TSFS12 HAND-IN EXERCISE 5
Learning for Autonomous Vehicles

August 28, 2021

1 OBJECTIVE

The objective of this exercise is to investigate different aspects of learning for autonomous vehicles. A first task is to explore reinforcement learning in a scenario with a grid world with a moving autonomous vehicle, modeled as a particle, driving in a world with inherent uncertainties modeled using probabilities. A second task is to investigate neural networks as a means of making predictions of lane changes and driver intent on highways. Compared to the previous four hand-in exercises in this course, most of the implementation code required is provided in this exercise. The intention of the exercises is to provide an understanding and hands-on experience with the methods discussed during the lectures on learning for autonomous vehicles. In the extra assignment for higher grades in Appendix A, neural networks and reinforcement learning are combined to solve a more complex highway-driving scenario with multiple vehicles in several driving lanes.

2 PREPARATIONS BEFORE THE EXERCISE

Before performing this hand-in exercise, please make sure that you have read and understood the material covered in:

- Lectures 10–11.
- Sections 4.1–4.4 and 6.1–6.5 in Sutton, R. S., & A. G. Barto: *Reinforcement learning: An introduction*. MIT Press, 2018, [8].

To get code skeletons to be used for the different exercises, download the latest files from `https://gitlab.liu.se/vehsys/tsfs12` and familiarize yourself with the provided code. The data set needed for the exercise on intent prediction using neural networks is available in Lisam under `Course documents/i80_data`. Please download the whole directory and save it on your computer, and then unzip it in the same directory as the remaining files. In the student labs, the data can be found at `/courses/tsfs12/i80_data` and you do not need to download the data (about 380 MB in total).

For the value-iteration and Q-learning exercises, you are free to choose language for implementation and code skeletons are available in both Matlab and Python. For the neural-network assignment and the deep Q-learning assignment for higher grades, only Python is available. Note, though, that you will only be required to write very limited lines of code in these exercises, so a background in Python is not necessary. In the student labs, all packages needed are pre-installed in the virtual environment activated by

```
1 source /courses/tsfs12/env/bin/activate
```

If you are running on a private computer, make sure to install the packages needed

```
(env) % pip install ipython numpy jupyter jupyterlab matplotlib scipy seaborn
(env) % pip install sklearn torch highway-env torchinfo
```

3 REQUIREMENTS AND IMPLEMENTATION

The objective of the exercise is to get a practical understanding of different methods for learning for autonomous vehicles and evaluate their properties in some example scenarios. Therefore, to pass this exercise you should solve the following tasks:

- Apply value iteration to solve a Markov decision process in a grid-world scenario under known environment dynamics.
- Apply Q-learning to the same problem as in the previous task, but without *a priori* known state-transition probabilities.
- Evaluate short-term vehicle-intent prediction using a neural network.

The exercise is examined by presenting your solution to one of the teachers in the course over Zoom. Time slots for presenting your solutions will be announced in Lisam. During the presentation you should be prepared to answer the questions stated in connection with the different tasks in this document. In addition, the following should be submitted on the Lisam course page by the deadline:

1. Runnable code. It is *not* required to submit code that automatically runs all the cases of parameter variations for the different methods. If your implementation consists of several files, submit a zip-archive.

It is allowed to discuss the exercises on a general level with other course participants. However, code sharing outside groups is not allowed. Moreover, both students in the group should be fully involved in performing all exercises, and thereby be prepared to answer questions on all subtasks.

See Appendix A for the extra assignment needed for higher grades. The extra assignment is done individually and is submitted by a document answering the questions, including suitable figures, plots, and tables. The document is to be submitted in PDF format. The code should also be submitted as a zip-archive. There is only pass/fail on the extra exercise and the document can not be revised or extended after first submission. It is not required to get everything correct to pass the assignment, we assess the whole submission. You are of course welcome to ask us if you have questions or want us to clarify the exercise.

Since these exercises are part of the examination of this course, we would like to ask you not to distribute or make your solutions publicly available, e.g., by putting them on github. (A private repository on gitlab@liu is of course fine).

Thanks for your assistance!

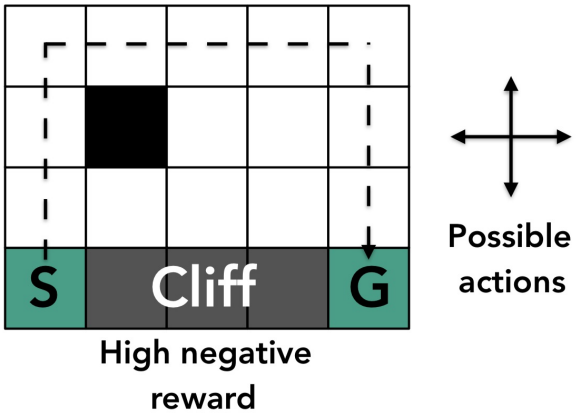

Figure 1: The grid world where the autonomous vehicle is moving from start (S) to goal (G).

