CVL, ETH Zurich | Deep Learning for Autonomous Driving

Project 2: Multi-Task Learning for Semantics and Depth

25% of the course grade | 01-04-2022 – 13-05-2022

Quick Links:
→ Course website
→ Piazza forum and announcements*
→ Codalab grader and leaderboard
→ Latest version of the solution template and training instructions
→ Latest version of this document
→ AWS tutorial
* Announcements about exercise updates will be made on Piazza. Questions about the exercise are welcome; however, if you believe that your question may reveal the solution, please make sure to *post to instructors* when creating a post.

Introduction: 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 with many other tasks nowadays, semantic segmentation and monocular depth estimation can be effectively tackled by using Convolutional Neural Networks (CNNs) [10]. To achieve state-of-the-art results, deep convolutional networks [8, 11] are typically trained on 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 perform semantic segmentation and monocular depth estimation jointly.

Training models in the cloud: Each team will be given an account for Amazon Elastic Compute Cloud (AWS EC2) to conduct the experiments required for solving problems in this assignment. You are allowed to use two spot p2.xlarge instances. The main source of technical information not covered by this document is the solution template `README.md` (see Quick Links). Everyone (including the advanced users of AWS) should study it and follow the explained steps for everything related to:

— AWS setup
— Training in the cloud
— Interactive development
— Progress monitoring
— Checkpointing and accessing submission archives

The solution template comes ready for cloud training and implements a baseline for the problems below. After each successful training of the model, the code will generate predictions on the RGB images of the test split (see Dataset section below). The submission archive containing predictions is created automatically. Apart from test predictions, the archive also contains the experiment configuration, trained model weights, source code, and the training log. This submission archive will also be uploaded to your S3 bucket, which you indicated upon setting up the AWS environment. For your convenience, the S3 link to the submission archive will be available in W&B overview of the experiment run (see W&B section below). Each submission archive can be downloaded locally and then uploaded to the grader (see "Grader" section) to (1) participate in the leaderboard and (2) obtain the test split metrics for the final submission (see "Final submission" section).

Dataset: For this exercise, we use a toy dataset of synthetic scenes in the autonomous driving context. The dataset is composed of predefined splits with 20000 training images, 2500 validation, and 2500 test images. The validation and test splits are quite similar, so observing the validation performance in W&B should give a good estimate of the expected score with the grader (see W&B, Grader). The dataset contains three modalities for each image sample: RGB, Semantic annotation, and the Depth map. The solution template and the AWS scripts automatically handle dataset downloading; nothing should be changed about the data loading pipeline.

Metrics: The following metrics are used to evaluate each experiment's 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 `metrics_summary/semseg` in W&B.

— 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 `metrics_summary/depth` in W&B.

— 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 `metrics_summary/multitask` in W&B.

W&B: The code template uses Weights and Biases for the training progress monitoring. As part of the setup, each team will need to register a free account with the service. Navigate to https://wandb.ai, register a new account (or login with email), then navigate to `Settings` → `API keys`. A default key will be available: hover over the key, click the plus button to copy it, and use this key when prompted on the first time of running `aws_start_instance.py`. The default visibility of W&B projects is "private", as indicated by the closed lock icon against the project name – make sure to keep it that way throughout the course. Feel free to erase "bad" runs to avoid cluttering; however, keep all the runs that you reference in your final report (see "W&B sheet" section) until the end of the course.

W&B allows inspecting the training dynamics (e.g., loss curves, validation metrics), collecting advanced statistics (histograms of weights or activations), as well as displaying images of predictions. The code template makes heavy use of all these features. The main tabs of interest within each individual run (select a certain run from the dashboard first) are "Overview" and "Charts", which can be found on the left side. The former provides, among other run-specific information, external links to the S3 location with checkpoints (required for resuming an abruptly terminated experiment) and the final submission files.

However, the main benefit of W&B for efficient development comes in the aggregated view of the experiments (select the project DLAD-Ex2 from the dashboard): the "Charts" and "Table" tab allows one to compare different runs with different settings and identify configurations which cause improvement (not automatically though). Each experiment may be given a name (such as "feature1value1_feature2value2") by changing the value of the `--name` flag in `aws_train.py` script to allow easy identification of a run. Further, we recommend to add feature1 and feature2 keys to the configuration file (config.py). This will allow slicing the experiments in W&B according to the value of the corresponding config keys.

