Robot Learning

Exercise 1 – Imitation Learning

Release date: Wed, 11 Sept. 2024 - **Deadline for Homework: Wed, 25 Sept. 2024 - 23:59**

For this exercise you need to submit a **.zip** folder containing your report as a **.pdf** file (up to 4 pages), your pre-trained model as a **.pth** file and your code. Make sure to include supporting evidence, in the form of tables and plots, when appropriate, e.g., when training a network plot the loss and for each experiment show the score performance in a table. Please use the provided code templates for these exercises.
Changes to the CARLA environment, or to additionally installed packages will not be considered. Comment your code clearly, use docstrings and self-explanatory variable names and structure your code well.

1.1 Data Collection (1+2+4+1 Points)

Imitation learning requires collecting a dataset of expert operations and using a model to mimic how the expert manipulates the vehicle in response to the environment. By now, you should have installed CARLA simulator and explored it a bit. In this question, you will need to obtain data manually (yes, think you are the driving expert) by interacting with the CARLA simulator. Please follow the steps below to complete your data collection process.

a) CARLA setup. First launch the simulator at low resolution. Launch the CARLA simulator and select the desired map, weather conditions, and other parameters. We recommend using Town01 and dynamic weather conditions. We also recommend using low-resource rendering, e.g.,

```
DISPLAY= ./CarlaUE4.sh -opengl -quality-level=Low -resx=320 -resy=240
```

Open a new terminal. Change the map using the config.py script under PythonAPI/utils.

```
python config.py --map Town01
python dynamic_weather.py --speed 1.0
```

Next, let's simulate some urban traffic. Open a new terminal, and run spawn_npc.py under PythonAPI/examples to spawn vehicles and walkers. Let's just spawn 50 vehicles and the same amount of walkers:

```
python spawn_npc.py -n 50 -w 0 --safe
```

b) Control agent: Create a control agent and set RGB camera sensor. You will need to create a data collection script in pyton that interacts with the simulator and provides control inputs such as steering, throttle, and brakes. You will also need an RGB camera sensor to retrieve visual observations. This step can be done by modifying manual_control.py under PythonAPI/examples. The script PythonAPI/examples/no_rendering_mode.py can help create a minimalistic aerial view with Pygame, that will follow the ego vehicle. This could be used along with saving functionalities (examples in manual_control.py) to generate autopilot trajectories and record (hint: look into apply_control). Alternatively, you could drive manually and collect data with you as the controller.

c) Collect a Dataset. Use the control agent to collect data by controlling the vehicle and recording observations such as camera images, sensor data, and control inputs. Store the collected data in a format suitable for training (e.g. numpy arrays, csv files, jpg, etc.). Repeat the steps several time to collect multiple trajectories. Verify the quality of the collected data by visual inspection, checking for missing values, and testing for distribution shifts between expert and agent data.

d) Question. Explain what is a 'good' training data? Are there any problems with only perfect driving demonstrations?

1.2 Network design (2+1+2+2+1+2+2 Points)

Let $\mathcal{S}$ be the state space and $\mathcal{A}$ the action space. The CARLA environment encodes a transition function $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ that maps a (state, action) - pair to a new state.
The aim of this exercise is to design a network that learns a policy $\pi_\theta : \mathcal{S} \to \mathcal{A}$, parameterized by θ, to predict the best action for a given state. We formulate it as a supervised learning problem, where the objective is to follow the observed imitations of an "expert" driver.

a) Implement the function `carla_loader` in `dataset.py`. Given the folder of the imitations, it should load `observation` and `action` into a batch of `observations` and `actions`.

b) To start with, we formulate the problem as a classification network. Have a look at the actions provided by the expert imitations, there are three controls: accelaration, steering and braking. Which values do they take? Since they are not independent (accelerate and brake simultaneously does not make sense), it is reasonable to define classes of possible actions, like {steer_left}, {}, {steer_right and brake}, {gas} and so forth. Define the set of action-classes you want to use and write the methods that convert actions to classes and scores to action. Your code should map every action to its class representation which will be a one-hot encoding as well as compute an action from the scores predicted by the network during inference.

c) Design and implement an easy first network architecture in `network.py`. Start with 2 to 3 2D convolution layers on the images, followed by 2 or 3 fully connected layers (linear layers) to extract a 1D feature vector. Let each layer be followed by a ReLU as the non-linear activation (see code snippets below).
The output of the network should function as a controller for the car and predict one of the action-classes for each given state. At the end of the network, add a softmax layer to normalize the output. We can then interpret it as a probability distribution over the action-classes.

```
torch.nn.Sequential(
    nn.Conv2d(3, 32, kernel_size=5, stride=2),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.Conv2d(32, 32, kernel_size=3, stride=1),
    nn.BatchNorm2d(32),
    nn.ReLU()
    )
```

```
torch.nn.Sequential(
   torch.nn.Linear(in_size, out_size),
   torch.nn.LeakyReLU(negative_slope=0.2))
```

d) Implement the forward pass for your network, which is the function `forward` in `network.py`. Given an observation, it should return the probability distribution over the action-classes predicted by the network. You can decide whether you want to work with all 3 color channels or convert them to gray-scale beforehand. Motivate your choice briefly.
Train your network by running `python3 training.py`. Can you achieve better results when changing the hyper-parameters? Can you explain this?

e) The module `training.py` contains the training loop for the network. Read and understand its function `train`. Why is it necessary to divide the data into batches? What is an epoch? Please answer shortly and precisely.

f) Implement the loss function `cross_entropy_loss` in `training.py` to calculate the training loss for a given pair of predicted and ground truth classes. We ask that you implement the function and do **not use the default built-in** PyTorch function.

g) Train your network by running training.py. Record and plot the change of the loss with the training epoch. Save your best model in .pth format.

