CHALMERS
UNIVERSITY OF TECHNOLOGY

Online Planning Based Reinforcement Learning for Robotics Manipulation

Master's thesis in Systems, Control and Mechatronics

INDREK KIVI

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2019

Master's thesis 2019

Online Planning Based Reinforcement Learning for Robotics Manipulation

INDREK KIVI

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
Gothenburg, Sweden 2019

Online Planning Based Reinforcement Learning for Robotics Manipulation
INDREK KIVI

Supervisor: Shahbaz Khader, ABB Corporate Research
Examiner: Yiannis Karayiannidis, Department of Electrical Engineering, Chalmers University of Technology

Master's Thesis 2019
Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Online Planning Based Model Learning for Robotics Manipulation
INDREK KIVI
Department of Electrical Engineering
Chalmers University of Technology

Abstract

There is a growing interest for adding learning capabilities to industrial robots and thus reducing the demand for tedious programming. Future autonomous robots are envisioned to learn autonomously complex tasks. In this thesis we consider a task of inserting a peg into a tight-fitting hole. Due to a lot of contact forces involved, the dynamics of this process is difficult to model. A model-based reinforcement learning method is used to overcome this problem, i.e. first the dynamics is learned based on observations of actions and resulting states, then an optimal control policy is computed based on the learned model. For model learning an artificial neural network is used while the control actions are calculated online, using model predictive control. During each trial, state-action data is collected and after a fixed number of trials the model is relearned using the improved dataset. Several iterations of relearning the model and collecting new data are done until the task is completed successfully.

We explore three different approaches to optimization. In particular, a gradient descent based optimization is compared with stochastic optimization method called random-sampling shooting method, and a more sophisticated version of the latter, a cross entropy method (CEM). Evaluation is performed on simulations of low-dimensional cart-pole task and an insertion task with 7-joint robot arm. While in the simpler task all mentioned optimization methods achieve similar results regarding the success rate, CEM proves to the best in the insertion task. This is illustrated by its high data efficiency and high robustness regarding the success rate. It is able to achieve 100 % success rate after 6 iterations, each containing 5 trials, while the computation time at each time step stays below critical 50 ms required for a real robot.

Keywords: robotics, model predictive control, optimization, model-based reinforcement learning, neural networks.

Acknowledgements

I am glad that I could work with this exciting project and grateful to the ABB Corporate Research in Västerås for this opportunity. I am especially thankful to my supervisor, Shahbaz Khader, for offering me a part in his project and his guidance throughout. He was very patient in explaining me anything, whenever I had problems and his advice was valuable to keep me on track. I appreciate his interest not only on the results of my experiments, but also on my whole thesis process. Finally, I thank my examiner Yiannis Karayiannidis for his constant support. His feedback was essential to shape this report to its final form.

Indrek Kivi, Tallinn, August 2019

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background

The use of robots for assembly tasks has long been practiced in the industry. A typical task for a robot is to connect two mating parts. This is usually achieved by programming the robot to move between manually taught points with desired speed. However, recent advances in robot skill learning have made it possible for a robot to learn the best control policy itself.

The goal of skill learning is to find a control policy i.e. a strategy that defines the way of doing a task. The trajectory emerges from the execution of the learned skill and is not manually specified. We use a skill learning approach, called reinforcement learning (RL), where the policy is autonomously learned by the robot by a trial and error approach. In this case, no correct way of doing a task is given, but achieving the desired objective is rewarded [1]. Reinforcement learning has been widely successful in playing computer games and logic games. For example, Google Deepmind's program AlphaGo defeated the world champion in a game of Go in 2016 [2]. Using RL in robotics is more challenging due to high-dimensional systems and impossibility to observe the environment with full accuracy [3]. However, promising results have been achieved with numerous simpler tasks and simulations [4] [5].

There are two different approaches to RL: model-based and model-free method. In the case of model-free RL, the agent (controller) directly learns a policy, i.e. it chooses the best action to take (action that minimizes cost) based on the information it has acquired during the trials. Model-based method uses the acquired data to learn a model of the environment (dynamics) instead. In both cases the goal is specified by a user defined cost function. Model-based controllers are known to be able to learn a policy with fewer samples than model-free controllers [6] [7]. Since obtaining a large dataset of samples with real robots is difficult, we use model-based control in the thesis.

The optimal policy is obtained either by online optimization that searches for the best control action at every time step or by directly optimizing a parameterized policy offline. In this thesis, we prefer online optimization method, called MPC. The optimal policy is recalculated at every execution step, instead of using a policy calculated offline. That allows to always use the most recent state information, but also adds a significant computational burden to the controller. Another advantage of MPC is that it allows to include constraints to the states and input.

The objective of the thesis is to study different aspects of MPC based RL method while evaluating it on a challenging robotics manipulation task of assembly.

The effectiveness of the method is demonstrated on a simulated peg in hole task being performed by an ABB YuMi robot. The main contribution of this thesis is the comparison of different online optimization methods using learned forward dynamics model.

1.2 Related work

There have been numerous attempts to use reinforcement learning for control of manipulators. Kober et al. [4], for example, demonstrated a robot learning to hit table-tennis ball, while Deisenroth et.al. [5] used model-based learning to stack small blocks on top of each other.

A problem with reinforcement learning is data inefficiency, i.e. to learn to complete some task, a lot of trials are generally needed. Having some prior knowledge about the system helps to reduce the amount of data needed. One skill learning approach is imitation learning, where the user first demonstrates how to complete the task and the robot learns to mimic the user, using the data it collected while observing the user's demonstration. Pastor et al. [8] showed a robot learning to imitate human in simple manipulation tasks. Regarding more difficult contact-rich tasks, Tang et al. have performed a task of inserting peg into hole, using learning based on human demonstration.

Alternatively, model-based reinforcement learning is found by many researchers to manage with little data, even when there is no prior information. Deisenroth and Rasmussen use model-based algorithm called PILCO, which shows high data efficiency, however, only with tasks of low dimensionality [6]. Levine et al. present a method that can solve a wide range of contact-rich manipulation tasks, using model-based learning [7]. They learn a time-varying Gaussian mixture model and use it to optimize an LQ-controller. They conclude that the method can generalize well to changes in the task and can learn to solve the problems with only a few real-world samples. However, this method used only locally linear dynamics model and has to train an ANN policy as well. In our approach we do not perform linearization of the dynamics and also completely avoid learning a policy model.

Approaches have differed on question of how to model the dynamics. One choice has been to use probabilistic models, which use probability to take into account the uncertainty of dynamics. Afore-mentioned PILCO for example uses Gaussian processes(GP) as a probabilistic model, same method is used by Kamthe and Deisenroth [9]. The tasks that they solve are however low-dimensional (inverted pendulum, double pendulum, uni-cycle). According to Calandra et al. the standard GP has difficulties in case of highly non-linear functions [10].

Another choice is to approximate the dynamics by artificial neural networks (ANN) model. Fu et al., for instance, use NN model with complex manipulation tasks [11]. Nagabandi et al. have achieved impressive results with high-dimensional and contact-rich locomotion tasks using deterministic neural network models [12]. The problem with ANN can be that deterministic ANN models do not consider uncertainties, which often lead to model errors, due to overfitting when the amount of data is small. Chua et al. propose a method to include uncertainty in the ANN model [13].

Regarding policy optimization, a number of papers dealing with RL in robotics have suggested using MPC [9] [12] [14] [11]. MPC offers a way to handle state and control constraints [9] and the limited horizon of MPC helps to avoid the problem of model error accumulation [14].

We base our work mainly on the paper by Nagabandi et al. [12]. Similarly to them, we use deterministic neural network models to learn dynamics and an MPC approach to optimize the control. We attempt to reach to similar results with a different task. While they do locomotion, trying to move different legged robots in a certain direction, we do insertion with a 7-joint robot. In addition, we compare different optimization methods, particularly gradient based optimization and variants of stochastic optimization, to improve the computation time and performance of the controller.

1.3 Structure and approach

In the thesis we show our method of applying reinforcement learning for insertion task. Three methods for doing MPC optimization are considered, one is based on gradient descent, another is a stochastic optimization method called random-sampling shooting. The third is a cross-entropy method, which is an inproved version of the standard random-sampling. To evaluate our method, several experiments are done. First, we will show the results with the simulation of a simple cart-pole balancing task. Then simulation of YuMi model is shown for insertion task.

Regarding the structure of the thesis, first the problem is formulated in detail in Chapter 2. In Chapter 3, some theoretical background is given for the techniques utilized in the thesis. In Chapter 4, the method implemented by us, for controlling the robot with reinforcement learning, is thoroughly explained. Finally, the results of the experiments are shown in Chapter 5. All of this is concluded with a short discussion and suggestions for future work in Chapter 6.

2

Theoretical background

In this chapter, theoretical background is given for some methods used in the thesis. First, reinforcement learning is introduced and difference between model-free and model-based approach given. Then a control method, called model predictive control, is explained. Finally, it is shown how model learning can be used to estimate models, with special focus on neural networks.

2.1 Reinforcement learning

A reinforcement learning (RL) problem can be modelled as a Markov decision process (MDP). We can define MDP as a tuple $(\mathcal{S},\mathcal{U},\mathcal{T},\mathcal{C})$, where $\mathcal{S}$ is a state space and $\mathcal{U}$ is an action space, $\mathcal{T}$ is a transition function and $\mathcal{C}$ is a cost function.

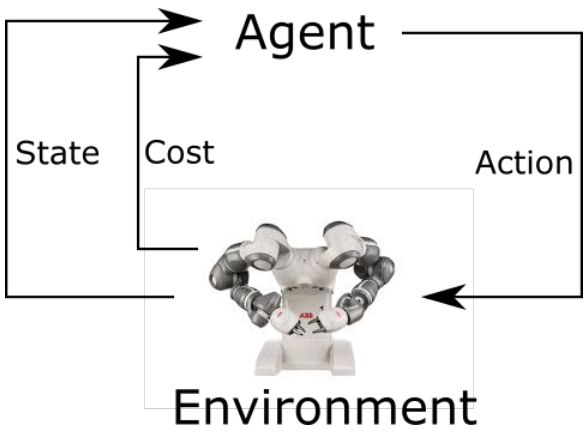


Figure 2.1: Interaction between the agent and the environment in reinforcement learning.

In MDP there are two interacting components. The decision maker is called the agent (controller) and the entity it interacts with is called the environment (system). The interaction between them happens at discrete time steps. At every time step t the agent gets information from the system about the system's *state* $s_t \in \mathcal{S}$. Based on the state information, the controller selects an *action* $u_t \in \mathcal{U}$. A policy π determines the way how the agent chooses actions given the current state and is formulated as a conditional distribution

$$\pi = P[u_t = u | s_t = s]. \tag{2.1}$$

After taking an action, the process moves to a new state $s_{t+1} \in \mathcal{S}$. The probability of ending up in a particular state $s_{t+1} = s^{'}$ is dependent on the action taken and the present state, according to a transition function $\mathcal{T}$:

$$\mathcal{T} = P[s_{t+1} = s'|s_t = s, u_t = u]. \tag{2.2}$$

The transition function completely defines the dynamics of a system. Immediately after taking action, agent receives information about the cost. $\mathcal{C}$ is an expected cost resulting from taking an action u at state s:

$$\mathcal{C} = c(s_t, u_t) = \mathbb{E}[c_t|s_t = s, u_t = u]. \tag{2.3}$$

The policy's goal is to minimize the total cost over future. This means that it should take into account all the cost over many time steps in the future, not only the immediate cost. Such cumulative cost can be calculated for a current time step as

$$V_t = c_t + c_{t+1} + ... = \sum_{k=0}^{T} c_{t+k}, \tag{2.4}$$

where T is a final time step[15].

