Model-Based Reinforcement Learning for Bipedal Robots

Sai Advaith Maddipatla

4th Year Project Report
Artificial Intelligence and Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

Bipedal locomotion is a challenging control task to learn due to the high dimensionality of the action and state vectors and the partially observable and non-linear nature of the agent's dynamics. In this thesis, we primarily focus on locomotion in bipedal robots like Cassie with the help of model-based reinforcement learning. First, we approximate the dynamics of a bipedal robot with a novel transformer model. Upon effectively training the transformer model, we deployed it in a zeroth-order planner to control a bipedal robot. However, due to the inability of zeroth-order planners to scale with the action's dimensionality, we devised a novel first-order planner and robust model retraining methods which could circumvent these problems and result in stable locomotion. Over the course of this thesis, we hope to motivate this problem and support the success of our methods with experimental results in simulation.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Sai Advaith Maddipatla*)

Acknowledgements

I want to thank my supervisor Professor Subramanian Ramamoorthy for his continual support and advice with the experimentation and writing stage.

I want to thank my family, especially my parents, brother, and cousins, for their unwavering support and love.

Lastly, I want to thank my friends at Edinburgh and Caltech for standing by me through thick and thin.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Bipedal locomotion is essential to the field of robotics. Over the past decades, there have been numerous approaches to developing robust control methods for dynamically complex and under-actuated bipedal systems in different environments. Previously, works have focused on using control-theory and analytical optimization-based methods to control these systems. These approaches include footstep planning using non-convex optimization approaches or converting this non-convex problem into multiple convex-optimization problems [18, 33].

The other broad way of solving this problem is to use learning-based approaches for continuous-time dynamical systems [42]. Learning-based approaches involve the usage of deep reinforcement learning methods for robust control. These approaches have the ability to learn a policy using simulations of the robot or real-world data collected by deploying the robot in a real environment [15].

In this thesis, we will work with learning-based approaches and use model-based reinforcement learning methods to control bipedal robots. Specifically, we will be working with Cassie (Figure 2.1). In model-based reinforcement learning, we first derive a dynamics model of the agent analytically or using deep-learning methods. Once a reliable dynamics model is retrieved, a joint-level controller of the agent uses this model to determine optimal actions which would maximize a reward function. However, in some cases, the model's predictions may lead to sub-optimal outputs from the controller. Hence, it is common practice to regularly retrain the model to avoid sub-optimal behavior. The standard model-based reinforcement learning pipeline has been summarized in Figure 1.1.

We have chosen bipedal robots because of the difficulty of controlling them over an extended period. This difficulty is due to the following reasons:

- Bipedal robots have numerous interconnected joints and degrees of freedom [23, 47].
- A bipedal robot is highly constrained and has hybrid dynamics, which makes it

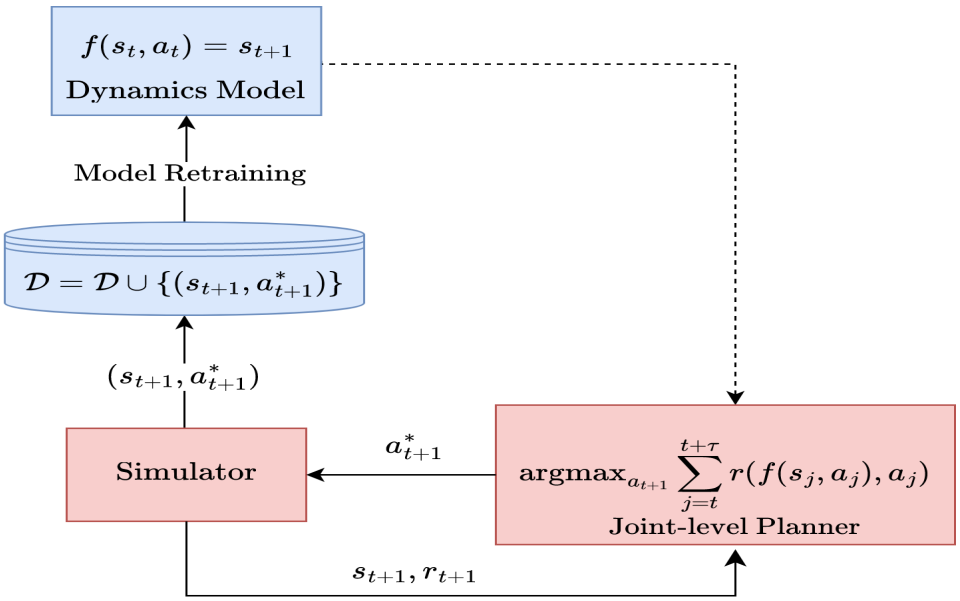

Figure 1.1: A typical model based reinforcement learning pipeline. First, a dynamics model f is trained on off-policy data. The joint-level planner then uses this model to optimize an action a^*_{t+1} by maximizing a reward function. After executing this action in a simulator, we collect on-policy data in dataset $\mathcal{D}$ for retraining f.

harder to learn the robot's dynamics [45].

- Robots have multiple passive joints, i.e., joints that can transmit rotatory motion without an actuator [29]. As a result, joints are indirectly affected by the torque applied to other joints. Hence, we have incomplete state information about our agent, which implies we have to model a partially observable Markov decision process [1, 21] to control our agent instead of a Markov Decision Process.

1.2 Contributions

The main contributions in this thesis include:

1. Approximate the dynamics model of the bipedal robot in an offline setting with the help of a novel transformer using state-action transitions aggregated from Cassie in a simulator.
2. Developed a novel memory and sample-efficient joint-level planner, which leverages the structure of existing joint-level planners and makes use of the learned dynamics model to determine the optimal action sequence.
3. Effectively retrain the deep-learning-based dynamics model using data collected throughout the planning phase.

1.3 Summary of results

In this subsection, we will summarize all the results obtained throughout the thesis. First, we will discuss the results from our model learning pipeline. Then, we will discuss the results from the joint-level controller and the effect of efficient model retraining methods.

During the model learning phase, we used an encoder-decoder-based transformer model to approximate the dynamics of a bipedal robot. We used an input sequence of H state-action transitions to predict the differences between consecutive states. To effectively predict τ steps in the future, i.e., from timestep $H+1$ to $H+1+\tau$, we used a variant of the teacher-forcing training strategy to gradually increase the difficulty of the prediction task. After using these methods on a dataset of state-action transitions, we approximated the dynamics of a bipedal robot quite well.

With a deep learning model which can approximate the dynamics of a bipedal robot, we can now use it in a joint-level controller. We developed a novel first-order planner which can scale well with a large deep-learning model. Furthermore, we observed improved performance over popular zeroth-order planners. Lastly, we were able to further improve upon the performance of our first-order planner by incorporating efficient model retraining strategies.

1.4 Thesis Structure

In Chapter 1, we primarily motivate the problem and the challenges involved and summarize our contributions and the results we obtained over the year. In Chapter 2, we get into greater detail about the background required to understand the essential components of the thesis. This includes the previous model-based and model-free methods applied to different robots and their implications. In Chapter 3, we discuss the methods used to solve the problem. We first discuss how we approximated the dynamics of our agent, the integration of the learned dynamics with our joint-level planning, and the model-retraining approach we used. In Chapter 4, we discussed the experimental setup and briefly introduced the specific robot we are considering. In Chapter 5, we get into the experiments performed and their results in the context of model-learning and joint-level planning. Lastly, in Chapter 6, we will reflect upon the results we received and possible extensions for the work.

Chapter 2

Background

In this chapter, we will go over the relevant background materials required to understand the different components of this thesis. First, we give a description of reinforcement learning, discuss the two main methods used to solve reinforcement learning problems, and briefly summarize past works relevant to this thesis. Second, we go over robot control, and the two main types of joint-level planning. Lastly, we give a brief overview of the deep learning model used for approximating the dynamics of our bipedal robot - the transformer model.

2.1 Reinforcement learning

Reinforcement learning is one of the essential paradigms in the broad field of machine learning along with unsupervised learning and supervised learning.

Formally reinforcement learning involves effectively modeling a Markov Decision Process (MDP). Specifically, an MDP involves a set of possible states $\mathcal{S}$ and a set of possible actions $\mathcal{A}$. At every timestep t, an agent executes an action a_t and transitions from s_t to s_{t+1} per the transition probability distribution of the environment or dynamics of the agent in the environment. Formally this is given by $\mathbb{P}(s_{t+1}|s_t, a_t)$. Consequently, the agent receives a reward $r_t = R(s_t)$ in a particular state which is an objective metric of the agent's performance. In reinforcement learning, our goal is to maximize the total reward aggregated throughout an episode.

There are two broad ways to solve reinforcement learning problems. The first is *Model-free reinforcement learning*, where the agent uses a learned control policy to make optimal decisions in an environment. The other is *Model-based reinforcement learning*, where the agent uses a predictive model of the environment to make optimal decisions.

We will now go over some previous model-free and model-based works which are relevant to our thesis.

Figure 2.1: Cassie Robot at Caltech developed by Agility Robotics in real life [34].

2.1.1 Previous Work

Numerous existing methods focus on applying model-free methods to control Cassie [22, 46, 8, 40]. These methods learn a policy that implicitly leverages the dynamics of an agent to maximize a reward function. In contrast, as a part of model-based methods, we use a reliable dynamics model in our joint-level controller. Often, in a model-free setup, the agent is trained to perform a single gait or selects gait/s from a gait library. After training, the robot can perform those tasks incredibly well.

However, the main advantage of model-based reinforcement learning over model-free reinforcement learning is the sample efficiency and the ability to generalize between multiple gaits once we have a robust dynamics model. As a result, numerous previous works also make use of model-based reinforcement learning [50, 10, 24].

2.1.1.1 Model-Free Approaches

One work [40] specifically focuses on using model-free reinforcement learning for training locomotion policies in bipedal robots. To learn these policies, the reinforcement learning algorithm indexes a gait from an offline gait library which would maximize a particular reward function. This parametric gait library can be indexed using:

- forward velocity (along x direction)
- lateral velocity (along y direction)
- height of the pelvis from ground - to implicitly control movement along the z direction.

After training, the robot was able to perform a multitude of dynamic tasks like walking, trotting, etc., with a specific target velocity. However, like other model-free policies, the training time and sample efficiency are significantly high. In addition, the robot's capabilities are limited by that of a finite gait library.

Another model-free approach of importance to us is [40], which uses a Long Short-Term Memory (LSTM) model in the context of a recurrent Proximal Policy Optimization

(PPO) algorithm [13, 39] for learning an expert reference motion of walking in a straight line at the speed of 1 m/s. In this approach, they can leverage the structure of the LSTM model to sample trajectories instead of individual timesteps, as in vanilla PPO. The reward function for this task enforces perfect walking by comparing the current state of the robot with that of an expert reference trajectory and maximizing the reward based on how closely the agent matches the reference trajectory. The LSTM-based policy can follow an optimal action sequence that corresponds to perfect walking. This method comes at the cost of generalizability in that the policy has to be retrained from scratch to accommodate a change in velocity when walking along a straight line.

2.1.1.2 Model-based Approaches

All issues mentioned above can be resolved using model-based reinforcement learning methods. Previous works have used model-based reinforcement learning methods to control quadrupedal robots in an optimal manner. In this work [50], the authors focused on using a model-based reinforcement learning setup to generate gaits in a quadrupedal robot. Specifically, they use model-based reinforcement learning to generate gaits with the help of the swing phase of the robot, i.e., the rotational velocity of the legs. Using this information, the robot can automatically transition between gaits by leveraging the phase-based gait generator and generate the most efficient gait at different speeds based on the minimization of mechanical energy. The pipeline consists of a high-level gait generator and a low-level model-predictive controller (MPC), which optimizes over energy and speed of the joints. This idea is similar to the approach used in [10], where an MPC optimizes over a reward that relies on energy consumed by the joints and speed of the robot.

While these methods establish the advantages of using model-based reinforcement learning methods, the dynamics of a quadruped are much more stable and tractable than that of a bipedal robot. Additionally, a model-based pipeline is incomplete without a good approximation of the robot's dynamics. To simplify this, one work [24] combines safety-critical control with model-based reinforcement learning by leveraging the fact that Cassie's dynamics can be reduced to a simple linear model when it is controlled by a model-free policy. This planner uses control barrier functions [48] to provide safety guarantees in Cassie via a high-level planner and robust joint-level controller. The essential aspect of this work was having a reliable model-free policy controlling the robot. We hope to exhibit learning behavior without relying on a model-free policy in an online setting.

2.2 Robot Control

A bipedal robot, like any other robot, is a multi-linked control system with multiple joints and motors. The robot can be effectively controlled with a sequence of action vectors. Each action vector $a_t \forall t$ represents the different motor positions. The torque in the robot's joints is generated with the help of a Proportional Derivative (PD) controller.