4 DISCUSSION TOPICS & EXERCISES

This hand-in exercise is divided into two different parts, with subtasks to be performed for each. The required implementation code for solving the different tasks is to a large extent provided for the exercises in the skeleton files.

4.1 Reinforcement Learning Using Value Iteration and Q-Learning

In this part of the hand-in exercise, we will investigate two different algorithms for solving a Markov decision process using reinforcement learning. More specifically, we will investigate value iteration under known environment dynamics as well as Q-learning in the more complex case of *a priori* unknown state-transition probabilities. The assignment is based on

- Example 6.6 in Sutton, R. S., & A. G. Barto: *Reinforcement learning: An introduction*. MIT Press, 2018.

The scenario is an autonomous vehicle, here modeled as a particle, moving in a grid world from state to state (see Figure 1). In each state in the grid world, the vehicle has four different actions to choose from: left, right, up, and down. These are counted as number 1–4 in Matlab and 0–3 in Python in the implementation, in the respective order. Because of disturbances such as wind acting in the modeled world, applying a certain action does not necessarily mean that the vehicle ends up with moving in the intended direction; instead there is a certain probability that the vehicle will move sideways compared to the intended motion. If the vehicle applies an action that would bring it outside of the world, it remains in the same state and once again receives the reward associated with that state. The overall task is to compute a policy (i.e., actions as function of state) such that it maximizes the expected accumulated reward for the vehicle from start to the goal state.

The motion starts in the lower left corner (S) and the goal is to move to the lower right corner (G). There are certain states in the world that are occupied by obstacles, and the states between the start and goal state in the lower row comprise a cliff where the vehicle gets stuck. The cliff states and the goal states are here considered as terminal states.

The reward associated with the cliff states is selected as a large negative number. The reward for the goal state is chosen to be zero and the remaining free states (i.e., states not occupied by obstacles) have a small negative reward associated with them. The latter choice is to make it cost to take (unnecessary) steps in the world before reaching the desired goal state, which can be interpreted as trying to keep the path

length as short as possible, though under consideration of the risk of getting stuck in the cliff states.

The code required to perform the exercises in the following is available in a number of provided files. In Matlab, the main files are `rl_val_iter.m` and `rl_q_learning.m`. Auxiliary functions are available in the file `select_eps_greedy.m` as well as in the directory `Functions`. In Python, there are notebooks with corresponding content called `rl_val_iter.ipynb` and `rl_q_learning.ipynb`. In the Python version, the auxiliary functions for this task are available in the file `grid_world.py`. Note that the functions `next_state` and `select_eps_greedy` are defined directly in the notebook `rl_q_learning.ipynb`. There are also script versions available if you prefer in the files `rl_val_iter.py` and `rl_q_learning.py`. The parameters defining the scenario are stored in the struct/dictionary `params`, and the properties of the parameters are described in the commented code.

4.1.1 *Value Iteration*

Exercise 4.1. Read through the code provided in the file `rl_val_iter.m` or `rl_val_iter.ipynb` and the associated auxiliary files, and study in particular the variables available in the struct/dictionary `params` that define the Markov decision process to be solved.

Exercise 4.2. In the first part of the exercise, we will investigate value iteration for determining the optimal value function $V(s)$ and the corresponding optimal policy $\Pi(s)$. Explain in your own words what the output of the value function corresponds to.

Exercise 4.3. Study the algorithm for value iteration in Lectures 10–11. Consider the computation of an updated value for $V(s)$ on Line 6 in the algorithm. The function `p_grid_world` (available in the directory `Functions` in Matlab and in the file `grid_world.py` in Python) implements important computations needed for performing this update. Study this function. Explain what the outputs from this function correspond to in terms of the grid world and the scenario considered.

Exercise 4.4. Execute the main loop of the value iteration (`rl_val_iter.m` in Matlab and `rl_val_iter.ipynb` in Python) for the default values of the parameters. For each iteration, the current value function and the associated actions are shown in two different plots for each state in the grid world (using the function `plot_value_and_policy`). The execution is paused between each iteration, and the execution is continued by pressing enter on the keyboard. Run the iterations until convergence.

Exercise 4.5. Which is the optimal path from start to goal? Note that the solution does not only provide this path, but an optimal policy for all states. Is the optimal policy obtained in agreement with your intuition? Relate the numbers in the plot illustrating the value function $V(s)$ to your interpretation of that function from Exercise 4.2.

Exercise 4.6. Now we will investigate how changes in the state-transition probabilities affect the resulting optimal policy. Vary the probability of the car to be subject to disturbances (the parameter `P_move_action`, in the interval (0,1]). What happens if you set this parameter to 1 and what does it mean in practice for this sce-

nario? Is the result in agreement with your intuition on how to optimally negotiate the uncertainty and associated risks in the scenario at hand?

Exercise 4.7. Vary the discount factor (the parameter `gamma`) and study the results on the optimal policy. Do you see any noticeable effects, and are the results in agreement with your intuition?

