The five resulting maps are then concatenated along the channel dimension and processed by a 1x1 convolution, which retrieves the original number of channels (256). Below is our implementation of the ASPP module. All operations that did not require the input resolution are defined in the `__init__` method, whereas the AvgPool and Upsample layers are introduced directly in `forward`.

```
370 class ASPP(torch.nn.Module):
371     def __init__(self, in_channels, out_channels, rates=(3, 6,
        ↪  9)):
372         super().__init__()
373         self.conv_out = torch.nn.Sequential(
374             torch.nn.Conv2d(in_channels, out_channels,
                ↪  kernel_size=1, stride=1, padding=0, dilation=1,
                ↪  bias=False),
375             torch.nn.BatchNorm2d(out_channels),
376             torch.nn.ReLU()
377         )
378         self.dil0_out = torch.nn.Sequential(
379             torch.nn.Conv2d(in_channels, out_channels,
                ↪  kernel_size=3, stride=1, padding=rates[0],
                ↪  dilation=rates[0], bias=False),
380             torch.nn.BatchNorm2d(out_channels),
381             torch.nn.ReLU()
382         )
383         self.dil1_out = torch.nn.Sequential(
```

```
            torch.nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=rates[1], dilation=rates[1], bias=False),
            torch.nn.BatchNorm2d(out_channels),
            torch.nn.ReLU()
        )
        self.dil2_out = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=rates[2], dilation=rates[2], bias=False),
            torch.nn.BatchNorm2d(out_channels),
            torch.nn.ReLU()
        )
        self.conv_bn_out = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=1, padding=0, dilation=1, bias=False),
            torch.nn.BatchNorm2d(out_channels),
            torch.nn.ReLU(),
        )
        self.conv_bn_out_fin = torch.nn.Sequential(
            torch.nn.Conv2d(5*out_channels, out_channels, kernel_size=1, stride=1, padding=0, dilation=1, bias=False),
            torch.nn.BatchNorm2d(out_channels),
            torch.nn.ReLU(),
        )

    def forward(self, x):
        input_res = x.shape[2:]
        conv_out = self.conv_out(x)
        dil0_out = self.dil0_out(x)
        dil1_out = self.dil1_out(x)
        dil2_out = self.dil2_out(x)
        avgpool = torch.nn.AvgPool2d(input_res)
        img_pool_out = avgpool(x)
        img_pool_out = self.conv_bn_out(img_pool_out)
        upsample = torch.nn.Upsample(size=input_res, mode='bilinear')
        img_pool_out = upsample(img_pool_out)
        out = torch.cat((conv_out, dil0_out, dil1_out, dil2_out, img_pool_out), dim=1)
        out = self.conv_bn_out_fin(out)
        return out
```

Looking at the architecture in ([17]), noting in particular that the defined *output_stride* parameter does not change before and after the ASPP module, one can conclude that the map resolution must not be altered by the ASPP. This is why the dilated convolutions require an adequate padding. Specifically, the following relation holds:

$$W_{out} = \frac{[W_{in}+2p-K-(K-1)(d-1)]}{s} + 1$$

Here W_{out} and W_{in} are respectively the output and input width, p is the padding, K is the kernel size, d is the dilation rate, and s is the stride. The same holds for the height dimension. By setting $W_{out} = W_{in}, K = 3, s = 1$, the above relation boils down to $p = d$.
Another popular method to address the loss of resolution and sharpness associated to the encoding and decoding performed by fully convolutional networks is adding skip connections. Low-level features from shallow layers of the encoder, which are rich in spatial information, are concatenated with higher level features from deeper layers (in this case, after the bottleneck and ASPP module), which encode semantic information and have a larger receptive field. This merging of spatial ("where") and semantic ("what") information yields a much sharper output map, whose boundaries are far less coarse.