A PD controller is a controller which uses a feedback control loop. Essentially, given a vector of reference target positions $a_t = a(t)$, the controller attempts to correct the

error $e_t = e(t)$ between motor positions $m_t = m(t)$ generated by the control system and desired motor positions ($a(t)$) by incorporating this error ("feedback") in the input. Hence,

$$e(t) = a(t) - m(t)$$

In a PD controller, this correction is done via a linear combination of the error $e(t)$ and a derivative of the error $\frac{de(t)}{dt}$. Formally,

$$u(t) = K_p e(t) + K_d \frac{de(t)}{dt}$$

Where K_p and K_d are non-negative constants and $u(t)$ acts as an input to our agent.

We will now go over the different ways to determine the optimal action $a(t)$.

2.2.1 Joint-level Planning

Joint-level planning is an essential component of the model-based reinforcement learning pipeline. The purpose of the joint-level planner is to determine the optimal action to be executed, given the current state of the robot. In our case, we will use a model-predictive controller to plan a series of action sequences. A model-predictive controller is a method that makes use of a model to predict the behavior of the agent and determine an optimal action by maximizing a reward function over multiple steps.

In our scenario, the model will be the learned dynamics model. Based on the model, we hope to predict optimal action sequences over a finite horizon in the future. In this thesis, by "n steps in the future," we mean n steps from the latest observed state or executed action. There are two main types of planners:

- **Sampling-Based Planner**: A sampling-based planner is a planner which uses importance sampling to optimize action sequences.
- **Gradient-Based Planner**: A gradient-based planner is a planner which makes use of gradients of the reward to update the action sequences.

We will now get into some important works around both these planners

2.2.1.1 Sampling-based planning

One of the most important Zeroth-order sampling-based planners is the Cross-Entropy Method (CEM) based planner [28]. In this planner, we first sample N action sequences for τ timesteps in the future from a standard normal distribution, i.e., $\mathcal{N}(\mathbf{0}, I)$. More formally, we can say $A = (\mathbf{a}_0^\tau, \mathbf{a}_1^\tau \dots \mathbf{a}_N^\tau)$. In our notation, the superscript represents the length of the action sequence, and the subscript represents indices between the samples. A reward function will evaluate the aggregate reward over τ steps for all N action sequences by rolling them out in a simulator or a dynamics model. Using this, we will select K ($N >> K$) elite action sequences from the N action sequences, i.e., actions with the top K highest rewards. Hence, the set of elite action sequences is $\hat{A} = (\hat{a}_0^\tau, \hat{a}_1^\tau \dots \hat{a}_K^\tau)$. We will now update the action distribution to $\mathcal{N}(\mathbb{E}[\hat{A}], \mathrm{Cov}(\hat{A}))$. In the next iteration, we will again sample N action sequences of length τ from this

updated distribution and repeat the process. After a given number of iterations, i.e., upon convergence, we would choose the action to be executed by the agent by selecting the sampled action with the highest reward. This action would then be executed in a simulator, and the process would be repeated for the subsequent timesteps.

However, zeroth-order planners suffer from slow convergence when the action vector is intricate and has a high dimensionality (curse of dimensionality) [2].

2.2.1.2 Gradient-based planning

In contrast to sampling-based planners, gradient-based planners leverage the gradient of a convex reward function and the sampling-based structure of CEM [3]. Furthermore, unlike CEM, this is a first-order planner.

Like a sampling-based planner, we use the learned model to plan our action sequence. As a first step, similar to CEM, we first sample N action sequences (over τ steps in the future) from $\mathcal{N}(\mathbf{0}, I)$. More formally, we can say the actions and their corresponding states are $A = (\mathbf{a}_0^\tau, \mathbf{a}_1^\tau \dots \mathbf{a}_N^\tau), S = (\mathbf{s}_0^\tau, \mathbf{s}_1^\tau, \dots \mathbf{s}_N^\tau)$. The subscript and superscript carry the same meaning. Using a reward function, we can calculate the aggregate reward for each action sequence over τ timesteps. After this, the action sequences are optimized a finite number of times using gradient ascent over the rewards. Formally, the actions are updated using the following gradient update

$$A^j = A^{j-1} + \alpha \nabla_{A^{j-1}} R(A^{j-1}, S), j = 1, 2, \dots, J$$

In the equation above, all action sequences A are optimized with respect to their rewards. A^j represents the set of N action sequences in the jth gradient iteration. α is the learning rate. After J gradient updates, we update the sampling distribution to $\mathcal{N}(\mathbb{E}[A^J], \mathrm{Cov}(A^J))$. Using a gradient-based planner, we can eliminate the problems often faced in Zeroth-order planners.

Despite the success of gradient-based planners in model-based reinforcement learning, they rely on a reliable model to effectively navigate through the action space using reward gradients.

2.3 Transformer

A transformer [44] is a deep learning model often used for sequence-to-sequence modeling. Previously, the most commonly used models for these tasks were LSTMs and Gated Recurrent Units (GRU). While these models achieve good results in sequence-to-sequence modeling, they either have high computational costs due to the serial nature of their architecture or are unable to capture long-range dependencies within the input sequence.

On the other hand, transformers can effectively capture long-range dependencies within the input sequence and can parallelize the sequence-to-sequence modeling task. They achieve this performance through self-attention. In self-attention, the model measures the significance (also known as attention) of each part of the input sequence with respect

to other parts of the input sequence. More formally, to calculate the attention score, the model makes use of a set of Query (Q), Key (K), and Value (V) matrices to compute the attention scores. This is done with a scaled dot product

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

Here d_k is the projection dimension of each component of the input sequence. However, depending on our downstream tasks, we would like to avoid self-attention between specific components of the input sequence. This is achieved with the help of a mask M. Formally,

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

However, due to the parallelization, the model is permutation invariant. To circumvent this, the authors use positional encoding to inject some information about the indexing in the input sequence [44]. This positional encoding is a piece-wise function of sine and cosine functions of different frequencies. In a typical encoder-decoder-based transformer model (as in [44]), an encoder takes in a raw input sequence and produces an embedding of the input, which is attained via self-attention. The decoder uses the encoder output and previous transformer outputs as inputs to predict the rest of the output sequence with the help of multihead-attention between the two inputs. Through these features, the transformer model has produced ground-breaking results in sequence-to-sequence modeling.

Chapter 3

Methodology

In this chapter, we will go over the different methods applied in model-based reinforcement learning for bipedal robots. This chapter has been split into two main parts. The first part goes over the specifics of the model learning pipeline and the techniques used to learn the dynamics of the bipedal robot. The second part goes over the joint-level planning of the agent and model retraining in great detail.

3.1 Model Learning

As mentioned before, an essential component of model-based reinforcement learning is to learn the robot's dynamics. Here, we will use a transformer model for our downstream task. Typically transformers are used in natural language processing and computer vision. To the best of our knowledge, there has been no past work that uses a transformer to learn the dynamics of a continuous-time dynamical system.

We will now get into the different steps involved in training the model. First, we will go over the learning task, the different architectures of the models used, the loss function, and the training strategy.

3.1.1 Prediction Task

In the past, numerous approaches have focused on analytically deriving the dynamics of the robot using numerous optimization methods. However, as the degrees of freedom increase, accurately deriving the dynamics of a robot gets harder. Unfortunately, bipedal robots have numerous degrees of freedom, which increases the difficulty of analytically deriving their dynamics. Furthermore, the presence of multiple passive joints in the robot complicates the task of deriving the dynamics from scratch. Lastly, accurately programming the dynamics could also prove to be tedious.

Given the difficulty of deriving the dynamics analytically, we will instead be focusing on using deep learning methods. We are aiming to learn the following function,

$$\hat{s}_{t+1} = f(a_t, s_t) \tag{3.1}$$

Here f is a non-linear function which is a representation of our deep learning model. In the equation, s_t represents the continuous time state of our agent at time t, a_t represents the joint torques (PD targets) we wish to execute, and $\hat{s}_{t+1}$ represents the state predicted by the deep learning model f.

Unfortunately, as mentioned before, the environment is partially observable. Furthermore, the complexity of the dynamics will require more than one past state-action transition. For these reasons, the Markovian assumption in Equation 3.1 will not suffice for accurately modeling the dynamics. Hence, we will have to use a history of state-action transitions as context to model the dynamics of the agent. In addition, using a history of state-action transitions will help a deep-learning model understand the dynamics better. Hence, we modify Equation 3.1 to

$$\hat{s}_{H+1} = f((s_1,a_1),(s_2,a_2),(s_3,a_3),\dots(s_H,a_H)) \tag{3.2}$$

Here, H is the number of past state-action transitions we plan on using to predict the next state of the robot. During training, this H will be an essential hyperparameter to tune. It is important to understand that if H is too short, we cannot extract the features important to generalize the dynamics. In contrast, if H is too long, it could lead to overfitting and increase the computational cost. However, as mentioned in [30, 6], the learning task is simplified if we were to predict $s_{H+1} - s_H$ instead of simply predicting s_H. This is because of the similarity between the states and actions over the H-length history. Hence, we build up on Equation 3.2 to get

$$\hat{\delta}_H = f((s_1,a_1),(s_2,a_2),(s_3,a_3),\dots(s_H,a_H)) \tag{3.3}$$

In Equation 3.3, $\hat{\delta}_H$ is a prediction for $s_{H+1} - s_H$. Informally, instead of predicting the next state by itself, we will be predicting the differences in states. This acts as a proxy for predicting the next state of the robot, i.e., we will instead be learning a feature of the next state. Lastly, we are taking the squared L2 norm of the difference.

In summary, using a dataset $\mathcal{D}$ of state-action transitions, our goal will be to predict the differences, i.e., $s_{H+1} - s_H$ given H previous state-action transitions.

3.1.2 Model Architecture

Above, we discussed that the model predicts $\hat{\delta}_t$ - a prediction of $s_{t+1} - s_t$ - as output. As an input, we take in H previous state-action transitions $\begin{bmatrix} s_t \\ a_t \end{bmatrix}$. Instead of directly using the raw states and actions as input, each state-action transition will act as an input to a representation learning layer. For every state-action pair, we learn a lower-dimensional representation ϕ_t. This representation learning layer consists of a fully connected neural network with 3 hidden layers and 1000 hidden units each to project the 59 dimensional state-action pair to a specific dimensionality. Through this representation, we can pass every state-action pair through a non-linearity (ReLu in our case). Upon processing the representation, we will add sinusoidal positional encoding [44] to incorporate temporal information in all ϕs. These are used as inputs in a transformer in time model [44] to predict the next state of the robot.

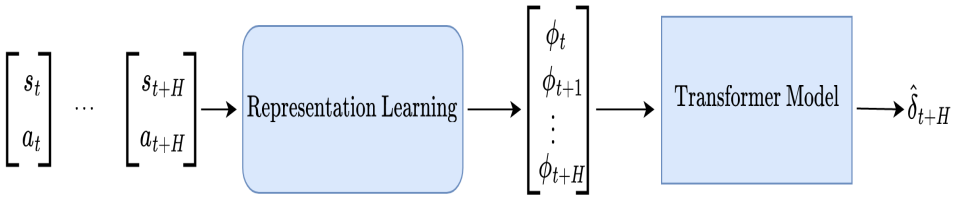

Figure 3.1: Model Learning Pipeline. H state-action transitions act as input to a representation layer which each of them to a lower dimensional space. These act as input to a transformer model to predict δ_{t+H}

For our downstream task, we will use an encoder-decoder-based transformer model [44]. We are using a transformer for our task because of its ability to model long-range dependencies between the components of an input sequence via the self-attention mechanism. An encoder-decoder-based transformer model is used because the input and output of the model are of different dimensions and represent different values. Specifically, the input is a representation of the state-action transitions, and the output is the difference between the states. The model learning pipeline has been depicted in Figure 3.1. If we consider our input sequence at time t to be $X = ([s_t, a_t]^T, [s_{t+1}, a_{t+1}]^T, \ldots, [s_{t+H}, a_{t+H}]^T), |X| = H$. We wish to model the output sequence $Y = (s_{t+1} - s_t, s_{t+1} - s_{t+1}, \ldots s_{t+H+1} - s_{t+H}), |Y| = H$. Initially, X acts as input to the encoder of the transformer f_{enc}. This gives

$$\hat{X} = f_{\text{enc}}(X)$$

Since all state-action transitions of X have been observed, there are no constraints imposed during encoder-level self-attention. Hence, we do not need an encoder mask. However, in the decoder phase, we are attempting to learn the distribution $\mathbb{P}(Y_{1:H}|\hat{X})$ via the decoder model f_{dec}.[1]. Using the conditional probability chain rule, we get

$$f_{\text{dec}} \approx \mathbb{P}(Y_{1:H}|\hat{X}) = \mathbb{P}(Y_1|\hat{X})\mathbb{P}(Y_2|Y_1, \hat{X}) \ldots \mathbb{P}(Y_H|Y_1, Y_2 \ldots Y_{H-1}, \hat{X})$$

This can be written as

$$f_{\text{dec}} \approx \mathbb{P}(Y_{1:H}|\hat{X}) = \prod_{i=1}^{H} \mathbb{P}(Y_i|Y_{1:i-1}, \hat{X})$$

During decoding, $\forall i$ components of the output sequence, we use the previous outputs and the encoder output $\hat{X}$ to predict the next step. Hence, during self-attention between components of the decoder input ($Y_{1:i-1}$), we need to prevent self-attention between unseen components of the output sequences with the help of a mask M. This would consequently establish a causal chain. Specifically, we would have to define an upper triangular matrix of size $M \in \mathbb{R}^{H \times H}$ as a mask with the components above the diagonal equal to $-\infty$ and other components as 0.

