CVL, ETH Zurich | Deep Learning for Autonomous Driving

Project 2: Multi-task learning for semantics and depth

29/03/24-05-2020

General Info: This project accounts for 30% of your grade. You are welcome to ask questions in Piazza. If you think that your question may reveal the solution, please feel free to email your question to Stamatios Georgoulis (stamatios.georgoulis@vision.ee.ethz.ch), Anton Obukhov (anton.obukhov@vision.ee.ethz.ch) or Dengxin Dai (dai@vision.ee.ethz.ch).

Overview: In this exercise we will delve into Multi-Task Learning (MTL) architectures for dense prediction tasks. In particular, for semantic segmentation (i.e. the task of associating each pixel of an image with a class label, e.g. person, road, car, etc.) and monocular depth estimation (i.e. the task of estimating the per-pixel depth of a scene from a single image). As many other tasks nowadays, semantic segmentation and monocular depth estimation can be effectively tackled by using Convolutional Neural Networks (CNNs) [7]. To achieve state-of-the-art results, deep CNN models [5] of fully convolutional networks [8] are typically trained in datasets that contain a large number of fully-annotated images, that is, images with their corresponding ground-truth label. This allows the networks to encode feature representations that are discriminative for the task at hand. In what follows, we are going to train MTL models to jointly perform semantic segmentation and monocular depth estimation.

Training models in the cloud: Each team will be given an account with Amazon SageMaker to conduct the experiments required for solving the problems in this assignment. The training script can be started from a Jupyter notebook as described in the SageMaker and PyTorch tutorials (28.02.2020 and 13.03.2020). For further details please refer to the slides and video published at the course website. All training hyperparameters can be set in the hyperparameter dictionary in the estimator definition and are described in `mtl/utils/config.py`. The notebook itself contains a few useful tips about the recommended workflow in SageMaker.

Refer to the slides we have compiled (31.03.2020) for (1) how to use your personal Git in SageMaker; (2) applying for SageMaker Training Instances; and (3) seeing overview of Training Jobs and notebooks. Each team can apply for and use **two** "p2.xlarge Managed Spot Training" instances (slide 9). Please apply for the training instances as soon as possible as it may take some time for the AWS support to process your request.

The new tutorial is an extension of the previous two SageMaker tutorials. Important are the following two differences: (1) Explanation how to use private Git repositories, for example by the one you get by forking our GitLab repository; (2) How to apply for "SageMaker Training" instances, which are different to EC2 instances and required for training networks in SageMaker. Note that if you already applied for EC2 instances you need to apply again for SageMaker Training instances, and if you did not yet applied for EC2 instances it is not required.

SageMaker instances may take up to a few minutes to launch. This should not be a problem if the new experiment runs smoothly, as is usually the case with changing hyperparameters. However, this is prohibitively long when submitting unchecked code changes, when a syntax error can prevent the experiment from starting. Therefore it is recommended to set up a local python virtual environment as per `README.md`, and run one-two training iterations (not epochs!) on a local machine without specifying `CUDA_VISIBLE_DEVICES`, before submitting the code to SageMaker. You will also need this local environment to inspect TensorBoard logs of multiple experiments in an aggregated view.

TensorBoard allows researchers to monitor vital parameters of their experiments. During the

training phase, one can check that the experiment did not crash using the `Scalars` tab, or observe the quality of predictions using the `Images` tab. Most scalars and images refresh once per epoch, except for the training loss, which is live. We recommend setting "Smoothing" to zero and disabling "Ignore outliers" checkbox after every page reload. When running TensorBoard on a parent root of multiple experiments, it becomes possible to compare training dynamics between experiments, and judge about the efficiency of improvements. Individual experiments' toggles can be found on the left-bottom side of TensorBoard.