Note that usually in RL framework one talks about rewards rather than cost. We use cost instead in our work. The only difference is that one seeks to maximize the reward, while the cost is sought to be minimized.

By knowing the transition function i.e. dynamics, it is possible to estimate future states and the resulting cumulative cost depending on control actions chosen. The main problem of MDP is to find a policy, according to which the agent at every time step chooses the action resulting in the smallest cumulative cost:

$$u_t = \arg\min_{u_t} V_t. \tag{2.5}$$

If the state transition function $\mathcal{T}$ is known, then policy can be found using optimal control theory. In case of RL, $\mathcal{T}$ and π are both unknown.

Based on the learning objective of the RL algorithm, two approaches to RL can be distinguished: model-free and model-based. In model-free RL, the algorithm does not learn $\mathcal{T}$, but tries to directly learn a policy π that minimizes the cumulative cost. Model-free methods are known to achieve better asymptotic performance, but may require thousands of trials. Since the experiments on a real robot are expensive, this approach is not feasible for real physical systems. [16]

In model-based RL a dynamics model $\mathcal{T}$ is learned and based on the learned model an optimization is performed to find a policy that minimizes the cost. Even though model-based methods can be suboptimal, they are more data efficient. Sample (information about action and following state) obtained from every step is useful for learning the dynamics better, while for model-free control only the samples that provide reward contribute to learning [16].

2.2 Model predictive control

Here we give a brief summary of model predictive control. More details are available in a book by Rawlings and Mayne [17].

Model predictive control (MPC) is an advanced control method, which computes the control online, i.e. at every sampling instant. Given a discrete time system model relating current state s_t and control input u_t to the next state s_{t+1},

$$s_{t+1} = f(s_t, u_t), \tag{2.6}$$

the objective is to minimize at every time step t some cost function in the form

$$V_H(s_t, \mathbf{u})) = \sum_{i=t}^{t+H-1} c(s_i, u_i). \tag{2.7}$$

which is dependent on a finite number of H future states and control inputs. A model of the system dynamics is used to predict the future states. If the model in (2.6) is not exact representation of the system, the predicted future states are inaccurate with the errors increasing the farther into future the predictions are made. Another problem is a long calculation time of the predictions. As a remedy, MPC uses a receding horizon, i.e. the predictions are limited to a finite number of H time steps.

A major benefit of the MPC is the possibility to subject the minimization problem to constraints:

$$s_k \in \mathbb{S}; \quad k = t, \ldots, t + H - 1\,, \tag{2.8}$$

$$u_k \in \mathbb{U}; \quad k = t, \ldots, t + H - 1\,. \tag{2.9}$$

The solution to the minimization problem is a control sequence

$$u_t, \ldots, u_{t+H-1} = \underset{u_t,\ldots,u_{t+H-1}}{\arg\min}\ V_H(s_t, \mathbf{u}). \tag{2.10}$$

Only the first control u_t from the solution is applied, the rest are discarded. A new control action is then calculated at the next time step.

Various optimization methods are possible for solving (2.10). In the thesis in Section 3.4 gradient-based optimization is considered, while Section 3.5 introduces stochastic optimization.

2.3 Model learning

For a simple system, for which the physics is well-known, it is possible to write an analytical model to describe the relationship between its inputs and outputs. For more complex situations it is either impossible or too time-consuming. If sufficiently many input-output pairs are observed, system identification, or in machine learning terms model learning, can be used to estimate a model of the system. The goal of model learning is to estimate parameters θ to make a model $p(x; \theta)$ map the inputs x to the output y:

$$y \sim p(y|x; \theta) \tag{2.11}$$

A common strategy is to use so-called black-box models. These assume no prior knowledge about the system and the parameters in black-box model have no physical meaning. Alternatively, a gray box modelling is possible, where the structure of the model is fully or partially known, but not all or none of the parameters are known [18].

Furthermore, one can distinguish between probabilistic and deterministic models. Probabilistic models express model uncertainty by giving posterior distribution as in (2.11), while deterministic models disregard the uncertainty and treat the model as a true system function. For deterministic cases, the model can be denoted as

$$y = \hat{f}(x; \theta). \tag{2.12}$$

Once the model structure and size are chosen, the parameters need to be estimated. This is done by minimization of some criterion function, we will call it loss function to distinguish it from cost function used by MPC optimization. A common loss function is sum of squared error between observed outputs and model outputs:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{N} ||y(i) - \hat{y}(i)||^2, \tag{2.13}$$

where N is the number of observations and $\hat{y}(i) = f(x(i); \theta)$. Alternatively, average of the squared error can be used:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} ||y(i) - \hat{y}(i)||^2. \tag{2.14}$$

2.3.1 Neural network as function approximator

One possible model learning technique is to employ artificial neural network (NN). We give here a quick overview of the NN. Interested readers can refer to the book by Goodfellow et. al. [19] for a more detailed insight.

We can see neural network as an approximation of a function $f(x; \theta)$ that maps the inputs x to an output y. NN model is a parametric model, which means that the model is characterized by a fixed number of parameters θ, the values of which are to be learned.

The main element of a neural network is called a node or a neuron. Each node can take a vector of inputs x and the output of the neuron is

$$h = g(x^T w + b), \tag{2.15}$$

where w is a vector of weight parameters, b a scalar bias parameter and g is an activation function. Most commonly used activation function is the rectified linear activation function (RelU), which is defined as

$$g(z) = max\{0, z\}. \tag{2.16}$$

If several of such nodes are used in parallel it is called a layer of nodes and the number of nodes in one layer is called layer size. The network can be made deeper

by adding more layers, so that the output of the previous layer is the input for the next layer. The output of the final layer is the model output y.

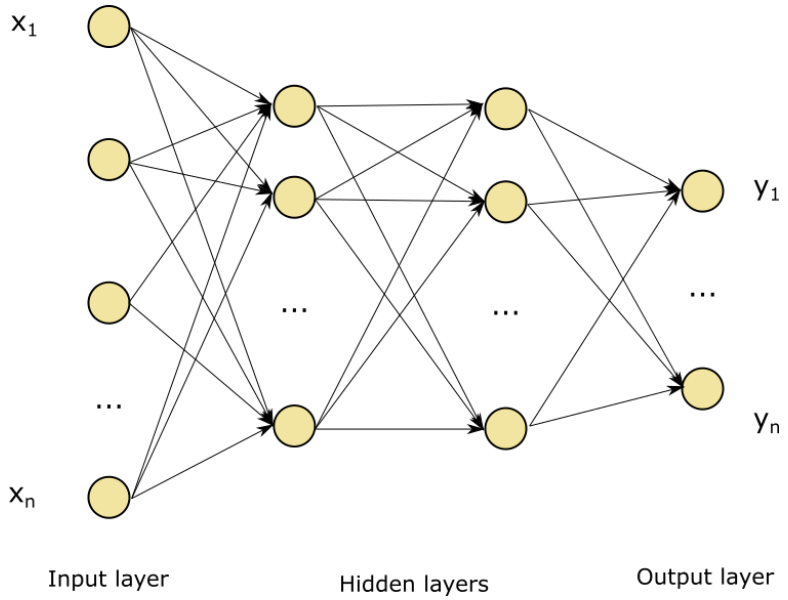


Figure 2.2: Artificial neural network.

The weight and bias parameters are learnable. This means that the parameters are changed in a way that the NN model best fits the function it tries to approximate. The fit is decided by minimization of a loss function. Most NN learning algorithms use gradient descent for the minimization.

2.3.2 Gradient descent for minimization

In the beginning, the parameters θ are initialized either randomly or in some other manner. Using the current parameters, the output of the model for all input data is calculated. That step is called forward propagation. Then, using the loss function, a loss is found and a back-propagation algorithm is used to compute the gradient of the loss function w.r.t. all parameters. Finally, the parameters are updated using the gradient information. Usually, a learning rate α is used, that way only a fraction of the gradient is subtracted from the parameters. Many iterations of this process are done, whereas in each iteration i, new parameter value

$$\theta(i+1) = \theta(i) - \alpha \cdot \frac{dJ}{d\theta(i)}. \tag{2.17}$$

This explanation applies for the most basic gradient descent algorithm. A modified version of this is generally used for training NN.

2.3.3 Generalization of neural networks

Important characteristic of an NN model is generalization. It means that the loss function needs to be small not only for the training data, but also for a new data. To evaluate the generalization of an NN model, two datasets are created - training

dataset and validation dataset. NN model is fitted using the training dataset, after that the parameters are not changed anymore. The loss is then calculated using the validation data. The validation loss is typically equal or larger than training loss. If the gap is too large, there is an overfitting problem, i.e. the model cannot predict well when dealing with unseen data. This is common when the model capacity is very high, i.e. it is able to fit very complex models. To deal with the problem of overfitting, the best solution is to increase the amount of training data if possible.

On the other hand, underfitting is the problem arising when the model is unable to reduce the loss on the training set. Then the model capacity needs to be increased, which can be done by increasing the network's depth or the amount of neurons in its hidden layers. It is generally recommended to choose as simple model as possible, that is still precise enough to solve the task at hand.

For the evaluation with the training and validation sets to have meaningful information, the data needs to satisfy i.i.d. assumptions. In particular, every data sample is assumed to be independent of each other and drawn from the same distribution.

3

Model-based reinforcement learning

The ultimate goal of the methods described in the thesis is to find a policy that enables a manipulator to complete an insertion task. While optimal control theory aims to find a control law with the aid of system dynamics for achieving the desired objective, here we do not assume any prior information about the system dynamics. Instead, we use model learning to learn the system dynamics and then apply online control to the learned model. By repeatedly relearning the model to improve it after acquiring new data, the method becomes reinforcement learning approach. In this chapter we first formulate the insertion problem as a Markov decision process (MDP) in Section 3.1 and follow by explaining our strategy of solving the problem with reinforcement learning in Section 3.2. Section 3.3 goes more into detail about the way we learn the model with neural network and sections 3.4 and 3.5 discuss the different methods compared in the thesis for solving an MPC optimization problem.

3.1 Skill learning problem as MDP

We assume the environment to be rigid and stationary (the dynamics are not dependent on time). Therefore, every new state is only dependent on the previous state and previous action, satisfying the Markov property. The problem can thus be formulated as an MDP.

As explained in Section 2.1, MDP is a tuple $(\mathcal{S}, \mathcal{U}, \mathcal{T}, \mathcal{C})$. For the insertion task, state space $\mathcal{S}$ consists of manipulator joint positions and velocities as well as end effector cartesian coordinates and their velocities. Actions $\mathcal{U}$ are the control inputs, specifically the torques applied to each joint. We consider a deterministic dynamics function

$$s_{t+1} = f(s_t, u_t). \tag{3.1}$$

The cost function

$$c_t = c(s_t, u_t) \tag{3.2}$$

defines our objective and is thus designed to penalize distance of the manipulator from the target position. In addition, the control inputs are added to the cost function to encourage policy to use minimal force.

