Project 2: Multi-task Learning for Semantics and Depth

Deep Learning for Autonomous Driving
Team ID 18
Ivan Alberico, Nicola Loi
March 26, 2021 - May 20, 2021

In this project a Multi-Task Learning (MTL) architecture is constructed and improved step by step, with the goal of enhancing the semantic segmentation and monocular depth estimation of the model. To evaluate the performance, the metrics used are the *IoU* (Intersection over Union) for the semantic segmentation (`semseg`), for which higher values indicate a better performance, and the *SI-logRMSE* for the depth estimation (`depth`), for which, instead, lower values are preferred. These two metrics are then combined in the `total` metric, for which higher values are better. These metrics are evaluated on a validation set, and all the choices made during the project to obtain better models are based on their scores. In addition to this, the results of the metrics computed on the test set, which will henceforth be called `grader`, will also be shown, but only for the best model of each subsection.

In order to better visualize how the network's performance improved with the different modifications, the prediction images generated by the best-performing model of each subsection are exhibited in the *Appendix* in Table 13.

To distinguish the multiple architectures that have been tested, they have been labelled with a unique tag. When the metrics achieved by a particular model is displayed in a table or whenever the model is simply mentioned in the text, it is cited using the abbreviation of its tag. Table 14 in the *Appendix* is provided to link each abbreviation to the corresponding complete model name.

Problem 1 - Joint architecture

1.1 Hyper-parameter tuning

For the first part of the problem, the goal is to tune some hyper-parameters of the network to achieve the best possible validation result. Many approaches exist to perform an automatized hyper-parameter tuning, and the `Keras Tuner` library offers a good starting point [1]. However, in this project it is required to manually tune the hyper-parameters, searching and choosing the best ones comparing the results of multiple training runs. This heuristic approach is not an optimal method to find the best hyper-parameters, however it could be a convenient way to find satisfactory hyper-parameters that, even if sub-optimal, can enhance the performance of the network.

a) **The optimizer and LR choice**

The first hyper-parameter to be optimized is the learning rate η, together with the choice of using the *Stochastic Gradient Descent* (*SGD*) or the *Adam* optimizer. The learning rate is one of the most important hyper-parameters: a small value executes at each step of the training a small update in the weights of the network, slowing down the convergence and probably getting stuck in a local minimum; a big value performs larger updates of the weights at each training step, and could help in getting out of a local minimum, but a large step could also outdistance an optimal minimum, and could continuously bounce back and forth. To overcome this issue, a decay of the learning rate is exploited.

The default network employs a *SGD* optimizer with $\eta = 0.01$ (1e-2). A sample training run with this configuration (original_37628) provides a sample result for the evaluation metrics: `total: 39.83`, `semseg: 67.194`, `depth: 27.357`. At the beginning, some preliminary test runs are carried out for each optimizer, changing logarithmically η to assess a generic hyper-parameter value that gives satisfactory results. From it, a more specific search is performed with the best optimizer, tuning η within a small range to refine the best learning rate value of the best optimizer.

Starting from the *SGD* optimizer of the original network, η values of 0.0001, 0.001, and 0.1 (1e-4, 1e-3, and 1e-1) are tested, with the results shown in Table 1. The run P1.1a_06c0d with $\eta = 0.1$ has the best results in all the metrics, beating also the default setting, which by the way is the second best result for the *SGD* optimizer.

SGD optimizer				
Run name	learning rate	Metrics		
		total ↑	semseg ↑	depth ↓
P1.1a_1f725	0.0001	-15.225	25.654	40.879
P1.1a_61964	0.001	22.144	55.079	32.935
original_37628	0.01	39.837	67.194	27.357
P1.1a_06c0d	0.1	**42.740**	**69.885**	**27.145**

Table 1: Results of the tested runs with the *SGD* optimizer for the tuning of the learning rate.

It is then evaluated the performance of the *Adam* optimizer, which should have a default learning rate of a couple of magnitude smaller than *SGD*. Therefore, in these preliminary tests η values of 0.00001, 0.0001, 0.001, and 0.01 (1e-5, 1e-4, 1e-3, and 1e-2) are tried out. The results are displayed in Table 2. The run P1.1a_d68b6 with $\eta = 0.0001$ (1e-4) has the top results in all metrics, and since it beats also the best results achieved with *SGD*, it is used as a reference to refine the learning rate hyper-parameter.

Adam optimizer				
Run name	learning rate	Metrics		
		total ↑	semseg ↑	depth ↓
P1.1a_47e22	0.00001	24.313	56.433	32.121
P1.1a_d68b6	0.0001	**43.482**	**69.925**	**26.443**
P1.1a_27e27	0.001	38.429	66.269	27.840
P1.1a_4d200	0.01	5.362	41.835	36.472

Table 2: Results of the tested runs with the *Adam* optimizer for the tuning of the learning rate.

The *Adam* optimizer is investigated more in depth with η values that deviate from the 0.0001 (1e-4) reference, but remaining inside the range 0.001-0.00001 (1e-3 - 1e-5). The complete results of these tests can be seen in Table 3. Since the evaluation metrics are getting worse even with small deviations ($\eta = 0.00007$ (7e-5) and 0.0002 (2e-4)) from the reference learning rate, it is concluded that as far as allowed by this heuristic search, the best results are achieved with the model P1.1a_d68b6, which has the *Adam* optimizer with a learning rate of 0.0001 (1e-4), and whose metrics results are `total: 43.482`, `semseg: 69.925`, `depth: 26.443`. The `grader` scores are `total: 43.298`, `semseg: 69.889`, `depth: 26.591`.

refining Adam optimizer				
Run name	learning rate	Metrics		
		total ↑	semseg ↑	depth ↓
P1.1a_d10d0	0.00002	33.525	63.039	29.513
P1.1a_237dc	0.00007	42.526	69.292	26.766
P1.1a_d68b6	0.0001	**43.482**	**69.925**	**26.443**
P1.1a_81237	0.0002	43.019	69.625	26.605
P1.1a_10da7	0.0004	42.024	68.886	26.862

Table 3: Results of the tested runs with the *Adam* optimizer for refining the tuning of the learning rate.

As introduced at the beginning of this subsection, the advantages and disadvantages of small/big learning rate values must be tuned; it does not exist a universal best value for every situation. The chosen learning rate and optimizer represent the optimal choices for this network and its tasks, taking into consideration, however, the limitation of the heuristic research carried out.

b) **The batch size**

The next hyper-parameter to be tuned is the batch size used during the training. A big batch size is slow to compute, but gives a better approximation of the true gradient, and so the correct direction to follow to update the weights. A small batch size is faster to compute, but the corresponding computed gradient is more noisy, and so the direction followed by the network for the weights update is less optimal, causing a slower convergence.