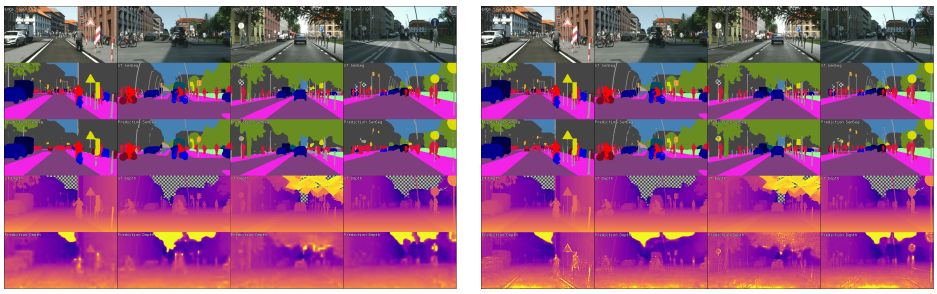


Figure 11: Validation predictions before (left) and after (right) adding ASPP and skip connections. Both configurations used pretrained weights and replace the last strided convolution with a dilated one.

Below is our code for the decoder with skip connections.

```
296 class DecoderDeeplabV3p(torch.nn.Module):
297     def __init__(self, bottleneck_ch, skip_4x_ch, num_out_ch):
298         super(DecoderDeeplabV3p, self).__init__()
299 
300         self.conv1_bn = torch.nn.Sequential(
301             torch.nn.Conv2d(skip_4x_ch, 48, kernel_size=1,
                ↪  stride=1, padding=0, dilation=1, bias=False),
302             torch.nn.BatchNorm2d(48),
303             torch.nn.ReLU(),
304         )
```

```
305
306         self.conv3_bn = torch.nn.Sequential(
307             torch.nn.Conv2d(48+bottleneck_ch, bottleneck_ch,
                ↪  kernel_size=3, stride=1, padding=1, dilation=1,
                ↪  bias=False),
308             torch.nn.BatchNorm2d(bottleneck_ch),
309             torch.nn.ReLU(),
310             torch.nn.Conv2d(bottleneck_ch, bottleneck_ch,
                ↪  kernel_size=3, stride=1, padding=1, dilation=1,
                ↪  bias=False),
311             torch.nn.BatchNorm2d(bottleneck_ch),
312             torch.nn.ReLU(),
313         )
314
315         self.features_to_predictions =
            ↪  torch.nn.Conv2d(bottleneck_ch, num_out_ch,
            ↪  kernel_size=1, stride=1)
316
317     def forward(self, features_bottleneck, features_skip_4x):
318
319         # DeepLabV3+ style decoder
320         # :param features_bottleneck: bottleneck features of
            ↪  scale > 4 (coming from ASPP)
321         # :param features_skip_4x: features of encoder of scale
            ↪  == 4 (coming from solid blue block)
322         # :return: features with 256 channels and the final
            ↪  tensor of predictions
323
324         features_4x = F.interpolate(
325             features_bottleneck, size=features_skip_4x.shape[2:],
                ↪  mode='bilinear', align_corners=False
326         )
327         conv_skip = self.conv1_bn(features_skip_4x)
328         features_4x = torch.cat((features_4x, conv_skip), dim=1)
329         features_4x = self.conv3_bn(features_4x)
330         predictions_4x =
            ↪  self.features_to_predictions(features_4x)
331
332         return predictions_4x, features_4x
```

Following the directions and experiments of ([17]), we fed the low-level features into a 1x1 convolution with 48 filters before concatenating it with the upsampled bottleneck features. The resulting tensor is then processed by two 3x3 convolutions with BatchNorm and ReLU, so that the number of channels is again 256. Then, a final 1x1 convolution returns the predictions for both tasks in a tensor with 20 channels, where the first 19 correspond to the number of

semantic classes and the last one is used for the depth prediction.
After adding ASPP and skipped connections to the model, the validation performance increases significantly in both tasks. Specifically, the depth score decreases by 14% and the semseg score grows by 6.6%.

Run name	Multitask score	Semseg score	Depth score
`Pretrained_TRUE_Dil_FFT_cb85`	**58.578**	79.964	21.386
`task1.3_relu7458e`	**66.907**	85.260	18.353

