LabMLISP: Lab Course Machine Learning in Signal Processing

Exercise V: Neural network

Prof. Veniamin Morgenshtern

Author: Kamal Nambiar, Christoph Hertle

The goal of this exercise is to build a neural network that detects road lane marking in the given input image. Upon completion of this exercise, the notebooks that are provided to you must run without any errors. This is to ensure that you have correctly implemented all components of the network. For this exercise, it is all right in case your network does not achieve good results. The network performance is largely dependent on the hyperparameters that you choose while training. We shall focus on finding hyperparameters that will result in better network performance in the next exercise.

Exercise Materials

Complete the missing code using the hints provided in this documents and the code comments. To access the lab materials for this task, execute:

```
$ cd labmlisp

$ cp -r ~/SHARED/DATA/mlisp-lab/ps5-neural-network/* .
```

The file structure of the network is given in fig. 1. Make sure that you have the files listed in the figure in your working directory, i.e., **labmlisp**, where you had previously copied the files from simulator exercise. In case any file is missing, please inform the lab tutors. You also may find some extra files that are not listed in the figure. You can safely ignore those extra files as you will not be directly working on those files in this task. But please don't delete those files!

In the working directory, you will find the five jupyter notebooks: **02-test-image-dataset.ipynb**, **03-test-augmentation.ipynb**, **04-test-network.ipynb**, **05-train.ipynb**, and **06-predict.ipynb** that can be used to test your implementation.

Dataloader

In the previous exercise (PyTorch tutorial), we used the MNIST dataloader to provide input to the network. In this exercise, you are required to implement a custom dataloader that will provide the images from the road simulator as input to the network [1, 2].

Fill in the missing code in **image_dataset.py** in the data folder. One of the tasks here is to read images from folders. Pay attention to details like the datatype used to store the image and the size/shape of the image. Once you are finished with the **image_dataset.py**, run the **02-test-image-dataset.ipynb** notebook to test your implementation.

```
labmlisp/
├─data/
│ ├─input_data/
│ ├─network_data/
│ ├─output_data/
│ ├─sky_img/
│ ├─tensor_board_logs/
│ ├─__init__.py
│ ├─image_dataset.py
│ └─video_dataset.py
├─network/
│ ├─__init__.py
│ └─unet.py
├─simulator/
├─tests/
├─utils/
│ ├─__init__.py
│ ├─augmentation.py
│ ├─experiment.py
│ ├─file_handling.py
│ ├─metrics.py
│ ├─tensor_board_helper.py
│ └─visualization.py
├─config.json
├─model.py
├─pipelines.py
├─sim_config.json
├─01-test-simulator.ipynb
├─02-test-image-dataset.ipynb
├─03-test-augmentation.ipynb
├─04-test-network.ipynb
├─05-train.ipynb
└─06-predict.ipynb
```

Figure 1: The file structure of the project. The folders are indicated in bold-face font and the files that needs to be completed as part of this exercise are indicated in blue.

Data Augmentation

A sufficiently large and diverse training dataset is critical for training neural network. Acquiring data is usually expensive and time-consuming. Therefore one might consider enhancing the size and diversity of a data set by augmenting the data.

In the folder utils we already provided a starter code in the file **augmentation.py**. Your task is to implement data augmentation by completing the code in **augmentation.py**. We will use the following Python packages such as random, Numpy, PIL, to implement the augmentation functions.

One way to augment data in the form of pictures is to apply geometric transformations to the images. These augmentation functions are easy to implement but can prove to be quite effective nonetheless.

We have implemented one example each using PIL and Numpy. You may refer these examples while doing the exercise. It is important to note that the output must always be Numpy array and the size and datatype must be consistent with the input. You are required to implement the following augmentations operations:

- Rotate
- GaussianNoise
- ColRec - Colored Rectangles
- ZoomIn
- ZoomOut

Use the **03-test-augmentation.ipynb** notebook to test the augmentation functions you implemented above. Reference output for each augmentation function is provided in the notebook.

Network Architecture

In this task, we shall use a variant of the U-Net proposed in [3]. The U-Net is a very popular network architecture for semantic segmentation tasks. In the classification task from the PyTorch tutorial, the model predicted a single class for each image in the dataset. However, in semantic segmentation, our goal is to train a model that predicts a class for each pixel in the given image. The road lane detection task has two classes: Not-Lane (class 0) and Lane (class 1).