By default, the batch size is set to 4, with 16 epochs. Since increasing or decreasing the batch size affects the number of steps executed in a single epoch, it is chosen to change the number of epochs as well, proportionally to the batch size, to retain the same number of steps during the training. The network chosen in the previous task (`P1.1a_d68b6`) is used as reference, and since for theoretical and technical reasons the batch size should be a power of 2, it is only evaluated a reduction of the batch size to 2 and an enlargement to 8 and 16, with 8, 32, and 64 epochs respectively. Just for a better interpolation of these values, it is tried out also a batch size of 6 and 12 with 24 and 48 epochs respectively. The results are shown in Table 4.

It can be seen that the improvement or the deterioration of the metrics is strongly correlated to the batch size and the number of epochs employed. There is an overall enhancement of the performance augmenting both of them, but at the cost of increasing the total training time. Therefore, a trade-off must be made in the choice of this hyper-parameter; it must be considered not only the evaluation metrics, but also the acceptable maximum training time. Among the runs that increase the metrics, it is then chosen the run `P1.1b_0b891` with a batch size of 8, 32 epochs and a training time of 7h 20m, to stay in a reasonable time without exceeding the original training time of 4h 25m too much. Its metrics scores are `total: 46.276`, `semseg: 71.678`, `depth: 25.402`, while the `grader` performances are `total: 46.022`, `semseg: 71.655`, `depth: 25.633`.

batch					
Run name	Batch size | n. epochs	Metrics			training time
		total ↑	semseg ↑	depth ↓	
P1.1b_79cc1	2 | 8	38.122	66.419	28.297	3h20m
P1.1a_d68b6	4 | 16	43.482	69.925	26.443	4h25m
P1.1b_3bd9c	6 | 24	45.044	70.955	25.912	5h50m
P1.1b_0b891	8 | 32	46.276	71.678	25.402	7h20m
P1.1b_14fc8	12 | 48	47.456	72.561	25.105	10h35m
P1.1b_be212	16 | 64	**47.990**	**72.981**	**24.991**	13h20m

Table 4: Results of the tested runs for the tuning of the batch size.

It is important to emphasize the essential aspect of the number of epochs related to the batch size. From Figure 1 it can be seen that when a particular run reaches its last epoch, it outperforms the other ones, if they are contemplated only up to that epoch. However, it will be outperformed if it the maximum number of epochs of the other runs are considered, i.e. when the other run are completed. In other words, a run with a bigger batch size has better metrics than a run with a smaller one only if they have a number of epochs proportional to their batch size. Considering the same number of epochs, a smaller batch size outperforms a bigger one. The latter needs more epochs to reach the same level and eventually surpass the run with a smaller batch size, since each of its epochs performs less steps. With the same total steps completed during the training (thanks to a number of epochs proportional to the batch size), the run with the bigger batch size is capable to outperform a smaller batch size, as anticipated in the introduction of this subsection.

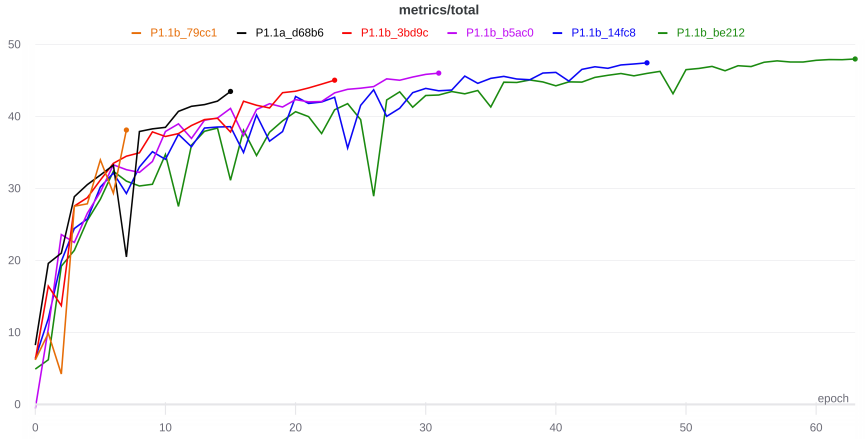


Figure 1: `total` metric of the tested models for the task weighting.

c) **Task weighting**

Finally, the last hyper-parameters to tune are the loss weights of the semantics and depth tasks. A proper balance must be found between the two losses, to improve the joint performance. By default, the tasks are equally weighted: the loss weights of the semantics and depth tasks are both 0.5. The network P1.1b_0b891 chosen in the previous subsection is used as reference. The heuristic search of the best hyper-parameters is initiated sampling uniformly the hyper-parameters range, to have a general view of what range of values enhances the performances. It is investigated the semantics/depth loss weight ratio of 0.2/0.8, 0.4/0.6, 0.6/0.4, and 0.8/0.2, with the results displayed in Table 5. For obvious reasons the semantics/depth weights range limits, 1/0 and 0/1, are not tested since this would completely disregard one of the tasks.

task weighting				
Run name	Loss weights semseg | depth	Metrics		
		total ↑	semseg ↑	depth ↓
P1.1c_756bb	0.2 | 0.8	44.538	69.653	**25.114**
P1.1c_53575	0.4 | 0.6	46.008	71.236	25.228
P1.1b_0b891	0.5 | 0.5	**46.276**	71.678	25.402
P1.1c_4a169	0.6 | 0.4	46.229	71.992	25.763
P1.1c_6a37c	0.8 | 0.2	45.647	**72.218**	26.571

Table 5: Results of the tested runs for the tuning of the tasks weights.

As one might expect, the results of the individual metrics `semseg` and `depth` are strongly related to their weights: increase the weight of a task brings a better score in that task, and vice-versa. However, the collective improvement and deterioration of the metrics of the two tasks do not cancel each other out. As it can be seen in Table 5, the `total` score begins to decrease when you move away from the semantics weight value of 0.5-0.6 (and consequently from the depth weight value of 0.5-0.4). Since the semantics loss weight values of 0.5 and 0.6 (runs P1.1b_0b891 and P1.1c_4a169) provide the best and very similar `total` performance (`46.276` and `46.229`), a more specific value investigation is carried out to determine the best weights ratio.