We describe two ways to keep track of the experiments: online and offline viewing. To get the online mode working, each team has to register an account with ngrok.com, and obtain an authentication token. This token should be passed as the value of `ngrok_auth_token` hyperparameter in the notebook code before running an experiment. After the training begins, you will find a link to the TensorBoard of your running experiment in the ngrok user account, under the tab "Status". Each ngrok authentication token can be used with just one running experiment at time. If your team needs to run two experiments in parallel, have each team member create an account with ngrok and share the tokens.

After an experiment ends, the online TensorBoard session becomes unavailable, but you can synchronize your S3 buckets containing experiment artifacts to a local machine, and run TensorBoard on the parent directory of multiple experiments, as it is described in the `README.md`. Remember to activate the virtual environment where the requirements were installed.

Metrics: The following metrics will be definitive for each experiment outcome:

— IoU (intersection-over-union) is a metric of performance of the semantic segmentation task. Its values lie in the range [0,100]. Higher values are better. It is shown as `semseg` in the TensorBoard metrics summary.

— SI-logRMSE (scale-invariant log root mean squared error) is a metric of performance of the monocular depth prediction task. Its values are positive. Lower values are better. It is shown as `depth` in the TensorBoard metrics summary.

— The Multitask metric is a simple product of the aforementioned task-specific metrics, computed as $max(iou-50,0)+max(50-silogrmse,0)$. Its values lie in the range [0,100]. Higher values are better. It is shown as `grader` in the TensorBoard metrics summary.

Submission: For each problem statement and question, the final report should contain an accurate and complete description of your solution. There is no page limit, but please avoid lengthy and redundant descriptions. You should also include a few indicative figures from TensorBoard and relevant code snippets (changes made on top of the template, or other incremental changes). Final reports must be sent by each team in a file named `dlad_ex2_report_YOUR_TEAM_NAME.zip` to Dengxin Dai (dai@vision.ee.ethz.ch) by 08 May 2020 with subject `"DLAD EX2 final report: YOUR_TEAM_NAME"`.

Each question (e.g. "How does SGD compare to Adam?") assumes your team running one or a few experiments, and using the metrics summary from TensorBoard to back up the claim (answer). Please only use the three metrics mentioned in Sec. Metrics in your final report. When choosing which data point to report, you should choose the best value of the `grader` metric over the whole experiment run, and report all three metrics at that point. Double check that "Smoothing" slider is set to zero before collecting metrics from TensorBoard.

Each time the training script ends without failure, you can find a `submission.zip` archive in the corresponding experiment log directory. For the sake of reproducibility and record-keeping, make sure to back up each `submission.zip` file, results from which are mentioned in the final report. We may ask you to share some of these files later for inspection. These files are considered a part of your final report, even if we never request them. Each `submission.zip` file automatically includes (1) the source code used to run the experiment, (2) hyperparameters, (3) best model checkpoint, (4) task predictions on the test split of the dataset, and (5) a trace

of all metrics. It does not include TensorBoard logs. The file has a certain structure that may be lost in case of repacking (unzip - zip), which in turn may cause grader errors. You can also submit `submission.zip` file to the grader, and include test scores on top of validation scores in your final report. You are expected to submit at least 3 experiments to the grader, however there is no upper limit.

If you are using AWS SageMaker, after successful training, the `submission.zip` file is accessible on the S3 storage. In the S3 console https://s3.console.aws.amazon.com select the bucket you attached to the notebook instance and open it. It will show you a list with folders named `<experiment name>-<start time>`, where each folder corresponds to one Training Job. Open the run you want to download, and given the training finished successfully, the folders `output` and `source` should be available. Navigate into `output` and download `model.tar.gz`. If you unpack this folder, you find all logs including the `submission.zip` file.

Grader: The grader is hosted by CodaLab, a free service for holding ML challenges. To register in the challenge navigate to https://competitions.codalab.org/ and create individual accounts for each team member.

NB: The leaderboard is not anonymous, so if you wish to remain anonymous to the peers choose cryptic user and team names.