We consider the task to be limited to a finite length of T time steps. It means that during one trial, the controller has T time steps to complete the task. We call such a trial an episode. Every episode starts from some state s_0 and the policy's goal is always to complete the task by minimizing the cumulative cost over future

$$V_T = \sum_{i=t}^{T} c_i. \tag{3.3}$$

3.2 Learning based MPC as RL

Generally model-based reinforcement learning can be divided into three steps: collecting new data, model learning, and policy optimization.

In the beginning we have no information about the dynamics, which is necessary for model-based optimization. To learn the dynamics model, a dataset D consisting of input-output observations is used. The data is collected by running some episodes and observing the actions taken and resulting states. More specifically, we start from some initial state s_0, apply some control input according to the policy, and observe the next state that resulted from this action. As a result we have gathered one sample of data, which is a tuple consisting of action and the next state (u_0, s_1). The same process is repeated in the new state and so on. In the end we have a dataset D storing $(s_0, u_0, s_1, u_1, \dots)$.

The policy used in this first data collection phase, which we call initial policy, can not be based on any model, since we do not have any yet. The easiest option is to use a completely random controller. This means that at every state, we apply a random control input from a uniform distribution with specified limits.

After we have sufficient data in the set D, we proceed with model learning. We use a deterministic neural network model $\hat{f}(s_t, u_t)$, which approximates a function $f(s_t, u_t)$, and the model is trained by learning the weights of a neural network. The details about the model learning process are given in the Section 3.3. This model represents the dynamics of the system, meaning that now, when it is learned, a model-based policy can be implemented.

The model can be used to predict the next state $s_{t+1} = \hat{f}(s_t, u_t)$ at any time step. Based on the predicted next state, even another future state can be predicted $s_{t+2} = \hat{f}(s_{t+1}, u_{t+1})$. In fact, we can predict as many futures states as we wish, which allows us to calculate cumulative cost V_T at every time step. However, since the model $\hat{f}$ is not perfect, the predictions are not accurate and the predictions that are based on previous predictions are even less accurate, making V_T also inaccurate. In MPC we consider limited horizon H and the cost function

$$V_H = \sum_{i=t}^{t+H-1} c_{t+k}, \tag{3.4}$$

which is less erroneous, given that $H < T$.

Knowing the model and the cost function, we can now use a model-based policy. There are offline optimization methods, e.g. policy search, that find a parametric policy function $\pi(\theta)$, before starting an episode. The function would then be used at every time step to calculate a control action. We use an MPC control instead, which calculates new action at every time step. The MPC policy's job is at every time step to choose the control input that minimizes V_H. This is done by solving

$$u_t, \ldots, u_{t+H-1} = \underset{u_t,\ldots,u_{t+H-1}}{\arg\min} \sum_{i=t}^{t+H-1} c(s_i, u_i) \tag{3.5}$$

Different optimization methods are possible and we consider two methods: gradient descent based optimization and stochastic optimization. These are discussed in detail in sections 3.4 and 3.5 respectively.

The dynamics of the manipulator is very complex. To learn the model $\hat{f}$ precisely enough, dataset D needs to be very large. This means that the robot needs to run for a long time, which is expensive. To overcome this problem, we use the idea of data aggregation. While utilizing the MPC policy, all states $s_0, \ldots, s_T$ visited and all the control signals $u_0, \ldots, u_T$ applied during the episode are added to the dataset D. After a fixed number of episodes, the enriched D is used to relearn the dynamics, i.e. to fit the data to a model. The new model should be better, because the data contains more states that the controller is likely to encounter during the execution. In other words there is a smaller mismatch between state-action distribution of the data and the MPC controller [20] [12].

Several such iterations of alternating between learning a model and collecting new data are done until the model is sufficiently accurate for MPC to complete the task.

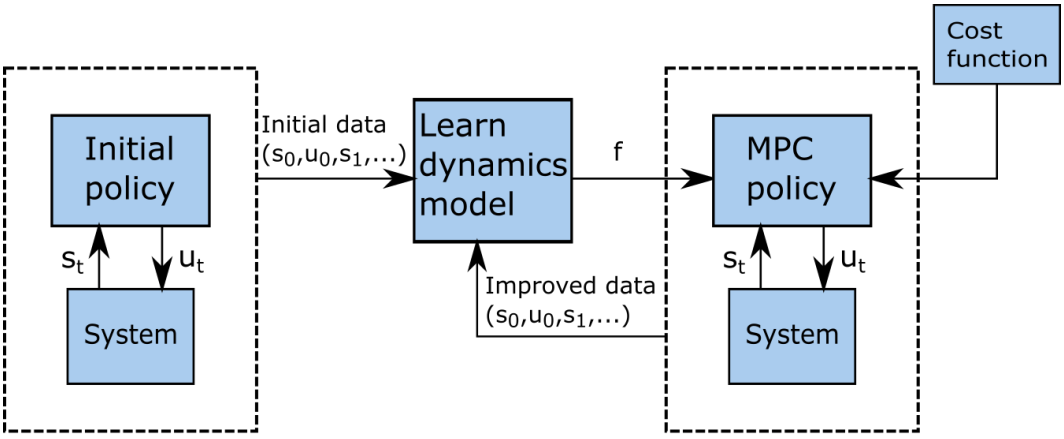


Figure 3.1: Model based reinforcement learning. The policy improves with every iteration due more accurate model as a result of improved dataset.

In every iteration several episodes are executed. While the initial state can be the same for all episodes, we decide to change it each time by a small amount to encourage exploration, i.e. to increase the chance that the controller finds a new, but more optimal route. Another technique that we use to encourage exploration, is a simple ϵ-greedy exploration. This means that at every time step there is a small chance for the controller to pick a random action from a limited range instead of calculating an optimal action. The likelihood of taking a random action is determined by value of $0 < \epsilon < 1$.

We summarize the model-based reinforcement learning method in Algorithm 1.

Algorithm 1: Model-based reinforcement learning

```
1  collect initial data D
2  for iteration = 0 to n_iterations do
3      train model f̂(s_t, u_t) using D
4      for episode = 0 to n_episodes do
5          move to initial state s_0
6          for t = 0 to T do
7              Generate a random number ρ from uniform distribution between
                0 and 1
8              if ρ < ε then
9                  pick random u_t from a uniform distribution of limited range
10             else
11                 estimate optimal control input sequence u_t ... u_H using
                    f̂(s_t, u_t) and c(s_t, u_t)
12                 apply the first input u_t from the sequence
13             end
14             observe the new state s_{t+1}
15             add (u_t, s_{t+1}) to D
16         end
17     end
18 end
```

Detailed explanation about the line 3 is given in Section 3.3 and about the line 11 in Section 3.4 and Section 3.5.

3.3 ANN based model learning

We have described how we get a dataset D, consisting of N input-output pairs, in every iteration of our RL algorithm. We will now look into how we obtain the model $\hat{f}(s_t, u_t)$ using this data.

An important first step for the training is data preprocessing. We normalize the inputs by finding the mean and standard deviation of input states and input actions. Then we subtract the corresponding mean from every sample and divide the result by the corresponding standard deviation. Such preprocessing makes sure that all states are weighted equally by the loss function [12].

The purpose of the model $\hat{f}$ is to predict the next state s_{t+1}, given current state s_t and control input u_t. Nagabandi et al. suggest learning the change in state instead of learning the next state directly, pointing out that s_t and s_{t+1} might be too similar and the effect of control input on output seemingly small as a result [12]. Following their advice, the predicted next state is

$$s_{t+1} = s_t + \hat{f}(s_t, u_t). \tag{3.6}$$

We use average squared error as our loss function:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} ||(s_{t+1} - s_t) - f(s_t, u_t; \theta)||_2^2. \tag{3.7}$$

To minimize the loss function, we use gradient descent. There are many variants of gradient descent. Standard batch gradient descent computes the gradient in 2.17 using the whole dataset. Stochastic gradient descent, on the contrary, approximates the gradient and updates the parameters at every data sample. We use a variant of gradient descent called mini-batch gradient descent, which is a middle ground between batch and stochastic gradient descent. It divides the dataset into mini-batches (smaller sets of data) and uses one mini-batch at a time to calculate gradient and perform parameter update. The number of data samples in one mini-batch is called batch size. The gradient calculation with the mini-batch method is the most efficient, allowing the fastest training.

One pass through the whole data is called an epoch. We shuffle our data after every epoch and split the data into mini-batches in a random order. During shuffling, we make sure that inputs remain connected to the correct output.

We are using The Adaptive Moment Estimation (Adam) optimization algorithm, which is one of the most widely used modification of gradient descent in neural network training. The Adam algorithm computes individual learning rates for all parameters, using the estimates of first and second moments of gradients [21].

All the parameters used by NN optimization algorithm, which are not learned, but need to be tuned by us, are called hyperparameters. The hyperparameters are:

- number of hidden layers,
- number of neurons in hidden layers,
- batch size,
- number of epochs,
- hyperparameters specific to Adam (β_1, β_2, ϵ and learning rate α)

3.4 Gradient descent based optimization

Once we have a dynamics function and cost function, we can perform policy optimization. The particular challenge is, how to solve (3.5). We have already talked about gradient descent as a way to minimize loss function for model estimation in Section 2.3.1. It is natural to try the same approach in minimization of the MPC cost function. The requirement for using the gradient descent is that the cost function is differentiable.

The cost function we want to minimize is

$$V_H(s_t, \boldsymbol{u}) = \sum_{i=t}^{t+H-1} c(s_i, u_i), \tag{3.8}$$

where $s_{t+1} = s_t + \hat{f}(s_t, u_t; \theta)$. Algorithm 2 explains how the optimal control input $u = u_t, \ldots, u_{t+H}$ is found iteratively. While it is common to continue the iterative process until the solution has converged i.e. the difference between two consequent solution $\boldsymbol{u}$ is below some threshold, we set a limit to the number of iterations instead. We call one iteration a gradient step, and we take *n_steps* gradient steps. Learning

rate γ determines how fast the algorithm moves in the negative direction of gradient. If it is too large, the algorithm may not converge, while too small γ makes reaching the solution slow.

Algorithm 2: Gradient descent based optimization

```
initialize u(0)
for i = 0 to n_steps do
    u(i+1) = u(i) − γ · dV_H / du(i)
end
```

At the first time step of every episode $u(0)$ is initialized to zero. In the next time steps $u(0)$ is equal to the solution of the previous time step.

An important challenge is to keep the computation time of this algorithm below 50 ms, as it is difficult to control a robot with frequency smaller than 20 Hz. We limit the computation time by choosing *n_steps* small and tuning γ appropriately. The most time-consuming part in the algorithm is the computation of gradient. We have implemented the algorithm in TensorFlow, as well as all the relations between current state, input and the summarized cost function, which are required to calculate the gradient. The TensorFlow function *tf.gradients()* then manages to compute the gradients with high efficiency.

Algorithm 2, which we have implemented, demonstrates a simple standard gradient descent for MPC optimization. One can also consider a constrained optimization to guarantee that the states and control inputs stay in a limited range. We have left this for possible future work. To keep the inputs low, we penalize them in the cost function with relatively large weights.

The gradient descent method has a few drawbacks. While working well with simple functions, it can become too slow if the dimensionality is high and the functions become nonlinear, as is the case with 7-joint robot dynamics in constraint-rich environment. Another problem is that the algorithm can converge to a local minimum instead of a global minimum. To tackle these issues we consider alternatively a stochastic optimization method.

