

Evaluating Decision Transformer Architecture on Robot Learning Tasks

Evaluation der Decision Transformer Architektur für Lernende Roboter Aufgaben
Bachelor thesis in the field of study "Computational Engineering" by Max Siebenborn
Date of submission: March 7, 2022

1. Review: Boris Belousov, M.Sc.
2. Review: Junning Huang, M.Sc.
3. Review: Prof. Jan Peters, Ph.D.
Darmstadt



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Max Siebenborn, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, March 7, 2022

Max Siebenborn

Abstract

Decision Transformer (DT) is a recently proposed architecture for Reinforcement Learning (RL) that frames RL as an auto-regressive sequence modeling problem and uses a Transformer model to predict the next action in a sequence of states, actions and rewards. Despite of the appealing performance of DT in the original paper, our empirical evaluations of DT on real-time continuous control tasks show two issues of the original DT architecture: the DT architecture yields bad performances in fine-grained stabilization tasks around unstable equilibriums, and long response times are necessary to generate actions, which precludes using DT for real-world tasks. To address these issues, we propose an extension of DT, called Decision LSTM (DLSTM), an architecture that replaces the Transformer model inside DT by an Long Short-Term Memory Network (LSTM) architecture. We evaluate the performance of DT, DLSTM and a Behavioral Cloning (BC) architecture in offline RL stabilization tasks, as well as their real-time capabilities in a real-world pendulum setting. Empirically, we show that DLSTM outperforms both BC and DT and achieves expert level in the stabilization tasks. We conclude that framing RL as a sequence modeling problem in principle enables solving the fine-grained stabilization tasks, as demonstrated by the good performances of DLSTM. Also, significant improvements regarding faster response times can be achieved by using DLSTM instead of DT. However, the performance of these models proves to be highly dependent on the sequence modeling architectures that are used to make predictions.

Zusammenfassung

Decision Transformer ist eine kürzlich vorgestellte Architektur für Reinforcement Learning, die RL als ein auto-regressives Sequenzmodellierungsproblem formuliert und ein Transformer-Modell nutzt um die nächste Aktion in einer Sequenz aus Zuständen, Aktionen und Belohnungen vorherzusagen. Trotz der guten Performanz von DT im ursprünglichen Aufsatz, zeigen unsere empirischen Evaluationen von DT an Echtzeit-Systemen zwei Probleme auf: die DT-Architektur erzielt schlechte Leistungen in Stabilisierungs-Aufgaben um instabile Gleichgewichtslagen und benötigt verhältnismäßig lange Reaktionszeiten, um Aktionen zu generieren, was die Anwendung von DT für Aufgaben in der echten Welt verhindert. Um diesen Problemen entgegenzuwirken, führen wir eine Erweiterung von DT mit dem Namen DLSTM ein, welche das Transformer-Modell in DT durch eine LSTM-Architektur ersetzt. Wir evaluieren die Performanz von DT, DLSTM und einer Behavioral Cloning Architektur in offline RL Stabilisations-Aufgaben, sowie ihre Performanz an einem echten Pendulum. Unsere Experimente zeigen, dass DLSTM sowohl BC, als auch DT in den Stabilisations-Aufgaben übertrifft. Der Ansatz, RL als ein Sequenzmodellierungsproblem zu beschreiben, erweist sich als prinzipiell in der Lage, instabile Gleichgewichts-Aufgaben zu lösen, was durch die Erfolge von DLSTM belegt wird. Außerdem können durch die Nutzung von der LSTM-Architektur schnellere Reaktionszeiten erreicht werden. Andererseits ist die Leistungsfähigkeit der untersuchten Architekturen stark von der genutzten Sequenzmodellierungsarchitektur abhängig.

Contents

1	Introduction	2
2	Foundations	4
2.1	Reinforcement Learning	4
2.2	Offline Reinforcement Learning	8
2.3	Sequence Modeling	11
2.4	Decision Transformer	15
3	Related Work	19
3.1	Foundation Models	19
3.2	Extensions of Decision Transformer	20
4	Methodology	21
4.1	Environments	22
4.2	Datasets	26
4.3	Model Architectures	28
4.4	Training Process	29
4.5	Hyperparameters	30
4.6	Evaluation Process and Metrics	31
5	Experiments & Results	33
5.1	Expert Dataset Experiments	34
5.2	Replay Dataset Experiments	43
5.3	Real Robot Experiments	48
5.4	Influence of Returns-To-Go Values	54
6	Discussion	56
7	Conclusion & Outlook	59

Figures and Tables

List of Figures

4.1	Visualization of the different simulation environments from the experiments.	23
5.1	Average episode returns on mountain car expert.	34
5.2	Average episode returns on Mujoco inverted pendulum expert.	35
5.3	Average episode returns on OpenAI pendulum expert.	36
5.4	Phase plane trajectories on OpenAI pendulum expert.	38
5.5	Average episode returns on Furuta pendulum stabilization expert.	40
5.6	Phase plane trajectories on Furuta pendulum stabilization expert.	41
5.7	Average episode returns on Furuta pendulum swing-up expert.	42
5.8	Phase plane trajectories on Furuta pendulum swing-up expert.	43
5.9	Average episode returns on OpenAI mountain car replay.	44
5.10	Average episode returns on OpenAI pendulum replay.	45
5.11	Phase plane trajectories on OpenAI pendulum replay.	46
5.12	Average episode returns on Furuta pendulum swing-up replay.	47
5.13	Phase plane trajectories on Furuta pendulum swing-up replay.	48
5.14	Phase plane trajectories on Furuta Pendulum RR swing-up task with and without additional PD-aided stabilization.	50
5.15	Response times of the evaluated architectures in Furuta Pendulum RR. . .	53



5.16 Average episode returns under different RTG values.	55
--	----

List of Tables

4.1 Overview of the used training datasets for the stabilization experiments. . .	27
4.2 Overview of the used hyperparameters for the different evaluated architectures.	31
5.1 Overview of the experimental results of the final models in the different simulation environments.	33
5.2 Overview of the experimental results of the final models on Furuta Pendulum RR.	49

Abbreviations, Symbols and Operators

Acronyms

BC	Behavioral Cloning.
DL	Deep Learning.
DLSTM	Decision LSTM.
DP	Dynamic Programming.
DT	Decision Transformer.
Furuta Pendulum RR	Furuta pendulum real-world setup.
LSTM	Long Short-Term Memory Network.
MC	Monte Carlo.
MDP	Markov Decision Process.
ML	Machine Learning.
MLP	Multi-Layer Perceptron.
MSE	Mean Squared Error.
NLP	Natural Language Processing.

PD	Proportional-Derivative controller.
PPO	Proximal Policy Optimization.
RL	Reinforcement Learning.
RNN	Recurrent Neural Network.
RTG	Returns-to-go.
SACo	Relative State-Action Coverage.
SL	Supervised Learning.
TD	Temporal Difference Learning.
TQ	Relative Trajectory Quality.
UL	Unsupervised Learning.

1 Introduction

Transformers [1] are a neural network architecture that has shown impressive successes across many domains such as Natural Language Processing (NLP) [2, 3, 4] and Computer Vision [5], and they have become state-of-the-art architectures for sequence modeling problems. Inspired by these successes, there have been two papers [6, 7] that frame RL as a sequence modeling problem, in which models predict the next element in a sequence of states, actions and rewards. In [6], DT is proposed, an architecture that uses a GPT-2 Transformer [8] to auto-regressively model sequences of states, actions and rewards. DT works in the field of offline RL where an agent is limited to learning from a fixed dataset of trajectories that were generated by another policy, and can not perform additional exploration and exploitation by itself.

In the original paper, the performance of DT is evaluated on tasks from the D4RL dataset [9]. The evaluations include discrete Atari tasks as well as continuous control tasks in OpenAI gym [10]. However, from these experiments it remains unclear whether DT is also competitive for dynamic tasks that require stabilization of systems around an unstable equilibrium, as well as for real robot control tasks.

In this thesis, we address the evaluation of DT for Robot Learning tasks, where we focus on two aspects. First, we evaluate DT on *stabilization tasks*, more precisely on different pendulum environments. The goal of an agent in these tasks is to reach an unstable equilibrium target state, and remain in this state. Second, we validate our experimental results in a *real-world robotic setting*. This evaluation is crucial since large gaps between simulations and real world systems are common in robotics and RL [11]. Moreover, important measures such as the computational capability of the learned models to generate actions in real-time are not accounted for in simulated environments.

In addition to our evaluations of DT on stabilization tasks and a real robot setting, we propose a related architecture to DT, called DLSTM, which builds on top of the DT architecture but replaces the GPT-2 Transformer by an LSTM model. We use DLSTM to test whether framing RL as a sequence modeling problem allows using other sequence

modeling architectures than Transformers, and if these other architectures can yield better results. We compare the performances of both DT and DLSTM against each other, and against the performance of a simple BC model which mimics actions based on the observed states.

The structure of this thesis is as follows. In Chapter 2, we provide an overview over the fundamentals of RL, offline RL and sequence modeling to then explain the DT architecture. In Chapter 3, we outline related work including extensions of DT and foundation models. Then, in Chapter 4 we introduce the methodology behind our conducted experiments. We describe the experimental environments and datasets as well as the evaluated architectures including their training and evaluation process. In Chapter 5, we present and analyze our experimental results for the simulated and the real system tasks. These results are further discussed in Chapter 6. Finally, Chapter 7 provides a conclusion of the experimental results and an outlook on further research regarding the use of DT and related architectures in RL.

2 Foundations

In this section, we provide an overview of RL, offline RL, sequence modeling, and the DT approach.

2.1 Reinforcement Learning

Reinforcement Learning, which forms a subfield of Artificial Intelligence, deals with sequential decision making problems under uncertainty where the problem setting can be described as follows. An *agent* explores its *environment* by applying *actions* based on the current *state* the agent finds itself in. For every time step, the agent receives a *reward*, which is given by the environment for the last taken action and the outcome of this action, i.e., the state following the action. This reward can be *discounted*, such that the reward of an action is smaller if it is taken at a later time step than the reward for taking this action at an earlier time step. An *episode* is defined as a sequence of states, actions and rewards which leads to a terminal state. The behavior of the agent is determined by the agent's *policy*. For every episode, the objective of the agent is to maximize the expected cumulative episode rewards, i.e., the episode *return*. To obtain high returns, the agent distinguishes which actions are good and bad by mapping them to the obtained rewards, which enables the agent to learn a desired behavior by applying actions which lead to high rewards, and avoiding actions that lead to low rewards. In this *trial-and-error* approach, the RL algorithm has to trade off between *exploration* and *exploitation*. During exploration, the agent takes actions that are suboptimal in regard to the currently learned policy, in hope of exploring new state-action pairs that lead to even higher rewards in the long-term. During exploitation, the agent uses the currently learned policy to generate actions that yield high rewards.

The setting of RL therefore distinguishes itself both from Supervised Learning (SL), where the goal is to generalize predictions to unseen inputs while learning from a fixed expert

dataset, and from Unsupervised Learning (UL), where the goal is to find patterns in unlabeled input data. According to [12], the distinguishing features of RL are trial-and-error search through exploration and exploitation, and the delayed reward signal.

2.1.1 Mathematical Formulation of Reinforcement Learning as Markov Decision Process

The following definitions align with the standard RL textbook definitions from Sutton *et al.* [12].

RL problems are mathematically described as Markov Decision Processes (MDPs), where an MDP is characterized as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \gamma)$. In this tuple,

- \mathcal{S} is the set of states (the *state space*),
- \mathcal{A} is the set of actions (the *action space*),
- \mathcal{R} is the set of immediate rewards R which the agent obtains for applying certain actions in certain states,
- $p(s', r|s, a) = \Pr(s_t = s', R_t = r | s_{t-1} = s, a_{t-1} = a)$ is a probability distribution which defines the probability of transitioning into the state s' and receiving the reward r after applying the action a when starting in state s . The *dynamics* of the MDP are defined by the probability distribution p .
- And $\gamma \in [0, 1]$ is the *discount factor*. This factor discounts future rewards such that high rewards, which are obtained in later time steps, are considered to be worth less in comparison to immediate high rewards.

This definition applies for *finite* MDPs where the sets \mathcal{S} , \mathcal{A} and \mathcal{R} all have a finite number of elements.

MDPs have the *Markov property* which states that the transition dynamics p only depend on the previous state s_{t-1} and the applied action a_{t-1} , but not directly on any other previous or future states and actions. In this regard, states are assumed to implicitly contain all the information about past actions and states.

RL policies $\pi(a|s)$ are described as distributions that assign probability to taking the action $a_t = a$ at time step t when the current state is $s_t = s$.

The objective of an RL agent is to maximize the expected return, i.e., the cumulative discounted sum of rewards in the next time steps. The return is mathematically defined as

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \gamma G_{t+1}. \quad (2.1)$$

For $\gamma = 1$, the discounted return corresponds to the undiscounted return. Note that the summation is infinite, since generally episodes can have an *infinite horizon*. For problems with a *finite horizon*, i.e., episodes with a fixed final time step T , we can also use the infinite summation of Equation 2.1 by considering all rewards $\{R_t : t > T\}$ to be zero.

We can define the objective of RL algorithms to maximize the expected episode return:

$$J(\pi) = \mathbb{E}_{\pi}[G_0] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{k+1}\right],$$

where we can again use this formula for finite horizon problems.

2.1.2 Bellman Equations of Optimality

Value functions express how good it is for an agent to be in a specific state, or taking a specific action when being in a specific state, under a given policy π . The *state value function*

$$V_{\pi}(s) = \mathbb{E}_{\pi}[G_t \mid S_t = s], \forall s \in \mathcal{S}$$

denotes the expected return when the agent is in state s at time step t , and afterwards follows the commands of the policy π . The *action value function*

$$Q_{\pi}(s, a) = \mathbb{E}_{\pi}[G_t \mid S_t = s, A_t = a], \forall s \in \mathcal{S}, a \in \mathcal{A}$$

expresses the expected return of taking the action a after starting in state s , and afterwards following the policy π .

To find an *optimal* policy, *Bellman's principle of optimality* [13] can be used. Bellman's principle of optimality states that independent of the initial state and action, the remaining decisions of an optimal policy must be optimal regarding the state which results from the initial action. The optimal value functions can be defined recursively, such that

$$\begin{aligned} V^*(s) &= \max_a Q_{\pi^*}(s, a) \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma V^*(S_{t+1}) \mid S_t = s, A_t = a] \end{aligned} \quad (2.2)$$

describes the optimal state-value function, and

$$Q^*(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') \mid S_t = s, A_t = a] \quad (2.3)$$

describes the optimal state-action value function.

2.1.3 Fundamental Reinforcement Learning Approaches

Equations 2.2 and 2.3, which derive from the Bellman optimality principle, provide conditions for optimal solutions for RL problems. Dynamic Programming (DP) methods calculate the value functions iteratively to obtain optimal policies. DP has two main components: in *policy evaluation*, the state-value function V^π is computed for the current policy π by iterating the Bellman equation

$$V^\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma V^\pi(S_{t+1}) \mid S_t = s]$$

until convergence to the stationary point. In *policy improvement*, the policy is *greedily* improved such that the actions which promise to lead to the highest return are taken. The repeated process of performing policy evaluation and policy improvement is called *policy iteration*.

Another related approach is *value iteration*, where the state-value function is directly iterated to be

$$V_{k+1}(s) = \max_a \mathbb{E}[R_{t+1} + \gamma V_k(S_{t+1}) \mid S_t = s, A_t = a].$$

Independent of the initialization of the value function V_0 , the state-value function converges to the optimal state-value function V^* from which an optimal policy π^* can be extracted.

Naively applying policy iteration and value iteration is not feasible for most problems, since these methods require full prior knowledge of the problem's dynamics. However, DP represents the fundamental idea behind most RL algorithms.

In contrast, Monte Carlo (MC) methods do not assume any knowledge of the system dynamics, but base the whole learning process on learning from experience, i.e., sampled observations from interaction with the environment. MC methods can learn value functions from the sample returns.

Temporal Difference Learning (TD) methods include aspects from both DP and MC methods: they directly learn from sampled experience, similar to MC approaches, and perform updates of the value functions based on other learned estimates, similar to DP approaches. In addition, TD methods do not wait for the final outcome to update their estimates such as MC methods do, but rather bootstrap and learn before obtaining the final outcome.

