CVL, ETH Zurich Deep Learning for Autonomous Driving

Project 2: Multi-task learning for semantics and depth

20% of the exam grade 26-03-2021 – 13-05-2021

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
→ Latest version of the AWS setup document
* 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) [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 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. 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. The suggested workflow is to ensure the baseline trains normally and produces the expected results before (or in parallel with) solving the exercise problems.

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) to (1) participate in the leaderboard and (2) obtain the test split metrics for the final report (see "Final report hand-in" section below).

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. Extra supervision with other datasets or the use of pretrained weights (except ImageNet weights) is not allowed.

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/grader` in W&B.

W&B: This year we changed the code template to use 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 Final report hand-in 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 "feature1_value1_feature2_value2") by changing the value of the `--name` flag in `aws_train.py` script. This is not necessary, however, as long as one spends a bit of time 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. Thus, convoluted naming of experiments is not required in principle.

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. The grader is also used to rank teams on the common leaderboard, to introduce a healthy dose of competition, and distribute bonus points based on team ranking.

The grader is hosted by CodaLab, a free service for ML challenges. To register, navigate to `https://competitions.codalab.org/` and create individual accounts for each team member. Next, follow the grader URL (see Quick Links), and click `"Participate" -> "Register"`. Then send an email to Anton Obukhov (anton.obukhov@vision.ee.ethz.ch) with the subject `"DLAD 2021 EX2 user: YOUR_CODALAB_USERNAME"`, with your real name and legi in the body. After all team members are approved, you need to form teams using the "Teams" tab functionality. You should choose the same team member as decided previously for the shared AWS accounts. 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"`, 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.

Solving the problems: Each question (e.g. "How does SGD compare to Adam?") assumes your team running one or a few experiments and basing your solution on validation scores of the runs in consideration. Make sure to mention run names used as a basis for each answer. Choose one run that produced the best validation score for each question, and report its grader scores. Use the full run names (`GXX_XX-XX_<run_name>_XXXXX`) to refer to individual runs throughout the report, whether you uploaded them to the grader or not.

All submission files of the runs reported in the final report are considered part of the report and should be kept backed up (e.g., in S3, but you can also download them locally) until you receive your final course grade. We may request these files to be shared with us throughout the course.

Final report hand-in: 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. 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 W&B and relevant code snippets (changes made on top of the template).

Report evaluation criterions:

- code correctness of each problem's solution;
- each problem's grader score on par with the provided reference score;
- clarity and delivery of the solution.

Final reports must be sent by each team in a file named `dlad_ex2_report_YOUR_TEAM_NAME.zip` to Anton Obukhov (anton.obukhov@vision.ee.ethz.ch) and Dengxin Dai (dai@vision.ee.ethz.ch) by 23:59 CET 13-05-2021 with subject `"DLAD 2021 EX2 final report: YOUR_TEAM_NAME"`. The zip archive should contain the PDF report and the solution code giving the best grader score (can be downloaded and extracted from the leaderboard).

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

Extra notes:

— Without code modification, the `submission.zip` will be graded approximately as follows: `grader`: 40.5, `semseg`: 67.5, `depth`: 27.1;

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

— Early stopping after a couple of epochs of observing W&B scores can be used to reject the poor choice of some hyperparameters (but not all; e.g., changes to batch size and the number of epochs require analyzing performance at the relative training progress intervals);

— 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;

— Code, dataset, configuration, and run artifacts (W&B) sharing is not allowed during and after the competition ends. To use versioning (GitHub or GitLab), make sure to use a private repository. Do not change the visibility of W&B project to the public. Erase all copies of exercise materials and W&B logs after receiving the final grade;

— Only the training split of 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;

— Each team can make a total of 20 submissions to the grader, at most five submissions per day. Use them carefully;

— For your convenience, we provide reference metrics values of the solution key after each problem.

