Practical Assignment for Knowledge-Based Control Systems (SC42050)

Introduction

This assignment based on MATLAB and Python is a compulsory part of the course Knowledge-Based Control Systems (SC42050). It will be graded and the mark counts for 20% in the final grade of the course (the exam grade is 60% of the final grade, and the literature assignment grade is 20%). The assignment is carried out in groups of two[1] students, and should take around 25 hours per person to solve, depending on your experience with MATLAB, Simulink, and Python. You can start signing up on **Monday 19 February 2018**. The assignment must be worked out in the form of a short written report (in English, one report per group), to be delivered along with the corresponding MATLAB/Simulink/Python software by **Wednesday 11 April 2018, 12:00 (noon) at the latest**. Please drop the report in the dedicated box placed in front of the DCSC secretariat[2]. Do not forget to include your names and student numbers on the title page of the report. Note that it is strictly forbidden to take over results from other students or make your results available to others.

Use MATLAB version 6.5 or higher and mention the version you used. Please include in your report complete listings of the MATLAB code (functions and scripts), Simulink models, and Python code that you developed for solving the assignment problems. In addition to that, please send your report in a PDF format and your software as a ZIP file by e-mail with the subject "SC42050 Practical Assignment submission" to the course assistant Thijs Greevink (**T.Greevink@student.tudelft.nl**).

Please post your questions on the Brightspace discussion forum[3]. On Wednesday 28 March 2018 from 15:45 to 17:30, there will be a question hours session for the Practical Assignment in IO-PC hall 1 (ENTER). You can bring your own computer here with you, or use the computers in the computer room.

The assignment consists of three problems. In the first one, you are asked to design in Simulink a fuzzy supervisor to improve the performance of a linear controller. The second problem concerns data-driven black-box modeling using a feedforward neural network. The third problem is based on reinforcement learning. For the first problem, each group will receive their own process simulation model.

To receive the files for the problems, send an e-mail to **T.Greevink@student.tudelft.nl** with the subject "[SC42050] Practical Assignment sign-up". Please mention the (first and last) names of the two members of your group, your student numbers, and your e-mail addresses.

Matlab programming

Strive for a compact and elegant MATLAB code, use functions where suitable, avoid loops (`for`, `while`, etc.) and also `if-then` constructs at places where you can easily use vector and matrix operations. Search for "vectorization" in Matlab help for helpful tips on the proper MATLAB programming style.

If you are unfamiliar with programming in Matlab, here are some pointers that should help you to quickly learn the basics. To access the Matlab documentation, type `doc` at the command line. A good place to start is the "Getting Started" node of the Matlab documentation. Focus especially on "Matrices and Arrays" and "Programming". A minimal knowledge of "Graphics" is required in order to present your results in a graphical form. For a more in-depth introduction, see "Mathematics" > "Matrices and Linear Algebra", and under this node: "Matrices in Matlab" and "Solving Linear Systems of Equations".[4]

[1] If you absolutely cannot find a partner, you may work alone. Note that groups of three or more students are not allowed.

[2] For students enrolled in TU Eindhoven or UTwente, it is not necessary to submit a printed copy.

[3] Students from TU Eindhoven or UTwente can send an email to the course assistant instead.

[4] These pointers assume the documentation structure in Matlab 7.3. While the the structure may vary in other versions, you should still be able to easily find these topics.

Problem 1. Fuzzy supervisory control

A nonlinear dynamic process is controlled by a linear controller. As the closed-loop behavior is not satisfactory, you are asked to design a fuzzy supervisor to improve the performance. Typically, the settling time and the overshoot of the closed-loop system must be maintained within specified limits for a given setpoint range. You will find the actual required values in the Simulink models you will receive.

The process model is a 'black-box', so you cannot inspect the equations, but you are allowed to carry out any open-loop or closed-loop simulation experiments in order to learn more about its properties. To design the supervisor, it is also possible (but not compulsory) to use the MATLAB's trimming and linearization functions `trim` and `linmod`. In the report, include the rule base, the membership functions and other parameters of the supervisor. Explain the rationale behind the rule base (inputs, membership functions, etc.) and the method you used to tune the parameters. Compare the closed-loop performance before and after introducing the supervisor. Discuss the results.