[1]The subscript for Y indicates indices of the sequence

3.1.3 Loss Function

A loss function is an essential component for training deep learning models. Quite simply, the robustness of the loss function is essential to training a robust deep learning model. Here, we will discuss the numerous components of the loss function and their implications.

3.1.3.1 State Prediction Loss

Since we are predicting $\hat{\delta}_H = s_{H+1} - s_H$, a straightforward method to improve this prediction is to use MSE between the true difference between s_{H+1} and s_H and the model's predicted difference. Formally, given a dataset $\mathcal{D}$ of state action transitions $(s_t, a_t)\forall t$, our single-step loss function can be defined as

$$\mathcal{L}_{\text{single}} = \frac{1}{n} \sum_{i=1}^{n-H} \left\| (s_{i+H} - s_{i+H-1}) - f(s_i, \ldots, s_{i+H-1}, a_i, \ldots, a_{i+H-1}) \right\|_2^2 \tag{3.4}$$

In equation 3.4, n is the number of state-action transitions in the dataset $\mathcal{D}$ or, more formally, $n = |\mathcal{D}|$. In the equation above, H is the number of past state-action transitions acting as context to predict the output sequence. We are implementing a sliding-window scheme to train the model. This is a commonly used method for time-series forecasting in machine learning [5, 7]. We will regard the loss function in Equation 3.4 as the *state prediction loss*.

In addition to simply measuring state prediction loss, we can also add greater emphasis to the prediction of specific components of the state space. A weighted loss function will provide us with the added flexibility of penalizing errors in components that are harder to learn.

$$\mathcal{L}_{\text{single}} = \frac{1}{n} \sum_{i=1}^{n-H} \left\| w \circ [(s_{i+H} - s_{i+H-1}) - f(s_i, \ldots, s_{i+H-1}, a_i, \ldots, a_{i+H-1})] \right\|_2^2 \tag{3.5}$$

In Equation 3.5, $\forall i$, we compute the squared $L2$ norm of the element-wise product (Hadamard Product [14]) between w and the difference between the predicted and ground truth values. Here, w has same dimensionality as s

However, as noted in [49], it is better to predict numerous steps in the future rather than simply implementing single-step prediction. Furthermore, in our planner, we hope to use an MPC which takes predictions for up to τ steps in the future. As we have seen before, an MPC predicts τ steps in the future and accumulates the reward over that period. The next optimal action executed maximizes this reward over τ steps in the future. To implement this, we need to make sure that our model can accurately predict multiple steps in the future.

In order to accommodate this change, we need to modify our loss function to reflect the accuracy of model predictions over τ steps in the future. Formally,

$$\mathcal{L}_{\text{multi}} = \frac{1}{n} \sum_{i=1}^{n-H-\tau} \frac{1}{\tau} \sum_{t=0}^{\tau-1} \left\| w \circ [(s_{i+k} - s_{i+k-1}) - f(s_{i+t}, \ldots, \hat{s}_{i+k-1}, a_{i+t}, \ldots, a_{i+k-1})] \right\|_2^2$$
$$k = H + t \tag{3.6}$$

In the above equation, n and H have the same meaning as they did in Equations 3.4,3.5. However, we have introduced a new variable, τ, which is the number of steps in the future we want to model to predict accurately. To predict more than one step, earlier predictions act as inputs to the model. For instance, if we consider $t \in \mathbb{Z}$ such that $1 < t < \tau$, we set up a recursive definition to predict future states.

$$\hat{s}_{i+H+t} = \hat{s}_{i+H+t-1} + f(s_{i+t}, \ldots, \hat{s}_{i+H+t-1}, a_{i+t}, \ldots, a_{i+H+t-1}) \tag{3.7}$$

As seen in equation 3.7, the model's outputs act as an input to predict the future steps (as t increases), i.e., states beyond timesteps $i+H$. Furthermore, this is similar to what will be done in the planner.

Predicting all τ steps accurately from the first epoch is a difficult learning task. The model inaccuracies would accumulate over the τ steps and make the learning task difficult and infeasible. To circumvent this issue, we use the Teacher forcing training strategy which will be discussed in a later section.

3.1.3.2 Gradient Penalty

Gradient Penalty is often used in the context of GAN's [12] to constrain the norm of the gradient of the model's output with respect to the model's input to be approximately one in an actor-critic setup. This penalty forces the gradient to be close to the unit norm and thereby constraining the Lipschitz constant to one. Lipschitz continuity and having a low Lipschitz constant is a desirable property for deep learning models [38]. A function (or deep learning model in our case) f is said to be Lipschitz Continuous if for $x_1, x_2 \in \mathbb{R}$

$$|f(x_1) - f(x_2)| \leq K|x_1 - x_2| \tag{3.8}$$

Here, $K \in \mathbb{R}^+$ is called the Lipschitz constant.

In our case, we will use the gradient penalty as a regularization method to make the model predictions more stable. For example, if $x_2 = x_1 + \varepsilon$, where $\varepsilon \in \mathbb{R}^+$ and our deep learning model f is Lipschitz continuous with Lipschitz Constant $K = 1$

$$\begin{aligned} |f(x_1) - f(x_2)| &\leq |x_1 - x_1 - \varepsilon| \\ |f(x_1) - f(x_2)| &\leq \varepsilon \end{aligned}$$

Above, we notice that by constraining the Lipschitz constant, we can reduce the model's sensitivity to small changes in input. This property is essential for dynamics models because the state predictions are not drastically altered when the state-action transitions are slightly perturbed. Formally, gradient penalty can be incorporated in loss function as,

$$\begin{aligned} \mathcal{L}_{\text{gradient}} &= \left\| \nabla_{(s_{k:j}, a_{k:j})} f(s_{k:j}, a_{k:j}) \right\|_2^2 \\ \mathcal{L} = \frac{1}{n} \sum_{i=1}^{n-H-\tau} \frac{1}{\tau} \sum_{t=0}^{\tau-1} \lambda \cdot \mathcal{L}_{\text{gradient}} &+ \left\| w \circ \left[(s_{j+1} - s_j) - f(s_k, \ldots, s_j, a_k, \ldots, a_j) \right] \right\|_2^2 \\ k = i+t, j &= i+H+t-1 \end{aligned} \tag{3.9}$$

Here, λ is a hyperparameter for scaling the gradient penalty. λ has to be tuned to avoid the gradient penalty from dominating the loss function.

3.1.4 Teacher forcing

Algorithm 1: Teacher Forcing($\mathcal{D}, \tau, f, H, \psi, \lambda, w$)

```
/* f:  Deep Learning Model                                          */
/* ψ:  Teacher forcing conditions                                   */
/* H:  Context vectors for modelling dynamics                       */
/* λ:  Scaling for gradient penalty                                 */
/* 𝒟:  Dataset of state-action transitions                          */
/* τ:  Number of future state-predictions                           */
/* w:  Weights for state-prediction loss                            */
ℒ ← 0
t ← 1
for epoch ← 0 to num_epochs do
    for i ← 0 to |𝒟| − H − τ do
        for j ← 0 to t − 1 do
            /* s̃, ã are previous H state-action transitions to model
               output sequence                                      */
            s̃, ã ← (s_{i+j}, ..., s_{i+H+j}), (a_{i+j}, ..., a_{i+H+j})
            δ̂_{i+H+j} ← f(s̃, ã)

            /* Loss computed using Equation 3.9                     */
            gradient_penalty ← ‖∇_{(s̃,ã)} f(s̃, ã)‖_2^2
            state_prediction_loss ← ‖w ∘ [(s_{i+H+j+1} − s_{i+H+j}) − δ̂_{i+H+j}]‖_2^2
            ℒ ← ℒ + state_prediction_loss + λ · gradient_penalty

            /* state prediction using previous state and predicted
               difference [2]                                       */
            ŝ_{i+H+j+1} = s_{i+j+H} + δ̂_{i+H+j}
        /* backpropagate loss and optimize weights                  */
    if ψ and t < τ then
        /* Predict additional step if conditions ψ are met          */
        t ← t + 1
        if t = τ then
            return f
return f
```

When predicting an output sequence of length greater than one, incorrect predictions in one of the intermediate steps can cause the subsequent predictions to be inaccurate. Consequently, the errors would accumulate.

Let us consider an example to better understand this problem in the context of state prediction. For instance, we will use $H = 5$ state-action transitions to predict $\tau = 3$

[2] This is achieved using PyTorch's backward and step API calls

steps in the future. If we consider f's outputs to be perfect, then

$$s_6 \approx \hat{s}_6 = s_5 + \hat{\delta}_5$$

In the equation above, $\hat{\delta}_5$ leads to approximately zero error. However, in the initial stages of model training, model predictions are expected to be inaccurate. Hence, we can model this inaccuracy as

$$s_6 \neq \hat{s}_6 = s_5 + \hat{\delta}_5 + \varepsilon$$

Where $\varepsilon \neq 0 \in \mathbb{R}$. ε represents the error in the model's prediction and $\hat{\delta}_5$ is the approximately the same as the ground truth. However, in a scenario with no prediction errors, the robot will end up at s_7 after executing a_6 at s_6. But, $\hat{s}_6$ is incorrect (assumed in previous equations). This would alter all future states (alter the trajectory of the agent). In this example, we notice how an incorrect intermediate prediction can lead to multiple errors getting accumulated. This issue will be exacerbated as τ increases.

To avoid this, we will use a strategy similar to Teacher Forcing - a technique often used to train different types of recurrent neural networks (RNN) [20]. Specifically, instead of directly predicting all τ steps, we will start with an easier learning task and predict $t = 1$ states in the future accurately. We keep training the model to predict $t = 1$ steps in the future until a set of empirically determined conditions ψ are satisfied. Once the conditions are satisfied, we increment t by one. This process will continue until $t = \tau$ states in the future are correctly predicted. It is important to note that t is the same during training and inference.

This is a variant of teacher forcing, as we are not feeding any ground truth states back into the model. We instead incrementally build the difficulty of the prediction task. As a result, we do not run into problems during inference, unlike in [20]. The teacher-forcing algorithm we use to train the model has been presented in Algorithm 1.

3.2 Planning

As mentioned earlier, joint-level planning is an essential component of a model-based reinforcement learning pipeline. The optimal action is executed at the current state of the robot with the help of a reward function.

In this section, we will first go over the reward function used for optimizing the action sequence. Then, we will go over our novel joint-level planning algorithm. Lastly, we will discuss the procedure for retraining the initial dynamics model.

3.2.1 Reward function

A reward function is an essential component of any reinforcement learning pipeline. It is an objective metric of the agent's performance at any timestep. In our case, we want a bipedal robot to perform a form of locomotion without falling. Without loss of generality, our reward function would enforce our agent to follow a walking trajectory.

We note that in [40], a bipedal robot uses a model-free policy to follow a walking trajectory. We plan on re-using this reward function for our problem too. In [40], upon executing a specific action, the state of the agent is compared with that of the state of the agent following a reference walking trajectory in Mujoco at time t. This reference trajectory is an open-sourced expert walking trajectory that has been carefully tuned by a human.

At each timestep, the reward is computed by measuring the differences in joint positions, toe spring positions, the center of mass positions, joint velocity, and joint orientation. These differences (or errors) are squared $L2$ norm between the desired and the ground truth values. More formally, as in [40], we define the reward function as a weighted sum of the negative exponent of the error.

$$\begin{aligned} R = 0.3 \cdot e^{-(\text{joint orientation error})} + 0.2 \cdot e^{-(\text{joint position error})} + 0.2 \cdot e^{-(\text{x velocity error})} \\ +0.2 \cdot e^{-(\text{y velocity error})} + 0.05 \cdot e^{-(\text{spring error})} + 0.05 \cdot e^{-(\text{center of mass position})} \end{aligned} \tag{3.10}$$

The sum of all weights is 1. Since each error is a squared $L2$ norm, we can say that each error term is non-negative. Furthermore, we know that $0 < \exp(-x) \leq 1$ when $x \geq 0$ as in our case. Hence, using this bound, we get

$$\begin{gathered} 0 < R \leq 0.3 \cdot 1 + 0.2 \cdot 1 + 0.2 \cdot 1 + 0.2 \cdot 1 + 0.05 \cdot 1 + 0.05 \cdot 1 \\ 0.0 < R \leq 1.0 \\ R \in (0, 1] \end{gathered}$$

Lastly, when rolling out an action sequence of length τ in the planner, we will compute the discounted sum of the rewards over τ steps. Essentially, if we start from state s_0 and execute an action sequence $(a_0, a_1, \ldots, a_\tau)$, our resulting state sequence can be represented as $(s_1, s_2 \ldots s_{\tau+1})$. To compute the aggregate reward for this sequence, we will be using a discounted reward function

$$R_\tau = \sum_{t=1}^{\tau+1} \gamma^t \cdot R(s_t) \tag{3.11}$$

Where $R(s_t)$ is the reward at state s_t and $\gamma \in [0, 1]$ is the discount factor. This reward is computed using Equation 3.10. We use a discounted reward function to give greater emphasis to the immediate rewards compared to rewards in the future [31, 9].