3.5 Stochastic optimization

Nagabandi et al. use a random-sampling shooting method (we will henceforth call it random shooting), claiming it to be a simple solution for nonlinear optimization. Indeed, the method as summarized by Algorithm 3 is straightforward.

Algorithm 3: Random shooting method for optimization

1. Generate K candidate control input sequences $u_t, \ldots, u_{t+H-1}$, picking each sequence from a uniform distribution
2. Using the cost function (2.7) and the dynamics model (3.6), calculate cost for all K sequences
3. Pick the control input sequence that results in the lowest cost

From now on we will call the candidate sequences *shootings*. The number of shootings K is a critical tunable parameter, that determines the likelihood that the best solution found is close to the optimal solution.

The random control input sequences are each picked independently from a uniform distribution of a limited range. This way we can easily constrain the control inputs to our desired limits, which is an advantage over the gradient descent method. The possible drawback can be a high number of shootings required for a satisfactory performance in high-dimensional problems.

3.5.1 Cross-entropy method

To improve on the random shooting method, we use a modified version of it, called cross-entropy method (CEM), which was first introduced by Rubinstein [22]. Chua et al. show that CEM can achieve better performance than standard random shooting, while using the same sample size [13]. We use the code provided by Chua et al. as a basis for our own implementation.

Similarly to the standard random search method, we want to generate K random control input sequences. However, we sample them from a multivariate normal distribution instead of a uniform distribution. Each candidate input sequence is a matrix

$$\boldsymbol{u} = \begin{bmatrix} u_t & u_{t+1} & \dots & u_{t+H-1} \end{bmatrix}^T, \tag{3.9}$$

that is sampled from a normal distribution

$$\boldsymbol{u} \sim \mathcal{N}_{H \cdot N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}), \tag{3.10}$$

where N is the input space dimension, $\boldsymbol{\mu} \in \mathbb{R}^{H \cdot N}$ is a mean vector and $\boldsymbol{\Sigma} \in \mathbb{R}^{H \cdot N \times H \cdot N}$ is a covariance matrix. Using the cost function, we find a fixed number of best sequences (elites). Based on the elites we estimate a new narrower Gaussian distribution for the next iteration. This process is repeated iteratively, whereas the distribution is expected to generally produce more optimal samples after each iteration. In detail, the process is described by Algorithm 4.

Algorithm 4: Cross-entropy method for optimization

1 Pick initial mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$
2 **for** $i = 0$ **to** *n_iterations* **do**
3 Generate K candidate control input sequences $\boldsymbol{u}$ from the mean μ and covariance Σ
4 Using the cost function (2.7) and the dynamics model (3.6), calculate cost for all K sequences
5 Order the sequences, from those that result in the smallest cost to the largest, $\boldsymbol{U}_{ordered} = \begin{bmatrix} \boldsymbol{u}_1 & \dots & \boldsymbol{u}_K \end{bmatrix}^T$
6 Choose *n_elites* first sequences from the ordered list $\boldsymbol{U}_{elites} = \begin{bmatrix} \boldsymbol{u}_1 & \dots & \boldsymbol{u}_{n\_elites} \end{bmatrix}^T$
7 new mean vector $\boldsymbol{\mu} = \mathbb{E}[\boldsymbol{U}_{n\_elites}]$ and new covariance matrix $\boldsymbol{\Sigma} = cov[\boldsymbol{U}_{n\_elites}]$
8 **end**

We set a limit to the number of iterations to avoid the iterative process taking too much time. The algorithm is expected to be more data efficient than the ran-

dom shooting method, i.e. the total number of samples created at each time step to be smaller. While the CEM algorithm does multiple iterations, each of these can use fewer shootings compared to the random shooting algorithm due to more sophisticated choice of distribution.

4
Evaluation

4.1 Description of experiments

In order to validate the method, we evaluate it on two different problems with different complexity. Making the manipulator to perform the insertion with reinforcement learning is an ambitious task and therefore we start with a simpler cart-pole problem, which is a well-studied problem in the control field. It consists of an inverted pendulum, that must be balanced on top of a moving cart. The cart has one degree of freedom and its movement can be controlled by applying horizontal force to it.

In reinforcement learning problems we define a state space $\mathcal{S}$ and action space $\mathcal{U}$. The state space for the cart-pole problem consists of cart position and velocity as well as pendulum angle and angular velocity. The action space consists of the horizontal force applied to the cart.

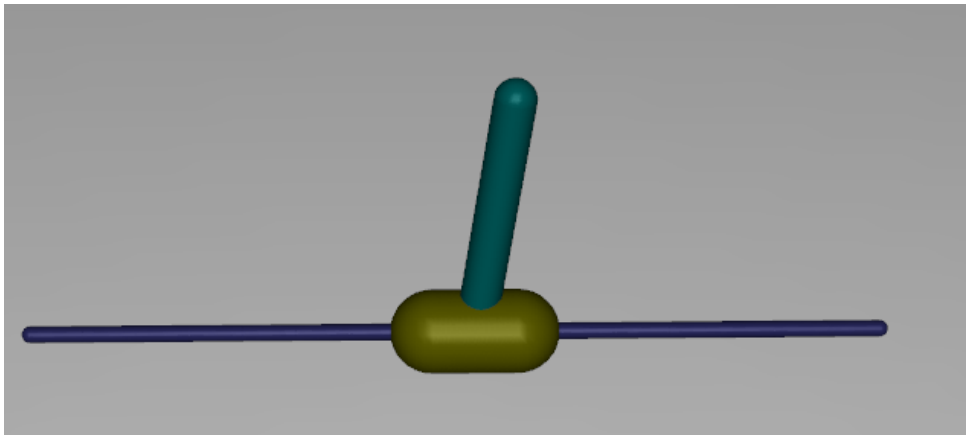
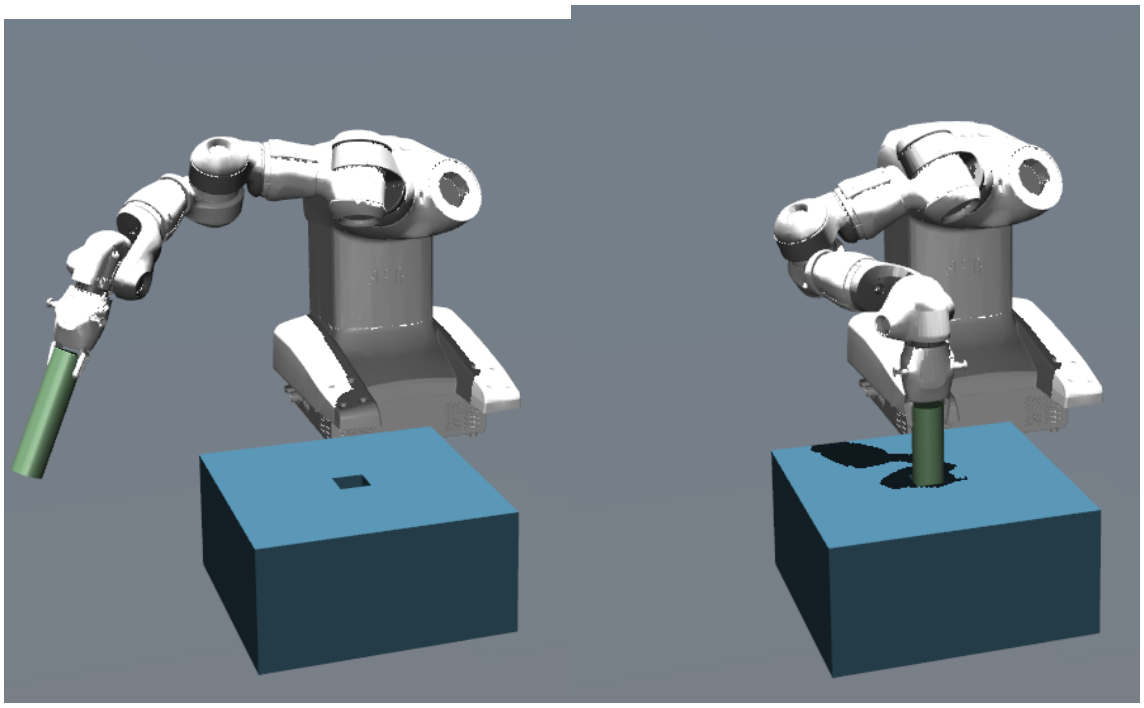


Figure 4.1: A screenshot of the cart-pole problem in MuJoCo simulation. Shown is the cart moving on the rail and the pendulum on top of the cart.

The second problem is the insertion task performed by robot manipulator. As a manipulator, the YuMi robot arm is used. There is a block with a hole inside it set up next to the robot. A cylindrical peg is attached to the end effector and the robot's task is to bring the object to the hole. Since the cross-section of the hole is not much larger than the cross-section of the cylinder, the manipulator is expected to collide with the block. The controller also cannot choose the most direct path to the target, but has to plan the way over the wall.

The state space $\mathcal{S}$ of the manipulator includes joint positions and velocities, while the action space $\mathcal{U}$ is a set of torques generated by the motors to control the



(a) Initial position **(b)** Target position

Figure 4.2: A screenshot of the insertion problem in MuJoCo simulation. Yumi robot is illustrated in the initial position (left) and in the target position (right).

joints. The high dimensionality of the state and action spaces makes it a much more complex problem compared to the cart-pole task.

4.2 Cost function

A cost function defines the objective which we want to achieve by minimizing that cost. In the case of cart-pole task, where we wish the pendulum to stay upright and the cart to stay near the initial position, we choose to penalize the pendulum angle θ and the cart position x. We would also like minimal force to be used, so we also penalize the control input. Penalizing the velocities did not increase the stability during the experiments, so they are not included in the cost function. Consequently, the cost function

$$c(s_t, u_t) = w_1 \cdot x_t^2 + w_2 \cdot \theta_t^2 + v \cdot u_t^2, \tag{4.1}$$

where w_1, w_2 and v are weights that help scale or prioritize some terms over others.

For insertion task, the state space for the controlled system consists of 32 states in total. First there are joint positions and velocities. To make it easier to achieve the objective, the positions of three points on the end effector and their velocities are also measured and added as states. In the end, states

$$s = \begin{bmatrix} q & \dot{q} & p & \dot{p} \end{bmatrix}^T, \tag{4.2}$$

where the joint positions

$$q = \begin{bmatrix} q_1 & q_2 & ... & q_7 \end{bmatrix}^T, \tag{4.3}$$

the joint velocities

$$\dot{q} = \begin{bmatrix} \dot{q}_1 & \dot{q}_2 & ... & \dot{q}_7 \end{bmatrix}^T, \tag{4.4}$$

the coordinates of three points on the end effector

$$p = \begin{bmatrix} p_x^1 & p_y^1 & p_z^1 & ... & p_x^3 & p_y^3 & p_z^3 \end{bmatrix}^T, \tag{4.5}$$

and the velocities of the three points on the end effector

$$\dot{p} = \begin{bmatrix} \dot{p}_x^1 & \dot{p}_y^1 & \dot{p}_z^1 & ... & \dot{p}_x^3 & \dot{p}_y^3 & \dot{p}_z^3 \end{bmatrix}^T. \tag{4.6}$$

A quadratic cost function is used, where we penalize the distance of the end effector from the target p_r, joint position difference from the target joint position q_r, joint velocities and control inputs:

$$c(s_t, u_t) = w_1 \cdot ||p_t - p_r||^2 + w_2 \cdot ||q_t - q_r||^2 + w_3 \cdot \dot{q}_t^2 + v \cdot u_t^2. \tag{4.7}$$

To obtain the target position, the manipulator is brought to the desired final position in the simulation environment. It is then possible to measure p_r and q_r.