A set of functions implementing fuzzy inference in MATLAB and an example of using them within a Simulink model will be sent along with the assignment. You may use these functions, but it is not compulsory. Also if you think that your ideas cannot be realized with these functions, feel free to implement you own methods.

Problem 2. Bicycle rental prediction in TensorFlow

In this assignment you will be asked to predict the number of bicycles that will be rented out, based on weather and seasonal data. This will be done by training a neural network using `TensorFlow`, a computational library. A Python script is provided, to which you will make changes to improve the predictions.

Installing TensorFlow

The assignment requires `Python 3`, `TensorFlow`, and `matplotlib` to be installed. See the installation instructions on the respective websites, we provide some tips below. If you encounter any problems, Google is your friend, ask fellow students for help (this is encouraged for this step, and this step only), or post a question on the Brightspace discussion forum.

Intallation tips

Windows If you are using Windows, you might first have to install Python 3 first. Tensorflow requires the 64-bit version of Python (`Windows x86-64`). Unfortunately, the standard download page links to the 32-bit version. So download it from the version specific page. Using the option "Add Python 3.X to PATH" is recommended to be able to call `pip3` and `python` from any directory. Afterwards, open the Windows `Command Prompt` and use `pip3 install --upgrade tensorflow` to install the CPU version of TensorFlow and `pip3 install --upgrade matplotlib` to install matplotlib.

Mac
1. Download Xcode from the app store.
2. Install Homebrew by executing the following command in the terminal: `ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"`
3. Install Python 3: `brew install python3`
4. Use the steps here to install the *Python 3 version* of TensorFlow **with the following exception:** For **step *3***, use the command `python3 -m venv directoryOfYourChoice` instead. This will prevent issues with matplotlib.
5. install matplotlib with `pip3 install --upgrade matplotlib`

Linux Python 3 should already be installed. Use the instructions here to install the Python 3 CPU version of TensorFlow. Then use `pip3 install --upgrade matplotlib` to install matplotlib.

If you installed TensorFlow in a virtual environment, make sure it is still activated
`source YourPreviouslyChosenDirectory\bin\activate`
when running the exercise files.

Testing the script

The assignment comes with a Python script: `bicycle_predictor.py` and a dataset: `hour.csv`. There is also a `Readme.txt` file which describes the dataset.

Test if your TensorFlow installation was successful by running the script. This can be done by extracting the exercise files `bicycle_predictor.py` and `hour.csv` to a directory and using the following terminal command[5] form that directory:

`python bicycle_predictor.py`

You should get a list of training and validation errors per episode in the terminal as well as a plot resembling Figure 1. Close the plot to return to the terminal.

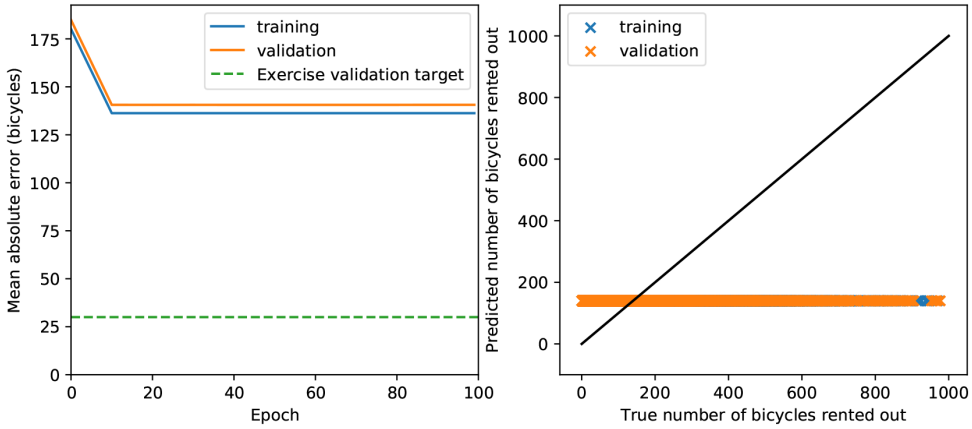

Figure 1: Expected output of `bicycle_predictor.py`

Improving the model

When the script is working, it is time to improve it. Open the `bicycle_predictor.py` file in your favorite editor and read through it to get a basic understanding of how it works.