Exercise 4.8. Finally, vary the negative reward received for ending up in the cliff states (the parameter `R_sink`). How does it influence the optimal path from the start state to the goal state, if it is made significantly less or more negative compared to the negative reward received for the remaining obstacle-free states?

4.1.2 *Q-Learning*

Now we will investigate the case when the state-transition probabilities are *a priori* unknown. The same scenario and Markov decision process as in the previous exercise will be solved, but now using Q-learning instead. The learning rate is determined by the parameter `alpha` and a total number of `nbr_iters` episodes of Q-learning are performed. The trade-off between *exploration* of new actions and states and *exploitation* of already acquired information during the iterations is decided using the epsilon-greedy strategy, where `eps` is the parameter governing the trade-off. This strategy is implemented in the function `select_eps_greedy`.

Exercise 4.9. Read through the code provided in the file `rl_q_learning.m` or `rl_q_learning.ipynb` and the associated auxiliary files, and study in particular the variables available in the struct/dictionary `params`. Q-learning relies on iteratively updating the function Q by running a sequence of episodes where the agent interacts with the environment. This interaction could be either in simulation or in experiments, here we will use simulations in the grid world.

Exercise 4.10. Study the algorithm for Q-learning from Lectures 10–11. A key part is to choose an action, making a state transition, and receiving an associated reward. This procedure is implemented in the function `next_state` (available in the directory `Functions` in Matlab and in the notebook `rl_q_learning.ipynb` in Python). The Q function is subsequently updated using the temporal-difference (TD) error (Line 8 in the algorithm). Explain in your own words what the output of the Q-function corresponds to. From this function, the optimal policy $\Pi(s)$ could also be extracted. Describe how this could be made.

Exercise 4.11. Execute the main loop of the Q-learning (`rl_q_learning.m` in Matlab and `rl_q_learning.ipynb` in Python) for the default values of the parameters (we here start with the case where there are no disturbances and the vehicle thus always moves according to the specified action, i.e., `P_move_action = 1.0`). In contrast to the value iteration, only the final resulting value function and corresponding actions are shown in the plots (using the function `plot_value_and_policy`) since there are typically many more iterations required for Q-learning with unknown state-transition probabilities. In addition, a plot is shown where the averaged accumulated rewards for the episodes are visualized. Does the Q-learning converge to the same optimal policy as the value iteration (if you have the same parameters for the Markov decision process)?

Exercise 4.12. Vary the probability of the car to be subject to disturbances (i.e., the parameter `P_move_action`) to something slightly lower than 1. How does it affect the Q-learning? Does the Q-learning find the same optimal policy from start to

goal as the value iteration (if you have the same parameters for the Markov decision process)? How about the policy at the states other than the start state?

Exercise 4.13. An important parameter in the Q-learning is the parameter `eps` governing the exploration vs. exploitation strategy. Typically, it is desirable with more exploration in the beginning of the episodes when less information has been acquired about the system, and then in the end the agent relies almost completely on the information that has already been acquired.

Update the implementation such that the parameter `eps` is made dependent on the episode number, and successively decreases towards zero, starting from one, when the iteration number increases. The iteration number `itr_nbr` is available as an input to the function `select_eps_greedy`. Do you see any improvements in the performance of the Q-learning and its convergence to the optimal policy? Study also the accumulated reward plot. As a further extension, several exploration/exploitation phases could also be performed, i.e., the parameter `eps` starts at 1 and decreases towards zero and then after a pre-defined number of episodes it is increased again to 1 to re-start the exploration and then subsequently decreased to zero again. Implement also such a strategy. Compare the plots of the accumulated reward as function of episode for the different strategies. Does this more advanced strategy improve performance in terms of the average accumulated reward towards the end of the learning (when `eps` is close to zero again)?

4.2 Short-Term Prediction of Vehicle Intent Using a Neural Network

Knowing where you are and in which environment, i.e., localization and mapping, is essential for an autonomous vehicle. A next step is to also estimate what your environment will look like in the future, i.e., predicting the future motion of the surrounding vehicles is a critical step that is necessary for safe and reliable motion planning under uncertainty [10]. However, modeling human behavior, in this case driving behavior, is challenging and one interesting option is to utilize recorded data from different real traffic situations to build models of behavior. In this exercise, you will design and experiment with a model of short-term vehicle behavior using a neural network.

The data used in this exercise are recorded trajectories of vehicles on a U.S. highway, I-80 in Emeryville, California[1]. The traffic site is shown in Figure 2a, and it is a six-lane highway with an on-ramp, with the high-speed lane to the left. In the data set, there are 4 383 recorded trajectories over 3 times 15 minutes. In Figure 2b, 250 example trajectories are plotted over the 500 m long highway section.