Inserting an object into tight-fitting hole is an objective that requires high precision. The given cost function is generally enough to reach the desired position, but can often fail to complete the final insertion step. Levine et. al. suggest using a logarithmic cost term, called Lorentzian ρ-function $w_l \cdot \log(d^2 + \alpha)$, where d is the penalized distance, w_l is a weight and α some small number [7]. This term gives a negative cost, which has a bigger effect close to the target. This way it encourages the completion of the task, when the end effector is close to the hole. The value of α determines the distance from the target, where the term starts having a considerable effect.

4.3 Cart-pole simulation

The goal of the cart-pole experiments, besides proving that the derived method works for a simple control problem, was to compare two methods: random shooting and gradient descent. An improved method of random shooting, CEM, was also tested. In addition to confirming that the methods are able to achieve the objective, we also wanted to evaluate the time it takes to calculate the control action at each time step. Since this time is limited for a real robot, we would like to minimize it as much as possible.

To do the simulations we use the inverted pendulum environment from OpenAI Gym MuJoCo library [23]. MuJoCo is a physics engine, that is efficient at simulating contacts [24].

To compare performances with different methods, two quantities are considered:

$$c_s = \frac{1}{M} \sum_{i=1}^{M} \sum_{j=1}^{N} \theta_j^2 \tag{4.8}$$

and

$$c_s = \frac{1}{M} \sum_{i=1}^{M} \sum_{j=1}^{N} \theta_j^2, \tag{4.9}$$

where M is number of episodes in iteration and N is episode length.

The first is the total cost over the episode, averaged over all episodes in the iteration. The cost function to calculate this is not the same that is minimized by optimization algorithm. Rather, only the angular distance of the pendulum from the target is considered as well as only the true state at every time step instead of the predicted states over the horizon.

The second quantity is the input cost in the episode, i.e. sum of squared forces applied to the cart at every time step in episode. Again the average over all the episodes in an iteration is taken.

4.3.1 Reinforcement learning

First, to gather data for training, 50 episodes with the length of 50 time steps each are run with a random controller generating random actions.

When using neural networks, there are several hyperparameters to choose. The choice can substantially affect the control performance. While bigger network can achieve more accurate model, smaller network is faster and sufficient for this relatively simple task. Thus, no advanced method like grid search was used for hyperparameter choice, as this would always rate unnecessarily large networks higher (smaller validation set error). Instead, small network was preferred with size that is enough to complete the task in small number of RL iterations.

The Adam optimization algorithm was used. The default parameters as given in the paper by Kingma and Ba, were chosen, with learning rate $\alpha = 0.001$, the exponential decay rates for moment estimates $\beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$ [21]. For other hyperparameters:

- network depth - 1 hidden layer with ReLU activation,
- layer size - 32 neurons,
- batch size - 64.

We train for 200 epochs as we have confirmed from learning curves that by that time the error is nearly zero for most iterations. Every iteration consists of 5 episodes. Each episode has a length of 100 time steps. The sampling frequency is 50 Hz, which means that episode length is 2 s. 10 iterations are run altogether. To encourage learning, the initial state is random. All initial states are drawn from a uniform distribution with range from -0.01 to 0.01.

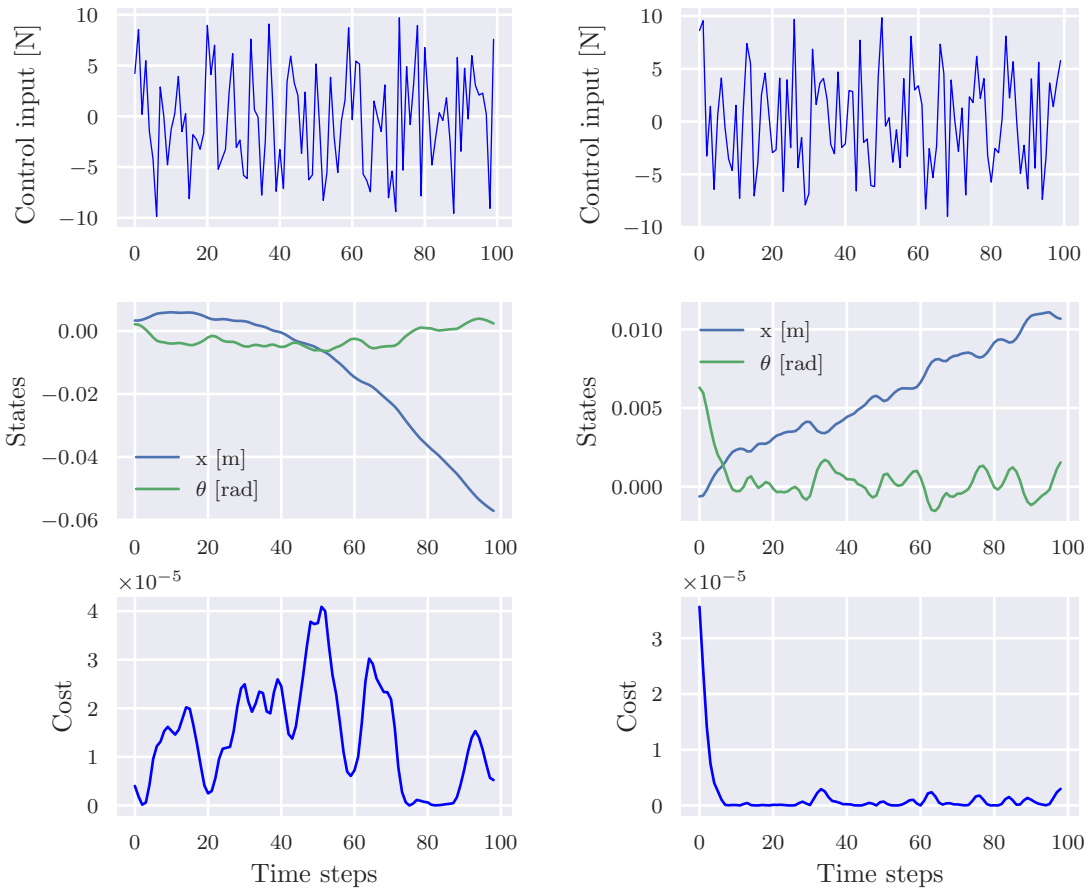
4.3.2 Optimization with random shooting

Penalizing the angles has a higher priority than penalizing the cart position. Therefore, the weights were chosen $w_1 = 1$ for the cart position and $w_2 = 1000000$ for the angle. In addition, the force applied to the cart is penalized with a weight of 0.1. A sum of costs over all the predicted future states is taken. Thus, the cost for one shooting at time t is given by:

$$V(t) = \sum_{i=t}^{t+H-1} (x_t^2 + 1000000 \cdot \theta_t^2 + 0.1 \cdot u_t^2), \tag{4.10}$$

where H is planning horizon.

The horizon H and number of shootings K were chosen as small as possible to decrease calculation time, while still achieving the objective. The chosen values were $H = 8$ and $K = 20$. Each random control input is drawn from a uniform distribution between -10 and 10 N.

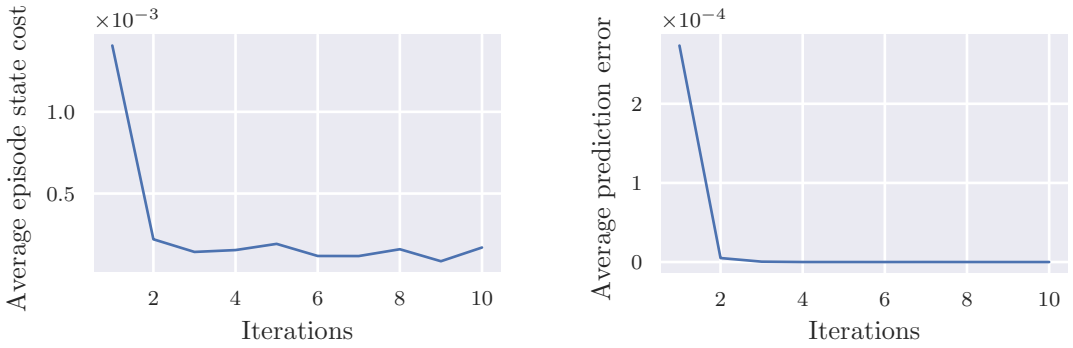


(a) Iteration 1 **(b)** Iteration 10

Figure 4.3: Input, cart position and pendulum angle and cost with random shooting at first iteration and tenth iteration.

Figure 4.3 shows that after doing 10 iterations of RL, the controller performance has improved. The cost quickly goes down and stays low. The performance is already satisfactory in the first iteration though, which shows that for this problem doing reinforcement learning is actually unnecessary and it is enough to just train the model on some initial randomly generated data once.

Figure 4.4 shows that already after 2 RL iterations model learning has converged and there is little improvement regarding the state cost in the following iterations. We can conclude that RL has been successful.



(a) State cost **(b)** Prediction error

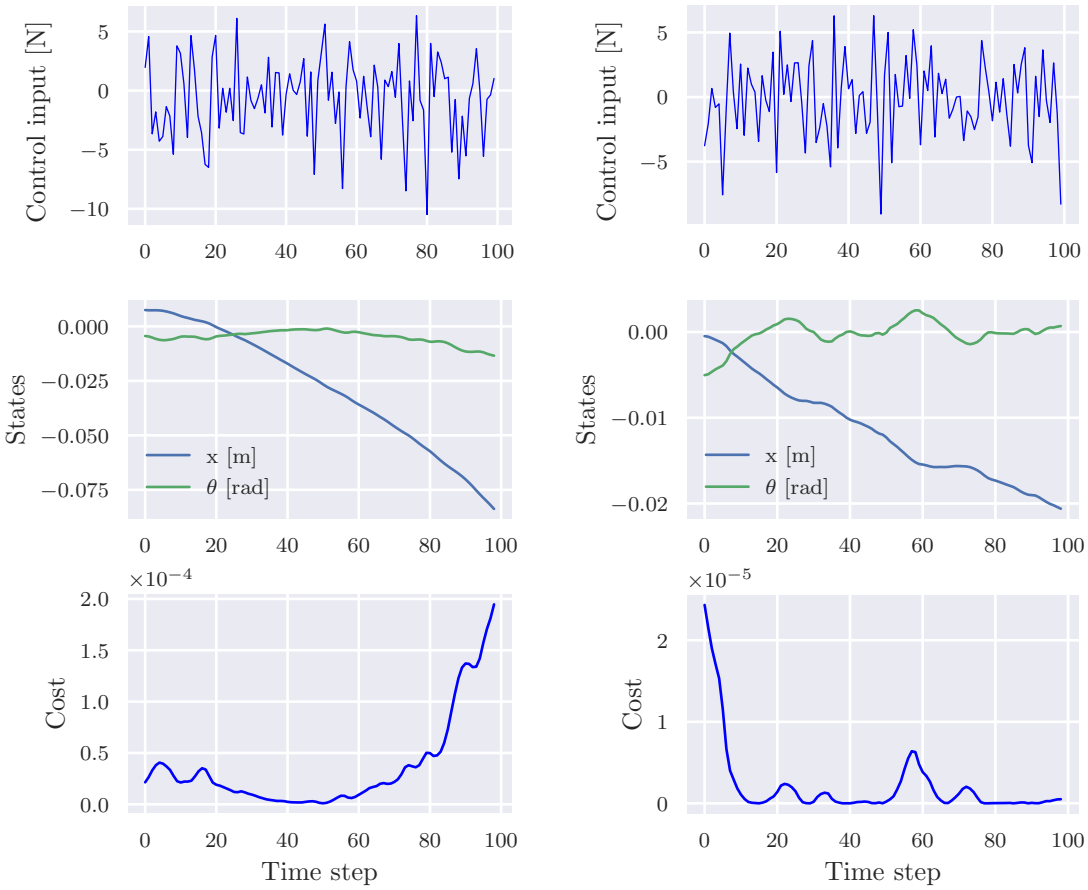
Figure 4.4: Average episode state cost and prediction error with random shooting over ten iterations for cart-pole task.