It is clear from fig. 2 that there are many repeating operations in the network. Hence, we shall implement the architecture in a modular manner, so that the code written once can be reused many times. The module level description of the network is provided in table 1.

The nn.Module class was introduced earlier in the PyTorch tutorial, where it was used to construct the complete network as a single module. Now, we shall use the nn.Module class to implement the modules from table 1. Recall that the `__init__` function is used to initialize (or define) various layers in each block using the PyTorch implementation of these layers and in the `forward` function,

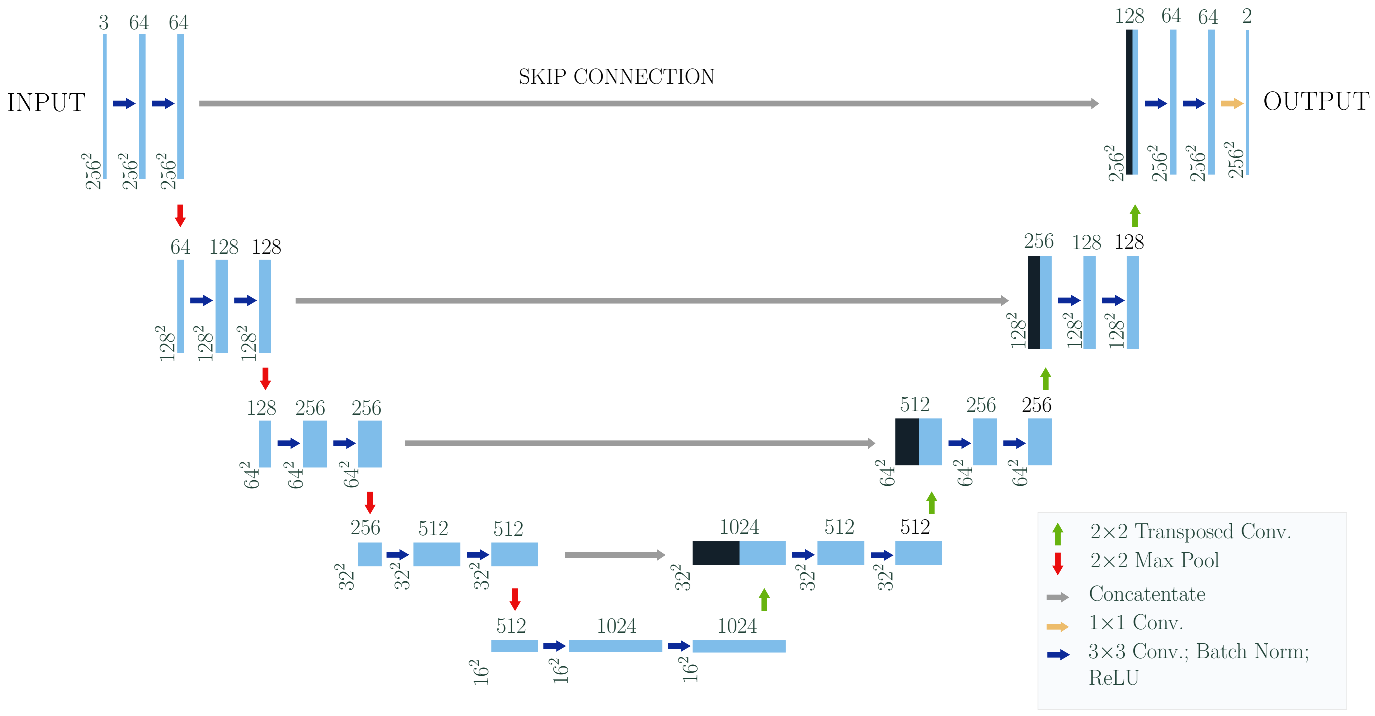


Figure 2: U-Net architecture [3]. The operations performed in the network is provided in the legend at the bottom right of the figure. The tensors after each operation is represented by the rectangles. The number on top of the rectangle indicates the number of channels (or feature maps) and the number at the bottom left indicates the spatial resolution, i.e, $256^2 \equiv 256 \times 256$ pixels.

