LabMLISP: Lab Course Machine Learning in Signal Processing SS 2019

Part V: Neural network

Prof. Veniamin Morgenshtern

Author: Kamal Nambiar, Angel Villar Corrales, Matthias Sonntag

Loading Dataset and problem set files

To copy the files required for this task to your project folder, execute:

```
$ cd labmlisp

$ cp -r ~/SHARED/DATA/mlisp-lab/ps5_network/* .

$ cp -r ~/SHARED/DATA/mlisp-lab/data .
```

Fill in the missing code in *Img_dataset.py* in the data folder. One of the tasks here is to read images from folders. We recommend to use scikit-image for reading images. However, you are free to use other modules too. Pay attention to details like the datatype in which the image is stored and the size/shape of the image. You can skip for now the step that applies the augmentation and focus on it once the network is working and predicting.

Once you are finished with the *Img_dataset.py*, run the *InspectDataset.ipynb* Jupyter notebook. This notebook creates a small experiment with a small dataset of only 8 images. Next, these images are displayed along with their labels. Make sure that these labels accurately correspond to the road lanes from the image, as this is necessary for the Network to learn.

Reference

`https://pytorch.org/tutorials/beginner/data_loading_tutorial.html`

U-Net Architecture

The U-Net architecture used in this lab is a variant of the U-Net architecture proposed in the paper: *U-Net: Convolutional Networks for Biomedical Image Segmentation by Olaf Ronneberger, Philipp Fischer, and Thomas Brox*. Using the block diagram and the table below, fill in the missing code in UNet.py from the models. For parameters that are not mentioned in the table, use the default parameter values of the corresponding layers as per pytorch documentation. The down-sampling operation in the block diagram is achieved using Maxpool. For the up-sampling operation you may use the nn.Upsample. Please note that the up-sampling and downsampling operation will not change the number of feature maps/channels of their respective input. For the Down and Up layer convolutions, calculate the input channels and output channels and set them accordingly. Pay

attention to the change in number of channels as a result of the concatenation step while setting up first convolution step of each Up layer. The nn.Module in pytorch is used for defining the various blocks in the diagram. The *__init__* function is used to specify the various components in each block using the pytorch implementation of these components. You may use nn.Sequential to configure components that follow one another inside the block. In the *forward* function, the objects already created in the *__init__* function are used to generate output from the input provided to the function.

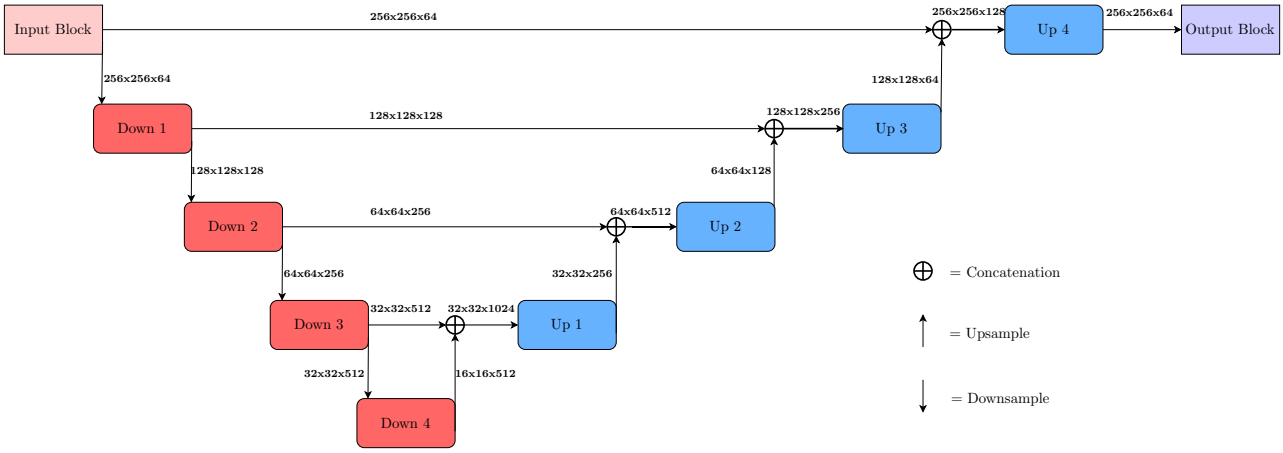

Figure 1: Block Diagram of the model

	Operation	Channels / Feature Maps		Comments
		in	out	
Input Block	Convolution	3	64	Kernel size:3, padding:1
	Batch Normalization			
	ReLU			
	Convolution	64	64	Kernel size:3, padding:1
	Batch Normalization			
	ReLU			
Down	Maxpool			Kernel size:2
	Convolution	x	2x[1]	Kernel size:3, padding:1
	Batch Normalization			
	ReLU			
	Convolution	Same		Kernel size:3, padding:1
	Batch Normalization			
	ReLU			
Up	Upsample			Scale factor=2
	Concatenation			
	Convolution	x	x/4[2]	Kernel size:3, padding:1
	Batch Normalization			
	ReLU			
	Convolution	Same		Kernel size:3, padding:1
	Batch Normalization			
	ReLU			
Output Block	Convolution	64	2	Kernel size:1