At the end of each epoch, the metrics (see "Metrics" section) are evaluated for the validation split of the dataset. This should serve as guidance to improving models when solving exercise problems. The test performance can only be evaluated by the grader.

Grader: The grader's purpose is to evaluate a submission archive corresponding to one experiment run on the test data. This is required to report scores of solutions to the exercise problems.

To register, navigate to http://dlad-codalab.com/accounts/signup/ and create a shared

team account with the username `dlad22_teamXX`, where XX is your team number (with a leading zero if ≤ 9). Please share the credentials with the team members. Next, follow the grader URL (see Quick Links), and click `"Participate" -> "Register"`.

To get a submission graded, navigate to `"Participate" -> "Submit / View Results"`, enter the W&B run name of the form `GXX_XX-XX_<run_name>_XXXXX` followed by some comment (e.g., "added ASPP") into the "description" field, then click on the large `"Submit"` button, and wait to get the submission archive uploaded. Codalab begins file upload immediately after it was selected in the system dialogue, and does not indicate the upload progress, so give it a minute to upload. After the upload has finished, the status will change to indicate grading has been scheduled. You may need to refresh the page with `F5` to see submission status updates after that. If you are satisfied with the scores, you can push a submission to the leaderboard by clicking the corresponding button. Each team can make a total of 20 submissions to the grader, at most five submissions per day.

W&B sheet: For traceability, we require you to annotate and export your W&B experiment table. To annotate a run, please go to the "Table" tab of the project on W&B and add an annotation in the column "Notes". The annotation should clearly indicate where the run was used in the final report (e.g. "Row 1 in Tab. 1: Adam with LR 1e-4"). If you did not use a run in the report, please indicate that by leaving the cell empty. You can export the W&B table using the download icon in the upper right and choose "CSV Export". Please include this csv file in the final submission (see "Final submission" Section). All submission files of the runs used in the final report are considered part of the report and may not be deleted from AWS S3 until you receive your final course grade. We may access these files throughout the course.

Grader score sheet: As of task 1.2, we require you to upload the run with the best validation multitask metric to the Grader (see "Grader" Section) in order to obtain the scores on the test set. You are not required to report the Grader scores in the report, but you have to enter them in the grader score sheet. You can find the grader score sheet in the solution template under `doc/dlad22_teamXX_ex2_scores.csv`. This file will be part of your final submission.

Report: The report should be prepared as a PDF document. We recommend using Overleaf for typesetting in LaTeX, but any text editor capable of exporting into PDF should do. There is no page limit, but please avoid lengthy and redundant descriptions.

Final submission: The final submission must be sent by each team in a file named `dlad22_teamXX_ex2_submission.zip`, where `XX` is your team number, to Lukas Hoyer (lhoyer@vision.ee.ethz.ch) and Dengxin Dai (dai@vision.ee.ethz.ch) by 23:59 CET 13-05-2022 with subject `"[DLAD22] EX2 final submission: teamXX"`. The zip archive should contain the PDF report `dlad22_teamXX_ex2_report.pdf`, the grader score sheet `dlad22_teamXX_ex2_scores.csv`, the W&B sheet `dlad22_teamXX_ex2_wandb.csv`, and the solution code `dlad22_teamXX_ex2_code.zip`.

Important: Sharing the code, dataset, configurations, and run artifacts (W&B) with anyone except your team mate is not allowed at any time (even after the end of the course). To use versioning (GitHub or GitLab), make sure to use a private repository. Do not change the visibility of W&B project to public.

Project evaluation:

The project is evaluated based on the following criteria:

- For each problem statement and question, the report contains an accurate and complete description of your solution (e.g. how the configuration was altered or how a certain module was implemented).
- For problems that required code changes on top of the template, the report contains code snippets[1] of the relevant changes (relevant as of task 1.3).

