COMP4901V: Deep Perception, Localization, and Planning for Autonomous Vehicles

Homework Assignment 1

Release Date: Mar. 03, 2023

The HKUST Academic Honor Code

Honesty and integrity are central to the academic work of HKUST. Students of the University must observe and uphold the highest standards of academic integrity and honesty in all the work they do throughout their program of study. As members of the University community, students have the responsibility to help maintain the academic reputation of HKUST in its academic endeavors.

1 Introduction

In this homework, we will have two sub-problems. First, we will implement a classification network to train and evaluate on a VehicleClassification Dataset. Then, we will make this classification network fully convolutional and jointly solve semantic segmentation and depth estimation tasks (labeling every pixel in the image). The submission deadline for this homework is **Apr. 03, 2023**. Please start as early as possible, and feel free to discuss with us if you have any questions in your design and implementation.

This assignment should be solved individually. No collaboration, sharing of solutions, or exchange of models is allowed. Please, do not directly copy existing code from anywhere other than your previous solutions, or the previous master solution. We will check assignments for duplicates. See below for more details about the homework.

2 Installation and setup

Installing python 3 Go to https://www.python.org/downloads/ to download python 3. Alternatively, you can install a python distribution such as Anaconda. Please select python 3 (not python 2). Installing the dependencies Install all dependencies using

```
python3 -m pip install -r requirements.txt
```

Note: On some systems, you might be required to use pip3 instead of pip for python 3.

If you're using conda, use `conda env create environment.yml` to create a development environment for the homework.

Manual installation of pytorch Go to https://pytorch.org/get-started/locally/ then select the stable Pytorch build, your OS, package (pip if you installed python 3 directly, conda if you installed Anaconda), python version, cuda version. Then run the provided command to have an installation. Note that cuda is not required, and you can select `cuda = None` if you don't have a GPU or don't want to do GPU training locally. We will also provide instructions and starter code for conducting remote GPU training of your models on Google Colab for free.

Manual installation of the Python Imaging Library (PIL) The easiest way to install the PIL is through pip or conda using a command as follows:

```
python3 -m pip install -U Pillow
```

3 Design and Train an Image Classification Network

In this part, we will design and train a convolutional neural network to classify images from a VehicleClassification dataset. The samples of the dataset are a part of one Kaggle competition (*i.e.*, TAU Vehicle Type Recognition Competition). The dataset contains six categories (*i.e.*, Bicycle, Car, Taxi, Bus, Truck, Van), as shown in Figure 1. These are very commonly observed categories on our roads. Correct recognition of these object classes is critical for a self-driving system. You may download your training and validation sets from here. This assignment should be solved individually. No collaboration, sharing of solutions, or exchange of models is allowed. More details about this assignment part are given below.

3.1 Data Loader (15 points)

As a first step, we will need to implement a data loader for the VehicleClassification dataset. Complete the `__init__`, `__len__`, and the `__getitem__` of the VehicleClassification class in the `utils.py`.

The `__len__` function should return the size of the dataset.

The `__getitem__` function should return a tuple of image and label. The image should be a torch.Tensor with possibly varying resolutions for different images, and the label should be an integer. There are 6 classes of objects. You may assign integer label for the 6 classes as follows: Bicycle 0, Car 1, Taxi 2, Bus 3, Truck 4, Van 5. Some examples of the dataset can be seen in Figure 1.

Figure 1: Examples of the VehicleClassification dataset.

You may use several relevant operations for the implementation of the DataLoader, including `torchvision.transforms.ToTensor`, `torch.utils.data.Dataset`, and `PIL.Image.open`.