2 Branched architecture

The goal of the second task was to implement a branched architecture with a shared encoder and separate ASPP modules and decoders. Using the same building blocks previously implemented, we composed the architecture as follows (`mtl/models/model_deeplab_branched.py`).

```
class ModelDeepLabBranched(torch.nn.Module):
    def __init__(self, cfg, outputs_desc):
        super().__init__()
        self.outputs_desc = outputs_desc

        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)

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

        self.decoder_seg = DecoderDeeplabV3p(256, ch_out_encoder_4x, outputs_desc['semseg'])
        self.decoder_dep = DecoderDeeplabV3p(256, ch_out_encoder_4x, outputs_desc['depth'])

    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_tasks_seg = self.aspp_seg(features_lowest)
        features_tasks_dep = self.aspp_dep(features_lowest)
```

```
41
42          predictions_4x_seg, _ =
            ↪  self.decoder_seg(features_tasks_seg, features[4])
43          predictions_4x_dep, _ =
            ↪  self.decoder_dep(features_tasks_dep, features[4])
44
45          predictions_1x_seg = F.interpolate(predictions_4x_seg,
            ↪  size=input_resolution, mode='bilinear',
            ↪  align_corners=False)
46          predictions_1x_dep = F.interpolate(predictions_4x_dep,
            ↪  size=input_resolution, mode='bilinear',
            ↪  align_corners=False)
47
48          out = {'semseg': predictions_1x_seg, 'depth':
            ↪  predictions_1x_dep}
49
50          return out
```

Similarly as task 1.3, each of the two decoders takes as inputs the bottleneck features (processed by task-specific ASPP modules) and the low-level features of scale 4 from the encoder. The output is a tensor with a number of channels equal to `outputs_desc['semseg']` and `outputs_desc['depth']` respectively. These two tensors are inserted in the `out` dictionary, which is the final output of the model.
Comparing the performances of the joint architecture of task 1.3 and of the aforementioned branched architecture, one can note a slight improvement in the semseg and a substantial reduction (6.5%) of the depth score. On the other hand, due to the introduction of separate trainable structures, the number of learnable parameters grows significantly, and so do the training time and GPU memory usage.

Run name	Multitask score	Semseg score	Depth score	Train time	Max GPU memory allocated	Number of parameters
`task1.3_relu7458e`	**66.907**	85.260	18.353	7h 14m	31.63%	26.73 M
`task2_relu_d0fc1`	**68.292**	85.451	17.159	10h 1m	39.25%	32.17 M

3 Task distillation

The third problem introduces a distillation mechanism, used to exchange information between the two tasks. As explained in ([16]), the underlying idea that guides this approach is to leverage the inherent correlation between different tasks, such as depth estimation and semantic segmentation. After the shared encoder, the first two sets of ASPP and decoder blocks provide intermediate predictions and features that are fed into a self-attention block and then summed to the features of the other task. Since the information flow exchanged across tasks might not necessarily be relevant, attention acts as a gate, so that the network learns which information from the other task is the most useful.
Specifically, the feature map of the k-th task (associated with the i-th training sample) F_i^k is used to produce an attention map G_i^k as follows:

$$G_i^k \leftarrow \sigma(W_g^k \otimes F_i^k)$$

where W_g^k are learnable convolution parameters and $\otimes$ denotes convolution.
The original task-specific features, summed with the gated features from the other task, are fed into two new decoders with no skip connections. In formulas, the input to Decoder 3, which is responsible for the final semantics prediction, is:

$$F_i^{o,semseg} \leftarrow F_i^{semseg} + G_i^{semseg} \odot (W_{depth} \otimes F_i^{depth})$$

where $\odot$ denotes element-wise multiplication (a similar relation holds for Decoder 4).
Below is our implementation, which can be found at `mtl/models/model_deeplab_distilled.py`.

```
class ModelDeepLabDistilled(torch.nn.Module):
    def __init__(self, cfg, outputs_desc):
        super().__init__()
        self.outputs_desc = outputs_desc

        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)

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

```

```
        self.decoder_1 = DecoderDeeplabV3p(256, ch_out_encoder_4x, outputs_desc['semseg'])
        self.decoder_2 = DecoderDeeplabV3p(256, ch_out_encoder_4x, outputs_desc['depth'])

        self.attention = SelfAttention(256, 256)

        self.decoder_3 = DecoderDeeplabV3p_noskip(256, outputs_desc['semseg'])
        self.decoder_4 = DecoderDeeplabV3p_noskip(256, outputs_desc['depth'])

    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_tasks_seg = self.aspp_seg(features_lowest)
        features_tasks_dep = self.aspp_dep(features_lowest)

        predictions_4x_seg, features_4x_seg = self.decoder_1(features_tasks_seg, features[4])
        predictions_4x_dep, features_4x_dep = self.decoder_2(features_tasks_dep, features[4])

        predictions_1x_seg = F.interpolate(predictions_4x_seg, size=input_resolution, mode='bilinear', align_corners=False)
        predictions_1x_dep = F.interpolate(predictions_4x_dep, size=input_resolution, mode='bilinear', align_corners=False)

        features_4x_4 = self.attention(features_4x_seg)  # goes into decoder 4
        features_4x_3 = self.attention(features_4x_dep)  # goes into decoder 3

