Practical Assignment for Knowledge-Based Control Systems (SC42050)

February 6, 2021

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 8 February 2021**. 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/Python software by **Wednesday 7 April 2021, 12:00 (noon) at the latest** via a BrightSpace assignment. Do not forget to include your names and student numbers on the title page of the report. The expected length of the report is ca. 10 pages (excluding title page and references, but including figures, tables, code snippets, etc.) with a maximum of 15 pages. 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 submit your report in a PDF format and your software as a ZIP file with the file names `GROUPID.pdf/zip`, where GROUPID stands for your group number. Please post your questions on the BrightSpace discussion forum. For organizational issues and questions that would be revealing the solutions, contact the course assistant Irene (**I.LopezBosque@student.tudelft.nl**). We will react to all questions received by March 24, 2021. We will only react to the most important questions received by March 31, 2021. We will not accept any new questions after March 31, 2021.

The assignment consists of three problems. The first problem concerns data-driven black-box modeling using a feedforward neural network. The second problem is based on reinforcement learning. The third problem is about model-based control.

You can get a total of 100 points (corresponding to a grade of 10) for this assignment. 10 points are for the quality of the report, the remaining 90 as indicated below.

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. Vision-based angle prediction (25 Points)

In this task, the goal is to train a neural network model that can predict the angle of a pendulum given pixel image observations. This will be done by training a neural network using the Keras API to `TensorFlow 2`, a computational library. In the ZIP file on Brightspace, a template Python script is provided, to which you will need to add your prediction models. You have several options of running the code. You run it either in an online environment or locally on your personal computer. Setting up the code to run locally is advised, because it will enable the option to see a "live" simulation of your trained prediction model. Furthermore, training times are likely less (depending on your hardware), especially when training the model that uses a convolutional neural network. Please state in your report whether you used the online or local setup.

Online setup `Google Colaboratory` has all the required components pre-installed. Click on the link and upload `train.ipynb` with `File-->Upload notebook`. You will need a Google account to save your work. Be sure to regularly check that your work is actually saved! You will also need to upload `dataset.h5`. See Figure 1 for instructions. Note that you will have to re-upload the dataset every time you restart the colab session. Proceed to add your code to the setup notebook. When you are finished, download your notebook as an `.ipynb`, and make sure to add it to your submission.

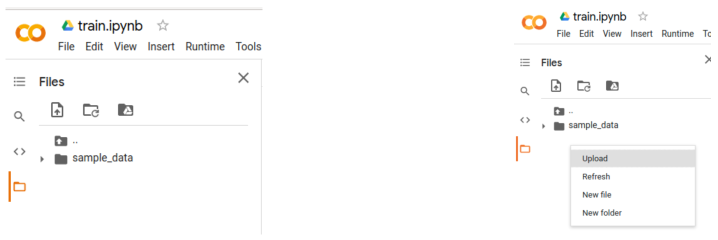


Figure 1: Click on the `Files` tab in the menu on the left image and upload `dataset.h5` there, alongside `sample_data`.

Local setup To set up the code locally, either download the ZIP file from Brightspace, or clone this `github repository` to your personal computer. Either use anaconda and `conda_env.yml` to setup a virtual environment or manually install the dependencies listed in `conda_env.yml`, to your local `Python 3`. More details on how to setup the code locally is provided in the `README.md` that can be found on the aforementioned github repository page (or inside the ZIP file on Brightspace). 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. In case you opt for the local setup, you can proceed to add your code either to `train.py` or to the Jupyter notebook `train.ipynb`. When you are finished, make sure to include this file in your submission.