2.2 Offline Reinforcement Learning

In *offline* RL, the agent can not explore its environment directly, but rather learns from a fixed dataset of trajectories which were generated by another policy. Offline RL is motivated by two aspects.

1. RL should be enabled to make use of large, previously collected datasets, such as it has lead to great successes in Supervised Learning.
2. Offline RL especially becomes important in scenarios where access to the environment is infeasible which prevents the agent from exploring and interacting with the environment to learn. An approach to dealing with this issue is to simulate the environment and let the agent learn there. However, simulators are often not available, and learning results suffer from the *sim to real gap* [11], which occurs due to wrong or missing modeling assumptions when creating the simulators. Here, offline RL promises to be an efficient way to make use of RL without requiring access to the environment.

2.2.1 Terminology

As [14] points out, there exist two related terms for *offline RL*. To avoid confusion, the terms are briefly described in the following.

- *Off-policy* RL refers to algorithms which in some sense make use of data that was generated from other policies than the learned policy. In contrast, in *on-policy* RL the exploring *behavior policy* is identical to the learned *target policy*. Off-policy RL can be seen as a generalization of offline RL, where offline RL is the special case of off-policy RL where the agent *only* has access to data from other policies, and can not perform any additional online exploration and exploitation.

-
- *Batch* RL is synonymous to offline RL.

2.2.2 Formulation

As stated in [14], offline RL is a *data-driven* formulation of RL. Rather than exploring the state-action space through direct interaction with the environment, offline RL algorithms learn from a static training dataset $\mathcal{D} = \{(S_i, A_i, S_{i+1}, R_i)\}$, which consists of transition-tuples containing the state S_i , the taken action A_i , the state after transition S_{i+1} , and the reward for the transition R_i . The *behavior policy*, i.e., the policy that generated the dataset's state-action pairs, is denoted as π_β , and the *learned policy* of the offline RL algorithm is denoted as π_θ . As in classical online RL, the offline RL algorithm attempts to learn a policy π_θ which maximizes the RL objective, i.e., the episode return.

2.2.3 Common Offline RL Approaches

In [14], common current approaches to offline RL are presented.

- **Importance Sampling Approaches.** The idea behind importance-sampling-based offline RL algorithms is that the policy tries to directly estimate its own return in offline manner, i.e., without directly evaluating its performance in the environment. After evaluation, the algorithm can select the expected best policy, i.e., the policy with the highest performance. Since sampling from the learned policy π_θ itself is not possible in the offline setting, the respective algorithms perform importance sampling using trajectories from the behavior policy π_β . Using this approach, the algorithms either estimate the RL objective $J(\pi_\theta)$ of the learned policy directly, or estimate its gradient $\nabla J(\pi_\theta)$ for gradient descent methods.
- **Dynamic Programming Approaches.** DP methods, as described in Section 2.1, can be utilized in offline RL. Using DP, optimal policies can be derived by learning the state-value function and the state-action-value function of the MDP. Offline RL algorithms either constrain the learned policy π_θ to be close to the behavior policy π_β , or evaluate the uncertainty in the estimates of the learned value functions to avoid distributional shift between π_θ and π_β , where distributional shift occurs when π_θ leads the agent into states that are not covered in the training dataset.

-
-
- **Model-Based Approaches.** In model-based RL, the RL algorithm learns a dynamics model from data, which can then be used to optimize the learned policy, or for planning. Using model-based approaches, prior knowledge on the environment dynamics can be incorporated into RL algorithms. To avoid the problems caused by wrong modeling assumptions, model-based RL approaches can estimate the uncertainty in the states and actions which were produced by the model to prevent learning from highly improbable data. In offline RL, one can directly use standard model-based RL approaches, or use the models for off-policy evaluation when online evaluation is not wanted or impossible.

2.2.4 Common Challenges in Offline Reinforcement Learning

Since classical online RL relies on direct interaction between the agent and its environment, and this interaction is not possible in the offline RL setting, *offline* RL is inherently prone to difficulties.

- **Distributional Shift.** The fundamental problem of offline RL occurs when the agent encounters unseen states at test time. Online methods deal with this problem by exploring possible actions in these states and learning from the received rewards. However, offline methods are limited to the fixed training dataset \mathcal{D} , which might not contain an optimal action, or even any action, for the particular state the agent currently finds itself in. When the offline RL agent at test time enters a state which is not contained in the dataset, i.e., the agent is out-of-distribution of the dataset's behavior policy π_β , the agent will likely take a suboptimal action, which endangers the agent to get even wider out of distribution (e.g., when it enters a state that is farther away of the state space which is covered by the dataset trajectories). In a broader sense, this problem is called *distributional shift*, and exists due to the fact that the behavior policy which generated the learning data, and the policy which is actually learning from this data, are different from each other. There are multiple approaches to limit distributional shift in offline RL, where a simple idea is to limit the learned policy π_θ such that it stays to a certain degree similar to the behavior policy π_β [14].
- **Limited Exploration.** Due to the missing direct access to the environment in the offline setting, the offline RL policy is limited to the state-action coverage of the training dataset, and it can not perform any additional exploration. Most offline RL algorithms therefore assume that the training dataset provides a sufficient coverage of the high reward regions of the environment's state-action space [7].

-
- **Influence of the Dataset Quality.** As [15] points out, certain characteristics of the training dataset, mainly the quality of the trajectories in terms of episode returns and the coverage of the state-action space, are crucial to the performance of offline RL algorithms. The performance of offline RL policies must therefore always be evaluated in comparison to the characteristics of the training dataset. Online RL algorithms are in principle not limited to any such characteristics, which makes them easier to compare and evaluate.
 - **Offline Evaluation.** In cases where access to the real setting or simulators for evaluation is infeasible, evaluating the learned policy is a hard task. This becomes especially critical for model validation and policy selection which are usually done by evaluating the learned models directly in the environment [16]. Offline evaluation is critically important for many RL algorithms which rely on estimating the RL objective $J(\pi_\theta)$ or its gradient $\nabla J(\pi_\theta)$. The problem of evaluating policies in an offline setting is addressed by *off-policy policy evaluation* [17].

2.2.5 Benchmarks and Datasets

To compare different offline RL methods, it is necessary to have comparable datasets and benchmarks. In NLP, such datasets are common to compare architectures across different domains. Examples are the GLUE benchmark [18], and decaNLP [19].

A notable framework for benchmarking offline RL methods is D4RL [9], which consists of standardized environments and datasets for a wide range of control and planning tasks, building on top of OpenAI gym [10]. Similarly, RL Unplugged [16] provides a benchmark suite to compare offline RL algorithms in different domains. SaLinA [20] is another library for learning agents in sequential decision processes which emphasizes sequential agents in RL.

2.3 Sequence Modeling

In sequence modeling [21], the task is to model sequences where sequences consist of a number of ordered *sequence elements*. Prominent examples of sequence modeling appear in NLP, where, for instance, a sentence can be considered as a sequence of words and sequence models predict the next word in this sequence by taking the previous words of the sentence as an input [22].

More precisely, we consider the task of predicting the next sequence element \hat{x}_{t+1} given the previous sequence elements x_i , by learning a function f that describes a temporal dynamics model

$$\hat{x}_{t+1} = f(x_1, \dots, x_t). \quad (2.4)$$

Note that here f processes inputs of varying length, since the number of sequence elements which the model receives as input increases by 1 at each time step. Sequence models are called *auto-regressive* if the outputs of the function f depend on its previous outputs. In Equation 2.4, these previous outputs are denoted as function inputs x_i , and in an auto-regressive sequence modeling setting they were generated by the function f .

In this section, we present common architectures which are used in sequence modeling. In Section 2.4, we then describe how RL can be abstracted as a sequence modeling problem.

2.3.1 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) [23, 24] are neural networks that implement recurrent structures to make information inside the network persistent. Therefore, RNNs are a natural choice for modeling sequences since sequence elements can be processed one after another, while information from previous sequence elements remains in the network. Also, unlike standard neural networks, RNNs are not limited to fixed input sizes but can in principle process inputs of arbitrary length. RNNs can be visualized as a sequential structure, in which, at every time step t , a network layer gets fed with two vectors: first, the input vector x_t , i.e., the (embedded) t -th element of the input sequence, and second the hidden state h_{t-1} which is the output from the previous layer. The drawback of standard RNNs is that they have a small *context*, i.e., the amount of information the network can attend to at a certain position in the sequence is limited to only the last few sequence elements.

2.3.2 Long Short-Term Memory Networks

LSTMs [25] overcome the small context issue of RNNs by refining the internal structure of the RNN, such that, in addition to the hidden state of the network, there is a *cell state* C_t which stores information along the sequence. The cell state is regularly updated after each layer, and the information flow through the network is controlled by three *gates*.

1. The *forget gate* decides, based on the last hidden state C_t and the input x_t , which information should be removed from the cell state.

-
-
2. The *input gate* decides which values from the cell state should be updated.
 3. The *output gate* determines the output of the current layer, based on the cell state. This output is then passed to the next RNN layer.

Through this mechanism, DLSTMs overcome the problem of vanishing and exploding gradients. We limit ourselves to this intuitive explanation of LSTMs, and refer to [25, 26] for the mathematical details on how information is processed inside LSTMs. We put emphasis on three aspects of RNNs and LSTMs that are important for this thesis.

- RNNs are a good choice for modeling sequential data since they are designed to process data sequentially while retaining important information of previous sequence elements to later attend to this information.
- LSTM is a widely used modified RNN model which has shown to perform well on a variety of sequence modeling and NLP tasks [27, 28].
- The most important bottleneck of LSTMs is that they process data sequentially, which precludes parallelization and often leads to long training times.

2.3.3 Transformers and (Self-) Attention

Transformers [1] represent an architecture for auto-regressive sequence modeling that is purely based on the *attention* mechanism. Other than most previous sequence modeling architectures, Transformers do not contain any recurrent or convolutional structures.

In general, an attention module maps a set of query vectors q , keys k and values v to an output value. The output is obtained as a weighted sum over the values, where the weights are computed via a measure of similarity between the query vector q and the key vector k_i which corresponds to the value vector v_i . Precisely, Transformers compute attention via *scaled dot product attention* according to the equation

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V,$$

where Q, K, V are matrices which contain the queries, keys and vectors respectively, and d_k is the dimension of the keys. Intuitively, the multiplication QK^T represents the matrix form of the dot-product between queries and keys, which itself is a measure of the similarity between queries and keys. This matrix product is scaled by d_k to avoid large intermediary results which would cause very small gradients. The softmax is then taken

to map the computed similarities to a probability distribution which computes the weights for each query-key pair. These weights are then used to compute the final attention output via weighted summation in matrix form, i.e., via matrix multiplication with the value matrix V . This way, every key-value pair *attends* to every query.

Transformers extend this notion of attention to *multi-headed attention*. In multi-headed attention, the original Q , K and V matrices are linearly projected h times into new subspaces, where h is the number of attention heads. For each attention head a different linear projection is used. After that, for each head the attention output is computed using Equation 2.3.3. These outputs are then concatenated and, using another linear projection, mapped to a single attention output again. Multi-head attention has the advantage that it allows the Transformer model to detect different dependencies in the sequence, and not only focus on one. Also, the head's dimension is reduced such that the total computational time is equal to that of a single attention module with full dimensionality.

Transformers consist of an encoder, which maps the Transformer inputs to an internal representation, and a decoder, which maps an internal representation to the outputs of the Transformer. Both the encoder and the decoder side contain multiple stacked attention layers. In this context, attention is used in two ways.

1. In encoder-decoder attention layers, queries come from the decoder side, and keys and values, which represents the Transformer inputs, come from the encoder side.
2. *Self-attention* modules are used both on the encoder and the decoder side. On the encoder side, self-attention is used to attend all positions of the current attention layer to all positions from the previous attention layer. On the decoder side, self-attention makes each position in the decoder attend to all other positions which are previous to this position because on the decoder side, access should only be allowed to positions which are already included in the decoder's output sequence, which resembles the generation of a sentence from left to right. Transformers ensure that only previous position are attended by using an attention mask that forbids attending to future positions in the sequence.

Since Transformers do not have a recurrent or convolutional structure, sequences are not required to be processed in sequential order. However, Transformers need to compute positional embeddings for the inputs to retain the sequence order. These embeddings can either be predefined, such as the (co-)sinusoid embeddings for the original Transformer architecture [1], or they can be learned. Using self-attention, the number of sequential operations becomes constant and is no longer linearly dependent on the length of the input sequence, as it is the case for recurrent and convolutional structures. Also, the

computational complexity for each layer becomes linear in the input dimensionality and quadratic in the input length, whereas it is linear in the input dimensionality and quadratic in the input length for recurrent and convolutional layers. This is advantageous for short inputs with high representation dimensions (such as it is the case for natural language sentences), but can be problematic for long input sequences (such as sequences of states, actions and rewards for RL problems). Still, with these properties the learning and evaluation process of Transformers can be parallelized over the different input items, which is a major computational advantage over other recurrent architectures.

Transformers have become state-of the art models in sequence modeling and NLP over the last years, as shown by the successes of BERT [2] and the GPT-x architectures [3, 8]. Both architectures build on top of the original Transformer model and implement a two-step learning process. First, the models are pre-trained on an unsupervised large training corpus to obtain a first estimate of the model parameters. In a second step, this pre-trained model is fine-tuned on smaller datasets which are specific to the concrete task that should be solved with the models, in a supervised learning setting. While there are also approaches towards transfer learning using LSTMs [29, 30], Transformer models that are pre-trained on large datasets are more common than pre-trained LSTMs models.

2.4 Decision Transformer

Decision Transformer [6] is an architecture for model-free offline RL which is proposed as an approach to solve RL problems without making use of conventional methods such as DP and TD that learn a value functions for this purpose.

2.4.1 Reinforcement Learning as Sequence Modeling

DT frames the RL problems as a sequence modeling problem, where a model auto-regressively models the joint distribution of states, actions and rewards, given a sequence of previous states, actions, and rewards.

A trajectory in the DT framework is represented as a sequence

$$\tau = (\hat{R}_1, s_1, a_1, \hat{R}_2, s_2, a_2, \dots, \hat{R}_T, s_T, a_T), \quad (2.5)$$

where \hat{R}_t is the Returns-to-go (RTG) value, i.e., the sum of the (desired) future rewards, s_t is the state, and a_t is the action at time step t respectively.

From an auto-regressive viewpoint, this formulation of the trajectory is appropriate, since it follows the order of the RL process at time step t . First, an RTG value \hat{R}_t is specified that tells the (expected) cumulative sum of future reward in this episode. This RTG value represents *hindsight information*, i.e., knowledge about future time steps (the future return) is introduced into the current predictions. Given \hat{R}_t and the current state s_t of the agent, the DT model then predicts an action a_t , which leads to a reward signal R_t . This reward signal is included into the next RTG value \hat{R}_{t+1} by subtracting the obtained reward from the previous RTG value, i.e., $\hat{R}_{t+1} = \hat{R}_t - R_t$. Rewards are not discounted in the DT model. The transition dynamics in the MDP also lead the agent into a new state s_{t+1} . From there on, the auto-regressive dynamics of the sequence generation is repeated again, where the action a_{t+1} is predicted from the RTG \hat{R}_{t+1} and the state s_{t+1} .

In DT, the generation of the sequence is conditioned on the future rewards in form of RTG value, similar to as it is the case in other related approaches [31, 32, 33]. Another model that frames RL as a sequence modeling problem and is highly similar to the DT approach is Trajectory Transformer [7]. The main difference between the approaches is that Trajectory Transformer does not use the RTG formulation for sequence generation like DT, but rather uses beam-search planning. Also, Trajectory Transformer uses state and reward prediction as well as discretization, which is not the case for DT.

2.4.2 Motivation Behind Decision Transformer

The key motivation of the DT approach can be summarized to two aspects.

1. In sequence modeling and NLP, there has been tremendous progress in recent years, especially through pre-trained Transformer networks such as BERT and GPT-x. DT proposes a framework in which RL can benefit from these advances through framing RL as a sequence modeling problem.
2. One of the main ideas behind framing RL as a sequence modeling problem and using a Transformer architecture is that the problem of *long-term credit assignment* can be solved efficiently using this setup. The long-term credit assignment problem states that for classical RL algorithms, it is generally hard to determine which of the previously taken actions of a sequence are responsible for the episode return. The Transformer architecture allows to explicitly find the relationships between states, actions, and rewards in a sequence using its attention module, which makes the solution to the long-term credit assignment problem inherent to DT, and avoids

the need of performing direct credit assignment or using Bellman backups for this purpose.