— If your solution performance is significantly worse than the provided reference values, that might be a sign to look deeper into the current problem before going to the next one. Poor performance may also have reasons in preceding problems' solutions due to code carry-over. Thus, we suggest making your solution close to the confidence interval of the reference for each problem 1–3 before proceeding to the next problem.

Problem 1. Joint architecture

(4+2+6=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 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.

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 on 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 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.

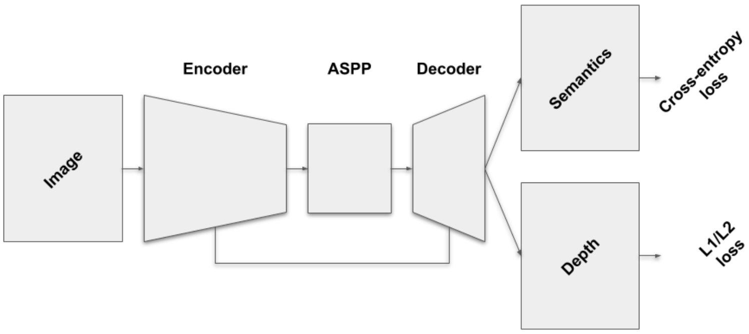

Figure 1: Joint architecture

(a) The *optimizer* and *LR* 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 of orders of magnitude smaller for Adam. Also, try different *learning rates*. How does logarithmically changing the learning rate affects the learning? You can try three different values for each optimizer as an indicative bracket.

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

(c) *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 the 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 the ImageNet classification task? What is the effect of switching this option? Make sure to persist the option leading to the improvement 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. 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 better performance.

3. *ASPP and skip connections* (6 pts): Next task is to implement the ASPP module [3, 4] 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

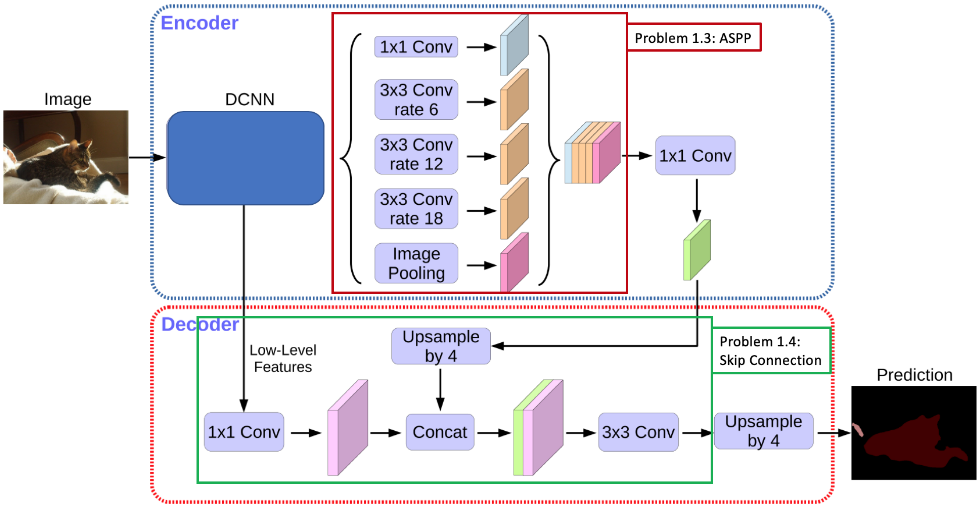

Figure 2: ASPP module and Skip Connection

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 [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. Further information about the configuration of the layers can be found in [4]. The code parts which require work are marked with `TODO` annotations. How does the model performance change with ASPP and skip connections functioning?

Expected performance: `grader`: 62.3 ± 4.0, `semseg`: 83.6 ± 2.0, `depth`: 20.5 ± 2.0

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 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 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?

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.