[1] Recommended: https://www.overleaf.com/learn/latex/Code_Highlighting_with_minted

— The solution code is complete and correct.
— For each question (e.g. "How does SGD compare to Adam?"), the team has run one of a few experiments to base their answers on.
— The results of the experiments are presented clearly and concisely. For instance, you can use tables, diagrams, and example predictions. The means of presentation should be as detailed as necessary to discuss all relevant aspects but still as concise as possible. For example, if you only discuss the performance at the end of training, you do not need the performance curve over the course of the training, but can report the metrics in a table. Do not mention model performances only in the text without including another mean of presentation (e.g. table or diagram).
— All relevant observations from the experiments are described and possible reasons are discussed.
— All questions within a task are answered completely and correctly.
— The grader scores submitted in the score sheet are on a par with the expected reference score.
— The experiments are traceable (see "W&B sheet" section).

The total number of exercise points will be communicated back before the exam. Review of graded hand-ins will happen at the exam review session upon request.

Problem 1. Joint architecture

(3.5+3+5=11.5 points)

Your starting point is a DeepLab model [2, 3, 4, 6] that consists of a ResNet-like encoder [8], an ASPP module, and a decoder with skip connection. The template code functions properly and can be trained straight away; however, the baseline performance will be poor: 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.

0. Baseline (0 pts): Reproduce the baseline (no changes to the code are required) and submit it to the Grader (CodaLab). The approximate graded performance of the `submission.zip` should be as follows: `multitask`: 40.5, `semseg`: 67.5, `depth`: 27.1. This is just a sanity check for you. There is no need to include this task in the report.

1. Hyper-parameter tuning (3.5 pts): As a first step, you need to familiarize yourself with the hyperparameters. The file `mtl/utils/config.py` describes hyperparameters, which

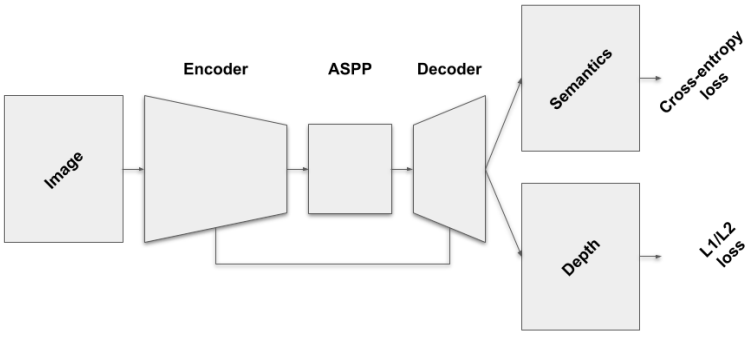

Figure 1: Joint architecture

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 on the final result in order to get a better understanding of the codebase. More specifically, you should investigate the following options and report informative conclusions about your findings.

(a) *Optimizer* and *learning rate*: What is the best choice for the optimizer (SGD vs. Adam) and the learning rate for this task? You can use a base of 10 for the learning rates (e.g. 0.01, 0.001, ...). Comment on the optimal learning rate for both SGD and Adam. How does the learning rate affect the learning process (remember to include an appropriate visualization as basis for your answer)? How does using SGD compare to Adam? From now on, we recommend using Adam with a learning rate of 1×10^{-4}.

(b) *Batch size*: How does the batch size affect the performance metrics of the multitask network? When changing the batch size, the number of steps per epoch will decrease proportionally. To alleviate this effect, we recommend changing the number of epochs proportionally to the batch size. Which other practically relevant aspects besides the task metrics does the batch size affect and how? For the following tasks, we recommend keeping the batch size 4.

(c) *Task weighting*: When multiple tasks are learned together, the individual losses of each task are usually part of a weighted sum for the total loss, which is used for update of the network parameters. What is the effect of increasing the loss weight for one task while decreasing the weight for the other task? How should the task weights be chosen and why? For the following tasks, we recommend keeping the weighting of 0.5:0.5.

2. Hardcoded hyperparameters (3 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) Encoder initialization: How are the weights of the encoder network initialized in the solution template? What happens when you switch the `pretrained` flag of the encoder? Please answer these two previous questions precisely. How do both variants compare performance-wise and what might be the reason for that? Make sure to persist the option leading to the improvement before proceeding to the next questions.