DT is as an expressive, yet simple implementation of the presented idea of framing RL as generative trajectory modeling for model-free offline RL. However, the general approach behind DT does not forbid using a modified version of DT in an online setting, and can be extended to work in online settings.

2.4.3 Architecture Details

The DT architecture accepts a sequence of RTG values, states and actions as input. For each modality (i.e., RTG, states and actions), DT gets fed with the tokens from the previous K time steps such that the input length is $3K$. We denote K to be the *context length* of DT. The inputs are mapped via linear layers to internal representations, where a separate linear layer is learned for each modality. The internal representations of the modalities are then stacked such that the trajectories are ordered according to the trajectory representation in Equation 2.5. In addition, the last K time steps are fed into the model to obtain positional embeddings for the tokens. For the positional embeddings, the time steps are processed by an embedding layer, and then added to the trajectory representation, similar to how positional embeddings are added to the representation of natural languages sentences in [1]. The obtained input sequence is then processed by a GPT-2 Transformer model [8]. The output of the GPT-2 model is ultimately fed into a final linear layer, which predicts an action based on the obtained trajectory representation.

2.4.4 Training Process

The training process for DT is fully offline, i.e., the model learns from a fixed dataset \mathcal{D} containing previously collected trajectories. For one training epoch, DT samples n mini-batches of length K from the dataset. These mini-batches are propagated through the architecture in a forward pass, which results in an action prediction a_{pred} for each mini-batch. Then, the loss is computed as the Mean Squared Error (MSE) between the predicted action a_{pred} , and the actual next action in the dataset $a_{\mathcal{D}}$:

$$\text{MSE} = \frac{1}{n} \sum (a_{\text{pred}} - a_{\mathcal{D}})^2.$$

Using the computed loss, a backward pass is performed, in which the DT model's parameters are updated according to the MSE. The forward-backward pass scheme is repeated multiple times for each training epoch.

2.4.5 Decision Transformer Performance Results

The authors show empirical evaluation results of the performance of DT on the D4RL dataset, specifically on continuous OpenAI Gym control tasks, as well as discrete tasks inside Atari games. The evaluation results indicate that DT performs comparably to state-of-the-art model-free offline reinforcement learning algorithms and even outperforms some of them. Also, DT is evaluated to be particularly powerful in sparse reward settings, as well as settings where long-term credit assignment is necessary.

3 Related Work

In this section we name several extensions of the original DT architecture and reference research on foundation models, i.e., powerful pre-trained models that can be applied to a variety of downstream tasks.

3.1 Foundation Models

Foundation models are models which are pre-trained on large datasets and used for *transfer learning*, in which these models can be adapted to a wide range of downstream tasks. DT, and especially its extensions from Section 3.2, can be viewed as foundation models, since they provide a framework to solve a variety of tasks using transfer learning. Also, the underlying Transformer model of DT, the GPT-2 architecture, is a foundation model itself.

In [34], an overview of the chances and risks of foundation models is provided and it is outlined that foundation models are applicable to a wide variety of tasks, including language, vision, and robotics. However, foundation models have several risks, e.g., it is critical that errors in the originally pre-trained model will also be integrated into the fine-tuned models. Also, foundation models might bear harmful problems with societal impact, such as ethical issues. It is concluded that foundation models require lots of future research with a strong emphasis on interdisciplinary collaboration.

Additionally to Transformer-based foundation models like BERT and GPT-x, as described in Section 2.3, recently Perceiver [35] and Perceiver IO [36] have been proposed. The Perceiver architectures are foundation models that are intended to be compatible with arbitrary inputs from different domains, like vision, language, and touch. Perceiver and Perceiver IO build on top of Transformer networks. Their main idea is to reduce the computational complexity of the architecture to be linear in the input size, other than

regular Transformers which scale quadratically with the input size [1]. Perceiver achieves the linear complexity in the input length by introducing latent units which build an *attention bottleneck*. The inputs are attended iteratively, which allows Perceivers to only consider the most relevant inputs. While the Perceiver architecture can only produce simple outputs, like class labels, Perceiver IO is an extension that is able to produce arbitrary outputs and scales linearly in both the input and the output size.

3.2 Extensions of Decision Transformer

As stated in Section 2.4, the original DT represents a rather simple approach to framing RL as sequence modeling. The DT approach allows for further extensions in multiple directions, some of which we reference here.

In [37], it is stated that DT belongs to a class of RL algorithms which perform hindsight information matching by generating trajectories that match some statistics of future state information. The standard DT architecture is expanded to *generalized DT*, an architecture that is in principle able to solve arbitrary hindsight information matching problems. The notion of the reward in the original DT is generalized to be an arbitrary feature function $\Phi(s, a)$, and the γ -discounted summation over rewards is generalized to an arbitrary aggregation operation. Through different choices of the feature function $\Phi(s, a)$ and the aggregation function, the authors propose different DT versions for hindsight information matching problems.

Online DT [38] presents a DT implementation which combines offline pre-trained DT models with an online fine-tuning process of the model's parameters. This combined approach promises to overcome the distributional shift in DT, since additional exploration can be performed in the online fine-tuning process, while remaining the advantages of offline RL, such as making use of large previously collected datasets. Similarly, [39] proposes an approach that combines offline pre-training of DTs with online fine-tuning for multi-agent RL tasks.

In [40], transfer learning for the DT architecture is addressed and achieved by first pre-training the DT model on large datasets from other domains, such as Wikipedia-trained Transformers from language domains, and then fine-tuning the model in the offline RL setting. This transfer learning approach outperforms standard DT and indicates the existence of some underlying universal structure across sequence modeling problems, which enables transfer learning across different domains.

4 Methodology

In the original paper [6], Decision Transformer is evaluated on Atari tasks, as well as control tasks from the D4RL datasets, where the latter include the Mujoco environments Hopper, Halfcheetah, Walker2D, and Reacher2D. In a robotics context, especially the results on D4RL are interesting, and they promise that DT is performing comparably to existing offline RL methods in fine-grained control tasks while even outperforming some of them.

In this thesis, we conduct further experiments to thoroughly evaluate DT and understand its properties as well as its drawbacks. The contributions are two-fold.

1. **Focus on stabilization tasks.** While DT has been evaluated on several D4RL tasks, these do not include control tasks where the target state is an unstable equilibrium. However, to evaluate DT and its potential use for robotic systems these kinds of tasks are particularly important. Stabilization tasks with unstable equilibrium points are generally of demanding nature since small deviations in the actions can lead to the policy failing to solve the task, which makes fine-grained control necessary. Therefore, we focus on several stabilization tasks with varying difficulty. The environments are presented in Section 4.1.
2. **Evaluation on a real robot platform.** While all of the tasks from the D4RL benchmark are evaluated in simulated environments, the performance of DT on real-world systems is unexplored. Sim-to-Real [11], i.e., the gap between simulated environments and the real environments which occurs due to modeling errors, is a particularly important challenge in Robot Learning. Also, Transformers are generally computationally demanding, since they scale quadratically with the context length. This computational complexity makes it important to thoroughly evaluate the real-time capabilities of DT, especially in comparison with existing BC methods. Therefore, we evaluate the performance of DT on the Furuta pendulum real-world setup (Furuta Pendulum RR).

4.1 Environments

We describe the environments for our experiments, which are visualized in Figure 4.1.

4.1.1 OpenAI Gym Mountain Car

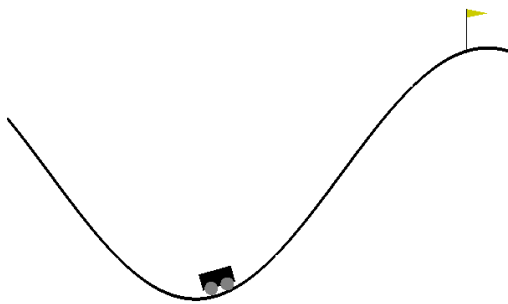
Mountain car, as depicted in Figure 4.1a, is a classical RL environment [41] in which an underpowered car tries to move up a hill. The car starts in the hill valley and can not climb the hill directly using its limited power, so it needs drive back and forth to accumulate enough kinetic energy to move up the hill. This movement resembles the movement of a pendulum which is swung up. In this regard, mountain car can be seen as a simple environment which we use to show the general capabilities of a policy to learn how to accumulate kinetic energy and perform a swing-up movement.

Once the target position on the hilltop is reached the episode terminates and is considered successful. The target pose can therefore be seen as a stable equilibrium, and a policy does not have to stabilize the car in this pose. For our experiments, we consider the continuous OpenAI mountain car environment, in which the policy receives a small negative reward for every time step in which the car has not reached its target position on the hilltop, and receives a large positive reward for reaching the hilltop. The observations are two-dimensional (the car’s position and velocity), while actions are one-dimensional.

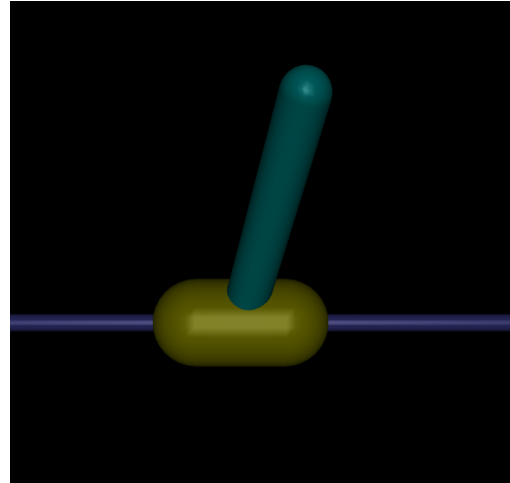
4.1.2 Mujoco Inverted Pendulum

In Mujoco [42] inverted pendulum, as visualized in Figure 4.1b, the goal of the agent is to stabilize an inverted pendulum such that it stays in an upright pose and does not fall down for 1000 time steps. However, this target state is unstable, since the pendulum’s center of mass is above its pivot point. A stabilizing policy needs to perform additional actions after reaching the target state, such that the pendulum is stabilized.

For our experiments, we enlarge the range of initial angles such that for the initial angle holds $\theta \in [-\pi/15, +\pi/15]$, where θ is the angle between the upright target pose and the actual pendulum’s pose. The wider range of initial states makes stabilization generally harder compared to the original environment. The initial angular velocity is always set to zero.



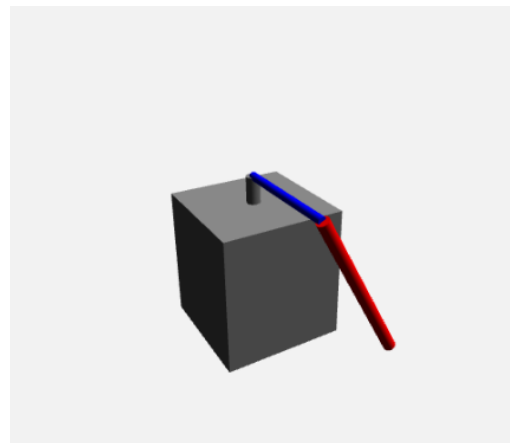
(a) OpenAI Mountain Car Environment.



(b) Mujoco Inverted Pendulum Environment.



(c) OpenAI Pendulum Environment.



(d) Furuta Pendulum Environment.

Figure 4.1: Visualization of the different simulation environments from the experiments.

In the environment, actuation is done by a motor which allows the pendulum to perform a translational movement along a rail. The motor applies one-dimensional actions, the observation space is four-dimensional. The episode terminates once the pendulum is not stabilized anymore, i.e., it falls down such that it reaches a critical angle. The episode return corresponds to the number of time steps in which the pendulum stays out of this critical range, and the maximum episode return is therefore 1000.

4.1.3 OpenAI Pendulum

The OpenAI pendulum environment is depicted in Figure 4.1c. Here, the goal is again to stabilize the pendulum in an upright unstable equilibrium. However, there are two main differences to the Mujoco inverted pendulum environment. First, the pendulum's initial state is not necessarily in an upright pose. Rather, the pendulum starts in an initial angle $\theta \in [0, +2\pi]$, where $\theta = \pi$ corresponds to the pendulum hanging down vertically, and both $\theta = 0$ and $\theta = 2\pi$ describe the upright target pose of the pendulum. To obtain high returns, a policy is required to first swing up the pendulum and then stabilize it in the upright target position. Also, once the pendulum reaches an upright position, it usually has high angular velocities due to the previous swing-up movement which makes stabilization harder. Second, the motor no longer moves the pendulum along a rail in a translational movement, but directly controls the pendulum's angle by applying forces at the pendulum's pivot point. The actions are again one-dimensional, while observations are three-dimensional.

In the environment, high rewards are obtained for stabilizing the pendulum in the upright target pose. It must be noted that the episode return is highly dependent on the initial pose the pendulum starts in. When the initial state is already an upright pose, policies often only need to stabilize the pendulum there or swing around the pendulum once which yields a high episode return. Meanwhile, in situations where the pendulum starts hanging down a policy must put more effort into swinging up the pendulum which also takes more time steps and limits the possible return for this episode. This dependency on the initial state generally leads to a higher variance in the episode returns. In general, episodes with a return larger than -400 can be described as successful, i.e., the pendulum is mostly stabilized in the target pose in these episodes.

4.1.4 Furuta Pendulum

Finally, we consider the Quanser Furuta pendulum, as visualized in Figure 4.1d, where the task is again to stabilize a pendulum. However, the system consists of two parts: a rotary arm which is controlled by the actuating motor, and a pendulum which is placed vertically on the rotated arm. The pendulum has a four-dimensional state space because there are two angles to consider: the angle θ of the rotated arm relative to the motor’s base coordinate system, and the angle α between the pendulum’s upright target pose and its actual pose, as well as their velocities. An angle of $\alpha = \pi$ corresponds to the pendulum hanging down vertically whereas $\alpha = 0$ and $\alpha = 2\pi$ describe the same equilibrium point, i.e., the upright target pose of the pendulum. The observation space is six-dimensional since angles are represented by their cosine and sine values instead of their absolute values. Actions are again one-dimensional.

To obtain a high episode return the pendulum must be stabilized in an upright position such that α is close to the target position. In addition, the reward is to a smaller degree dependent on θ as well as both angular velocities where both θ and the velocities should become zero to obtain the maximum reward.

In the Furuta pendulum environment we perform experiments on two tasks, *stabilization* and *swing-up*.

Furuta Pendulum Stabilization

In the stabilization task, both angles are initially deflected such that they differ from the target angles by not more than $\pi/12$ and not less than $\pi/24$. The goal is to stabilize the pendulum in the upright position, such that it does not fall down. Once the pendulum falls down, the environment theoretically allows the policy to still obtain a high return by swinging the pendulum up again, such as in the swing-up task. However, the training data for the stabilization task does not contain any swing-up movements which makes it practically impossible for an offline RL policy to learn the swing-up movement. Therefore, in our experiments a policy can only obtain high episode return through stabilizing the pendulum without letting it fall down. For the stabilization task, returns close to the environment’s maximum obtainable return of 6.0 are possible.

Furuta Pendulum Swing-Up

In the swing-up task, the pendulum initially starts hanging down. To obtain high returns a policy needs to swing up the pendulum and stabilize it in an upright pose. For the swing-up task the achievable maximum return is significantly lower than the maximum return of 6.0 since swinging up the pendulum takes many time steps that yield lower rewards than when stabilized in the target pose.

4.2 Datasets

As pointed out in [15], datasets are crucial for the performance of offline RL algorithms, which is mainly due to the fact that in offline RL the agent has no opportunity of interacting with its environment to perform exploration and exploitation. Offline RL algorithms are therefore limited to learning from the experience which is contained in the dataset. Similar to the D4RL benchmark, we consider two types of datasets for our experiments: *expert* datasets which contain high return episodes, and *replay* datasets which are obtained during the training process of a Proximal Policy Optimization (PPO) policy [43] and therefore contain suboptimal data (trajectories which are obtained during the first training epochs of the PPO policy) as well as expert data (trajectories which are obtained in late training epochs, after the performance of the PPO policy has reached expert level).