The overall question is then how to predict vehicle motion? On a very short time-scale, vehicle velocity and inertia are sufficient to estimate where the vehicle will be in the immediate future. However, on a time-scale beyond roughly one second, drivers start to act and interact with its surroundings, e.g., change lanes, and inertial prediction becomes less accurate. Learning approaches have been shown to be effective for prediction on a three-second horizon [1]. The objective of the exercise can therefore be summarized as

> *Make a model that predicts if a vehicle will shift to the lane on the left, on the right, or continue in the same lane for the next 3 seconds.*

1 I-80 data set citation: U.S. Department of Transportation Federal Highway Administration. (2016). Next Generation Simulation (NGSIM) Vehicle Trajectories and Supporting Data. [Dataset]. Provided by ITS DataHub through Data.transportation.gov. Accessed 2020-09-29 from http://doi.org/10.21949/1504477. The data are open source and on this link you can also find additional information about the data set.

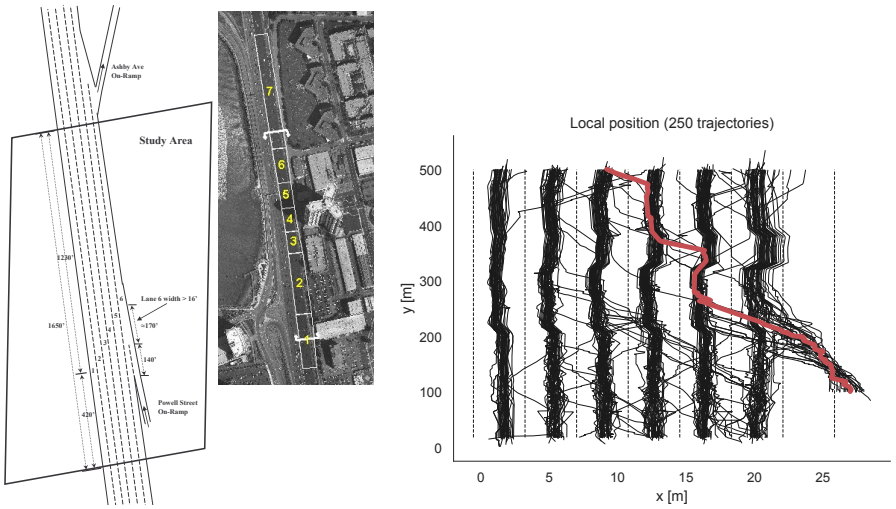

Figure 2: I-80 at Emeryville, California, extracted from the data set from the U.S. Department of Transportation data hub described in the text.

In this exercise you will experiment with a basic approach, if you are interested in this topic you are encouraged to read the survey on vehicle behavior prediction [2]. A more recent survey [7] focuses on learning-based approaches which is closer to the topic of this exercise. There are also interesting application studies, e.g., [5, 1, 10].

There are many aspects that can influence how a driver chooses lane and indicators of impending lane change. Naturally, positions, velocities, and accelerations of surrounding vehicles, the relation between the same lane and the neighboring lanes and the ego vehicle's position, velocity, and acceleration are main factors to consider. But these are not the only ones, for example vehicles more often tend to aim for the fast-moving lanes, but this also depends on the density of vehicles in each lane and of course the (unknown) destination. To make a rule-based model of lane-change prediction, weighing all of these factors, is non-trivial and one possible alternative is to use black-box models and data to model driver behavior. In this exercise, neural networks will be investigated as a tool for accurate lane-change prediction. Recommended readings are the lecture notes and the associated reading material referred to in this document.

For implementation of (deep) neural networks, it is recommended to use any of the well-established tools like, e.g., Tensorflow or PyTorch. We do not recommend to use Matlab for this, so this exercise is only available in Python. However, there will be very limited lines of code to write, a fully running skeleton is provided, so it is still possible to do this exercise without any prior Python knowledge. The skeleton is written for PyTorch. The models you will develop here will be small enough to run on almost any computer, so access to, e.g., GPU resources is not necessary. However, the provided code runs unchanged on a GPU so try that if you have GPU access.

To run the Python code provided, you need some additional Python packages: pandas, scikit-learn, and torch. All can be pip-installed on your own computer and they are pre-installed at the Linux computer labs at campus in the virtual environment that is activated by

```
1 % source /courses/tsfs12/env/bin/activate
```

4.2.1 *Data and Model Description*

Pre-processed data are prepared for this exercise and can be downloaded from the directory `Course documents` in Lisam. This includes pre-processed features and the

whole set of vehicle trajectories, ready to be used for learning your predictive model. If you have not downloaded the complete data set already, download the whole `i80_data` directory, unzip it, and place the directory in your code directory. In the student labs at campus, the data can be found at `/courses/tsfs12/i80_data` and you do not need to download the data (about 380 MB in total).

