university of groningen

faculty of science and engineering

Combining Model-Based and Model-Free approaches in achieving sample efficiency in Reinforcement Learning

Anjali Nair

November 18, 2022

university of groningen / faculty of science and engineering

University of Groningen

Combining Model-Based and Model-Free approaches in achieving sample efficiency in Reinforcement Learning

Master's Thesis

To fulfill the requirements for the degree of
Master of Science in Artificial Intelligence
at University of Groningen under the supervision of
Prof. dr. Raffaella Carloni (Artificial Intelligence, University of Groningen)
and
Dr. Matthia Sabatelli (Artificial Intelligence, University of Groningen)

Anjali Nair (s4234790)

November 18, 2022

Contents

Acknowledgments

I would like to thank Prof. Dr. Raffaella Carloni and Dr Sabatelli for their support and expert insights during the course of this project. Their supervision provided direction and clarity in understanding the intricacies and challenges of the problem.

I would also like to thank my friends and family for the constant moral support through this journey.

Abstract

Reinforcement Learning is broadly classified into model-free (MF) and model-based (MB) approaches. While MF approaches have repeatedly proved successful in solving a variety of robotic applications, the training is often accompanied with the need of large number of learning samples [1]. In absence of a simulation, sampling from the real environment can be expensive and lead to hardware wear and tear. Model-based (MB) reinforcement learning approaches on the other hand, plan trajectories in a learned model and execute only a subset of the transitions in the real environment. However, the reliance on the learned model and inherent modelling errors cause model-based approaches to struggle in achieving performance comparable to MF.

In an attempt to get the best of both worlds, we propose to combine the two approaches, MB and MF, with a novel architecture. The MB counterpart of the architecture involves learning an approximate model of the real environment and planning trajectories by means of a modified Model Predictive Controller (MPC). Here, planning refers to rolling out trajectories in the learned environment without the agent making these trajectories in the real environment. While a traditional MPC plans trajectories by random action selection at every timestep, we propose to have an in-loop policy, trained through a MF approach in directing the actions. The samples collected through this planning are used to further train the policy for the agent. The policy attained through this approach is then fed as the initial policy to warm start pure MF training. The MF training here serves the purpose of fine-tuning our policy to combat incorrect planning due to model-errors in the learned environment.

While MB and MF approaches have been combined in the past, the main contribution of the proposed architecture is in combining a traditional planner such as MPC and an MF policy, specifically in the planning stage. A fully connected feed-forward neural network is used in learning the environment. We choose Proximal Policy Optimisation (PPO) as the model-free algorithm, simply due to its relevance and popularity in robotics.

In evaluating our architecture, we test it on the Half-Cheetah Mujoco environment, where the task is to make the half-cheetah run forward. As in reinforcement learning, comparisons are made based on the rewards attained in each case. We compare the performance of planning with MPC and a MF policy as opposed to planning with only MPC. To test the superiority of the hybrid MB and MF architecture (MBMF) to its pure MF counterpart (PPO), we compare the rewards obtained in each case. Further, we test the sensitivity of our architecture to different rewards. We also modify our architecture to replace PPO, an on-policy algorithm, with Soft Actor-Critic (SAC), in testing the applicability to an off-policy algorithm. Off-policy algorithms are model-free approaches where trajectories following old policies are also used in updating the current policy.

We find that planning with MPC and PPO together achieves higher scores than planning with MPC alone in all scenarios tested (different rewards and MF algorithms). We also find, when trained with the default reward, our architecture achieves scores that PPO does in 1e6 timesteps, but with 5e5 fewer timesteps. However, our architecture proves to be sensitive to the rewards. Further, since scores do not justify the quality of the defined rewards, we analyse the gaits achieved for each reward based on the torques applied at each joint and the stability of the centre of mass of the half cheetah. We also note that our architecture works just as well with SAC as it does with PPO, showing its applicability to on-policy and off-policy MF algorithms.

1 Introduction

Up until the 90's, robotics and Artificial Intelligence (AI) had a clear distinction. While robotics was concerned with machine automation, AI focused on building intelligent systems [2]. Since then, however, the two fields have frequently crossed paths and today, application of AI in robotics is a highly researched field. In particular, the sub-field of reinforcement learning (RL) in AI proves to be a good fit in solving complex locomotion tasks [3][4]. RL algorithms have shown success in bipedal [5], multi-legged [6] and humanoid [7] locomotion tasks. In spite of the suitability of these methods, there are evident drawbacks too. Among the various downsides of reinforcement learning [8], such as the exploration problem, training stability and low sample efficiency, in this research, we focus on dealing with the latter.

While deep reinforcement learning (DRL) has the ability to learn highly complex tasks, this comes at the expense of the robot, more generally referred to as the agent, taking a great deal of wrong actions until it learns the good ones. Sample complexity in DRL refers to the number of samples from the real environment required to train a policy in successfully learning a behaviour. A method that requires a large number of real-world samples to learn is said to be sample inefficient while that which uses fewer real world samples is considered a sample efficient method. When training in the real world, collecting large number of samples requires the robot or the agent to be operated for long periods while it performs various actions. Long periods of operation maybe associated with high power consumption costs. Moreover, the actions taken by the robot or agent during training may lead it to unsafe regions, requiring manual intervention or resulting in hardware wear and tear. This makes the application of many DRL methods in the real-world a subject of concern. In presence of a simulation of the real environment, these drawbacks can be mitigated to certain extent, but the development of the simulation itself becomes a new challenge. Moreover, simulations very rarely capture the complexity of the real world and a robotic controller developed in a simulation may very well fail in the real world. This is referred to as the sim-to-real gap and is a subject of much research today [9][10]. In this research, we take inspiration from DRL methods that first learn the environment through machine learning approaches and use this learned environment as a substitute to sample from. Samples drawn from a substitute environment rather than the real environment do not count towards sample complexity, as these samples do not pose the risks and drawbacks discussed for sampling from real world.

Model-free reinforcement learning (MF) methods such as PPO [11] and TRPO [12] are the popular choice of architecture in solving locomotion tasks with continuous action space. Considering the complexity and the potentially large search space, such MF algorithms allow a simple implementation while guaranteeing convergence to a policy. However, these models face the disadvantage of very poor sample efficiency owing to the need of drawing large number of samples from the real environment. This makes MF methods unsuitable in many real world robotic applications. On the other hand, model-based (MB) methods attain sample efficiency by planning on a learned environment and taking only the relatively better or safer actions in the real environment. Thus, with MB methods, the risk of MF methods leading to robot damage and high energy consumption can be circumvented. The success of MB approaches relies heavily on the quality of the learned model and the planning strategy used. Pure MB approaches, due to these bottlenecks, often fall short of the performance MF approaches could easily provide with similar tasks.

Inspired by the implementations combining MF and MB approaches [13][14][15], we propose a new

hybrid MB and MF architecture to alleviate the problem of low sample efficiency in learning an optimum policy using solely an MF approach, while still maintaining a performance equivalent to these MF methods. Considering the limited development in MB approaches when compared to its MF counterpart [16], we also hope to provide support and reasoning to further boost research in this direction. Our method is tested on the Half-Cheetah, part of Mujoco [17] environments.

1.1 Research Questions

The challenge of sample complexity is concerned with requiring a large number of transitions to be collected from the real environment. This poses a problem as with MF approaches, these transitions are initially made on a trial and error basis. The problem of sample complexity, thus proves to be a major hindrance in the application of DRL to robotics. Our approach proposes to achieve sample efficiency by planning transitions on a learned environment before taking them in the real environment.

To achieve sample efficiency while still attaining a promising performance, we propose to combine MB and MF approaches of RL. The novelty of this work is in the proposed architecture where we use a PPO policy in combination with a traditional planner (Model predictive Controller (MPC) [18]) to plan over the learned environment. To evaluate the architecture, we compare it against using a vanilla PPO approach and a simplification of the architecture where only a planner is used in the planning stage. The architecture is tested on Half-Cheetah [19], part of the Mujoco environments. We further compare the architecture when switching PPO with Soft actor-critic (SAC), an off-policy model-free approach and analyse the results. We also experiment with reward functions for the Half-Cheetah environment and analyse the obtained gait. The experimentation with different rewards helps in testing our architecture's robustness or drawbacks. To summarize, this thesis focuses on the following problems:

Q1. Can our proposed MBMF architecture solve the RL task with fewer training data (higher sample efficiency) than its model-free counterpart?

Q2. Does planning with an in-loop trained policy directing the actions for MPC give superior performance (higher rewards) to planning with a naive MPC?

Q3. How robust is our architecture in handling various rewards and how do these rewards compare against each other?

Q4. Does the proposed architecture solve the RL task with fewer training data even when replacing PPO with an off-policy MF algorithm?

1.2 Thesis Outline

This paper is divided into various sections. In chapter 2, we provide an introduction to reinforcement learning, explaining the common terminologies and approaches followed. We also provide details regarding the specific algorithms and components involved in our architecture, including the environment.

Chapter 3 gives a brief summary on the work conducted with MB and MF approaches in the past. We discuss how specific methods combine MB and MF approaches, how this differs from our approach

and the inspirations we draw from.

Chapter 4 gives details on the process of data collection for our MB approach and discusses how and why we pre-process this data. Once we have set the baselines for the problem at hand, with chapter 5, we discuss the architecture proposed and how the various components of our architecture ties up together. We further discuss the various experiments conducted in evaluating the architecture. This section is followed up by chapter 6, giving details about the tools used in developing the project and the set hyper-parameter values. We also discuss the evaluation criteria used for the various experiments.

Finally, in chapter 7 we document all obtained results and interpret them.

2 Theoretical Framework

In this section, we give an introduction to reinforcement learning and the various terminologies accompanying it. All prior knowledge required to understand our methodology is covered in this section.

2.1 Reinforcement Learning

Reinforcement learning is an approach to learning from mistakes. In the world of AI and Machine Learning (ML), RL can be defined as the process of an agent learning a behaviour based on its interaction with the environment. Formally, an RL problem is defined as a Markov Decision Process (MDP). MDP is a decision process where the decision at timestep $t+1$ is only dependent on the decision at timestep t and independent of the past. To further delve into RL and its formulation as an MDP, one must be exposed to the basic elements of RL [20]. The basic structure of an RL problem is as shown in figure 1, where,

- *Agent* is the learner or the decision making entity. When solving an RL problem, it is the agent whose behaviour we wish to optimise.
- Everything outside the agent, i.e the world in which the agent behaves is referred to as the *environment*. The agent interacts with the environment at every timestep t, by taking an action a_t that transitions the agent from its current state, s_t to the next state s_{t+1}.
- *Transition function* defines the probability of the agent transitioning from one state to another when taking a particular action and is defined as:

$$T(s_{t+1}|s_t, a_t) = p(s_{s+1}|s_t, a_t) \tag{1}$$

  The transition function is in essence the dynamics of the environment.

- *Reward* is a signal that informs the agent if a particular action taken from a state is favourable, given the current task to be performed. The reward is predefined and is part of the environment. This entails that the reward cannot be changed by the agent. The agent therefore must change its behaviour to collect better rewards. The reward received at timestep t is denoted as r_t. Just as the transitions are probabilistic, the rewards too are sampled from a probabilistic function defined as:

$$R(r_t|s_t, a_t) = p(r_t|s_t, a_t) \tag{2}$$

  In short, when the transition function T and reward function R of an MDP are known, the probability of a state and reward pair (s_{t+1}, r_t), given a state and action pair (s_t, a_t) is written as $p(s_{t+1}, r_t|_t, a_t)$. This formulation completely defines the environment.

- *Policy* defines the behaviour of the agent in the environment. It maps a state in the environment to an action the agent takes when in that state. It is denoted as π_θ, where θ

is the policy parameter. In a general case, a policy is probabilistic in nature, i.e a policy maps a state to a probabilistic distribution of actions to be taken from that state. It is defined as:

$$\pi_\theta = p(a_t|s_t) \tag{3}$$

Therefore, if an agent following policy π_θ, is in state s_t, the probability that the agent will take action a_t would be $\pi_\theta(a_t|s_t)$. The policy is essentially the controller that we desire to learn.

- A series of transitions defined by (s_t, a_t, r_t, s_{t+1}) is called a *trajectory*, denoted by τ.

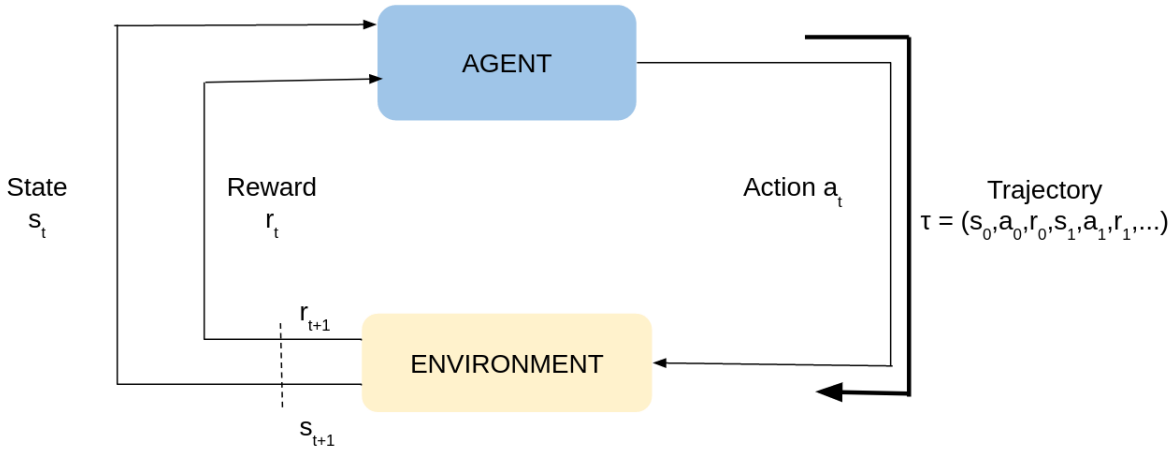


Figure 1: Reinforcement Learning Structure

The sum of rewards collected over a trajectory is called the *return*. As the trajectory length increases so would the return. For a trajectory of infinite length, this definition of return would lead to an infinite sum. Thus, the rewards gained at every timestep are weighted such that, immediate rewards are prioritized over those in the future. The weights are defined by the *discount factor*, γ. With this consideration, the discounted return in RL is defined as:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} r_{t+k+1}, \quad \textit{where } 0 \leq \gamma \leq 1 \tag{4}$$

Thus, the goal of an RL problem is to maximize this discounted return rather than the cumulative return. While the reward gives immediate feedback, the next element of RL, *value function* $V_\pi(s)$ gives an estimate of how good it is for the agent to be in state s. Value function maybe defined with respect to the state (state-value function) or a state-action pair (action-value function).

State-value function, written as $v_\pi(s_t)$, gives the expected return of being in state s_t, given that the agent follows a policy π. In other words, while reward tells the agent if taking an action a_t is good at

time step t, the state-value function gives an estimate of how advantageous it is for the agent to be in state s_t, given that it follows a policy π in order to achieve the final goal. The state-value function is mathematically defined as:

$$v_\pi(s_t) = \mathbb{E}_\pi[G_t|s_t] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t] \tag{5}$$

where $\mathbb{E}_\pi[.]$ denotes the expected value of a random variable where the agent follows policy π.