Training data In order to train the model, we require data. Therefore, we simulate 120 trajectories of a swinging pendulum initialized at a random state that is driven by random actions and record the observations and state information. Every time-step the simulator outputs a downsampled RGB image with dimensions `28x28x3` as observation $\boldsymbol{o}$ together with the corresponding pendulum state $\boldsymbol{s} = (\theta, \dot{\theta})$. The angle θ is defined with respect to the upright position and wrapped to the $[-\pi, \pi]$ domain. $\dot{\theta}$ represents the angular velocity. Every trajectory consists of 100 state transitions, so if we include the starting state of the pendulum each trajectory is of length 101. This results in $N = 120$ trajectories that consist of $\{\boldsymbol{o}_i, \boldsymbol{s}\}_{i=0}^{T=100}$, which we concatenate into a single sequence and save as a single dataset. In the code, we split this dataset into a training, validation, and test dataset. We already provide a pre-generated `dataset.h5`. You can test whether your local setup is properly configured (e.g. dependencies, python version, etc...) by regenerating the dataset with `generate_data.py`. Note that depending on your hardware, it could take some time ($\pm$20 min) to generate the data. If you are working in google colab, you can skip this step and use the pre-generated dataset.

Warning! To help you, we have marked parts of the code where we want you to contribute. These parts are often marked with a start comment "Task 1.x: ..." and a closing comment "Task 1.x: END". You are free to add code outside of the designated areas. However, we cannot guarantee that this won't affect the intended behavior of the code.

Task 1.1: Learn to predict angles.

We are going to create a model M^θ that predicts the angle $\hat{\theta}$ given an image $\boldsymbol{o}$, so $\theta \approx \hat{\theta} = M^\theta(\boldsymbol{o})$. For this, we will use Keras within Tensorflow 2.3.1. The following references provide documentation that might aid you to build Keras models:

- Defining a sequential model: `https://keras.io/guides/sequential_model/`
- Training a model: `https://keras.io/api/models/model_training_apis/`

Create Create a sequential Keras model with `tf.keras.Sequential([...])` and define the following architecture,

- Start by flattening the input with `tf.keras.layers.Flatten()`.
- Then, add a fully connected hidden layer `tf.keras.layers.Dense(...)` with 128 units and ReLU activation.
- Finally, add a final fully connected linear layer `tf.keras.layers.Dense(...)` without activation. The number of units must match the dimension of our target data which is 1 as we try to predict the scalar angle θ.

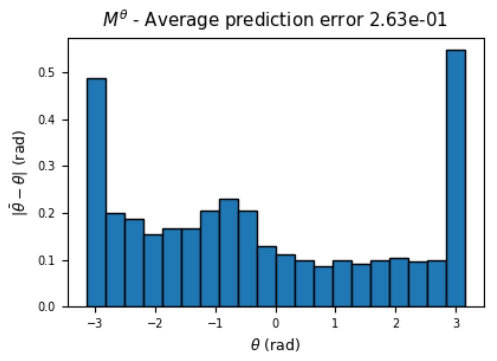
Compile Compile the model using `model.compile(...)` with the ADAM optimizer and a mean squared error loss. Use the default parameters for both.

Train Now that the model is defined, we can train it with `model.fit(...)` using (`train_obs`, `train_theta`) as the (input data, target data). Make sure that we have a train-validation split of 0.2 and a batch size of 64 that is shuffled every epoch. Train the model for 30 epochs with the aforementioned training data.

Run After implementing all the previous steps, run the code! The code will plot samples of your training dataset, train your model, and plot the model's accuracy. In case you locally run `train.py`, manually closing the 2 generated plots will start a live simulation of your prediction model.

Evaluate Evaluate the trained model's accuracy by analyzing the average error in the angle prediction. The provided code divides $\theta \in [-\pi, \pi]$ into 20 discrete bins and calculates the average error in the angle predictions per bin using the examples in the test dataset. Then, the "per bin average prediction error" is plotted. The "average prediction error" over the complete test dataset is also calculated and stated in the plot's title. See Figure 2 for two example plots.

- Run the code multiple times and record the "average prediction error" for each run. Calculate the mean and standard deviation of the "average prediction error" over all runs.
- Notice that the results vary across different runs. Sometimes the model learns to predict with an "average prediction error"< 1.0, while sometimes the model does not learn anything at all. Figure 2 shows two typical plots that qualitatively resemble both cases. How can the results be different, even though the underlying code and test dataset remained unchanged?