Expected performance: `grader`: 65.3 ± 4.0, `semseg`: 84.5 ± 2.0, `depth`: 19.2 ± 2.0

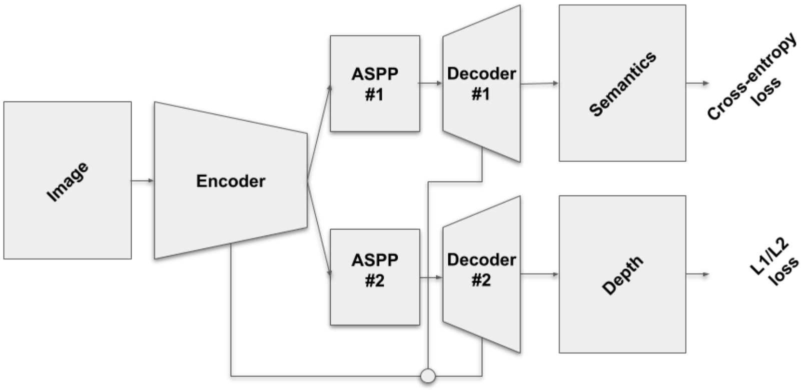

Figure 3: Branched architecture

Problem 3. Task distillation (4 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 to every task.

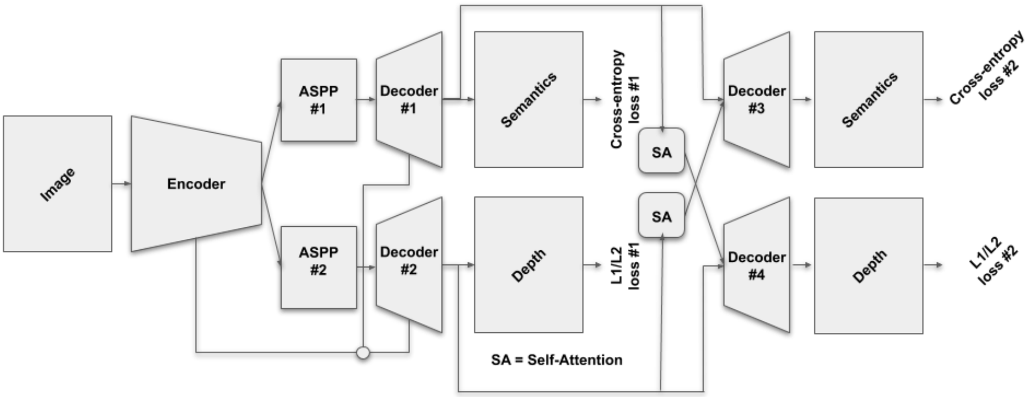

Figure 4: Branched architecture with task distillation

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`).

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

Expected performance: `grader`: 67.3 ± 4.0, `semseg`: 84.5 ± 2.0, `depth`: 17.3 ± 2.0

Problem 4. Winning the Leaderboard (3 points)

Ideas and improvements outlined in this problem may only be tried after finishing the previous problems. Please do not mix novelties into the solutions of problems 1–3.

To ensure fairness, this problem has a few more constraints on the solution and the training process:

— Only a single ResNet34 (with pretrained ImageNet weights) encoder is allowed;

— Changing dilations in the encoder is allowed;

— Skip connections from encoder stages are allowed;

— The final model for the leaderboard has to be trained in a single run within not more than 36 hours on the provided AWS account. This means, in particular, that if the AWS instance was preempted while training the model, you should start over instead of resuming from the checkpoint;

— Loading checkpoints, external weights, or datasets (except whitelisted) is not allowed in this problem – the solution should be self-contained.

All teams will be awarded additional bonus points based on the ranking according to the "Multitask" score in the leaderboard after its closure. The scores will be assigned linearly on the interval [0, 3] based on team position (rank) in the leaderboard.

Please document your ideas clearly in the final report.

Below are a few ideas that can help win the leaderboard (you can choose which to try or pick an idea from some paper):

— Recommended `batch_size` interval is between 2 and 16. Larger batches give cleaner gradients but slow down the 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 a 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, the 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, 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 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`.

The highest performance we observed (but probably not the best possible) with these and some other state-of-the-art techniques: `grader`: 78.8, `semseg`: 90.9, `depth`: 12.1

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)