SML Project 1

SML 2024

1 Introduction

In this project, your task is to help the Robotic Systems Lab (RSL) at ETH Zurich to make their robot ANYmal (Figure 1) estimate how close obstacles are. ANYmal has complex and advanced algorithms to detect obstacles and navigate in challenging environments, as you can see in in the following videos:

1. ANYbotics Introduces End-to-End Robotic Inspection Solution.
2. Robotic Hike with ANYmal.

However, researchers at RSL also need a lightweight distance estimator as a backup in case these existing algorithms fail. In this project, you must help them. You must use ML to produce an estimator. It takes as input an image captured by ANYmal's camera and it outputs *an estimate of the distance to the closest obstacle in the image.*

Figure 1: ANYmal

2 Data

2.1 Folder structure

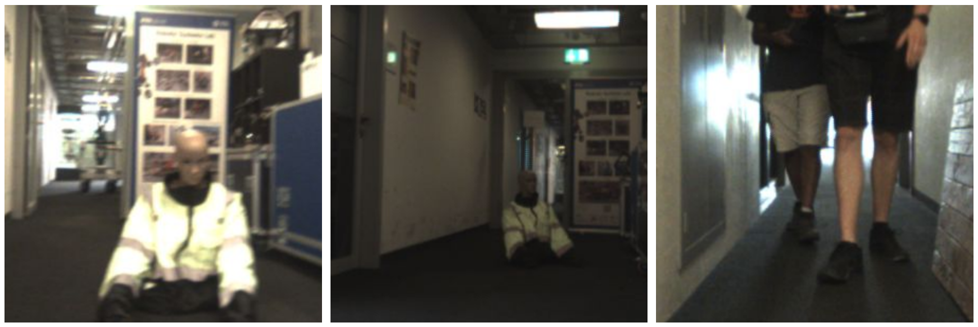
If you choose to work in this project locally in your computer, you can find the project material in **project_1.zip**, in Moodle. Please extract the zip file in a folder where you want to have the project. The datasets are given in the folder **data**. In **train_images**, you see the public training dataset. You see there the robot's observations from its camera while it is travelling around an office. In the file **train_labels.csv**, you see the distance to the closest obstacle for each image in meters. These values are obtained using the robot's depth sensor. You can see some examples from the dataset in Figure 2.

In the folder **public_test_images** you find the public test dataset. The labels for these images are in the file **public_test_labels.csv**. This dataset is intended for validation and not for training. You may use it for training at your own risk of overfitting.

In the folder **private_test_images** you find the private test dataset. We do not tell you the labels for these images as you must estimate them with a trained model.

If you choose to work with JupyterHub, go to the JupyterHub in Moodle. After starting your server, you find all python scripts in the folder **project1**. The folder structure for the data is similar, but you cannot directly access the images from the JupyterHub. You can only load them from the python scripts that we gave you.

2.2 Image representation



(a) 1.95 meters (b) 2.73 meters (c) 1.67 meters

Figure 2: Examples from the dataset.

In this project, features are images and labels are distance values. Images are not one dimensional feature vectors as we saw in the lectures. Images are represented by the pixels' intensity values in our computers (pixel: the smallest unit of an image). You can see a visualization of pixels in Figure 3. In sufficiently high resolution images, we do not see each individual pixel but when you actually zoom-in enough, you can see the pixels. In a typical color image, the colors of pixels are acquired by mixing primary colors: red, green and blue (RGB). Therefore, an image can be described as a matrix M of dimensions $W \times H \times 3$ with $W, H \in \mathbb{Z}^+ (W : \textit{width}, H : \textit{height})$. For $i \leq W, j \leq H, k \leq 3$, the element M_{ijk} describes the intensity of color k for the pixel (i, j) of the matrix. Combinations of different levels of these colors give us every color we see on the screen. See Figure 4. To treat images as feature vectors, we "flatten" the images, i.e. the rows of the image matrices are concatenated sequentially in a single row. The flattening operation is visualized in Figure 5. In a color image, we further flatten the color dimension similarly. Then, each color component of each pixel value can be thought of as a feature. For example, an RGB image with height = 30 pixels and width = 30 pixels gives us $30 \times 30 \times 3 = 2700$ features.

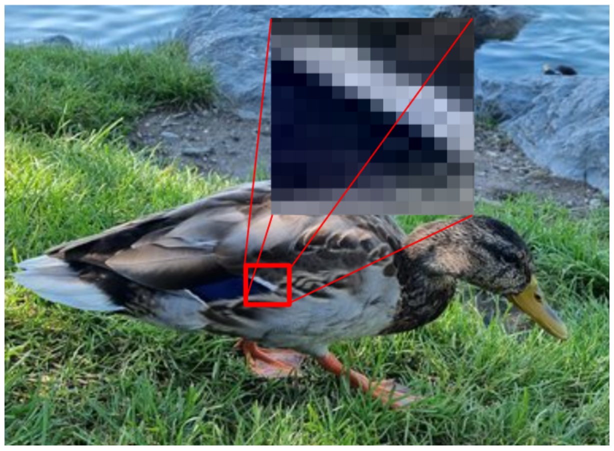


Figure 3: Pixel visualization.