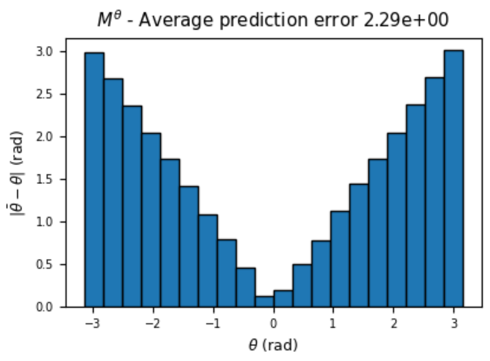


Figure 2: Two typical "per bin average prediction error" plots. **Left** Run where the model learns (to some extend) to predict the angle. **Right** Run where the model does not learn anything at all.

- Analyze the "per bin average prediction errors" plot of a "successful" run (i.e. a run with a plot that resembles the left plot in Figure 2). Where does this model have the lowest accuracy? What could be an explanation for the loss of accuracy in that region?
- In general, a separate test dataset is used to evaluate a trained model. Why?

Task 1.2: Indirectly predict the angles.

We are going to improve the accuracy by pre-processing the target data. Specifically, we are going to create a model M^{trig} that learns to predict $\sin(\theta)$ and $\cos(\theta)$ instead of directly predicting θ. Then, we can use the trigonometric relation[3] $\theta = \arctan(\frac{\sin(\theta)}{\cos(\theta)})$ to retrieve an estimate of θ.

Create Copy the model you created in Task 1.1. Change the number of hidden units of the final layer from 1 to 2. We do so, because we now want to predict two outputs $(\sin(\theta), \cos(\theta))$, instead of only one output θ.

Compile Compile the code in the same way as was done for M^θ.

Train Now that the model is defined, we can train it with `model.fit(...)` using (`train_obs`, `train_trig`) as the (input data, target data). Make sure that we have a train-validation split of 0.2 and a batch size of 64 that is shuffled every epoch. Train the model for 30 epochs with the aforementioned training data.

Run In order to run the code with M^{trig}, change `model_type` to `'model_trig'`.

Evaluate

- Run the code multiple times and record the "average prediction error" for each run. Calculate the mean and standard deviation of the "average prediction error" over all runs.
- Compare the "per bin average prediction error" plot for M^{trig} with the plot for M^θ. Why does indirectly predicting the angle improve the prediction accuracy?
- Why is it not sufficient to predict only $\sin(\theta)$ and use its inverse $\theta = \arcsin(\sin(\theta))$ to get an estimate of the angle?

[3]In practice you would use the "atan2" implementation, as the regular arctangent only covers $[-\frac{1}{2}\pi, \frac{1}{2}\pi]$

Task 1.3: Indirectly predict the angles with a convolutional neural network.

Instead of using a ‘vanilla’ fully-connected neural network, we are going to build a prediction model M^{cnn} that uses a convolutional neural network (CNN).

Create Again, create a sequential Keras model with `tf.keras.Sequential([...])` and define the following architecture,

- Start with a CNN `tf.keras.layers.Conv2D(...)` with 32 filters, kernel size of `3x3`, and ReLU activation.
- Then, add a max pooling layer `tf.keras.layers.MaxPooling2D(...)` with a pool size of `2x2`.
- Flatten the input with `tf.keras.layers.Flatten()`.
- Finally, add a final fully connected linear layer `tf.keras.layers.Dense(...)` without activation. The number of units must match the dimension of our target data which is 2 as we will predict the trigonometric functions again.

Compile & train Compile and train in the same way as was done for M^{trig}.

Run In order to run the code with M^{cnn}, change `model_type` to `‘model_cnn’`.