(b) Dilated convolutions: 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 [6]. 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. 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? Also, include a comparison of example predictions from W&B into your discussion. Use the model with dilation as a reference when reporting the effects of ASPP and Skip Connection.

3. *ASPP and skip connections* (5 pts): Next task is to implement the ASPP module [4, 6] along with skip connections to the decoder, whose design details are provided in the referenced papers. You are already given the skeleton of the *ASPP* class, 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 last missing part is the decoding stage with skip connection as done in [6]. 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. Further important information about the configuration of the layers can be found in [6]. The code parts which require work are marked with `TODO` annotations. What is the intention of adding ASPP and skip connections? How does the

model performance change with ASPP and skip connections functioning? Also, include a comparison of example predictions from W&B into you discussion. Do not forget to describe your solution and to provide the code snippets in your report as mentioned in the section "Project evaluation".

Roughly expected validation performance: `multitask`: ≥ 65 , `semseg`: ≥ 84, `depth`: ≤ 20

If your solution performance is worse than the provided reference values, that might be a sign to look deeper into the current problem before going to the next one.

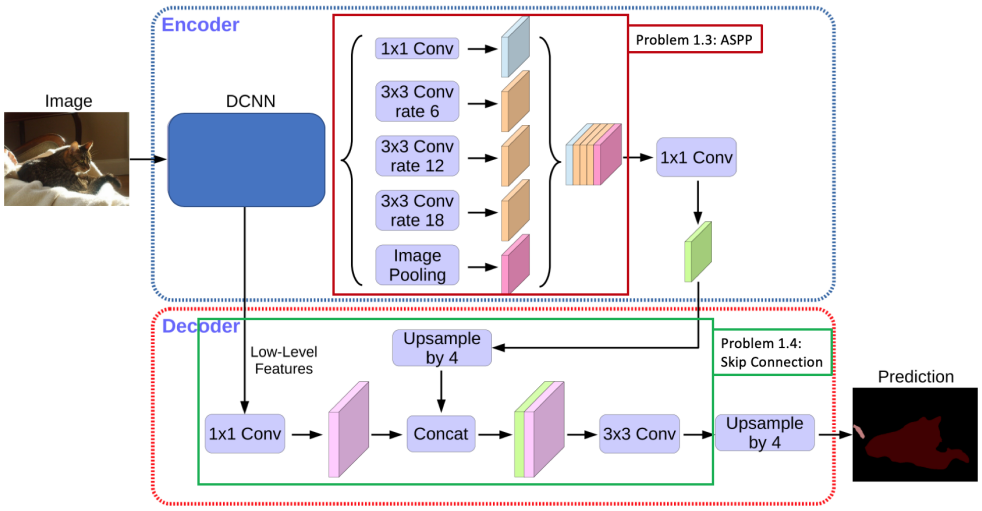

Figure 2: ASPP module and Skip Connection

Problem 2. Branched architecture (5 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 [13, 15], where a common encoder is used for both tasks, but task-specific ASPP modules and decoders are implemented for semantic segmentation and monocular depth estimation, respectively. Figure 3 gives an overview of this architecture.

As part of this problem, you are asked to implement this branched architecture using the same

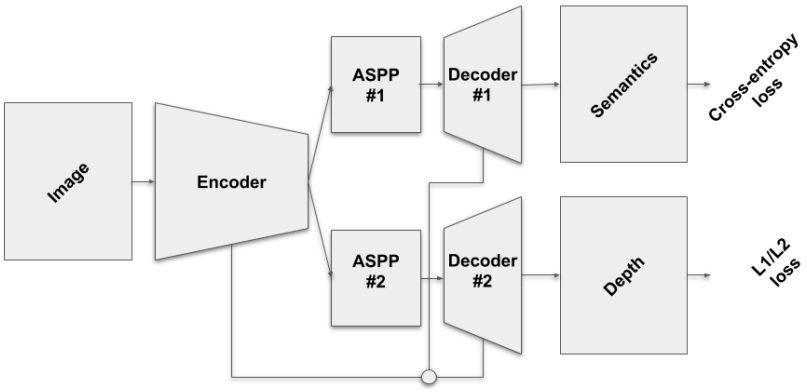

Figure 3: Branched architecture

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 GPU memory consumption, the number of network parameters, and the training time? Please also explain possible reasons for your observations.

