Matlab Assignment for Knowledge-Based Control Systems (SC42050)

Introduction

This MATLAB-based assignment 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 and Simulink. It 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 software by **Wednesday 12 April 2017, 12:00 (noon) at the latest**. Please drop the report in the dedicated box placed in front of the DCSC secretariat. 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) and Simulink models 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 to the course assistant Divyam Rastogi. His e-mail address, that you will need to obtain the files for the assignments as well, is **`d.rastogi@student.tudelft.nl`**.

On 29/Mar/2017 from 15:45 to 17:45, there will be a question hours session for the Matlab assignment in the computer room of Industrial Design, IO-Zebra 1. 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 of an unknown system by using a feedforward neural network. The third problem is based on reinforcement learning. For the first two problems, each group will receive their own process simulation model and their own data set.

To receive the files for the first two problems, send an e-mail to **`d.rastogi@student.tudelft.nl`**. Please mention the names and initials of the two members of your groups, the student numbers and an e-mail address at which you wish to receive the files.

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".[2]

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

[2] 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. Black-box data-driven modeling

Given is a data set measured on an unknown **dynamic** system with one or more inputs and one output. The order of the dynamics is not larger than three. Your task is to develop a black-box model for this system, using a sigmoidal neural network. A second data set measured on the same system is provided for validating the developed model. The two data sets will be given as a MATLAB data file, containing one structure for each set. The identification data set is named `iddata` and the validation data set `valdata`. Each contains two column vectors `u`, `y` of identical lengths holding the measured input and output, and the sampling time `Ts` used in collecting the data (e.g., `iddata.u`, `iddata.y`, `iddata.Ts`).

Report the one-step-ahead and the simulation root-mean-squared errors for both sets and show a representative plot for the fit on the training and the validation data sets. Discuss the results, including the quality of the model fit on the two data sets. Refer to Lecture 'Construction of Fuzzy Systems' (sheet 12) for the difference between one-step-ahead prediction and simulation.

It is recommended that you use the Neural Network toolbox of MATLAB to solve this problem. The `newff`, `train` and `sim` functions respectively create, train and simulate a feed-forward neural network. Use `help nnet`, `help newff`, etc. to get started with the Neural Network toolbox.

Problem 3. Reinforcement Learning

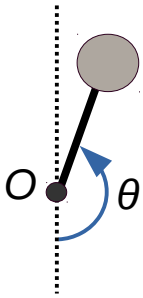
Most of the theory needed to answer the questions in this assignment can be found in the book "Reinforcement Learning: an Introduction" by Sutton and Barto (S&B), chapters 1,2,3,4 and 6. An online HTML version of this book can be found here: http://www.cs.ualberta.ca/~sutton/book/ebook/the-book.html. You can also look at the lecture slides available on Blackboard.

The goal of this exercise is to have our 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. 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. Feel free to discuss all unclear matters with other students or student assistant, if necessary.



The angle θ is equal to 0 when arm is in the bottom position and is equal to πin the upright position.

Task 1. Understanding the code

Read *assignment.m* and *swingup.m*. Note how *swingup* function can be used it three settings: learning, testing and verification. Compare the structure of the learning part to the 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 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 $\pi/15$ radians.
c) Assuming that the velocity stays in the interval $[-5\pi, 5\pi]$ rad·s^{-1}, set the velocity discretization such that there is exactly one state for every $\pi/3$ 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. 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 NxMxO, 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 4. Discretization
In Task 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, par.pos_states]. This means that π (the "up" direction) will be in the middle of the range. Refer pendulum model shown above.
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, 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 [-par.maxtorque, 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 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 6. The policy and learning update

It is time to implement the action selection algorithm in *execute_policy*. Refer 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 1.

Task 7. Make it work

Finally, refer the figure 6.9 from S&B and fill finish 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 3-6 in a right order. Also make sure that initial state is always slightly perturbed, that is use *swingup_initial_state.m* for initialization of a state.

It is time to see how your learning algorithm behaves! Run assignment5 and check the progress. A successful run looks somewhat like Figure 2.

a) How many simulations steps on average does a swing-up take (after learning has finished)? Will it be wise to reduce number of steps per trail during learning?
b) Large parts of the policy in the upper-right graph are quite noisy. What reasons of it 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 of it.
d) Try several values of discount rate, spanned across [0,1] interval. What discount rate allows the algorithm to converge faster? Explain the reason of it.

