SC42050 – Knowledge Based Control Systems (2018-2019)

Report – Practical Assignment

Jorge Bonekamp
Student number 4474554
Master Systems & Control

Casper Spronk
Student number 4369475
Master Systems and Control

April 11th, 2019

Contents

*Matlab® R2018b was used all throughout this assignment.

Problem 1: Bicycle rental prediction in TensorFlow

In this section we will used the provided python script 'bicycle_predictor.py' to train a neural network in Tensorflow in order to predict the number of bicycles that will be rented out, based on weather and seasonal data.

Task 1.1

Initially, the loss is defined as the mean absolute error between the network predictions and the true number of bicycles. According to theory when we use the mean absolute error as cost function the output of the network should converge to the **median** of the dataset.
Now we change line 75 in 'bicycle_predictor.py' such that the mean squared error is used as the training criterion instead. According to theory the output of the neural network should converge to the **mean** of the dataset.

Task 1.2 Linear regression

Now we change *create_neural_network* function (line 84) such that the prediction is an affine function of the input features. We get the following results.

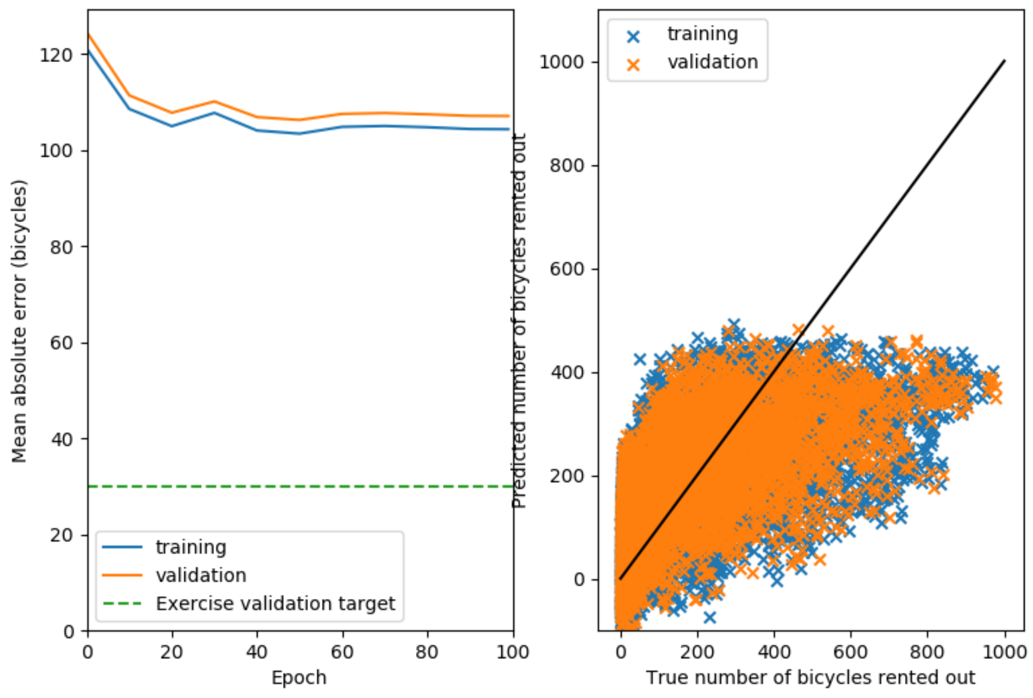

Figure 1 Neural network training

The neural network training converges to a mean absolute error of 104.32 bikes.

If we would increase the amount of linear layers in the neural network by one we get a slight increase in performance (to output of 102.18 bikes). However, if we add too many linear layers, overfitting of the data occurs and we get decreasing performance. The neural network converges slower and to only slightly better output relative to the additional computational power.

If we add extra units to the linear layers, we get only a slight increase in performance compared to the additional computational power.

1.3 Nonlinear regression

Next we changed the hidden layer to use a ReLU (Rectified Linear Unit) activation function. We get the following results.