The weights 0.5/0.5 and 0.6/0.4 are tested again, together with their middle values 0.55/0.45. Since also the previous tests with the same hyper-parameters must be taken into account for the evaluation of the best model, the 0.55/0.45 values are tested twice, so for each weights ratio there will be two runs to examine. A finer sampling of the range is not investigate since the results of the two networks are very similar, so it won't make sense to do a bigger analysis as the one performed in Problem 1.1.a. In the current case, the differences between the final `total` score are smaller than the metrics variance, hence it will be difficult to assess a precise best value inside the range. Due to the random initialization of the weights but also to the

random order of the training samples, each model will not always have the same final score values, the metrics will have a certain variance. This is the reason of why two runs for each hyper-parameter value are now considered, to try to have a raw estimation of this variance. A more recommended strategy is to test more than two runs (theoretically, infinite runs) to have more precise statistics of the mean and variance of each network.

refining task weighting		
Run name	Loss weights semseg | depth	metrics/total ↑
P1.1b_0b891	0.5 | 0.5	46.276
P1.1b_b5ac0	0.5 | 0.5	46.027
P1.1c_73e8c	0.55 | 0.45	46.345
P1.1c_a3cae	0.55 | 0.45	**46.467**
P1.1c_da60c	0.6 | 0.4	46.262
P1.1c_4a169	0.6 | 0.4	46.229

Table 6: Results of the tested runs for refining the tuning of the tasks weights.

The results of the `total` metric of the tested runs are displayed in Table 6. Since the first and second best results (`46.467` and `46.345`) are achieved with the 0.55/0.45 weights, these values are chosen as the final hyper-parameters. The two tasks should have approximately the same importance in the weights update, or the joint performance will be negatively affected. The best run `P1.1c_a3cae` with `total: 46.467` has individual metrics performance of `semseg: 71.961`, `depth: 25.495`, while the `grader` scores are `total: 46.383`, `semseg: 71.981`, `depth: 25.598`.

1.2 Hardcoded hyperparameters

a) **Initialization with ImageNet weights**

All of the different networks tested until now during the hyper-parameters tuning were trained from scratch, with a random initialization of the weights, without exploiting any external knowledge realated to the tasks. Generally, a common solution to enhance the training for a certain task is to benefit from the weights learned by another network on a similar task, a procedure called *transfer learning*. Instead of making a fresh start, the network is initialized with these pretrained weights to improve and speed-up the performances.

To test if the model designed up to now could benefit from transfer learning, the *ResNet34* encoder of the network is initialized with the pretrained weights from a model trained with the *ImageNet* database. As shown in Table 7, with this simple and fast modification, the performances are indeed improved from the previous model `P1.1c_a3cae`, which did not use the pretrained weights in the encoder. The tested run `P1.2a_44ba0` with the *ImageNet* weights gives the following metrics values: `total: 49.417`, `semseg: 74.108`, `depth: 24.691`, while the `grader` scores are `total: 49.232`, `semseg: 74.113`, `depth: 24.881`. Given the results, from now on the models that will be tested for the next problems will employ a pretrained encoder.

pretrained encoder				
Run name	pretrained	Metrics		
		total ↑	semseg ↑	depth ↓
P1.1c_a3cae	no	46.467	71.961	25.495
P1.2a_44ba0	yes	**49.417**	**74.108**	**24.691**

Table 7: Comparison between the current best network and the same network but with pretrained weights.

b) **Dilated convolutions**

Staying within the encoder module, another possible improvement could be the implementation of dilated convolutions to expand the receptive field of the features of the layers of the encoder. With a standard 3x3 convolution, each feature of a layer has a receptive field of 3x3 on the previous layer, while with a dilated convolution of dilation 2, the feature will now have a 5x5 receptive field on the previous layer. This extension makes the features able to better infer the context of their neighborhood, since they can draw more information from a larger area around themselves. Theoretically, the encoder should accomplish a more valuable extrapolation of information from the input image, generating more advantageous features to be passed over in the network.

The dilated convolutions are tested with the run P1.2b_172ab, in which a dilation of 2 is activated in the fourth layer of the encoder, which is its last layer. The results obtained can be visualized in Table 8, where the previous model P1.2a_44ba0 with standard convolutions (i.e. with dilation 1) is used as a reference.

dilation encoder				
Run name	dilation last layer	Metrics		
		total ↑	semseg ↑	depth ↓
P1.2a_44ba0	1	49.417	74.108	24.691
P1.2b_172ab	2	**60.497**	**81.177**	**20.68**

Table 8: Comparison between the current best network and the same network but with dilated convolutions.

The dilation provides a great leap in performance: its activation significantly improves the evaluation metrics, with the new enhanced results of `total: 60.497`, `semseg: 81.177`, `depth: 20.68`. The `grader` scores are `total: 60.547`, `semseg: 81.287`, `depth: 20.740`. Thanks to the extension of the receptive field, the network is relying on better features provided by the encoder. For a visual example of the improvement, in Table 13 of the *Appendix* it can be observed that the model P1.2b_172ab produces more details and more defined contours of the semantic segmentation of the objects, which become more distinguishable, while the previous models provide a less detailed segmentation, especially in edges and small objects. Also the depth estimation is sharper, some contours of the objects become recognizable, but the overall output image prediction is still a bit blurred. Due to the improvements, for the subsequent problems the dilation on the last layer of the encoder will be kept activated.

1.3 ASPP and skip connections

The aim of this task is to implement the *ASPP* module and the skip connections to the decoder, in a way similar to the *DeepLabv3+* model proposed in [5]. The *ASPP* module, which stands for *Atrous Spatial Pyramid Pooling*, consists in computing different atrous convolutions with different rates in parallel, in order to encode the multi-scale context information. In addition to these atrous convolutions, an *ImagePooling* module is implemented, in which global average pooling is applied to the output feature map of the encoder, followed by a 1x1 convolution with 256 filters and a batch normalization layer, as proposed in [4]. The output of the batch normalization layer is then upsampled through bilinear interpolation in order to recover the initial height and width of feature map. The *ImagePooling* module is introduced to integrate global context information into the model. Moreover, an additional 1x1 convolution is performed in parallel to the previous operations.