4.3.3 Optimization with cross entropy method

The same cost function was used as with random shooting. Planning horizon was kept 8, while number of shootings could be lowered to $K = 10$. The CEM algorithm is stopped after four iterations are run. Various parameters related to CEM were chosen as follows:

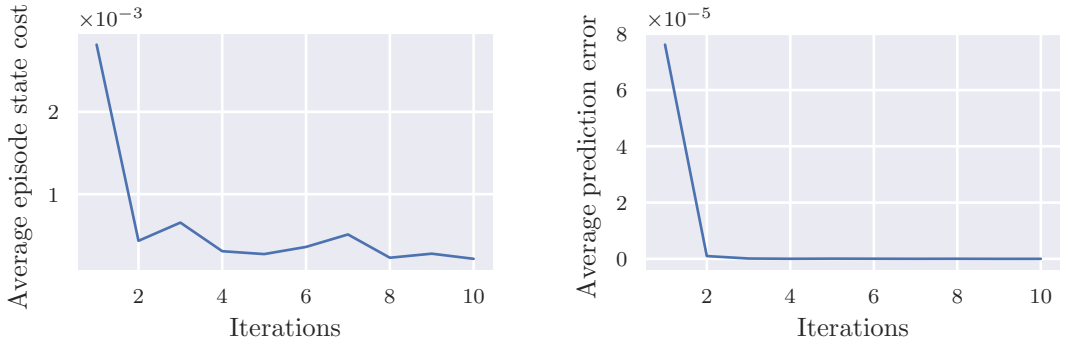
- rarity parameter - 5
- initial variance - 25
- initial mean - 0

From Figure 4.5 we see that even though the controller fails to balance the pole in the first iteration, it has learned to do it successfully by the tenth iteration. There is a significant improvement between the first and second iteration and not much change after that, as seen in the Figure 4.6



(a) Iteration 1 **(b)** Iteration 10

Figure 4.5: Input, cart position and pendulum angle and cost with CEM at first iteration and tenth iteration.

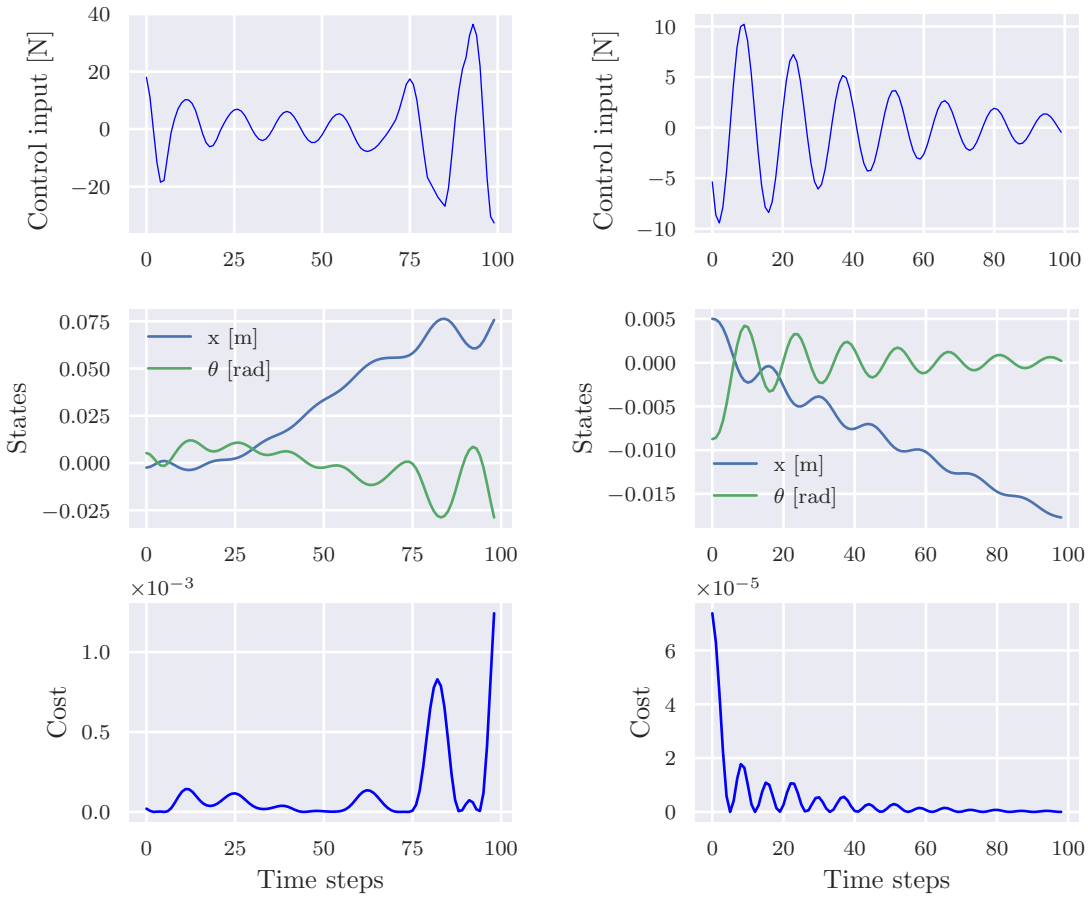


(a) State cost **(b)** Prediction error

Figure 4.6: Average episode state cost and prediction error with CEM over ten iterations for cart-pole task.

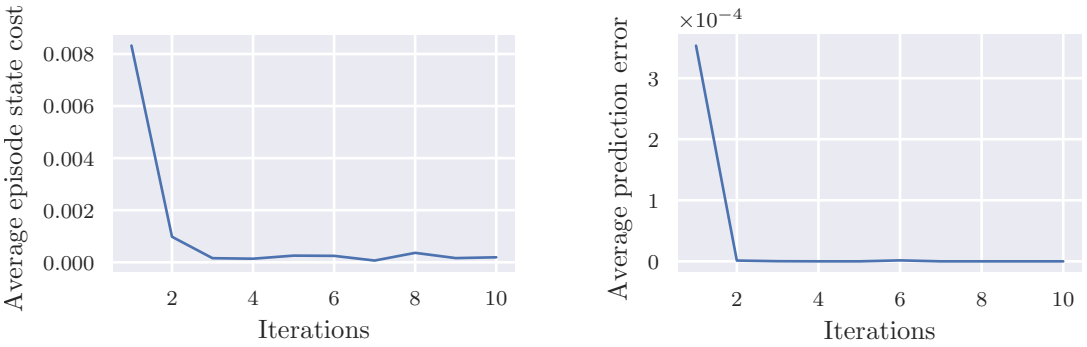
4.3.4 Optimization with gradient descent

With gradient descent method, the same cost function was used as with random shooting method. Learning rate of 0.01 was used for the optimization of control input. The number of gradient steps could be reduced to 3, while the planning horizon $H = 5$. Figure 4.7 shows that after ten RL iterations, gradient descent method achieves smaller cost than in the first iteration. Again, the performance is good enough already in the first iteration. Figure 4.8 shows only slight improvement on state cost and model accuracy after first 2 iterations.



(a) Iteration 1 **(b)** Iteration 10

Figure 4.7: Input, cart position and pendulum angle and cost with gradient descent at first iteration and tenth iteration.



(a) State cost **(b)** Prediction error

Figure 4.8: Cost and prediction error with gradient descent over ten iterations for cart-pole task.

4.3.5 Comparison

The table 4.1 compares the time it takes to calculate control with different methods as well as their performance evaluation quantities. It can be seen that gradient descent and random shooting have similar computation time, also their performance as implied by state cost is similar. Gradient descent method, however, uses lower control inputs on average. CEM is about two times slower than other methods. The reason is that with CEM multiple iterations of cost calculation for each shooting are done at every time step. At the same time CEM does not achieve better performance than other methods.

The worse performance of CEM compared to random shooting is surprising, since we expected it to be better in both performance and computation time. The results are also not sufficient to make a choice between random shooting and gradient for the robot insertion task, since in a high dimensional task, there can be significant differences. Thus, we decided to repeat the experiments with all three methods also in the insertion task.

	Computation time (ms)	State cost	Input cost
Shooting	13	$1.7 \cdot 10^{-4}$	2245
CEM	25	$2.2 \cdot 10^{-4}$	1073
Gradient	11	$1.9 \cdot 10^{-4}$	703

Table 4.1: Comparison of different methods for cart-pole problem. Shown are average input calculation time and state cost and input cost in the final iteration.

4.4 Insertion

To do the simulations, YuMi model in MuJoCo environment was used. The same three methods that were used for cart-pole problem were tested again, in order to decide which method would be the best for the insertion task.

Again two quantities are created to measure the performance of the different methods. The state cost is defined as

$$c_s = \frac{1}{M}\sum_{i=1}^{M}\sum_{j=1}^{N}(||q - q_t||^2 + ||p - p_t||^2), \tag{4.11}$$

where j_t is the target joint positions and p_t is the target end effector point positions. The input cost is defined as

$$c_i = \frac{1}{M}\sum_{i=1}^{M}\sum_{j=1}^{N} u^2, \tag{4.12}$$

4.4.1 Reinforcement learning

To collect initial data, 15 episodes with the length of 100 time steps are run with an initial policy. The initial policy can not simply take completely random actions, since it would clearly lead to unwanted collisions in case of a real robot. As a simple solution we implement a proportional-derivative (PD) control.

First, we define the error in joint position at current time step

$$q_t^e = q_r - q_t \tag{4.13}$$

and error in joint velocity

$$\dot{q}_t^e = \dot{q}_r - \dot{q}_t \tag{4.14}$$

Then at each step we find the control as

$$u_t = K_p \cdot q_t^e + K_d \cdot \dot{q}_t^e + n, \tag{4.15}$$

where K_p and K_d are proportional and derivative coefficient of PD controller respectively. n is a noise that is added to ensure sufficient exploration.

We have tuned the coefficients as $K_p = diag([10, \ldots, 10])$ and $K_d = diag([3, \ldots, 3])$. Trying to keep the neural network small, we choose the hyperparameters:

- network depth - 1 hidden layer with ReLU activation,
- layer size - 64 neurons,
- batch size - 128.