```

```
        predictions_4x_seg_2, _ = self.decoder_3(features_4x_3 + features_4x_seg, features[4])
        predictions_4x_dep_2, _ = self.decoder_4(features_4x_4 + features_4x_dep, features[4])

        predictions_1x_seg_2 = F.interpolate(predictions_4x_seg_2, size=input_resolution, mode='bilinear',
                                             align_corners=False)
        predictions_1x_dep_2 = F.interpolate(predictions_4x_dep_2, size=input_resolution, mode='bilinear',
                                             align_corners=False)

        out = {'semseg': [predictions_1x_seg, predictions_1x_seg_2], 'depth': [predictions_1x_dep, predictions_1x_dep_2]}

        return out
```

It is worth noting that the output is returned as a dictionary of lists, since there is a total of four separate losses to be optimized.
Comparing the results to those of task 2, one can note that the validation performances are almost identical (there is actually a very slight decay, but it is most likely due to the inherent variability of the training process). Given that, one might infer that the two tasks do not exchange relevant information between each other, hence the learned weights for the attention mechanism are quite small.
Nonetheless, since two additional decoders are introduced, the number of parameters grows, and so does the training time. The growth is not tremendous, perhaps due to the fact that the majority of the parameters is related to the encoder and the first two decoders, which were already present in the branched architecture.

Run name	Multitask score	Semseg score	Depth score	Train time	Max GPU memory allocated	Number of parameters
`task2_relu_d0fc1`	**68.292**	85.451	17.159	10h 1m	39.25%	32.17 M
`task3_relu_bf2fa`	**68.250**	85.445	17.195	13h 7m	39.97%	33.36 M

4 Open challenge

To enhance the performance of the network, we applied several of the suggested ideas, analyzing their motivation and performances. Since the performances of the last two versions (task 2 and task 3) of the network are equivalent, we decided to adopt the branched architecture as a baseline model to improve, as it has a smaller train time.

4.1 ResNet50 & Squeeze-And-Excitation

As our first contribution, we included the already implemented Squeeze-And-Excitation mechanism in the architecture.
As explained in ([15]), the underlying idea of this mechanism is to model and exploit channel dependencies of a certain convolutional feature map by introducing channel-specific weights. In fact, it is worth noting that a convolutional layer weights each of the input channels equally when creating the output feature maps. In other words, there is a sort of entanglement between channel dependencies and the local spatial correlation captured by the filters.
The construction of the aforementioned channel weights is carried out with two blocks, Squeeze and Excitation.
The Squeeze operation is used to extract global channel information, which otherwise could not be captured by a simple convolutional layer, whose receptive field is inherently local (except the top-most layers). This embedding is achieved by using global average pooling to generate a single descriptor for each channel. The output $\mathbf{z}$ is a vector of c elements, where c is the number of filters of a given convolutional layer.

$$z_c = \frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} u_c(i,j)$$

where $u_c(i,j)$ is a generic element of the aforementioned feature map.
The Excitation operation processes the channel-wise descriptors to capture complex dependencies between channels and weight them accordingly. For that to be possible, the transformation should be expressive enough to learn non-linearities. This is achieved by feeding the $\mathbf{z}$ vector to a fully connected layer (with ReLU activation), which reduces the number of elements by a factor of r. Then, another FC layer increases the dimensionality back to the original number of channels and a Sigmoid activation is used to squish each element between 0 and 1, so that the resulting vector can be used as a gate that emphasizes or dampens the influence of certain channels. Finally, this weight vector is multiplied channel-wise with the feature channels. In formulas:

$$\mathbf{s} = \sigma(\mathbf{W}_2 ReLU(\mathbf{W}_1 \mathbf{z}))$$
$$\tilde{\mathbf{x}}_\mathbf{c} = s_c \mathbf{u}_c$$

where the last operation indicates that the scalar weight s_c is multiplied by the whole feature map $\mathbf{u}_c$.
Since the other referenced paper ([14]) suggested inserting the Squeeze-And-Excitation module in a *ResNet50* backbone, we decided to switch our encoder

from the initial *ResNet34* to *ResNet50* as well. The main difference between these two architectures lies in the composition of the base residual block that is repeated throughout the encoder. The basic block of *ResNet34* (*Figure 7*)