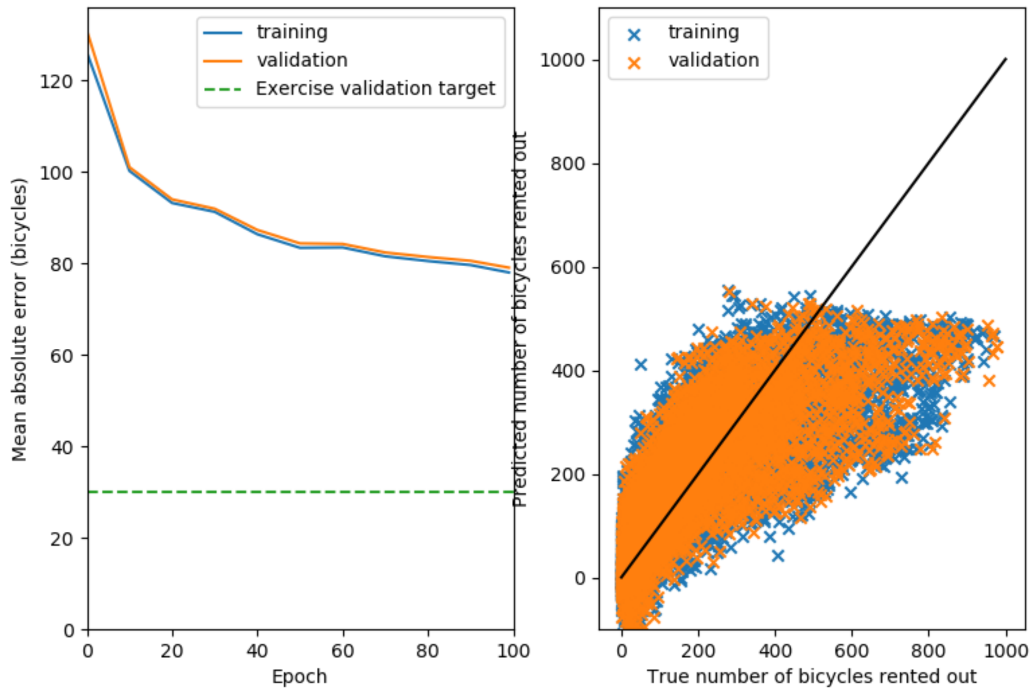

Figure 2 Training with ReLU activation function

Now the performance has increased to a training value of 77.99 bikes. We observe that the validation value is really close to the training value. Therefore, we conclude that the neural network is currently underfitting the system. This means we could increase the performance by increasing the model complexity and doing more training iterations.

Validation = 79.04

1.4 Going Deep

Next, we’ve changed the neural network such that we have two hidden ReLU layers. Now the training value has improved to a mean absolute error of 45.30 bikes.

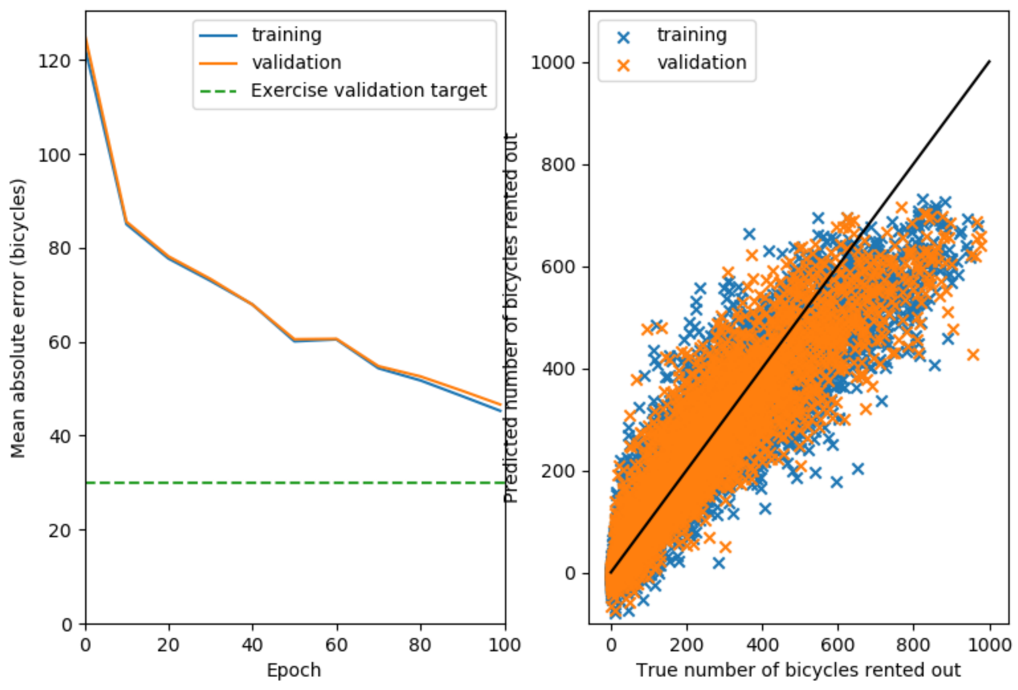

Figure 3 Training using two hidden ReLU layers

If we also change the prediction layer to use a ReLU activation function the mean absolute error doesn’t decrease at all. This may be because a large gradient may cause a unit in the prediction layer to be forever zero. Meaning that neuron will never activate another again, so that the network may stagnate. When we instead use a sigmoidal activation function in the prediction layer by definition the unit will never become zero and the network will not stagnate.