3.2.2 Gradient-CEM Planning

Algorithm 2: Gradient-CEM planning$(t, \tau, \mu_0, \Sigma_0)$

/* τ: Number of Steps in the future being predicted */
/* t: Current timestep */
/* μ_0, Σ_0: Initial distribution parameters */
/* N, K: number of initial samples and elite samples */
/* C, G: Number of CEM and gradient iterations */

for $i \leftarrow 0$ *to* C **do**
  $a^N_{t:t+\tau} \sim \mathcal{N}(\mu_i, \Sigma_i)$
  /* f predicts states $\hat{s}^N_{t:t+\tau}$ after executing $a^N_{t:t+\tau}$.Compute the discounted reward for those predicted states [3] */
  $r^N \leftarrow \text{reward}(\hat{s}^N_{t:t+\tau})$
  /* topk gives K/N actions with highest reward (K elite actions) [4] */
  $a^K_{t:t+\tau} \leftarrow \text{topk}(r^N, k = K)$
  for $j \leftarrow 0$ *to* G **do**
    $r^K \leftarrow \text{reward}(\hat{s}^K_{t:t+\tau})$
    $\text{negative\_reward\_sum} \leftarrow -\sum_{k=1}^{K} r^k$
    /* backpropagate negative_reward_sum, optimize $a^K_{t:t+\tau}$ inplace via SGD [5] */
  $\mu_{i+1}, \Sigma_{i+1} \leftarrow \mathbb{E}[a^K_{t:t+\tau}], \text{Cov}(a^K_{t:t+\tau})$
$a^N_{t:t+\tau} \sim \mathcal{N}(\mu_C, \Sigma_C)$
$r^N \leftarrow \text{reward}(\hat{s}^N_{t:t+\tau})$
/* first action of action-sequence with highest reward */
$a^*_t \leftarrow \text{topk}(r^N, k = 1)$
return a^*_t

In the background chapter, we described two essential methods of joint-level planning. The first is sampling-based planning, and the other is gradient-based planning. However, in CEM and gradient-based planning, the experiments either use an analytical model or a lightweight deep learning model (fully connected neural networks) [28, 3].

Unfortunately, we have to use a larger model like the transformer to approximate the dynamics of a bipedal robot. As a result, our model will have more parameters than a fully connected neural network. This will increase the computational cost in computing the gradient of the reward of the predicted state with respect to the latest action. In addition to this, [3] proposes to perform the gradient computation for all N samples in

[3] This involves rolling out the action sequence $a_{t:t+\tau}$ and using a sequence of past state-action pairs to predict the differences in states for the next τ timesteps. The differences in states predictions are converted to state predictions by simply addition

[4] Equivalent to PyTorch's topK library function

[5] This is achieved using PyTorch's backward and step API calls

every gradient iteration. This is not feasible when the number of model parameters is in the range of millions.

However, we can circumvent this issue by modifying the algorithm 1 in [3]. The trick is to perform gradient updates on the K elite samples. Furthermore, since $K << N$, we can use far less memory when optimizing the actions. We will update the action distribution using the mean and covariance of the K elite samples, which were updated using gradient ascent. With these modifications, we can scale algorithm 1 of [3] to larger deep learning models without reducing the number of initial samples N.

The modified algorithm has been presented in Algorithm 2. In the pseudocode, C is the number of CEM iterations, and G is the number of gradient iterations per CEM iteration. $f, \texttt{reward}$ are the dynamics model and the reward function, respectively. As described before, the deep learning model takes in previous H state-action transitions, including the latest action sequence we sampled/optimized, and predicts the differences in states. The reward function takes in the model predictions and computes the discounted reward function for the action sequence of length τ. In line 7 of the pseudocode, we use a vanilla sum of the discounted reward of the K elite action sequences. This sum allows us to optimize all K action sequences in batches instead of individually optimizing every K action sequence. Furthermore, this backward call is made on the negative reward sum because we are performing SGD, which would minimize the negative reward sum. This is similar to performing gradient ascent to maximize the reward sum.

μ_0, Σ_0 are the sampling distribution's initial mean and covariance matrices, respectively. Initially, $\mu_0 = \mathbf{0}$ and $\Sigma_0 = I$. $N, K(K << N)$ are the number of initial samples and elite samples, respectively. At the implementation level, outside the gradient updates loop, we compute predictions without gradients, i.e., as inference. Lastly, τ is the number of steps in the future from timestep t we are planning for in order to prevent overfitting to the next time step.

In this planner, we are obtaining the best action for the next timestep by leveraging the sampling-based aspect of CEM and efficiently performing gradient ascent on the elite samples.

3.3 Model Retraining

Algorithm 3: Model Based DAgger(f, π^*, ρ, ϕ, M, sim)

```
/* f:  Initial dynamics model                                          */
/* π*:  Expert LSTM policy which uses current state s_t               */
/* ρ:  Model-based controller which uses f, current state s_t         */
/* M:  Number of state-action transitions to aggregate                */
/* ϕ:  Time taken for two walking steps in simulator                  */
/* sim:  Cassie simulator                                             */
D ← ∅
for n ← 1 to ϕ do
    D_n ← ∅
    /* Initial state of simulator                                     */
    s_0 ← sim.state()
    t ← 0
    while |D_n| < M do
        /* Boolean model_free decides if π* should be used [6]        */
        if model_free then
            a_t ← π*(s_t)
        else
            /* Model based controller which decides optimal action
               given s_t using f [7]                                  */
            a_t ← ρ(s_t, f)
        /* Execute action a_t in simulator and store (s_t, a_t) in D_n */
        s_{t+1} ← sim.execute(a_t)
        D_n ← D_n ∪ {(s_t, a_t)}
        t ← t + 1
    /* Combine D_n and D                                              */
    D ← D ∪ D_n
    /* Retrain f using D                                              */
return f
```

Model retraining is an essential component of model-based reinforcement learning. When the state and action spaces are huge, it is not computationally tractable to perfectly model the dynamics of our agent without any errors. Using the training methods described above, we can attempt to reduce the error over a finite dataset. However, some unseen states and actions could be challenging to predict for the model. As explained before, with every wrong prediction, model inaccuracies could accumulate and consequently cause our agent to fall. As a result, the dynamics model has to be retrained effectively to improve performance on unseen states and actions.

Unfortunately, training a transformer model requires a large training set. To collect these many points soon, the planner would have to continue walking over an extended period of time without falling. As a solution, we propose to use a variant of the

[6] This flag is arbitrarily False n times every ϕ timesteps.

[7] Optimal a_t is computed using the planner in Algorithm 2

Dataset Aggregation Method (DAgger) [37]. DAgger is commonly used in a model-free setting, where throughout N timesteps, the action executed in the simulator is a linear combination of the action from our initial policy $\tilde{\pi}$ and an expert policy π^* at the same state. These state-action transitions are aggregated in $\mathcal{D}$. Upon collecting N points, the model is retrained using $\mathcal{D}$. We repeat this process until the policy achieves a certain result. A detailed explanation is given in Algorithm 3.1 in [37].

In our case, we will implement a variant of DAgger to collect more data for retraining the initial transformer model. The approach used has been summarized in algorithm 3. As our expert policy π^*, we will use the model-free LSTM policy from [40]. As $\tilde{\pi}$, we will use the actions generated by the model-based reinforcement learning controller ρ. Instead of a linear combination of the two actions (as in [37]), we will either execute the action from π^* or the action from ρ. To seamlessly transition to ρ, we will execute n arbitrary actions from ρ every ϕ timesteps. For every ϕ timestep, we will execute actions from ρ at n random timesteps. Otherwise, we will execute actions from π^*. This is encapsulated by `model_free` flag in the pseudocode. ϕ is the number of actions it takes our agent to take two steps in a given simulator. We start from $n = 1$ and aggregate $N >> 0$ state-action transitions. We first retrain the model using the teacher forcing strategy and then increment n (frequency of actions from ρ). We repeat the process until $n = \phi$.

Using this, we will retrain f and quickly collect a large dataset of state-action transitions corresponding to a walking trajectory from the simulator without falling during the episode.

Chapter 4

Experimental Setup

In this section of the thesis, we will get into greater detail about the experimental setup. First, we will briefly describe Cassie - the bipedal robot we are considering for our experiments. Second, we will discuss the potential simulators and the simulator we are currently using for visualization, data collection, and training. Lastly, we will delve into the data collection process.

4.1 Cassie Robot

The Cassie Robot is a bipedal robot [26] humanoid robot which walks like humans (has two legs) and can reproduce different humanoid gaits.

As mentioned earlier, the robot has numerous interconnected joints and twenty degrees of freedom. Out of these, six are in the pelvis. Each state vector of the robot can be represented using a 49 dimensional vector ($s_t \in \mathbb{R}^{49}$). A summary of the entire state space (components of s_t) is included in Table 4.1. We notice that the state-space consists of zeroth (positions), first (velocity), and second (acceleration) order information of the robot.

Component Indices	Description
[0]	Pelvis Height
[1, 4]	Pelvis Orientation
[5, 14]	Motor Positions
[15, 17]	Pelvis Translational Velocity
[18, 20]	Pelvis Rotational Velocity
[21, 30]	Actuated Joint Velocities
[31, 33]	Pelvis Translational Acceleration
[34, 39]	Unactuated Joint Positions
[40, 46]	Unactuated Joint Velocities

Table 4.1: State Space Vector Summary

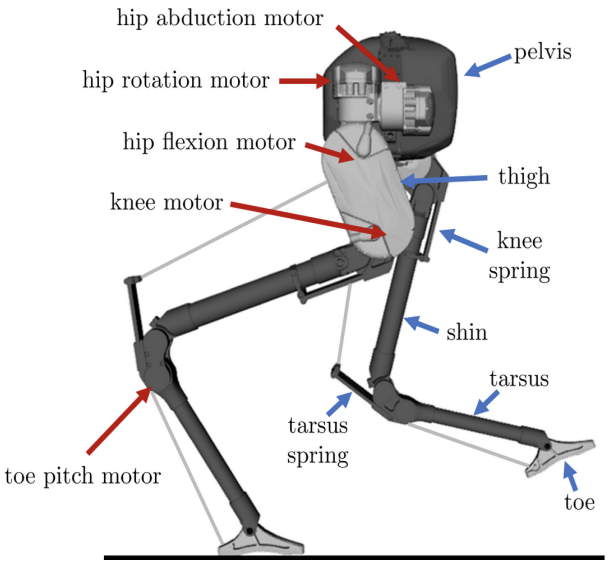

Figure 4.1: The different motor positions in the Cassie Robot [23]. These motor position are controlled by PD targets in the form of an action vector.

The robot is controlled using a 10 dimensional action vector (i.e. $a_t \in \mathbb{R}^{10}$). This vector specifies the motor positions in Cassie. These desired positions are converted to torque with the help of a PD controller.

The motors controlled by the action correspond to the abduction, rotation, hip pitch, knee, and toe motors in both legs. These motor positions can be visualized in Figure 4.1. In the figure, the red arrows correspond to the joints which are directly affected by the action vector [23, 22].

4.2 Simulator

In robotics research, it is more feasible to train and collect data in a simulator than in the real world because deploying robots without extensive training tends to be difficult, expensive, and, sometimes, unsafe. Furthermore, simulators provide the means to swiftly understand the implications of the changes made to code or algorithm. As a result, reinforcement learning algorithms are implemented and tested in a simulator that approximately models the real world. For our task, we considered existing and open-sourced simulators like Mujoco and Gazebo rather than building one from the ground up.

4.2.1 Gazebo

Gazebo [17] is an open-source 3D simulator used for simulating and visualizing numerous robots and objects. Most importantly, Gazebo provides a Robot Operating System (ROS) [32] backend, which is useful for deploying the robot in the real world.

The Gazebo simulator for Cassie [35] (Figure 4.2) makes use of ROS and the Gazebo

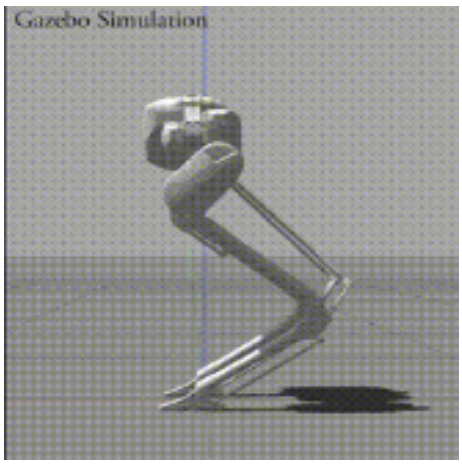

Figure 4.2: A depiction of the Cassie Robot in Gazebo Simulator [35].

physics engine to simulate and visualize Cassie. The repository consists of numerous controllers, which decide the torque to send to the robot and an interface to send control inputs to the robot. As in the GitHub repository, this simulator for Cassie uses C++ to effectively determine constraints imposed on the robot and compute the kinematics of the robot. In this setup, users launch `roslaunch` files for the simulator, hardware, and controller. A `roslaunch` file is a script used to launch several ROS nodes (a program that performs computation via ROS) locally. As a result of this setup, we can easily send control inputs to Cassie. Lastly, the repository also consists of a script to plot the data logged from experiments.