Action-value function gives an estimated value for a state action pair as opposed to the state-value function, which gives an estimate for a state alone. The action-value function is defined as

$$q_\pi(s_t,\ a_t) = \mathbb{E}_\pi[G_t|s_t,\ a_t] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t,\ a_t] \tag{6}$$

To find an optimal policy implies to find the path with highest returns from a state and is denoted as π^*_θ. An RL problem may have multiple optimal policies. However, all optimal policies share a common value-function. This is the optimal value-function that yields highest value compared to all other value-functions. The optimal value-function is defined as:

$$V^*(s_t) = max_\pi\ V_\pi(s_t) \tag{7}$$

While the state-value and action-value functions are described as expectations in equation 5 and 6, they are recursive in nature. Thus, formulating the state-value function with the MDP components, we get the central equation of RL -the Bellman Equation:

$$v_\pi(s_t) = \sum_{a_t} \pi(a_t|s_t).\sum_{s_{t+1},r_t} p(s_{t+1}, r_t|s_t, a_t)(r_t + \gamma v_\pi(s_{t+1})), \tag{8}$$

As seen, this central equation of reinforcement learning breaks down the value function defined in equation 5 into two parts - the immediate reward and discounted value function or the expected reward for the next state. The inner sum is further multiplied by the probability of transitioning between the two states with the given reward. This probability terms encapsulates the transition and reward functions. Finally, the product obtained so far is multiplied by the probability of taking an action from the current state. This probability as stated previously is the policy π. Summing over all possible actions from a state, we get the value for the value-state function from state s_t.

Model is the final element of RL that may or may not be available. The model is a formulation of the real environment, including the transition and reward functions. The model encapsulates the behaviour of the real environment. Based on the availability of a model, RL approaches are broadly classified into model-based and model-free learning [21].

2.1.1 Model-free Reinforcement Learning

Model-free (MF) RL approaches are followed in the absence of a model. This entails that the agent has no information regarding the environment (transition probability and reward functions) and must learn largely through trial and error. Thus, in MF methods, the agent must take an action in blind and learn the quality of actions as they are experienced. With model-free approaches, the agent tries to learn a policy that maximizes the returns over finite number of timesteps taken in the environment.

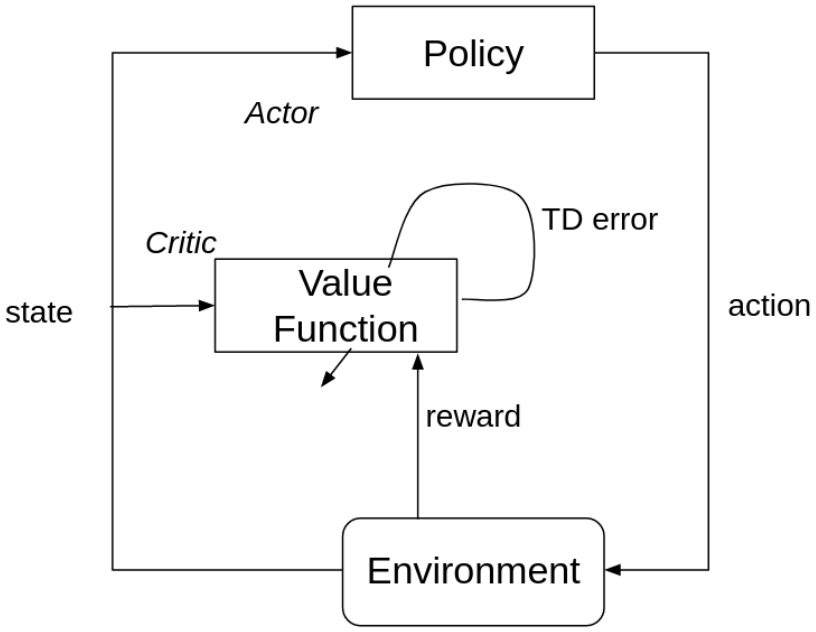


Figure 2: Actor-Critic Architecture

The goal of MF learning is to optimise the value functions (state-value or action-value) or in other words, to find the policy that maximizes the value functions for every state.

Some common MF approaches include Q-learning [22], policy-gradient [23] and actor-critic [24]. In this thesis, we work with actor-critic category of MF algorithms. Structurally, actor-critic approach is composed of two blocks as shown in figure 2 - the actor, which represents the policy and the critic, which critiques the actions taken by the policy. In DRL, the two blocks are traditionally neural networks. The policy takes the agent's state as input and produces an action. The critic network models the value function and critics the actions taken by the policy. Thus, the policy is run to collect samples (s_t, a_t) while the critic which models the value function for the current policy, gives the value for the state s_t. Using these values, the policy is updated by gradient descent. Depending on the formation of the policy gradient, actor-critic methods could be of various types. In this section, we discuss the standard actor-critic approach - TD actor-critic as it forms the basis. TD stands for temporal difference and is defined as:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t), \tag{9}$$

where V is the value function modeled by the critic and s_{t+1} is the state agent arrives on taking action a_t from state s_t. The TD error evaluates the quality of action a_t from state s_t. Intuitively, if δ_t is positive, it signifies that the probability of selecting a_t from s_t, i.e $p(s_t, a_t)$ must be increased while a negative δ_t indicates a reduction in $p(s_t, a_t)$. Formally, this change in probabilities is the updation of the policy network. The policy network learns through the classical policy-gradient method [25]. The gradient for the policy network is calculated as:

$$\nabla J(\theta) \approx \sum_{t=0}^{T-1} \nabla_\theta log_{\pi_\theta}(a_t, s_t)\delta_t \tag{10}$$

Model-free approaches are named such, as the policy is constructed without having an insight into the environment dynamics. This entails that during the initial stages of training, model-free approaches require the agent to make blind trajectories in the environment. This also means, that these approaches require the agent to take many trajectories in order to find an optimal policy, thus making them sample inefficient.

2.1.2 Model-based Reinforcement Learning

Model-based (MB) methods are set of methods that use a model and planning in solving the RL problem. Unlike with MF methods, MB methods do not learn through plain trial-and-error. A model is used to predict how the environment behaves on taking an action without the agent requiring to take the action in the real environment. This behaviour defines the transition and the associated reward. Therefore, given a state and action, the model produces the next state and reward. If the model is stochastic, it will instead produce a distribution of states and rewards. Models that output a distribution are called distribution models while those that produce a definite state and reward are called sample models. The advantage of having such a model is to predict future experience and learn from it without the agent actually experiencing it. To gather experiences, a planning algorithm is employed.

While the word planning is used with varying meanings in different fields, in the field of RL, planning refers to using a model of the environment to learn a policy. An algorithm that performs planning is called a planner. Typically, the planner uses a strategy to produce multiple trajectories. The planner generates the actions to be taken from a state and the model predicts the transition state and reward on taking the given state and action. A series of such transitions makeup a trajectory. These trajectories are evaluated to quantitatively define their quality and these evaluations are then used to update the policy accordingly. The quality of a trajectory is defined by the discounted sum of rewards of the transitions or an expectation of this value. The strategy used to recommend the actions varies among the planners used. The policy itself is not updated by the planner, which is only responsible for generating the trajectories. This is called planning over state-space as the planning strategy explores various states in the environment. In brief, the difference between planning and learning through an MF approach is that the planner simulates experiences, while MF approaches require the agent to encounter the experiences.

The model-based approaches focus on learning the transition dynamics and reward function of the environment and planning over this learned model. A model is mathematically defined as:

$$f_\psi(s_t, a_t) = p(r_t, s_{t+1}|s_t, a_t), \tag{11}$$

where ψ parameterise the weights of the model f. In most cases this model is an approximation of the real world. In creating the model, real world transitions are collected, generally using a random policy to form a dataset. This dataset is then used to train, in our case, a neural network such that, given a state and action, the network produces the next state or a distribution of next states and the reward or distribution of rewards associated with the transition. Thus, the neural network learns the environment behaviour from labelled data collected from the environment. This is parallel to supervised learning.

The model and planner together only simulate experiences. These experiences may then be used to learn the policy through maintaining a tabular record of actions, a process of updating the value functions as with MF approaches or through other means [26][27]. While at first glimpse, model-free methods seem more straightforward and applicable, model-based approaches have been found to have their benefits. The most obvious advantage of model-based methods is its sample efficiency. Samples in this case are drawn from the real environment, only to construct the model. The controller or policy, which requires a large number of samples for updating can be drawn from the learned model. In robotic applications, if the controller must be trained directly on the real system, we risk damages to the agent's physical body or surroundings during action exploration and succumb to high energy consumption. In such situations, compressesing the environment dynamics into a function and allowing planning on this function, can alleviate the burdens of a real environment.

2.1.3 Model Predictive Controller

Model Predictive controller (MPC) [18] is a method of process control while keeping in accordance with specified constraints. In the realm of model-based RL, MPC is one of the most popular planners used [14][28][29]. The structure of a traditional MPC is as shown in figure 3. One of the major boons of creating a model mimicking the real environment is the possibility to plan in future. This entails that one can use a planner to imagine the future without the agent having to make a move in the real environment. Under the assumption, that the learned model is a good approximation of the real world, this planning would require the agent to take only the relatively good transitions and learn from those. This future in MPC is planned with the goal to optimize returns at each timestep. From the start state, MPC plans m action sequences, each of length h. The value of h is usually kept small unlike with standard model-free techniques. Each of the planned trajectories are completely independent and the actions chosen at every timestep is random. For each of the m action sequences, a value indicating its worth for the task is calculated. For RL tasks, an action sequence is considered superior if it has a high score or low penalty. However, MPC only cares about the first action of the best action sequence from the current state, referred to as the elite action. Only this elite action is performed by the agent in the real environment. Thus, the agent is able to take the most optimal action from m possible actions without any backtrack or trial and error runs in the real environment. The state reached on performing this action is the new start state for MPC in the next iteration. In this manner, MPC optimizes the action taken at each timestep, by performing short model roll-outs.

MPC is only a controller scheme and various modifications of it are in use. The variation lies in the strategy used to select the actions for the m action sequences. In the most simple case, a naive random-shooting strategy is employed [30][31]. While random-shooting has shown applicability in non-linear dynamics, with a large search space, its relevance decreases. We need schemes which apply some constraint to the search space. Various works also use Cross-Entropy Method (CEM) [32] to restrict the distribution from which the actions are sampled. In general, the effectiveness of MPC can be highly sensitive to the choice of the action selection scheme.

2.1.4 Proximal Policy Optimization

As mentioned previously, MF approaches are widely used in the field of reinforcement learning. Its utilisation has led to a consequential rise in research and improvements in this sector. Proximal Policy Optimization (PPO) [11], a model-free actor-critic algorithm was released in 2017 by OpenAI [33]. PPO is an on-policy training algorithm. This entails that in contrast to maintaining a replay buffer

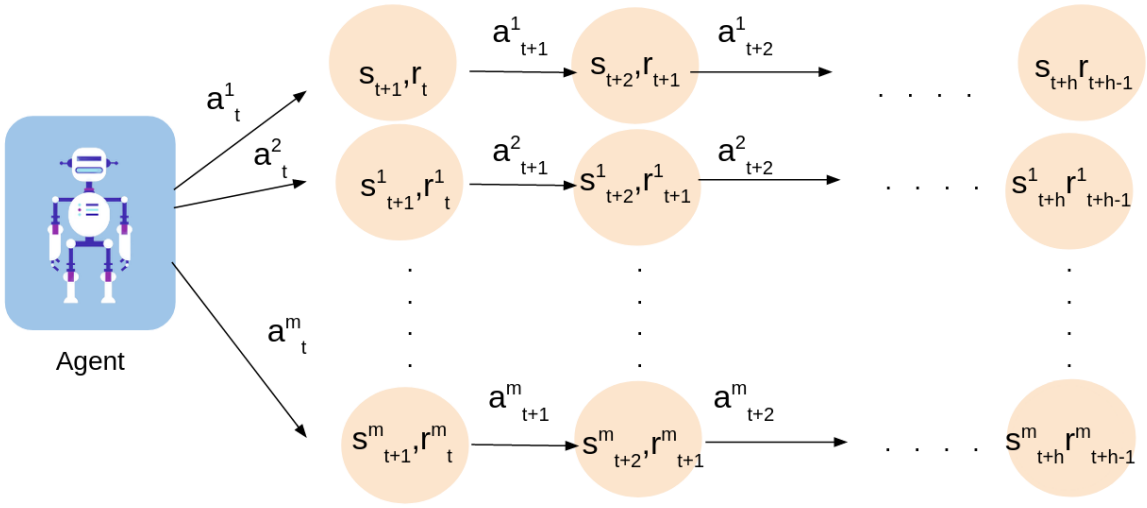


Figure 3: Model Predictive Controller Architecture

[34] of transitions, PPO discards all collected transitions after every policy update iteration.

$$\theta = \theta - \beta \nabla J(\theta) \tag{12}$$

A general equation for policy gradient updates is as shown in equation 12, where β is the stepsize for update, θ is the neural network parameter and J is the objective function. Actor-critic algorithms have been plagues by the issue of finding an optimal stepsize. Too small a value can result in very long training periods while a very large stepsize could lead the policy into regions of poor performance. PPO was developed with the motivation to alleviate this problem while maintaining a first derivative gradient optimisation.

In achieving stable updates, PPO maintains two policy networks - a current policy network and an old policy network, parameterised by θ and θ_{old} respectively. It further defines a policy ratio $r(\theta)$ as:

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} \tag{13}$$

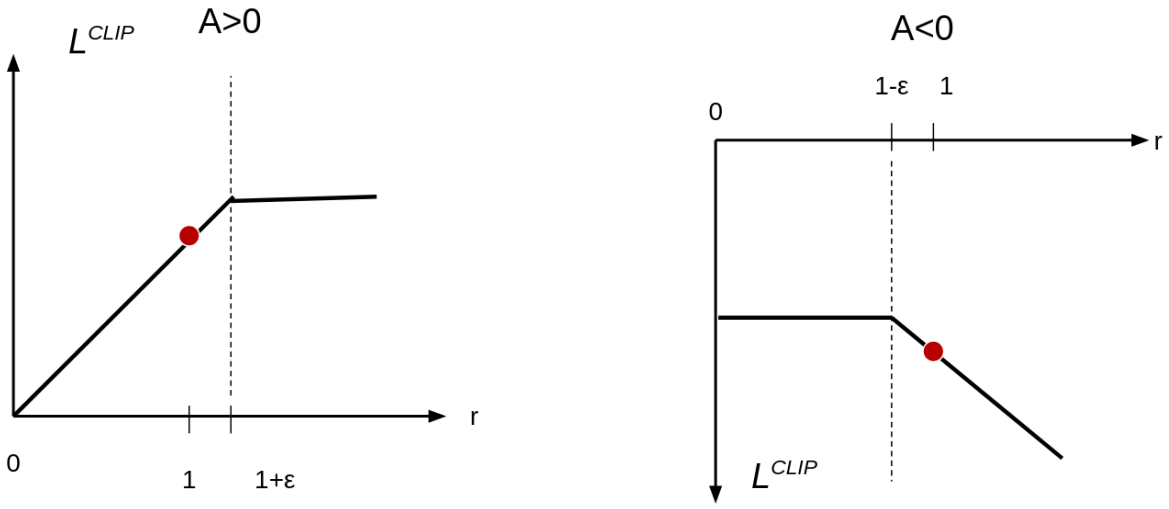
This fraction determines how much the new policy deviates from the old policy and is used in controlling the degree of policy updation. Another important factor for PPO is the advantage function. Just as the critic evaluates the policy's action by calculating the TD error in TD actor-critic, with PPO the evaluation is done through an advantage function $A(s_t, a_t)$. Thus, PPO is an advantage actor-critic approach. The advantage function is defined as the difference between the action-value for a state-action pair and the state-value for the state. Intuitively, The advantage function informs the policy how much better it is to take action a_t from state s_t as compared to the average of all actions possible from state s_t. However, our critic only model the state-value function and not the action-value. Thus, PPO uses an estimate for the advantage function defined as:

$$\hat{A}(s_t, a_t) = r_t + \gamma V(s_{t+1}) - V(s_t), \tag{14}$$

Finally, the objective function for the PPO policy network is defined as:

$$J^{CLIP}(\theta) = \mathbb{E}[\min(r_t(\theta)\hat{A}_{\theta_{old}}(s_t,a_t),\ \text{clip}(r_t(\theta),1-\varepsilon,1+\varepsilon)\hat{A}(s_t,a_t))], \tag{15}$$

where ε is a PPO hyper-parameter. The $clip()$ function further adds to the stability of PPO by ensuring that if the policy ratio falls beyond $1+\varepsilon$ and $1-\varepsilon$, it is clipped to within this range. We further explain the purpose of the clip function through the figure 4. Plot 4a shows the scenario where the A, the advantage is positive. This indicates that $\pi_\theta > \pi_{\theta_{old}}$ and thus, would lead to $r > 1$ where r is the reward. If π_θ is much larger than $\pi_{\theta_{old}}$, the computed value for r maybe larger than $1+\varepsilon$, in which case, the value of r is clipped to $1+\varepsilon$. Similarly, in plot 4b, A is negative, indicating that $\pi_\theta > \pi_{\theta_{old}}$ or $r < 1$. However, if π_θ is much smaller than $\pi_{\theta_{old}}$, the computed value for r risks being smaller than $1-\varepsilon$, in which case, the value of r is clipped to $1-\varepsilon$. The range between $1+\varepsilon$ and $1-\varepsilon$ is called the comfort zone.