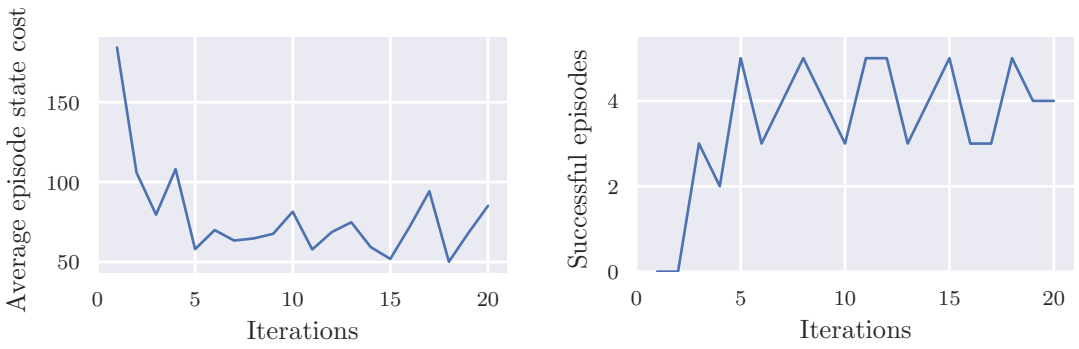
Adam optimization algorithm with its default parameters was used for this case as well. We train for 100 epochs.

Each experiment consists of 20 RL iterations, each consisting of 5 episodes. Every episode has a length of 100 time steps which equals to 5 s. To reduce the time it takes to train a new model after each iteration, only the data produced in the most recent five iterations is used for training, while the rest of the data is forgotten.

4.4.2 Optimization with random shooting

The weights used for the cost function are

- $w_1 = 1$ for the distance of the end effector from the target,



(a) State cost **(b)** Success rate

Figure 4.9: State cost and success rate over 20 iterations with random shooting method.

- $w_2 = 1$ for the joint position difference from the target joint position,
- $w_3 = 0.02$ for the joint velocities,
- $v_3 = 0.001$ for the control inputs.

In addition, we found that adding a Lorentzian ρ-function, significantly increases the rate of success. The weight for this term $w_l = 1$ and $\alpha = 10^{-5}$. The resulting cost function is

$$V_t = \sum_{i=t}^{t+H-1} (||p_t - p_r||^2 + ||q_t - q_r||^2 + 0.02 \cdot \dot{q}_t^2 + 0.001 \cdot u_t^2 + \log(||p_t - p_r||^2 + 10^{-5})). \quad (4.16)$$

The weights for joint velocity and control input are selected low not to seriously weaken the performance, while still reducing the average input cost and velocities.

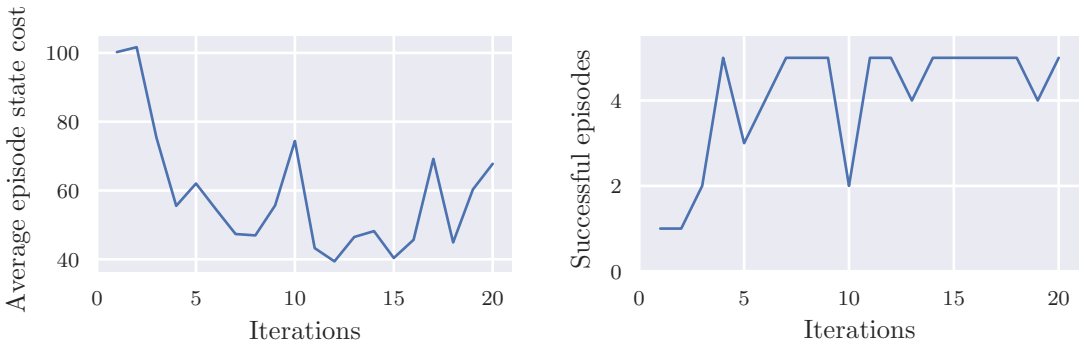
We set the time limit for computing one control input as 50 ms. We choose the horizon $H = 10$, which gives a good performance. Lower horizon can suffer from being too shortsighted, while with longer horizon model errors are more likely. We manage to increase the number of shootings to $K = 2000$, without violating the time limit.

The control inputs are drawn randomly from a uniform distribution. The limits of this distribution are different depending on the joint. Particularly, it is 6 N for the first three joints, 2 N for the fourth joint and 1 N for the last three joints. These limits were deliberately chosen small to keep the control inputs low, as limiting them this way is more efficient than penalizing the control inputs.

Figure 4.9 shows that the RL method is able to learn to complete the insertion task. As seen in Figure 4.9a, the average episode cost comes down in 5 iterations, after which there is no significant improvement in further iterations. Figure 4.9a displays in how many episodes out of five in each iteration the controller managed to complete the insertion task successfully. We see that after first 4 iterations, at least three episodes are always successful.

4.4.3 Optimization with CEM

With CEM the exact same cost function is used as with random shooting. We also keep the horizon same $H = 10$, but we can reduce the number of shootings to $K = 200$. Instead of setting a stopping criterion for CEM algorithm, we limit the number of iterations to 4. The initial variance is different for different control inputs and is designed to limit the actions to the similar range as with standard random shooting. This is achieved by selecting the initial variance vector as $\begin{bmatrix}36 & 36 & 36 & 4 & 1 & 1 & 1\end{bmatrix}$. Initial mean is zero and rarity parameter is 10.



(a) State cost **(b)** Success rate

Figure 4.10: State cost and success rate over 20 iterations with CEM.

As visible in Figure 4.10a, the average episode state cost comes down in 4 iterations. In iteration 10, the controller has probably learned a poor model and consequently completes the task successfully in only one episode out of five. With this exception the results are robust, with the controller being successful in 4 or 5 episodes out of five in every iteration after the fifth (Figure 4.10b).

4.4.4 Optimization with gradient descent

Since our implementation of gradient descent does not constrain the control inputs, we keep it in the desired range by using larger weight to penalize control inputs and we choose a similar cost function as follows:

$$V_t = \sum_{i=t}^{t+H-1} ||p_t - p_r||^2 + ||q_t - q_r||^2 + 0.02 \cdot \dot{q}_t^2 + 0.05 \cdot u_t^2 + \log(||p_t - p_r||^2 + 10^{-5}). \quad (4.17)$$

We use the same horizon as with other methods $H = 10$ and manage to do 9 gradient steps, while ensuring that the computation time does not exceed 50 ms. We use learning rate 0.5.

Figure 4.11 illustrates how the average episode cost decreases greatly in few iterations. It stays low, but after 8 iterations increases again and stays on higher level further on. The success rate also decreases. The reason might be that the gradient descent finds some local minimum and gets stuck in it.

4.4.5 Comparison

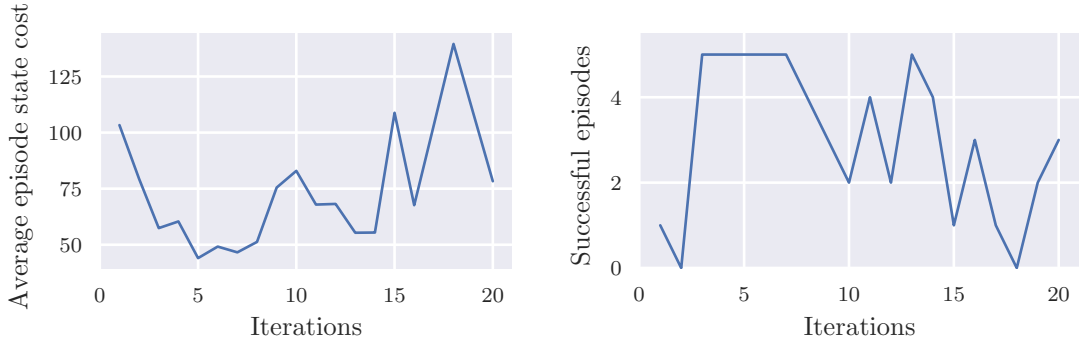
	Computation time (ms)	State cost	Input cost
Shooting	$31.6, \sigma = 4.1$	50.2	3658
CEM	$38.5, \sigma = 1.3$	39.4	2579
Gradient	$43.1, \sigma = 2.0$	44.2	764

Table 4.2: Comparison of different methods for insertion problem in simulation. Shown are average input calculation time, state cost and input cost in the best iteration.

The Table 4.2 compares the computation times, the average state costs and the average input costs. The average state cost over five episodes is taken from the best iteration, i.e. in the iteration where this cost is lowest. The average input cost is taken from the same iteration.

First when looking at the computation time, we see that in all cases they are more than 3 σ away from the limit of 50 ms. Thus, the risk of exceeding this limit is minimal as was intended with the choice of parameters. The average state cost in the best iteration is similarly low for CEM and gradient descent, while being a bit higher for random shooting. Considering the average input cost, the gradient descent method is surely the best, while the random shooting is the worst.

We expected CEM to give better results than standard random shooting. Indeed, it seems to achieve a bit lower average state cost as well as input cost. At the same time it is more data efficient: there are 4 iterations for 200 shootings, which means the predictions for $H = 10$ future states need to be made and their cost calculated 800 times. In contrast, for standard random shooting it is done 2000 times. We expected data efficiency to also lead to reduced computation time, however we see CEM having a higher computation time on average. This is probably caused by implementation issues. The predictions are calculated in TensorFlow, while rest of the code is not implemented in TensorFlow. Accessing TensorFlow takes time and in CEM it is done 4 times every time step, since there are 4 iterations, whilst



(a) State cost **(b)** Success rate

Figure 4.11: State cost and success rate over 20 iterations with gradient descent method.

in the random shooting there's only one iteration. We leave it for future work to implement the code in TensorFlow.

4.5 Insertion with obstacle

To add further solidity to the results, we decided to do another insertion experiment, with slightly alternated conditions. Particularly, we make a task more difficult by adding an obstacle into the end effector's path. The initial state stays the same, but there is a wall in front of the target hole. The policy has to plan the way around the wall. Figure 4.12 shows the task with the robot in its initial state. The results for all methods are presented in the figures below (Figures 4.13, 4.14, 4.15), while Table 4.3 compares the state cost and input cost for the best achieved iteration of each method.

All the tunable parameters, that define the model learning and optimization stay the same as is described in the previous section. The results are presented below, showing the state cost and success rate.