Next, each team member should follow the grader URL displayed in Sec. Links, and click `"Participate" -> "Register"`. After that send an email to Anton Obukhov and Dengxin Dai with the subject `"DLAD EX2 user: YOUR_CODALAB_USERNAME"`, with your real name and legi in the body. After all team members are approved, you must form teams using the "Teams" tab functionality. Choose a team leader, who will create a new team, and handle other team members' requests. Make sure to not make any individual submissions before joining a team properly.

To get a submission graded, navigate to `"Participate" -> "Submit / View Results"`, click on a large `"Submit"` button, and wait to get the `submission.zip` file uploaded. You may need to refresh the page with `F5` to see submission status updates. If you are satisfied with the scores, you can push a submission to the leaderboard by clicking the corresponding button. Make sure each submission has a short description of what was implemented or which hyperparameters were changed.

Extra notes:

— Running multiple experiments in parallel is possible by creating multiple SageMaker Notebook instances;

— Working with python code is especially convenient with PyCharm (free student license);

— Early stopping after a couple of epochs of observing TensorBoard can be used to reject poor choice of some of the hyperparameters (but not all);

— If training crashes after the first step, most likely the process is out of GPU memory, with either batch size, model, or crop size too large;

— Without code modification, the `submission.zip` will be graded in following ballpark: "Multitask": 40.48, "IoU": 67.46, "SI-logRMSE": 26.98;

— Code sharing is not allowed during and after competition ends. To use versioning (github or gitlab), make sure to use a private repository;

— Make sure to inspect each archive for leaks of sensitive information before submitting it to the grader or sharing with us. Keep in mind that some operating systems (OSX for one) unpackers remove the original zip file - see the pink note about retaining all `submission.zip` files;

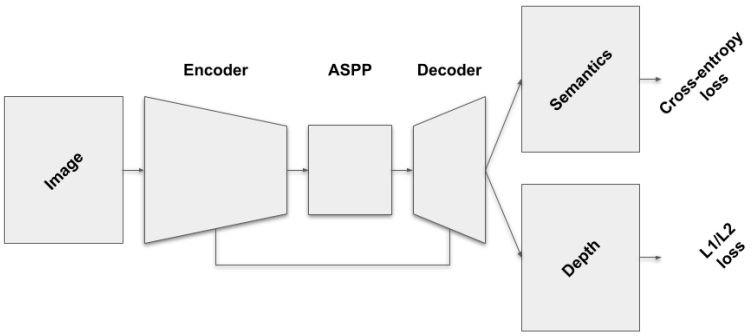

Figure 1: Joint architecture

— Only the provided MiniScapes dataset is allowed to train the model;

— Only ResNet-34 is allowed as the Encoder backbone;

— Only ImageNet-pretrained or random weight initializations are allowed to initialize a model during creation;

Links:
Course website: `https://www.trace.ethz.ch/teaching/DLAD/spring2020/index.html`
Solution template: `https://gitlab.ethz.ch/obukhova/dlad_ex2_multitask` (login required)
Codalab challenge: `https://competitions.codalab.org/competitions/24021?secret_key=fe94fd11-9ef9-41ba-bf84-493f15f70c35`
MiniScapes dataset: `https://s3.amazonaws.com/dlad-miniscapes/miniscapes.zip` (1.6 GB)
Please keep the code and data private and do not release them.

Problem 1. Joint architecture

(4+2+3+3=12 points)

Your starting point is a DeepLab model [1, 2, 3, 4] that consists of a ResNet-like Encoder [5], an ASPP module, and a Decoder with skip connection. The starting point functions properly and can be trained straight away, however, the performance will be subpar: we intentionally chose sub-optimal default values for some of the hyperparameters, and short-circuited some of the model parts, namely ASPP and the Decoder. Since we need to solve both tasks (semantics and depth) under a single model, a naive MTL solution is to share all operations (i.e. Encoder, ASPP, Decoder) between tasks except for the last convolution that maps the features of the preceding layer to $n_{\text{classes}} + 1$ channels. The former n_{classes} channels correspond to pixel-wise class probability distribution before softmax (logits) for the semantic segmentation task, while the latter 1 channel corresponds to the regressed depth values (normalized distance in meters) for the monocular depth estimation task. This joint architecture is depicted in Figure 1.