(a) Clip function for $A > 0$

(b) Clip function for $A < 0$

Figure 4: Plots depicting the clip function [11]

While TRPO [12] also restricts the update step, the formulation of the cost function makes PPO much more efficient and is hence popularly used with continuous control tasks. Although, PPO shows great promise in even the most complex tasks, being an on-policy algorithm, demands a large number of samples for training. However, with our architecture, we aim to combat this drawback without compromising on its performance.

2.1.5 Environment

All experiments have been conducted on the Half-Cheetah-v3 environment, part of Mujoco. The half-cheetah agent has 6 actuated joints, represented by the grey circles in figure 5. The numbers label each end of the line segments that makeup the half-cheetah's body. The length of each segment is as shown in table 1. The observation space is 17 dimensional, including measures of joint positions and velocities. The 6 dimensional action space describes the torques applied to each of the joints. All observation space values have a lower and upper limit of $-\infty$ to $+\infty$. All angles and velocities are

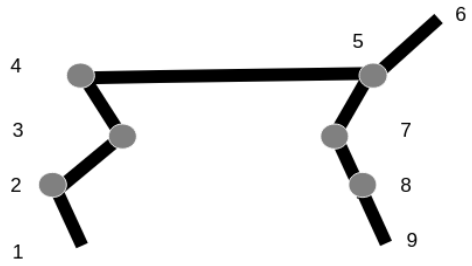


Figure 5: Half-Cheetah with labelled joints

measured in radiance and m/s respectively. The angle of rotations allowed for each joint is given in table 2. Torques applied to the joints have limits of $-1Nm$ to $+1Nm$.

The particular environment was chosen due to its relevance as one of the common evaluation environments in the field of DRL. By default, solving the environment is achieved by finding the controls that helps the cheetah run. The rewards are thus defined as:

$$r_t = \frac{(X_t - X_{t-1})}{dt} - \text{ctr\_weight} * \sum_{i=1}^{6} a_i^2, \tag{16}$$

where X_t is the position of the centre of mass of the agent at time t, along x axis, dt is the inverse of timesteps per frame time and a_i is the torque defined in the i^{th} dimension of the action space. By default, *ctr_weight* which add a weight to the penalty term is 0.1. The penalty on torque applied encourages an efficient gait in running forward.

Segment number	Segment name	Segment length
1-2	back foot	$\frac{0.2}{cos\frac{\pi}{2}}$
2-3	back shin	$\frac{0.15}{cos\frac{\pi}{3}}$
3-4	back thigh	$\frac{0.25}{cos\frac{\pi}{6}}$
4-5	torso	1
5-6	head	0.3
6-7	front thigh	$\frac{0.25}{cos\frac{\pi}{6}}$
7-8	front shin	$\frac{0.15}{cos\frac{\pi}{3}}$
8-9	front foot	$\frac{0.2}{cos\frac{\pi}{2}}$

Table 1: Segment name and lengths for Half-Cheetah-v3

Joint number	Lower bound	Upper bound
2	$-\frac{5}{6}\pi$	$-\frac{1}{6}\pi$
3	$\frac{1}{4}\pi$	$\frac{3}{4}\pi$
4	$-\frac{5}{6}\pi$	$-\frac{1}{3}\pi$
5	$\frac{1}{18}\pi$	$\frac{1}{2}\pi$
7	0	$\frac{8}{9}\pi$
9	$-\frac{2}{3}\pi$	$\frac{1}{9}\pi$

Table 2: Limits on Half-Cheetah-v3 joint angles

3 Background Literature

3.0.1 Literature Survey

Model-free reinforcement learning methods have proved to solve variety of tasks such as grasping [35], locomotion [36], driving [37] and navigation [38]. Their success and ease in implementation are contributing factors to their popularity in the field of RL. However, when learning in the real-world, these algorithms that rely heavily on a trial and error method can prove to be highly inefficient and even hazardous. For instance, in industrial robots, model-free approaches might force the agents to take actions that cause harm to objects in its vicinity and lead to mechanical wear and tear. These challenges along with sim-real problem limit the application of DRL in robotics. Model-based (MB) approaches, on the other hand, can prove to be highly sample efficient [39]. However, the limitation in MB approaches lies in two areas - non-linearity of the environment to be learned and an efficient planning mechanism. MB depends largely on the capacity of learning the real environment. Initially, the model-based approaches used simple function approximators such as Gaussians or linear models, to capture the dynamics of the environment. However, such simple models approximated the true environment dynamics to a point that hampered their utility. PILCO [40] improved on these simple model approximations by capturing the model uncertainty and including this uncertainty in planning over longer horizons. Many recent works with model-based approaches have shown success in using deep neural networks in capturing the model dynamics [41][42] with comparatively good accuracy and applicability to higher-dimensional tasks. Stochastic environments are further modelled on adding Bayesian approximations or probabilistic uncertainties [43] to deep neural networks. In spite of the attempts, pure model-based methods struggle in modelling the complexities of the environment and a policy learned through these methods demonstrate model-bias [22]. Our work also models the world dynamics using a deep neural network but tackles the model-bias through a shorter planning horizon and iterative retraining of the model to match the policy search space distribution.

Model based and model free DRL approaches have predominantly been treated separately. One of the earliest implementations of combining model-free methods with model-based planning was introduced by Sutton [44] with the Dyna architecture. This architecture suggested using random samples from a learned model of the environment in updating a value function through MF learning. Intermittently, this framework also draws samples from the real environment to update the learned model and the value function simultaneously. Dyna-Q [45] is a version of the Dyna framework wherein, the MF learning process follows the Q-learning approach. This framework was one of the first works to use MF methods in learning an efficient policy while using fewer real world samples and benefiting largely from MB samples. This was a huge milestone and was adapted by many future works [46][47]. Most of these works focused on making the planning more focused, i.e, instead of randomly sampling actions with the planner, the action were selected based on certain constraints. For example, [47] introduced Dyna-Queue, where the samples collected by planning in the model were put into a prioritized queue and only the samples with high rewards were used in updating the value function. Focused planning was able to elevate the sample efficiency attained by the original framework and lead to faster convergence. In this work, although we use a completely different framework of combining model-based and model-free learning, our main contribution also pivots around a more focused planning over the learned model.

Over the years, much focus has been laid on planning over a learned model, when combining model-based and model-free architectures. Planning has two main function - selecting actions to be executed

in the model and evaluating the quality of trajectories collected by executing the chosen actions. A planner such as MPC uses the standard random action selection procedure but compares multiple trajectories from a given state and chooses the action that leads to the best trajectory. Thus, MPC improves on evaluation of the trajectories rather than action selection. On the other hand, using Cross Entropy Method (CEM) as a planner, entails that every action is sampled from some distribution that is periodically updated to capture regions of best actions based on past samples. With CEM, the distribution is generally modelled as a gaussian. Thus, CEM attempts to optimise the action selection rather than trajectory evaluation. A prominent work in this field [32], compares naive planning with CEM and MPC planners and demonstrates the performance on OpenAI's cartpole task. The base architecture of this work builds a model from samples drawn from the real-world and then plans over this model, to generate samples that are used to learn a policy. It was found that their architecture gave highest sample efficiency with CEM planner being used to select actions. However, as mentioned, CEM models a gaussian distribution which has limited expressiveness. As tasks get complex, it may not be practical to represent the region of good actions by a simple gaussian. In this work, we use a policy learned through actor-critic RL to represent the best actions. Moreover, different states map to different regions of best actions. With a CEM, it's impractical to have a different gaussian distribution for every state in high dimensional tasks. In our case, since the policy used in suggesting actions is a neural network, it's more adaptable to high-dimensional tasks. The policy in [32] is learned by a neural network that is trained through gradient optimization, while using the the model-based samples.

Other works such as [14], instead of using a distribution to model region of best actions, they formulate a Bayesian regret bound that indicates regions of the learned model that have uncertainty. This information allows planning with caution and avoiding model-bias. Similarly, the work by [48] also employs the idea of capturing model uncertainty, but through a probabilistic ensemble of models as opposed to mathematically formulated bounds. This uncertainty is not quantitatively added to the planning, and presents itself as a form of randomness in the transition function. [48] also employs an MPC planner with CEM providing the constraints on the regions of best actions.

In the previous discussed works, although model-based samples are used in updating a value function through model-free methods, the policy is never used to collect better samples during the planning stage. In truly integrating the policy learning and planning stages, [49] proposes a method where a CEM planner and a policy network update each other. Their work, as in [32] also uses CEM in planning trajectories over the learned model and intermittently updates the learned model with real world samples. The candidate gaussian distributions of a CEM planner is used in sampling actions with the model used to predict the transitions and reward for each transition and thus, collecting trajectories with their associated cumulative reward. A neural network is trained to learn a policy that maximises the cumulative returns from a state through stochastic gradient descent (SGD). This updated neural network or policy is used to plan trajectories from the next timestep. Thus, in this iteration, instead of the of CEM planning the trajectories, the trajectories are planned by the policy and the CEM gaussian distributions are updated to match the distribution of actions from these policy sampled trajectories. This methodology was evaluated on the Half-cheetah and Pendulum environments of OpenAI. It was found that the proposed architecture performed was able to converge to an optimum policy in 250 iterations lesser than it took using a simple CEM planner on the Pendulum environment. However, on the half-cheetah environment, this architecture performed on par to using CEM. Our work takes inspiration from this idea of learning a policy and using this learned policy in sampling better actions during model-based planning. However, instead of using a neural network trained through SGD, we use an actor-critic approach to model-free learning.

Instead of using policy in selecting the action for focused planning, [50] used a policy in attempting to reduce the computational power utilised by MPC. [50] successfully proposes to use an RL learned policy in triggering the re-computation interval of MPC. However, keeping in mind the computational expense of random shooting (RS) in MPC, we propose to truly combine model-based and model-free approaches by directing the sampling of actions with a policy network learned through an iterative process, using a model-free approach.

The work by Nagabandi [15] incorporates the model-based and model-free learning in achieving sample efficiency with Mujoco locomotion task. Much like with many previous works [41][42], Nagabandi also models the environent dynamics with a deep neural network. They use a naive MPC in planning over short horizons on the learned model and collect the trajectories sampled from this training into a dataset. However, instead of using a simple neural network to learn a policy from this dataset, they use TRPO, an actor-critic model-free architecture in learning the policy. Previous works show that learning a policy through only model-based samples shows sub-optimal performance with high-dimenisonal tasks. In combating this issue, [15] proposes to use the dataset of model-samples to only initialise a start policy for TRPO. This initialisation warms starts the TRPO training and beyond this point, TRPO trains purely on the real world as any traditional model free approach. They refer to this model-free training as fine-tuning as the initial policy is fine-tuned with respect to the real world dynamics. This architecture is able to achieve a sample efficiency of 1e4 timesteps compared to pure TRPO trianing on Half-Cheetah environment. Taking inspiration from this architecture, our work implements a strategy to make MPC planning more focused. This strategy involves training a policy network every few planning iterations and using the learned policy in directing the actions chosen by MPC. With more focused planning, we are able to show our architecture gives better sample efficiency than the inspiration on the Half-Cheetah environment.

Table 3 gives a summary of some of the prominent research in the field of model-based and combining model-based model-free reinforcement learning.

Table 3: Summary of prominent Model-based RL papers

Article	Main contribution	Policy learning	Specifications	Environment	State / Action space
[40]	PILCO - learning probabilistic model and using uncertainty in planning with RS	gradient optimization	Planning with RS in discrete action space	Cartpole, riding a unicycle	12/2 (unicycle)
[44]	Dyna: Planning with random action selection	gradient optimization	Planning in discrete action space	Maze navigation	1/1
[45]	Dyna-Q: Planning with random action selection	Q-learning (Off-policy)	Planning in discrete action space	Maze navigation	1/1

[47]	Dyna-Queue: Planning with random action selection and prioritized sweeping for training	Q-learning (Off-policy)	Planning in discrete action space	Maze navigation	1/1
[14]	Planning with MPC with a defined regret bound	-	Planning with MPC over discrete action space	Pendulum, Reacher, Pusher, Cartpole	23/7 (Pusher)
[32]	Online Planning Based Reinforcement Learning for Robotics Manipulation	gradient optimization	Comparing planning with MPC using CEM,RS over discrete action space	Cartpole	4/2 (Cartpole)
[49]	Using in-loop trained policy and CEM planner in updating each other	gradient optimization	Planning with CEM and policy over continuous action space	Half-Cheetah, Pendulum	17/6 (Half-Cheetah)
[50]	Using in-loop trained policy in determining planning horizon for MPC	PPO (On-Policy)	Planning with MPC over continuous action space	Pendulum	4/1
[48]	Planning with MPC using CEM, modelling uncertainty in dynamics	-	Planning over discrete and continuous action space	Cartpole, Reacher, Pusher, Half-cheetah	23/7 (Pusher)
[15]	Model-Based learning with Fine-tuning	TRPO (On-Policy)	Policy trained with model-based roll-outs, MPC as planner, and used as warm start for model-free learning	Swimmer, Half-Cheetah, Ant, Hopper	26/8 (Ant)

4 Material

No external data was used for this project. However, the model-based segment required training data as we explain here.

4.1 Data collection

For our architecture, the first task at hand was to learn a model that approximates the real environment dynamics, which in our case is the Mujoco simulation. This meant that we required to collect many trajectories from the real environment. The trajectories were collected based on roll-outs from a random stochastic policy. The initial roll-outs were collected and stored in a file, to avoid repeated collection for every experiment. New data was only collected when the defined rewards were changed in the real environment. It is to be noted that the roll-outs collected in the model's initial training wasn't included in the count of total samples required to train our full architecture. This was owing to the re-usability of the data and the learned model, i.e, a model once learned could be used in multiple experiments unless the real environment itself is modified.

4.2 Data pre-processing

Traditionally, a learned dynamics model predicts the next state and reward given current state and action. However, very often, the current state and next state tend to be very similar. This made it difficult for our neural network to differentiate between the two. In order to work around this, as suggested by [15], we modified our data such that the output vectors would be the difference between current and next state, which we refer to as delta, rather that just the next state. Further we used standard scaling to scale the data. In particular, we maintained a separate mean and standard deviation for the state and actions from the collected data. The input states and actions were then scaled by subtracting their mean and dividing by the their standard deviation. The scalars for the deltas were calculated separately and they too were scaled in the same manner.

5 Methods

This section describes the proposed architecture and the roll of various contributing sub components.

5.1 Architecture

The proposed architecture is diagrammatically represented in figure 6. Before diving into the working of our architecture, we discuss the various components that constitute the proposed approach:

- *Model*: The first component of the architecture is the model (a). The term model is used here in sense of the term "model" in model-based RL. Specifically, our architecture maintains a sample model (a deterministic model). Therefore, the model mimics the real environment by encapsulating the environment's transition function and reward function. In other words, the model takes a state-action pair (s_t, a_t) and predicts the reward and next-state pair (r_t, s_{t+1}). The model is essentially a feed-forward neural network.

- *Planner*: The second component of our architecture is the planner (b). The planner always accompanies a model in model-based RL. While the role of a planner remains the same as discussed in section 2.1.2, the constitution of the planner is different. Instead of using MPC, the planner combines MPC with PPO. Given a state s_t, a naive MPC planner would generate m trajectories by planning over the model, using actions randomly sampled. However, in our architecture, these actions are not sampled randomly and instead are generated by the PPO policy network.