4.2.1 Dataset Characteristics and Metrics

According to [15] there are two main characteristics which determine the quality of offline RL datasets. These are the average episode return in the dataset and the coverage of the state-action space. Two simple measures are proposed to evaluate offline RL datasets.

The **Relative Trajectory Quality (TQ)** compares the average returns of the dataset trajectories to the maximal possible return. When the dataset policy performs better than a random policy, which is the case for all datasets we consider, TQ can be defined as

$$g_{\mathcal{D},\text{norm}} = \frac{g_{\mathcal{D}} - g_{\text{random}}}{g_{\text{online}} - g_{\text{random}}}$$

where g_{random} is the mean return of a random policy in the environment, g_{online} is the maximum return in the dataset, and $g_{\mathcal{D}}$ is the mean return in the dataset. The normalization is necessary to compare datasets across different environments.

Intuitive explanation: TQ measures the quality of the dataset trajectories by their return. A dataset with a high TQ can be described as an *expert* dataset, whereas datasets with low TQ contain bad quality demonstrations. We use TQ as a metric for our datasets and present their TQ values in Table 4.1.

Also, the authors present the **Relative State-Action Coverage (SACo)** as a measure of the coverage of the state-action space where a high SACo value corresponds to a high number of unique state-action pairs in the dataset. The state-action space is assumed to be discrete, because in continuous spaces in principle all state-action pairs are unique. We find that for our purpose SACo is not an appropriate measure, since for the continuous spaces of our environments it would require to discretize the state-action space. We include no empirical measure of the state-action space coverage of the datasets, but rather state that the coverage of the expert datasets is in general smaller compared to the coverage of the replay datasets since during exploration the PPO policy generates many suboptimal state-action pairs, whereas the expert behavior policies focus on a smaller subset of actions that lead to high rewards.

4.2.2 Dataset Overview

An overview of the used datasets can be seen in Table 4.1. For every environment from

Name	Environment	n_{traj}	Behavior Policy	Task	TQ
mountain-car-expert	OpenAI Mountain Car	200	PPO	Drive up	0.99
mountain-car-replay	OpenAI Mountain Car	80	PPO	Drive up	0.50
mujoco-inverted-pendulum-expert	Mujoco Inverted Pendulum	500	PPO	Stabilization	1.00
openai-pendulum-expert	OpenAI Pendulum	250	PPO	Swing up	0.83
openai-pendulum-replay	OpenAI Pendulum	100	PPO	Swing up	0.32
furuta-pendulum-stabilize-expert	Furuta Pendulum	500	PPO	Stabilization	0.99
furuta-pendulum-swing-up-expert	Furuta Pendulum	500	PPO	Swing up	0.49
furuta-pendulum-swing-up-replay	Furuta Pendulum	515	PPO	Swing up	0.29

Table 4.1: Overview of the used training datasets for the stabilization experiments. n_{traj} denotes the number of trajectories in the dataset.

Section 4.1 we provide an expert dataset, which contains trajectories from a PPO policy which we consider to solve the respective task well. For OpenAI mountain car, OpenAI pendulum, and the Furuta pendulum swing-up task, we also provide *replay* datasets. Comparing the TQ values we see that in general the expert datasets have a higher TQ value

than the replay datasets. We also find that the expert demonstrations in the stabilization tasks, as well as in OpenAI mountain car yield very high TQ values, while the expert dataset in the Furuta pendulum swing-up task only has a TQ value of 0.49, which is due to the generally higher difficulty of the task itself and the fact that even expert policies do not achieve the maximum possible return for the task.

4.3 Model Architectures

In our experiments we compare three architectures against each other: Decision Transformer (DT), Decision LSTM (DLSTM) and Behavioral Cloning (BC).

4.3.1 Decision Transformer Architecture

To evaluate DT, we use the original architecture as presented in Section 2.4. We only extend the DT architecture by an additional *scaling layer* which scales all generated actions by a constant. The scaling layer is necessary because in the original architecture the last layer is a tanh activation which restricts the range of actions to $a \in [-1, 1]$. This restriction is fairly appropriate for environments in which actions are normalized to be in this range, however, it is problematic for environments where the action range is larger, such as it is the case for our environments. We choose the scaling constant of the scaling layer to be the absolute maximum action which is possible in the respective environment. The scaling layer can also be used to restrict the actions to an arbitrary range in safety-critical environments.

4.3.2 Decision LSTM Architecture

Framing RL as a sequence modeling problem, as proposed in [6, 7], in principle allows to use other sequence modeling architectures than Transformers. We propose DLSTM, a novel architecture that replaces the GPT-2 model inside DT by a vanilla PyTorch [44] LSTM. For DLSTM, the following additional architectural adjustments are made in comparison to the DT model from Section 4.3.1.

- The LSTM model’s hidden state and cell state are both initialized with zero vectors.

-
- The attention mask is removed because the LSTM in our model does not perform attention.
 - Positional embeddings are removed since LSTMs, unlike Transformers, process their inputs sequentially which makes further positional encodings unnecessary.

In the following, we will at some points reference DT and DLSTM as *decision architectures* to highlight their similarities and the fact that both are based on the idea of framing RL as a sequence modeling problem.

4.3.3 Behavioral Cloning Architecture

We use a simple BC implementation as a comparison for the decision architectures. We build our BC implementation on top of the BC implementation from the DT codebase. This BC model is a Multi-Layer Perceptron (MLP) which takes a set of states as input, and outputs a set of actions. Training is done by minimizing the squared loss between target actions (as provided in the dataset) and the predicted actions from the BC model. Again, we introduce a final scaling layer, as described for the DT architecture in section 4.3.1. Also, we replace the ReLU activations inside the MLP by tanh activations since we find this to improve the model's performance in evaluation.

It must be noted that the considered BC model is only one implementation of a family of Behavioral Cloning and imitation learning algorithms [45, 46, 47], and a fairly simple one. We use this BC implementation as a baseline for comparisons. However, we emphasize that the performance of this architecture does not serve as a general measure to evaluate whether BC or imitation learning approaches are able to solve the respective tasks, but rather indicates that the respective tasks pose challenges for conventional BC models which require more sophisticated approaches.

4.4 Training Process

In [48], it is pointed out that for deep RL methods the reproducibility of experimental results poses an important challenge. There are multiple factors of non-determinism in deep RL, such as network architectures, random seeds, and environment randomness, which make it difficult to compare experimental results. It is proposed that when evaluating a deep RL algorithm, on the one hand strong hyperparameter searches are necessary, and

that on the other hand experimental runs have to be reproduced multiple times to obtain measures of confidence in the results. Following this proposal, we let every training run on five different random seeds, and we include the results from all random seeds for the model evaluation. We plot the experimental results using confidence bounds, i.e., the standard deviation in the episode returns, when evaluating the model performance.

4.5 Hyperparameters

Also, as proposed in [48], we put emphasis on finding an appropriate set of hyperparameters for our models. In this section, we name the most important hyperparameters and sources of non-determinism for model training in the evaluated models, and provide an overview of the used hyperparameters for each environment to allow for reproducible and comparable results. We present an overview of the used hyperparameters for the experiments in Table 4.2.

- *Context length K* . The context length defines the length of the sequence of previously encountered states, actions and RTG values which is passed as an input to the models.
- *Number of hidden layers*. MLPs are built of a number of layers of neurons. The number of hidden layers determines the number of such layers inside the architectures.
- *Hidden layer size*. The hidden layer size determines the number of neurons inside a hidden layer.
- *Batch size*. For one training epoch, the models sample mini-batches of length K from the dataset. The batch size determines the number of sequences that is sampled in one training epoch.
- *Number of training steps per training epoch*. This hyperparameter determines the number of forward and backward passes that the network performs in one training epoch.
- *Input normalization*. In the models, states are normalized via

$$s_{norm} = \frac{s - \bar{s}_{data}}{\hat{s}_{data}},$$

where s is a state in the inputs which will be propagated through the network, \bar{s}_{data} is the mean of the states in the dataset, \hat{s}_{data} is the standard deviation of the states

in the dataset, and s_{norm} is the normalized state. Concretely, the last K time steps are passed as input.

- *Dropout.* Dropout is used as a regularization technique to avoid overfitting. The dropout hyperparameter determines the rate at which dropout occurs.
- *Activation function.* Both inside the GPT-2 model and the prediction layers, activation functions are crucial parts that help the networks to learn complex patterns.
- *Learning rate.* The learning rate determines the step size of the parameter updates during backpropagation. Too high learning rates can lead to model divergence, while too low learning rates can yield slow training processes.
- *Number of attention heads.* As outlined in Section 2.3.3, Transformers use multi-headed attention for training stability. The number of attention heads corresponds to the number of heads used for multi-headed attention.

	DT	DLSTM	BC
Context length K	20	20	20
Number of hidden layers	3	3	3
Hidden layer size	128	128	256
Batch size	64	64	128
Number of training steps per training epoch	3000	3000	3000
Input normalization	yes	yes	yes
Dropout	0.1	0.1	0
Activation function	tanh	tanh	tanh
Learning rate	3×10^{-5}	3×10^{-5}	3×10^{-5}
Number of attention heads	1	-	-

Table 4.2: Overview of the used hyperparameters for the different evaluated architectures.

4.6 Evaluation Process and Metrics

Evaluation in offline RL can generally be performed in two ways: either in a pure offline setting, or in an online setting. While the offline evaluation is generally harder and less

intuitive, the online evaluation violates the pure offline assumptions of offline RL methods and is not possible in many environments.

For our experiments, we only consider online evaluations since we have access to simulated environments, as well as to Furuta Pendulum RR. We consider the mean and the standard deviation of the episode returns that are obtained during evaluation as metrics to measure how successful the models are at solving the respective tasks. We do not perform offline evaluation, but refer to [14] for some directions towards offline evaluation.

4.6.1 Online Evaluation in Simulation

We perform evaluations in simulation for OpenAI mountain car, OpenAI pendulum, Mujoco inverted pendulum, and the Furuta pendulum environments. For all experiments we evaluate each model training epoch on 30 evaluation episodes for each of the five random seed which totals to $30 \cdot 5 = 150$ evaluation episodes per model and training epoch.

4.6.2 Online Evaluation in Furuta Pendulum RR

We also evaluate the learned policies in Furuta Pendulum RR. We use the policies which have been trained on simulation data and apply them to Furuta Pendulum RR. The swing-up task is evaluated using two approaches: first, using the learned decision architectures and BC directly, and second, for DLSTM only, using the learned DLSTM to swing up the pendulum and an additional PD controller for stabilization around the equilibrium point. Also, we perform evaluations on the stabilization task by swinging up the pendulum until it is stabilized using a energy-based PD controller, and then letting the learned decision model or BC take over to stabilize the pendulum in the equilibrium for an additional 1500 time steps. Detailed explanations of the evaluation procedure on Furuta Pendulum RR can be found in Section 5.3. In the real setting, we again evaluate on 30 evaluation episodes, but only for the final model of the best random seed of the training runs.

5 Experiments & Results

In this chapter, we describe and analyze the experimental results we obtained for the evaluations of DT and DLSTM in comparison to BC. We present the results from training the architectures on expert and replay datasets as well as the results we obtained for applying the architectures on the Furuta pendulum real-world setup (Furuta Pendulum RR). Finally, we analyze the influence of the specified Returns-to-go value.

In Table 5.1 we present the mean and standard deviation of the episode return in the simulated environments for the different models. When analyzing the experimental results we will often refer to this Table. Additionally, we provide plots for the mean episode return of the different training epochs in the subsections for every environment. For some experiments, we also evaluate the phase plane plots of the evaluation trajectories to explain the behavior of the learned policies in more detail.

Environment	Dataset	$\overline{G_{Data}}$	Mean Evaluation Episode Returns		
			DT	DLSTM	BC
OpenAI Mountain Car	Expert	91.80 ± 1.54	94.39 ± 0.89	94.03 ± 1.22	94.05 ± 1.17
OpenAI Mountain Car	Replay	-379.86 ± 744.23	93.58 ± 0.09	96.50 ± 5.15	95.46 ± 0.86
Mujoco Pendulum Stabilization	Expert	1000.00 ± 0.00	454.72 ± 360.12	985.31 ± 71.96	61.61 ± 170.16
OpenAI Pendulum Swing-up	Expert	-207.53 ± 167.75	-761.44 ± 375.71	-252.86 ± 233.21	-235.78 ± 204.45
OpenAI Pendulum Swing-up	Replay	-837.35 ± 414.12	-1083.78 ± 346.79	-569.89 ± 568.49	-815.41 ± 577.83
Furuta Pendulum Stabilization	Expert	5.95 ± 0.02	0.46 ± 0.03	5.93 ± 0.01	1.82 ± 1.60
Furuta Pendulum Swing-up	Expert	2.93 ± 0.63	0.74 ± 0.24	1.79 ± 1.12	0.87 ± 0.21
Furuta Pendulum Swing-up	Replay	1.56 ± 1.70	0.51 ± 0.25	1.30 ± 1.28	0.89 ± 0.83

Table 5.1: Overview of the experimental results of the final models in the different simulation environments. The table shows the datasets with their mean trajectory returns ($\overline{G_{Data}}$) and the mean and standard deviation of the episode returns from evaluating all final models.

5.1 Expert Dataset Experiments

In this section, we analyze the results from the experiments which were conducted on expert datasets.

5.1.1 OpenAI Mountain Car Expert Results

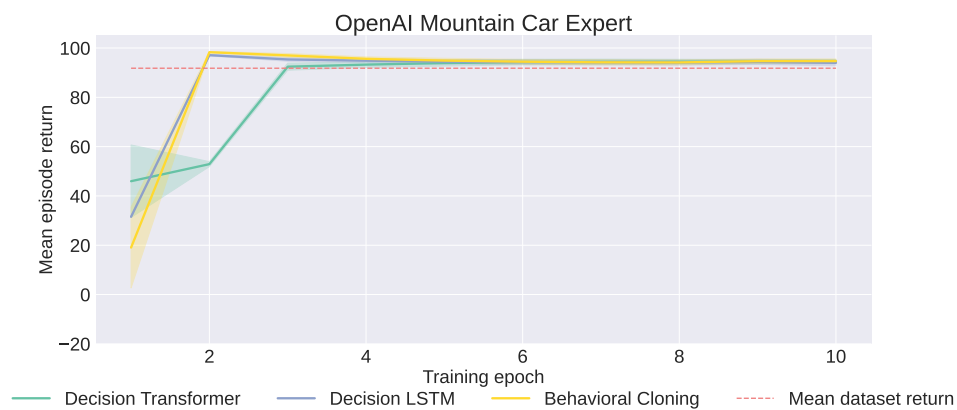


Figure 5.1: Average episode returns obtained in evaluation for the OpenAI mountain car expert dataset experiments.

The results of the models in OpenAI Mountain Car can be seen in Figure 5.1. All three architectures are capable of solving the mountain car task after only a few training iterations. In the OpenAI mountain car setting, the models therefore yield optimal results and all three models are even capable of performing slightly better in average than the behavior policy of the dataset. In general, all three architectures prove to be able to solve the, fairly simple, task of controlling the mountain car such that it reaches its stable target state. The strong performance of the models indicates that all policies should in principle also be capable of learning the necessary movement to swing up a pendulum since moving the mountain car up the hill resembles this swing up movement.

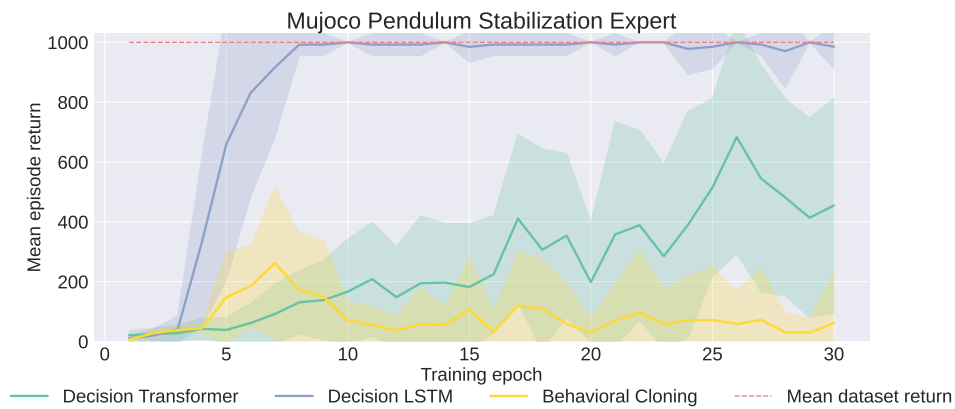


Figure 5.2: Average episode returns obtained in evaluation for the Mujoco inverted pendulum expert dataset experiments.