Unfortunately, despite these advantages, there are quite a few disadvantages. Specifically, this simulator uses C++, MATLAB, and roscpp for sending control inputs to the robot and receiving observations from the robot. Unfortunately, training, debugging, and deploying deep learning models in MATLAB or C++ is tedious compared to other languages like Python.

4.2.2 Mujoco

Like Gazebo, Mujoco [43] is a physics engine used for research and development in robotics. However, their applications span beyond the field of robotics. To deploy the robot in the real world using Mujoco, the robot receives control inputs over a User Datagram Protocol (UDP) connection. A UDP connection is used so that applications can send messages as datagrams to hosts and servers over the IP network.

The Mujoco simulator for Cassie (Figure 4.3) by Agility Robotics is written in Python and has a C++ backend [36]. The simulator consists of several safety mechanisms and state estimation packages. In addition, the simulator supports numerous visualization features such as highlighting constraints, actuator, and joint positions in the robot. Furthermore, the newer versions of the simulator provide several features to plot contact forces applied to the robot. Unlike the Gazebo simulator, the Mujoco simulator for Cassie does not have a built-in controller. As a result, to effectively visualize the robot in the simulator, we need to provide action vectors to the PD control.

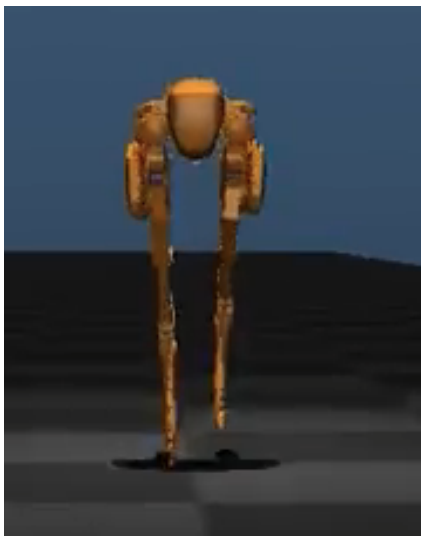

Figure 4.3: A depiction of Cassie Robot in the Mujoco Simulator [36].

For this work, we have decided to use Mujoco as the primary simulator because of the following reasons

- Numerous previous works [40, 46, 8] involving Cassie use the Mujoco simulator because of the ease with which it integrates with Python and PyTorch.
- Mujoco gives access to several essential physical quantities like contact positions and forces, which are useful for training deep learning models.
- When controlled by a specific policy that produces a desired gait, it is straightforward to collect state-action transitions.
- Given a fixed action sequence $(a_0, a_1, a_2, \ldots, a_t)$, and an initial state s_0, Mujoco simulator adds no noise to the resulting states $(s_1, s_2, \ldots, s_t)$, i.e., there is no simulator level noise which can alter the state sequence. While this is an advantage in simulation, it does not model real life.

4.3 Data Collection

An essential component of this project is to collect state-action transitions from the Mujoco simulator. These transitions are useful when using deep learning techniques to learn the dynamics of Cassie. Formally, if the robot were to start at state s_0 at the beginning of an episode, it transitions to s_1 upon executing action a_0. Likewise, we transition to s_2 after executing action a_1 at state s_1. As a part of our data collection process, our goal is to collect a sufficient amount of state-action transitions to effectively model the dynamics of the agent. In addition, we would like this data to execute a specific gait throughout the episode.

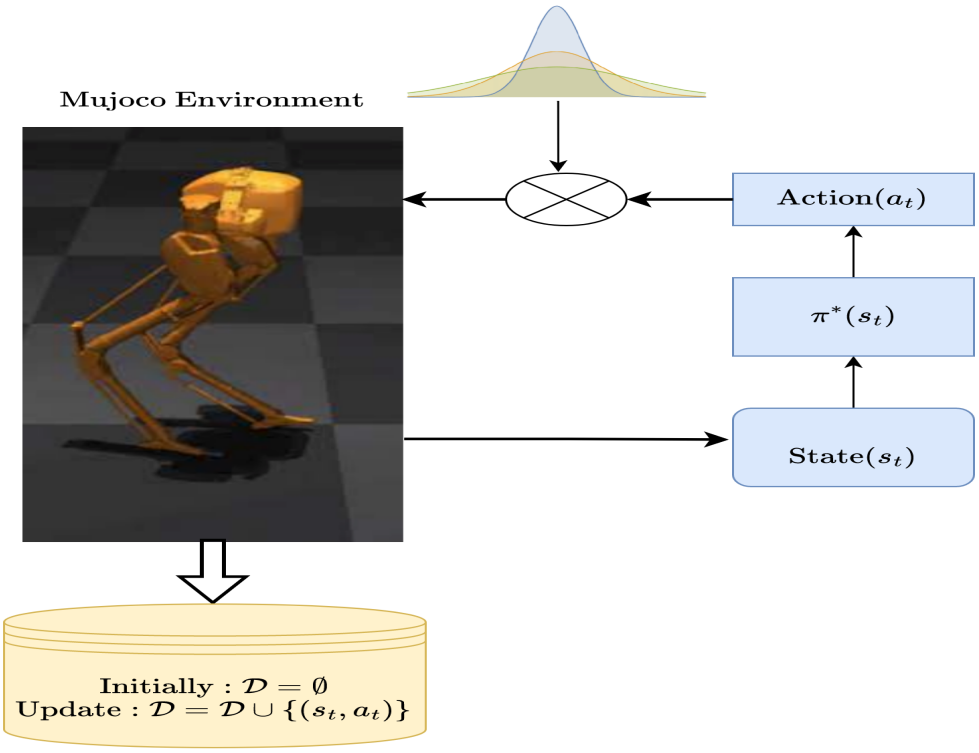

Figure 4.4: Data Collection Pipeline using $\pi^*(s_t)$. In this case, give s_t, π^* outputs the optimal action a_t. We collect $(s_t, a_t) \forall t$ in a dataset $\mathcal{D}$. In the pipeline, we add Gaussian noise $\mathcal{N}(0, \sigma^2)$ to every component of a_t to increase exploration.

4.3.1 Deep learning model for data collection

Unfortunately, manually making the robot perform a specific gait is challenging, time-consuming, and can also result in the robot falling in the long run (actions not being stable in the longer run). Instead, we will use a model-free deep learning policy to collect the state-action transitions for our task.

Next, our goal will be to develop a model-free deep learning policy for Cassie with the ability to walk. This would require a robust controller. However, there is no built-in controller included in Mujoco. One of the past works, [40], uses an LSTM deep learning model to learn a walking policy. Hence, as a part of our data collection pipeline, we plan on training the LSTM-based walking policy using the pseudo-code specified in [40] (Algorithm 1 - Recurrent PPO).

This algorithm was re-implemented and trained on a regular CPU for 48 hours using Mujoco 2.0 as a simulator. Furthermore, we successfully reproduced the rewards they received. Most importantly, the robot exhibits walking behavior when controlled with this LSTM policy. For the sake of notation, we will refer to this policy as π^* in the future.

4.3.2 Using model-free policy for data collection

Based on the previous section's discussion, we will use π^* to collect the state-action transitions of Cassie. Since it has been trained to walk, given a specific initial state, $\pi^*(s_t)$ can predict an action sequence corresponding to walking over a long time. Since it is a pre-trained model, we can quickly collect large volumes of data. To do this, we will define a simulator-level velocity of $0.0509m/s$. Furthermore, we will fix an initial state s_0 for each episode (s_0 for all experiments specified in Appendix A.3).

To store state-action transitions in $\mathcal{D}$, we will assume that at $t = 0$, $\mathcal{D} = \emptyset$ and the robot is at s_0 - initial state. This is an input to π^* and it outputs a_0. Once this action is executed in the simulator, (s_0, a_0) are appended to $\mathcal{D}$ (similar to a union operation). We repeat this as t increases in an episode. We repeat this process over numerous episodes. When collecting data, we need to decide the length of a single episode and how many episodes of state-action transitions from initial state s_0 we intend on collecting.

To increase Cassie's exploration in Mujoco and increase the stochasticity of the dataset, we will add Gaussian noise to each dimension of the action vector a_t outputted by π^*, i.e., actions that correspond to perfect walking. More formally,

$$\hat{a}_t = a_t + \mathcal{N}(\mathbf{0}, \sigma^2)$$

From the above equation, we will add Gaussian noise to every dimension of the action vector a_t with mean 0 and standard deviation σ. We will name this *action noise distribution*. Adding noise to a_t will result in $\hat{a}_t$. As a result of this change, $\mathcal{D}$ will consist of state-action transitions produced by $\hat{a}_t$ instead of a_t. σ needs to be chosen such that the noise does not cause the robot to fall. The complete data collection pipeline has been summarized in Figure 4.4.

Chapter 5

Experimental Results

In this section, we will go over the experiments performed and their results. The experiments are divided into two parts. The first set of experiments is related to model learning and approximation of Cassie's dynamics. Through them, we hope to motivate the use of a transformer model, the importance of the teacher forcing training procedure, and shed more light on the specifics of the transformer model (hyperparameters) and the performance of the model.

In the second part, we will go over experiments related to the planner. In the planner, we hope to shed some light on the flaws of a zeroth-order planner like CEM and the performance improvement with a gradient-based planner. In the end, we also hope to go over the improvements introduced by effective model retraining methods like DAgger.

5.1 Model Learning

Here, we will primarily focus on experiments related to model learning and the different steps involved in approximating the dynamics of Cassie. First, we will discuss how we attempted to learn the dynamics using a fully connected neural network. Upon discovering its failures, we will discuss how we used the transformer model to approximate the model's dynamics and achieve good out-of-sample and in-sample performance.

5.1.1 Fully Connected Neural Network Dynamics Model

Before delving into the Transformer model, we hope to motivate the use of a powerful model like the Transformer. To that end, we used state-action transitions collected using the pipeline described in Figure 4.4. We collected ≈ 10000 state-action transitions as our training set using action noise distribution $\mathcal{N}(0, 0.01)$ for each component. As our validation, we used a subset of our training set. This meant we used the first 70% of the dataset as our training set and the final 30% as our validation set. Hence, we used ≈ 7000 state-action transitions for training and ≈ 3000 state-action transitions for validation. Likewise, as our test set, we collected a set of ≈ 4000 unseen state-action transitions which correspond to a walking trajectory using action noise distribution

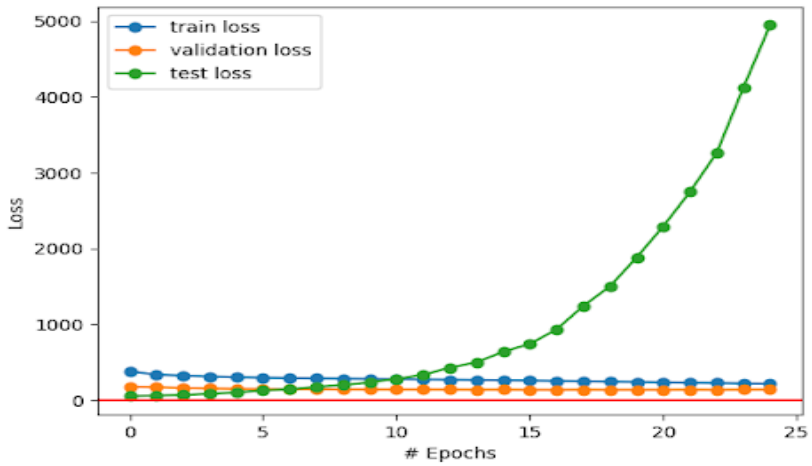

Figure 5.1: Learning Curve after training Fully Connected Neural Network based dynamics model for 25 epochs.