- *Dataset*: The dataset (c) is used to store experiences collected by the agent. The agent only performs actions in the real environment. Therefore, all experiences stored in the dataset are real and not those simulated by the model.

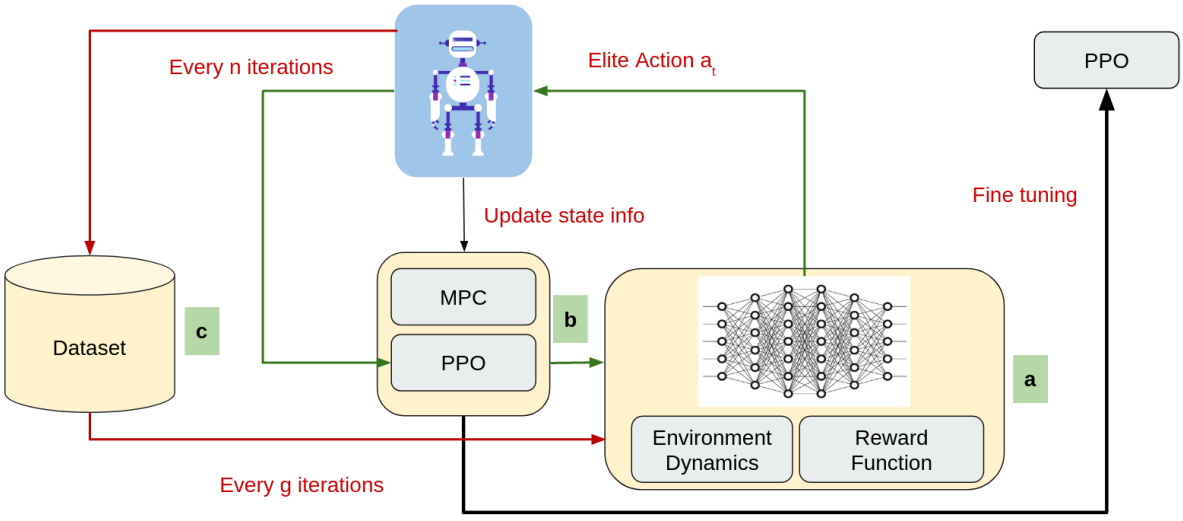


Figure 6: Proposed hybrid Model-based Model-free framework with model-free tuning

Following the orange line in figure 6, the first step with our architecture is to build and train the model. To acquire the dataset required to train the model, the agent follows a random policy in the real environment and collects multiple experiences (s_t, a_t, r_t, s_{t+1}). These experiences are stored in the dataset (c) as they are collected. Once the required number of experiences have been collected, the agent and the environment are reset. The model uses the dataset to learn the transition and reward function of the environment through supervised learning. Once a model of the environment is created, we are ready to plan over this learned model.

The policy training process is shown with the green arrows in figure 6. Taking the agent's current state $s_{t'}$ as the start state, the planner must roll-out m trajectories from $s_{t'}$ of length h. Here, t' represents the timestep in the model while we reserve t to represent timestep in the real environment. For a better understanding of the interaction between the planner and the model, we refer to figure 7. At every model timestep t', the PPO policy network suggests an action $a^k_{t'}$, where $0 < t' < h$ and $0 < k < m$, for every state $s^k_{t'}$. This generates m state-action pairs, $(s^0_{t'}, a^0_{t'}), (s^1_{t'}, a^1_{t'}), ...(s^{m-1}_{t'}, a^{m-1}_{t'})$. For $t' = 0$, all states $s^k_{t'} = s_t$, for k ε $[0, m-1]$ and s_t is the agent's current state in the real environment. These state-action pairs are then fed to the model, which in turn predicts the reward and next-state for each of the pairs. This generates a quadruple defined as $(s^k_{t'}, a^k_{t'}, r^k_{t'}, s^k_{t'+1})$ for each state-action pair $(s^k_{t'}, a^k_{t'})$. This process of the policy sampling actions for m states and the model predicting the reward and next state for each of the m state-action pairs generates m trajectories of length 1. In order to generate a trajectory of length h this process must be iterated h times with t' being incremented with every iteration. For each iteration, the next-states predicted by the model in the model's previous timestep, is fed as the current state to the policy, for each of the m trajectories.

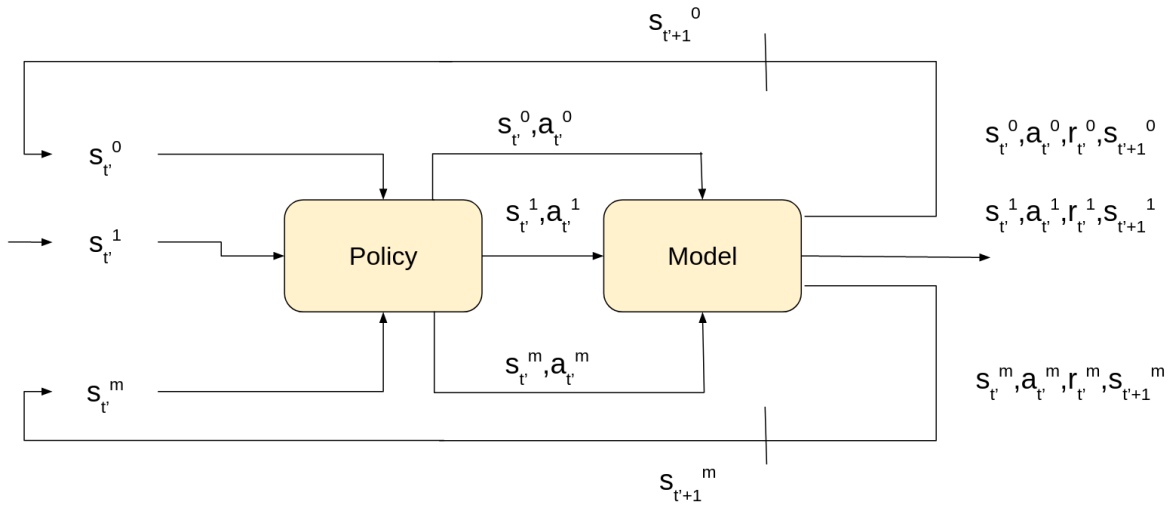


Figure 7: Detailed visualisation for the interaction between the planner and model

Once the m trajectories of length h are generated, the elite action a_t is extracted as described in section 2.1.3. Only this elite action is executed by the agent in the real environment. Action a_t transitions the agent from s_t to s_{t+1} along with a reward r_t. This real transition (s_t, a_t, s_{t+1}, r_t) is added to the dataset (c). The real transitions are also stored in PPO's training buffer. One iteration of planning m trajectories and executing the elite action in the real environment, moves the agent by one timestep in the real environment. With s_{t+1} as the agent's current state, our architecture once again plans over the

model in generating m trajectories and selecting the elite action a_{t+1}. This loop of planning, updating agent's state and saving the real transitions is continued for n iterations. Every n iterations, the PPO policy is updated using the real transitions stored in its training buffer. Thus, every n iterations, the policy improves and the actions sampled by our planner get better.

As PPO is an on-policy algorithm, after every update, PPO clears its training buffer. As the policy improves, the distribution of states shifts and maybe unknown to the learned model. Under such a circumstance, our model is retrained every g iterations with the trajectories stored in the dataset (c). This is as shown with the the red line path in figure 6. Through this iterative process, we train our PPO policy till it attains a sub-optimal performance. Due to model-errors and consequential error in planning, the learned policy would not be optimal, thus requiring fine-tuning. This partially trained policy is used to warm start pure PPO training. In other words, PPO's policy network weights are now initialized to that of the sub-optimal policy. From this stage, pure model-free training is conducted on the real environment, fine-tuning the policy.

We hope that with prior planning our policy encounters and learns from better trajectories. With good trajectories to train on, we suppose that our policy is able to learn an optimal mapping from states to actions taking fewer samples from the real environment.

The architecture is inspired by the amalgamation of model-based and model-free approaches presented by [15]. However, our architecture differs in three core aspects:

1. In the planning stage (b), we incorporate the model-free algorithm (PPO) with MPC while the inspiration uses only a naive MPC (with random shooting). The intuition behind our modification is the possibility that using a policy to select actions would lead to better trajectories than randomly selecting actions at every timestep.

2. We propose to use PPO instead of TRPO as the model-free learner. This choice was made due to the computational superiority of PPO as compared to TRPO.

3. The inspiration defines the reward function and calculates the reward for each transition using this defined function. However, we learn the reward function along with the transition function, using a feed-forward neural network. Thus, the reward is predicted for every model-based transition instead of calculating it.

5.2 Model-Based training

A complete environment has two components - the dynamics and the transition rewards. We build a feed forward neural network to learn both these functions. We require our network to take a state-action pair as input and predict a next-state-reward pair. However, since state and next-state are often very similar for 1 step transitions, the neural network struggles in learning the next-state predictions. Thus, instead of trying to predict the next-state, we plan our network to predict the delta (D) (difference between current state and next state). Thus, the network must take as input a vector of the 18 dimensional current state and 6 dimensional action. The 18 dimensional state includes the 17 default values and the x position of the body's centre of mass. The network must produce an 18 dimensional delta along with a scalar reward as output. In getting the next state prediction, we add the delta to the current state value.

To build the model, we must train the neural network through supervised learning. Before training the model, we collect experiences by running a random policy in the real environment. Each experience is of the form (s_t, a_t, r_t, s_{t+1}). Since we require our model to map (s_t, a_t) to (D_t, r_t), we modify all experience tuples to $(s_t, a_t, r_t, s_{t+1} - s_t)$, where $s_{t+1} - s_t$ is D_t. The mean and standard deviation for all states, actions, delta and reward values are individually calculated. To scale all values in the dataset, for each state, action, delta and reward, we subtract from it the mean of the respective quantity and divide the result by the quantity's standard deviation. This is called standard scaling and crunches all values to between 0 and 1. The data is then fed to the neural network which learns the mapping through gradient descent. With gradient descent, the weights of the network are adjusted with the aim to reduce the objective function. The objective function in our case is defined as:

$$\text{MSE} = \text{mse}(D_{t+1} - D'_{t+1}, r_t - r'_t) \tag{17}$$

Here mse() is a function which calculates the mean square error, D'_{t+1} and r'_t represents the delta and reward predicted by the model and MSE represents the objective function to be minimised. Thus a single feed forward network is used to model the transition and reward function of the environment. Further, during predictions, the action values were clipped to between -1 and +1, which is the permissible range for action values in the Half-Cheetah environment.

5.3 Planning with MPC and PPO

The novelty of our research lies in combining MPC and PPO for model-based planning. As already described in section 2.1.3, a naive MPC planner takes random actions at every timestep in planning trajectories, i.e, it samples from a uniform distribution $U(-1, +1)$ for all states. In our architecture, the actions at every timestep is planned by our PPO policy. Since our policy is stochastic, all m actions directed by the policy from a particular state would be different. However, a policy samples the actions chosen for a state s, from a Gaussian distribution $\mathcal{N}(\mu_s, \sigma_s^2)$. As described in 5.1, the policy is trained every n iterations, which results in the mean and standard deviation being updated to encompass the best actions for each state based on observed historical trajectories. Thus, with a policy, we map every state to a probabilistic distribution of best actions. In view of control theory, this policy forms the constraints for MPC planning [51]. Consequently, a policy increases the probability of choosing actions which are optimum as compared to naive MPC.

Further, keeping in mind that the model is only an approximation and not an exact replica of the real environment, the planning horizon over the learned model is kept short. Since error in model prediction accumulates at every timestep, with short horizon planning, we ensure that our planned trajectories are very much in line with the real world dynamics. It is also to be noted that although we plan in the learned environment, our policy is trained only on the real trajectories.

5.4 Task Description

Solving the Half-Cheetah environment is achieved by making the half-cheetah walk or run for 1000 timesteps. However, this is the basic necessity. Higher rewards are obtained as the agent learns to walker faster. Further, the agent must also attain a standard and symmetric gait.

In order to test our architecture and how various factors impact the outcomes, we run a few experiments, as described here.

5.4.1 Experiment 1 - Deciding on planning horizon length

As with any DRL approach, hyper-parameter tuning can make all the difference. While for our MF counterpart, these hyper-parameter values were adapted from [11], MB required some experimentation in determining the apt hyper-parameter values. In particular, when dealing with planning in MB approaches, one must determine the horizon length over which our planner can safely plan. The longer the planning horizon, the more model error our architecture is encountering. The planning horizon should be set such that, we are able to extract the advantages of planning into future while ensuring that the model errors in the planning are small enough to be overshadowed by these gained performance benefits. Our first set of experiments is thus run to determine the ideal number of timesteps for planning into the future, over the learned model.

5.4.2 Experiment 2 - Comparing planning with MPC to planning with MPC and PPO

The novelty of our research lies in the proposal of using a PPO policy to direct the actions chosen by MPC. For all result discussion and plots, we refer to our architecture as *PPO-MPC*. If the planner block (block a in figure 6) is replaced by a naive MPC, we refer to the architecture as *MPC* and when using a pure model-free algorithm such as PPO, we refer to it by the name of the algorithm. To evaluate our method's effectiveness, we compare the performance of our architecture, *PPO-MPC* against the performance of *MPC*. We also compare our architecture against pure model-free learning, which is PPO in this case, to thoroughly analyse its performance. These comparisons are made based on the returns attained by each architecture.

5.4.3 Experiment 3 - Robustness to rewards

The next set of experiments were conducted to analyse the impact of different reward functions on our architecture. The reward function used in experiment 2 was the default set for the environment by Mujoco. Considering the popularity of the Half-Cheetah environment in DRL evaluation and the high success rate of solving it, it is safe to assume that the reward has been set based on careful experimentation. Thus, it is worth analysing the sensitivity of our architecture to the quality of the reward functions. To evaluate the robustness of our architecture to the defined reward functions, we further test it with three different reward functions for solving the Half-Cheetah environment. The custom rewards are based on our observations during training and focus on adding a penalty for unwanted behaviour.

We experiment with three reward functions, including the default Mujoco reward (equation 16). For all the three rewards, we retain the first component of equation 16, that rewards the forward movement of the centre of mass along the x direction. As the primary task is to have the half-cheetah move forward, this reward is essential. The second term of equation 16, is modified to experiment with different penalties. Thus, our rewards share the common structure:

$$r_t = \frac{(X_t - X_{t-1})}{dt} - penalty_t, \tag{18}$$

where R_t, X_t and dt hold the same meaning as for equation 16. $penalty_t$ is the penalty terms that varies with each reward function. The default reward function penalises actions as described in section 2.1.5. The second reward function, HeadPen defines a penalty for the half-cheetah's head falling

too low along the y direction. Based on observations, if the y co-ordinate of the head ($head_y$) falls lower than -0.2, the half-cheetah tips over. The third reward, ShinPen penalises the half-cheetah moving the front leg too far back. Based on observations, the half-cheetah tips over when the positions for front thigh, front shin or front foot are larger than 0.2, 0 and 0 respectively. Thus, ShinPen penalty is the sum of the penalties for front thigh, front shin and front foot. The different penalties are summarised in table 4. In analysing the performance of our architecture with HeadPen and ShinPen reward functions, we plot the performance of *PPO-MPC* against *MPC* and PPO in each case. It is to be noted that when experimenting with the differet reward functions, we only redefine the reward function in the real environment. The model must re-learn the transition and reward functions from data collected by a random policy run in the modified real environment.

We also compare the gaits achieved with each reward function in determining their quality. The gait is studied using plots that show the vertical and horizontal displacement of the half-cheetah's centre of mass over time, the power consumption per body joint and screenshots of the simulation on the agent following the trained policy. We study these plots and figure and analyse what they entail for our architecture.