The output of the *ASPP* module is then obtained by concatenating the feature maps of the different atrous convolutions and the one of the *ImagePooling* module, and by applying an additional 1x1 convolution that reduces the number of output channels to 256. The code of the implemented *ASPP* module is reported in Figure 2. Each atrous convolution corresponds to an *ASPPpart*

module, which contains a 3x3 convolution layer, followed by a batch normalization and a *ReLU*. In each of the 3x3 convolutions, the *kernel_size* is set to 3, the *stride* is set to 1, while the *dilation* takes different values according to which atrous convolution has to be implemented, and the values are expressed in the *rates* list. The default values of the *rates* list are (3, 6, 9).

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

        # TODO: Implement ASPP properly instead of the following
        self.aspp_modules = torch.nn.Module()
        self.aspp_modules.add_module("1x1conv", ASPPpart(in_channels, out_channels, kernel_size=1, stride=1, padding=0,
                                                         dilation=1))

        for val in rates:
            self.aspp_modules.add_module("3x3 conv with rate={}".format(val), ASPPpart(in_channels, out_channels,
                                                                                       kernel_size=3, stride=1,
                                                                                       padding=val, dilation=val))

        self.aspp_modules.add_module("Image Pooling", ImagePooling(in_channels, out_channels))

        self.conv_out = ASPPpart(out_channels * (len(rates) + 2), out_channels, kernel_size=1, stride=1, padding=0,
                                 dilation=1)

    def forward(self, x):
        # TODO: Implement ASPP properly instead of the following

        aspp_output_list = [module(x) for module in self.aspp_modules.children()]
        aspp_output = torch.cat(aspp_output_list, dim=1)
        out = self.conv_out(aspp_output)

        return out
```

Figure 2: Code snippet of the *ASPP* module.

```
class ImagePooling(torch.nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.global_avg_pooling = torch.nn.AdaptiveAvgPool2d(1)
        self.conv1x1 = torch.nn.Conv2d(in_channels, out_channels, kernel_size=(1, 1), bias=False)
        self.batch_norm = torch.nn.BatchNorm2d(out_channels)

    def forward(self, x):
        _, _, height, width = x.shape
        out = self.global_avg_pooling(x)
        out = self.conv1x1(out)
        out = self.batch_norm(out)
        out = F.interpolate(out, size=(height, width), mode="bilinear", align_corners=False)
        return out
```

Figure 3: Code snippet of the *ImagePooling* module.

As far as the *padding* parameter is concerned, in order to preserve the spatial dimensions (width and height) in the different convolutions, its value is set equal to the *dilation* value of each specific atrous convolution. The reason is that, whenever a 2D convolution is applied, the height and width of the output feature map depends on the dimensions of the input through the following relations:

$$H_{out} = \left[\frac{H_{in} + 2 \times padding_H - dilation_H \times (kernel\_size_H - 1) - 1}{stride_H} + 1 \right] \tag{1}$$

$$W_{out} = \left[\frac{W_{in} + 2 \times padding_W - dilation_W \times (kernel\_size_W - 1) - 1}{stride_W} + 1 \right] \tag{2}$$

The condition for which $H_{out} = H_{in}$ and $W_{out} = W_{in}$ apply, in the specific case in which $stride = 1$, is the following:

$$2 \times padding - dilation \times (kernel\_size - 1) = 0 \tag{3}$$

which is satisfied for $padding = dilation$, since the *kernel_size* is set to 3 in this case.

From a more technical point of view regarding the implementation, a new *torch.nn.Module()* is initialized in the *ASPP* class, in which the parallel operations are represented by sub-modules

added through the *add_module* function. In the *forward()* method the output is given by a list containing the outputs of the different children modules, whose values are then concatenated along the dimension of the features channels and given to the last 1x1 convolutional layer. This last convolution, followed by a batch normalization and a *ReLU*, takes an input of $out\_channels \times (len(rates) + 2)$ channels (the 2 refers to the 1x1 convolution and the *ImagePooling* module which run in parallel with the atrous convolutions) and outputs a tensor with 256 channels.

Although the output feature map of the *ASPP* module encodes rich semantic information, due to the numerous pooling and convolution operations in the process, most of the detailed information like the ones related to the object boundaries are not preserved. The *DeepLabv3+* model introduces skip connections to the decoder, which integrate the features coming from the encoder network, in order to recover the spatial information and obtain sharper segmentation results along the object boundaries. The implementation of the decoder is shown in Figure 4.

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

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

        self.conv1x1 = torch.nn.Conv2d(skip_4x_ch, 48, kernel_size=(1, 1))

        self.conv3x3 = torch.nn.Sequential(
            torch.nn.Conv2d(bottleneck_ch + 48, bottleneck_ch, kernel_size=(3, 3), padding=(1, 1), stride=(1, 1)),
            torch.nn.Conv2d(bottleneck_ch, bottleneck_ch, kernel_size=(3, 3), padding=(1, 1), stride=(1, 1)))

        self.features_to_predictions = torch.nn.Conv2d(bottleneck_ch, num_out_ch, kernel_size=(1, 1), stride=(1, 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.
        features_aspp_4x = F.interpolate(
            features_bottleneck, size=features_skip_4x.shape[2:], mode='bilinear', align_corners=False)

        features_skip_4x = self.conv1x1(features_skip_4x)
        features_4x_raw = torch.cat((features_aspp_4x, features_skip_4x), dim=1)
        features_4x = self.conv3x3(features_4x_raw)
        predictions_4x = self.features_to_predictions(features_4x)

        return predictions_4x, features_4x
```

Figure 4: Code snippet of the implementation of the skip connections in the Decoder network

The decoder takes as input both the low level features from the encoder, in particular those corresponding to scale 4, and the output feature map of the *ASPP* module. First, it applies a 1x1 convolution to the features from the encoder, with the aim of reducing the number of channels. In [5] it is investigated that reducing the channels to 48 brought the best results. The resulting feature map is then concatenated with the features coming from the *ASPP* module once they have been upsampled to the same dimensions. After that, the concatenated feature maps undergo two consecutive 3x3 convolutions with 256 filters, still proposed by [5], and a final 1x1 convolution that computes the output predictions of the network.

Table 9 shows the comparison between the evaluation metrics of the previous model P1.2b_172ab and the ones obtained by implementing the *ASPP* module with the skip connections (run P1.3_ddebf). The implemented modules led to a significant improvement in the network performance, achieving the following results: `total: 64.538`, `semseg: 84.798`, and `depth: 20.26`. The scores of the `grader` are `total: 64.537`, `semseg: 84.931`, and `depth: 20.393`. It can be observed that the main contribution of this improvement is given by the segmentation part, which increased from `81.177` to `84.798`, while the depth estimation only shows a slight decrease of the *SI-logRMSE*. A potential reason could be that the refinement of the object boundaries, which is the result of adding skip connections to the decoder, is more beneficial for the segmentation, since it requires sharp edges to correctly separate the different elements in the scene, rather than the depth estimation.