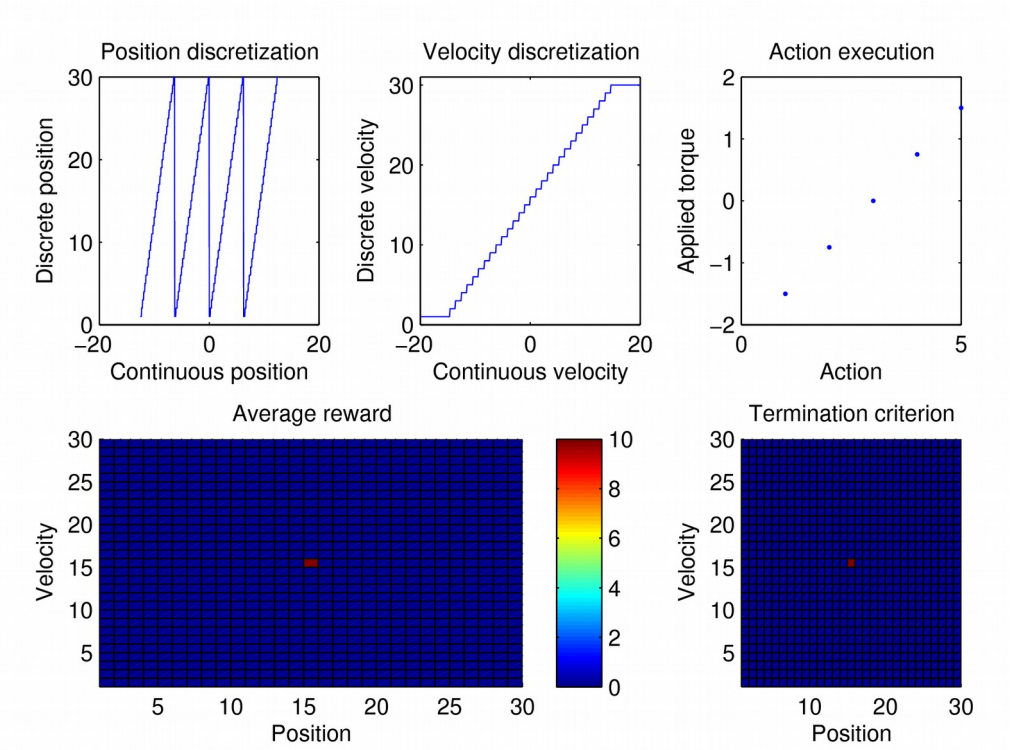


Figure 1: Output of assignment_verify after completing Tasks 1-6.

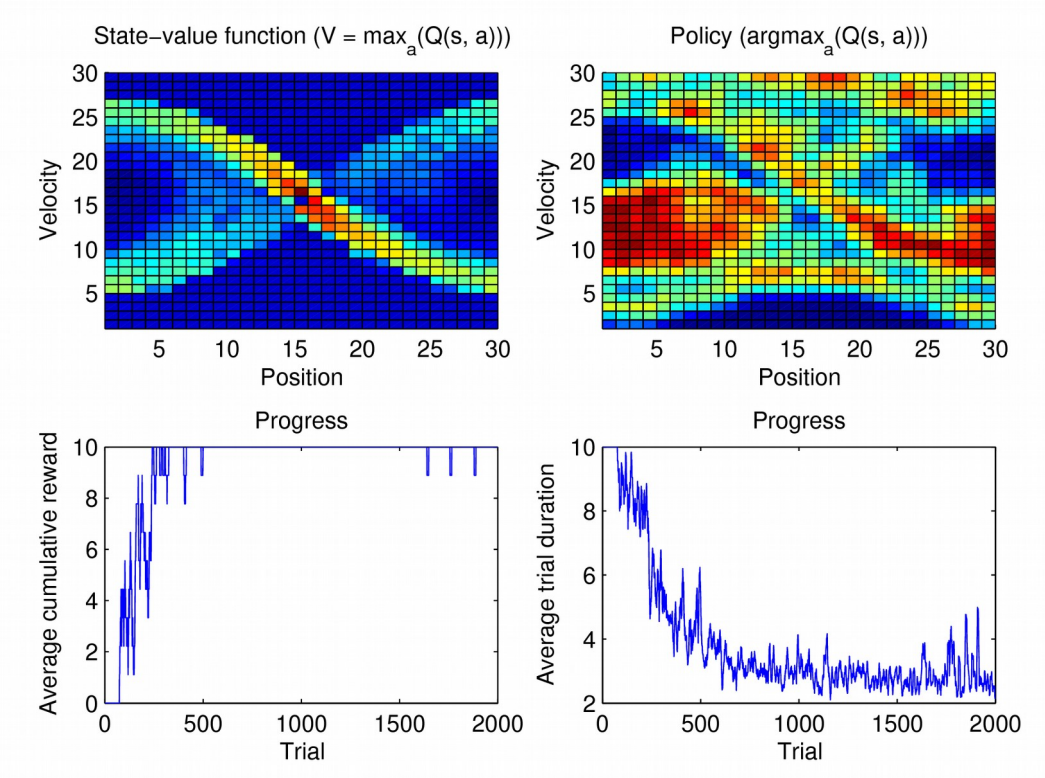


Figure 2: Output of a successful run of *assignment.m*