Table 1: U-Net Architecture Details

Reference

https://arxiv.org/pdf/1505.04597.pdf
https://pytorch.org/docs/stable/nn.html

[1]For the Down4 block use input channel = output channel = 512
[2]For the Up4 block use input channel =128; output channel = 64

Training and Prediction

Fill in the missing code from the *Training.py* file. The tasks in this script includes setting up the model and executing training. Refer to Train.ipynb as an example to see how your code will be used and make sure that the code in the setup_model() will setup the model according to the parameters that are passed to this function. While executing training, test the model (using test_epoch function) after each epoch(1 iteration over all images in the dataset).The testing of the model would include only the forward pass step of the network. However, to visualize the progress of the training, we will calculate the loss at each step. The loss is stored in a list which is used for displaying certain stats like average, minimum,maximum loss etc. Tensorboard is used in the visualize the network and the training processes. To access tensorboard, click on the 'New' button on the top right corner in Jupyter and select tensorboard from the drop down. The code provided to you will generate graph,image and scalar information in tensorboard. You can refer the tensorboard documentation and include other information that you may find useful.

The code to be filled in the *Prediction.py* is similar to the code in the *test_epoch* function. However notice that the dataloader will only provide images to this function and you are not required to calculate losses here.

As you fill in the code for Training and Prediction, simultaneously complete the code for the *visualization.py* in the utils folder. Functions in this script will be used in the for visualizing purposes in Training and Prediction scripts. The tasks in script involve image processing and you are free to use any python module of your choice. We recommend using PIL module. To generate the overlay image, the output image should contain the the input image as a background and the detections of the network in red color overlayed on top of this background image. Pay attention to the input that is provided to the function and do the necessary type conversions if required.

Experiments

Once you complete writing the code, you can train the code using the *Train.ipynb* notebook and use the *Predict.ipynb* notebook for prediction. Pay attention to the results that you obtain from cell 9 and 10. The cells will test your trained model against real world data. Cell 9 has a true negative image (with no road) and Cell 10 consist of multiple road images. You can use these two cells to get a visual impression of how your network is performing with real images when you conduct different experiments mentioned below.

There are several features and parameters that you can tune to change the behavior of the network:

- Experiment with the number of epochs used for the training. Does the performance improve when you train for more epochs?
- Experiment with the number of training images. Does the performance improve when you use more images for training?
- One of the problems you might have observed when evaluating the performance of the network on real images is that it produces false positives in the bright portions of the images. To mitigate this problem, the code that you have created supports training with both simulated

and real world true negative images. Compare the performance using only simulated images and when using both simulated and true negative images. Do you see the reduction in false positive detections when the network is trained with real world true negative images?

- Vary the learning rate of the optimizer and other parameters if applicable.
- Experiment with different optimizers available in Pytorch (Must try: Adam Optimizer)
- As you may have noticed, in each image the number of pixels corresponding to lane is much less than number of pixels corresponding to non lane. This imbalance might lead to the network preferring to predicting most pixels as 'not lane' in order to achieve a lower loss. However, our goal is to detect the lanes and hence we need to force the network to learn about the lanes. The solution to this problem is to add higher weights to the pixels corresponding to lane in our loss function and thereby compensate the class imbalance. We can vary the weights using *lane_to_nolane_weight_ratio* variable in the cell 7 of the *Train.ipynb* notebook.
- Complete the augmentation task (next pdf on the webpage) and repeat the experiments to see the effect of data augmentation in your network.
- It has been found from certain practical experiments that normalizing the input will make training faster and reduce the chances of getting stuck in local optima. You can normalize the input tensor by removing the mean and scaling it to unit variance. Compare the performance by training the network using normalized data and un-normalized data.

As the total number of combinations of parameters is quite large and trying all of them would require too much time, you can find in Table 2 a list of insightful experiments that must be carried out. Which ones work best? Which ones do not output meaningful results? Why? Feel free to further tune these parameters in order to optimize them to your network and code.

Experiment	num epochs	lr	ltnwr	num imgs
1	8	1e-7	15	1250
2	5	1e-3	15	150
3	5	2e-4	1	1250
4	1	5e-4	4	2000
5	4	0.1	200	1500
6	4	0.1	200000	1500
7	4	1e-3	60	1500

Table 2: Experiments

In engineering in general (and Machine Learning in particular), it is crucial to document the results to be able to compare different experiments and to reproduce research. Therefore, document your results for every experiment by saving the predictions and by writing down the training time and some observations about the obtained results.

For the different experiments that you will be conducting, use the *create_experiment* function and create experiments with appropriate names that will help you understand how the network in that experiment is setup. When you create the experiment, all folder that you would require for the experiment will be created automatically and in the network_data folder, you will find the config file specific to your experiment. You are allowed to train a network in an experiment only once. After you complete the training, you can use the model from the experiment by loading the experiment in the predict notebook using the *open_experiment* function. If you would like to delete an experiment and start over again with the same experiment name, use the *delete_experiment* function. If you are interested to retrain a network using an already trained model, you can use the *create_retrain_experiment* function and pass the model filename as a parameter. You can see the implementation of these functions in *serde.py*. Also, have a look at the various parameters defined in the config file of your project and make use of these in your code as required.

Model selection

When you train the network, store the model after each epoch. Apply each of the saved networks to 10 simulated images from the test set and to 10 real images from the test set. Visually choose the best network among all models you have saved.

Reference

`https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html`