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 (4.5 points)

Building upon a branched MTL architecture with a shared encoder followed by task-specific operations, recent works [19, 21, 16] 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 to every task.

As part of this problem, you are asked to implement the aforementioned architecture. Note that the `SelfAttention` class is already implemented for you. How does the distillation procedure compare to the branched architecture in the previous problem w.r.t. the performance, training time, and GPU memory consumption? What are possible reasons for that? When implementing the final decoder modules (`Decoder #3` & `#4`), you can adopt the design of the initial ones (`Decoder #1` & `#2`) before adding skip connections.

Similarly to the branched architecture, put a new model into a separate file and hook it up to the training code in two places.

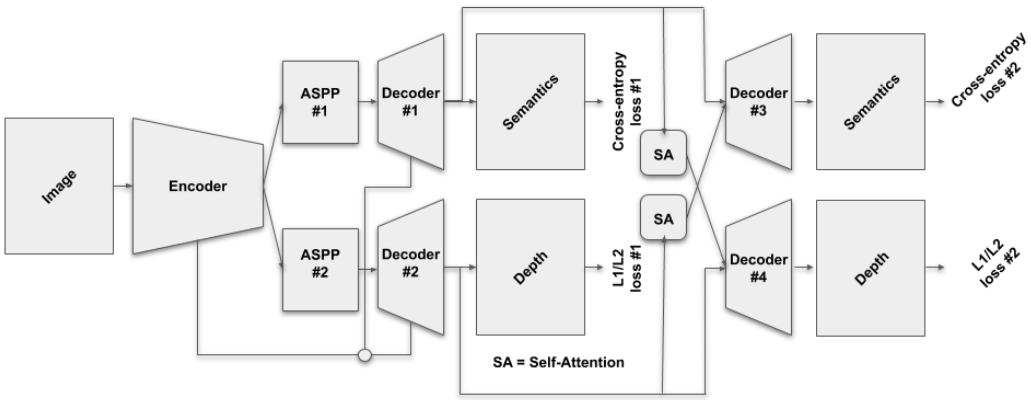

Figure 4: Branched architecture with task distillation

Problem 4. Open Challenge (4 points)

In this task, you can try your own ideas to improve MTL for semantic segmentation and depth estimation.

This task will be graded based on the following criteria:

- Your idea significantly improves the MTL performance over a baseline network without that change. The comparison of your improved model and the baseline has to be fair in the sense that no confounders are introduced. For example, if you train your new model with a higher batch size or more iterations, you have to do the same for the baseline. If your idea fails to improve the MTL performance and you do not have time to try another one, please report it nevertheless. You can still earn points based on the other grading aspects.
- Your idea requires to modify the source code of the framework. Mere changes of configuration flags will not count as sufficient contribution. Please include the relevant code in the report. If you have many small changes across different files, you can also provide the color-coded (git) diff[2].
- An in-depth analysis of the chosen approach is provided. If your idea consists of multiple components, please ablate them to show the effect of each. If your idea introduces hyperparameters, please study their influence. You should discuss possible reasons why your idea works. These should be backed-up with an experimental analysis. It should go beyond just discussing the multitask metric.
- Related works of your idea are discussed. What are the similarities and the differences?

Below are a few ideas that you can use as a starting point. You can choose which to try, pick an idea from another paper, or come up with your own idea. Please note that we do not guarantee that all ideas below give a significant improvement of the MTL performance.

- Test-time augmentation methods aim to improve the prediction quality from a trained model by aggregating the predictions across transformed versions of a test input. One of the most widely used techniques in test-time augmentation is multi-scale testing, which is adopted in Deeplab models [3], and can date back to [5]. The basic implementation can be summarized as following: multi-scale versions of the same test image are fed to the trained model. The predictions are then aggregated (average or maximum response for each position) across scales to make the final predictions. The choice of scales can be different for different vision tasks. Can this popular method also boost performance for MTL? Are there better aggregation approaches? Will additional augmentation methods further improve test-time augmentation performance? Note that it is not necessary to train a model for this task. You can use one of the previously trained checkpoints.