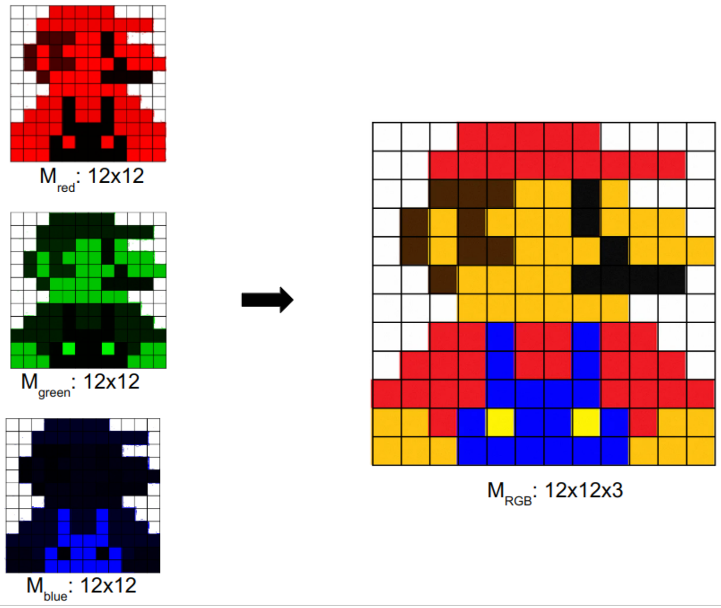


Figure 4: Typical color image is formed by mixing three primary colors. You can see that the yellow pixels are acquired by mixing green and yellow while pure blue pixels do not have red or green components.

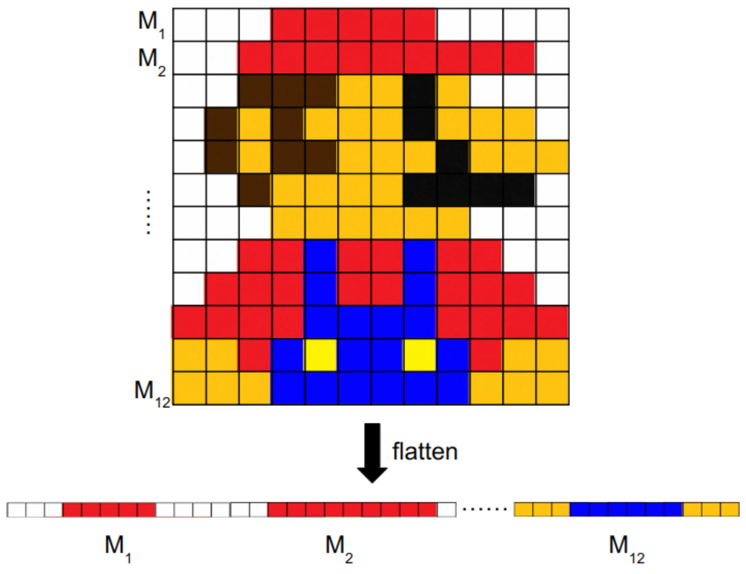


Figure 5: The flatten operation.

3 Setup

First, follow the instructions given in the `README` file to set up your Python environment. For the implementation of the project, you must use the scikit-learn library. It has all the processing methods and machine learning models with utility functions already implemented. Hence, you will not need to implement any function from scratch. You will just need to use the existing functions and models in a correct way. You are advised to go through the documentation and examples in scikit-learn for reference.

In the following sections, we will go over some implementation details.

4 Loading data

The function `load_dataset` loads the images and corresponding distance values as NumPy arrays. It is already provided to you in **utils.py** and it is called in **main.py** where you need to implement your solution. It gives you *images* and *distances*. You should not change this function. There are two settings for the data you can modify in **config.yaml** in the folder **data**.

- *load_rgb*: As described earlier, a color image consists of three channels: red, green and blue. So, every pixel in the image is represented by three values in an RGB image. If you set *load_rgb* as False, the image is converted to grayscale, meaning there is only one channel and each pixel is represented by only one value. If you load as color image, you now have three times the features of grayscale.
- *downsample_factor*: The images in the dataset are originally 300x300. *downsample_factor* resizes the height and width of the image, e.g. if *downsample_factor*=5, the image dimensions become 60x60. While reducing the resolution of an image, we lose some information. You can think about the fact that nearsighted people cannot read texts they can read with glasses on, where some information is lost due to low resolution. On the other hand, a higher-resolution image has more information while having more features.