1. Hyper-parameter tuning (4 pts): As a first step you need to familiarize yourself with the hyperparameters. The file `mtl/utils/config.py` describes hyperparameters, which can be changed using command line keys to the training script. We encourage you to try different settings and examine the effect that each parameter has in the final result in order to get a better understanding of the codebase. Once a better hyperparameter value is found, it is safe to keep it for the future experiments, provided the training time did not increase by too much. More specifically, you should investigate the following options and report informative conclusions about your findings.

   (a) The *optimizer* choice. How does using *SGD* compare to *Adam*? Keep in mind that the default value of `optimizer_lr` hyperparameter corresponds to the default value of `optimizer`, and should be a couple orders of magnitude smaller for Adam.

(b) The *learning rate*. How does logarithmically changing the learning rate affect the learning? You can try 3 different values as an indicative bracket.

(c) The *batch size*. Is a larger batch size preferable over a smaller one for our tasks?

(d) *Task weighting*. When multiple tasks are learned together, their individual losses should be accumulated before updating the network weights during the training stage. This creates the need to properly balance the losses of the different tasks to avoid a scenario where one task overwhelms the others. Can you find proper loss weights for the employed tasks to improve their joint performance?

It is recommended to repeat hyperparameters search during or after completing the rest of the programming assignments. Normally, as the model changes, the best hyperparameters drift away from their initial values.

2. Hardcoded hyperparameters (2 pts): now you need to study the main building blocks of the experiment, model, and loss modules (arranged in the respective subdirectories under `mtl` directory), and perform one-line changes of the code to improve the model.

   (a) Initialization with ImageNet weights (1 pts): Can you verify whether the encoder network is initialized with weights of a model trained on ImageNet classification task? What is the effect of switching this option? Make sure to keep the option enabled before proceeding to the next questions.

   (b) Dilated convolutions (1 pts): Look closely at the Encoder code in `model_parts.py` and check whether dilated convolutions are enabled. This aspect is closely related to the term "output stride" used in [4]. You can use the commented `print` statement in `model_deeplab_v3_plus.py` to help you see the mapping of each scale of the feature pyramid to the respective number of channels. This mapping does not change throughout training, so finding the right flags is possible quickly in the standalone environment. The largest scale factor in the pyramid corresponds to the "output stride". Set dilation flags to (False, False, True) and train the model. Does the performance improve? If so, why? Use this model as a reference when reporting the effects of ASPP and Skip Connection. Keep the flags set this way until you start Problem 4, in which you can revisit these flags for the better performance.

3. *ASPP module* (3 pts): You are to implement the ASPP module [3, 4], whose design details are provided in the referenced papers. You are already given the skeleton of the *ASPP* class with the proper inputs and outputs, and you are asked to replace the current trivial functionality with the proper one. The details of the ASPP module can also be found in Figure 2. If desired, you can use the *ASPPpart* class as well. The code parts which require work are marked with `TODO` annotations. Does the inclusion of the ASPP module in your model help or hurt the performance? Why?

4. *Skip connection* (3 pts): You are to implement the decoding stage with skip connection as done in [4]. You are given the *DecoderDeeplabV3p* class that already contains the appropriate inputs and outputs. You have to replace the current functionality with the intended one, essentially processing the features that come from the encoder and the ASPP module, and outputting the final channels that contain the predictions. The detailed diagram of this part can be found in Figure 2. The code parts which require work are marked with `TODO` annotations. How does the model performance change with skip connections functioning?