3.2 CNN Model and Non-Linear Classifier (30 points)

Implement the ResNet50 CNN model and a linear Classifier for a class `CNNClassifier` in `models.py`. You will use ResNet50 as your backbone. The ResNet50 can be initialized from an ImageNet pretrained model. The non-linear classifier will map the spatial feature map from the last convolutional layer of ResNet50 to score predictions for classification. The

non-linear classifier will be implemented based on several multi-layer perceptron (MLP) layers. The detailed implementation of it will be as follows: **(i)** First, you may need to perform a global average pooling for each channel of the final spatial feature map of the ResNet50 network, which results in a 2048-dimensional output, **(ii)** and then you use another MLP layer to aggregate the features, and output a lower-dimensional (1024) feature vector, and you would use a ReLU non-linear activation function upon the feature vector, **(iii)** and finally, you use another MLP to produce a prediction score vector for all the C(C = 6 in our case) classes. Your forward function receives a $(B, 3, H, W)$ tensor as an input, where B stands for the batch size; H and W are the height and width of the input image, respectively, and should return a (B, C) torch.Tensor (overall C classes to be recognized). Next, we will learn to implement the classification loss (*i.e.*, `SoftmaxCrossEntropyLoss`) in `models.py`. We will later use this loss to train our classifiers. You need to implement the log-likelihood of a softmax classifier, *i.e.*, $-\log\left(\frac{\exp(x_l)}{\sum_j \exp(x_j)}\right)$, where x are the logits and l is the label. You need to implement this loss function by yourselves, instead of directly using existing PyTorch functions.

3.3 Logging (15 points)

In this part, we learn how to use `tensorboard`. We created a dummy training procedure in `train_cnn.py`, and provided you with two `tb.SummaryWriter` as logging utilities. Use those summary writers to log the training loss at every iteration, the training accuracy at each epoch and the validation accuracy at each epoch. Log everything in global training steps. Here is a simple example of how to use the `SummaryWriter`.

```
import torch.utils.tensorboard as tb
logger = tb.SummaryWriter('cnn')
logger.add_scalar('train/loss', t_loss, 0)
```

3.4 Training your CNN model (20 points)

Train your model and save it as `cnn.pth`. You can reuse some of the training functionality in `train_cnn.py`. We highly recommend you incorporate the logging functionality from section 3.3 into your training routine. You will tune your CNN model, to achieve as high an accuracy as possible. This might require several tricks:

- Input normalization
- Dropout
- Data augmentations (Both geometric and color augmentations are important. Be aggressive here. Different images may have radically different lightings.)
- Weight regularization
- Early stopping
- ...

Single-Task	Vehical Classification (Accuracy)
Train	
Validation	

Table 1: Quantitative performances of the image recognition model on the VehicleClassification Dataset.

3.5 Model Evaluation (15 points)

After you finish training your model, you need to implement an evaluation script in `test_cnn.py` for you to evaluate your selected best model. We have provided a class `ConfusionMatrix` in utils.py which contains an implementation of the measurement of the classification accuracy metric. You need to report your classification performances on both your training and validation sets in a table, with the format of Table 1. The result table, together with the training curve produced from Section 3.3, needs to be put into a pdf document with the name of `cls_results.pdf` under the main folder of the homework. You also need to put a link in the same PDF, for us to download your trained CNN classification model which produces the results shown in the table. We are going to release the performance of our classifier 10 days before the deadline. You may try to further boost your performance by tuning the hyperparameters and any other possible ways. If you manage to reach or outperform our performance, you will get **10 extra credits**.

4 Dense prediction for Semantic Segmentation and Depth Estimation

In the second part of the homework, you will make your CNN fully convolutional (FCN). Instead of predicting a single output per image, you will now predict an output per pixel. Our current VehicleClassification dataset for object recognition does not support this output, we thus switch to a new dense prediction dataset, which is constructed by using partial data from Cityscapes. The constructed dataset contains 128×256 sized images, their 19 class semantic segmentation labels, and **inverse** depth labels. The images and labels are processed into `.npy` format, and you can easily load them using `Numpy`. Each image is associated with a ground-truth semantic mask and a depth map. Here, all the label maps have the same spatial resolution as the input images. You can download this dataset for your model training, validation, and testing from here.

You will first implement an FCN for only the semantic segmentation task (a single-task setting), and then you will additionally implement a depth prediction head to jointly learn semantic segmentation and depth estimation in a unified FCN framework (a multi-task setting).