$\mathcal{N}(0, 0.03)$ for each component. As formalized before, we hope to use H past state-action transitions to predict $\hat{\delta}_H = s_{H+1} - s_H$. At all times t, we will use the sliding window protocol across the entire train set to model the following function

$$f(s_{t-H}, a_{t-H}, \ldots, s_t, a_t) = \hat{\delta}_H$$

As this is an initial experiment, we will only perform single-step prediction. Furthermore, we will use the simple loss function in Equation 3.4 to gauge the ability to predict the entire state space (uniform weight to each component). The model architecture and hyperparameters can be summarized as,

- **H**: 100
- **Optimizer**: Adam [16]
- **Loss function**: Loss function in Equation 3.4.
- **Activation**: Rectified Linear Unit (ReLU)
- **Model**: Fully Connected Neural Network with 3 hidden layers and 5000 hidden units each with and ReLU activation between each layer.
- **Batch Size**: 128
- **Epochs**: 25
- **Learning Rate**: 1e-4
- **Model Initialization**: Xavier Glorot Initialization [11]

After training the model on this training set, we get the learning curve in Figure 5.1. From the learning curve, we can draw the following conclusions

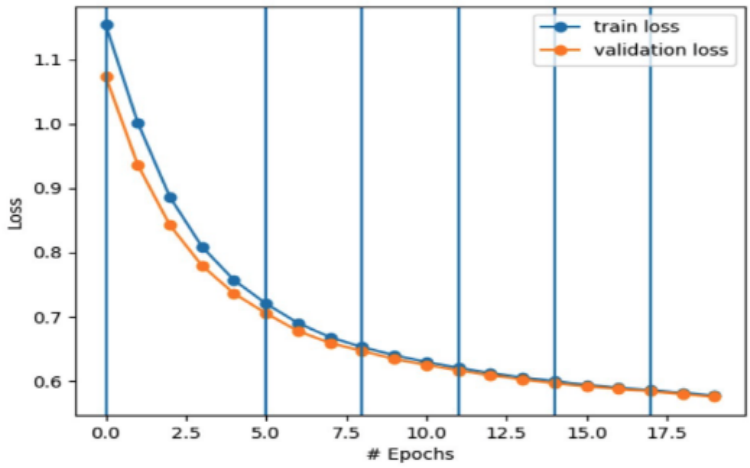

Figure 5.2: Learning Curve after training Fully Connected Neural Network based dynamics model for 20 epochs using teacher forcing training strategy.

- Due to the reduction in training and validation loss, we can conclude that a fully connected neural network is exhibiting learning behavior when it comes to capturing the dynamics of Cassie.
- Unfortunately, we notice that the mean squared error loss of test loss grows after every epoch. We suspect this is because the distributional shift introduced by the test set and the lack of generalization of the model. However, we do not observe this behavior with validation loss because it is a subset of the training dataset and, as a result, is from a similar distribution as the training set.

As a result of this performance, we believe it would be unreasonable to expect different test-set performances when predicting numerous steps because of the increased difficulty in predicting multiple steps. However, we can attempt to understand if a fully connected neural network can exhibit learning behavior when predicting multiple future states using teacher forcing. Before we get into the results, teacher forcing introduces additional hyperparameters.

- **Number of states in future prediction** (τ): 6
- **Teacher Forcing patience** (p): 4
- **Teacher forcing conditions**: At every epoch, we compute the slope between validation loss at epoch e and epoch $e - p$. If the slope is less than 0.05, we increase the number steps predicted.

The learning curve after applying teacher forcing strategy is given in Figure 5.2. We notice learning behavior for multiple-step prediction. However, this is not enough for the following reasons

- A mean squared error loss of 0.6 is very high for our prediction task. Given the problem with loss accumulation, we believe that a loss this high can cause problems during inference.

- The model lacks structure, i.e., it does not model the temporal information between the stacked state-action transitions.
- Training time and $H = 100$ are high for the results we obtained. Computational time increases with higher H.

Given these results, we intend to use a transformer model, which uses potentially lower H and is also able to effectively capture temporal relationships between state-action transitions.

5.1.2 Transformer Model

We now get into the experiments using a Transformer based dynamics model for Cassie. We will first get into the specifics of the model architecture, the number of parameters, the different hyperparameters used, teacher-forcing patience, and teacher-forcing conditions. In this experiment, we first used a fully connected neural network to project H state-action transitions to a lower dimensional space rather than using the "raw" state-action transitions. This is the representation learning layer of our pipeline. After this, they act as an input to a transformer model which predicts the future states of Cassie. The specifics of the representation learning layer include

- **Hidden Layers**: 2
- **Hidden Units per hidden layer**: 1000
- **Embedding dimension**: 30

The specifics of the transformer model are

- **Number of Encoders**: 4
- **Number of Decoders**: 4
- **Number of attention heads**: 8
- **Dimensionality of Transformer**: 256

The total parameters in both models are $12,919,393$. As mentioned in the earlier sections, we will use a mask during decoder-level self-attention to avoid self-attention between unseen components of the output sequence. The mask is an upper triangular matrix $M \in \mathbb{R}^{H \times H}$ with elements above the diagonal being a very small number $-\infty$ and the remaining elements in the matrix being 0.

As the training, test, and validation data, we will independently collect different state-action transitions for all three datasets using the pipeline in Figure 4.4. In the previous section (specifically Figure 5.1), we noticed using a subset of the training dataset as validation could result in incorrect inferences about the model's out-of-sample performance. In summary, we collected the following datasets

- **Training Set**: 90650 points with action noise distribution $\mathcal{N}(0, 0.06)$
- **Validation Set**: 41650 points transitions with action noise distribution $\mathcal{N}(0, 0.03)$.
- **Test Set**: 41650 points transitions with action noise distribution $\mathcal{N}(0, 0.01)$.

Lastly, we will discuss the numerous hyperparameters and other essential components used for training the models. We empirically determined that position or zeroth-order components were easier to predict than velocity or first-order components. Likewise, velocity or first-order components were easier to predict than acceleration or second-order components. Hence, we calibrated the weights in Equation 3.6 to reflect this.

Furthermore, as a result of the sensitivity of the model to different hyperparameters we had to perform a hyperparameter grid search [25] to decide optimal values for batch size, H, $L2$ regularization weight decay, dropout [41], and learning rate. The specifics of the grid search are discussed in Appendix A.1. We will now summarize hyperparameters and other essential components of model training

- **Optimizer**: AdamW [27]
  - **Learning Rate**: $1e-4$
  - $L2$ **regularization weight decay**: $1e-2$
- **Batch Size**: 128
- **Dropout**: 0.1
- **H**: 30
- **Learning Rate Scheduler**: Reduce Learning Rate on Plateau (ReduceLROnPlateau). After certain number of patience epochs, reduce the learning rate by a factor if validation loss stagnates.
  - **Scheduler Patience**: 75
  - **Scheduler Factor**: 0.9
- **Teacher Forcing**:
  - **# Future steps predicted** (τ): 10
  - **Teacher forcing patience (p)**: 75
  - **Teacher forcing conditions** (ψ): If percentage change in validation loss over the last p epochs is less than 5%
- **Loss function**: We will use loss function summarized in Equation 3.9.
  - **Gradient Penalty Scaling** (λ): $1e-2$
  - **w**: zeroth-order components have weight of 1.0, first order components have weight of 2.0, and second order components have weight of 5.0.
- **Activation**: ReLU
- **Model Initialization**: Xavier Glorot Initialization [11]

With these hyperparameters in mind, we will now discuss the results obtained. The vertical red lines in the plots in Figures 5.3, 5.4, and 5.5 represent an increment in steps. First, we will discuss Figures 5.3 and 5.4. Over 1000 epochs, the state-prediction error (Equation 3.6) between training, validation, and test sets follows a downward trend.

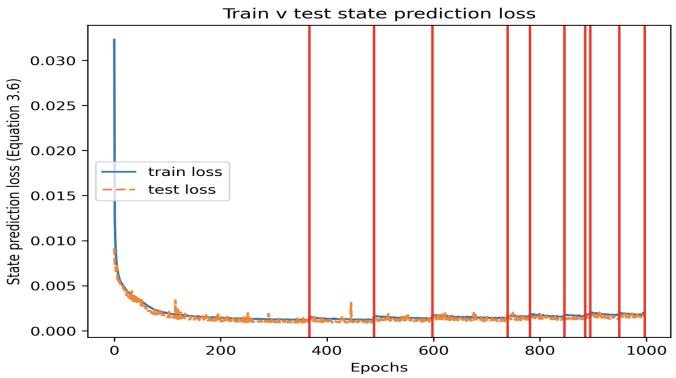

Figure 5.3: Train vs test state prediction error (Equation 3.6) over 1000 epochs using transformer model. The vertical red lines represent an increment in the number of steps predicted.

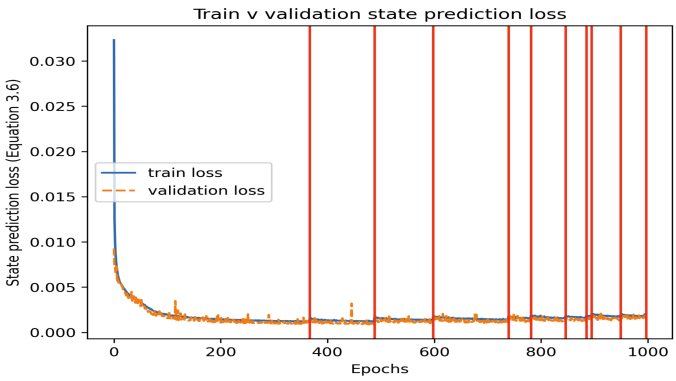

Figure 5.4: Train vs validation state prediction error (Equation 3.6) over 1000 epochs using transformer model. The vertical red lines represent an increment in the number of steps predicted.

Furthermore, after 1000 epochs, we achieve a state prediction training loss of 0.0018, state prediction validation loss of 0.0016, and state prediction test loss of 0.0013. We achieve the state-prediction errors mentioned above after predicting 10 states in the future. Based on the plot, the first increment takes the largest number of epochs. After which, the learning task is relatively simpler (it does not take as many epochs). Lastly, we have included a separate train and test state-prediction loss for each step in the future in Appendix A.2.

In the case of Figure 5.5, we are visualizing the loss function in Equation 3.9. In Figure 5.5, we notice that for 1000 epochs, the curve is on a downward trend. This is indicative of the Lipschitz constant and state prediction loss of the model decreasing as training continues. After 1000 epochs, we achieve a training loss (as per Equation 3.9) of 0.023. We have not plotted gradients for validation or test sets, as there are no gradients during inference.

From these results, we have shown that the transformer-based model can learn the underlying dynamics of Cassie.

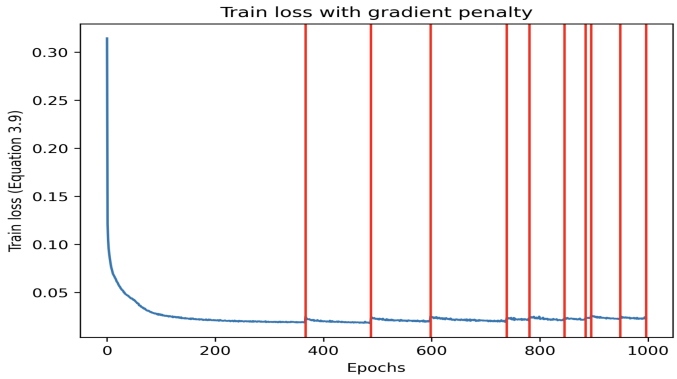

Figure 5.5: Train loss with gradient penalty as in Equation 3.9 over 1000 epochs using transformer model. The vertical red lines represent an increment in the number of steps predicted.

5.2 Planner

In this section of the experiments, we will discuss the results associated with the joint-level planner for Cassie. As mentioned earlier, the optimization process for the action sequences is done using the model trained in the previous section.

Here, we will look into the reward plots (reward from the simulator after executing action a_t at time t) over time for the planners. Through the reward plot, we can visualize the planner's performance (from reward values) and ability to sustain continued locomotion. First, we will discuss some initial results obtained using the sampling-based planner CEM. Through this, we hope to motivate the need for a gradient-based planner. Later, we will discuss the results associated with the Gradient-CEM planner mentioned in Algorithm 2. Lastly, we will go over the results from model retraining and how it helps improve the planner's performance.

5.2.1 CEM

In this experiment, we start Cassie at an initial state $s_0 \in \mathbb{R}^{49}$ in Mujoco. This initial state will be common across all experiments in this subsection. However, several other hyperparameters are affecting the performance of the CEM planner. Firstly, using the identity matrix as the initial covariance matrix could prove detrimental to the performance of our planner because the values in an action vector are in the range $[-0.5, 0.5]$. Hence, using a variance of 1 per dimension of the action vector will result in abnormally high values in the action samples. We will have to tune the diagonal of the covariance matrix for all 10 dimensions. Since CEM is a zeroth-order planner, the number of initial samples and elite samples will play an important role in the planner's performance.

After tuning the hyperparameters based on the cumulative reward over an episode, we finalized the following hyperparameters for our planner.

- **Number of steps in the future planned for** (τ): 10
- **Number of elites** (K): 50

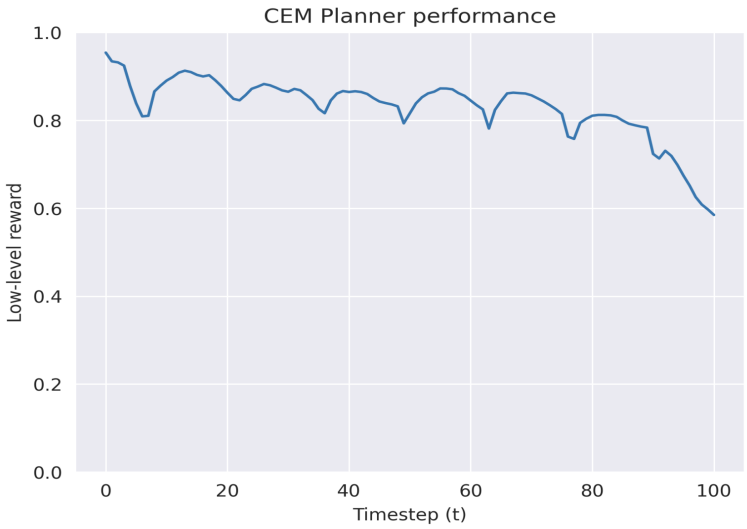

Figure 5.6: Reward obtained after executing optimal action from CEM planning starting from s_0 in Mujoco. The rewards are not logged once the robot falls.