Evaluate

- Run the code multiple times and record the “average prediction error” for each run. Calculate the mean and standard deviation of the “average prediction error” over all runs.
- Make a comparison of the different models (i.e. M^{θ}, M^{trig}, M^{cnn}) based on the the prediction accuracy on the test dataset and the number of trainable parameters. The model.summary() function prints useful information about the model to the terminal. Which model would you prefer and why?
- If you change the activation of the last fully connected layer to ReLU, the prediction accuracy completely deteriorates. Why?

Task 1.4: Why are my results different over multiple runs?!

In the previous tasks, the prediction accuracy varied across different runs even though the underlying code and dataset remained unchanged.

Run Uncomment the `seed_experiment(...)` line and change `model_type` to `‘model_theta’`. Then, run the code to train model M^{θ} several times while having the `seed_value` be either 0 or 1. Don’t forget to comment the `seed_experiment(...)` line agian when working on the other tasks.

Evaluate

- For both seed values, run the code multiple times and record the “average prediction error” for each run. Then, calculate the mean and standard deviation of the “average prediction error” over all runs for each seed value separately. What do you observe?
- What is the benefit of seeding the pseudo-random generator in practice?

Task 1.5: Predicting the angular velocity.

If our goal is to control the pendulum, the angular velocity $\dot{\theta}$ is generally also required. However, temporal information cannot be extracted from individual images. Describe a strategy (architecture, data pre-processing, etc...) to estimate both the angle and angular velocity from images.

Problem 2. Reinforcement Learning (30 Points)

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 version of this book can be found here: `http://incompleteideas.net/book/the-book.html`[4]. 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.

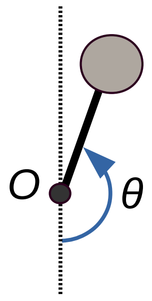


Figure 3: The angle θ is equal to 0 when arm is in the bottom position and is equal to π in the upright position.

Task 2.1. 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.

[4]The second edition does not allow for easy links to specific chapters, so the links in the remainder of the exercise are to the first edition.

Task 2.2. 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?

Set the action discretization to 5 actions. Set the amount of trials to 2000. Set the position discretization to 31. Set the velocity discretization to 31.

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

Task 2.3. 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 2.4. 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 $[0, 2\pi]$ rad, 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 rad (the "up" direction) will be in the middle of the range. See the pendulum model shown in Figure 3.

b) Implement the velocity discretization. Even though we assume that the values will not exceed the range $[-5\pi, 5\pi]$ $\mathrm{rad\,s^{-1}}$, 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 $[-2\pi, 2\pi]$ $\mathrm{rad\,s^{-1}}$?

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?