4.1 Data Loader (20 points)

You will need to implement a data loader in the `Class DenseCityscapesDataset` in `utils.py`, for you to load data for training/validation/testing. The data includes both input images, and their corresponding label masks (both semantic and depth). You may

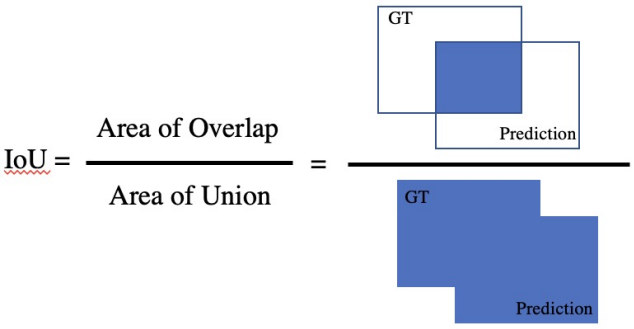

Figure 2: Illustation of calculation of the IoU metric.

need to perform transformations for your loaded data, and then return your data. Some examples from the dense Cityscapes dataset can be observed from Figure 3. Please be noted that the real depth can be calculated from the provided disparity data using the formula depth $= Bf/d$, where $B = 0.222384$ indicates the baseline of the cameras, and $f = 2273.82$ is the focal length, and d is the disparity.

4.2 FCN Design for Semantic Segmentation (Single-Task) (50 points)

Design and implement your FCN by writing the model in a class `FCN_ST` in `models.py`. Make sure to use only convolutional operators, pad all of them correctly, and match strided operators with up-convolutions. Use skip and residual connections. The backbone architecture will be still ResNet50 as used for the previous task, *i.e.*, the vehicle classification. Make sure your FCN handles an arbitrary input resolution and produces an output of the same shape as the input. Use `output_padding=1` if needed. Crop the output if it is too large. The loss function for pixel-wise segmentation can also use Softmax-CrossEntropy which is implemented in the previous vehicle classification part. You can reuse it for the classification of each pixel here. As different semantic classes are highly imbalanced, you need a weight balancing strategy for the different classes in the Softmax-CrossEntropy loss function. The weights for the different semantic classes are calculated based on the ratio of the number of pixels of each class to the overall number of pixels, which are given below:

0 road: 3.29; 1 sidewalk: 21.90; 2 building: 4.68; 3 wall: 121.32; 4 fence: 266.84; 5 pole: 117.60; 6 traffic light: 1022.23; 7 traffic sign: 205.68; 8 vegetation: 6.13; 9 terrain: 118.81; 10 sky: 35.17; 11 person: 168.36; 12 rider: 460.62; 13 car: 15.53; 14 truck: 272.62; 15 bus: 501.94; 16 train: 3536.12; 17 motorcycle: 2287.91; 18 bicycle: 140.32.

The weights of the 19 different classes are also given in `classweight.cls`. when you call the CrossEntropy loss, we should also ignore label 255 (it is for unlabeled categories and pixels) as follows:

```
CEloss = nn.CrossEntropyLoss(loss_weights, ignore_index=255)}
```

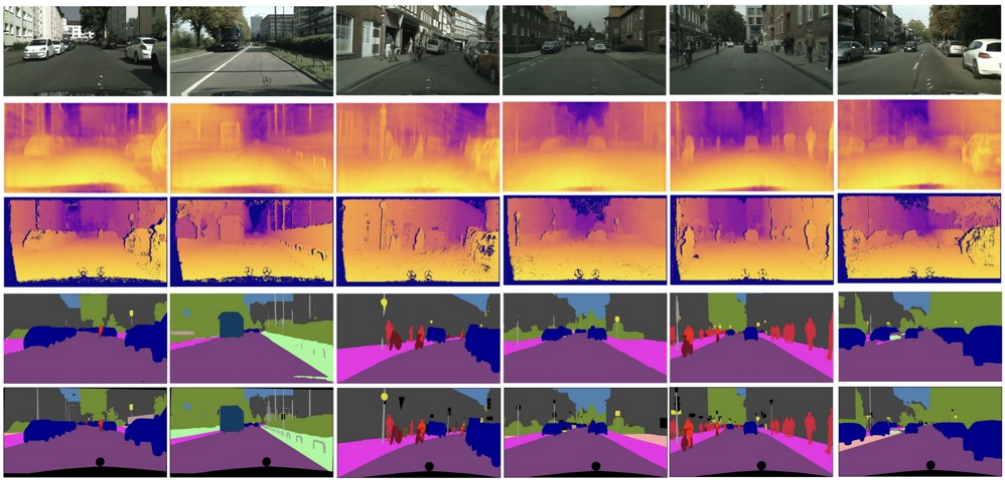

Figure 3: Qualitative examples of predicted segmentation and depth maps.