Task 2.1 "Look ma, without any features!"

- On line `75` and above of the python script you can see that the loss is defined as the mean absolute error between the network predictions and the true number of bicycles (The regularization term is set to zero for now on line `8`). On lines `87 - 97` you can additionally see that the prediction of the network currently does not depend on any input, as the inputs are multiplied by zero. As a result, when the network is trained it learns to output a single value (the bias). To what property of the dataset does the output of the network converge?
- Change line `75` so that the mean *squared* error is used as the training criterion instead (and keep it like that for the rest of the assignment). To what property of the dataset does the output of the network converge now?

[5] Depending on your installation, the command `python` might default to Python 2, then use `py -3 bicycle_predictor.py` (Windows) or `python3 bicycle_predictor.py` (Linux & Mac)

Task 2.2. Linear regression

Change the `create_neural_network` function starting on line 84 such that the prediction of the number of bicycles is an affine function of the input features (by removing the multiplication with zero). Note down the validation performance (mean absolute error in bicycles) that you get. How would that performance change when adding additional linear layers or changing the number of units in the currently used layer? Explain your answer.

Task 2.3. Nonlinear regression

Change the function further to use a ReLU (Rectified Linear Unit) activation function for the hidden layer. Run the script and use the image it generates to discuss whether you think the model might be under-fitting or over-fitting. What does your conclusion mean for the number of training iterations and the model complexity? Should you increase or decrease them to get better validation performance?

Task 2.4 Going deep

Change the `create_neural_network` function further so that you get a neural network with two hidden ReLU layers. How does this change the performance?

You could also make the activation function in the final layer ReLU instead of linear. This would prevent the model from predicting that a negative number of bicycles is rented out. However, under some conditions this might prevent the model from learning anything. Why is that? And what would be an alternative to the ReLU activation that prevents this phenomenon?

Task 2.5. Hyper-parameters

At the start of the Python file several hyper-parameters are defined. Change these hyper-parameters and the neural network architecture such that the mean absolute validation error is around 30 bikes. Report and *motivate* the changes you make.

Optional Bonus question

Even though the neural network might predict the number of rented out bicycles fairly accurately on most days, it still makes large errors on other days. Use the `largest_data_point_errors` function of the `BicycleRentalPredictor` class to see for which dates your network makes the largest mistakes. Can you use the knowledge that the rental data is from Washington DC to explain one of these mistakes?

Problem 3. Reinforcement Learning

Most of the theory needed to answer the questions in this assignment can be found in the book "Reinforcement Learning: an Introduction" by Sutton and Barto (S&B), Chapters 1, 2, 3, 4 and 6. An online HTML version of this book can be found here: `http://incompleteideas.net/book/ebook/the-book.html`. You can also look at the lecture slides.

The goal of this exercise is to have a robot soccer player swing up a ball with its arm, even though the torque it can apply to its shoulder joint is not enough to do this in one go. This is called the "underactuated pendulum swing-up" problem. By answering the theoretical questions and implementing their solutions you will construct a temporal difference reinforcement learning solution to this problem using the tabular SARSA(0) algorithm.

The Matlab code for this exercise contains five main files:

`assignment.m`	Main function. Run it to learn and test your controller.
`assignment_verify.m`	Verification harness. Run it after implementing each question to verify your code.
`swingup.m`	Implements the SARSA learning loop described in S&B (Section 6.4, Figure 6.9), and a testing loop. The file is partly incomplete and your task is to complete it (search for TODO).
`swingup_initial_state.m`	Sets the initial state of the arm as a slightly perturbed bottom position.
`body_straight.m`	Simulates the dynamics of the body.