ASPP module with skip connections				
Run name	Atrous rates	Metrics		
		total ↑	semseg ↑	depth ↓
P1.2b_172ab	-	60.497	81.177	20.680
P1.3_ddebf	(3,6,9)	**64.538**	**84.798**	**20.260**

Table 9: Comparison between the current best network and the one with *ASPP* and skip connections implemented.

The benefits of this model can be observed also in Table 13 of the *Appendix*, by comparing the prediction results obtained with the *ASPP* module and the skip connections, with those computed previously. While in the previous models the object boundaries in both the segmentation and depth predictions were still quite blurred, now, in particular thanks to the addition of the skip connections to the decoder, the edges are much more well-defined and sharp. Moreover, a lot of small details are now much more visible and distinguishable, like the sidewalk poles, the road signs and the foliage of the trees.

Problem 2 - Branched architecture

In the previous problem, segmentation and depth estimation tasks were trained on the same joint architecture, sharing all the parameters in both the encoder and the decoder networks, except for the last convolutional layer that generated the corresponding predictions. Now a branched architecture will be implemented, based on what is proposed in [7] and [8], in which segmentation and depth estimation share the same parameters of the encoder network, but they have separated branches with task-specific *ASPP* modules and decoders. Branched architectures are usually recommended for multi-task learning since they allow, with respect to the previous joint architecture, more specific training along the different branches for each task. At the same time, they are also preferred to multiple separated networks for single-task training, since the memory allocation and the inference time are reduced due to the shared layers, and they also seem to perform better and improve generalization. However, with respect to the joint architecture encountered earlier, the computational overhead is undoubtedly higher, but it is the price to pay for an increase in the performance, as it will be discussed later.

The implementation of the branched architecture, whose code is shown in Figure 5, is based on the *ASPP* and decoder modules already used in the joint architecture. Firstly, the channel outputs for the segmentation and the depth estimation are computed from the *outputs_desc* dictionary. Then, a *torch.nn.Module()* is initialized, in which the two added modules represent the segmentation and the depth branches of the model. An *ASPP_Decoder* class is defined, containing the *ASPP* module and the decoder previously implemented, in order to simplify the code in the main class. In the *forward()* method, instead, the feature maps are computed independently for each branch, and skip connections are added for both decoders (the low level features coming from the encoder, namely the tensor *features[4]*, are given as input to the decoder in both the segmentation and the depth branches). The final output predictions are then upsampled in order to have the same resolution as the input, and they are assigned to the corresponding keys of the *out* dictionary.

DeepLabv3+ model						
Run name	Branched architecture	#Parameters	Computational complexity	Metrics		
				total ↑	semseg ↑	depth ↓
P1.3_ddebf	✗	26.72 M	13.66 GMac	64.538	84.798	20.26
P2_115c1	✓	32.14 M	19.99 GMac	**65.417**	**84.883**	**19.466**

Table 10: Comparison between the branched and the joint architectures of the *DeepLabv3+* model.

A comparison between the performance of the branched model P2_115c1 and of the joint one P1.3_ddebf is presented in Table 10. A slight improvement in the branched architecture is observed, which comes mainly from the depth estimation task, while the performance of the segmentation

```python
class ModelDeepLabV3Plus_branched(torch.nn.Module):
    def __init__(self, cfg, outputs_desc):
        super().__init__()
        self.outputs_desc = outputs_desc
        ch_out_semseg = list(outputs_desc.values())[0]
        ch_out_depth = list(outputs_desc.values())[1]

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

        ch_out_encoder_bottleneck, ch_out_encoder_4x = get_encoder_channel_counts(cfg.model_encoder_name)

        self.branched_architecture = torch.nn.Module()
        self.branched_architecture.add_module("segmentation", ASPP_Decoder(ch_out_encoder_bottleneck, ch_out_encoder_4x,
                                                                            ch_out_semseg, tuple(cfg.ASPP_rates)))
        self.branched_architecture.add_module("depth", ASPP_Decoder(ch_out_encoder_bottleneck, ch_out_encoder_4x,
                                                                    ch_out_depth, tuple(cfg.ASPP_rates)))
    def forward(self, x):
        input_resolution = (x.shape[2], x.shape[3])
        features = self.encoder(x)
        lowest_scale = max(features.keys())
        features_lowest = features[lowest_scale]

        # Segmentation branch
        features_tasks_seg = self.branched_architecture.segmentation.aspp(features_lowest)
        predictions_4x_seg, _ = self.branched_architecture.segmentation.decoder(features_tasks_seg, features[4])
        predictions_seg = F.interpolate(predictions_4x_seg, size=input_resolution, mode='bilinear', align_corners=False)

        # Depth estimation branch
        features_tasks_depth = self.branched_architecture.depth.aspp(features_lowest)
        predictions_4x_depth, _ = self.branched_architecture.depth.decoder(features_tasks_depth, features[4])
        predictions_depth = F.interpolate(predictions_4x_depth, size=input_resolution, mode='bilinear',
                                          align_corners=False)

        out = {}
        for task, num_ch in self.outputs_desc.items():
            if task == "semseg":
                out[task] = predictions_seg
            if task == "depth":
                out[task] = predictions_depth
        return out
```

Figure 5: Code snippet of the branched architecture of *DeepLabv3+* model.

stays more or less unchanged. In fact, the `semseg` metrics yields an improvement of just 0.1%, going from `84.798` in `P1.3_ddebf` to `84.883` in `P2_115c1`, while with these modifications the `depth` metrics reaches the value `19.466`; overall, the branched architecture achieves a `total` score of `65.417`. The `grader` performances are `total: 65.453`, `semseg: 84.993` and `depth: 19.539`.
By observing the results in Table 13 of the *Appendix*, it can be seen that there is no significant difference from the results of the previous task. While the segmentation predictions are almost comparable, the only visible difference is that the depth predictions are now smoother, with more gradual changes as we move away from the camera.