Reward title	**penalty term**
Default	$0.1 * \sum_{i=0}^{d}(a_i^2))$
HeadPen	$0.1\ if\ head_y < -0.2$
ShinPen	$0.1\ if\ front\ thigh > 0.2 +$ $0.1\ if\ front\ shin > 0 +$ $0.1\ if\ front\ foot > 0$

Table 4: Rewards used in experimentation

5.4.4 Experiment 4 - Applicability to other model-free algorithms

Our architecture uses PPO at its core. The choice of this algorithm was made based on the application and the inherent sample inefficiency in PPO. However, with booming research in DRL, new and improved algorithms are always at the next turn. Further, while PPO has seen to perform better in most continuous control tasks, certain task specificity requires the application of other model-free techniques. With these considerations, we test our architecture when PPO is switched with a different model-free algorithm. Here we choose, soft actor critic (SAC) [52] algorithm for two reasons:

1. For our architecture's purpose, the chosen MF algorithm must be applicable to continuous action space and must have a stochastic policy network. Both these criteria are satisfied by SAC.

2. Since PPO is an on-line policy optimization method, we test our architectures effectiveness for off-policy algorithms with SAC.

When running our architecture with SAC, we replace the PPO component of of the planner block (block a in figure 6) with SAC. We refer to this modified architecture as *SAC-MPC* and compare it's performance against using just SAC in training the half-cheetah.

6 Experimental Setup

This section describes the implementation details, hyper-parameter turnings and hardware as well as software specifications. Having taken care of the details mentioned in this section, it is possible to achieve similar results as described in this paper.

6.1 Tools and Technologies

All code is written in Python 3.7. Tensorflow 2.2 is used in building the DRL framework. The PPO and SAC algorithms used were adapted from OpenAI Baselines [33]. However, both algorithms have been modified to amalgamate with the model-based components. Further, the code is GPU optimised.

The architecture was evaluated on the Half-Cheetah v3 environment, which is a part of the Mujoco environments [1]. Mujoco is a physics engine that provides environments with multi-joint dynamics and contacts. Maintained by OpenAI as part of their gym environments, Mujoco tasks have become a standard in DRL evaluation. Further, OpenAI Baselines provide easy integration with the Mujoco environments and provide various functionalities through simple function calls.

All experiments were run on Ubuntu 18.04 operating systems. The system included an Intel-i7 8 core processor with 16GB memory and an NVIDIA GTX 1060 GPU.

6.2 Performance Criteria

In this section, we describe the criteria and metrics used in evaluation of various sections of our architecture.

6.2.1 Cumulative Rewards

When evaluating a policy in RL, the most prominent criteria for judgement is the discounted cumulative rewards or the returns. As already discussed, a policy is trained to optimize the sum of rewards it can receive over specified timesteps or episodes. An increase in the returns obtained indicates an improvement in policy. It is natural however, to see some peaks and falls in the returns as a side effect of exploration and exploitation.

6.2.2 Mean Square Error

The model-based segment of our architecture follows the framework of traditional supervised learning. Thus, the quality of our model is measured in terms of the mean squared error (mse) between the predictions and the actual values. mean square error is calculated as:

$$\text{mse} = \frac{\sum_{i=0}^{N}(f_i - \hat{f}_i)^2}{N}, \tag{19}$$

where N is the length of output vector (state dimensions), f_i is the real vector and $\hat{f}_i$ is the predicted vector. In our case the output vector is an 18 dimensional delta vector concatenated with a scalar reward value. Thus, our model learns a mapping from the current state and action to the delta for the

next state and a reward. From this prediction, next-state is calculated by adding delta to the input state vector.

Lower the mse, better the predictions. Traditionally mse is calculated for one step predictions. In our case however, we plan multiple steps over the model. To account for this behaviour and analyse the expected error in our planning, we also calculate the mse over multiple steps. We further use the multi-step mse to determine how long of planning horizons remains useful without being overshadowed by the model error.

6.2.3 Gait analysis

In this thesis, we also test the behaviour of our architecture to various rewards. In our case, these rewards are written to achieve the half cheetah running task. To understand the behaviour of our architecture and how it relates to the quality of our rewards, we analyse the best gaits obtained with each reward. In gait analyses, we note the following:

1. Average power consumed by each joint for achieving a certain gait. The average power is calculated as:
$$P_i = \frac{(\tau_i^j * \omega_i^j)}{N} \tag{20}$$
where P_i is the power consumed by joint i, averaged over N tiemsteps. τ is the torque and ω is the angular velocity.

2. Stability of gait based on the vertical displacement of the centre of mass during a gait. More the vertical displacement, lower the stability.

3. How far the agent moves along the x direction in a certain number of timesteps. This is depicted with a plot showing the horizontal displacement of the half-cheetah's centre of mass

6.3 Hyper-parameters

In this section we describe the hyper-parameters relevant to our algorithms and the values chosen for each. We divide the hyper-parameters into three sections - those pertinent to feed forward neural networks, those to model-free and finally those to model-based.

6.3.1 Feed forward neural network hyper-parameters

The actor, critic model along with the learned model in the MB section of our architecture, are all essentially feed forward neural network. In constructing a neural network, the following hyper-parameters are considered:

Input layer: This is the first layer of the neural network through which the input is fed to the network.

Output layer: This is the final layer of the neural network. The values that come out of this layer are the predictions of the neural network.

Hidden layers: These are layers of neurons that lie between the input and output layer. A hidden layer takes the input from the previous layer, multiples it with a weight matrix and passes it through a function to get the output, that is fed to the next layer. Each additional hidden layer allows the network to realise more non-linear relationships in the data. However, more the layers, larger the network and more data required in training. Thus, having a large number of hidden layers can result in underfitting.

Hidden Neurons: These are the neurons that make up the hidden layer. Each layer is allowed to have different number of neurons. Hence, the number of hidden neurons is specified as a vector, where each value of the vector corresponds to one hidden layer. Hidden layers with more neurons can also help the network learn complex relationships at the cost of more training.

Activation function: The activation function is applied to the output of a layer before passing it to the next layer. The function of the activation function is to determine which neurons of the layer will be active by bounding the values of the output vector within a range. There are various activation functions that have different bounding ranges. Some of the common activation functions are sigmoid, tanh, linear and ReLu.

Optimiser: These are algorithms that determine the change in the networks trainable parameters, in order to minimise the error in the network's predictions. Traditionally, the direction of change in parameters to reduce loss is determined by gradient descent or stochastic gradient descent. Other popular optimisers such as Adam also are based on these traditional methods, with slight improvements.

The actor and critic neural networks are designed as shown in table 5.

Hyper-parameter	**Value**
Actor and Critic hidden layers	2
Actor and Critic hidden units	[64 64]
Actor and Critic hidden layer activation	ReLu
Actor output layer activation	tanh
Critic output layer activation	linear
Actor and Critic optimiser	Adam

Table 5: Actor and Critic network configuration

6.3.2 Model-free Hyper-parameters

Epochs: This value indicates the number of passes made through the training buffer before updating the network weights. Larger this value, more the network learns from the samples in the training buffer. However, too large a value could lead to the network over-fitting to the current set of samples.

Minibatch size: This value indicates the number of timesteps or transitions used in one epoch. Larger minibatch sizes allow the network to get a more general view of the data thus leading to stable updates. However, minibatch sizes must be a number divisible by the buffer size such that all samples

get an equal opportunity in contributing to the weight updates.

Clip value: The clip value which is usually between 0.1 and 0.3 is specific to PPO. This value adds a lower and upper bound to the possible divergence between the new and old policy when computing the loss function. The clip value ensures that PPO has stable policy updates.

Entropy coefficient: The entropy coefficient plays the role of a network regularizer. A well tuned entropy coefficient prevents premature convergence of a policy.

Value function coefficient: The value function coefficient determines the weightage of the value network loss as opposed to the policy network loss, when computing the global loss function.

Gamma: Gamma is the discount factor that indicates the weight our network assigns to future rewards. The smaller the value of gamma, the more our policy is optimized for immediate rewards.

Lambda: This parameter is a smoothing factor that ensures stable training by reducing the variance in each network update.

Learning Rate: Learning rate value indicates by how much the weights are updated with each iteration. Higher learning rates lead to faster learning but at the risk of converging to a local minima.

All hyper-parameter values for PPO are adapted from the official implementation of PPO [11], on the Half-Cheetah environment. The values set for all these hyper-parameters are set as shown in table 6. Further, The actor and critic networks are configured as shown in table 5.

Hyper-parameter	**Value**
PPO Horizon	2048
Minibatch size	32
Epochs	4
Clip Value	0.2
Entropy Coefficient	0.0001
Gamma	0.99
Lambda	0.95
Learning Rate	0.0003
Value function Coefficient	0.5

Table 6: PPO Hyper-parameters configuration

6.3.3 Model-based Hyper-parameters

Horizon : The number of timesteps over which we plan using the learned model. A very short horizon would make our planning myopic while a long horizon could lead to accumulation of model prediction error.

Number of trajectories: This value indicates the number of trajectories simultaneously planned from a given state. More the number of trajectories, more optimal the chosen elite action. However, planning multiple trajectories at a time adds a computational overhead and thus must be maintained at a nominal value balancing the computation expense and the planner's performance.

Iters per aggregation: As the policy learns, the distribution of the data sampled shifts. Thus, our learned dynamic model must be refit every few iterations of policy updation. Iters per aggregation determines how many updated of the policy we must wait before refitting our learned model.

Apart from these hyper-parameters, as in every neural network, we also tune the number of layers, hidden units per layer, activation function, optimiser, batch size and epochs for training. All values for the hyper parameters for the model-based architecture are as shown in table 7.

Hyper-parameter	**Value**
MPC Horizon	10
Number of trajectories	15
Hidden layer	2
Hidden Units	500
learning rate	0.0001
Hidden layer activation	Relu
Output layer activation	Linear
Batch size	512
Epochs	200
Regularisation	Early stopping
Iters per aggregation	50

Table 7: MB hyper-parameters configuration

7 Results and Discussion

In this section, we document the results for various experiments and discuss and interpret the outcomes.

7.1 Results

7.1.1 Experiment 1

With regard to our architecture, an important hyper-parameter to be tuned was the horizon length (h) over which we plan in the learned model. We ran our architecture thrice for horizon length of $5, 10$ and 25. Each experiment was run for 4e5 timesteps or 200 iterations. All other hyper-parameters were kept constant throughout the 3 experiments. For each value of h, we ran our experiment for 3 different seed values and averaged the returns, in order to generalise the results. Figure 8 shows the average returns achieved for each candidate value of h. It is also to be noted, this experiment is run as a preliminary in confirming the best horizon length and does not lead to the agent walking.

From figure 8, there's an evident lag in the performance of our architecture with $h = 25$. Among runs with $h = 5$ and $h = 10$, we see a superior performance with a planning horizon of 5 timesteps, in the initial stages of training. However, around 1e5 timesteps, this trend begins to reverse with $h = 10$ gaining higher returns. Until around $2e5$ timesteps, the horizon lengths don't signify a large difference in performance, although having averaged over multiple seeds, we can be certain that the little difference is a reliable trend estimate of how the planning horizon effects our architecture performance. Beyond $2e5$ timesteps, we notice a stagnation in the performance improvement with $h = 5$ and $h = 25$, while $h = 5$ performs slightly better in comparison to $h = 25$. On the other hand, the experiment run with $h = 10$ continues to improve steadily and achieves a maximum average return of 1908.

To visually analyse the effect the horizon lengths have on our learned model's predictions, we show a comparison between our model's predictions of the dynamics and the real world dynamics in figure 9. We present the predictions for x position of the the half cheetah's centre of mass for $h = 5, 10, 25$. The model also predicts the rewards associated with a transition. Figure 10 shows the reward predictions against the real world rewards for $h = 5, 10, 25$. While these images give a qualitative idea of the model predictions, table 8 gives the average mse in prediction for each of the horizon lengths for a quantitative comparison. Similarly, table 9 reports the mse in prediction of the rewards by the model. Based on our analysis, we set the horizon length to 10 and move on with the remaining experiments.

Horizon length (h)	Mean Square Error
5	2.359
10	2.489
25	3.0342

Table 8: Mean square error in the model's prediction of the x-position of half-cheetah centre of mass, for $h = 5, 10, 25$

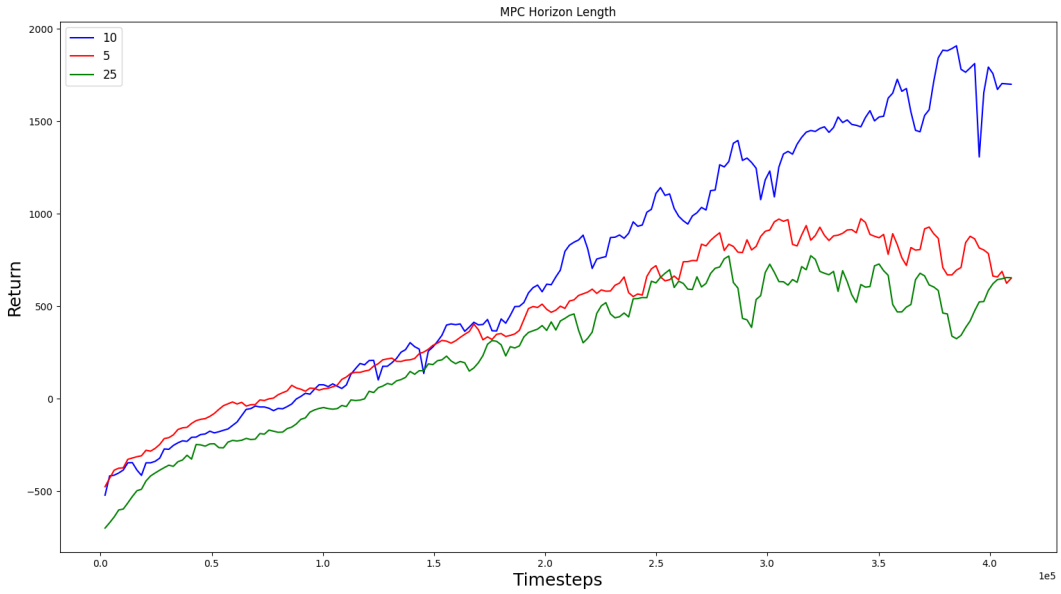


Figure 8: Preliminary analysis: Comparing return on running the architecture with planning horizon lengths (h) of 5,10 and 25. Plot shows highest returns when $h = 10$

Horizon length (h)	Mean Square Error
5	0.3856
10	0.50635
25	0.6238

Table 9: Mean square error in the model's prediction of rewards for $h = 5, 10, 25$