Layer name	ResNet34	ResNet50
conv1	7×7, 64, stride 2	7×7, 64, stride 2
pool1	3×3, max pool stride 2	3×3, max pool stride 2
conv2_x	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$
conv4_x	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$
conv5_x	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
fc1	9×1, 512, stride 1	9×1, 2048, stride 1
pool_time	$1 \times N$, avg pool stride 1	$1 \times N$, avg pool stride 1

Figure 12: Network structure of ResNet34 and ResNet50

comprises two 3x3 convolutions with a variable number of filters (64, 128, 256, 512). *ResNet50* substitutes this block with a "Bottleneck" structure made up of a 1x1 convolution followed by a 3x3 and then another 1x1 convolution. The 1×1 layers are responsible for reducing and then restoring dimensions, leaving the 3×3 layer a bottleneck with smaller input/output dimensions. The authors of ([9]) show how this allows for a deeper architecture with a similar time complexity.

In this setting, we inserted the Squeeze-And-Excitation module at the end of each Bottleneck block. In particular, the last Bottleneck layer is fed to S&E and then gets summed with the first Bottleneck layer, which goes through a shortcut connection. Below is our implementation for the `BottleneckSE` build-

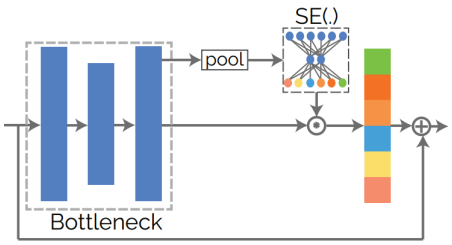


Figure 13: Structure of the Bottleneck building block enriched with Squeeze-And-Excitation

ing block, which we used to build `DecoderSE`, which looks identical to the base decoder (except for the basic block), since the two versions of ResNet repeat their respective residual block the same number of times.

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

    expansion = 4

    def __init__(self, inplanes, planes, stride=1, downsample=None, groups=1,
                 base_width=64, dilation=1, norm_layer=None):
        super(BottleneckSE, self).__init__()
        if norm_layer is None:
            norm_layer = torch.nn.BatchNorm2d
        width = int(planes * (base_width / 64.)) * groups
        if groups != 1 or base_width != 64:
            raise ValueError('BasicBlock only supports groups=1 and base_width=64')
        self.conv1 = resnet.conv1x1(inplanes, width)
        self.bn1 = norm_layer(planes)
        self.relu = torch.nn.ReLU(inplace=True)
        self.conv2 = resnet.conv3x3(width, width, dilation=dilation, stride=stride, groups=groups)
        self.bn2 = norm_layer(planes)
        self.conv3 = resnet.conv1x1(width, planes*self.expansion)
        self.bn3 = norm_layer(planes*self.expansion)
        self.downsample = downsample
        self.stride = stride
        self.se = SqueezeAndExcitation(planes*self.expansion)

    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.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        out = self.se(out)
```