On the other hand, it is witnessed a significant increase in the training time, with every single epoch lasting around 56 minutes instead of the 43 minutes in the previous task (an increase of 30% of the training time). This is justified from the evident increase in the amount of trainable parameters. In the branched architecture, while the parameters of the encoder are shared between the two tasks, the remaining parameters have doubled in number, since there are now two separate branches with their own *ASPP* module and decoder with skip connections. In fact, as it can be seen in Table 10, the number of parameters has increased from 26.72M in the joint architecture to 32.14M in the branched architecture. Even though the number of parameters in the *ASPP* module and decoder is doubled, the percentage increase is only 20.28%. The reason is that, as it can be observed by giving a closer look into the computed statistics, the majority of parameters is coming from the encoder network instead of the decoder. The number of parameters and the computational complexity is referred to an input RGB image (3 channels) with size 256×256 (the dimension of the crop size used during the training), and they are computed through *get_model_complexity_info()* function of the `ptflops` library [2].

Problem 3 - Task distillation

The branched architecture already increased the performance of the model, but further improvements can still be achieved thanks to more advanced techniques, such as task distillation. This approach is implemented through *Self-Attention* modules (*SA*) [9], which control the information flow across the different tasks and behave as a sort of gate function that learns to either exploit or ignore the features coming from other tasks. The whole implementation results in a new decoding stage, which fuses together the output of the previous decoder with the information distilled from the other tasks. Therefore, there are now two losses for each task, which are summed up together to have a single optimization objective.

The already implemented *SA* module consists in computing an attention mask, which is basically a sigmoid activation function applied to the output of a 3x3 convolution on the feature map coming from the decoders of the first stage. The attention mask is then multiplied with the same feature map of the previous 3x3 convolution. An example of how these computations look like, the formula for the segmentation branch is:

$$G_i^{depth} \leftarrow \sigma\left(W_g \otimes F_i^{depth}\right) \tag{4}$$

$$F_i^{o,seg} \leftarrow F_i^{seg} + G_i^{depth} \odot \left(W_g \otimes F_i^{depth}\right) \tag{5}$$

where G_i^{depth} is the attention mask computed on the features of first decoder of the depth estimation branch, W_g represents the kernel parameters (in this case it is a 3x3 convolution with *padding=1*), F_i^{seg} and F_i^{depth} are the output features of the decoders in the first stage, and $F_i^{o,seg}$ is the feature map given as input to the final decoder in the segmentation branch.

In fact, the output of the *SA* module is added to the output features of the decoders of the first stage (the feature maps before the final 1x1 convolution that computes the output predictions), and the result is given as input to the final decoders, which are implemented in the class *DecoderDeepLabV3p_no_skip_connections*. The two decoders in the final stage are different from the ones previously used, since skip connections are not added back this time. The code of the *DeepLabv3+* model with skip connections is displayed in Figure 6.

The task distillation yields a significant improvement in the performance with respect to the `P2_115c1` model, as shown in Table 11. The `total` metrics of the new model `P3_9387a` increases from `65.417` to `67.972`.

DeepLabv3+ model							
Run name	Branched architecture	Task distillation	#Parameters	Computational complexity	Metrics		
					total ↑	semseg ↑	depth ↓
P1.3_ddebf	✗	✗	26.72 M	13.66 GMac	64.538	84.798	20.260
P2_115c1	✓	✗	32.14 M	19.99 GMac	65.417	**84.883**	19.466
P3_9387a	✓	✓	35.69 M	39.34 GMac	**67.972**	84.837	**16.865**

Table 11: Comparison between the branched, the distillated, and the joint architectures of the *DeepLabv3+* model.

The main improvement is constituted by the `depth` metric, which drops from `19.466` to `16.865`. On the other hand, the `segmentation` does not show any improvements, with its value remaining steady around `84.8`. In particular, it can be observed that a slightly higher value was achieved in the run `P2_115c1`, namely `84.883` against `84.837`. However, the difference is in the order of 1e-2 and it is mainly due to the some fluctuations in the training and due to random initialization, which introduces some variance. The semantic segmentation does not take benefit from the features of the depth estimation branch, but the latter is instead improve by the segmentation features. A reason could be that thanks to the boundaries of the objects defined by the segmentation, the

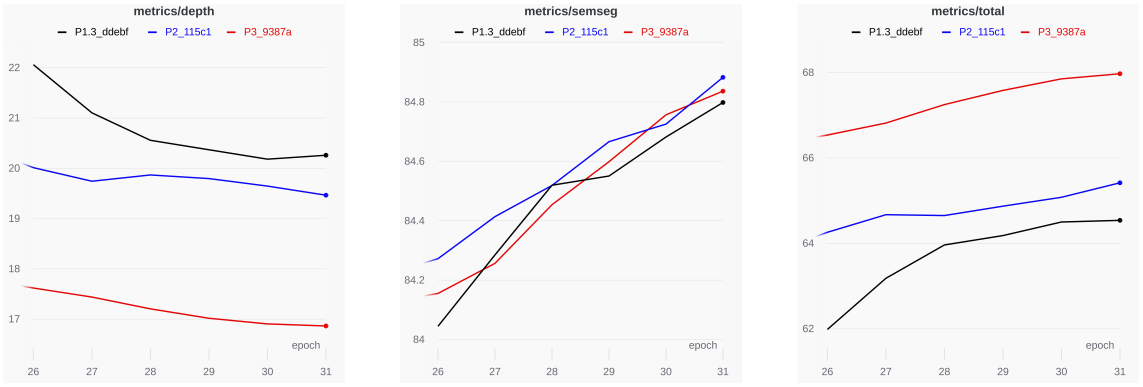
```python
class ModelDeepLabV3Plus_branched_distillation(torch.nn.Module):
    def __init__(self, cfg, outputs_desc):
        super().__init__()
        self.outputs_desc = outputs_desc
        ch_out_semseg = list(outputs_desc.values())[0]
        ch_out_depth = list(outputs_desc.values())[1]

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

        ch_out_encoder_bottleneck, ch_out_encoder_4x = get_encoder_channel_counts(cfg.model_encoder_name)

        self.branched_architecture = torch.nn.Module()
        self.branched_architecture.add_module("segmentation", ASPP_Decoder(ch_out_encoder_bottleneck, ch_out_encoder_4x,
                                                                            ch_out_semseg, tuple(cfg.ASPP_rates)))
        self.branched_architecture.add_module("depth", ASPP_Decoder(ch_out_encoder_bottleneck, ch_out_encoder_4x,
                                                                    ch_out_depth, tuple(cfg.ASPP_rates)))

        self.self_attention = SelfAttention(256, 256)
        self.decoder3 = DecoderDeeplabV3p_no_skip_connections(256, ch_out_semseg)
        self.decoder4 = DecoderDeeplabV3p_no_skip_connections(256, ch_out_depth)

    def forward(self, x):
        input_resolution = (x.shape[2], x.shape[3])
        features = self.encoder(x)
        lowest_scale = max(features.keys())
        features_lowest = features[lowest_scale]

        # Segmentation branch
        features_tasks_seg = self.branched_architecture.segmentation.aspp(features_lowest)
        predictions_4x_seg_1, features_4x_seg_1 = self.branched_architecture.segmentation.decoder(features_tasks_seg,
                                                                                                  features[4])
        predictions_seg_1 = F.interpolate(predictions_4x_seg_1, size=input_resolution, mode='bilinear')

        # Depth estimation branch
        features_tasks_depth = self.branched_architecture.depth.aspp(features_lowest)
        predictions_4x_depth_2, features_4x_depth_2 = self.branched_architecture.depth.decoder(features_tasks_depth,
                                                                                               features[4])
        predictions_depth_2 = F.interpolate(predictions_4x_depth_2, size=input_resolution, mode='bilinear')

        # Multi-task distillation
        features_SA_depth = self.self_attention(features_4x_depth_2)
        features_SA_seg = self.self_attention(features_4x_seg_1)

        predictions_4x_seg_3, _ = self.decoder3(features_4x_seg_1 + features_SA_depth, features[4])
        predictions_4x_depth_4, _ = self.decoder4(features_4x_depth_2 + features_SA_seg, features[4])

        predictions_seg_3 = F.interpolate(predictions_4x_seg_3, size=input_resolution, mode='bilinear')
        predictions_depth_4 = F.interpolate(predictions_4x_depth_4, size=input_resolution, mode='bilinear')

        out = {}
        for task, num_ch in self.outputs_desc.items():
            if task == "semseg":
                out[task] = predictions_seg_1 + predictions_seg_3
            if task == "depth":
                out[task] = predictions_depth_2 + predictions_depth_4

        return out
```