7.1.2 Experiment 2

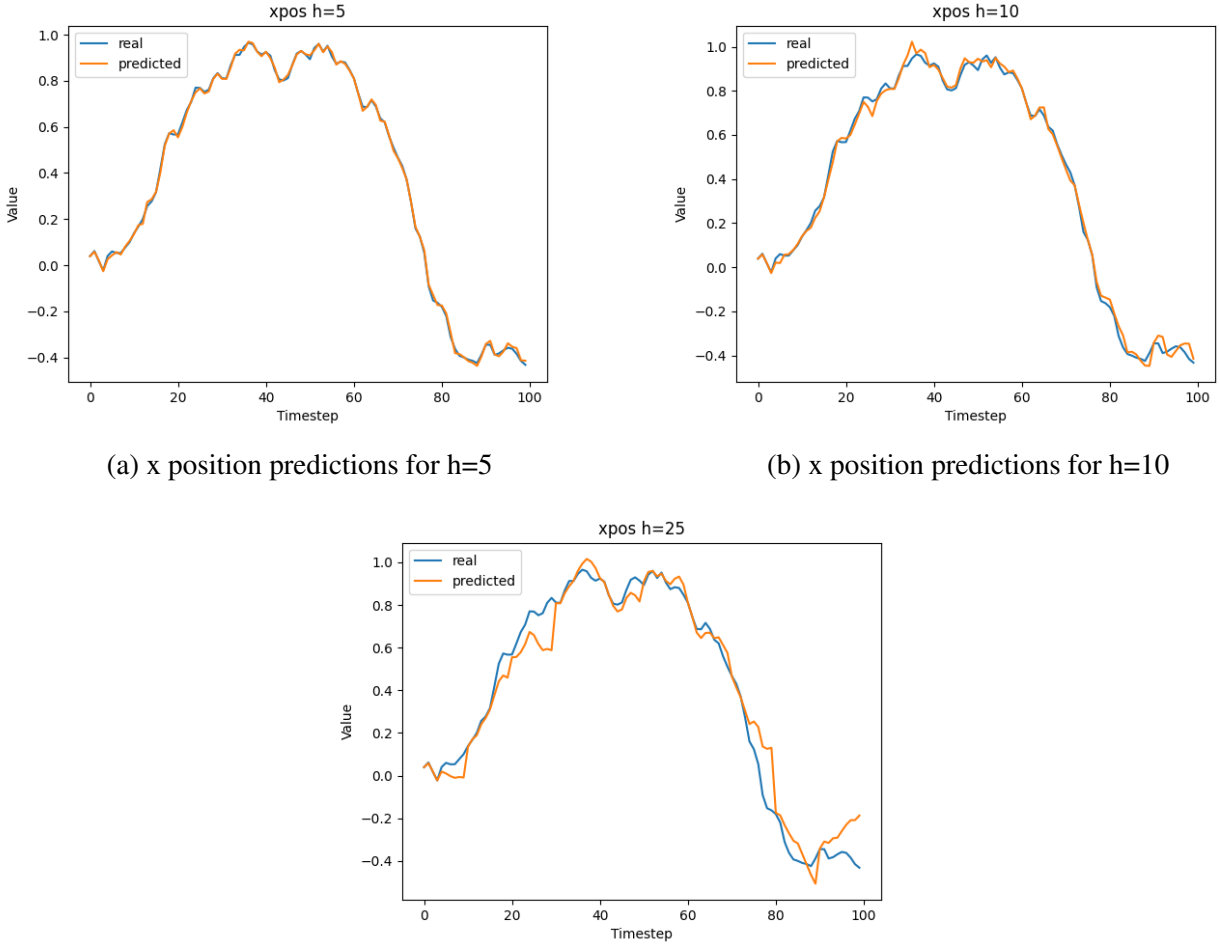
Figure 11a shows the returns attained for vanilla PPO, *MPC* and *PPO-MPC* over 200 updates or around $4e5$ timesteps. The solid coloured lines represents the average return over last 10 episodes. The shaded region shows the range of minimum and maximum returns over the past 10 episodes. In the initial stages of training, *MPC* and *PPO-MPC* both show better returns than PPO. The short horizon planning has an evident advantage over PPO. However, around $2e5$ timesteps, *MPC* performance stagnates with the average returns fluctuation between 100 to 600. The performance of PPO and *PPO-MPC* is seen to improve as the return increase over timesteps. Further, the plot also shows *PPO-MPC* hitting the highest return achieved by PPO, almost $1e5$ timesteps earlier than PPO itself. This point at which PPO achieves maximum score is marked by the black non-continuous line in the plot and is here on referred to as "PPO-maximum". Table 10 notes the exact timesteps at which each of the architectures hit the PPO-maximum return. As the *MPC* architecture never achieves the PPO-maximum return during training, its value is not tabulated. Beyond $3.5e5$ timesteps, we also notice a slower rate of increase in performance with *PPO-MPC*. Although our architecture plans on policy, the short horizon planning may not benefit the policy in long run. However, at 200 updates, *PPO-MPC* still achieves a score higher than PPO by 1000 points. At this point, we use the policy trained so far in warm starting pure MF training. Thus, taking advantage of a better optimised policy,

we proceed to fine-tune it with PPO alone in the real environment.

Figure 11b compares the score of PPO and *PPO-MPC* after fine-tuning. We stopped the combined MB and MF training at 200 updates, while the policy still shows steady improvement and continue training the policy with PPO in the real environment. We notice that the leverage our architecture provides in the first 200 updates is carried on. While PPO attains a score of around 3000 after $1e6$ timesteps, *PPO-MPC* attains the same score around $7e5$ timesteps. Thus, in the specific experiment settings, our architecture provides a sample efficiency of $5e5$ samples (or timesteps).

Architecture	**Score**	**Timesteps**
PPO	1247	393216
MPC	-	-
PPO-MPC	1265	274432

Table 10: Values describing Figure 11a PPO-maximum



(a) x position predictions for h=5

(b) x position predictions for h=10

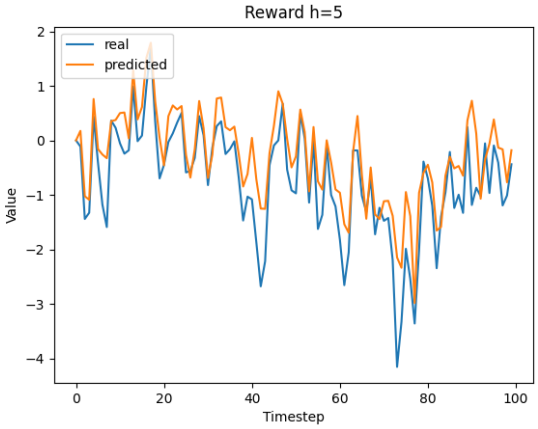
(c) x position predictions for h=25

Figure 9: Comparing the predictions for x position for various planning horizons. Blue shows the real states while orange shows the model predictions.

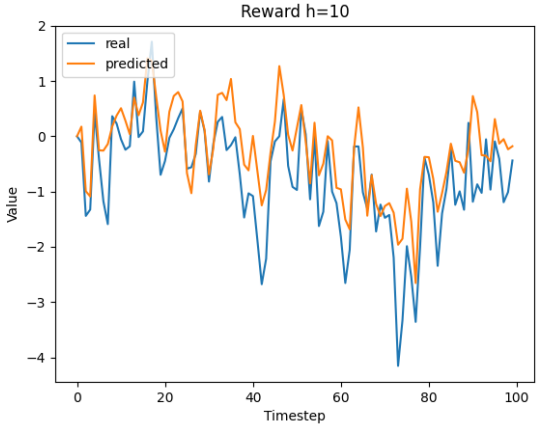
7.1.3 Experiment 3

For further analysis, we run experiments to test the sensitivity of our architecture to different rewards. From figure 12a, we notice that with the HeadPen reward function, in the initial stages of training, PPO shows the lowest performance followed by *MPC* and *PPO-MPC*. Just as in our experiments with the default reward function, we also notice a stagnation in performance around 1.5e5 timesteps with HeadPen reward function. However, with this reward function, we note a slower rate of improvement with *PPO-MPC* around $1.5e5$ timesteps. In fact, around $2e5$, PPO's performance overtakes that of *PPO-MPC*. Plot 12b shows the returns from PPO and *PPO-MPC* policy after fine-tuning. We see that *PPO-MPC* policy achieves lower returns than PPO, even after the former is fine-tuned. There is also a significant drop in the returns achieved with *PPO-MPC* around $3.25e5$ timesteps. Once again, as *MPC* architecture performance stagnates early on, we do not continue training *MPC* beyond $4e5$ timesteps.

With the ShinPen reward function, in figure 13a we notice our architecture once again falling short of PPO. However, unlike in case of the HeadPen reward function, *PPO-MPC* scores do no stagnate



(a) Reward predictions for h=5



(b) Reward predictions for h=10

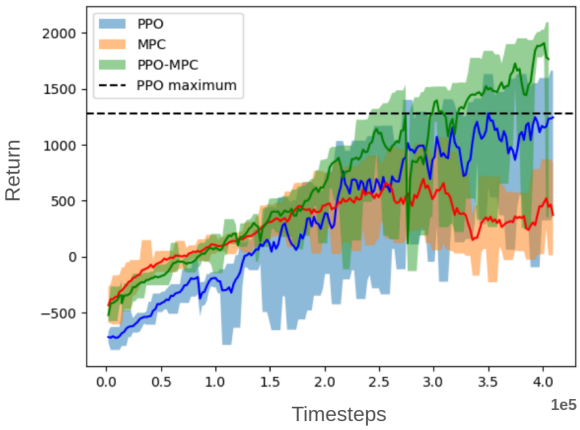
(c) Reward predictions for h=25

Figure 10: Comparing the predictions for rewards for various planning horizons. Blue shows the real rewards while orange shows the model predicted rewards.

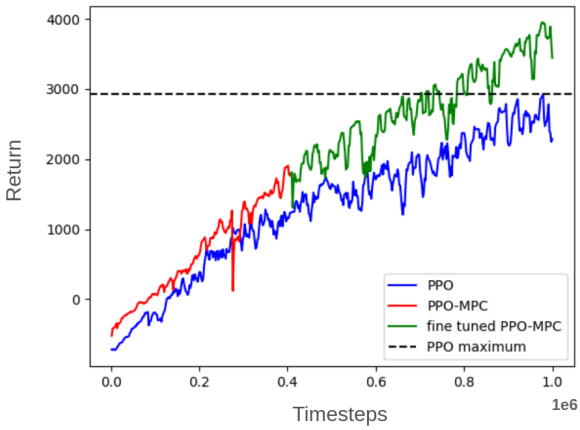
as early on. It is still found to improve beyond $4e5$ timesteps. As with previous reward functions, the *MPC* architecture's performance stagnates once again, but around $2e5$ timesteps. Considering at each step, PPO still performs better than *PPO-MPC* for this reward function, we do not train it any further. On fine-tuning the *PPO-MPC* policy with PPO, as seen in figure 13b, our architecture shows worse performance than PPO with lower returns. These results throw light on the high sensitivity of our architecture to the reward function.

Further, considering our architecture's sensitivity to the reward function, we also study the quality of these functions. While an increasing return indicates a policy being learned, it tells very little about the quality of the policy in relation to the task we wish to achieve. In order to study the gait with minimal bias of any architectural details, we compare the policy obtained from our architecture for all three rewards, keeping all hyper-parameters constant. In our case, the RL task is solved when the half-cheetah is able to run in the environment. All three rewards were successful in developing a gait enabling the half cheetah to move forward. However, with changing penalty terms, the gait itself differed. Figure 14 shows the power consumed by each joint for each of the rewards. Although the default reward penalises the torques applied, the ShinPen reward produces most energy efficient gait. In general, for all the gaits, we notice that the maximum power is consumed by the back thigh joint while the minimum is consumed by the back shin joint.

A constant factor of all three rewards is the bonus on the forward velocity achieved. However, we analyse if the gait developed in moving forward is biologically relevant to a cheetah's gait in nature. A stable gait would have the least vertical movement of centre of mass (COM) of the body. Thus, we plot the vertical position of the body's COM or the centre of torso in our case, over 60 timesteps, shown in figure 15. The HeadPen reward clearly has the most COM vertical displacement as is seen from a highly unstable gait in the visualisation. Although the half-cheetah manages to move for-

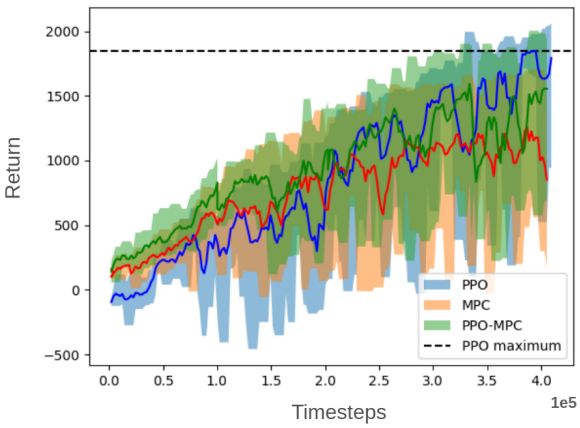


(a) Comparing returns of PPO, MPC and *PPO-MPC* architectures for $4e5$ timesteps. The solid lines depict the average returns over past 10 episodes. The shaded region depicts the range between minimum and maximum returns over the past 10 episodes.

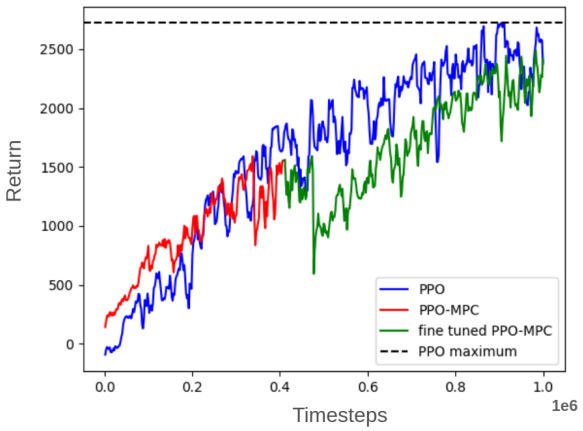


(b) Comparing returns of PPO and *PPO-MPC* architectures. The red line shows the returns achieved during training with model-based planning. After $4e5$ timesteps, the blue line depicts *PPO-MPC* returns as the policy is fine-tuned with PPO

Figure 11: Plots to analyse the returns from *PPO-MPC* architecture in comparison to PPO and *MPC*. Half-Cheetah environment reward function is set to Default

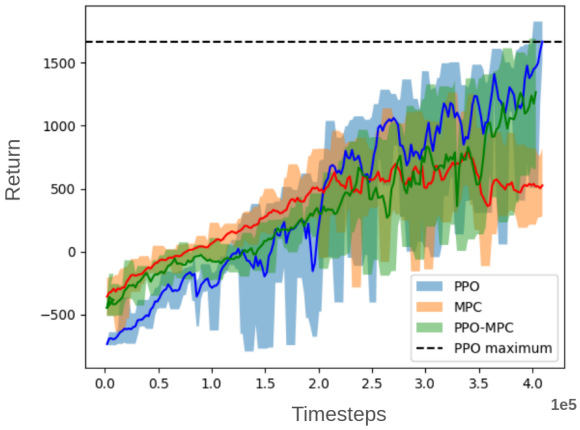


(a) Comparing returns of PPO, *MPC* and *PPO-MPC* architectures for 4e5 timesteps. The solid lines depict the average returns over past 10 episodes. The shaded region depicts the range between minimum and maximum returns over the past 10 episodes.

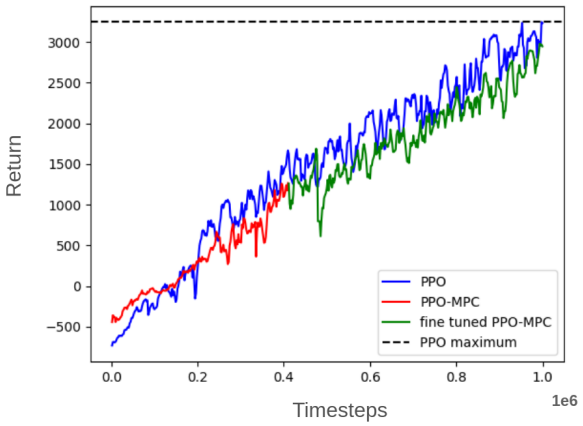


(b) Comparing returns of PPO and *PPO-MPC* architectures. The red line shows the returns achieved during training with model-based planning. After 4e5 timesteps, the blue line depicts *PPO-MPC* returns as the policy is fine-tuned with PPO

Figure 12: Plots to analyse the returns from *PPO-MPC* architecture in comparison to PPO and *MPC*. Half-Cheetah environment reward function is set to HeadPen



(a) Comparing returns of PPO, MPC and *PPO-MPC* architectures for 4e5 timesteps. The solid lines depict the average returns over past 10 episodes. The shaded region depicts the range between minimum and maximum returns over the past 10 episodes.