Problem 2. Branched architecture (4 points)

In the previous problem, we used a joint architecture, which shared all network components except the last convolutional layer – to learn both tasks. Another MTL solution is the adopt a branched architecture [10, 11], where a common encoder is used for both tasks, but task-specific

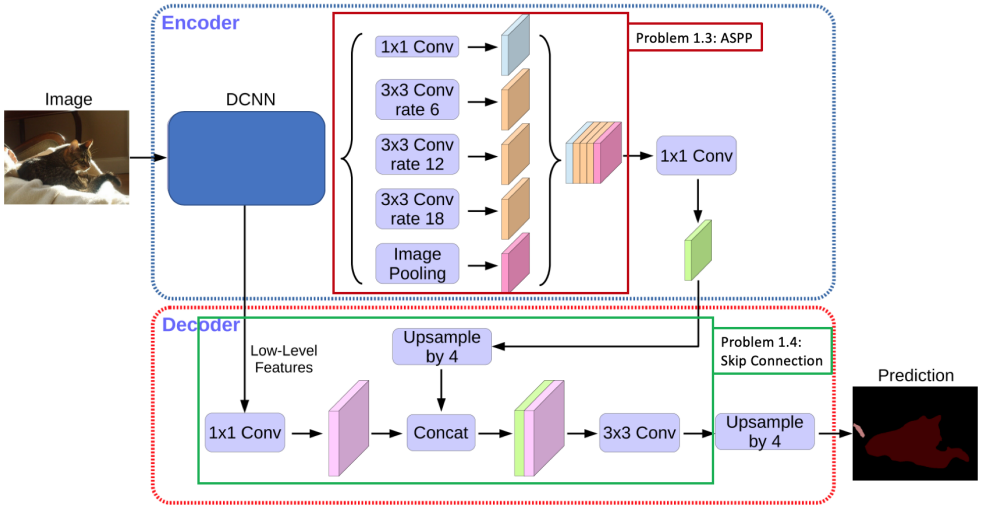

Figure 2: ASPP module and Skip Connection

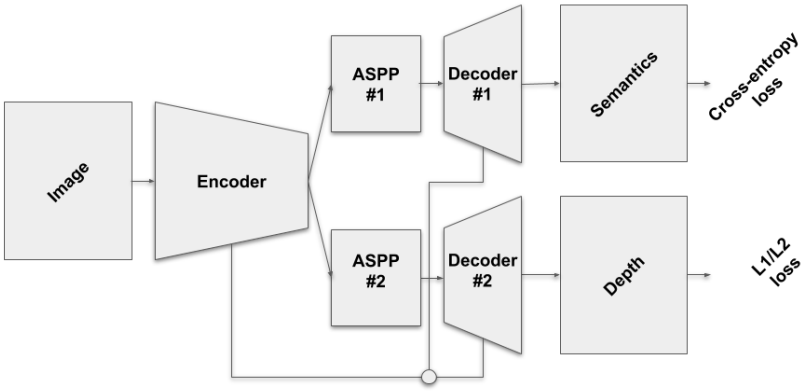

Figure 3: Branched architecture

ASPP modules and decoders are implemented for semantic segmentation and monocular depth estimation respectively. Figure 3 gives an overview of this architecture.

1. As part of this problem you are asked to implement this branched architecture using the same building blocks: Encoder, ASPP, and Decoder modules. To narrow down the scope of effort, and prevent unintentional breaking of the pipeline, all code changes should be restricted to `mtl/models` path. How does it compare to the joint architecture both in terms of performance, but also w.r.t. the model size and the required computations?

NB: Instead of modifying `ModelDeepLabV3Plus` class in `mtl/models/model_deeplab_v3_plus.py`, create a new file in `mtl/models` directory, and hook up your new model to the framework in two places: (1) add some new text identifier of it to the `choices` dictionary of the `model_name` command line parameter in `mtl/utils/config.py`, and (2) add the mapping of this new identifier to your new model's class name in `mtl/utils/helpers.py` in `resolve_model_class` function. Now you can dispatch between your two models using the `model_name` command line argument.