Task 1.5 Hyper-parameters

We increase the number of epochs to 200, (keeping everything else the same) this gives a mean absolute error of 35.62 bikes. Increasing the amount of epochs much further would result in a lot of extra computational effort for only a slight performance increase.
When we also change the batch_size to 32 instead of 64, we find a mean absolute error of 31.87 bikes. A smaller batch_size means that you update your gradients more often, therefore your neural networks is also updated more often and achieves better performance.

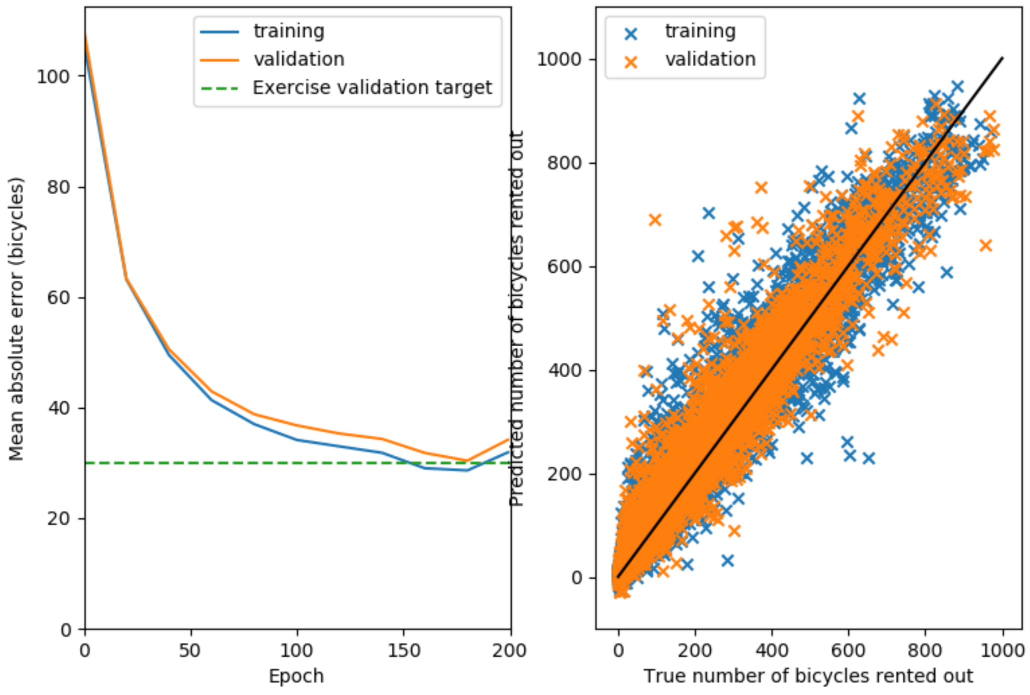

Figure 4 Training with now 200 epochs and batch size of 32

Bonus:

Using the largest_data_point_errors function we found that on the 23'th of March 2012 a large mistake in the absolute mean error was made. This can be explained by the fact that every year on that day there is a *Cherry Blossom Festival* in Washington DC, therefore more people may opt to rent a bike that festive day.

Problem 2:

Task 2.1

In every simulation we use a simulation time of 10 seconds, and a simulation step of 0.05. Therefore, in every trial 200 simulation steps are executed.

When running assignment_verrify.m it is reported that *the random action rate is out of bounds*, in other words, the value of epsilon is not between 0 and 1. The current value of epsilon is 0. This is incorrect because it will never allow for a random step to be taken. This will cause the learning algorithm to get stuck in a local optimum and therefore it is possible for it to never reach the global optimum.

Task 2.2

A slower learning rate may be desirable since setting your learning rate too high may cause you to 'overshoot' or pass the optimum, because our step size is too large. This way we avoid that we don't converge to the optimum at the cost of being slower.

In swingup.m we set the position discretization such that there is exactly one state for every $\frac{pi}{15}$ rad. Also we set the velocity discretization such that there is exactly one state for every $\frac{pi}{15}$ rad^-1 in the interval. We set 'par.pos_states = 30' and par.vel_states = 30'.

We set the action discretization to 5 actions and the amount of trials to 2000.

Task 2.3

We chose to initialize Q as a 30x30x5 matrix of all zeros. This is because the worst potential value that the Q function can have is 0.

Task 2.4

In this task we implemented the position and velocity discretization in 'swingup.m'.

If the velocity was clipped at a range that is to low, it may become impossible for the robot to reach the top of the arc, or it could possibly not use its max available torque.

Furthermore we altered the *take_action* function in 'swingup.m', such that the actions are uniformly distributed over the allowable torque range.