- **Number of samples** (I): 1000
- **CEM Iterations** (C): 10
- $\text{diag}(\Sigma) = [0.01, 0.02, 0.01, 0.05, 0.01, 0.03, 0.02, 0.04, 0.01, 0.01]$

Upon using the CEM planner for these iterations, we received the reward curve in Figure 5.6. We stop recording the rewards once the robot falls because the goal is to sustain locomotion in Cassie without falling. From the plot, we notice that for the first 80 timesteps (equivalent to 10 seconds of walking in real-time), the CEM planner walks with a relatively high reward. After that point, there is a drastic reduction in the reward. Eventually, at $t = 101$, the robot falls. We believe this is because of the large dimensionality of the action. As noted in [3], a large dimensionality of the action space can require numerous samples and multiple CEM iterations. Unfortunately, this might not be possible with the computational resources at hand.

5.2.2 Gradient-CEM Planner

Given the failure modes of the zeroth-order planner, a gradient-based planner, to a certain extent, might be able to resolve those problems. Specifically, we used the Gradient-CEM planner described in Algorithm 2. As a start state, we used the same initial state s_0 as in previous experiments. However, using a gradient-based planner introduces multiple hyperparameters in addition to the ones used in the previous experiment (which will remain the same). In this experiment, we will be using the following set of hyperparameters

- **Number of Gradient Iterations** (G): 7
- **Optimizer**: Stochastic Gradient Descent (SGD)
  - **Learning rate**: $7e-2$

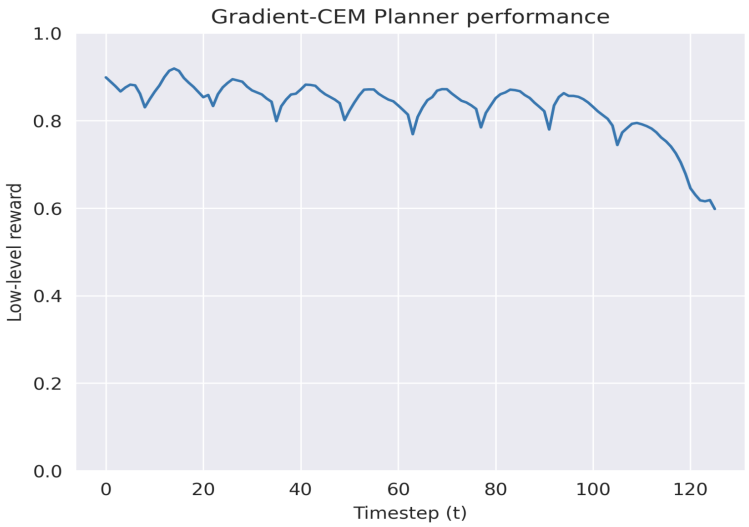

Figure 5.7: Reward obtained after executing optimal action from Gradient-CEM planning starting from s_0 in Mujoco. The rewards are not logged once the robot falls.

- **Momentum**: 0.8

Using these hyperparameters, we will now update the elite actions using the gradient of the negative reward 7 times every CEM iteration.

With these parameters, we let the Gradient-CEM planner execute optimal actions in a simulator. As a result, we obtain the reward curve in Figure 5.7. As in the previous experiment, the rewards are not logged after the robot falls. Unlike sampling-based planning, we notice an improvement in the number of high-reward steps executed by the planner and the number of timesteps before the robot falls (126). However, this is still not promising as the model inaccuracies accumulate, which ultimately causes the robot to fall.

To avoid this issue, we used model retraining via DAgger to effectively retrain the model and allow locomotion without falling for a longer time.

5.2.3 Gradient-CEM with DAgger

To retrain the model, we devised a variant of DAgger [37], which can be used in a model-based setting. The retraining method has been summarized in Algorithm 3. In this experiment, we will use DAgger in conjunction with Gradient-CEM, as this planner has empirically proven to be more promising than CEM. This introduces two additional hyperparameters, ϕ and M. Since we have access to the Mujoco simulator, we can determine ϕ through experimentation. As defined before, ϕ is length of action sequence for completing two steps in Mujoco. Also, M, is the number of state-action transitions we plan on aggregating at every iteration of model retraining. Furthermore, we will choose M such that it is on a similar scale as the dataset used in the initial model learning pipeline. The hyperparameters used for model retraining using DAgger are,

- ϕ: 27

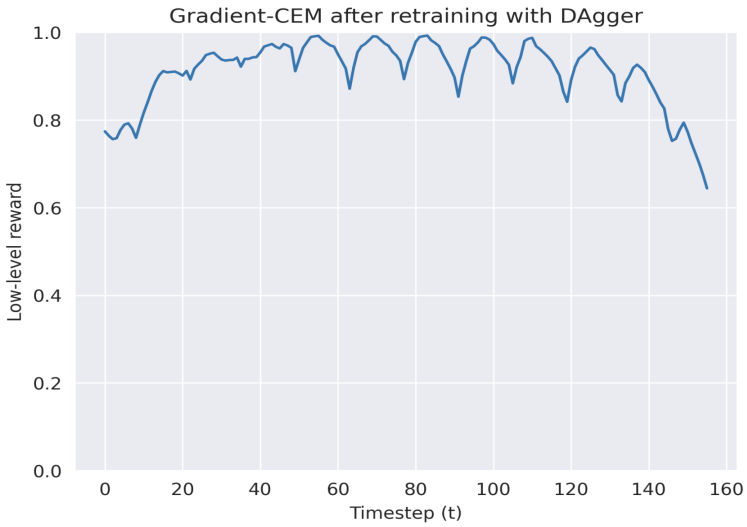

Figure 5.8: Reward obtained after executing optimal action from Graident-CEM planning starting from s_0 in Mujoco using model retrained twice with DAgger. The rewards are not logged once the robot falls.

- M: 100000

Once we obtain M data points, we will retrain the model by splitting the data into train test and validation sets. We use the validation step for teacher forcing and the test set for understanding our out-of-sample performance. They are divided in a $60-20-20$ ratio. Once the dataset is ready, we will retrain the model using similar training strategies. Given the computational resources, and the time involved in training a large model on such a large dataset, we have limited the model to just two retrain iterations. The model's performance after retraining was similar to the results obtained in the model learning section. We will now re-evaluate Gradient-CEM using the retrained model.

Starting from s_0 in Mujoco, we will now use Gradient-CEM as a controller with a model which has been retrained twice with DAgger. We obtain the reward curve in Figure 5.8. As in the previous experiment, the rewards are not logged after the robot falls. In the reward curve in Figure 5.8, there is a drastic improvement in the controller's performance. To that end, our agent can sustain walking for approximately 155 timesteps. Despite falling after that, we believe that upon retraining the model numerous times, we will be able to achieve significantly better performance through DAgger and Gradient-CEM.

Chapter 6

Conclusions and Future Work

In this chapter, first, we will critically examine the experimental results we obtained from model learning, joint-level controller, and the model retraining process. Then, we will discuss some methods we could potentially use to improve upon them.

6.1 Model Learning

We have effectively shown how an encoder-decoder-based transformer model can be used to approximate the dynamics of a bipedal robot. Furthermore, we used robust and stable training strategies like teacher forcing to help predict τ steps in the future. All our claims about the model's performance were supported by evidence in the form of learning curves during training.

Despite these results, we believe there are certain shortcomings in our approach which have to be addressed. These include,

- Our model does not use additional features when predicting the next state of the robot. Incorporating more robust physical features in the input to the transformer could result in more robust attention weights during self-attention and improve the model's accuracy.
- To better understand the model's prediction accuracy, we could visualize the model's state predictions in the Mujoco simulator.

Regarding using more features in the model, we could use contact forces and positions. A contact force is the magnitude of force applied on a surface along each dimension when the robot's foot is in contact with a rigid surface. Likewise, a contact position is the 3D coordinates of the bipedal robot's foot when it is in contact with the environment's surface. In Figure 6.1, when executing a robotic gait without falling, a bipedal robot's contact forces follow a discernible pattern. We believe that these patterns could be encoded into the state information like in [19] where the authors use a CNN-based autoencoder to learn an encoding of raw data like contact information which is only available in the simulation. Incorporating an encoding of contact information in the state-action vector could help improve the transformer's predictions.

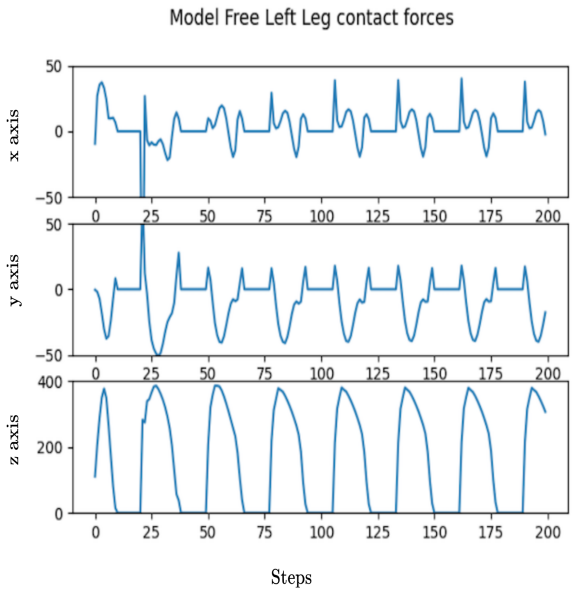

Figure 6.1: Contact forces along the 3 axes on left leg obtained from a robust model-free controller π^* which can walk over an extended period of time without falling

On the other hand, we did attempt to visualize predicted states in Mujoco. Unfortunately, however, we realized that this would involve significantly modifying the back end of Cassie's Mujoco simulator.

6.2 Joint-level Planner and Model Retraining

In the results section related to the planner, we observed how gradient-based planning combined with effective model retraining methods like DAgger could lead to significant improvements in the performance of a model-based controller compared to zeroth-order planners. While these results are encouraging, we are yet to see if we can use our model-based controller in the context of bipedal locomotion over an extended period. One of the main issues of our planner is the amount of time it takes to collect a sufficient amount of data and retrain the model using that data. While DAgger does help in improving the controller, we were able to retrain the model twice with the computational resources at hand. As the number of retraining attempts increases, we would retrain the model on a larger dataset, which we believe will take a long time.

We believe that we could potentially reduce the data used for training without affecting the model's performance. Currently, in Algorithm 3, we collect M transitions $n = 1$ to ϕ times and use all $\mathcal{D}_i$ datasets where $i < n$. Instead, as n increases, we could decay the data points we used from the earlier retraining iterations since we have already trained the model on them. For instance, if $n = 3$, we could use $|\mathcal{D}_3|$ (whole dataset) points from $\mathcal{D}_3$, $\beta \cdot |\mathcal{D}_2|$ points from $\mathcal{D}_2$, and $\beta^2 \cdot |\mathcal{D}_1|$ points from $\mathcal{D}_1$ where $\beta \in \mathbb{R}$ and $0 < \beta < 1$.

Another issue with our controller is the time it takes to determine the optimal action to execute given the current state s_t. Despite reducing the number of gradient computations

in Algorithm 2 compared to other first-order planners, using a deep model like a transformer could be a bottleneck when computing reward gradients. Unfortunately, using a controller which takes a long time to determine an optimal action can be problematic in an online setting. We could circumvent this by parallelizing the optimization process of the rewards by using batch optimization.

Lastly, another unexplored avenue in this thesis is whether we get similar results across different gaits. In our experiments, we have primarily explored walking in Cassie. However, we could experiment with other gaits like trotting or hopping in different bipedal robots.

Despite these shortcomings, our current approach does pave the path for effectively using model-based reinforcement learning methods to control bipedal robots.

Bibliography

[1] Karl Johan Åström. Optimal control of markov processes with incomplete state information. *Journal of mathematical analysis and applications*, 10(1):174–205, 1965.

[2] Richard E Bellman. *Dynamic programming*. Princeton university press, 2010.

[3] Homanga Bharadhwaj, Kevin Xie, and Florian Shkurti. Model-predictive control via cross-entropy and gradient-based optimization, 2020.

[4] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[5] Gianluca Bontempi, Souhaib Ben Taieb, and Yann-Aël Le Borgne. Machine learning strategies for time series forecasting. *Business Intelligence: Second European Summer School, eBISS 2012, Brussels, Belgium, July 15-21, 2012, Tutorial Lectures 2*, pages 62–77, 2013.

[6] Marc Peter Deisenroth and Carl Edward Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, page 465–472, Madison, WI, USA, 2011. Omnipress.

[7] Thomas G Dietterich. Machine learning for sequential data: A review. In *Structural, Syntactic, and Statistical Pattern Recognition: Joint IAPR International Workshops SSPR 2002 and SPR 2002 Windsor, Ontario, Canada, August 6–9, 2002 Proceedings*, pages 15–30. Springer, 2002.