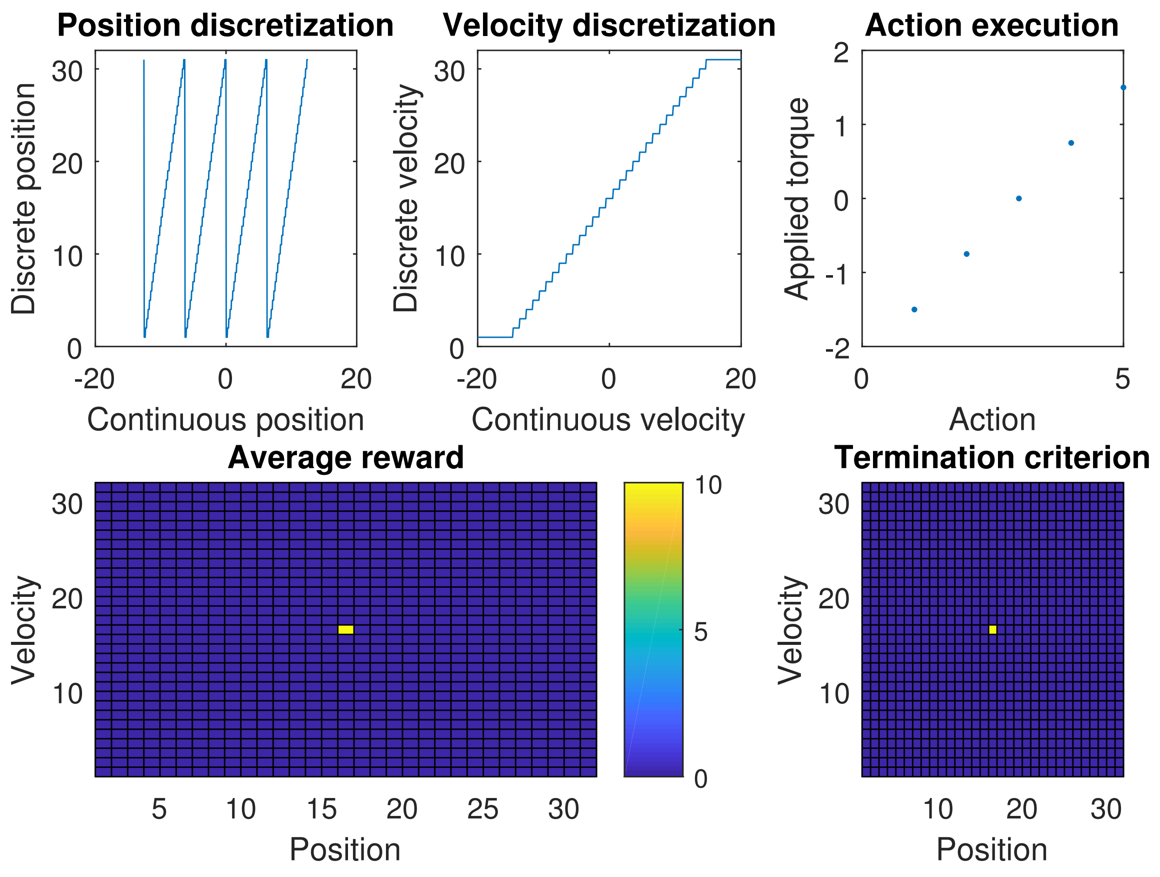


Figure 4: Output of `assignment_verify` after completing Tasks 3.1-3.6.

Task 2.5. 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 2.6. 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`.

Run `assignment_verify` a final time to check for errors. The result should be similar to Figure 4.

Task 2.7. 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 5.

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.