For each recorded trajectory, a feature vector x and a label y is recorded every 3 seconds. Thus, for a 24 seconds long trajectory, we have 8 data points in total for that trajectory. The label $y \in \{0, 1, 2\}$ is defined as

$$y = \begin{cases} 0 & \text{if the vehicle will change to the left within three seconds} \\ 1 & \text{if the vehicle will stay in lane for the next three seconds} \\ 2 & \text{if the vehicle will change to the right within three seconds} \end{cases}$$

This means that the variable y is what we want to predict. The feature vector x consists of 41 values. There are 6 surrounding vehicles tracked, in front and behind in the current lane and in the two adjacent lanes. For each of these 6 vehicles, velocity and acceleration and the time and space distances are used as features. This gives $6 \times 4 = 24$ features. Then for each of the 6 lanes, mean velocity and density are used and that gives another $6 \times 2 = 12$ features. Finally, for the ego vehicle, the lane number, previous lane number, velocity, acceleration, and relative position within the lane are recorded, which gives another 5 features. This in total gives the 41 features. The model is then a function f such that

$$\hat{y} = f(x) \tag{1}$$

where $\hat{y}$ is the predicted action by a vehicle at some time instant and we want to find the predictor in (1) such that $\hat{y}$ predicts the true labels y as accurately as possible. The feature vector x is summarized in the file `features.md`.

The complete data set consists of 4 383 trajectories resulting in 95 591 data points, where 963 are left-lane changes, 304 right-lane changes, and 94 324 straight ahead. This strong imbalance in data, where the vast majority of data points correspond to straight-ahead motion, is important to handle, otherwise a learned predictor risks to be severely biased. A basic way to do this is to weight underrepresented classes corresponding to the imbalance. Here, balancing of data is done by creating a training data set with M (a parameter to choose) samples from each class. Here, we want M to be significantly larger than the size of the smallest class (304 right-lane changes) so the sampling is done *with replacement*. Choosing the value of M is something you will experiment with during the exercise. There is also a step where data are separated into training and validation data sets. In the notebook, you will get data arrays in the variables

- `x_train` and `y_train` – training data points and class labels
- `x_val` and `y_val` – validation data points and class labels

that are ready to be used to estimate a model. There is also a normalization step of `x_train` and `x_val`, such that each feature has mean value 0 and standard deviation 1 in `x_train`. Do not forget to apply this normalization also when predicting!

4.2.2 *Exercises*

There is a skeleton notebook `intent_prediction.ipynb` that you can use as a starting point and the objective is for you to experiment with all hyperparameters to find a model and evaluate the performance. You do not need to elaborate with advanced neural networks, basic structures are sufficient. The problem to predict vehicle behav-

ior is not easy, so you can not expect 99 % accuracy in your model, but performance around 80 % is possible to achieve.

Exercise 4.14. Explain why the imbalanced data set is a problem and what would happen if no measures were taken, e.g., the resampling step implemented in the code skeleton?

Exercise 4.15. The command `summary(model)` summarizes the model and also prints out the number of trainable parameters in the model. For a chosen model, preferably your final choice of model, show the output of `summary(model)` and explain the number of parameters in the model.

Exercise 4.16. Experiment with all hyperparameters in the model, e.g.,

- number of resampling points in the balancing step,
- number of data points chosen for the validation data,
- number of layers and number of nodes in each layer,
- regularization parameters, e.g., dropout layers,
- learning rate (specified using the `lr` argument to the model class).

Plot the evolution of loss function and accuracy, for both the training set and the validation set, during training. You can expect results similar to what is shown in Figure 3.

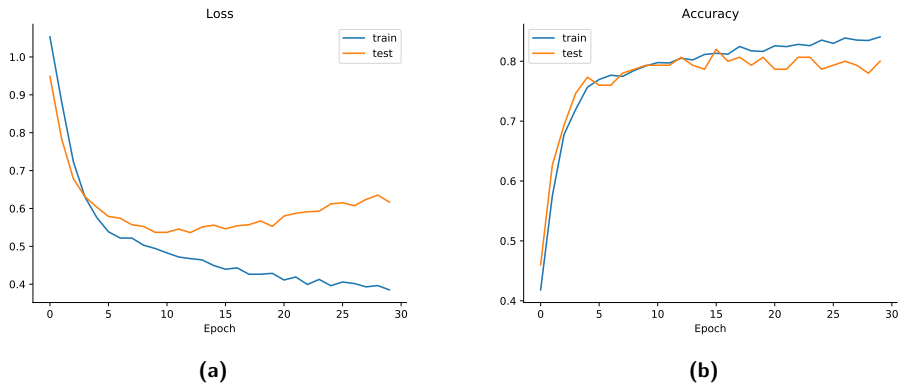

Figure 3: Loss and accuracy of training and validation data during training.

Report on your findings and comment on what the plotted curves mean.

Exercise 4.17. Compute a confusion matrix for the validation and training data using the function `sklearn.metrics.confusion_matrix` (already imported in the skeleton file). Explain the output and relate it to the prediction task at hand.

Based on the confusion matrix, there are other common performance metrics. Compute, for both train and validation data, *accuracy*, *precision* for each class, and *recall* for each class. The performance metrics for multi-labels are defined as

$$\text{accuracy} = \frac{\text{TP}}{n}, \quad \text{precision}_j = \frac{\text{TP}_j}{\text{TP}_j + \text{FP}_j}, \quad \text{recall}_j = \frac{\text{TP}_j}{\text{TP}_j + \text{FN}_j}$$

where n is the number of samples, TP the true positives (predicting the correct class), TP_j true positives for class j (predicting class j when class j is the true class), FP_j false positives for class j (predicting as class j when the true class is not j), and FN_j false negative for class j (prediction not j when true class is j).

