Visual Navigation: A Deep Learning Perspective

The goal of this practical is to design and implement (in python) a data-driven object detection algorithm for vision-based mobile robots. You will first train and test a Convolutional Neural Network (CNN) model for detecting a particular object on a public dataset. You will then validate your model on a Segway robot, named Loomo, for object following. If time permits, you will further improve your model to overcome practical challenges under various conditions. The emphasis of this lab will be placed on perception; ready-to-use control packages will be provided.

1 Object Detection

1.1 Overview

In this section you will develop a simple object detector using Python and the deep learning library PyTorch. The goal is to detect black patches with white contours which has been randomly applied to images. In Figure 3 some examples on how the images look like.

Most of the code has been already implemented for you. You can download the ipython notebook at https://colab.research.google.com/drive/1DxtK3s-_K1MYzHOhKLfxKqssFb03jFK0#scrollTo=N3H4HLOfxreK and the dataset at https://www.dropbox.com/s/8ch8l3h70ke6s9e/selected_imgs.zip?dl=0. We use a free cloud service called Google Colab https://colab.research.google.com/, which offers free GPUs. Figure 1 shows how to load the given notebook in Colab.

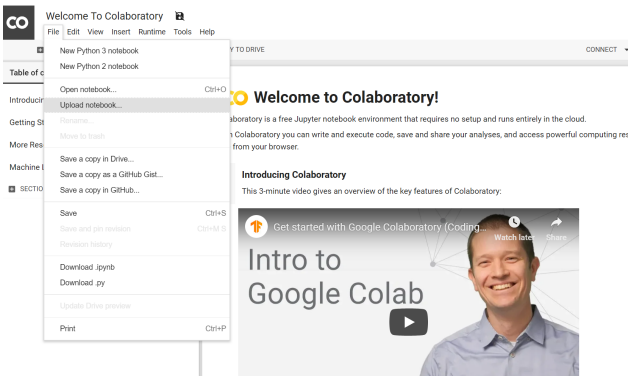


Figure 1: Load iPython notebook in Google Colab

There are two parts you need to complete to successfully run the code: the metric for evaluating the detector and the loss functions for training.

1.2 Intersection over union

The detector needs to find if there is a black patch in the image and where is located. The outputs of the network are:

- A confidence of the patch being present in the image
- Location and size of the bounding box, parameterized with the coordinates of its top left corner (**x**, **y**), its height **h** and width **w**.

To compare the similarity of the predicted bounding box B1 with the ground truth patch B2 we use the metric called *Intersection over Union (IoU)*, which can be defined as:

$$IoU = \frac{B1 \cap B2}{B1 \cup B2} \tag{1}$$

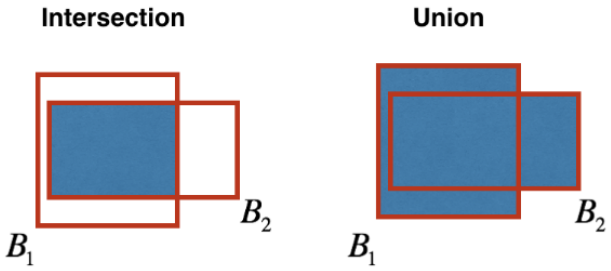


Figure 2: Illustration of intersection over union (IoU) metric

The first task is to implement a Python function which receives as input two boxes and outputs the IoU between them. Further instructions are provided in the Colab Notebook.