[8] Helei Duan, Jeremy Dao, Kevin Green, Taylor Apgar, Alan Fern, and Jonathan Hurst. Learning task space actions for bipedal locomotion, 2020.

[9] William Fedus, Carles Gelada, Yoshua Bengio, Marc G. Bellemare, and Hugo Larochelle. Hyperbolic discounting and learning over multiple horizons, 2019.

[10] Zipeng Fu, Ashish Kumar, Jitendra Malik, and Deepak Pathak. Minimizing energy consumption leads to the emergence of gaits in legged robots, 2021.

[11] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[12] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of wasserstein gans, 2017.

[13] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[14] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, 1985.

[15] Zhong Ping Jiang, Tao Bian, and Weinan Gao. Learning-based control: A tutorial and some recent results. *Foundations and Trends in Systems and Control*, 8(3):176–284, December 2020. Publisher Copyright: © 2021 IGI Global. All rights reserved. Copyright: Copyright 2020 Elsevier B.V., All rights reserved.

[16] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[17] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, 2004.

[18] Scott Kuindersma, Robin Deits, Maurice Fallon, Andrés Valenzuela, Hongkai Dai, Frank Permenter, Twan Koolen, Pat Marion, and Russ Tedrake. Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot. *Autonomous Robots*, 40, 07 2015.

[19] Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. Rma: Rapid motor adaptation for legged robots, 2021.

[20] Alex Lamb, Anirudh Goyal, Ying Zhang, Saizheng Zhang, Aaron Courville, and Yoshua Bengio. Professor forcing: A new algorithm for training recurrent networks, 2016.

[21] Mikko Lauri, David Hsu, and Joni Pajarinen. Partially observable markov decision processes in robotics: A survey. *IEEE Transactions on Robotics*, pages 1–20, 2022.

[22] Zhongyu Li, Xuxin Cheng, Xue Bin Peng, Pieter Abbeel, Sergey Levine, Glen Berseth, and Koushil Sreenath. Reinforcement learning for robust parameterized locomotion control of bipedal robots, 2021.

[23] Zhongyu Li, Christine Cummings, and Koushil Sreenath. Animated cassie: A dynamic relatable robotic character, 2020.

[24] Zhongyu Li, Jun Zeng, Akshay Thirugnanam, and Koushil Sreenath. Bridging model-based safety and model-free reinforcement learning through system identification of low dimensional linear models, 2022.

[25] Petro Liashchynskyi and Pavlo Liashchynskyi. Grid search, random search, genetic algorithm: A big comparison for nas, 2019.

[26] Seong Chiun Lim and Gik Hong Yeap. The locomotion of bipedal walking robot with six degree of freedom. *Procedia Engineering*, 41:8–14, 2012. International Symposium on Robotics and Intelligent Sensors 2012 (IRIS 2012).

[27] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

[28] Shie Mannor, Reuven Rubinstein, and Yohai Gat. The cross entropy method for fast policy search. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, page 512–519. AAAI Press, 2003.

[29] Edgar Alonso Martinez-Garcia and José A Aguilera. Dynamic modelling and control of an underactuated quasi-omnidireccional hexapod. In *Handbook of Research on Advanced Mechatronic Systems and Intelligent Robotics*, pages 377–400. IGI Global, 2020.

[30] Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, page 7559–7566. IEEE Press, 2018.

[31] Silviu Pitis. Rethinking the discount factor in reinforcement learning: A decision theoretic approach, 2019.

[32] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[33] Marc H. Raibert and Ernest R. Tello. Legged robots that balance. *IEEE Expert*, 1(4):89–89, 1986.

[34] Jacob Reher, Wen-Loong Ma, and Aaron D. Ames. Dynamic walking with compliance on a cassie bipedal robot, 2019.

[35] Jenna Reher, Claudia Kann, and Aaron D Ames. An inverse dynamics approach to control Lyapunov functions. In *2020 American Control Conference (ACC)*, pages 2444–2451. IEEE, 2020.

[36] Agility Robotics. cassie mujoco sim. `https://github.com/osudrl/cassie-mujoco-sim`, 2018.

[37] Stephane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning, 2011.

[38] Kevin Scaman and Aladin Virmaux. Lipschitz regularity of deep neural networks: analysis and efficient estimation, 2019.

[39] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[40] Jonah Siekmann, Srikar Valluri, Jeremy Dao, Lorenzo Bermillo, Helei Duan, Alan Fern, and Jonathan Hurst. Learning memory-based control for human-scale bipedal locomotion, 2020.

[41] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[42] S.H. Strogatz. *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry and Engineering*. Studies in nonlinearity. Westview, 2000.

[43] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[45] E.R. Westervelt, J.W. Grizzle, and D.E. Koditschek. Hybrid zero dynamics of planar biped walkers. *IEEE Transactions on Automatic Control*, 48(1):42–56, 2003.

[46] Zhaoming Xie, Patrick Clary, Jeremy Dao, Pedro Morais, Jonathan Hurst, and Michiel van de Panne. Iterative reinforcement learning based design of dynamic locomotion skills for cassie, 2019.

[47] Xiaobin Xiong and Aaron Ames. Bipedal hopping: Reduced-order model embedding via optimization-based control, 2018.

[48] Xiangru Xu, Paulo Tabuada, Jessy W Grizzle, and Aaron D Ames. Robustness of control barrier functions for safety critical control. *IFAC-PapersOnLine*, 48(27):54 - 61, 2015.

[49] Yuxiang Yang, Ken Caluwaerts, Atil Iscen, Tingnan Zhang, Jie Tan, and Vikas Sindhwani. Data efficient reinforcement learning for legged robots, 2019.

[50] Yuxiang Yang, Tingnan Zhang, Erwin Coumans, Jie Tan, and Byron Boots. Fast and efficient locomotion via learned gait transitions, 2021.

Appendix A

In this appendix, we will go over two important results which we discussed briefly in the experimental results section. First, we will go over the grid search results we used to choose a subset of the hyperparameters for our model performance. Later, we will provide more detailed results and learning curves in the context of model learning using a transformer model.

A.1 Hyperparameter Grid search

As mentioned in the experimental results section, we tuned the following hyperparameters using a grid search

- Batch size during model training
- The length of the input sequence to the transformer (H)
- Dropout probability in both models
- Optimizer learning rate
- Optimizer weight decay

In a grid search, we choose a set a possible values for each hyperparameter we wish to tune and determine which of these values will lead to best performance of the model. We decided to perform grid search on only these hyperparameters because of the additional time taken in introducing new variables to the search space. Hence, in the interest of time, we decided to confine our search to a limited set of hyperparameters.

In order to empirically measure the quality of the chosen hyperparameters, we evaluated the performance of the model on a validation set. For our grid search run, we choose the following set of values

- Batch Size: [64, 128, 256]
- H: [30, 40, 50]
- Optimizer learning rate: [1e-3, 1e-4, 1e-5]
- Dropout probability: [0.1, 0.2, 0.3]
- Optimizer weight decay: [1e-3, 1e-2, 1e-1]

To efficiently perform the hyperparameter grid search, we made use of the weights and bases platform [4]. After performing grid search on these values we decided to use the following hyperparameter values to train our model

- Batch Size: 128
- H: 30
- Dropout probability: 0.1
- Optimizer learning rate: 1e-4
- Optimizer weight decay: 1e-2

A.2 Step-wise prediction

In this section of the appendix, we will be providing more verbose results for the multi-step prediction by the transformer model. As mentioned earlier, we have trained the model to predict numerous steps in the future with the help of the teacher forcing training strategy.

First, we will go over the step-wise performance in the training set. The results have been summarized in Figure A.1. If we are considering the line plot labelled Train step i loss, it is equivalent to the loss in equation 3.9 (with gradient penalty) except we will be predicting states until the ith step. As a part of our teacher forcing strategy, we cannot predict the next step until certain conditions are met. As a result, we note that $i = 1$ lineplot starts from epoch 0 because we initially predict only one step in the future. Likewise, we start predicting the second step after approximately 375 epochs. The loss increases as the number of steps increase because of the accumulation of errors from the previous steps.

We also notice after an increment in the number of predicted steps, other remaining plots experience a surge in the loss. Despite this, the model is able to effectively learn to predict multiple steps in the future in the training set.

We will now look at validation performance summarized in Figure A.2 with the loss function in Equation 3.6. The loss mean the same as they did in training scenario. However, in the validation loss the learning curve's trajectory is not the same as it was with the training loss. Despite that, the state prediction is as low as ≈ 0.01 for upto 10 steps in the future on an unseen dataset.

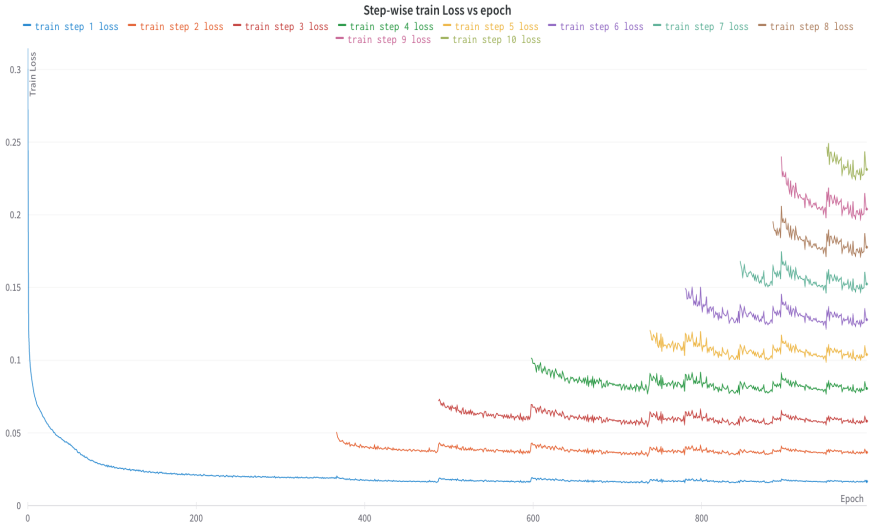

Figure A.1: Training set state prediction loss (Equation 3.9) when predicting multiple steps in the future using a transformer model. Each line plot represents the loss in predicting states upto the ith step.

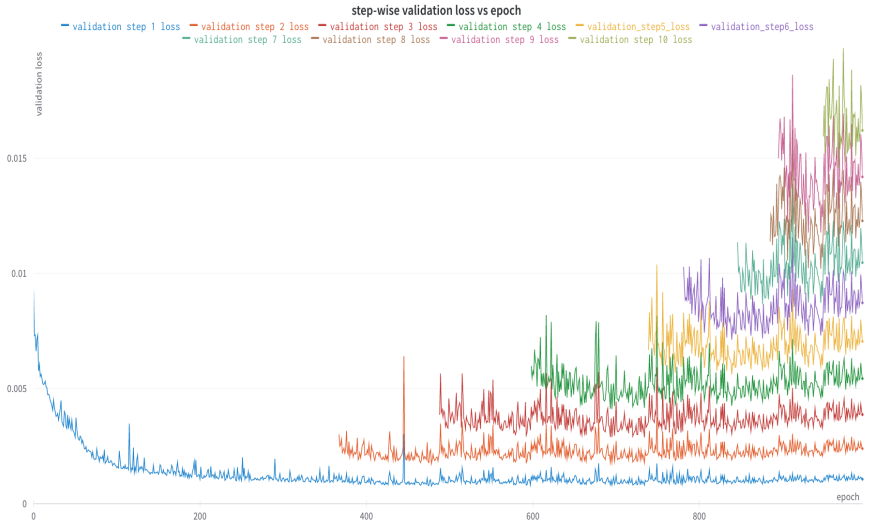

Figure A.2: Validation set state prediction loss (Equation 3.9) when predicting multiple steps in the future using a transformer model. Each line plot represents the loss in predicting states upto the ith step.

A.3 State Initialization

In our experiments, we refer s_0 as the initial state of our agent. This can be given by,

- Pelvis Height: 0.80524
- Pelvis Orientation (quaternion): [1.0, 0.0,0.0,0.0]
- Actuated Joint positions: [4.4792e-03, 0.0, 4.9730e-01,-1.1997,-1.596,-4.4792e-03,0.0, 4.9730e-01,-1.1997,-1.5968]
- Pelvis Translational Velocity: [-3.4741e-08, 9.9165e-15, 4.0239e-04]
- Pelvis Rotational Velocity: [0.0,0.0,0.0]
- Actuated Joint Velocities: [0.0, 0.0,0.0,0.0,0.0, 0.0,0.0,0.0,0.0, 0.0]
- Pelvis Translational Acceleration: [-7.8386e-02, 1.9637e-05, -9.8094e+00]
- Unactuatoed Joint Positions: [0.0, 1.4267e+00, -1.5961e+00, 0.0, 1.4267e+00, -1.5961e+00]
- Unactuated Joint Velocities: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
- Simulator Level velocity: 5.0900e-02
- Clock Phase: [-1.1609e-01, -9.9324e-01]