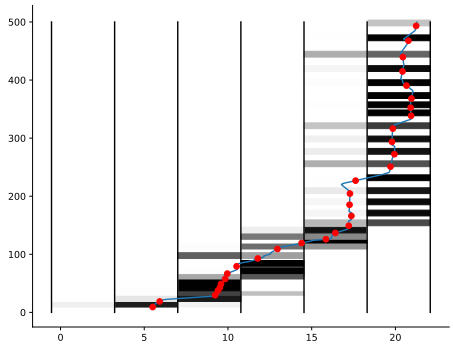

Figure 4: Lane prediction for a specific vehicle. Note that surrounding vehicles are not included in the figure. A darker color indicates probable lane during the next three seconds.

Computations of the performance metrics correspond to frequency estimations of the following three probabilities

$$P(y_j|\hat{y} = y_j) \tag{2a}$$
$$P(\hat{y} = y_j|y_j) \tag{2b}$$
$$P(\hat{y} = y) \tag{2c}$$

where $y_j \in \{0, 1, 2\}$ is a class label, $\hat{y}$ is the predicted class of the true label y. Determine which performance metric that corresponds to which probability.

Discuss the results and comment on how the performance metrics are affected by imbalanced data.

Exercise 4.18. Code to visualize predictions on a trajectory is included, resulting in plots like Figure 4. Predictions in the plot are made at the red dots and the corresponding colors represent predicted action during the next three seconds. Plot similar figures for trajectories in the validation set and explain what you see. Discuss strengths and weaknesses of your model.

Exercise 4.19. If you have experience with other machine-learning tools or methods, you are encouraged to compare the performance of your neural network with other classifiers. The package scikit-learn (`https://scikit-learn.org/`) has a number of classifiers out-of-the-box applicable to the data, e.g., support vector machines and random forest classifiers.

A EXTRA ASSIGNMENT

Deep Q-Learning for Highway Driving

In this exercise, you will explore a reinforcement-learning approach, Deep Q-Learning [6], to navigate in the simplified highway scenario illustrated in Figure 5. The objective

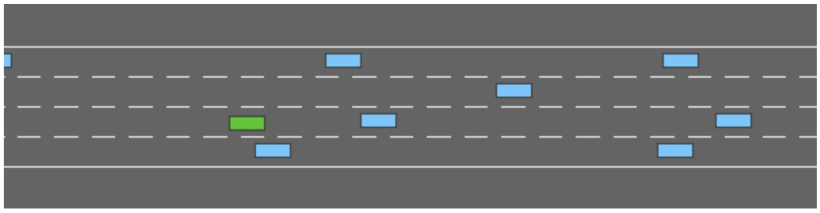

Figure 5: A snapshot from the highway scenario where the green vehicle is the ego vehicle and the blue vehicles are surrounding traffic.

of the exercise is to explore a model-free approach to reinforcement learning in a non-trivial task. The exercise will require very limited coding, and mainly aims at exploring and understanding existing code and models. There is pre-prepared code for all exercises. For more background on the approach, see the slides from Lectures 10–11, the reading instructions in Section 2, and the paper [9] (also [6] can be read for reference). The assignment is examined by submitting any relevant code you may have written and a short report with relevant plots and discussions.

The Highway Scenario — Actions, Observations, and Rewards

The exercise is based on the highway environment [3] for Open AI Gym[2]. The objective is to navigate fast and smoothly on the highway without any collisions using observations of relative positions and velocities of the surrounding traffic. The scenario will end if there is a collision *or* a maximum of 40 actions have been taken.

In the scenario setup used here, there are 5 discrete **actions** available:

$$\mathcal{A} = \{\text{lane\_left}, \text{idle}, \text{lane\_right}, \text{faster}, \text{slower}\}. \tag{3}$$

An **observation** is a 5-by-5 matrix where each row corresponds to a vehicle and the columns correspond to the features $(\text{presence}, x, y, v_x, v_y)$. The first row corresponds to absolute positions and velocities of the ego vehicle, and the four remaining rows correspond to four surrounding vehicles and their relative positions and velocities with respect to the ego vehicle (normalized). A sample state/observation can look like

$$s = \begin{pmatrix} 1 & 1 & 0.5 & 0.416 & 0 \\ 1 & 0.133 & -0.25 & -0.019 & 0 \\ 1 & 0.277 & 0.25 & -0.030 & 0 \\ 1 & 0.427 & 0.25 & -0.052 & 0 \\ 1 & 0.583 & -0.25 & -0.020 & 0 \end{pmatrix} \tag{4}$$

which means that, e.g., the surrounding vehicle 1, corresponding to the second row, has relative position and velocity

$$(\Delta x, \Delta y, \Delta v_x, \Delta v_y) = (0.133, -0.25, -0.019, 0).$$

Positions are normalized and a y value of 0 means center of the leftmost lane, 0.25 center of the the second leftmost, etc., until $y = 0.75$ for center of the rightmost lane.