```
76
77          if self.downsample is not None:
78              identity = self.downsample(x)
79
80          out += identity
81          out = self.relu(out)
82
83          return out
```

Unlike ([14]), where the S&E block is only used in the encoder, we also added it after the ASPP modules and at the end of the decoders, right before the last convolutional layer that turns features into predictions.
The segmentation task benefits from this procedure, whereas the depth score stays basically equal. Since the Squeeze-And-Excitation module introduces a limited number of parameters, the train time does not increase remarkably.

Run name	Multitask score	Semseg score	Depth score	Train time	Max GPU memory allocated
`task2_relu_d0fc1`	**68.292**	85.451	17.159	10h 1m	39.25%
`task4_branched_se_3ba31`	**68.916**	86.058	17.141	13h 27m	36.8%

4.2 Data normalization

In the context of autonomous driving, even small errors in distance estimates for object close to camera - and hence the vehicle - may prove fatal. Distances for faraway objects, on the other hand, only need to be estimated with lower accuracy. Modern methods use sophisticated loss functions in order to incorporate such desired properties. The scale invariant logarithmic error (SILog) for example is a commonly used metric as it considers a non-linear weighting of errors in log-space. Utilising extensive loss functions for training, however, might considerably slow down the training process, as the gradients get more complicated. Simple losses, such as MSE, while resulting in faster training, are limited in terms of performance.
In order to overcome the limitations of both approaches, it is possible to relocate computationally expensive operations to the data pre-processing stage. One way to do that is to perform different normalization of the dataset, as suggested in the handout. The training is therefore conducted on normalized data. Then, the final predictions obtained by the network have to be scaled back to the original unit of measurement.
Our approach to study the contribution of normalization has been to perform different transformations on the same model. We have studied the results of different approaches on the improved model (*ResNet50 + Squeeze-And-Excitation*). This is a reasonable choice as the encoder model and the data

pre-processing yield independent contributions to the final model.
In addition to the default standardization method, which consists of bringing the depth values to a standard normal distribution with zero mean and unit variance, we have adopted two alternative approaches:

1. **MIN-MAX NORMALIZATION**:
   In the Min-Max normalization the depth values are normalized in the [0,1] interval. This is achieved by applying the following transformation. Let d be the depth value of a certain pixel and $d_{\min}$ and $d_{\max}$ be the minimum and maximum depth value in the dataset (those values were provided in the code):

   $$d_{\text{norm}} = \frac{d - d_{\min}}{d_{\max} - d_{\min}}$$

2. **LOG NORMALIZATION**:
   This normalization technique is used in ([12]) for the *KITTI* dataset. Again, the normalized values are brought to the [0,1] interval. However, a different transformation is applied:

   $$d_{\text{norm}} = \frac{\log(d - d_{\min} + 1)}{\log(d_{max} - d_{\min} + 1)}$$

   where log denotes the natural logarithm.

Let us now compare the results of the three normalization:

Analysis of different normalization techniques			
Normalization technique	Multitask score	Semseg score	Depth score
Standardisation $\mathcal{N}(0, 1)$	**68.916**	86.058	17.141
Min-Max Normalization	**65.604**	86.074	20.469
Log Normalization	**67.823**	85.954	18.131

Obviously, the differences in the results only involve the depth score, as the three implementations do not feature any difference with regard to the semantic segmentation task. Among the three normalization techniques, the classic standardization seem to have an edge with the log normalization ending up as a close second. On the other hand, the affine transformation of the Min-Max Normalization seems to be too gross to interpret the true distribution of the depth values.
Although the log-normalization technique yields excellent results, it does not quite overcome the performance of the the classical standardization. In ([12]), the proposed transformation is used to normalize the data in log-space. In their proposed architecture *MultiDepth*, a shared *ResNet101* encoder is used to create

the features which are then shared among task specific decoders. *MultiDepth* combines two approaches of depth estimation: by a main regression task and by an auxiliary classification task. Our depth estimation task, instead, was only conducted by a single regression task. Moreover, in ([12]), the experiments are conducted using the KITTI dataset, whereas we use the *Miniscapes* dataset.

4.3 Loss functions

The standard loss function for optimizing the depth regression is the *L2* loss, minimizing the squared euclidean distance between the predictions and the ground truth. Although this produces good results in our test case, we found that the so called **berHu loss** (reverse Huber loss) yields a better final error than *L2*. This result is confirmed by ([13]).

$$\mathcal{B}(x) = \begin{cases} |x| & |x| \leq c, \\ \frac{x^2+c^2}{2c} & |x| > c. \end{cases}$$

In every gradient descent step, when we compute $\mathbf{B}(|\hat{y}_i - y_i|)$ we set $c = \frac{\max_i(|\hat{y}_i - y_i|)}{5}$, where i indexes all pixels over each image in the current batch, that is 20% of the maximal per-batch error. Empirically, BerHu shows a good balance between the two norms in the given problem; it puts high weight towards samples/pixels with a high residual because of the *L2* term and, at the same time, *L1* accounts for a greater impact of smaller residuals' gradients than *L2* would.

Analysis of different training losses			
Training loss	Multitask score	Semseg score	Depth score
L2	**68.916**	86.058	17.141
L1	**70.558**	86.21	15.651
BerHu loss	**70.645**	86.222	15.576

We can observe that both the *L1* loss and the *BerHu* loss yield substantial improvements to the the depth estimation metric. This is probably due to the fact that most of the errors are small, and are therefore better corrected with the *L1* loss.

4.4 Test-time augmentation

A popular method to improve the test performance is to perform testing on transformed version of a certain batch and then aggregate the results.
As explained in ([18]), unaltered off-the-shelf features are often not sufficiently robust to adapt to variations in the target data including changes in scale, illumination, orientation, color, contrast, deformations, and background clutter.

This is due to these networks (including *ResNet*) being predominantly trained with natural images and light data augmentation. Therefore, off-the-shelf features are not scale, rotation or contrast invariant. However, transform invariant features are desirable for real-world visual tasks.
In particular, we addressed the issue of scale invariance, by performing testing on multi-scale versions of each test batch.