In this problem, you will go through questions and implementation tasks which eventually will lead the robot to perform an arm swing-up.

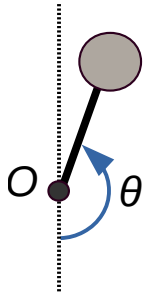

Figure 2: The angle θ is equal to 0 when arm is in the bottom position and is equal to π in the upright position.

Task 3.1. Understanding the code

Read `assignment.m` and `swingup.m`. Note how the `swingup` function can be used it three settings: learning, testing and verification. Compare the structure of the learning part to Figure 6.9 from the textbook.

a) How many simulation steps are executed in a trial?

Now run `assignment_verify.m`. This will report any basic errors in your code.

b) What does it report?

c) Find the source of the error. Why is this value not correct? Think about what it means in terms of the learning algorithm.

Task 3.2. Setting the learning parameters

Look at the `get_parameters` function in `swingup.m` and set the random action rate to 0.1, and the learning rate to 0.25.

a) Learning is faster with higher learning rates. Why would we want to keep it low anyway?

b) Set the position discretization such that there is exactly one state for every $\pi/15$ rad.

c) Assuming that the velocity stays in the interval $[-5\pi, 5\pi]\ \mathrm{rad\,s^{-1}}$, set the velocity discretization such that there is exactly one state for every $\pi/3\ \mathrm{rad\,s^{-1}}$.

Set the action discretization to 5 actions. Set the amount of trials to 2000.

Run `assignment_verify` to make sure that you didn't make any obvious mistakes.

Task 3.3. Initialization

The initial values in your Q table can be very important for the exploration behavior, and there are therefore many ways of initializing them (see S&B, Section 2.7). This is done in the `init_Q` function.

a) Pick a method and give a short argumentation for your choice.

b) Implement your choice. The Q table should be of size $N \times M \times O$, where N is the number of position states, M is the number of velocity states, and O is the number of actions.

Run `assignment_verify` to find obvious mistakes.

Task 3.4. Discretization

In Task 3.2, you determined the amount of position and velocity states that your Q table can hold, and the amount of actions the agent can choose from. The state discretization is done in the `discretize_state` function.

a) Implement the position discretization. The input may be outside the interval $[0, 2\pi]$, so be sure to wrap the state around (hint: use the `mod` function). The resulting state must be in the range $[1, \texttt{par.pos\_states}]$. This means that π (the "up" direction) will be in the middle of the range. See the pendulum model shown in Figure 2.

b) Implement the velocity discretization. Even though we assume that the values will not exceed the range $[-5\pi, 5\pi]$, they must be clipped to that range to avoid errors. The resulting state must be in the range $[1, \texttt{par.vel\_states}]$. This means that zero velocity will be in the middle of the range.

c) What would happen if we clip the velocity range too soon, say at $[-2\pi, 2\pi]$?

Now you need to specify how the actions are turned into torque values, in the `take_action` function.

d) The allowable torque is in the range $[-\texttt{par.maxtorque}, \texttt{par.maxtorque}]$. Distribute the actions uniformly over this range. This means that zero torque will be in the middle of the range.

Run `assignment_verify`, and look at the plots of continuous vs. discretized position. Are they what you would expect?

Task 3.5. Reward and termination

Now you should determine the reward function, which is implemented in `observe_reward`.

a) What is the simplest reward function that you can devise, given that we want the system to balance the pendulum at the top?

b) Implement `observe_reward`.

Run `assignment_verify`, and verify in the lower left plot that you have indeed implemented the reward function you wanted.

You also need to specify when a trial is finished. While we could learn to continually balance the pendulum, in this exercise we will only learn to swing up into a balanced state. The trial can therefore be ended when that goal state is reached.

c) Implement `is_terminal`.

Run `assignment_verify`, and verify that your termination criterion is correct.

Task 3.6. The policy and learning update

It is time to implement the action selection algorithm in `execute_policy`. See S&B, Sections 2.2 and 6.4.

a) Implement the greedy action selection algorithm.

b) Modify the chosen action according to the ε-greedy policy. Hint: use the `rand` and `randi` functions.

c) Finally, implement the SARSA update rule in `update_Q`.

d) Do we use eligibility traces in the `update_Q`?