5.1.2 Mujoco Inverted Pendulum Expert Results

The results from evaluating the learned models in the Mujoco inverted pendulum environment are shown in Figure 5.2. Here, it is obvious that DLSTM outperforms both DT and BC by a large margin. DLSTM shows a reasonable training process in the first epochs, and converges to expert performance after eight iterations. The mean episode return of the final DLSTM model of 985.31 is slightly below the expert dataset mean which shows that there are still evaluation episodes where DLSTM fails to achieve the maximum reward of 1000, and lets the pendulum fall down in a late time step. However, these episodes are very rare for DLSTM.

In the Mujoco inverted pendulum task DT further yields better results than BC while being significantly worse than DLSTM. For DT, we observe that the learned policy improves over the training epochs, and the performance settles at an average episode return of around 500. DT solves the task with an adequate episode return in some episodes, while it fails to do so in other episodes, which is indicated by the high standard deviation in the episode returns. The behavior of the DT policy is highly dependent on the initial state, where for some initial states DT achieves reasonable stabilization and for other initial states it fails to do so.

Meanwhile, it becomes clear that BC fails to solve the Mujoco inverted pendulum task because of the very low mean episode return of 61.61. Furthermore does the learning

curve of BC show no significant improvement over the training epochs.

5.1.3 OpenAI Pendulum Expert Results

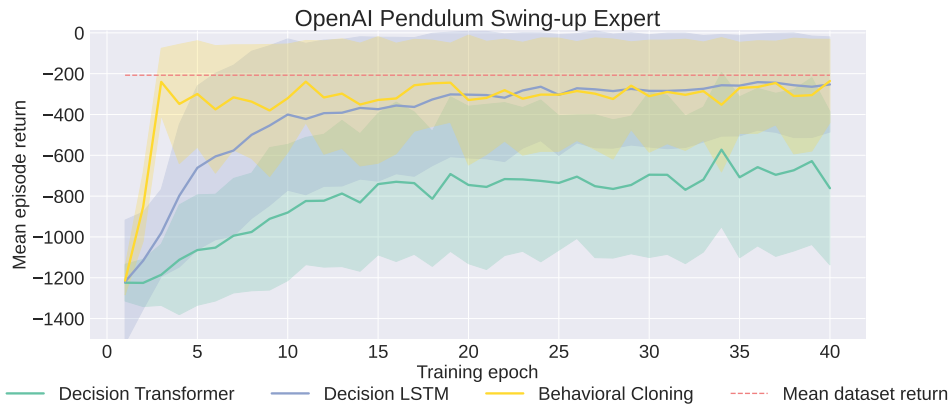


Figure 5.3: Average episode returns obtained in evaluation for the OpenAI pendulum expert dataset experiments.

The evaluation results for the OpenAI pendulum expert dataset task are depicted in Figure 5.3. Considering the learning curves and the final model results we find that both DLSTM and BC prove capable of solving the OpenAI pendulum swing-up task at expert level, while DT does not. The BC policy reaches expert level already after the third training epoch, and its performance stays slightly below the mean dataset return for the remaining training epochs. In comparison, the convergence takes DLSTM slightly more training epochs, and the model converges more smoothly. After roughly ten training epochs the performance of BC and DLSTM is nearly identical. However, there still takes place a learning process inside DLSTM since the model performance improves over the next training iterations. While DLSTM and BC prove to perform on an expert level, DT does not achieve expert level and its performance lacks behind the other two models as well as the mean dataset return by a large margin. After convergence, the performance of DT settles at a mean episode return between -800 and -600, which, as described in Section 4.1.3, is too small to solve the pendulum task adequately.

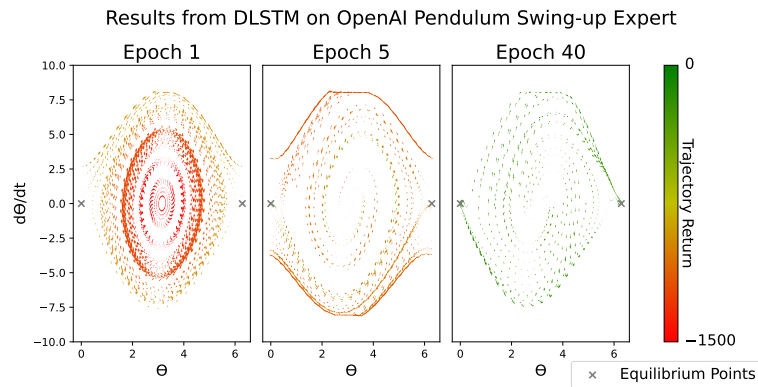
In Section 4.1.3, it is pointed out that the OpenAI pendulum environment is generally prone to high variance when evaluating the episode returns since the achievable episode

return is determined by the initial pose of the pendulum. We observe this high variance in the results shown in Figure 5.3 since the confidence intervals, i.e., the standard deviations of the mean episode returns, are larger than in the previous environments. Therefore, the relatively large confidence intervals can be explained with the variance in the environment itself, and generally do not indicate a high variance inside the models. Still, the high variance for the DT curve shows that there are also episodes in which DT proves capable of solving the task and stabilizing the pendulum in an upright position. Otherwise, its mean episode return would be much smaller.

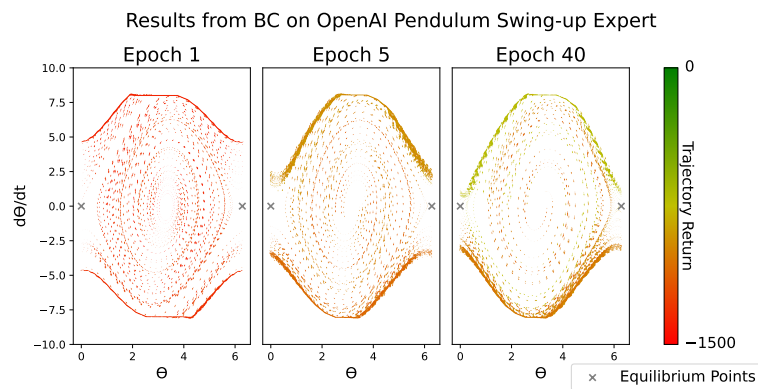
To provide further analysis of the behavior of the learned policies we present phase plane plots which show how the model performance of the evaluated architectures changes over different training epochs in Figure 5.4. We plot the phase plane trajectories for every model for the first, the fifth, and the final training epoch. The episode returns of the respective trajectories are indicated by the color of the lines, where red lines signal low episode returns, and green lines signal high episode returns. Since states in the OpenAI pendulum environment range from 0 rad to 2π rad, the single equilibrium state is represented by two markers where one marker represents the target state in which the pendulum tends slightly to the left, and the other marker represents the target state in which the pendulum tends slightly to the right.

Starting with the behavior of the DLSTM policy, as seen in Figure 5.4a, after the first training epoch most trajectories yield only low returns. We observe two characteristic behaviors in this subplot. The first behavior is represented by the red circles in the center of the phase plane plot. In these very low return episodes, the pendulum swings from side to side, but does not accumulate enough energy to swing up to a higher position such that the upright target position is reached. The policy mostly shows this behavior when it starts hanging down, i.e., close to an initial angle of $\theta = \pi$. The second behavior occurs when the pendulum starts more closely to the target state: the pendulum falls down at first, but then the DLSTM policy swings it up again such that it passes the equilibrium point, and the same movement starts over again. In this case the pendulum sees the target equilibrium state multiple times but the policy does not attempt to stabilize it there. In the phase plot, the second behavior can be detected at the dense top curves. These lines yield higher episode returns than the first characteristic behavior that are indicated by the circles in the plot. After the first training epoch of the DLSTM model, the policy does not manage to stabilize the pendulum in the target equilibrium state in any episode.

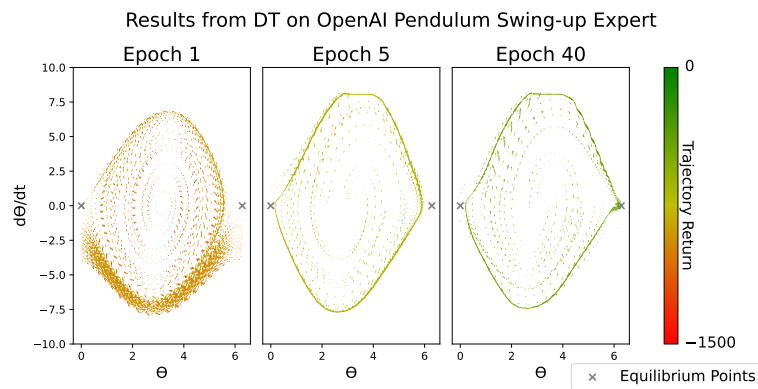
After the fifth training epoch, another characteristic behavior appears in the DLSTM plot: in the episodes which are indicated by green lines, the pendulum manages to swing up the pendulum and stabilize it in the target equilibrium point. In the phase plot, these



(a) DLSTM phase plots on OpenAI pendulum expert.



(b) BC phase plots on OpenAI pendulum expert.



(c) DT phase plots on OpenAI pendulum expert.

Figure 5.4: Phase plane plots of the trajectories over the first, fifth and final training epochs obtained in evaluation for the OpenAI pendulum swing-up expert dataset experiments.

trajectories reach the equilibrium point markers and stay there. When starting hanging down, the DLSTM policy mostly achieves this high-return behavior. However, as indicated by the top red lines, when starting closer to the target position, the DLSTM model fails to stabilize the pendulum, but rather generates the high velocity behavior of swinging up the pendulum multiple times as after the first training epoch.

Finally, after 40 training epochs, the behavior in the right subplot shows that the DLSTM policy manages to stabilize the pendulum in all evaluation episodes, and achieves high returns. This high-return behavior is also achieved in situations where the pendulum is initialized in an upright position.

The behavior of the BC policy, as visualized in Figure 5.4b, is notably different from the behavior of the DLSTM policy. First, the characteristic trajectories in which the DLSTM policy does not manage to swing up the pendulum after the first training epoch, which are visualized as circles in the phase plane plots, do not occur for the BC policy. Rather is BC prone to the high velocity swing up behavior after the first training epoch as indicated by the dense yellow lines at the bottom of the left subplot. BC overcomes this problem after the fifth training epoch and shows high-return behavior. However, the characteristic behavior of swinging up the pendulum and passing the equilibrium pose multiple times remains for the BC policy. We observe that, in most episodes, BC swings up the pendulum and then achieves stabilization in the target pose for some time steps. However, after few time steps the policy lets the pendulum fall down, and afterwards the policy swings up and stabilizes the pendulum again.

For the DT model, the behavior in Figure 5.4c shows that after the first training epoch the model manages to swing up the pendulum but does not stabilize it. Rather we see the high-velocity behavior of swinging up the pendulum multiple times. However, as expected considering the low mean episode return, DT does not overcome the problem of high angular velocities, and even after the 40-th training epoch shows lots of trajectories in which the pendulum swings up and down with high speed. Also, we only rarely observe higher-return trajectories which indicates that the DT policy fails to stabilize the pendulum and achieve high episode returns. Episodes in which the trajectories converge into the equilibrium point markers are rare for the DT architecture.

5.1.4 Furuta Pendulum Stabilization Expert Results

Results from training the models on expert data from the Furuta pendulum stabilization task can be seen in Figure 5.5. Again, DLSTM is the only model to solve the task on an

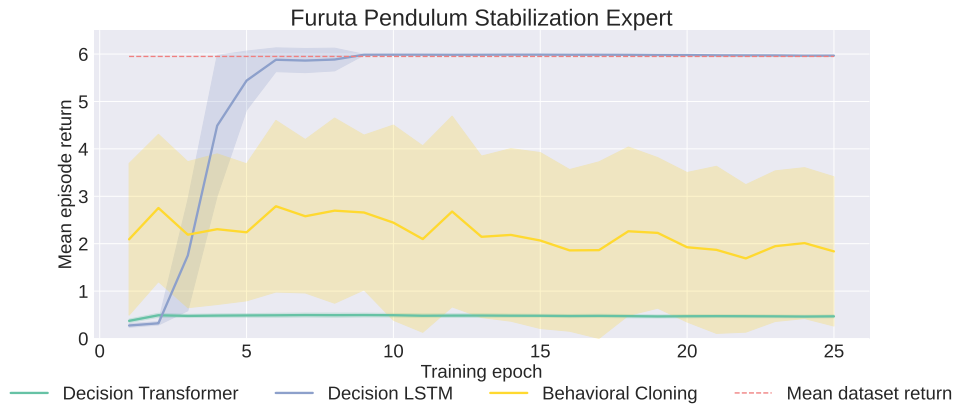
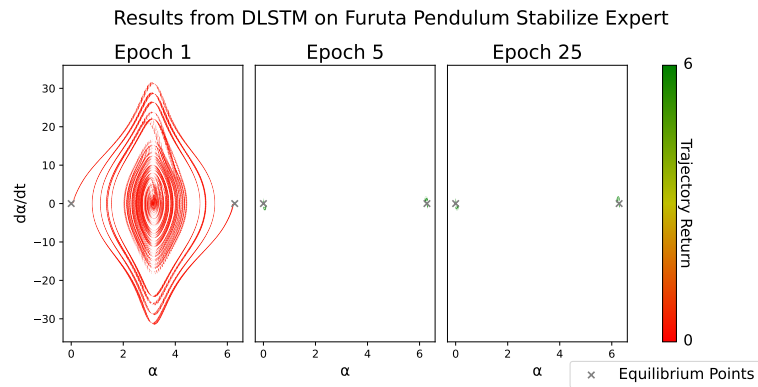


Figure 5.5: Average episode returns obtained in evaluation for the Furuta pendulum swing-up expert dataset experiments.

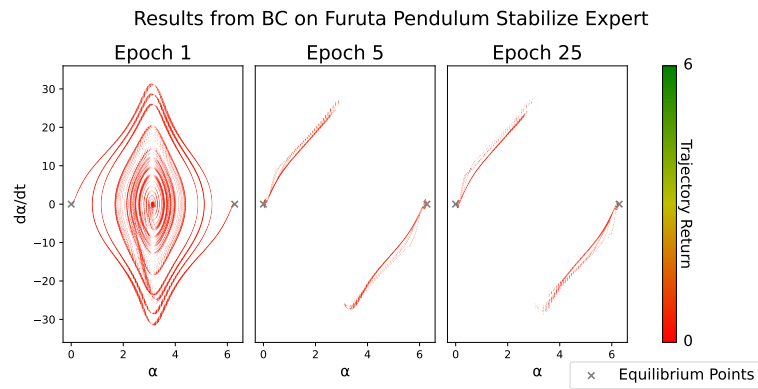
expert level. After roughly eight training epochs the model performance converges to be 5.93 which is almost as high as the near-optimal average dataset return of 5.95. The phase plot for the DLSTM policy is depicted in Figure 5.6a and shows that after five training epochs, the DLSTM policy stabilizes the pendulum in every evaluation episode.

For BC it is notable that the policy does not improve its performance over the training epochs. BC yields an average episode return of roughly 2.0 after the first training epoch but refuses to show any training progress after that. We can explain this learning curve as follows. The mean episode return of around 2.0, combined with the high standard deviation in the episode returns, signifies that BC stabilizes the pendulum successfully under some initial states while it fails to do so under other initial states. This explanation is further supported by the phase plane plots of the performance of the BC policy in Figure 5.6b, where we observe high-return trajectories when starting very close to the target pose, and other low-return trajectories which do not show successful stabilization.