The **reward** r is upper bounded by 1 and positive rewards are given for higher speed and driving in the rightmost lane, while collisions have a negative reward.

2 https://gym.openai.com

Exercises

Skeleton files are available in `extra.{ipynb/py}` and the Deep Q-learning agent is implemented in the file `dqn_agent.py`. The implementation is based on the DQN agent in [4] but is simplified for educational purposes.

Exercise A.1. In Section 4.1, Q-learning was implemented by direct tabulation of the state-action value-function $Q(s, a)$ where s is the state and a the action. This was possible because of a discrete and finite state space of limited size, which is not the case here since observation/state $s \in \mathbb{R}^{5\times5}$ is a continuous quantity.

One approach would be to discretize the state space. To illustrate why this approach is difficult here, assume that the 20 features (here ignore the 5 presence features) are discretized into 3 discrete values, e.g., {low, medium, high}. What is then the size of the Q-table?

Exercise A.2. Based on the observations in the previous exercise, one approach to handle the complexity is to make a continuous functional approximation of the Q-table with a deep feed-forward network $Q : \mathbb{R}^{5\times5} \to \mathbb{R}^5$. Note the slight difference here, instead of a function from s and a to a scalar Q-value, the network function is from a state $s \in \mathbb{R}^{5\times5}$ to $\mathbb{R}^5$, the Q-values for all 5 actions.

The architecture for the provided pre-trained model is shown in Figure 6. The model has 3 dense layers with ReLU activation functions and 256, 256, and 5 units in the respective layer. This results in a model with 73 753 parameters. Comment on the number of parameters in the model and relate to your answer in Exercise A.1.

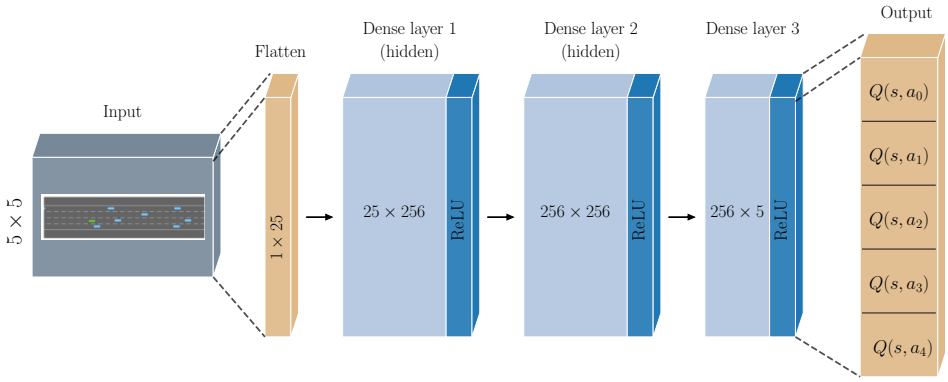

Figure 6: The neural network functional approximation of $Q(s, a)$.

Exercise A.3. The Deep Q-learning agent is fully implemented in the file `dqn_agent.py`. Study the class methods `train` and `update` and write down in pseudo code how the training loop works. In particular, note how there is not one but two neural networks interacting during learning. See the first page of [6] or the background in [9] for a description of the two networks and how they interact. Train the model for a few episodes, with scene rendering turned on, to further understand the process.

Training a well-functioning agent for this problem takes quite a long time, at least 4-5 000 episodes are necessary. Therefore, a pre-trained model is provided that has been trained for 8 000 episodes that you can use if you like. Figure 7 shows some statistics from a training process with multiple exploration phases. The cumulative reward is the total reward obtained during an episode and 40 is the maximal value since $r \leq 1$ and the episode stops at collision or 40 steps. The exploration/exploitation plots show the probability of taking a random action vs. using the current learned policy, the

ε-greedy approach. Comment on the plots, in particular how the cumulative rewards should be interpreted with respect to the exploration/exploitation profile.