We ran *'assignment_verify' and got the following results.*

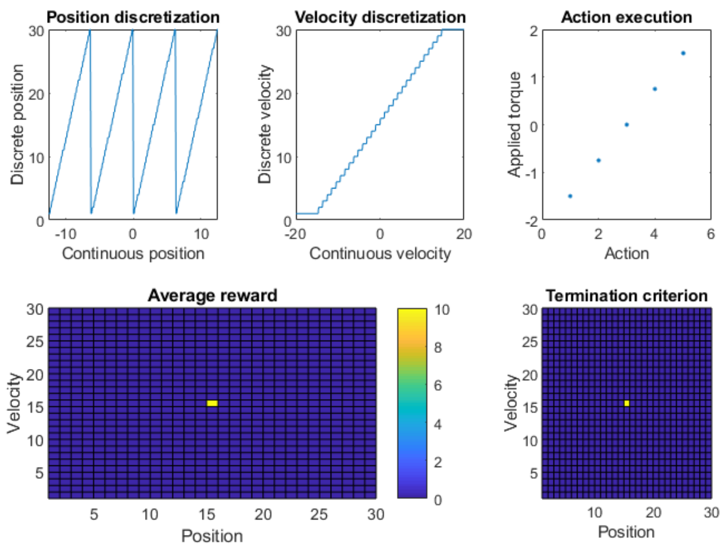

Figure 5 Assignment verification plot

Task 2.5

The simplest reward function would be one that gives no reward to all values that do not have a discreet velocity and position of 15 (velocity = 0 and position = pi).

Task 2.6

The epsilon greedy policy implemented takes either a random action when a random number generated is less than or equal to epsilon. If this is not the case, the function will return the action with the highest reward. Should multiple actions have the same reward, the function will pick on of these actions at random.

The update_Q function has the standerd SARSA update rule:$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - \ Q(s_t, a_t)$.

Task 2.7

Due to an error in one of the functions we were unable to get the arm to swing up to the correct position. We tried multiple reward functions but all were unable to get the arm to swing up and get enough positive rewards to get the arm to stop at state velocity = 0 and position = pi. As can be seen in figure 6, the algorithm does start to learn. However, increasing the number of simulations does not seem to result in an arm that does stabilize at velocity = 0 and position = pi.

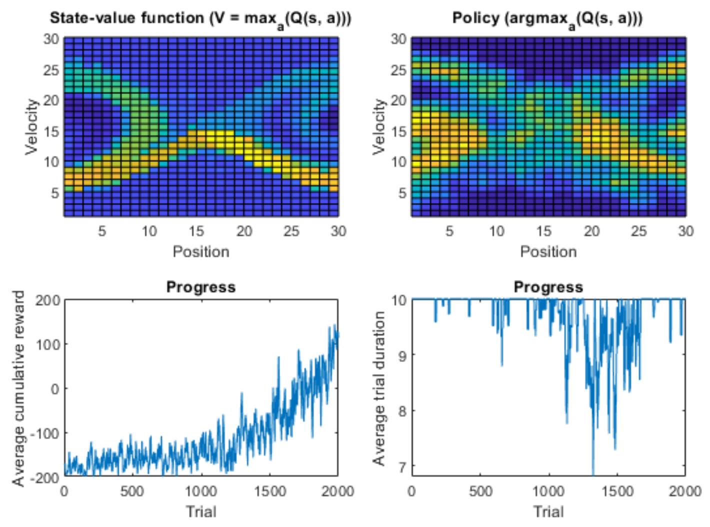

Figure 6: Final result obtained by running assignment

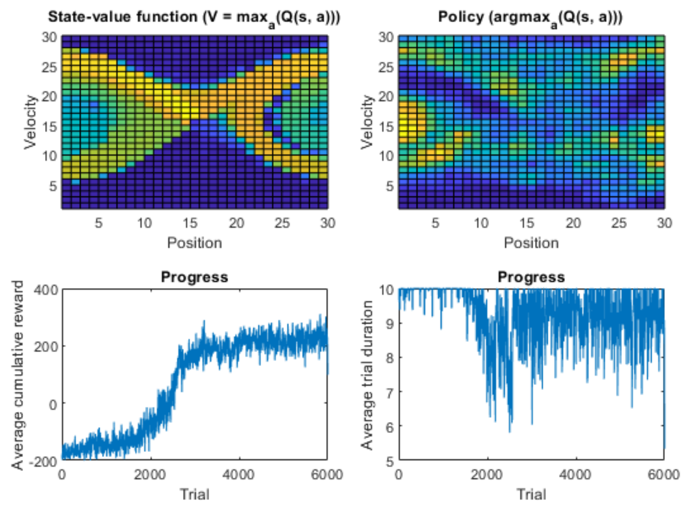

Figure 7: Final result obtained by running assignment.m with 6000 simulations

Reducing the number of steps is only a good idea if the algorithm is quickly getting good rewards. The algorithm needs enough time to test the various states it reaches and if it is unable to do so the trial would be wasted in the sense that it didn't learn anything from that attempt. We could test this by reducing the amount of time that each simulation takes.