Figure 6: Code snippet of the branched architecture of *DeepLabv3+* model with task distillation.

depth estimation can better predict homogeneous depth values inside an individual segmented region, and heterogeneous depth values at the boundaries of different segmented regions. It seems instead that the semantic segmentation does not take advantage of the depth features: they are not adding any new useful information. The `grader` scores of the model P3_9387a are `total: 67.962`, `semseg: 84.936`, and `depth: 16.974`.

Finally, Figure 7 shows a graphical comparison of the three different architectures implemented so far, namely the standard *DeepLabv3+* (P1.3_ddebf, black curves), the corresponding branched architecture (P2_115c1, blue curves) and the one with task distillation implemented (P3_9387a, red curves). The behaviour of the different metrics during the training, namely `depth`, `semseg` and `total`, is shown in the three charts, providing an overview of what has been discussed so far.

From Table 13 of the *Appendix*, there is no significant visual improvement in the predicted output image for the semantic segmentation. However, in the output image for the depth estimation, most of the artifacts have disappeared and now the predictions are more local coherent, giving to the image a smoother, but not blurred, appearance. Moreover there is a significant improvement in



(a) (b) (c)

Figure 7: Overall view of the metrics of the three architectures. (a) depth (b) semseg (c) total

the depth estimation of the foliage in the foreground, which is very close to the camera: definitely a very positive improvement for a safer autonomous driving, since it is needed a special attention to the detection of close objects. An improvement that from the metrics values alone it was not possible to see.

As a final remark, many other approaches and modifications can be made to enhance even more the performance of the network. For instance, activating the dilation on the second and third layer of the encoder instead of the fourth layer; adding one more convolutional layer to the decoders with skipped connections and two more to the decoders placed after the *Self-Attention* modules; exploiting a *L1* loss instead of the *L2* loss for the depth estimation; increasing the rates of the *ASPP* module from 3,6,9 to 6,12,18; adding multiple *Squeeze&Excitation* modules [6]: one after the encoder, two after the *ASPP* modules (one each), four inside the decoders (one each), before the prediction layer. To not exceed the training time required up to now ($\sim$ 35 hours), the batch size and the number of epochs are reduced respectively from 8 to 4 and from 32 to 16. With these architecture improvements and model modifications, the network `P3+_39a6e` performs `total: 71.698`, `semseg: 87.129`, `depth: 15.43`. There is an improvement not only in depth estimation but also a in the semantic segmentation, a good result since the latter was until now practically unaffected neither by the branched architecture nor by the addition of the task distillation. The `grader` scores of this enhanced model are `total: 71.816`, `semseg: 87.317`, `depth: 15.501`.

extra improvements			
Run name	Metrics		
	total ↑	semseg ↑	depth ↓
P3_9387a	67.972	84.837	16.865
P3+_39a6e	**71.698**	**87.129**	**15.430**

Table 12: Comparison between the branched and joint architectures of the *DeepLabv3+* model.

From Table 13 of the *Appendix* the improvement of the semantic segmentation can be seen in the boundaries, especially the ones of small or thin objects, like the poles at the side of the road. The output image for the depth estimation is smoother and with more recognizable boundaries with respect to model `P3_9387a`, but the most obvious difference is that now the foliage in the foreground is not predicted as close as it should be. Even if the `depth` is decreased even more in the enhanced model (probably for the use of the *L1* loss), the metric value should not be the only assessment resource. Since the goal is to construct a network specifically for autonomous driving, it is very important to correctly detect objects close to the camera. The enhanced model achieves better metrics scores both in the segmentation and the depth estimation, but regarding the latter it could be less safe in an autonomous driving task.

Naturally, many other improvements can be made. The modifications shown are weighted so to not increase the training time, but if it is not a big concern, then more addition to the architecture can be assembled, such as more convolutional layers, bigger kenels, adding skip connection also to the decoders after the *Self Attention* modules and to the encoder, increasing the batch size and the number of epochs, etc. Moreover, the network could benefit for some other modifications that do not change the architecture, such as exploiting the training data augmentation (using geometric but also colour augmentation), testing the reverse Huber loss for the depth estimation, or using a boxcox transformation [3] to improve the depth normalization process. The latter is useful since the depth values normally do not follow a normal distribution, which is an important requirement for a better training. The box-cox transformation converts the depth values to a normal distribution, which could enhance the training of the depth estimation.