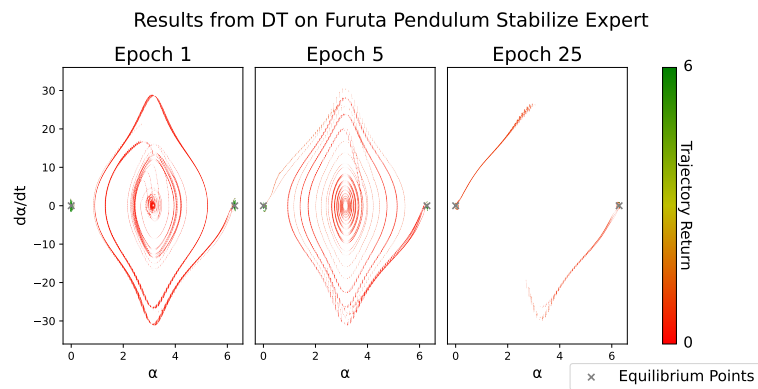
Meanwhile, the DT policy fails to solve the stabilization task. The DT model shows only minimal learning progress in the first few iterations, and converges to a very low mean episode return of roughly 0.5. Also, the evaluation plots of DT show only small standard deviations in the obtained episode returns which indicates that DT never solves the task of stabilizing the pendulum for the whole episode, and that there are no outlier episodes with high returns. Further, the phase plane plots in Figure 5.6c indicate that no successful stabilization episodes occur for DT. The only notable trend between the first and later



(a) DLSTM phase plots on Furuta pendulum stabilization expert.



(b) BC phase plots on Furuta pendulum stabilization expert.



(c) DT phase plots on Furuta pendulum stabilization expert.

Figure 5.6: Phase plane plot of the trajectories over the first, fifth and final training epochs obtained in evaluation for the Furuta pendulum stabilization expert dataset experiments.

training epochs is that the trajectories become shorter which is due to the fact that DT learns a behavior in which it stabilizes the pendulum for a few time steps and then lets the pendulum fall down. The behavior of DT when stabilizing the pendulum for these few time steps leads to an early episode termination because the pendulum reaches its borders.

5.1.5 Furuta Pendulum Swing-Up Expert Results

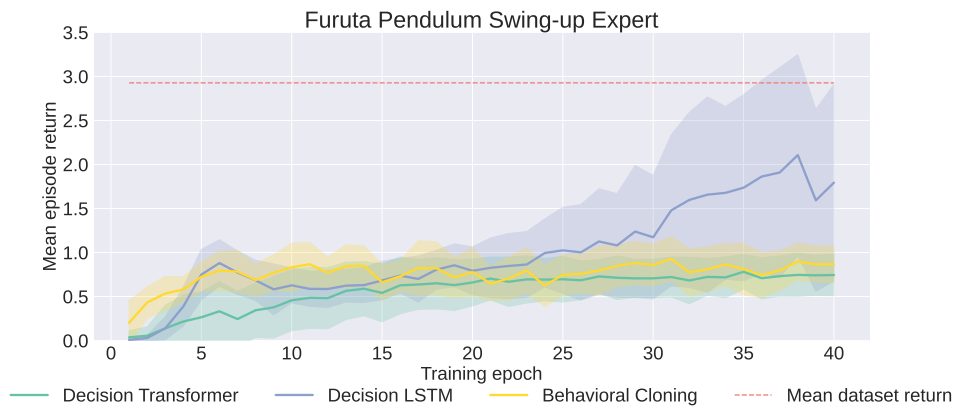


Figure 5.7: Average episode returns obtained in evaluation for the Furuta pendulum swing-up expert dataset experiments.

The mean evaluation returns for the Furuta Pendulum swing-up task are depicted in Figure 5.7. Again, DLSTM outperforms both DT and BC in terms of evaluation episode returns. In late training epochs, DLSTM achieves a reasonable average return of around 2.0 which indicates that it solves the Furuta Pendulum swing-up task in many episodes. Still, the final average episode return is significantly below the mean dataset return of 2.93. The fact that DLSTM fails to stabilize the pendulum in some of the evaluation episodes is clearly indicated by the relatively high standard deviation of 1.12.

Both DT and BC converge to a mean episode reward below 1, and both policies show the behavior of swinging up the pendulum, but not stabilizing it. Also, DT and BC have relatively small standard deviations in the episode returns which signifies that these models do not have large outliers episodes with high returns.

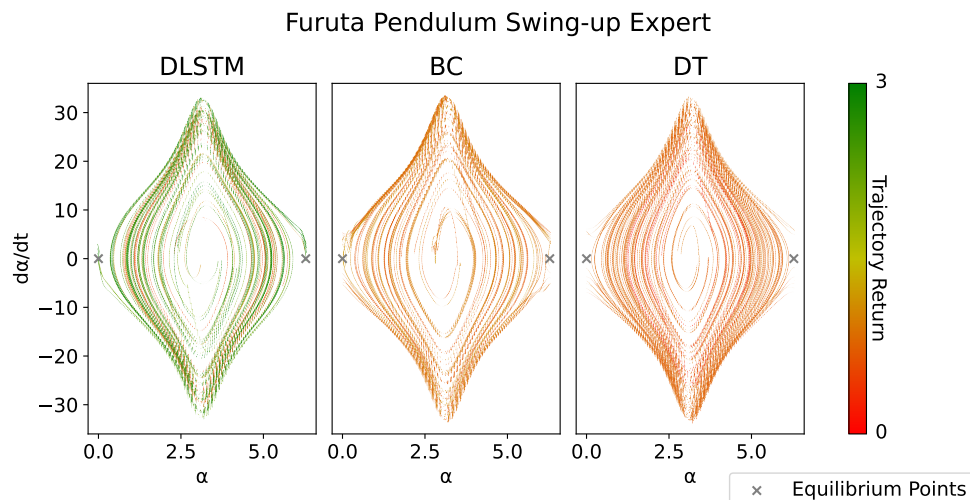


Figure 5.8: Phase plane plot of the trajectories obtained in evaluation for the Furuta pendulum swing-up expert dataset experiments for the final training epochs of the models.

Comparing the evaluation results of the final models in the phase plane, as depicted in Figure 5.8, we see that all models in principle learn to swing up the pendulum to reach the target position. However, DLSTM is the only policy which proves to have successful episodes where it stabilizes the pendulum, as indicated by the green lines which lead into the equilibrium points. The DT and BC policies learn to swing-up the pendulum multiple times which accumulates a higher return than when the pendulum stays in a low position, but they do not achieve stabilization.

Therefore, the behavior of DT and BC is suboptimal and both fail to learn stabilization, which would be crucial to obtain high episode returns. Meanwhile, DLSTM manages to learn stabilization, and achieves higher average returns.

5.2 Replay Dataset Experiments

The following evaluations describe the experimental results from training DT, DLSTM and BC on replay data from the training process of an online RL policy, precisely a Proximal Policy Optimization (PPO) policy [43]. Learning from replay datasets requires policies

to extract higher-return trajectories and prioritizing them for the learning process, while ignoring trajectories of the dataset which yield low returns. In theory, DT and DLSTM are able to use hindsight return information, provided by the sequences' RTG values, to distinguish low reward from high reward actions, while BC only attempts to mimic the state-action pairs from the dataset without taking the obtained rewards of these pairs into account.

5.2.1 OpenAI Mountain Car Replay Results

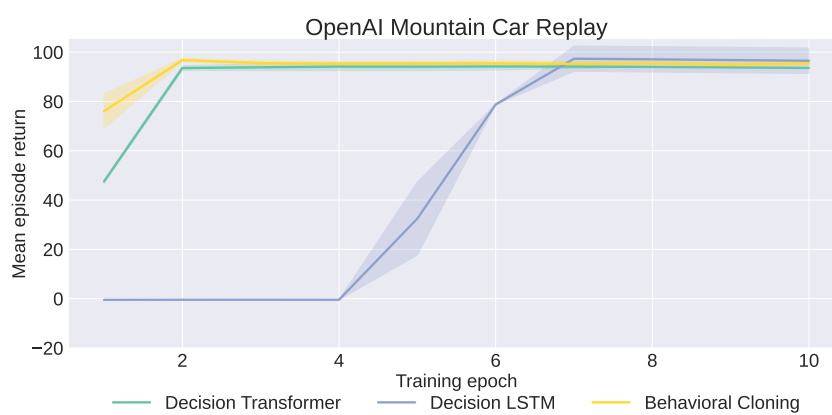


Figure 5.9: Average episode returns obtained in evaluation for the OpenAI mountain car replay dataset experiments. The mean dataset return of -379.86 is not included since it is too low for a proper visualization.

The results from training DT, DLSTM, and BC on replay data from the OpenAI mountain car environment are depicted in Figure 5.9. All of the models converge to expert level performance after few training epochs as it has been the case on the mountain car expert dataset in Section 5.1.1. It is notable that DLSTM lags behind both DT and BC in the first training epochs and only converges to expert level after the seventh training epoch while both other models already converge after two training epochs. However, all of the three models prove to be capable of solving the mountain car task when being trained on replay data from the learning process of an online policy.

5.2.2 OpenAI Pendulum Replay Results

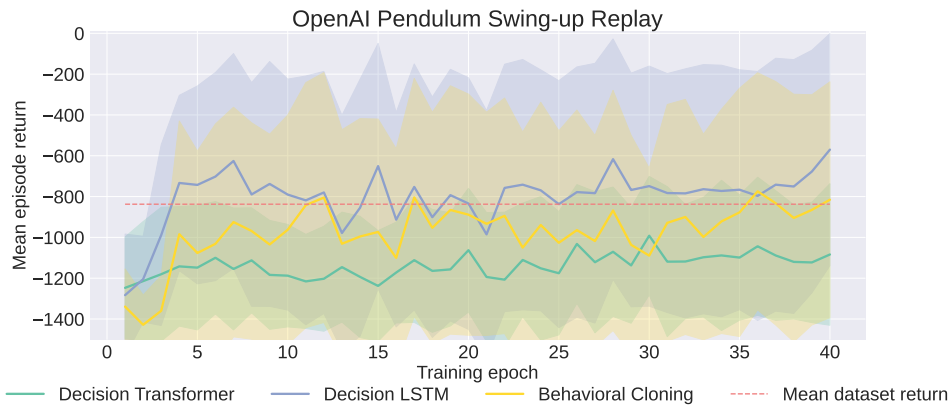


Figure 5.10: Average episode returns obtained in evaluation for the OpenAI pendulum replay dataset experiments.

The evaluation results of the replay experiments on the OpenAI pendulum swing-up task can be seen in Figure 5.10. In the OpenAI pendulum replay setting, the DLSTM policy surpasses the average dataset return and performs best out of the three evaluated models. Still, when compared to the results from training on expert data in Table 5.1, the average episode return of the final model of -596.89 is by a large margin smaller than the average return of the final DLSTM model from the expert dataset experiments, which is -252.86. Also, a very large standard deviation in the episode returns for DLSTM occurs. This high standard deviation can not be explained alone by the high variance in the environment, but it indicates that the model performance differs heavily between different episodes. In some episodes, the DLSTM policy manages to stabilize the pendulum and achieves a high episode return, while in other episodes it fails to do so.

The BC model lacks behind DLSTM throughout almost the whole learning process and does not surpass DLSTM, as it has been the case on the expert dataset. The high standard deviation also occurs for BC which indicates that the evaluations contain both high and low return episodes. Finally, the performance of DT is again worse than the performance of the other two models. Also, the DT policy has the lowest standard deviation out of the models which indicates that high-return outlier episodes do not occur often. DT again proves to be unable to solve the task adequately.

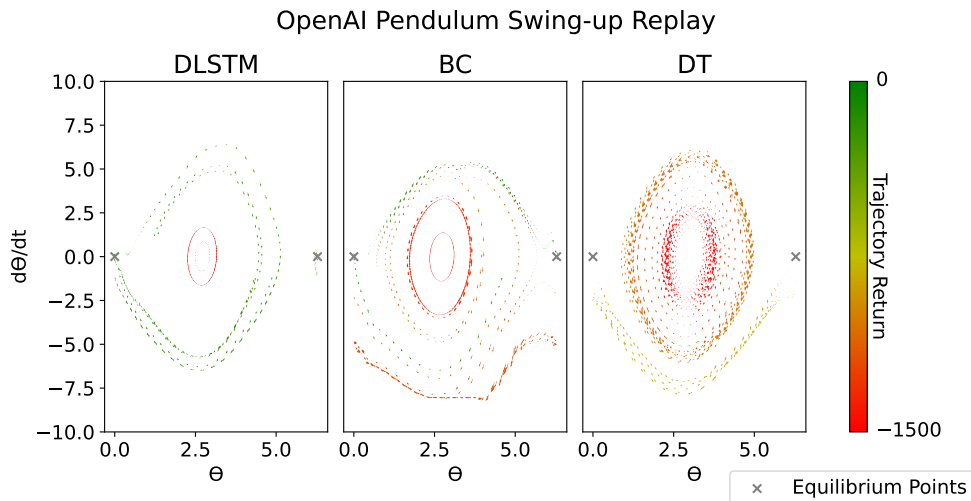


Figure 5.11: Phase plane plot of the trajectories obtained in evaluation for the OpenAI pendulum replay dataset experiments for the final training epochs of the models.

The phase plots of the final learned models in Figure 5.11 confirm that the evaluations of DLSTM and BC contain both high return and low return episodes. The final DLSTM model, as depicted in the left subplot, solves the task with high episode returns when it starts in initial angles that deflect enough from the pendulum hanging down vertically, as can be observed by the green trajectories. However, when the pendulum starts hanging down vertically, i.e., close to an initial angle of $\theta = \pi$, the DLSTM policy does not accumulate enough energy to swing up the pendulum and reach the upright target position, as characterized by the red trajectories in the center of the plot.

For the BC model we observe all three characteristic behaviors of the pendulum as they have been outlined in Section 5.1.3. First, for some episodes there are circular trajectories which show that the pendulum is not swung up high enough. Second, the dense red lines at the bottom indicate the behavior of swinging up the pendulum multiple times. And finally, the green trajectories show episodes where the pendulum reaches its target pose and gets stabilized by the BC model there.

For DT, we observe that in some episodes it encounters the problem of not stabilizing the pendulum, as indicated by the red circles in the center of the plot, but also does not achieve stabilization in episodes where the pendulum accumulates enough kinetic energy

to reach the target position. In these episodes the model swings up multiple times with high velocities, but fails to stabilize the pendulum.

5.2.3 Furuta Pendulum Swing-Up Replay Results

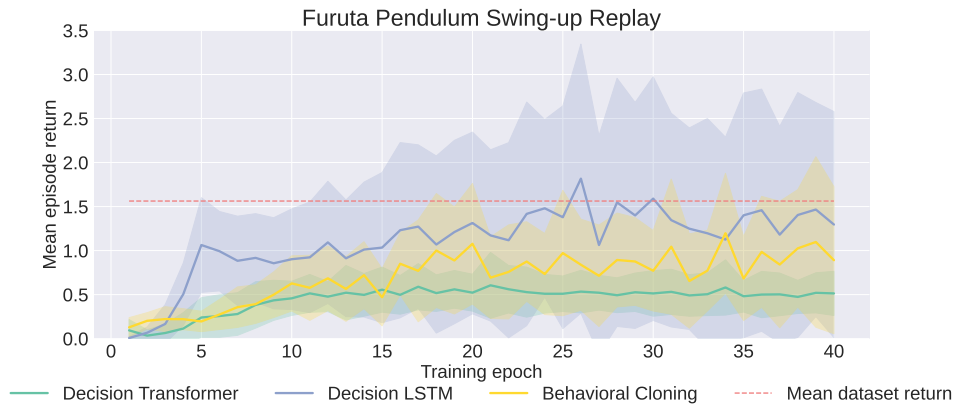


Figure 5.12: Average episode returns obtained in evaluation for the Furuta pendulum swing-up replay dataset experiments.

Figure 5.12 shows the evaluation results for training the evaluated models on replay data from the Furuta pendulum swing-up task. As it is the case for the other replay experiments, DLSTM performs best with an average episode return of 1.30. The final model performance of DLSTM stays slightly under the average dataset return of 1.56. The high standard deviation of 1.28 shows that the model performance differs significantly from episode to episode, and that the model performance is dependent on the initial states of the evaluation episodes. Again, the high standard deviation also indicates that there are many episodes with high returns.

For the BC policy, the model performance after training on replay data is even slightly better than after training on expert data, as shown in Table 5.1. Also, the standard deviation of 0.83 is higher than for the expert dataset which shows that there are outlier episodes with high returns.

The evaluation results for DT show that the performance of the DT model converges early to an average episode return which is significantly lower than for DLSTM and BC. Again, DT does not prove to be able to solve the Furuta pendulum swing-up task.