- 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, it 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 a normal distribution, which leads to excessive concentration of values around the lower bound of the data range (check W&B "Histograms"). Can you think of a more balanced unit of measurement (or a function) to use for the task of depth regression? If yes, augment the depth normalization process to use this new unit, such that the model will learn to predict values on the normalized

[2] https://tex.stackexchange.com/questions/105995/is-there-a-ready-solution-to-typeset-a-diff-file

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`. Also, is there a better loss function for depth estimation?

— Depth estimation can be formulated as a classification problem of depth ranges / bins as done in DORN [7] or AdaBins [1]. Could this approach be helpful for MTL?

— The `SqueezeAndExcitation` module that implements the Squeeze-And-Excitation mechanism [9] is provided in the code package. Check the paper and possibly include this mechanism in your model too [12].

— The DeepLab architecture utilizes just one skip connection from the encoder at 4x scale. In the code provided, the encoder output is a feature pyramid with every scale present. Would it make sense to add further skip connections?

— Transformers have gained increasing attention in computer vision in the last few years. More specifically, several Transformers for dense prediction were proposed [18, 17, 14, 20]. Can you adapt their concepts for MTL?

References

[1] Bhat, S.F., Alhashim, I., Wonka, P.: Adabins: Depth estimation using adaptive bins. In: IEEE Conf. Comput. Vis. Pattern Recog. pp. 4009–4018 (2021)

[2] 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)

[3] 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 Trans. Pattern Anal. Mach. Intell. **40**(4), 834–848 (2017)

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

[5] Chen, L.C., Yang, Y., Wang, J., Xu, W., Yuille, A.L.: Attention to scale: Scale-aware semantic image segmentation. In: IEEE Conf. Comput. Vis. Pattern Recog. pp. 3640–3649 (2016)

[6] Chen, L.C., Zhu, Y., Papandreou, G., Schroff, F., Adam, H.: Encoder-decoder with atrous separable convolution for semantic image segmentation. In: Eur. Conf. Comput. Vis. pp. 801–818 (2018)

[7] Fu, H., Gong, M., Wang, C., Batmanghelich, K., Tao, D.: Deep ordinal regression network for monocular depth estimation. In: IEEE Conf. Comput. Vis. Pattern Recog. pp. 2002–2011 (2018)

[8] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: IEEE Conf. Comput. Vis. Pattern Recog. pp. 770–778 (2016)

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

[10] Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Adv. Neural Inform. Process. Syst. pp. 1097–1105 (2012)

[11] Long, J., Shelhamer, E., Darrell, T.: Fully convolutional networks for semantic segmentation. In: IEEE Conf. Comput. Vis. Pattern Recog. pp. 3431–3440 (2015)

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

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

[14] Ranftl, R., Bochkovskiy, A., Koltun, V.: Vision transformers for dense prediction. In: Int. Conf. Comput. Vis. pp. 12179–12188 (2021)

[15] 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)

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

[17] Wang, W., Xie, E., Li, X., Fan, D.P., Song, K., Liang, D., Lu, T., Luo, P., Shao, L.: Pyramid vision transformer: A versatile backbone for dense prediction without convolutions. In: Int. Conf. Comput. Vis. pp. 568–578 (2021)

[18] Xie, E., Wang, W., Yu, Z., Anandkumar, A., Alvarez, J.M., Luo, P.: SegFormer: Simple and Efficient Design for Semantic Segmentation with Transformers. In: Adv. Neural Inform. Process. Syst. (2021)

[19] 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)

[20] Yuan, Y., Fu, R., Huang, L., Lin, W., Zhang, C., Xilin, C., Wang, J.: Hrformer: High-resolution vision transformer for dense predict. In: Adv. Neural Inform. Process. Syst. (2021)

[21] Zhang, Z., Cui, Z., Xu, C., Yan, Y., Sebe, N., Yang, J.: Pattern-affinitive propagation across depth, surface normal and semantic segmentation. In: IEEE Conf. Comput. Vis. Pattern Recog. pp. 4106–4115 (2019)