Problem 3. Task distillation (6 points)

Building upon a branched MTL architecture with a shared encoder followed by task-specific operations, recent works [13, 14, 12] proposed to leverage the initial task predictions to distill information across tasks. This is typically done by using an attention module to select the relevant features from another task that can be useful for our main task. One such architecture is depicted in Figure 4. Here, the features before the last convolutional layer of each task-specific decoder (e.g. *Decoder #1*) are summed with the corresponding features coming from the other task (*Decoder #2*) after applying self-attention (*SA*) to the latter. Then, the summed features are passed through another decoder module (*Decoder #3*) to get the final task prediction. This distillation procedure is applied for every task.

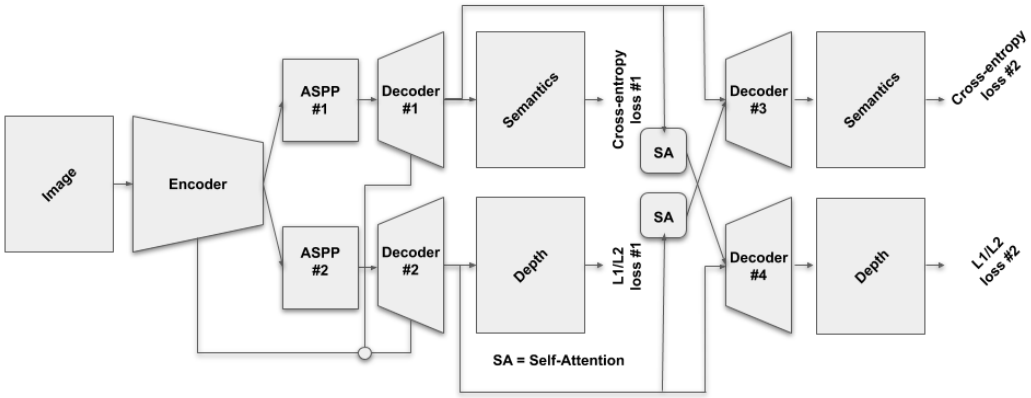

Figure 4: Branched architecture with task distillation

— As part of this problem you are asked to implement the aforementioned architecture (4 pts). Note that, the *SelfAttention* class is already implemented for you. How does the distillation procedure compare to the branched architecture in the previous problem? When implementing the final decoder modules (*Decoder #3* & *#4*), you can adopt the design of the initial ones (*Decoder #1* & *#2*).

— You can further use your creativity to improve the decoders (2 pts). What are your changes, why, and what are the results?

NB: Similarly how it is recommended for the branched architecture, put a new model into a separate file, and hook it up to the training code in two places.

Problem 4. Winning the Challenge (4+4=8 points)

All teams based on the Multitask metric will be awarded some scores based on the ranking. The scores will be spread evenly (i.e., linearly) throughout the interval [4/N, 4] based on the ranking, where N is the number of total teams. The best-performing team will get 4 points.

You can bring in your own ideas for multi-tasking networks here. The novelty of your ideas can be awarded a score between 0 and 4. Reading more papers in the literature and adapt their ideas to this task can be a starting point. Your ideas can lead to further collaboration beyond this course such as conducting a semester/master project with us.

Please document your ideas and your ranking clearly in the final report. The challenge will be closed two days before the submission deadline.

Below are a few ideas which can help with the leaderboard (you can choose which to try):

— Recommended `batch_size` interval is between 2 and 16. Larger batches give cleaner gradients, but slow down training. Smaller batches give noisy gradients, but speed up training. To maintain the same number of training steps, one has to adjust `batch_size` and `num_epochs` proportionally. The default values are chosen low, to let you run more experiments with faster turnaround.

— Geometric data augmentations can often help with model generalization. Which of them are enabled by default? Check for example the DeepLab paper [2] for data augmentation, try enabling some or all of them. How does this change the model performance?