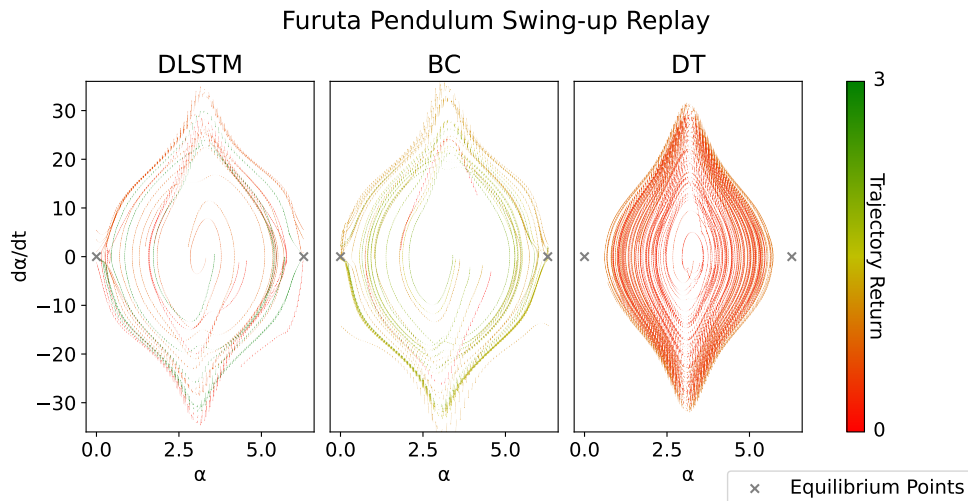


Figure 5.13: Phase plane plot of the trajectories obtained in evaluation for the Furuta pendulum swing-up replay dataset experiments for the final training epochs of the models.

In the phase plane plots of the final models in Figure 5.13 we see that DLSTM is the only policy to achieve successful episodes where the pendulum is swung up and then stabilized, as indicated by the green trajectories in the left subplot. However, there are also many unsuccessful episodes as indicated by the yellow and red trajectories. The BC plot looks similar but lacks green lines which indicate successful stabilization. As can be seen in the right subplot, the DT model fails to even reach the equilibrium, similar to as it is the case for some policies in the OpenAI pendulum environment in Sections 5.1.3 and 5.2.2 where this behavior was indicated by red circular trajectories. We see that, other than DLSTM and BC, DT does not accumulate enough energy to swing up the pendulum after being trained on the replay dataset which is also the cause for the low mean episode return of DT.

5.3 Real Robot Experiments

In the original DT paper, the models are only trained and evaluated in simulated environments. As explained in Chapter 4, we also contribute an evaluation of the decision architectures on Furuta Pendulum RR. We evaluate the learned models from Sections

5.1.4 and 5.1.5 in the real setting. The models are trained in simulation and evaluated on the real Furuta pendulum platform. We note that this approach generally suffers from the *sim-to-real gap* [11]. Since simulations often fail to resemble the real setting adequately the model performances in simulation and reality can differ drastically.

First, we evaluate the performance of the learned models in the Furuta Pendulum RR swing-up task. Then, we consider the stabilization task and evaluate the models there. Finally, we provide additional evaluations regarding the real-time capabilities of the models, namely the response times to obtain actions.

Environment	Dataset	$\overline{G_{Data}}$	Evaluation Returns		
			DT	DLSTM	BC
Furuta Pendulum RR Swing-up	Expert	2.93 ± 0.63	0.38 ± 0.15	1.11 ± 0.52	0.22 ± 0.18
Furuta Pendulum RR Swing-up with PD stabilization	Expert	2.93 ± 0.63	–	2.17 ± 0.60	–
Furuta Pendulum RR Stabilization	Expert	5.95	0.38 ± 0.08	5.98 ± 0.00	5.96 ± 0.02

Table 5.2: Overview of the experimental results of the final models on Furuta Pendulum RR. The table shows the datasets with their mean trajectory returns ($\overline{G_{Data}}$) and the mean and standard deviation of the episode returns from evaluating all final models. We do not report results for the swing-up task with PD stabilization for the DT and BC policy since evaluations are only done for DLSTM.

Table 5.2 summarizes the obtained results for the experiments on Furuta Pendulum RR.

5.3.1 Swing-Up Without Additional Help

First, we present the results from evaluating the models, which are trained on expert simulation data, on the Furuta Pendulum RR swing-up task. As expected after the simulation results, DLSTM performs the best out of the three models with an average episode return of 1.11. However, this return is small compared to the average return of 1.79, which is obtained for DLSTM in simulation. On the real setting, the DLSTM policy suffers from similar problems as DT and BC suffer from in simulation: it manages to swing up the pendulum such that it sees the target position, but fails to stabilize it most of the times.

Again, both DT and BC fail to solve the task appropriately, which is indicated by low average episode returns of 0.38 (DT), and 0.22 (BC) respectively. It is notable that DT performs significantly better than BC. Also, the relatively low standard deviations indicate that there are no outlier episodes with higher returns. Both the DT and the BC policy are

not able of accumulating enough kinematic energy, such that the pendulum is swung up to an upright pose, unlike DLSTM which proves capable of swinging up the pendulum and only lacks the ability to stabilize it in the target pose.

5.3.2 Swing-Up With PD-Aided Stabilization

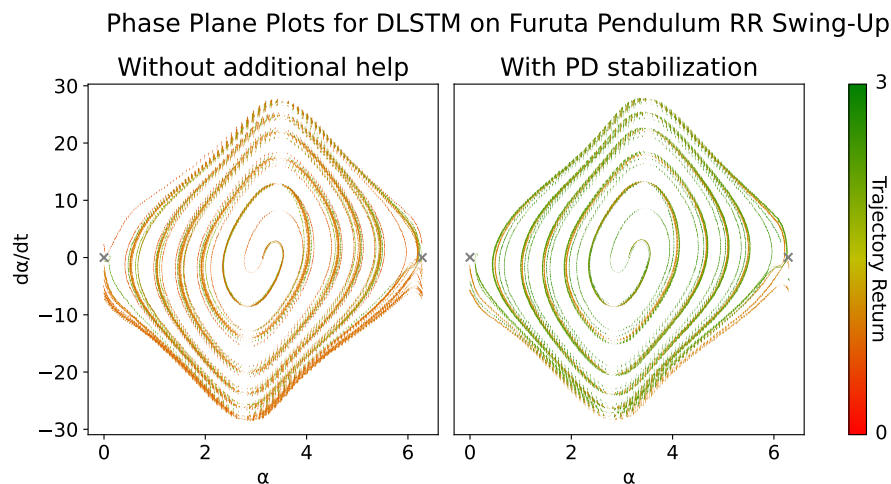


Figure 5.14: Phase plane plots to compare the trajectories obtained during evaluation for the Furuta Pendulum RR swing-up task with and without additional PD-aided stabilization.

To show that with additional help when stabilizing the pendulum the DLSTM model is capable of solving the Furuta Pendulum RR swing-up task, we evaluate the performance of a controller which uses the trained DLSTM model to swing up the pendulum, and lets a Proportional-Derivative controller (PD) controller take over once the pendulum is swung up and must be stabilized. The combined controller switches from the swing-up model to the PD controller once it has encountered a high reward (here: 0.0039) in the previous time step. We only evaluate the performance of DLSTM for this task since both BC and DT have proven incapable of swinging up the pendulum such that an appropriate pose is reached, from where the PD controller could stabilize the pendulum.

With a mean episode return of 2.17, we obtain a significantly better performance of the combined DLSTM-PD controller in comparison to pure DLSTM control. The PD controller

proves capable of dealing with the high velocities after the DLSTM policy has swung up the pendulum. The high average episode return proves that stabilization is what causes the pure DLSTM policy to fail achieving good performance on Furuta Pendulum RR.

In addition, we compare the phase plane plot of the evaluation of the final DLSTM model on the swing-up task without additional help with the phase plane plot of the final DLSTM model with additional PD-controlled stabilization, as depicted in Figure 5.14. We observe that while the plots show that the swing-up movement is achieved by both policies, the DLSTM model without additional help does not stabilize the pendulum in the equilibrium point. The DLSTM policy rather swings up the pendulum multiple times as indicated by the dense yellow lines at the bottom of the left subplot. The right subplot shows that the PD-aided DLSTM model achieves stabilization in the equilibrium in most episodes.

5.3.3 Pure Stabilization

For the stabilization task in Furuta Pendulum RR, evaluation has not been possible directly since resetting the pendulum to an arbitrary initial state from where a policy should stabilize it is hard due to the influence of gravity. To evaluate the stabilization performance of the learned models on the real robot we let an energy-based PD controller swing up the pendulum until a certain reward has been reached for the last time step. This high reward, which we choose to be 0.0039 (the maximum reward for a time step in the environment is 0.004) signifies that the pendulum has been stabilized by the energy-based controller. After this reward has been reached, we switch the controller to the learned model (DLSTM, DT, or BC), which then must stabilize the pendulum for an additional 1500 time steps. The evaluation setting can be seen as in contrast to the situation in Section 5.3.2, where it is the task of the model to swing up the pendulum, while stabilization is afterwards done by a PD controller.

The results from these evaluations can be found in Table 5.2. We find that both DLSTM and BC show near-optimal performance and prove capable of stabilizing the pendulum for the whole episodes with average episode returns of 5.98 and 5.96 respectively. Meanwhile, DT fails to solve the stabilization task, achieving an average episode return of only 0.38. This very low return corresponds to a behavior, in which the pendulum is only stabilized for a few time steps, and then falls down.

5.3.4 Real-Time Capabilities and Response Times

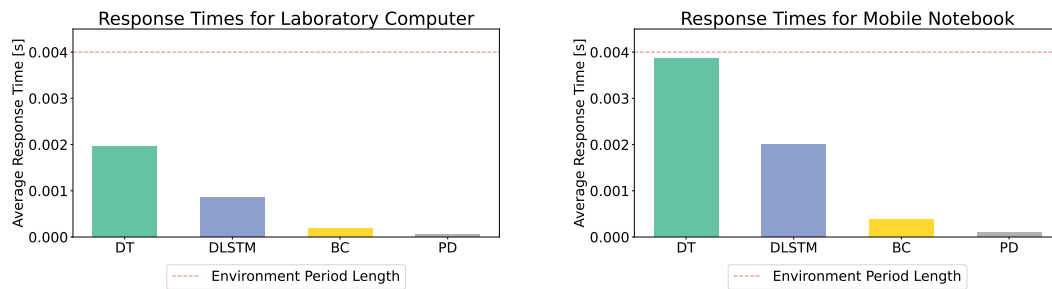
Another important aspect to consider for the models which has been underexplored in the original DT paper is if they are capable of generating actions in real time since the D4RL dataset, which serves as an evaluation framework for DT, only contains simulation data. Using simulations often is not appropriate for evaluating the real-time capabilities of policies because the simulation can stop and wait until the policy has generated an action to continue the simulation, even if this takes very long. For real-world settings, dynamic properties such as gravity make it necessary to consider real-time capabilities of the models. These circumstances make it critical that controllers generate actions in an appropriate time frame, because otherwise the applied action would lack behind the observations which are used to generate the action. Such an offset can lead to a worse policy behavior or even to total failure.

We denote the time which is necessary for the policy to generate an action based on the current observation as the *response time*. The response time depends heavily on the computational power of the system that computes the actions which makes evaluations difficult. However, we can observe certain trends regarding the response times of different architectures when evaluating the response times under different computational settings. For our evaluations we test the response times of DLSTM, DT, BC and a PD controller on two settings with different computational power:

1. Laboratory Computer with Intel(R) Core(TM) i7-9700K CPU @ 3.60 GH (8 cores).
2. Mobile Notebook with Intel(R) Core(TM) i5-7200U CPU @ 2.50 GH (2 cores).

We note that, especially for the DT models which are able to process inputs in parallel a speedup is expected when making use of a dedicated graphics card which we do not test here. We consider two main criteria for our evaluation of the real-time capabilities of DT and DLSTM: First, we observe whether, under the given hardware setup we are using, actions are generated in appropriate time such that the response time aligns with the frequency in which the environment applies actions. For the Furuta Pendulum, this frequency is 250 Hz, i.e., an action is applied every 0.004 s. Then, we compare the models to each other, as well as to the energy-based PD controller which has been used to generate the training dataset itself. This comparison allows to understand whether one model needs significantly more time than another to generate an action and if this can cause problems for real-time usage.

We present the results from evaluating the response times of the different policies in Figure 5.15. For the computationally stronger laboratory computer setup, as seen in Figure 5.15a,



(a) Average response times for the different models using the laboratory computer with 8-core Intel(R) Core(TM) i7-9700K CPU. (b) Average response times for the different models using the mobile notebook with 2-core Intel(R) Core(TM) i5-7200U CPU.

Figure 5.15: Response times of the evaluated architectures in Furuta Pendulum RR under two different computational setups.

all response times are far below the environment period length of 0.004 s. Still, it becomes apparent that DT by far has the highest average response time, more than twice as large as the second highest average response time, which is DLSTM. Both the BC model and the PD controller are much faster at generating actions.

We observe the same trend for the second setup with a weaker CPU in Figure 5.15b. It is important to notice here that the average response time of the DT model is already equal to the period length in the Furuta Pendulum environment, which prohibits a real-time usage of the DT model in this setting. Real-time usage of DT is not possible in this setup because on the one hand there are time steps where the model's response time is already above the period length, and on the other hand other computations need to take place during the period as well, such as communication between the devices and data processing. Under these circumstances, DT is not able to generate an action in an appropriate time, which leaves the DT policy lacking behind the current observations and leads to suboptimal behavior. Again, DLSTM has the second longest response time, which however is still appropriate for real-time usage in this setting. Both BC and the PD controller again have notably smaller response times than DT and DLSTM.

We conclude that for using the decision architectures on real systems the models' response times are an important factor to consider. Crucial for the application of the models is the computational power of the devices which use the models: for our experiments in the robotic lab setting with an appropriately strong CPU, the models are fast enough to perform in real-time settings, while with weaker computational power we find especially

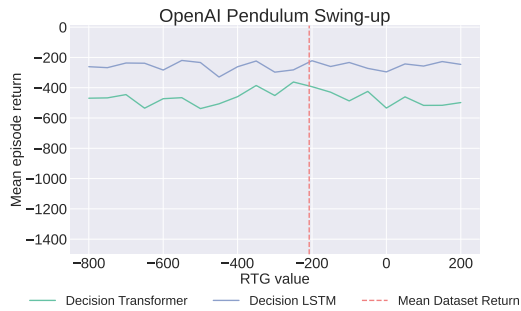
DT, but also DLSTM, to have significantly longer response times to generate actions. It is necessary to consider this weak scalability of the models in comparison to standard BC when using the models on real systems, since for more complex models, or settings with even higher action frequencies, the computational power will be a bottleneck which either allows or prohibits real-time usage of the decision architectures.

5.4 Influence of Returns-To-Go Values

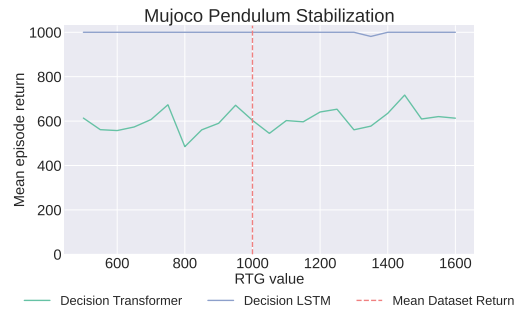
As described in Section 2.4, the DT architecture works with Returns-to-go values to condition the evaluation on the (desired) episode return. Before evaluation, such a desired target return must be specified in advance. In the original DT paper, it is stated that the specified RTG value and the episode return are highly correlated and that using an adequate RTG value is crucial to obtain good model performance. To specify the RTG value, the authors mostly use the mean episode return of an expert policy. However, in some settings, there might be only suboptimal data available, and the optimal episode return might be unknown in advance. For these settings, specifying an appropriate RTG value would therefore pose an important problem.