Appendix

(a) (b) (c)

Figure 8: Example of one of the images used for the validation. (a) RGB image fed to the network (b) Ground Truth for the semantic segmentation (c) Ground Truth for the depth estimation

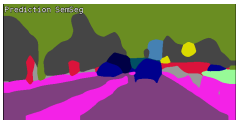
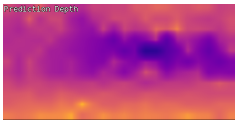
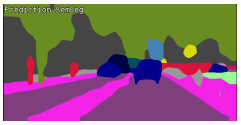
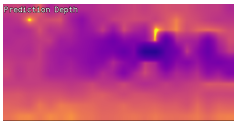
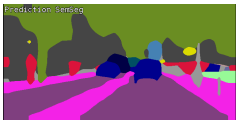
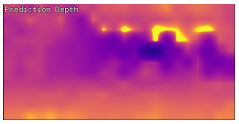
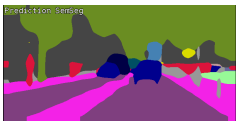
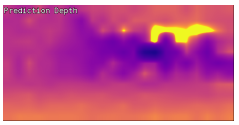
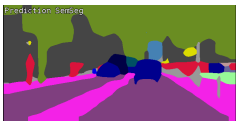
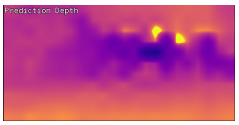
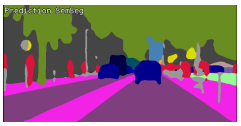
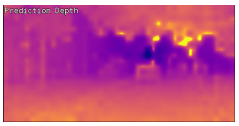
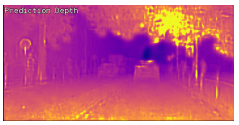
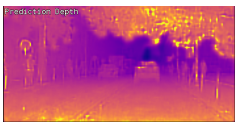
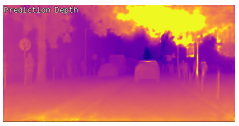
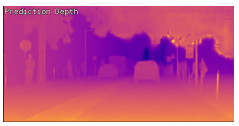
Run name	Semantic segmentation	Depth estimation
original_37628		
P1.1a_d68b6		
P1.1b_0b891		
P1.1c_a3cae		
P1.2a_44ba0		
P1.2b_172ab		
P1.3_ddebf		
P2_115c1		
P3_9387a		
P3+_39a6e		

Table 13: Prediction results of the best models of each subsection of the project.

Problem	Run name	Complete run name
1.1.a	original_37628	G18_0403-2221_P1_original_test_37628
	P1.1a_1f725	G18_0404-2015_P1.1a_SGD_lr_test4_1f725
	P1.1a_61964	G18_0404-1826_P1.1a_SGD_lr_test3b_61964
	P1.1a_06c0d	G18_0404-1527_P1.1a_SGD_lr_test1b_06c0d
	P1.1a_47e22	G18_0405-1837_P1.1a_Adam_lr_test4_47e22
	P1.1a_d68b6	G18_0405-1318_P1.1a_Adam_lr_test1b_d68b6
	P1.1a_27e27	G18_0405-0837_P1.1a_Adam_lr_test2_27e27
	P1.1a_4d200	G18_0405-1347_P1.1a_Adam_lr_test3_4d200
	P1.1a_d10d0	G18_0407-0125_P1.1a_Adam_lr_test6_d10d0
	P1.1a_237dc	G18_0408-1946_P1.1a_test8_237dc
	P1.1a_81237	G18_0406-2049_P1.1a_Adam_lr_test5_81237
	P1.1a_10da7	G18_0407-2049_P1.1a_Adam_lr_test7_10da7
1.1.b	P1.1b_79cc1	G18_0407-1220_P1.1b_test7_79cc1
	P1.1b_3bd9c	G18_0407-2212_P1.1b_test8_3bd9c
	P1.1b_0b891	G18_0409-1032_P1.1b_test9_0b891
	P1.1b_14fc8	G18_0409-1225_P1.1b_test10_14fc8
	P1.1b_be212	G18_0409-2316_P1.1b_test11_be212
1.1.c	P1.1c_756bb	G18_0411-1526_P1.1c_test4_756bb
	P1.1c_53575	G18_0410-1025_P1.1c_test1_53575
	P1.1c_4a169	G18_0410-1414_P1.1c_test2_4a169
	P1.1c_6a37c	G18_0411-1459_P1.1c_test3_6a37c
	P1.1b_b5ac0	G18_0410-1917_P1.1b_test9b_b5ac0
	P1.1c_73e8c	G18_0410-2315_P1.1c_test6b_73e8c
	P1.1c_a3cae	G18_0412-2033_P1.1c_test6d_a3cae
	P1.1c_da60c	G18_0411-0702_P1.1c_test2b_da60c
1.2.a	P1.2a_44ba0	G18_0413-1240_P1.2a_test1_44ba0
1.2.b	P1.2b_172ab	G18_0415-1021_P1.2b_test1_172ab
1.3	P1.3_ddebf	G18_0415-0651_P1.3_test2_ddebf
2	P2_115c1	G18_0417-2157_P2_test2_115c1
3	P3_9387a	G18_0427-1826_P3_test4_9387a
	P3+_39a6e	G18_0507-1717_P3+_enhanced_39a6e

Table 14: Full name of all the models presented.

References

[1] Keras Tuner documentation. URL https://keras-team.github.io/keras-tuner/.

[2] URL https://pypi.org/project/ptflops/.

[3] G. E. P. Box and D. R. Cox. An analysis of transformations. *Journal of the Royal Statistical Society. Series B (Methodological)*, 26(2):211–252, 1964. ISSN 00359246. URL http://www.jstor.org/stable/2984418.

[4] 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.

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

[6] J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. *CoRR*, abs/1709.01507, 2017. URL http://arxiv.org/abs/1709.01507.

[7] D. Neven, B. D. Brabandere, S. Georgoulis, M. Proesmans, and L. V. Gool. Fast scene understanding for autonomous driving. *CoRR*, abs/1708.02550, 2017. URL http://arxiv.org/abs/1708.02550.

[8] S. Vandenhende, B. D. Brabandere, and L. V. Gool. Branched multi-task networks: Deciding what layers to share. *CoRR*, abs/1904.02920, 2019. URL http://arxiv.org/abs/1904.02920.

[9] D. Xu, W. Ouyang, X. Wang, and N. Sebe. Pad-net: Multi-tasks guided prediction-and-distillation network for simultaneous depth estimation and scene parsing. *CoRR*, abs/1805.04409, 2018. URL http://arxiv.org/abs/1805.04409.