(b) Comparing returns of PPO and *PPO-MPC* architectures. The red line shows the returns achieved during training with model-based planning. After 4e5 timesteps, the blue line depicts *PPO-MPC* returns as the policy is fine-tuned with PPO

Figure 13: Plots to analyse the returns from *PPO-MPC* architecture in comparison to PPO and *MPC*. Half-Cheetah environment reward function is set to ShinPen

ward, it tends to fall very low and even flip to its back with the Hedpen reward. It also has the most non-uniform gait as is observed from its highly asymmetric gait signals. While both the default and ShinPen rewards show a fairly constant gait with similar vertical displacements in every gait cycle, the ShinPen reward has lower overall vertical COM displacement. Figure 16 shows the variation in the x coordinate of the half-cheetah's COM. Both figures 15 and 16, plot the respective COM coor-

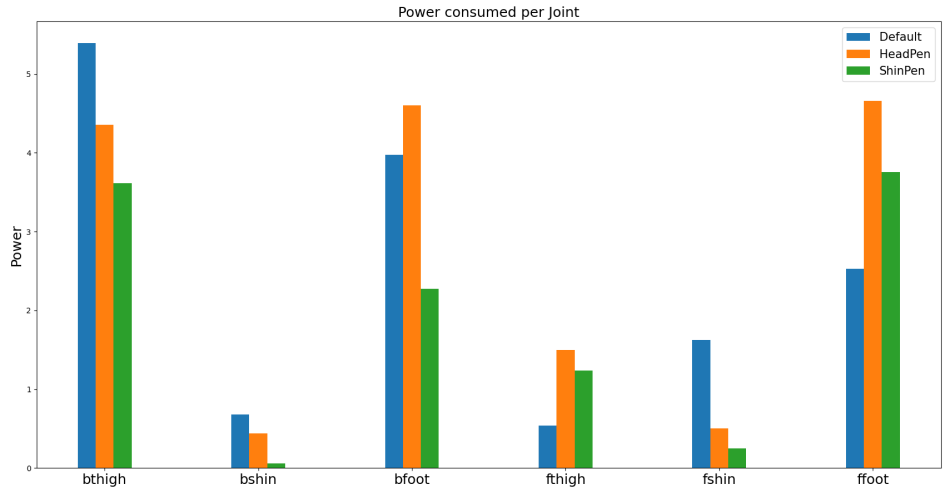


Figure 14: Comparing the power consumed by each joint for different reward functions

dinates for the same timesteps. This horizontal displacements indicates the forward movement of the half-cheetah. We notice that the half-cheetah moves much slower with the HeadPen reward function than with the other two. In 60 timsteps, the half-cheetah moves furthest with the ShinPen reward function.

7.1.4 Experiment 4

Finally, we test the applicability of our architecture when replacing PPO for a different model-free algorithm. In our case, we choose to replace PPO with SAC for the reason explained in section 5.4.4 and refer to this modified architecture as SAC-MPC. As seen in figure 17, our architecture enables the policy to achieve higher scores in fewer samples. With SAC-MPC, our policy achieves the SAC's maximum score in only 1.25e5 timesteps. SAC achieves the same score in 2e5 timesteps, giving SAC-MPC an upper hand with sample efficiency of 75e3 samples.

7.2 Discussion

7.2.1 Experiment 1

Based on our analyses of plot 8, training our architecture with a planning horizon length of 10 resulted in highest rewards. As already discussed, the goal of DRL is to optimise long horizon rewards rather than immediate ones. Had our learned model been an exact mimic of the real environment, we could have had a very long planning horizon. In fact, in such a situation, a simple planner would be sufficient to learn the policy. However, as we discussed, our model is only an approximation of the environment. Thus, with every step planned ahead in the learned model, the model error builds up in the collected trajectories. We observe this model error build up with longer horizon lengths, in the predictions for x position of the half-cheetah's centre of mass in plots 9. This leads to incorrect planning and consequently causes harm to our overall architecture performance. In our case, a horizon length of 25 seems to have this effect. Not only is our architecture's performance poor, as seen in plot 8, the model predictions for a planning horizon length of 25 is also unreliable, as seen in figure 9c.

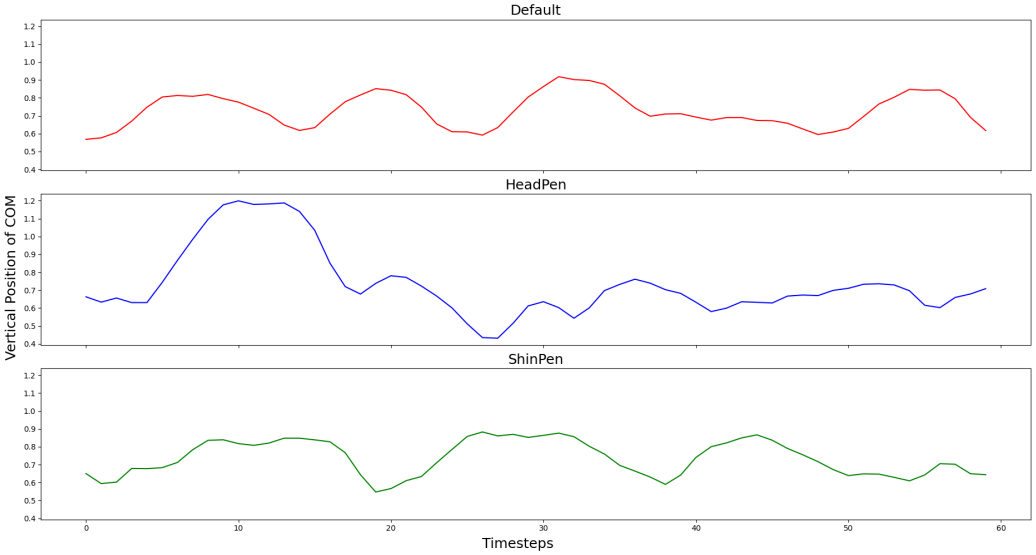


Figure 15: Comparing the gait stability for different reward functions

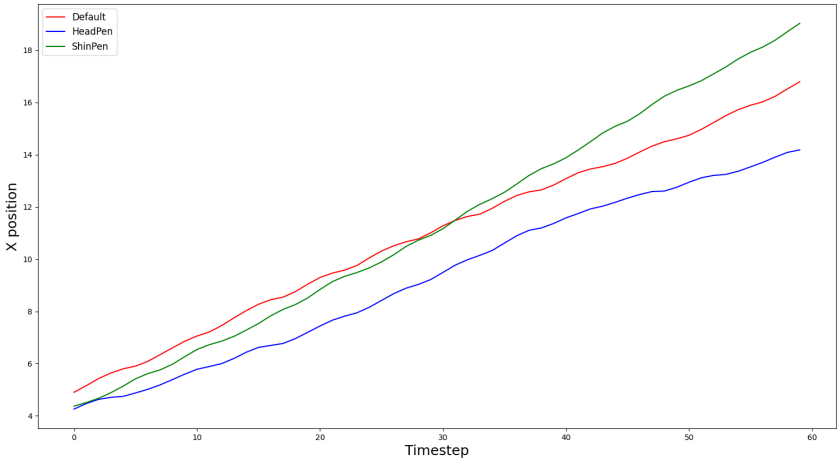


Figure 16: Comparing the reward functions based on the half-cheetah's motion of the centre of mass along the x axis

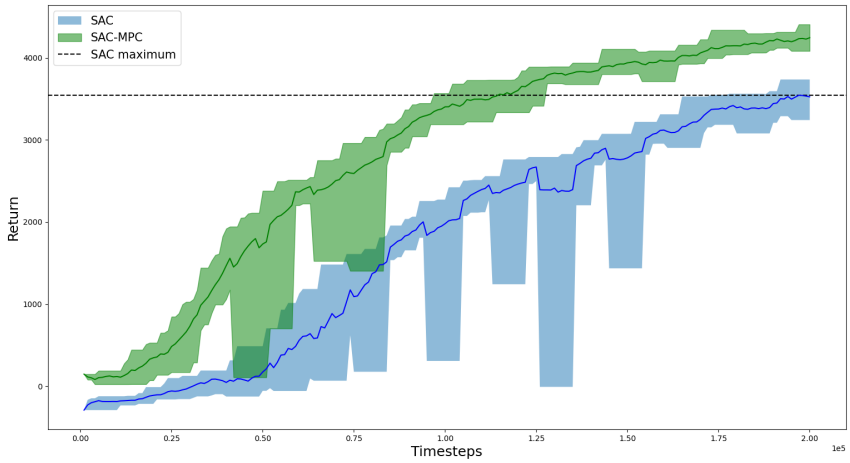


Figure 17: Architecture with SAC

The model prediction of the environment dynamics for $h = 25$, has a mean square error of 3.0342. Naturally, as we increase the horizon length, the mse for the model predictions of the environment dynamics increases. However, on increasing the planning horizon length from 5 to 10, we only notice an mse increase of 0.13. On the other hand, there is a noticeable increase in mse of 0.55 when increasing the horizon length from 10 to 25. Based on these observations, we conclude that 25 is too long a planning horizon for our model.

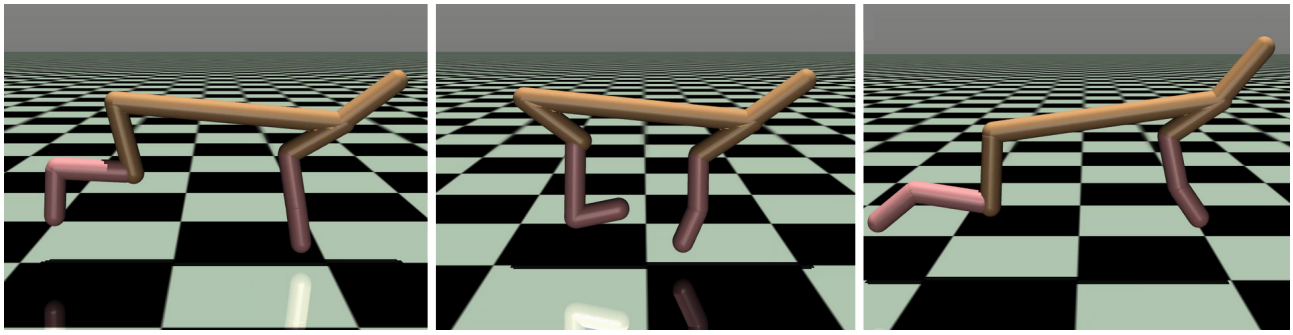
On the contrary, a horizon length of 5 is too short. During initial stages of training, planning with $h = 5$ shows a superior performance to using $h = 10, 25$. This indicates that when we set $h = 5$, the planning over our model is able to recognise good actions which when executed in the real environment, results in the agent encountering good experiences. These experiences are in turn used to train the policy. Better the trajectories, better our policy is trained. As our policy is completely random in the starting stage, even a short horizon optimisation helps it improve. However as the policy trains, the focus shifts to long term rewards rather than instant rewards. This entails that, our planner (MPC and PPO together) optimises actions over short trajectories of length 5, whereas our policy requires trajectories with better actions taken further into the future to improve. Thus, a planning horizon as short as 5 timesteps fails to be beneficial beyond around $1e5$ timesteps. This is the point where we notice the performance of the architecture with $h = 10$ overtake. With our model, a horizon length of 10 finds the right balance between circumventing model-error due to very long planning horizons and the myopic behaviour in planning with short horizons.

7.2.2 Experiment 2

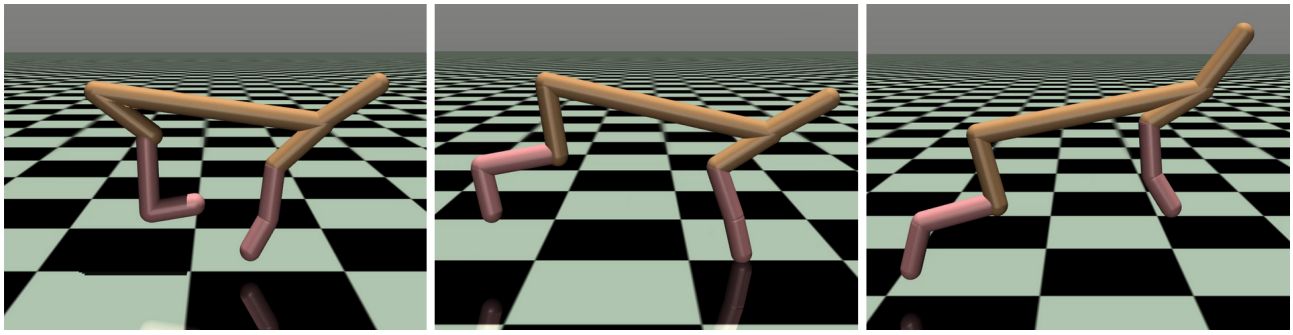
As we see in figure 11a, while the short horizon planning with *MPC* is beneficial in the early stages of training, the randomness of actions picked by MPC coupled with a short horizon planning fails to provide PPO with useful trajectories. Moreover, considering PPO clips large updates, actions chosen off-policy leads to further destabilisation of the PPO policy. With *PPO-MPC*, the chosen actions are directed on-policy and hence is keeping in line with PPO's principles. Thus, *PPO-MPC* shows better performance than *MPC*. It also gives higher scores than PPO at every timestep as while PPO sees

one among 15 possible actions on-policy, *PPO-MPC* sees the best among these 15 possible actions. From the large fluctuations in policy score for *PPO-MPC* beyond 3.5e5 timesteps, we acknowledge that although our architecture plans on policy, the short horizon planning may not benefit the policy in long run. It is to be noted that the policy optimises over an entire episode return and not just a short horizon of 10 timesteps. Thus the short horizon planning can only help the initial training stages. However, at 200 updates, *PPO-MPC* still achieves a return higher than PPO by 1000 points. Thus, instead of relying completely on model-based learning, we exploit the superior performance obtained over 4e5 timesteps by using the policy to warm start a pure model-free learning.

Figure 18 shows screenshots of the half-cheetah's gait, when following the policy trained using the *PPO-MPC* architecture. The screenshots here show the Cheetah ending the previous gait cycle and starting a new one. We see the cheetah starts from ground level, takes a leap using its hind leg and lands on its front leg. The leap results in the half-cheetah's body being propelled in the vertical direction too.



(a) timesteps of snapshots from left to right are 2.50s, 2.60s, 2.75s



(b) timesteps of snapshots from left to right are 2.90s, 3.10s, 3.30s

Figure 18: Screenshots of the half-cheetah following policy trained with *PPO-MPC* and the Default reward function

7.2.3 Experiment 3

As was observed in figure 12a and 13a, our architecture *PPO-MPC*, shows high sensitivity to the defined reward functions. In order to understand how these rewards effect the policy learning, we plotted the log of standard deviation for the policy networks attained when training with *PPO-MPC* architecture, for all the three defined reward functions. The plot is as shown in figure 19. The standard deviation is indicative of the uncertainty or stochastic nature of the policy. Higher the uncertainty,

more if the variation in predicting the actions. From figure 19, it's evident that the Default reward function shows very low uncertainty when compared to our experimental rewards. Intuitively this implies, if we sample 10 actions for a state using the policies trained with each of the reward functions, the actions sampled by the Default reward policy would be very similar to each other, while those samples by the HeadPen policy would differ the most from each other. Since the policy trained with ShinPen reward function shows high standard deviation than that trained with the default reward function, the actions would still differ amon each other to some extent. Since the stochasticity is high, the planner has a larger search space even when choosing actions on policy. Thus, the two experimental reward would require to be trained for far more timesteps to show good performance. It is also noteworthy that HeadPen reward has higher stochasticity than ShinPen. This ties up with our reasoning for our architecture achieving returns closer to that achieved with vanilla PPO training, when trained with ShinPen reward as compared to training with HeadPen reward function.