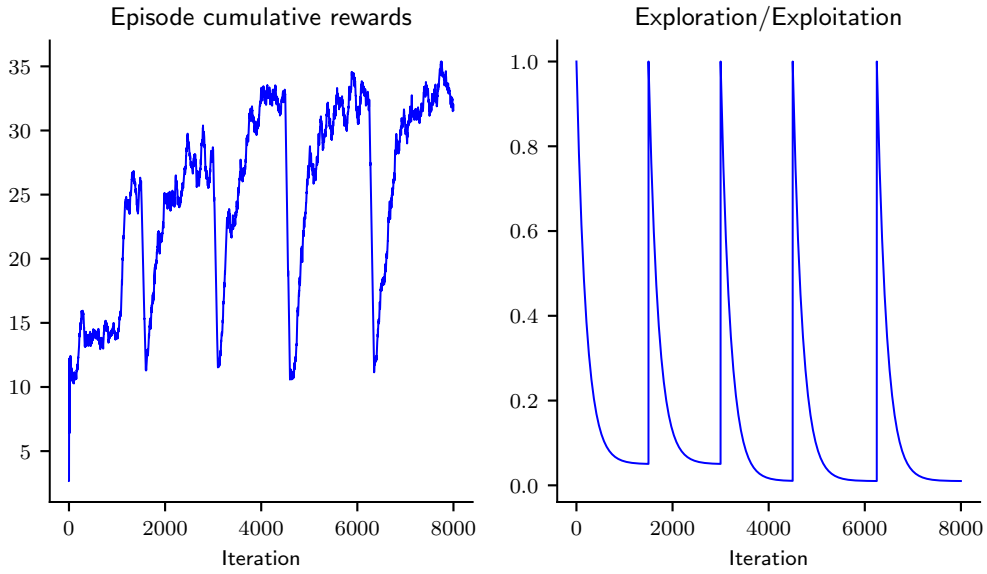

Figure 7: Cumulative reward and exploration/exploitation probability during training. The plots are averaged over 100 episodes.

Exercise A.4. This exercise will explore the pre-trained model and the obtained function $Q(s)$. First, generate an observation s from a random starting state and render the scene by executing

```
s = env.reset()
env.render()
```

Print the observation s and evaluate the pre-trained $Q(s)$. Interpret the results and explain which action that will be taken. The actions are ordered as in (3). Compare with the model from Exercise A.2 that has only been trained for a few episodes.

Design an observation by hand, for example one where there is no vehicle in front or an example with a vehicle in front but vacant space in neighboring lanes. Evaluate this particular observation on the pre-trained model, does the model act the way you expect?

Now, run the pre-trained model on a number of scenarios to get further understanding of how it performs. Increase the traffic density (see code) to increase the difficulty.

Exercise A.5. Until now, the state-action value-function $Q(s, a)$ has been explored and now we will investigate the value function $V(s)$ closer. In the scenario, the reward satisfies $r \leq 1$. Find an upper bound on $V(s)$ and $Q(s, a)$ as a function of the discount factor γ. What is this bound in this case, with a discount factor $\gamma = 0.8$?

Run a scenario with the pre-trained model and compute $V(s)$ for all visited states in the simulation. Plot the result and compare with the bound. The value $V(s)$ can be computed using the method available in the `Q` class.

Try different scenarios and select one that you find interesting. Comment on how the value of $V(s)$ can be interpreted.

Exercise A.6. This problem has to be considered rather simple and idealized compared to a real-world situation. Make a short reflection on the role of reinforcement-learning algorithms for autonomous vehicles, pointing out possibilities, challenges, and potential problems. Comment on ideas how to overcome possible issues.

REFERENCES

[1] Nemanja Djuric, Vladan Radosavljevic, Henggang Cui, Thi Nguyen, Fang-Chieh Chou, Tsung-Han Lin, Nitin Singh, and Jeff Schneider. Uncertainty-aware short-term motion prediction of traffic actors for autonomous driving. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, March 2020. https://openaccess.thecvf.com/content_WACV_2020/papers/Djuric_Uncertainty-aware_Short-term_Motion_Prediction_of_Traffic_Actors_for_Autonomous_Driving_WACV_2020_paper.pdf.

[2] Stéphanie Lefèvre, Dizan Vasquez, and Christian Laugier. A survey on motion prediction and risk assessment for intelligent vehicles. *ROBOMECH journal*, 1(1):1–14, 2014. DOI: 10.1186/s40648-014-0001-z.

[3] Edouard Leurent. An environment for autonomous driving decision-making. https://github.com/eleurent/highway-env, 2018.

[4] Edouard Leurent. rl-agents: Implementations of reinforcement learning algorithms. https://github.com/eleurent/rl-agents, 2018.

[5] Kaouther Messaoud, Itheri Yahiaoui, Anne Verroust-Blondet, and Fawzi Nashashibi. Attention based vehicle trajectory prediction. *IEEE Transactions on Intelligent Vehicles*, 6(1):175–185, 2021. DOI: 10.1109/TIV.2020.2991952.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. DOI: 10.1038/nature14236.

[7] Sajjad Mozaffari, Omar Y Al-Jarrah, Mehrdad Dianati, Paul Jennings, and Alexandros Mouzakitis. Deep learning-based vehicle behavior prediction for autonomous driving applications: A review. *IEEE Transactions on Intelligent Transportation Systems*, 2020. DOI: 10.1109/TITS.2020.3012034.

[8] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT Press, 2018. http://incompleteideas.net/book/the-book-2nd.html.

[9] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016. arXiv:1509.06461v3.

[10] Sai Yalamanchi, Tzu-Kuo Huang, Galen Clark Haynes, and Nemanja Djuric. Long-term prediction of vehicle behavior using short-term uncertainty-aware trajectories and high-definition maps, 2020. arXiv:2003.06143v2.