The large parts that are quite noisy likely have to do with the arm rarely getting to that state. This will lead to the values of Q being relatively uncertain and therefore noisier than states of Q that the arm reached often.

Due to the algorithm not working properly we were unable to test our code for subtasks 2.7 c and d.

Problem 3:

Task 3.1 Warming up

In this task we design a model-based controller for a 2 link robot arm that is tracking an ellipse.

Firstly, we look at the provided controller0.m. To show the effects of the gains of the PD controller we plot the results for 4 different sets of gains. For the default gains we have the following trajectory.

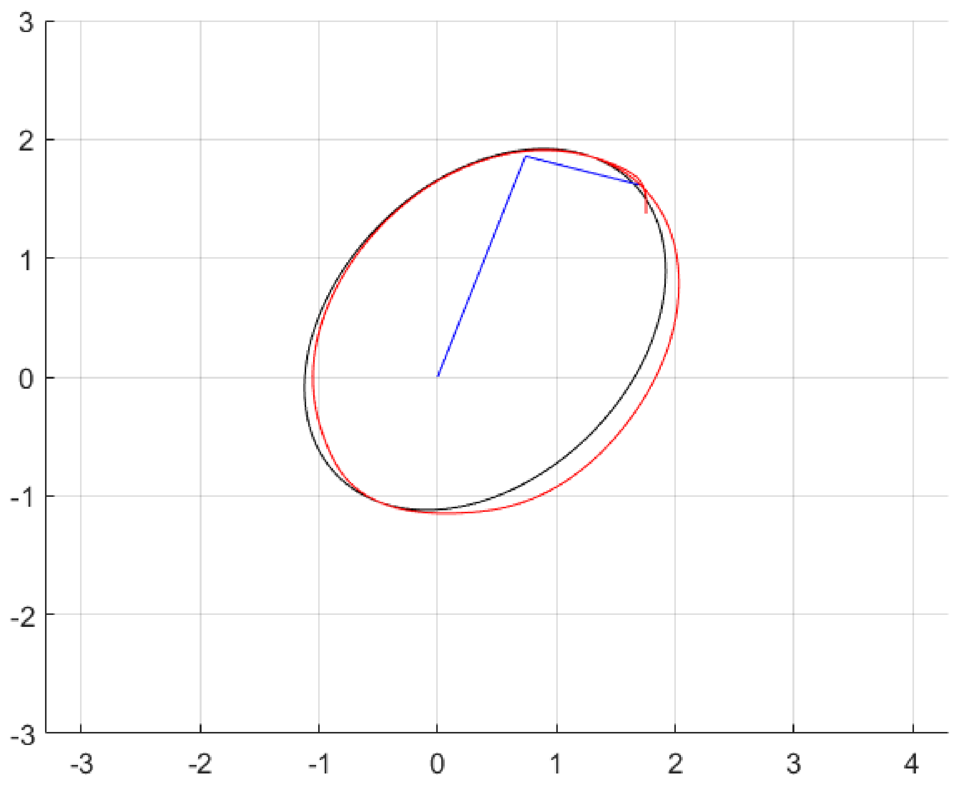

Figure 8 Trajectory for Kp = [2000,2000], Kd = [100,100].

If we now increase both proportional gains to 4000 we get the following trajectory.