Single-Task	Segmentation	
	Accuracy	mIoU
Train		
Validation		
Test		

Table 2: Quantitative performances of segmentation of the single-task FCN model.

4.3 FCN Training (30 points)

To train your FCN you will need to modify your CNN training code a bit, and implement your training script in `train_fcn_singletask.py`. First, you need to use the Dense Prediction Dataset. To have a better performance, data augmentation can be performed by transforming the dense data. Most standard data augmentation in torchvision does not directly apply to dense labeling tasks. We thus provide you with a smaller subset of useful augmentations that properly work with a pair of image and label in `dense_transforms.py`. You may use it for data augmentation on the fly for your training. Besides, you will also need to use the same bag of tricks as for classification described in Section 3.4 to make the FCN train well for semantic segmentation.

4.4 Evaluation of Segmentation Performance (25 points)

After the training is finished, we need to evaluate the performance of the model quantitatively. Since the training set has a large class imbalance, it is easy to cheat in a pixel-wise accuracy metric. We additionally measure the Intersection-over-Union evaluation metric. This is a standard semantic segmentation metric that penalizes largely imbalanced predictions and evaluates the performance in terms of completeness. The calculation is based on the prediction and the ground-truth masks. An illustration of the calculation is illustrated in Figure 2. You can compute the IoU and accuracy using the `ConfusionMatrix` class in `utils.py`. You can test your model and report your performance on different data subsets in a table with a format shown in Table 2.

Multi-Task	Segmentation		Depth			
	Accuracy	mIoU	rel	$\delta < 1.25$	$\delta < 1.25^2$	$\delta < 1.25^3$
Train						
Validation						
Test						

Table 3: Quantitative performances of segmentation and depth estimation results of the multi-task FCN model.

4.5 Additional Prediction Head for Depth Estimation (Multi-Task) (25 points)

In this part, you will need to design and implement an additional prediction head for depth estimation, as shown in Figure 4. You will reuse the architecture that you already implemented for semantic segmentation. You add the designed depth prediction head from the Backbone (*i.e.*, ResNet50). In this way, the segmentation and the depth prediction heads share the spatial feature maps produced from the ResNet50 backbone (see Figure 4). Depth estimation is also a dense prediction task, in which we need to predict a continuous depth value for each pixel, which is different from the discrete classification problem in semantic segmentation. We need to implement an upsampling depth prediction head as we do for the semantic segmentation task, to output a **one-channel** depth map that has the same resolution as the input image.

4.6 Multi-task Training (20 points)

When the architecture of joint semantic segmentation and depth estimation is built, we need to optimize it with loss functions for semantic segmentation (softmax-CrossEntropy for discrete classification) and depth estimation (L1 loss for continuous regression). Here, you are allowed to directly use `torch.nn.L1Loss` function to optimize the depth estimation, and `torch.nn.CrossEntropyLoss` function to optimize the semantic segmentation.

We need to initialize the model with the pretrained parameters from the single-task semantic segmentation model we have previously obtained, instead of training the multi-task network from scratch. This initialization can effectively speed up the training and convergence of the multi-task FCN model. The training of the multi-task network will be implemented in `train_fcnunderline multitask.py`.