Module	**Operations**	**Parameters**
Double Convolution	3×3 Convolution	filters:k, padding:1
	Batch Normalization	
	ReLU	
	3×3 Convolution	filters:k, padding:1
	Batch Normalization	
	ReLU	
Encoder	2×2 Maxpool	stride:2
	Double Convolution	
Decoder	2×2 Transposed Convolution	stride:2
	Concatenation	
	Double Convolution	
Output	1×1 Convolution	filters:2

Table 1: Module level description of the U-Net network architecture. Note that the both convolution layers in the double convolution module have the same number filters (k). However, the value of k is different for each layer of the network and this can be determined from the output tensor shapes in fig. 2.

the layers defined in the `__init__` function are called in a sequential manner to generate output from the input provided to the function.

Alternatively, you may use nn.Sequential class, to define a linear stack of layers that are connected one after another, in the `__init__` function. This will simplify the `forward` function to a single line of code. For more details, refer the documentation of PyTorch nn modules in [4]. Pay attention to the difference between the usage of torch.nn and torch.nn.functional. For example, when would you use `nn.ReLU` and `F.ReLU`?

Refer the tensor shape/size at each layer from the fig. 2 and set the input channels and number of kernels (or filters) for the layer accordingly. For parameters that are not mentioned in table 1, use the default parameter values from PyTorch documentation. The down-sampling operation in the block diagram is achieved using the maxpool layer. For the up-sampling operation, we use the transposed convolution. Please note that the parameters of transposed convolution must be set such that the number of output channels (or feature maps) is half that of the input channels. Before performing the concatenation operation, you are required to check if the two tensors have same height and width. If they are unequal, apply padding on the edges of the smaller tensor with zeros.

The final layer of the network returns a 2-channel tensor as output. The network predictions and Lane-probability for each pixel can be obtained by applying the softmax function on the network output.

Complete the implementation for the network architecture in **unet.py** from the network folder.

Training and Prediction

Metrics

In addition to the loss and accuracy, we will also use the F_1 score (or dice coefficient) to evaluate the performance of the network. In order to make these calculations, we need to first calculate the confusion matrix. Given the predicted labels and true labels for each mini-batch of the dataset, calculate the total number of true positives, false positives, true negatives and false negatives using fig. 3. These values will be stored in the form of an array and they will be incremented for each mini-batch. At the end of the epoch, the metrics for the entire dataset can be calculated with this array.

		Predicted Labels	
		1	0
True Labels	1	#True Positive	#False Negative
	0	#False Positive	#True Negative

Figure 3: Sample representation of confusion matrix. The rows and columns of the confusion matrix may be interchanged to obtain another valid representation of the confusion matrix. However, it is important that the true positive, true negative, false positive and false negative are indicated accordingly. For example, #False Negative is the total number of samples (i.e. pixels) whose true class label was 1, but was predicted as class 0 by the model.

The F_1 score is the defined as the harmonic mean of the precision and recall metrics. This can be further simplified as follows:

$$F_1 = \frac{2 \times \#\text{True Positive}}{(2 \times \#\text{True Positive }) + \#\text{False Positive} + \#\text{False Negative}}$$

Similarly, the accuracy of the model can be defined as follows:

$$\text{Accuracy} = \frac{\#\text{True Positive} + \#\text{True Negative}}{\#\text{True Positive} + \#\text{True Negative} + \#\text{False Positive} + \#\text{False Negative}}$$

Complete the missing code in **metrics.py** found in the utils folder.

Visualization

The predictions of the network will be visualized using an overlay image. The overlay image is a superimposition of the input image and the network predictions. The output needs to be created such that the pixels that predicted as Lane (class 1) by the network are red in color. Complete the implementation in **visualization.py** from the utils folder.

Model Execution

The Model class is implemented in the **model.py** file. This class handles tasks such as setting up the model parameters, training and evaluation of the model, saving and loading training models, etc. You may refer the implementation of the forward pass and backward pass from the PyTorch tutorial if required. Refer to **05-train.ipynb** as an example to see how the functions are called and make sure you understand the sequence of execution.

Recall that the data is processed as mini-batches of the dataset. After each mini-batch is processed by the network, the loss is captured in a list and this list will be used to calculate the average loss at the end of each epoch. Using the predicted labels and true labels, the confusion matrix is calculated as described in the Metrics section. The F_1 score and accuracy for the entire data is computed from the confusion matrix at the end of the epoch. The same calculation procedure is followed for train and validation steps.