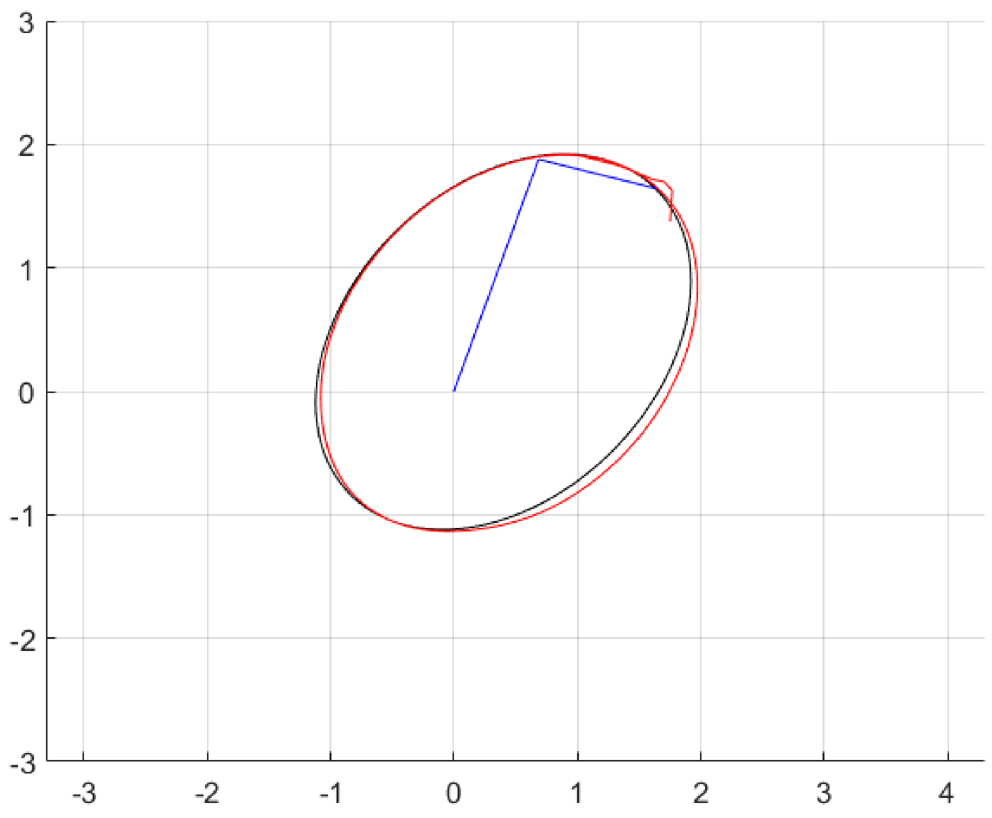

Figure 9 Trajectory for Kp = [4000,4000], Kd = [100,100].

It can be observed that the robot arm now follows the reference trajectory more closely, but there is a larger overshoot from the error in the initial position.

Actually the offset apparent from the first two figures is mainly caused by the too small first proportional gain on the larger link. If we now put Kp = [8000, 2000] we get the following trajectory.

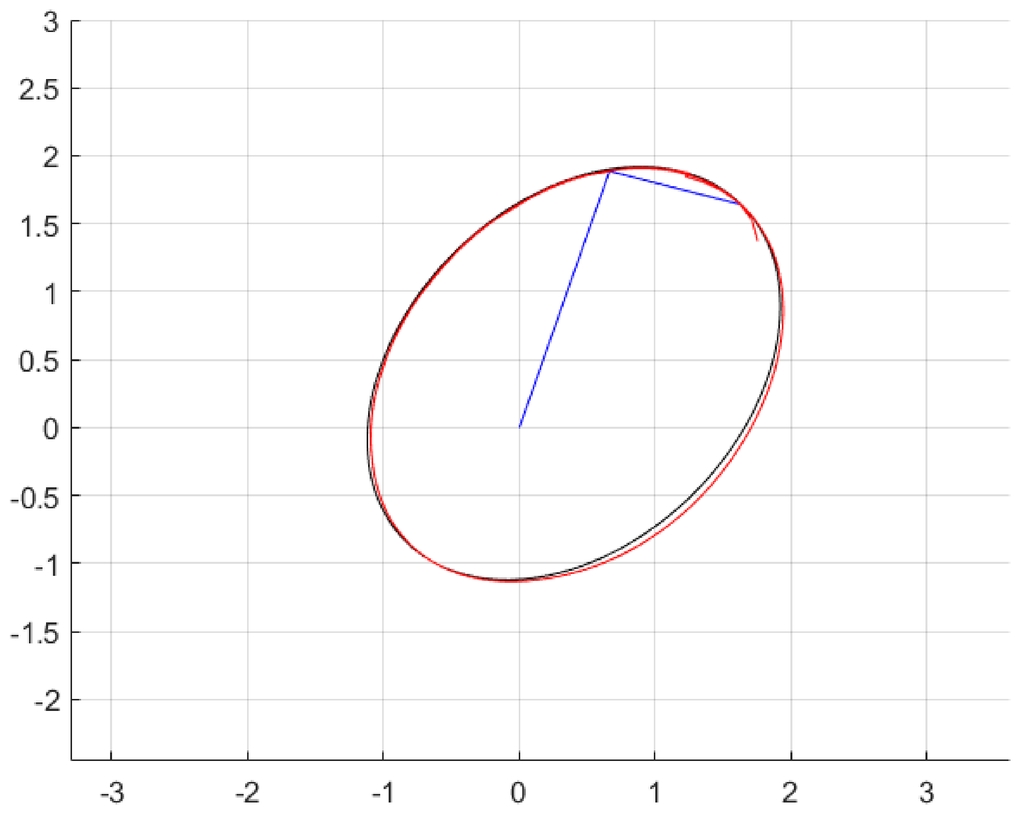

Figure 10 Trajectory for Kp = [8000,2000], Kd = [100,100].

The robot arm now follows the trajectory much more closely, however if we set Kp1 too high the system will become unstable.

The effect of the derivative gain is shown by leaving Kp to default and setting Kd=[150,150].