```
207     def test_step(self, batch, batch_nb):
208         _, y_hat_semseg_lbl, _, y_hat_depth_meters =
            ↪  self.inference_step(batch)  # WITHOUT AUGMENTATION
209         y_hat_semseg_lbl_tens =
            ↪  y_hat_semseg_lbl.unsqueeze(1).unsqueeze(4)
210         y_hat_semseg_lbl_tens = copy.copy(y_hat_semseg_lbl)
211         y_hat_semseg_lbl_tens = y_hat_semseg_lbl_tens.cuda()
212         y_hat_semseg_lbl_tens =
            ↪  y_hat_semseg_lbl_tens.unsqueeze(1).unsqueeze(4)
213
214         scales = [0.5, 2, 3]
215
216         batch_int = {'id': batch['id']}
217
218         for scale in scales:
219             batch_int[MOD_RGB] = F.interpolate(batch[MOD_RGB],
                ↪  scale_factor=scale, mode='bilinear')
220             _, _, _, y_hat_depth_meters_int =
                ↪  self.inference_step(batch_int)
221             y_hat_semseg_lbl_int =
                ↪  y_hat_semseg_lbl_int.unsqueeze(1)
222             y_hat_semseg_lbl_int =
                ↪  y_hat_semseg_lbl_int.type(torch.cuda.FloatTensor)
223             y_hat_semseg_lbl_int =
                ↪  F.interpolate(y_hat_semseg_lbl_int,
                ↪  scale_factor=1/scale, mode='bilinear')
224             y_hat_depth_meters_int =
                ↪  F.interpolate(y_hat_depth_meters_int,
                ↪  scale_factor=1/scale, mode='bilinear')
225             y_hat_semseg_lbl_int =
                ↪  y_hat_semseg_lbl_int.type(torch.cuda.LongTensor)
226             y_hat_semseg_lbl_int =
                ↪  y_hat_semseg_lbl_int.unsqueeze(4)
227             y_hat_depth_meters += y_hat_depth_meters_int
228             y_hat_semseg_lbl_tens =
                ↪  torch.cat((y_hat_semseg_lbl_tens,
                ↪  y_hat_semseg_lbl_int),dim=4)
229             y_hat_semseg_lbl_int = y_hat_semseg_lbl_int.cpu()
```