Run `assignment_verify` a final time to check for errors. The result should be similar to Figure 3.

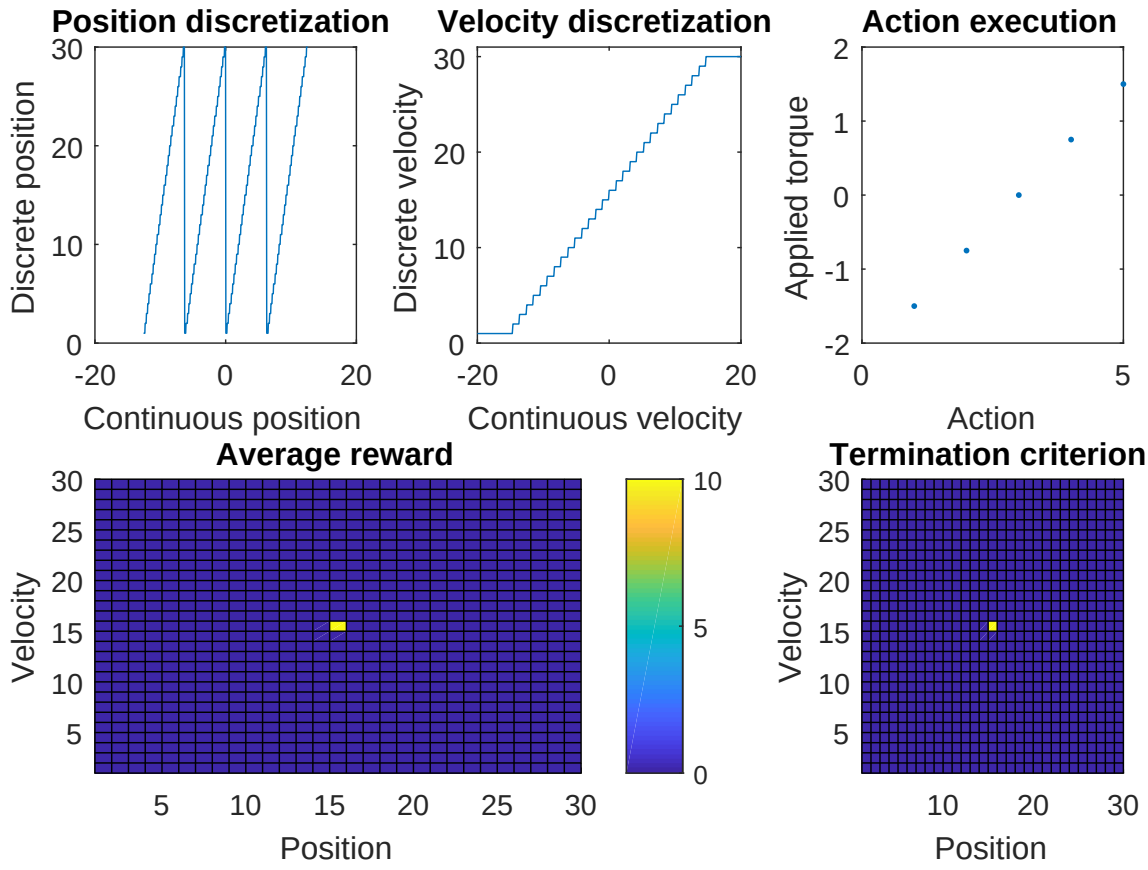

Figure 3: Output of `assignment_verify` after completing Tasks 3.1-3.6.

Task 3.7. Make it work

Finally, use Figure 6.9 from S&B and complete all the code of the learning section in `swingup` (initializations of outer and inner loops, calculation of torque, learning and termination). Basically you need to call all functions prepared in Tasks 3.3-3.6 in a right order. Also make sure that initial state is always slightly perturbed, i.e., that `swingup_initial_state.m` is used for initialization.