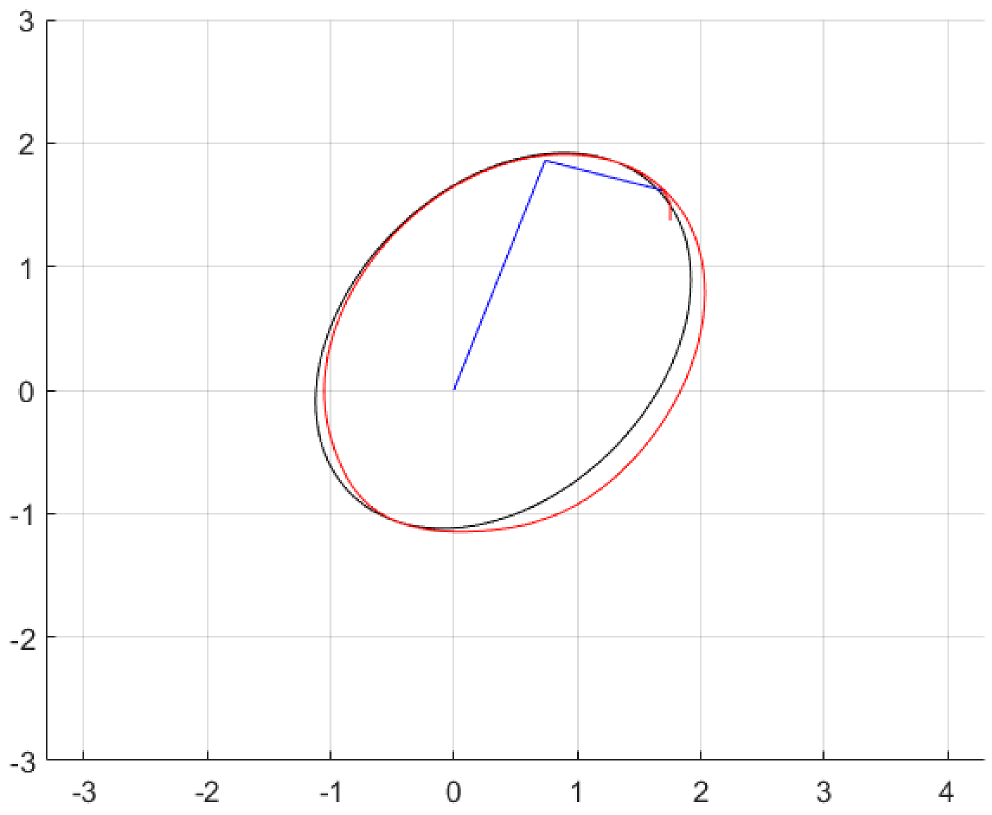

Figure 11 Trajectory for Kp = [2000,2000], Kd = [150,150].

Comparing this with the first figure we observe that increasing the derivative gain gets rid of some of the overshoot from the initial position error. However, if we increase Kd any further the system will quickly become unstable.

Next, we look at controllers 1 and 2, which use a model based control approach. Controller 1 uses a model based on current angle information, while controller 2 uses a dynamic model based on the desired angles information. Since controller 1 and 2 have access to the current and desired trajectory, it can create an inverse model control structure (which is done in ff_dyn_model(1&2).m).

If we now switch off the feedback in both controllers we get the following trajectories.

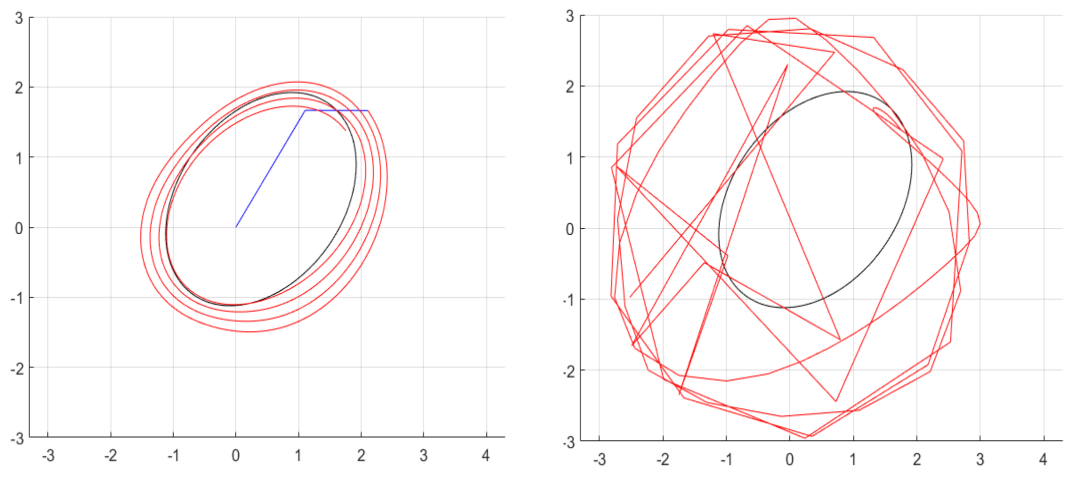

Figure 12 No feedback for controller 1 (left) and controller 2 (right)

Clearly, both systems are now unstable. Both systems have now no feedback control action on the error between current and desired position/angles. Since, the open-loop system is inherently unstable we get these unstable trajectories.