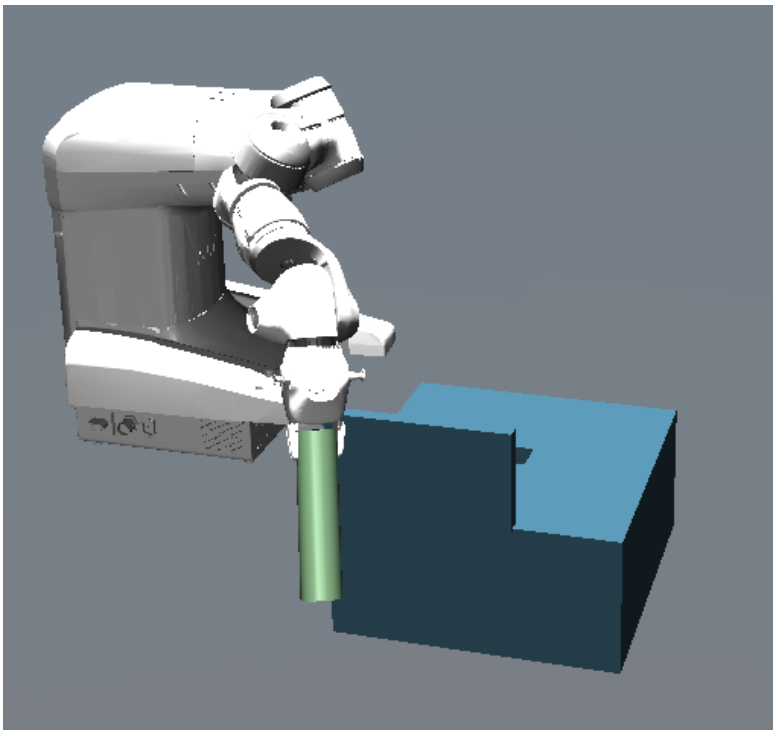
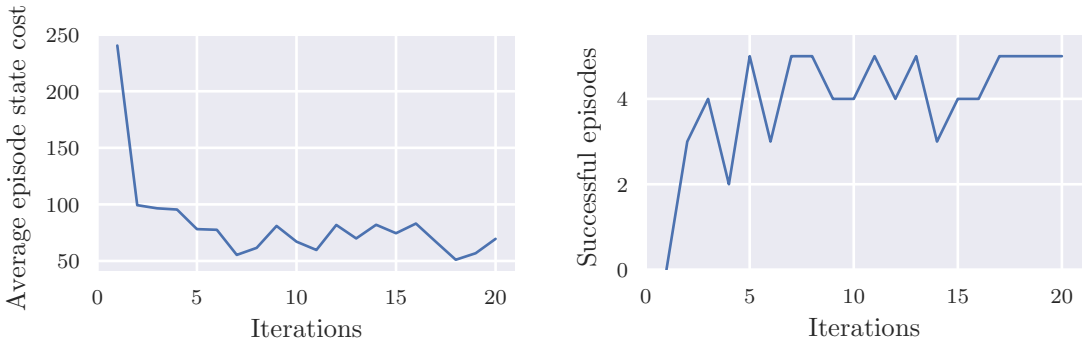
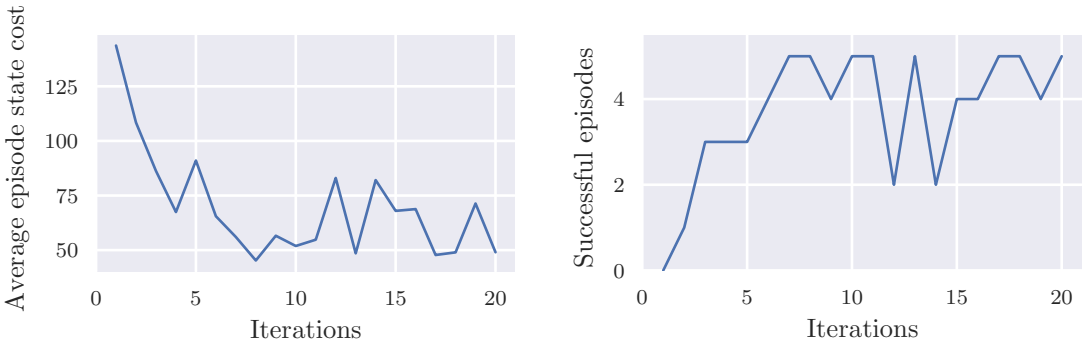


Figure 4.12: Initial position of the modified insertion task. A wall is added as an obstacle in front of the target.



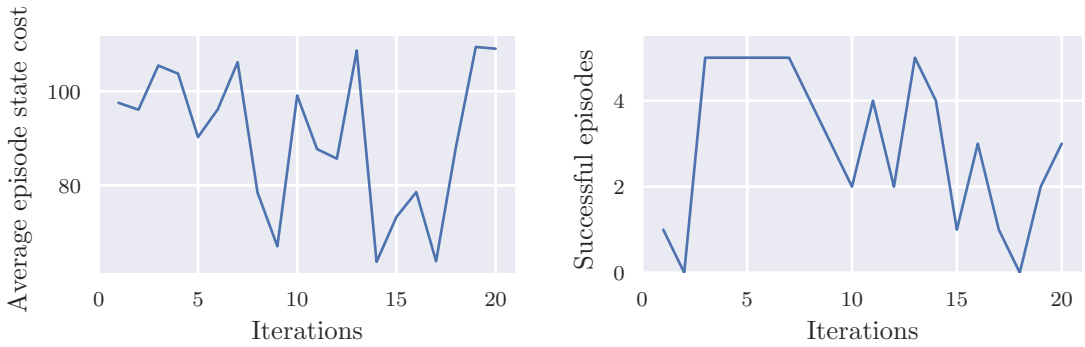
(a) State cost **(b)** Success rate

Figure 4.13: State cost and success rate over 20 iterations with random shooting method for obstacle task.



(a) State cost **(b)** Success rate

Figure 4.14: State cost and success rate over 20 iterations with CEM method for obstacle task.



(a) State cost **(b)** Success rate

Figure 4.15: State cost and success rate over 20 iterations with gradient descent method for obstacle task.

	State cost	Input cost
Shooting	51.1	3645
CEM	45.3	2846
Gradient	63.7	878

Table 4.3: Comparison of different methods for insertion problem with an obstacle. Shown are state cost and input cost in the best iteration.

We notice that the results are similar as in the case without the obstacle. Random shooting and CEM methods still achieve first successful result quickly, already in the second iteration. By 7th iteration both have achieved a 100 % successful iteration. The robustness of CEM is weaker than in the task without an obstacle, with more than one iteration in the latter part having only two successful episodes. Nevertheless, comparing the average state cost of the two methods, CEM is clearly better. Gradient descent achieves also very strong results in the beginning, but in the latter iterations is failing a lot. This is similar to the behavior noticed in the initial task.

5
Conclusion

The problem that this thesis sought solution for was controlling a robotics manipulator using reinforcement learning. The specific method used was a model-based reinforcement learning, which instead of learning a control policy directly, aims to learn the dynamics of the environment. Based on the learned dynamics model, an optimal policy is calculated online at each time step, using model predictive control.

For evaluation of the results, different experiments were conducted in a simulation environment. We started with a simple cart-pole balancing task, before we moved on to an insertion task with a manipulator. In the insertion task, the goal was to move a peg towards a target hole in a block and then insert the peg into the hole. In the final experiment, the situation was complicated further, by introducing an obstacle in front of the hole.

To learn the dynamics model, an artificial neural network was used. We noticed that even for the complex 7-joint manipulator, the neural network can learn a sufficiently accurate model with a simple structure consisting of only one layer.

Regarding online policy optimization, we compared three different methods: a gradient descent method, that iteratively calculates gradient of the cost function and moves in a negative direction of the gradient, a random-sampling shooting method that generates numerous candidate actions from a uniform distribution and picks the one that results in a minimal cost, and a cross-entropy method (CEM), which generates candidate actions from a normal distribution, while the distribution is narrowed iteratively to produce potentially better candidates.

When evaluating the methods, 10 iterations were run for the cart-pole task and 20 iterations for the insertion tasks. Each insertion consists of learning a new dynamics model based on the information acquired from the previous iterations, and making five attempts (episodes) to complete the task.

We found that all methods are able to solve the tasks. In the cart-pole task, no significant difference was observed between the three methods regarding the performance. In insertion tasks, the gradient descent method achieved good results in early iterations, but its performance decreased in later iterations, with many episodes failing. A possible explanation for the failure is that the gradient descent gets stuck in some local optimum. On the other hand, the gradient descent method used the lowest control inputs on average, which is positive for preserving energy.

We concluded, however, that the other two methods should be preferred, as the experiments showed more robust results in these cases. Out of the two, the CEM was regarded as better, because it achieved a smaller average state cost. More importantly, it is more data efficient, which should result in faster policy calculation. This is critical to ensure that the robot runs smoothly. We did not see that the CEM

was faster, when we measured the computation time, which we reasoned to be due to inefficient implementation of algorithm. A TensorFlow implementation could potentially reduce the computation time.

There is a lot of potential for future work with this project. Certainly the most interesting would be to test the developed method on a real robot. The learned model is critical to the performance, probabilistic models could serve better instead of a neural network. We offered one idea, how to reduce computation time. That would leave room to more possible calculations at each time step (more candidates, more iterations), leading to more optimal control actions.

Bibliography

[1] Jan Peters, Jens Kober, Katharina Mülling, Oliver Krämer, and Gerhard Neumann. Towards robot skill learning: From simple skills to table tennis. volume 8190, pages 627–631, 09 2013. doi: 10.1007/978-3-642-40994-3_42.

[2] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016. doi: 10.1038/nature16961.

[3] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32:1238–1274, 09 2013. doi: 10.1177/0278364913495721.

[4] Jens Kober, Erhan Oztop, and Jan Peters. Reinforcement learning to adjust robot movements to new situations. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Three*, IJCAI'11, pages 2650–2655. AAAI Press, 2011. ISBN 978-1-57735-515-1. doi: 10.5591/978-1-57735-516-8/IJCAI11-441. URL `http://dx.doi.org/10.5591/978-1-57735-516-8/IJCAI11-441`.

[5] H. Durrant-Whyte, N. Roy, and P. Abbeel. *Learning to Control a Low-Cost Manipulator Using Data-Efficient Reinforcement Learning*. MITP, 2012. ISBN 9780262305969. URL `https://ieeexplore.ieee.org/document/6301026`.

[6] Marc Deisenroth and Carl Edward Rasmussen. Pilco: A model-based and data-efficient approach to policy search. pages 465–472, 01 2011.

[7] Sergey Levine, Nolan Wagener, and Pieter Abbeel. Learning contact-rich manipulation skills with guided policy search. 2015.

[8] P. Pastor, H. Hoffmann, T. Asfour, and S. Schaal. Learning and generalization of motor skills by learning from demonstration. In *2009 IEEE International Conference on Robotics and Automation*, pages 763–768, May 2009. doi: 10.1109/ROBOT.2009.5152385.

[9] Sanket Kamthe and Marc Peter Deisenroth. Data-efficient reinforcement learning with probabilistic model predictive control, 2017.

[10] Roberto Calandra, Jan Peters, Carl Edward Rasmussen, and Marc Peter Deisenroth. Manifold gaussian processes for regression, 2014.

[11] Justin Fu, Sergey Levine, and Pieter Abbeel. One-shot learning of manipulation skills with online dynamics adaptation and neural network priors, 2015.

[12] Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning, 2017.

[13] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models, 2018.

[14] Aviv Tamar, Garrett Thomas, Tianhao Zhang, Sergey Levine, and Pieter Abbeel. Learning from the hindsight plan – episodic mpc improvement, 2016.

[15] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981.

[16] Vitchyr Pong, Shixiang Gu, Murtaza Dalal, and Sergey Levine. Temporal difference models: Model-free deep rl for model-based control, 2018.

[17] James B. Rawlings and David Q. Mayne. *Model Predictive Control: Theory and Design*. Nob Hill Publishing, Madison, Wisconsin, USA, 2015.

[18] Gábor Horváth. Neural networks in system identification. 2002.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[20] Stephane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning, 2010.

[21] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[22] Reuven Y. Rubinstein. Optimization of computer simulation models with rare events. *European Journal of Operations Research*, 99:89–112, 1996.

[23] Invertedpendulum-v2. URL `https://gym.openai.com/envs/InvertedPendulum-v2/`.

[24] Mujoco. URL `mujoco.org`.