— Refer to another DeepLab paper [4] to find out the rationale behind the crop size selection. How does the crop size correlate with the random scaling augmentation bracket, and the dataset image dimensions? Which crop size could work better with MiniScapes?

— The DeepLab architecture utilizes just one skip connection from the Encoder at 4x scale. In the code provided, Encoder output is a feature pyramid with every scale present. Try to improve the level of detail of the task predictions by making use of more skip connections at finer resolutions.

— The `SqueezeAndExcitation` module that implements the Squeeze-And-Excitation mechanism [6] is provided in the code package. Check the paper and possibly include this mechanism in your model too [9].

— Normal distribution is a recurring assumption behind many design decisions in neural networks. You can often see in regression problems, that the ground truth data is whitened, that is, has zero mean and unit variance. This way CNNs are easier to train and produce higher quality models. Depth estimation is no exception; despite the ground truth values being in meters (taking positive values $\gg 0$), we compute a mean and standard deviation over the training split (refer to `mtl/datasets/dataset_miniscapes.py:181-187`) and use these values to convert between depth in normalized meters (having zero mean unit variance over the training split), and depth in meters. However, depth values in meters (both normalized and absolute) in 2D images do not follow normal distribution, which leads to excessive concentration of values around the lower bound of the data range (check TensorBoard "Histograms" tab). Can you think of a more balanced unit of measurement (or a function) to use for the task of depth regression? If yes, augment depth normalization process to use this new unit, such that the model will learn to predict values on the normalized scale of this new unit. Do not forget to collect statistics for the new unit. Implementation of this idea would affect the following files: `mtl/scripts/compute_statistics.py`, `mtl/utils/transforms.py`, `mtl/experiments/experiment_semseg_with_depth.py`.

References

[1] Chen, L.C., Papandreou, G., Kokkinos, I., Murphy, K., Yuille, A.L.: Semantic image segmentation with deep convolutional nets and fully connected crfs. arXiv preprint arXiv:1412.7062 (2014)

[2] Chen, L.C., Papandreou, G., Kokkinos, I., Murphy, K., Yuille, A.L.: Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. IEEE transactions on pattern analysis and machine intelligence **40**(4), 834–848 (2017)

[3] Chen, L.C., Papandreou, G., Schroff, F., Adam, H.: Rethinking atrous convolution for semantic image segmentation. arXiv preprint arXiv:1706.05587 (2017)

[4] Chen, L.C., Zhu, Y., Papandreou, G., Schroff, F., Adam, H.: Encoder-decoder with atrous separable convolution for semantic image segmentation. In: Proceedings of the European conference on computer vision (ECCV). pp. 801–818 (2018)

[5] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 770–778 (2016)

[6] Hu, J., Shen, L., Sun, G.: Squeeze-and-excitation networks. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 7132–7141 (2018)

[7] Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems. pp. 1097–1105 (2012)

[8] Long, J., Shelhamer, E., Darrell, T.: Fully convolutional networks for semantic segmentation. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 3431–3440 (2015)

[9] Maninis, K.K., Radosavovic, I., Kokkinos, I.: Attentive single-tasking of multiple tasks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 1851–1860 (2019)

[10] Neven, D., De Brabandere, B., Georgoulis, S., Proesmans, M., Van Gool, L.: Fast scene understanding for autonomous driving. arXiv preprint arXiv:1708.02550 (2017)

[11] Vandenhende, S., Georgoulis, S., De Brabandere, B., Van Gool, L.: Branched multi-task networks: deciding what layers to share. arXiv preprint arXiv:1904.02920 (2019)

[12] Vandenhende, S., Georgoulis, S., Van Gool, L.: Mti-net: Multi-scale task interaction networks for multi-task learning. arXiv preprint arXiv:2001.06902 (2020)

[13] 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: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 675–684 (2018)

[14] Zhang, Z., Cui, Z., Xu, C., Yan, Y., Sebe, N., Yang, J.: Pattern-affinitive propagation across depth, surface normal and semantic segmentation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 4106–4115 (2019)