Now we set the initial position to match the desired initial position and keep no feedback. The systems still have an unstable response. The trajectories look the same as in the figure above except for the initial points now lay on the desired trajectory. This matches theory from the lectures since we know an unstable open loop system cannot be stabilized using inverse model control without feedback control action.

Task 3.2 Design your own controller

We now design our own controller, which is also based on an inverse model control structure. Except now we use the current angular values plus some additional term, to construct the inverse model RBD_matrices. The additional term is halve the difference between the desired and current trajectory. The controller was implemented in 'controller_yours.m' and 'ff_yours.m', we show the results by running 'controller_yours_evaluate.m' for several initial states.

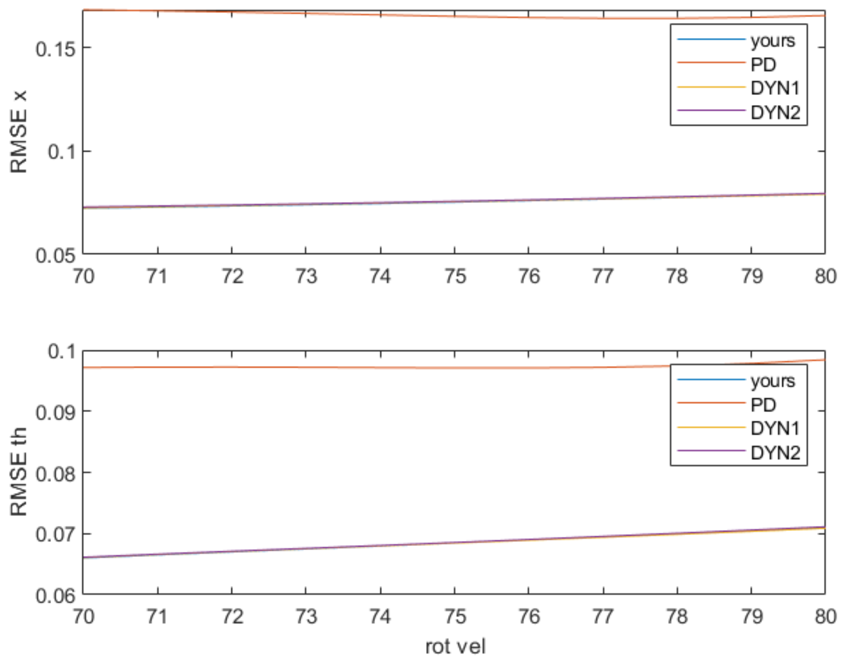

Figure 13 Controller evaluation for th_0 = th_des - [0.1; 0.2]

From the figure above, it can be seen that 'controller_yours' has nearly equal the performance of controller 1 and 2, for a initial angle bias of [0.1,0.2].

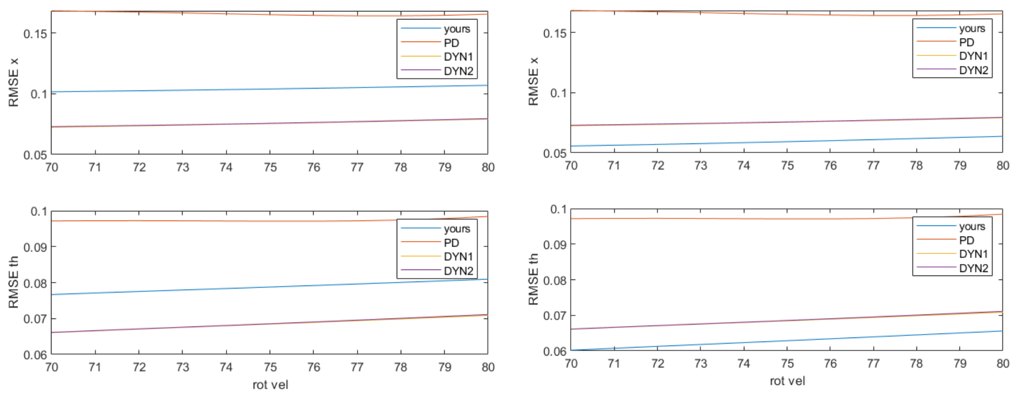

Figure 14 Controller evaluation for bias [0.2;0.3] (left) and bias [0;0] (right)

From this figure we conclude we get even better performance with 'controller_yours' if the initial angles equal the desired initial angles. However, our controller gets worse than controller 1 and 2 if we increase the initial bias beyond [0.1;0.2];