To investigate the influence of the RTG value on our experiments, we plot the mean episode return of the final training models for each environment over different RTG values. We evaluate the final DT and DLSTM models for the OpenAI pendulum, Mujoco inverted pendulum, and Furuta pendulum tasks in simulation. The BC policy is not evaluated in this regard since it does not make use of any form of hindsight information regarding the returns, and therefore the RTG value is irrelevant for BC. We evaluate each RTG-model combination on 50 evaluation episodes. The results from these experiments can be seen in Figure 5.16.

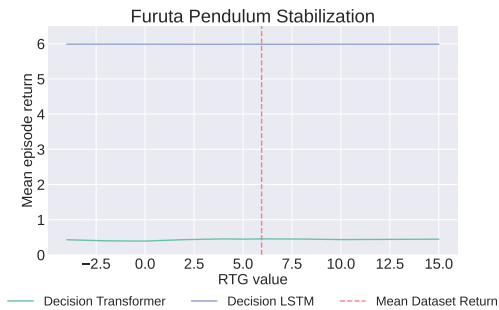
For the OpenAI pendulum swing-up task in Figure 5.16a, there are only minimal differences between the evaluation runs with different RTG values. Similar results are obtained for the Mujoco inverted pendulum stabilization task as can be seen in Figure 5.16b: the DLSTM model is able to achieve optimal performance for all RTG values, and the DT model does not show to be significantly impacted by the RTG value. The same holds for the Furuta pendulum stabilization task, as visualized in Figure 5.16c. Finally, for the Furuta pendulum swing-up task in Figure 5.16d, the results between different RTG values only differ slightly again, and we can not detect any significant trend towards better performance with different RTG values.



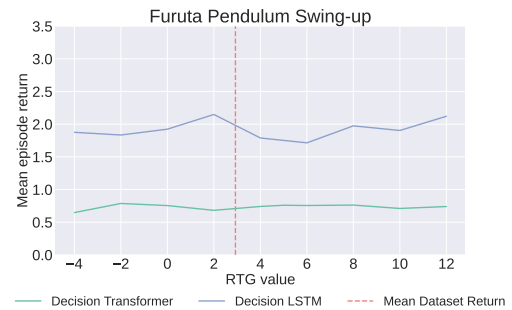
(a) RTG evaluations on OpenAI pendulum swing-up.



(b) RTG evaluations on Mujoco Inverted Pendulum.



(c) RTG evaluations on Furuta Pendulum Stabilization.



(d) RTG evaluations on Furuta Pendulum Swing-up.

Figure 5.16: Average episode return of the final DT and DLSTM models for different Returns-to-go values in the experiment environments.

We conclude that there is no observable trend regarding a better model performance by using certain RTG values for our experiments, and that the minimal fluctuations between the evaluations in the environments are due to the randomness in the environments and the models themselves. These results question the expressiveness of return conditioning inside the decision architectures.

6 Discussion

Considering the simulation experiments from Sections 5.1 and 5.2, we obtain the following general results.

- DLSTM proves capable of solving all stabilization experiments in simulation at expert level when trained on expert data. For the experiments on the replay datasets, we find that the learned DLSTM policy yields worse results compared to when trained on expert data, but is still able to reach appropriate performance to solve the stabilization tasks in all environments.
- The BC model reaches expert level in the OpenAI mountain car environment as well as high average episode returns in the OpenAI pendulum environment when trained on expert data. For the other environments, we find that the used BC model performs significantly worse than DLSTM. Furthermore, in the OpenAI pendulum and the Furuta pendulum environments, we observe that BC is not able to solve the respective stabilization tasks when trained on replay data.
- DT yields the lowest episode returns in our experiments and only solves the mountain car task appropriately, while failing at all tasks that require stabilization around an unstable equilibrium.

Our experimental results clearly indicate that DT struggles heavily with tasks that require stabilization around an unstable equilibrium. Even though we test different datasets, hyperparameters, and environments, DT does not learn reliable stabilization in any of the tasks. Meanwhile, we find that DLSTM significantly outperforms DT in the conducted stabilization experiments. The central conclusion of our experiments is that solving fine-grained stabilization tasks is possible using the approach of framing RL as a sequence modeling problem if a suitable model is used for making predictions, such as DLSTM for our experiments. We show that DLSTM, which fully builds on top of DT, solves the stabilization tasks with high average returns. However, the results are heavily dependent


on the used sequence modeling architecture, and for our stabilization experiments using a LSTM model yields significantly better results than using a Transformer.

The results of the BC model further indicate that these stabilization tasks are non-trivial to solve for standard imitation learning methods. However, as stated in Section 4.3.3, the BC architecture we use is a simple MLP to predict actions from a sequence of states, and we assume that more sophisticated architectures are able to solve the stabilization tasks appropriately.

For the experiments on the Furuta pendulum real-world setup (Furuta Pendulum RR), the sim-to-real gap leads to worse performance of all models in comparison to when evaluated in pure simulation. Especially the fine-grained stabilization after swinging up the pendulum poses an important challenge. Our results from Section 5.3.2 show that with the additional help of a PD controller for stabilizing the pendulum, DLSTM is able to solve the task. For the stabilization task on Furuta Pendulum RR, we find that both the DLSTM and the BC policy are able to stabilize the pendulum in the target state. These results seem confusing at first since we show that DLSTM is both capable of swinging up the pendulum, and stabilizing it when we test both of the tasks independently on Furuta Pendulum RR. However, the DLSTM policy is not able to combine the swing-up and stabilization movements on its own. We assume that this problem is mainly due to high angular velocities which occur after swinging up the pendulum, and the DLSTM policy is not capable of performing the fine-grained stabilization in these states.

Another possible reason for the suboptimal behavior of DLSTM and an important aspect to consider for real-time usage is the response time of the models, as we evaluate in Section 5.3.4. We show that the GPT-2 architecture inside DT yields significantly higher average response times than the other evaluated models, and that these response times can become critical to solve the experimental tasks under certain computational settings. By using DLSTM, we obtain faster response times, but the evaluated BC model still works significantly faster. The relatively long response times of DT are mainly due to the fact that Transformers scale quadratically with the input size, while the computational complexity of other architectures like DLSTMs is linear in the input size. However, we expect that DT can strongly benefit from using graphic cards which enables parallelization at training and test time.

Finally, the results from Section 5.4 question the expressiveness of Returns-to-go values as task-defining conditions inside the decision architectures. The evaluations indicate that the specified RTG value has no significant influence on the performance of DT and DLSTM in our experiments. These results clearly contradict the results which are shown in the original DT pape, where a strong correlation between the RTG value and the evaluation



performance is claimed. Using RTG values requires further evaluations to prove or refute their expressiveness as task-defining inputs.

7 Conclusion & Outlook

Framing RL as a sequence modeling problem proves to be a reasonable approach for achieving stabilization around unstable equilibriums, when using an appropriate sequence model for predictions, which is indicated by the good performance of DLSTM on the stabilization experiments. The sequence modeling framework behind DT and DLSTM is simple and allows to bridge the gap between advances in NLP and RL such that they benefit from each other.

However, our experiments also show potential drawbacks and open questions regarding the DT architecture. DT yields bad performance on the stabilization tasks even though we test various different environments and hyperparameters. Apparently, the performance of the decision architectures is highly dependent on the used sequence modeling architecture. The DT approach can benefit from using LSTMs instead of Transformers even though Transformers are considered to be state-of-the-art architectures in sequence modeling. The potential benefits of using Transformers inside DT, such as effective long-term credit assignment through self-attention, and good results on control tasks, can not be validated by this thesis. Further evaluations are necessary to prove whether DT is capable of learning stabilization around unstable equilibriums, and if DT shows notable advantages in comparison to BC approaches.

It remains an open question whether approaches that frame RL as a sequence modeling problem provide significant advantages over standard BC. Our results indicate that there is no correlation between the RTG values and the model performance in the stabilization experiments. The effectiveness of using RTG values as task-defining inputs that provide hindsight information for the decision architectures is therefore unclear. Moreover, the frameworks and training processes of the decision architectures and BC are very similar. In our experiments, DLSTM outperforms BC on stabilization tasks, but only one specific BC architecture is evaluated. Therefore, we can not state that decision architectures are generally superior to BC in stabilization tasks.

Finally, to make DT and DLSTM applicable for real-world settings, the sim-to-real gap and the response times of the models are crucial aspects to consider. The response times of the decision architectures are significantly longer in comparison to standard BC architectures and yield problems in our real-time experiments, which makes it necessary to further investigate the response times of the models. Purely relying on successful simulation runs where the actual response times of the models are ignored will certainly cause problems under real-world conditions. For settings where DT performs too slow to be used in real time, an LSTM architecture may be preferred over the GPT-2 Transformer model due to the faster run time.

Based on our experimental results, there are multiple directions for further research on Decision Transformer and framing RL as a sequence modeling problem. Especially, more evaluations of DT under conditions such as sparse rewards environments and low data regimes are necessary to find whether the advantages of DT over BC that are claimed in the original DT paper [6] can be confirmed. Moreover, extensions of the DT architecture such as online DT [38] and generalized DT [37] promise to make DT applicable to a wider variety of tasks and settings. Online DT approaches are promising since, as outlined in Section 2.4, the idea behind DT is not limited to the offline RL setting which is considered in this thesis, but can also be useful for online RL.

The successes of DLSTM in the stabilization experiments support the claim that framing RL as a sequence modeling problem can be a powerful and attractive approach for Reinforcement Learning. We expect that further research on the mentioned topics will help to bridge the gap between advances in sequence modeling and RL.

Bibliography

- [1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” 2017.
- [2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” 2019.
- [3] A. Radford and K. Narasimhan, “Improving language understanding by generative pre-training,” 2018.
- [4] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language models are few-shot learners,” in *Advances in Neural Information Processing Systems* (H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, eds.), vol. 33, pp. 1877–1901, Curran Associates, Inc., 2020.
- [5] A. Kolesnikov, A. Dosovitskiy, D. Weissenborn, G. Heigold, J. Uszkoreit, L. Beyer, M. Minderer, M. Dehghani, N. Houlsby, S. Gelly, T. Unterthiner, and X. Zhai, “An image is worth 16x16 words: Transformers for image recognition at scale,” 2021.
- [6] L. Chen, K. Lu, A. Rajeswaran, K. Lee, A. Grover, M. Laskin, P. Abbeel, A. Srinivas, and I. Mordatch, “Decision transformer: Reinforcement learning via sequence modeling,” 2021.
- [7] M. Janner, Q. Li, and S. Levine, “Offline reinforcement learning as one big sequence modeling problem,” 2021.
- [8] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” 2019.

-
-
- [9] J. Fu, A. Kumar, O. Nachum, G. Tucker, and S. Levine, “D4rl: Datasets for deep data-driven reinforcement learning,” 2021.
- [10] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “Openai gym,” *arXiv preprint arXiv:1606.01540*, 2016.
- [11] W. Zhao, J. P. Queralta, and T. Westerlund, “Sim-to-real transfer in deep reinforcement learning for robotics: a survey,” in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 737–744, 2020.
- [12] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.
- [13] R. Bellman, *Dynamic Programming*. Dover Publications, 1957.
- [14] S. Levine, A. Kumar, G. Tucker, and J. Fu, “Offline reinforcement learning: Tutorial, review, and perspectives on open problems,” 2020.
- [15] K. Schweighofer, M. Hofmarcher, M.-C. Dinu, P. Renz, A. Bitto-Nemling, V. P. Patil, and S. Hochreiter, “Understanding the effects of dataset characteristics on offline reinforcement learning,” *ArXiv*, vol. abs/2111.04714, 2021.
- [16] C. Gulcehre, Z. Wang, A. Novikov, T. L. Paine, S. G. Colmenarejo, K. Zolna, R. Agarwal, J. Merel, D. J. Mankowitz, C. Paduraru, G. Dulac-Arnold, J. Z. Li, M. Norouzi, M. W. Hoffman, O. Nachum, G. Tucker, N. M. O. Heess, and N. de Freitas, “Rl unplugged: A suite of benchmarks for offline reinforcement learning,” 2020.
- [17] C. Voloshin, H. M. Le, N. Jiang, and Y. Yue, “Empirical study of off-policy policy evaluation for reinforcement learning,” 2021.
- [18] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, “Glue: A multi-task benchmark and analysis platform for natural language understanding,” 2019.
- [19] B. McCann, N. S. Keskar, C. Xiong, and R. Socher, “The natural language decathlon: Multitask learning as question answering,” 2018.
- [20] S. D. J.-B. G. P.-A. K. D. H. T. Ludovic Denoyer, Alfredo de la Fuente, “Salina: Sequential learning of agents.” <https://github.com/facebookresearch/salina>, 2021.
- [21] S. Bai, J. Z. Kolter, and V. Koltun, “An empirical evaluation of generic convolutional and recurrent networks for sequence modeling,” *ArXiv*, vol. abs/1803.01271, 2018.

-
-
- [22] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” 2014.
- [23] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning internal representations by error propagation,” in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations* (D. E. Rumelhart and J. L. McClelland, eds.), pp. 318–362, Cambridge, MA: MIT Press, 1986.
- [24] M. I. Jordan, “Serial order: a parallel distributed processing approach. technical report, june 1985-march 1986,” 5 1986.
- [25] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, pp. 1735–80, 12 1997.
- [26] A. Sherstinsky, “Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network,” *Physica D: Nonlinear Phenomena*, vol. 404, p. 132306, Mar 2020.
- [27] S. Wang and J. Jiang, “Learning natural language inference with lstm,” 2016.
- [28] D. Hazarika, S. Poria, and E. Cambria, “Recent trends in deep learning based natural language processing [review article],” *IEEE Computational Intelligence Magazine*, vol. 13, pp. 55–75, 08 2018.
- [29] C. Gulcehre, O. Firat, K. Xu, K. Cho, L. Barrault, H.-C. Lin, F. Bougares, H. Schwenk, and Y. Bengio, “On using monolingual corpora in neural machine translation,” 2015.
- [30] P. Ramachandran, P. J. Liu, and Q. V. Le, “Unsupervised pretraining for sequence to sequence learning,” 2018.
- [31] M. Liu, M. Zhu, and W. Zhang, “Goal-conditioned reinforcement learning: Problems and solutions,” 2022.
- [32] A. Kumar, X. B. Peng, and S. Levine, “Reward-conditioned policies,” 2019.
- [33] J. Schmidhuber, “Reinforcement learning upside down: Don’t predict rewards – just map them to actions,” 2020.
- [34] R. B. et al., “On the opportunities and risks of foundation models.”
- [35] A. Jaegle, F. Gimeno, A. Brock, A. Zisserman, O. Vinyals, and J. Carreira, “Perceiver: General perception with iterative attention,” 2021.

-
-
- [36] A. Jaegle, S. Borgeaud, J.-B. Alayrac, C. Doersch, C. Ionescu, D. Ding, S. Kop-pula, D. Zoran, A. Brock, E. Shelhamer, O. Hénaff, M. M. Botvinick, A. Zisserman, O. Vinyals, and J. Carreira, “Perceiver io: A general architecture for structured inputs & outputs,” 2021.
- [37] H. Furuta, Y. Matsuo, and S. S. Gu, “Generalized decision transformer for offline hindsight information matching,” 2022.
- [38] Q. Zheng, A. Zhang, and A. Grover, “Online decision transformer,” 2022.
- [39] L. Meng, M. Wen, Y. Yang, C. Le, X. Li, W. Zhang, Y. Wen, H. Zhang, J. Wang, and B. Xu, “Offline pre-trained multi-agent decision transformer: One big sequence model tackles all smac tasks,” 2021.
- [40] M. Reid, Y. Yamada, and S. S. Gu, “Can wikipedia help offline reinforcement learning?,” 2022.
- [41] A. W. Moore, “Efficient memory-based learning for robot control,” tech. rep., 1990.
- [42] E. Todorov, T. Erez, and Y. Tassa, “Mujoco: A physics engine for model-based control,” in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, IEEE, 2012.
- [43] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017.
- [44] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.
- [45] J. Kober and J. Peters, “Imitation and reinforcement learning,” *Robotics Automation Magazine, IEEE*, vol. 17, pp. 55 – 62, 07 2010.
- [46] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, “Imitation learning: A survey of learning methods,” *ACM Comput. Surv.*, vol. 50, apr 2017.
- [47] F. Torabi, G. Warnell, and P. Stone, “Behavioral cloning from observation,” 2018.
- [48] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, “Deep reinforcement learning that matters,” 2019.