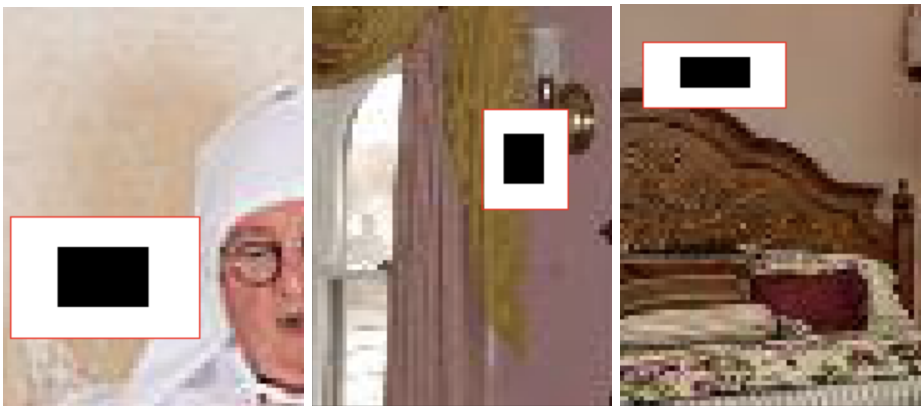


Figure 3: Examples of images with a patch on a random position.

1.3 Loss functions

As already mentioned, the network objectives are composed of a classification term (deciding if there is a patch in the image) and a regression one (location and size of the patch in the image). You need to define the two loss functions for the two objectives separately and then combine them together to create a final loss function. In particular:

1. define the two loss functions for the classification and the regression task
2. combine them together using a weight parameter γ to balance the relative influence of the two loss functions

Here a list of common loss function you can check and test (after deciding if they are for classification or regression):

- Mean Squared Error / L2 Loss
- Mean Absolute Error / L1 Loss

- Hinge Loss
- Cross Entropy Loss

The most common loss functions can be instantiated using PyTorch predefined classes. This assures that they are compatible with the backward pass of the neural network. A good practice is to explore PyTorch official page at *https://pytorch.org/docs/stable/nn.html*.

Once your model has been trained successfully, the Colab notebook provides instructions to download the trained model to your local machine.

2 Model Deployment

By this stage, you should have a trained model downloaded on your local machine. Below are the instructions to make your model run with Loomo.

2.1 Virtual Environment

Create a virtual environment for this project so that it does not interfere with the other projects on your local machine.
$pip install virtualenv

Go to the project folder and create
$cd name_of_your_project_folder
$virtualenv name_of_your_environment

Activate the virtual environment
$source name_of_your_environment/bin/activate
Note: Download all the requirements with the virtual environment activated

Deactivate the virtual environment
$deactivate

2.2 Requirements

Download the prerequisites using the following commands:
$pip install opencv-python
$pip install image
$pip install matplotlib

Install the pytorch version using the following link:
https://pytorch.org/get-started/locally/
Note: CUDA should be None and install using Pip

2.3 Model Path

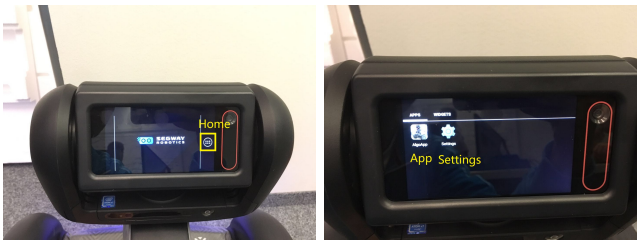
Move the downloaded trained model to the project folder containing the files necessary for communication with the robot, *i.e.*, detector.py and client.py. Name the trained model as 'saved_model.pth'.

3 Robot Experiments

Now, it is time to deploy your model to a robot in the real world!

3.1 Turn on your robot

To turn on a Loomo, press the button on its body and the button on its head in turn. Figure 4 shows the main pages after you turn on the robot.

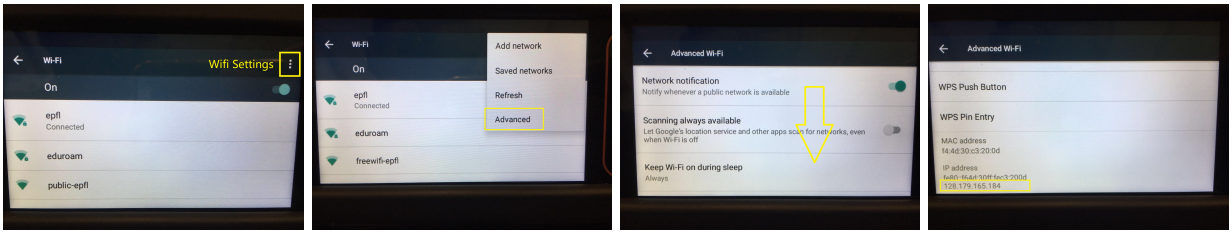


(a) Home **(b)** Apps

Figure 4: Turn on robots

3.2 Find dynamic IP address

To establish the communication between the robot and your pc via Wi-Fi, you need to find the IP address, as shown in Figure 5. Once you acquire the IP address, substitute it in the placeholder host under 'IP Address of server' in client.py.