```
230
231         y_hat_depth_meters = y_hat_depth_meters / len(scales)
232
233         y_hat_semseg_lbl_tens = y_hat_semseg_lbl_tens.squeeze(1)
234         y_hat_semseg_lbl = torch.mode(y_hat_semseg_lbl_tens,
            ↪  dim=-1)
235         y_hat_semseg_lbl = y_hat_semseg_lbl[0]
```

For each of the selected scales, the batch RGB values are interpolated to the desired resolution. The resulting depth and semseg predictions are then converted back to the original resolution (by interpolating with an inverse scale factor). The aggregated depth prediction is obtained by computing a simple mean of the scaled maps, whereas the segmentation prediction comes from a process of majority voting (in particular, the prediction tensors are stacked along a new dimension and the mode is computed). However, since the function `torch.mode` does not support CUDA, making the handling of tensors very slow, we did not aggregate the segmentation predictions in the end. Below are the test scores for the current best submission (branched architecture with SE, *ResNet50*, and berHu loss) and the same model with TTA. With the given scales, the test performance deteriorates, maybe because the scaled predictions derail the original one. We did not have much time to tune this hyperparameter, but it would have been interesting to find the optimal augmentation scales and observe the related test metrics.

Submission name	Multitask score	Semseg score	Depth score
`task4_branched_se_dec_berhu_9c208`	**70.60**	86.36	15.76
`task4_branched_se_dec_berhu_testaug_a7339`	**67.67**	86.36	18.69

In typical computer vision problems, many more transformations might be implemented (e.g., rotation, illumination change, mirroring), however, not all of them might be relevant for autonomous driving scenarios. For instance, considering that the scenes viewed from on-board cameras are almost always flat roads, a significant rotation transformation may not make sense and not contribute to the generalization performances.

5 Literature

([1]) - L. N. Smith. “Cyclical Learning Rates for Training Neural Networks”. In: WACV. 2017, pp. 464–472.

([2]) - Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: ICLR. 2015, pp. 1–15.

([3]) - Zhang, M. R., Lucas, J., Hinton, G., Ba, J. (2019). Lookahead optimizer: k steps forward, 1 step back. arXiv preprint arXiv:1907.08610.

([4]) - Choi, D., Shallue, C. J., Nado, Z., Lee, J., Maddison, C. J., Dahl, G. E. (2019). On empirical comparisons of optimizers for deep learning. arXiv preprint arXiv:1910.05446.

([5]) - Hardt, M., Recht, B., Singer, Y. (2016, June). Train faster, generalize better: Stability of stochastic gradient descent. In International Conference on Machine Learning (pp. 1225–1234). PMLR.

([6]) - Wilson, A. C., Roelofs, R., Stern, M., Srebro, N., Recht, B. (2017). The marginal value of adaptive gradient methods in machine learning. arXiv preprint arXiv:1705.08292.

([7]) - Dmytro Mishkin, Nikolay Sergievskiy, Jiri Matas (2016).Systematic evaluation of CNN advances on the ImageNet.arXiv:1606.02228.

([8]) - Raz Rotenberg. (2020), How to Break GPU Memory Boundaries Even with Large Batch Sizes.Published in Towards Data Science

([9]) - Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun (2015), Deep Residual Learning for Image Recognition.arXiv:1512.03385

([10]) - Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun (2015),Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, ICCV, 2015. arXiv:1502.01852

([11]) - Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, (2015), Deep Residual Learning for Image Recognition. arXiv:1512.03385

([12]) - Lukas Liebel;Marco Körner; MultiDepth: Single-Image Depth Estimation via Multi-Task Regression and Classification,2019 IEEE Intelligent Transportation Systems Conference (ITSC)

([13]) - Iro Laina, Christian Rupprecht, and Vasileios Belagiannis, “Deeper depth prediction with fully convolutional residual net- works,” in Proc. 3DV, 2016.

([14]) Maninis, K.K., Radosavovic, I., Kokkinos, I.: Attentive single-tasking of multiple tasks. In:IEEE Conf. Comput. Vis. Pattern Recog. pp. 1851–1860 (2019)

([15]) Hu, J., Shen, L., Sun, G.: Squeeze-and-excitation networks. In: IEEE Conf. Comput. Vis. Pattern Recog. pp. 7132–7141 (2018)

([16]) Xu, D., Ouyang, W., Wang, X., Sebe, N.: Pad-net: Multi-tasks guided prediction-and- distillation network for simultaneous depth estimation and scene parsing. In: IEEE Conf. Comput. Vis. Pattern Recog. pp. 675–684 (2018)

([17]) L. Chen, G. Papandreou, F. Schroff, and H. Adam. Rethinking atrous convolution for semantic image segmentation. CoRR, abs/1706.05587, 2017. URL http://arxiv.org/abs/1706.05587.

([18]) Osman Tursun, Simon Denman, Sridha Sridharan, Clinton Fookes;Learning Test-time Augmentation for Content-based Image Retrieval. arXiv:2002.01642 (2021)