It is time to see how your learning algorithm behaves! Run `assignment.m` and check the progress. A successful run looks somewhat like Figure 4.

a) How many simulations steps on average does a swing-up take (after learning has finished)? Will it be wise to reduce the number of steps per trial during learning?

b) Large parts of the policy in the upper-right graph are quite noisy. What reasons can you name?

c) Test your code with greedy and ε-greedy policies. Which method allows the algorithm to converge faster and which method results in a higher cumulative reward (on average)? Explain the reason.

d) Try several values of discount rate, spanned across the $[0, 1]$ interval. What discount rate allows the algorithm to converge faster? Explain the reason.

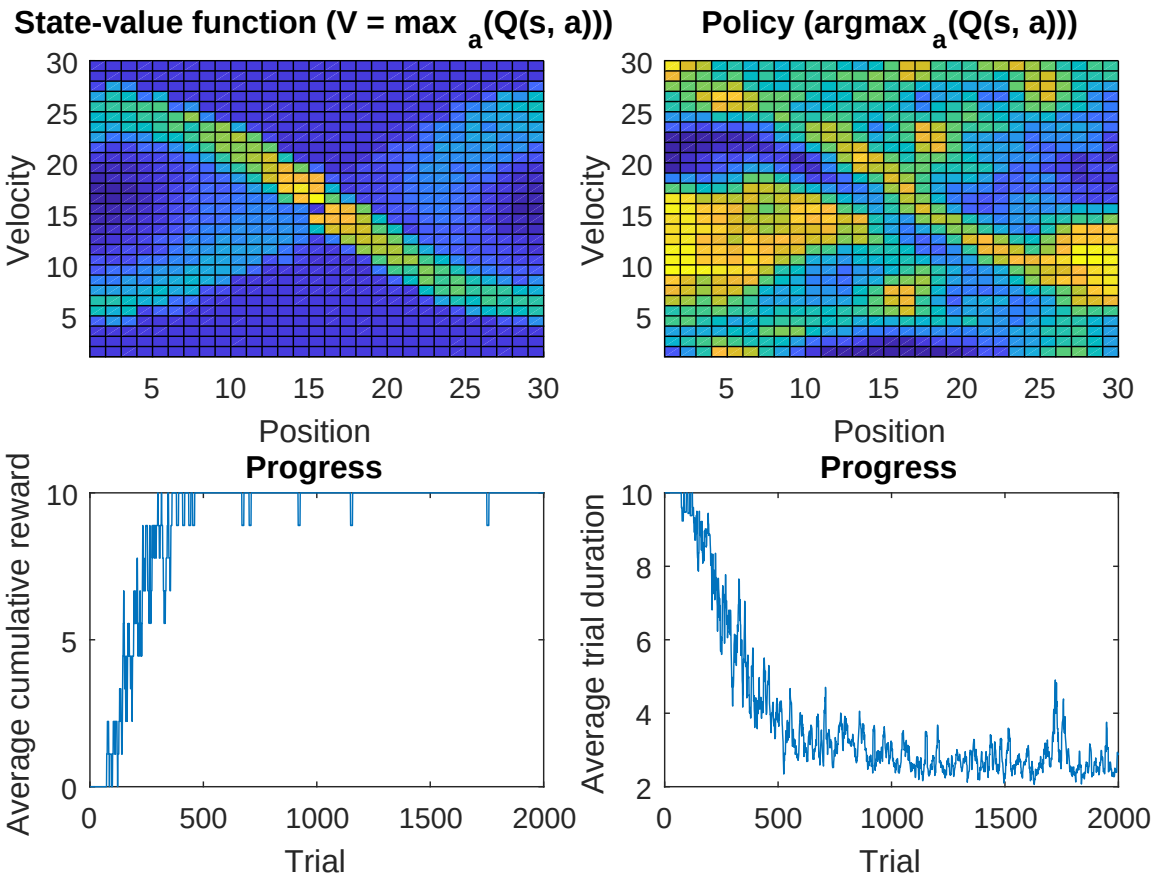

Figure 4: Output of a successful run of `assignment.m`