1.3 Network Improvements (2+2+2+2+2+2 Points)

Now that your network is up and running, it's time to increase its performance! Each of the following tasks adds to its architecture. It is up to you to choose which of them you use for participating in the competition. However, all subtasks need to be answered. Evaluate and compare different methods always on the same training data (no matter whether that is the provided or self-recorded data or a mix of both).

a) Observations. The training data of the network can also contain more information than just the image from the car in the environment. Look at the mounted sensors in `manual_control.py`. What are those sensors? Incorporate it into your network architecture. How does the performance change?

b) MultiClass prediction. Design a second network architecture that encodes a multi-class approach by defining 4 binary classes that represent the 4 arrow keys on a keyboard and stand for: steer right, steer left, accelerate and brake. Since those don't all exclude each other, let the network learn to predict 0 or 1 for each class independently.
You will need to implement another loss function and might find a sigmoid-activation function useful. Again, compare the results to the previous classification approach.

c) Classification vs. regression. Formulate the current problem as a regression network. Which loss function is appropriate? What are the advantages / drawbacks compared to the classification networks? Is it reasonable to use a regression approach given our training data?

d) Data augmentation. As discussed in the lecture, the more versatile the training data is, the better generally. Investigate two ways to create more training data with synthetically modified data by augmenting the (observation, action) - pairs the simulator provides. Does the overall performance change?

e) Historical observations As autonomous driving is a sequential decision-making process, past observations also have a certain impact on current decision-making. Concatenate prior frames as a current observation - do you notice an improvement in driving performance? Why or why not?

f) Fine-tuning. What other tricks can be used to improve the performance of the network? You could think of trying different network architectures, learning rate adaptation, dropout-, batch- or instance normalization, different optimizers or class imbalance of the training data. Please try at least two ideas, explain your motivation for trying them and whether they improved the result.

1.4 DAGGER Implementation (4+5 Points)

a) The traditional approach to imitation learning ignores the change in distribution and simply trains a policy π that performs well under the distribution of states encountered by the expert d_{π^*}. This can be achieved using any standard supervised learning algorithm. It finds the policy $\hat{\pi}_{sup}$:

$$\hat{\pi}_{sup} = \underset{\pi \in \Pi}{\operatorname{argmin}} \ \mathbb{E}_{s \sim d_{\pi^*}}[\ell(s, \pi)] \tag{1}$$

Prove that by assuming $\ell(s, \pi)$ is the 0-1 loss (or upper bound on the 0-1 loss) implies the following performance guarantee with respect to any task cost function C bounded in $[0, 1]$:

Theorem 1.1 *Let* $\mathbb{E}_{s \sim d_{\pi^*}}[\ell(s, \pi)] = \epsilon$, *then* $J(\pi) \leq J(\pi^*) + T^2\epsilon$.

DAGGER Failure What could be done to help DAGGER converge faster? Please also answer when does DAGGER lead to a performance similar to Behavior Cloning? Are there common scenarios in Urban Driving that could lead to quadratic cost?

b) **Implement the DAGGER algorithm.** Implement the DAGGER algorithm [3] below and apply it to your dataset. Do you observe an improvement in performance? What scenarios benefit most from DAGGER? Compute and plot the regret, what do you observe? Compare the result and report your findings.

Initialize $\mathcal{D} \leftarrow \varnothing$.

Initialize $\hat{\pi}_1$ to any policy in Π.

FOR $i = 1$ **to** N

Let $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$.

Sample T-step trajectories using π_i.

Get dataset $\mathcal{D}_i = \{(s, \pi^*(s))\}$ of visited states by π_i and actions given by expert.

Aggregate datasets: $\mathcal{D} \leftarrow \mathcal{D} \bigcup \mathcal{D}_i$.

Train classifier $\hat{\pi}_{i+1}$ on $\mathcal{D}$.

END FOR

Return best $\hat{\pi}_i$ on validation.

DAgger Algorithm.

1.5 Competition (3 Points)

When you submit your final models to us, the competition evaluation works as follows: the models are tested on a set of validation routes. The code should expect an input image size of 320 by 240, and output an action for CARLA, of steer amount, brake, and throttle. For an example, please see an agent in:

```
1 performance_benchmark\leaderboard\team_code\test_agent.py
```

You will have to modify the run_step function to include your neural network model. Your code should load the model once during initialization (in _init), and then run inference in run_step. We will evaluate the model on routes in Town 2. The routes will be kept hidden. For each route, we will compute the success rate (higher the better), route completion (higher the better), and the number of collisions (lower the better). The final score is the average rank of the three metrics. The winner is the student with the highest score. Good luck!

1.6 References

[1] https://papers.nips.cc/paper/95-alvinn-an-autonomous-land-vehicle-in-a-neural-network.pdf
[2] https://arxiv.org/pdf/1604.07316.pdf [3] Ross, Stéphane, Geoffrey Gordon, and Drew Bagnell. "A reduction of imitation learning and structured prediction to no-regret online learning." Proceedings of the fourteenth international conference on artificial intelligence and statistics. JMLR Workshop and Conference Proceedings, 2011.