The value of the F_1 score for the validation dataset after each epoch is stored in a list. After all training epochs are completed, the model from the epoch with the highest F_1 score is saved as the best model. This model will be used for making predictions in the **06-predict.ipynb**.

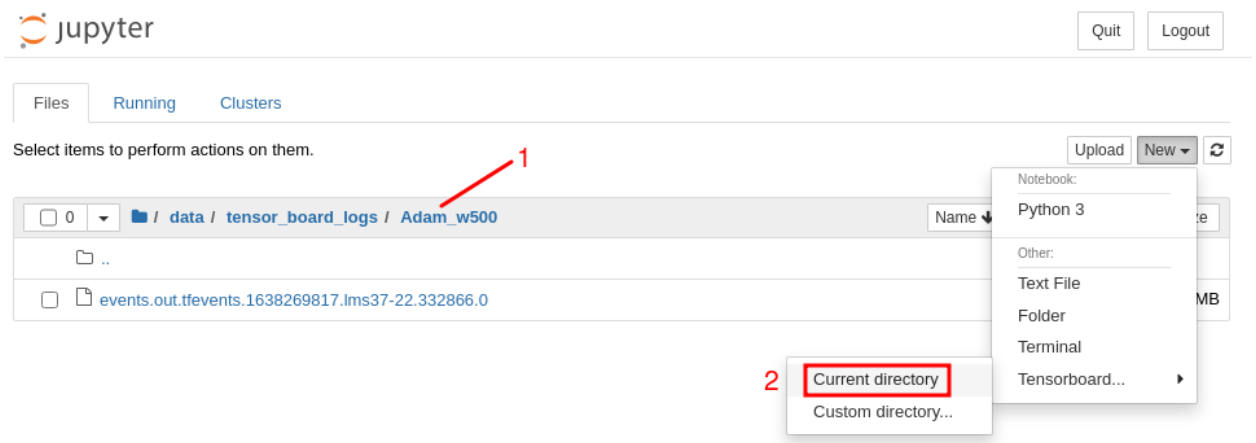
The implementation of the predict step is mostly similar to the validation step. The difference is that we don't have the true labels in case of the predict step and hence, we cannot calculate the metrics and loss as in case of the test step. Only visual evaluation is possible for the predict data.

Complete the missing code in **model.py**. This is the last step for this exercise. Upon completion, you can test your code and train a model using the **05-train.ipynb** notebook.

Accessing Tensorboard

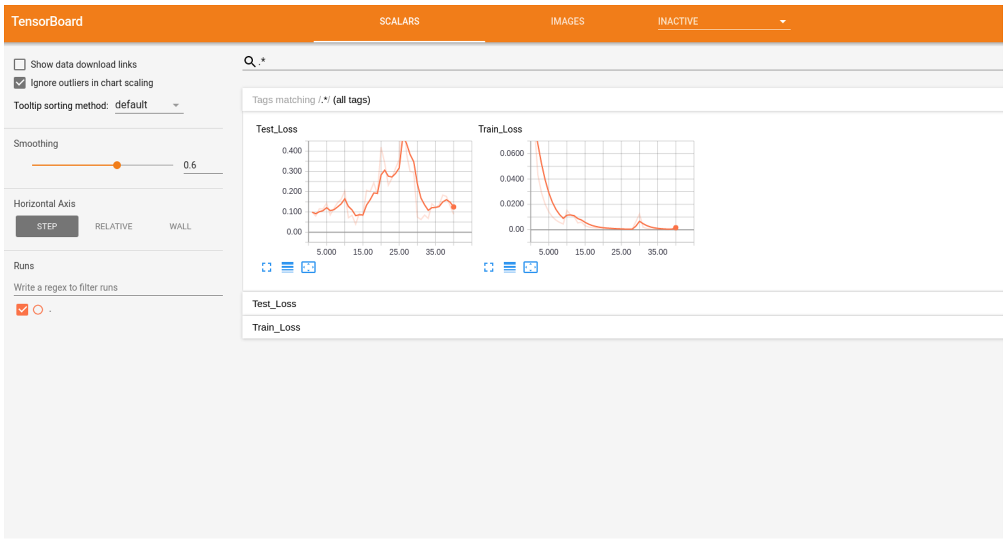
Tensorboard is used in the visualize the training process. The code required to generate the graph and images in Tensorboard is already provided and you are not required to add any additional code here. To access Tensorboard:

1. Navigate to your experiment folder in data/tensorboard_logs. For example, if your experiment folder is Adam_w500:

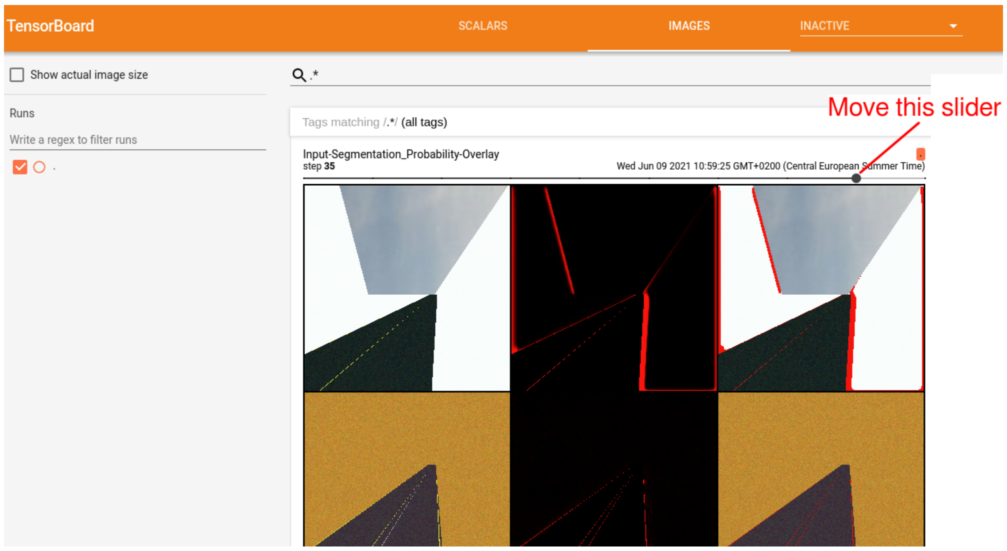


2. Click on the 'New' button on the top right corner of the window, select 'Tensorboard' from the drop down menu and finally click on 'Current directory' from the side menu.

3. The Tensorboard application will open in a new tab of the browser as shown below. On the orange colored header, we will find two options, SCALARS and IMAGES. The Scalars option shows the graph of train and test loss for each epoch.



4. In the images section of Tensorboard, you can visually compare the network performance for the validation data over each training epoch.


TensorBoard
SCALARS
IMAGES
INACTIVE
Show actual image size
Runs
Write a regex to filter runs
Tags matching /.*/ (all tags)
Move this slider
Input-Segmentation_Probability-Overlay
step 35
Wed Jun 09 2021 10:59:25 GMT+0200 (Central European Summer Time)

Recommended Road Map

1. Complete in **image_dataset.py**

——— Test using **02-test-image-dataset.ipynb** ———

2. Complete in **augmentation.py**

——— Test using **03-test-augmentation.ipynb** ———

3. Complete in **unet.py**

——— Test using **04-test-network.ipynb** ———

4. Complete in **metrics.py**

——— Test using **04-test-network.ipynb** ———

5. Complete in **visualization.py**

——— Test using **04-test-network.ipynb** ———

6. Complete in **model.py**

——— Test using **05-train.ipynb** ———

Submission

- After completing the code implementation, run the **05-train.ipynb** notebook and make sure that your are able to train the model. Also test the model prediction using **06-predict.ipynb** notebook.
- Show your work to one of the tutors.
- Upload the **05-train.ipynb** notebook as an html file to StudOn.

References

[1] Sasank Chilamkurthy. Pytorch tutorial: Writing custom datasets, dataloaders and transforms. [https://pytorch.org/tutorials/beginner/data_loading_tutorial.html].

[2] Sasank Chilamkurthy. Pytorch tutorial: Transfer learning for computer vision tutorial. [https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html].

[3] O. Ronneberger, P.Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, volume 9351 of *LNCS"*, pages 234–241. Springer, 2015. (available on arXiv:1505.04597 [cs.CV]).

[4] PyTorch. torch.nn documentation. [https://pytorch.org/docs/1.9.0/nn.html].