4.7 Performance Evaluation (20 points)

After you finish training your FCN multi-task model, you can evaluate your model quantitatively and qualitatively. You need to implement your testing code in `test_fcn_multitask.py`. Please note that you need to mask invalid depth values (*i.e.*, zero depth values) in GroundTruth (GT) before calculating the metrics, and the GT depth should be also calculated from the disparity as in the training stage for the evaluation. We use four different evaluation metrics for the depth estimation, including mean relative error (rel): $\frac{1}{N}\sum_i \frac{|d_i - d_i^*|}{d_i^*}$, and and accuracy with threshold t percentage (%) of d_i^* subject to max $max(\frac{d_i^*}{d_i}, \frac{d_i}{d_i^*}) < t (t \in [1.25, 1.25^2, 1.25^3])$. The evaluation code is provided in a class DepthError in `utils.py`. You need to call it for

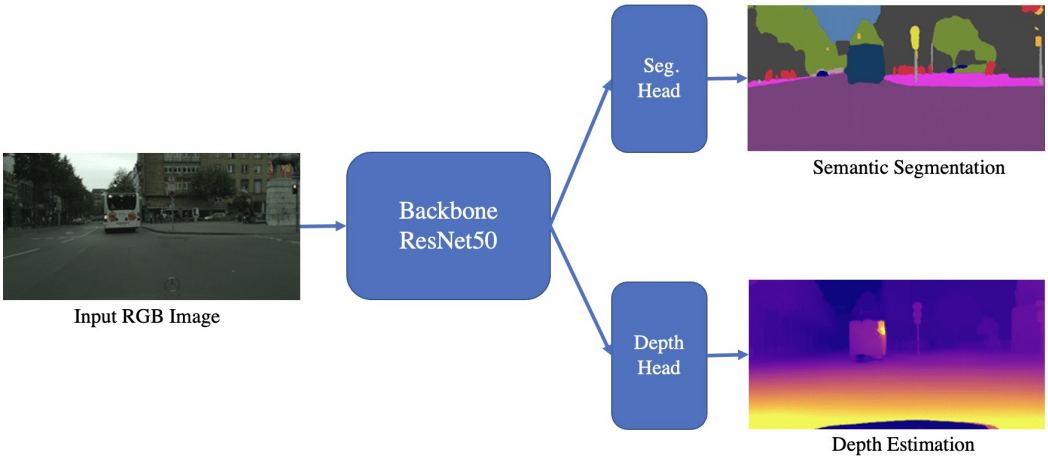

Figure 4: Illustration of the CNN framework for joint semantic segmentation and depth estimation.

your quantitative evaluation and report your performance as shown in a table with a format showing in Table 3. You could also qualitatively visualize your predicted semantic and depth maps by implementing `Class DenseVisualization` in `utils.py`. You may randomly sample 6 testing images, and visualize them in the format as shown in Figure 3. The first row is the input testing images; the second and third rows are the predicted and ground-truth depth maps; the fourth and fifth rows are the predicted and ground-truth semantic maps. You could choose your own colormaps to visualize your results. Finally, you will put Table 2, Table 3 and Figure 3 in the same PDF, with a file name of `densepred_results.pdf` under the main folder of the homework. You also need to put a link in the same PDF, for us to download your trained (single-task and multi-task) FCN models which produce the results shown in the tables.

4.8 Extra Bonus Points

Your goal is to make your model achieve the best performance on your testing data. You should not use the provided testing data to tune your model. Instead, you could tune your model based on your training and validation data sets. We will also test your provided models on our own testing data, which has a similar distribution to the testing data provided to you but with additional samples added. You'd better not overfit your testing data since it would very possibly cause poor performance on our testing data. The top 30% performances of the provided models on our testing data will receive **extra 30 points**.

5 Submission

Once you finished the homework assignment, create a submission bundle using

```
python3 bundle.py homework [YOUR UST ID],
```

and then submit the zip file on canvas. Please remember to put your pdf files that contain your result tables and figures under the folder `homework/`. Please note that the maximum

file size for your submission is 20MB.

6 Running your assignment on google colab

You might need a GPU to train your models. You can get a free one on google colab. We provide you with an ipython notebook that can get you started on colab for each homework.

If you've never used colab before, go through colab notebook (tutorial) When you're comfortable with the workflow, feel free to use colab notebook (shortened)

Follow the instructions below to use it.

- Go to http://colab.research.google.com/. Sign in to your Google account.
- Select the upload tab then select the .ipynb file.
- Follow the instructions on the homework notebook to upload code and data.