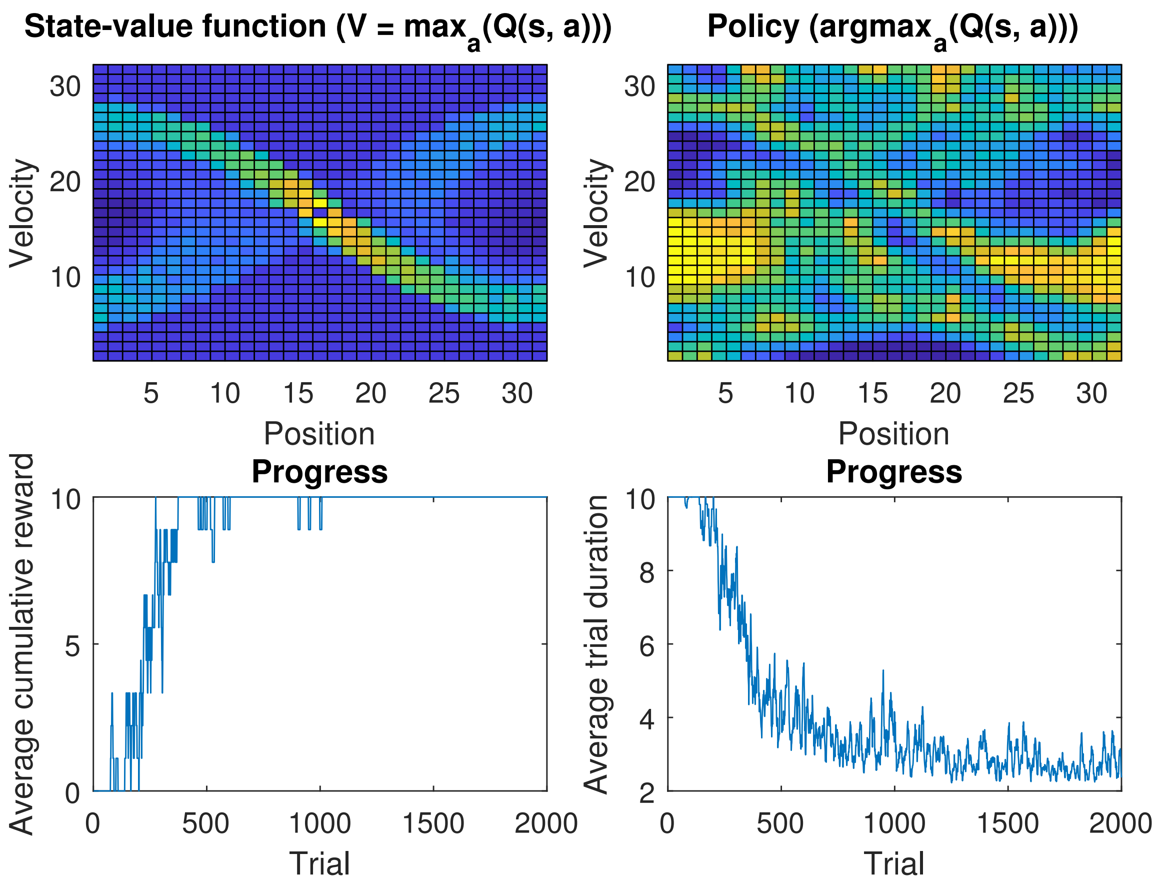


Figure 5: Output of a successful run of `assignment.m`

Problem 3. Model-based Control (35 Points + Bonus)

Your task is to design a model-based controller for a simulated 2 link robot arm that is tracking an ellipse.

Task 3.1. Warming Up

1. Run `controller_0.m`, `controller_1.m`, `controller_2.m`.
2. `controller_0` is a simple tracking PD controller on the joint level, try 3 other gain settings to improve the performance.
3. `controller_1` and `controller_2` use a model-based control approach (with the perfect analytical model). Note that the PD gains are a lot lower. There are subtle differences in how the model is used. Which control structures discussed in the lectures do they correspond to? Switch off the feedback (PD) in both controllers. What happens? Set the initial position to the desired initial position for both controllers. What happens? Do the effects of switching off the feedback and setting the initial position correspond to the properties of the controllers discussed in the lecture?

Task 3.2. Design your own Controller

The goal is to replace the analytical model in the feedforward part by a data-driven model (GP, neural network, fuzzy system, basis functions, etc.) or a qualitative one (naïve physics, knowledge-based, etc.). That is, you cannot make use of the physical equations and values of the analytical model. With feedback gains of `Kp = [500; 500]; Kd = [50; 50];` your model needs to get a lower RMSE than the pure PD controller as defined in `controller_0`; and all that for a range of the rotational velocity `tp.w` between 70 and 80, also see `controller_yours.m` and `controller_yours_evaluate.m`

For this evaluation only the feedforward model `ff_yours.m` can be modified (its input parameters are the current joint position and velocity, as well as the desired joint position, velocity and acceleration, *not* the current joint acceleration), the rest of the code (besides loading the model, variables, etc. and passing them to `ff_yours`) should remain functionally unchanged. For collecting data, training the model, etc. you can modify more things.

You can use any toolboxes you like, however, `controller_yours_evaluate.m` needs to be directly run-able on a standard TUD installation (`https://weblogin.tudelft.nl`) after unzipping.

Task 3.3. Bonus Points

You can get full points for the assignment without this task. With this task you can get bonus points to make up for points you missed, the maximum grade is still a 10. You can get up to 10 points for this task plus an additional bonus if your group is among the top 10 `RMSE x` scores (lowest RMSE of all groups gets 10 points, second lowest 9 points, etc. until tenth place 1 point).

The task is to make your controller robust to additional variations in the initial joint positions (we will evaluate a secret test set with deviations of up to ± 30 deg per joint compared to the desired initial position).