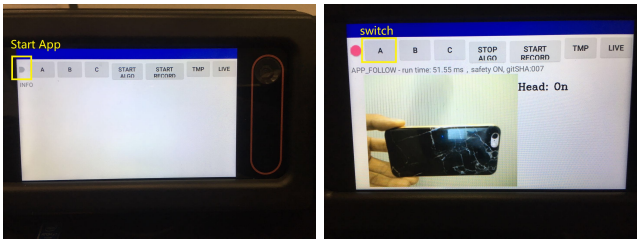


(a) Step 1 **(b)** Step 2 **(c)** Step 3 **(d)** Step 4

Figure 5: Find IP address

3.3 Start robot app

You can now get back to the main page by tapping the robot's ear. Figure 6 shows how to start the robot app for object detection. Once the app is initialized, your can switch the head controller between on and off using the button A.



(a) Before Start **(b)** After Start

Figure 6: Start App

3.4 Run your detector

You can finally use your model by the command:
python client.py

4 Report

You have one week after the lab to submit a group report answering the following two questions:

4.1 Hyperparameter tuning

Problem Description Deep learning performances are highly dependant on hyperparameter tuning. Hyperparameters are parameters whose value you can decide in order to control the learning process. The name *hyperparameters* allows to distinguish from the matrix *parameters* (also called weights), which are learned by the network during the training phase.

An example of hyperparameter is the learning rate, also called step size, of the optimization algorithm. Too high learning rates may lead to jumping over minima and too low learning rates may lead to slow converge or even get stuck in undesirable local minima.

In this task you are required to perform the hyperparameters tuning of your object detector, i.e. to retrain your model with different combinations of your hyperparameters and choose the ones that lead to the highest accuracy.

There are at least three ways you can perform your hyperparameters search:

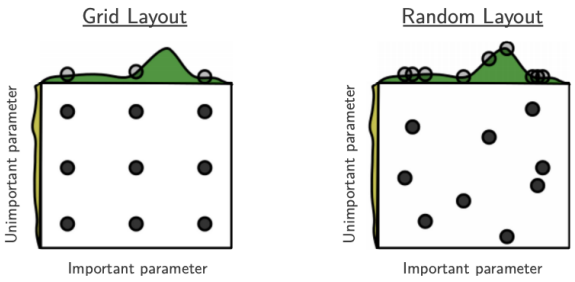


Figure 7: An example of Grid Search vs Random Search.

1. **Babysitting**. The simplest solution you can think of: trial and error. It consists in manually training the model every time with a specific combination of hyperparameters until you are satisfied with the results.
2. **Grid Search**: Define a grid of **n** dimensions (where n is the number of hyperparameters you want to search for), for each dimension define the range of possible values and then search for all possible combinations
3. **Random Search**: Similar to grid search but now values for every dimension are defined randomly instead of grid-based, as show in Figure 7.

Questions

1. In your opinion, which of the three techniques is the most effective for hyperparameters tuning?
2. Why do you think is the reason? A hint is that not every hyperparameter has the same importance.
3. You can now run your hyperparameters search and fullfil Table 4.1. Which are the hyperparameters the model is most sensible of?

Hyperparameter	Best Value	Min value	Max Value	Number of combinations (different)
Learning rate				
Number of epochs				
Batch Size				
Gamma				
...				

4.2 Performances on a real-world application

In the second part of your practical you deployed your model on Loomo robot. When using a model on a real-world application, usually you cannot expect the model to obtain the same performances.

1. In your opinion, why the object detector is less effective on images taken from a real-world camera?
2. Which are the main differences with respect to the settings the model has been trained on?