The function `load_dataset` reads the image files and resizes them based on the *down-sample_factor* you set. Then it converts the images to grayscale if *load_rgb* is false. Then the images are flattened. The result of these operations is a NumPy array of size (number of images)x(number of features). The labels are loaded as a one dimensional NumPy array of size (number of images).

Remark: Note that this function takes two arguments. The second argument is by default the string `train`. With this argument the function loads the train data. If you use `public_test`, it will load the public test data.

We have also provided the function `load_private_test_dataset`. This function is implemented in **utils.py** and does the same as `load_dataset` except that it loads **only** the images of the private test set.

5 Restriction

The only restriction in this project is that you must use only scikit-learn libraries for this. **In particular, you are not allowed to use CNN technologies like PyTorch or Tensorflow.**

6 Tips for solving this project

6.1 Creating splits

Now that we loaded the features and the labels, we need to split the data. It is a common practice in machine learning to divide the data in train, validation, and test splits. The train split is used to learn the parameters of the model while the validation split is used to learn the hyper-parameters of the model. Hyper-parameters are parameters that are not learned from the data during the training process but are set prior to training. They control various aspects of the learning algorithm. The test split is the data you use to evaluate your final model.

By using validation splits, you make sure that your model is not over-fitted to the training split. Overfitting is a common issue in machine learning where a model learns the training data so well that it performs poorly on new, unseen data. Overfit models have high training accuracy but low testing accuracy, making them unreliable for real-world applications. Your grade will be calculated on your model's performance on a private test set, which will not be shared with you. Therefore, you need to make sure that your model is not overfitted to the training data.

6.2 Preprocessing data

We now have our data which is divided in train and validation splits. Next step is to preprocess the data. If you go through the images, you realize that the illumination in the scene and the objects present in the scene differ a lot between frames. This leads to variations in pixels values. In order for a machine learning model to perform well, it is important that the features are scaled in a common range. Therefore, you might need to use a scaling method from the ones available in scikit-learn library. You can take a look at the related documentation.

6.3 Dimensionality reduction

As you have learned about the *the curse of dimensionality*, it helps machine learning models to reduce higher dimensional features to lower dimensions by keeping as much information as possible. You have learned principal component analysis (PCA) back in Informatik II. You can consider using it and decide number of components to keep. Note that other ways of dimensionality reduction are down-sampling the image and converting to grayscale while loading as explained in 4.

6.4 Choosing the right model

The data is ready for training and now we need to choose a regressor from the scikit-learn library to train the model. There is not a simple answer in general on how to choose the right model. You must consider the size of dataset, the task etc. For example, do you think there is a linear relation between the pixel values of an image and the distance to the closest obstacle? If yes, you can opt for linear regression, if not, you need to choose a model that captures higher order relations.

We recommend that you explore the following models:

- Kernel ridge
- Support vector regression
- Random forests

6.5 Hyper-parameter Tuning

As there is no direct way to choose the right model to use for a machine learning task, the selection of hyper-parameters also requires experimenting and seeing what works best. You should use grid search to find the right hyper-parameters. Check here to learn more about grid search.

6.6 Pipelines

In this project, you can use pipelines. A pipeline in scikit-learn combines multiple steps in data processing and the ML model in a single entity. Please go through the documentation to learn about pipelines and how they are used.

Combining pipelines with grid search can lead to very powerful methods for finding the right model for your data. You can find an example here: here.

7 Submitting your code

Only one member of your team will submit a zip file through Moodle with the following:

- Your **predictions** for the examples in the private test set. You can produce this file using the function `save_results` which takes as argument your predictions for a particular dataset and saves it as a .csv file. Make sure that you call this function using your predictions for the *private test set*.
- Your **source code**. Make sure to include all files needed to reproduce your code like **config.yaml**, **main.py**, and **utils.py**. We need your code for plagiarism checks and, in some cases, to validate your results.
- A text file called **"teammates.txt"** with one line per team member, including the person making the submission. Each line contains the legi and the student's name separated by a comma. For example, "11-222-333,Max Muster".

8 Grading

We evaluate your models using the mean absolute error (MAE) of the predicted distance with the ground truth distance.

- The baseline for a 1.0 in the private test set is 35cm.
- The baseline for a 4.0 in the private test set is 26cm.
- The baseline for a 6.0 in the private test set is 12cm.

If you use grid search, consider using the following constructor: `GridSearchCV(model, param_grid, cv=5, scoring='neg_mean_absolute_error', verbose=2)`, where `model` defines a machine-learning model and `param_grid` is the grid with the candidates for each hyperparameter. This constructor creates a grid search that evaluates estimators using the MAE.