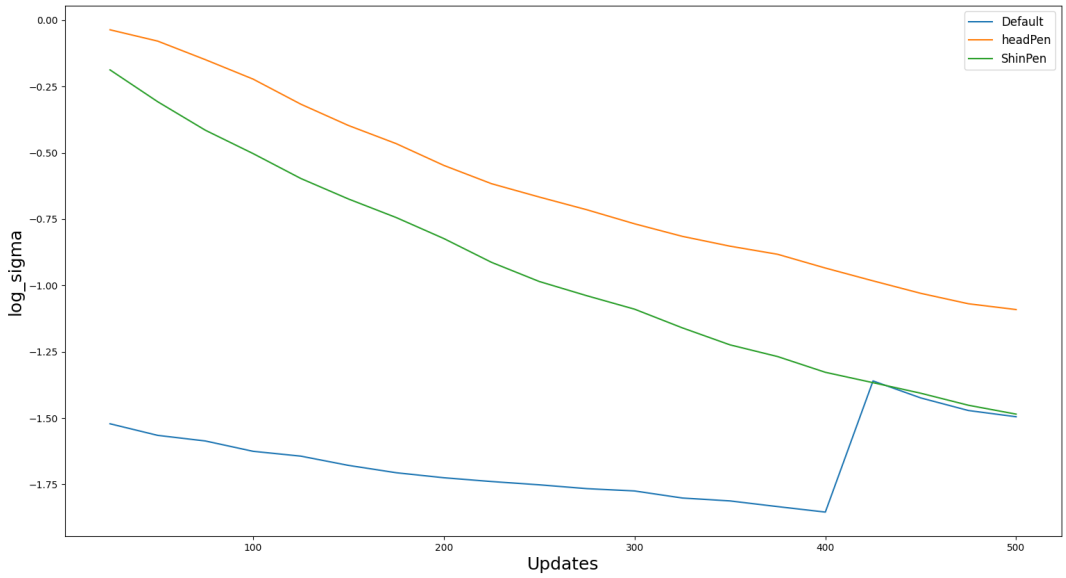


Figure 19: Plot showing how the log standard deviation for the policy networks varies during training with *PPO-MPC* architecture. The results are plotted for training with all three rewards functions

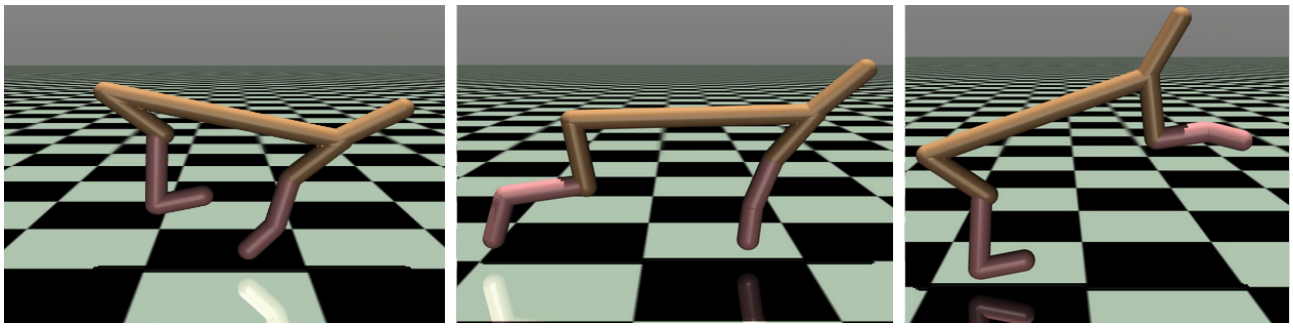
Figure 20 and figure 21 show screenshots for a the half-cheetah's gait cycles, when following a policy trained with our architecture using the HeadPen and ShinPen reward functions respectively. When comparing the screenshots in both cases, we notice that with the HeadPen reward, the half-cheetah jumps higher into the air. This is also supported by the larger vertical displacement of COM shown in figure 15 when compared to the other two rewards. Moreover, the high jump in this case results in an unstable landing position which sometimes even results in the half-cheetah tipping over. In most cases however, the half-cheetah drags its head for a few timesteps before re-stabilising itself and taking the next leap. Such an unstable gait makes it hard for the agent to move forward as we see in the horizontal displacement of its COM in figure 16. The ShinPen reward function attains a gait wherein, rather low leaps are taken during the gait. However, as we see from the screenshots in 21, these leaps are longer, resulting in lower displacement along the vertical and more displacement along the horizontal directions. These gait analysis are once gain supported by the plots for COM

(15, 16) displacement along y and x direction. Further, as seen from all experimental screenshots, the half-cheetah is propelled by the hind leg in making the leap. This requires a large force in the hind leg, supporting our observation in plot 14 showing highest power consumption by the back thigh.

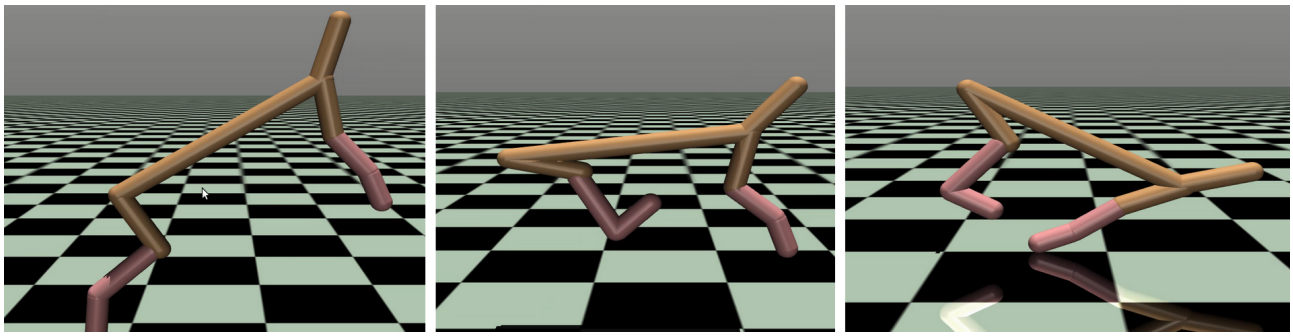
Based on the power consumption, stability in gait and distance moved forward, we conclude that the ShinPen reward produces the most efficient gait among the three that we compare, closely followed by the default reward. However, we notice that the returns achieved with ShinPen is lower than that attained with default reward. This only implies a stronger penalisation with ShinPen reward when compared to the Default. In spite of not penalising the torque applied, ShinPen reward is able to achieve a gait with minimal power consumed by the joints.

7.2.4 Experiment 4

As shown in figure 17, SAC-MPC can prove beneficial when training an offline policy. In fact, the lower uncertainty bound of our SAC-MPC policy is higher than the upper-bound of SAC, for most of the training. Traditionally, SAC uses a transition multiple times in updating the policy. With SAC-MPC, at each step we generate better transitions than we would with SAC and the benefit of each transition is exploited multiple times over training. We suppose, this could be a reason for SAC to benefit more from our architecture. This further goes to show that focused sampling of trajectories benefits learning as opposed to random sampling. Thus, even with model-free architectures that rely on many experiences to learn, the architecture, which is actor-critic, benefits from better quality of trajectories as opposed to larger quantity of trajectories.

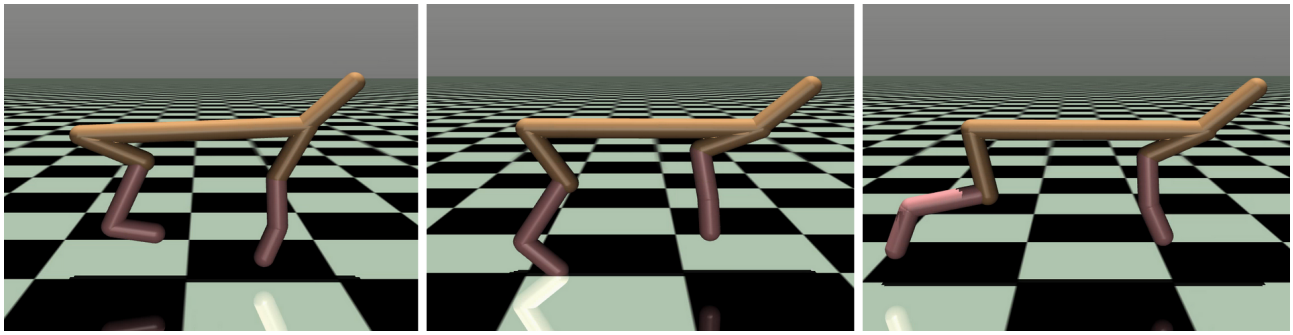


(a) timesteps of snapshots from left to right are 2.50s, 2.65s, 2.80s

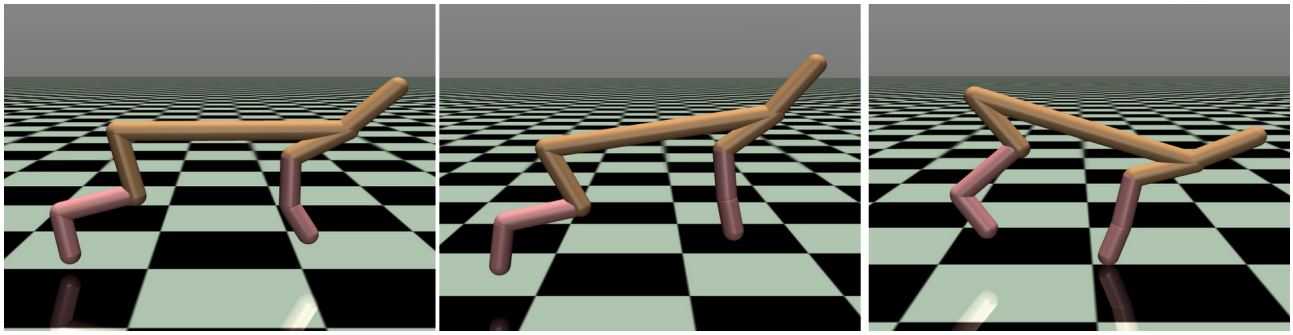


(b) timesteps of snapshots from left to right are 3.00s, 3.40s, 3.65s

Figure 20: Screenshots of the half-cheetah following policy trained with *PPO-MPC* and the HeadPen reward function



(a) timesteps of snapshots from left to right are 2.50s, 2.60s, 2.70s



(b) timesteps of snapshots from left to right are 3.20s, 3.55s, 3.60s

Figure 21: Screenshots of the half-cheetah following policy trained with *PPO-MPC* and the ShinPen reward function

8 Conclusion

The problem of sample complexity is one that has challenged the application of DRL in robotics. This paper introduces a novel architecture combining model-based and model-free learning. With this architecture we aim to combine the sample efficiency of model-based methods with the performance superiority of model-free. With model-based approaches, MPC planners are popularly used in planning ahead. With our architecture, we propose to use a policy that directs the actions selected by an MPC as opposed to using a naive MPC with random action selection. Our architecture is evaluated on the Half-Cheetha environment. We further test its sensitivity to custom designed rewards and also test the quality of these rewards. Finally, we test the applicability of our architecture when replacing PPO with an off-policy algorithm such as SAC, as an attempt to generalise the architecture to other MF algorithms.

Based on the observations and analysis of our experiments, we conclude the following with respect to our research questions:

- As seen in Experiment 2, our MBMF architecture shows promise in solving RL tasks while being sample efficient when compared to its model-free counterpart. Our architecture achieves a sample efficiency of 1e5 timesteps compared to vanilla PPO on solving the Half-Cheetah environment. This is an improvement over the 1e4 timesteps of sample efficiency achieved by [15], that we take inspiration from. However, with further experimentation we notice that this success is dependent on the quality of the reward and highly sensitive to hyper-parameter tuning. Specifically, we found that the planning horizon length had a prominent impact on our architectures performance as explained in section 7.1.1.

- We find that the policy directed action selection (PPO+MPC or SAC+MPC) always works better than using a naive MPC planner. Thus, our intuition of using an in-loop trained policy to constraint the action space proves to be advantageous.

- Although our architecture can achieve sample efficiency, it is highly sensitive to the rewards. On analysing the rewards experimented with, we also noticed that our architecture benefits specifically when the policy network stochasticity is considerably lower. However, we suspect that increasing the number of trajectories computed by our planner at each timestep could improve our architecture's performance. Further, we studied the quality of these rewards based on the stability of the gait and power utilised. Based on our analysis metrics, the reward defined based on the leg movement (ShinPen) achieved the most stable gait with minimum vertical displacement of COM and minimum power utilisation in joints.

- In order to generalize our architecture to other MF algorithms, we also tested our architecture with SAC replacing the PPO component. We found that planning with MPC and an in-loop training policy can achieve sample efficiency even with off-policy training algorithms. Specifically, on replacing PPO with SAC, we found that our architecture achieves a sample efficiency of 75e3 samples on solving the Half-Cheetah environment.

In conclusion, our intuition of combining model-free algorithm with MPC in planning over a learned model shows superiority over planning with a naive MPC. However, the overall performance of our architecture is highly sensitive to horizon length, number of planned trajectories and quality of reward.

8.1 Limitations and Future Work

This research focuses on introducing the idea of using a policy trained in-loop to direct actions with an MPC planner. While, the proposed idea does show promise, it doesn't always guarantee the sample efficiency we hope for. The most prominent drawback that we found was the sensitivity of our architecture to the quality of the reward. Further, we implemented a 2 layer feed-forward neural network in capturing the dynamics of the environment. The loss function off this network only optimizes over one step predictions. Thus, there is a gap in what our neural network optimizes and what we apply the neural network to. Model-based approaches often tend to fail in complex environments with high non-linearity. In an attempt to evaluate this architecture on a musculoskeletal model with 17 muscles and 2 actuators, we faced the challenge of modelling the dynamics first hand.

For future works, we propose dynamically changing the model-based hyper-parameters. The sensitivity of our architecture to reward functions could be a result of requiring more exploration at certain stage of training. Dynamically changing the number of trajectories to be computed at a time based on the learned policy's uncertainty could potentially combat this obstacle. Further, although we are able to prove the relevance of planning with MPC and PPO rather than MPC alone, the short horizon planning doesn't achieve results of pure model-free learning. To promote long horizon planning, we must build a model that is able to learn the environment dynamics more accurately. In future works, we aim to use an ensemble of probabilistic models as in [48] in capturing the environment's aleatoric and epistemic uncertainties, allowing one to plan for longer horizons. Further, the uncertainty in model prediction could be included in dynamically setting the planning horizon length. Specifically, given that our learned dynamic model can output a prediction for the next state and a value indicating its certainty of the predictions, the planner only plans upto a horizon where the uncertainty in prediction is tolerable. In conclusion, the proposed architecture does show promise but is highly sensitive. In order to make it more robust and generalized, the hyper-parameter tuning must be strategically conducted.

Bibliography

[1] O.-L. Ouabi, P. Pomarede, N. Declercq, N. Zeghidour, M. Geist, C. Pradalier, N. F. Declercq, and C. Edric Pradalier, "Learning the Propagation Properties of Plate-like Structures for Lamb Wave-based Mapping Learn-ing the Propagation Properties of Plate-like Structures for Lamb Wave-based Mapping Learning the Propagation Properties of Plate-like Structures for Lamb Wave-b," *Ultrasonics*, vol. ..., no. ..., p. 106705, 2022.

[2] A. Chella, L. Iocchi, I. Macaluso, and D. Nardi, "Artificial Intelligence and Robotics.," *Intelligenza Artificiale*, vol. 3, pp. 87–93, jan 2006.

[3] O. Kilinc and G. Montana, "Reinforcement learning for robotic manipulation using simulated locomotion demonstrations," *Machine Learning*, vol. 111, no. 2, pp. 465–486, 2022.

[4] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter, "Learning agile and dynamic motor skills for legged robots," *CoRR*, vol. abs/1901.08652, 2019.

[5] M. Taylor, S. Bashkirov, J. F. Rico, I. Toriyama, N. Miyada, H. Yanagisawa, and K. Ishizuka, "Learning bipedal robot locomotion from human movement," 2021.

[6] C. Yu and A. Rosendo, "Multi-modal legged locomotion framework with automated residual reinforcement learning," 2022.

[7] R. A. Garza Bayardo, "Dynamic locomotion for humanoid robots via deep reinforcement learning," 2022.

[8] Z. Ding and H. Dong, "Challenges of Reinforcement Learning," pp. 249–272, jun 2020.

[9] P. Trentsios, M. Wolf, and D. Gerhard, "Overcoming the Sim-to-Real Gap in Autonomous Robots," *Procedia CIRP*, vol. 109, pp. 287–292, 2022.

[10] W. Zhao, J. Peña Queralta, Q. L., and T. Westerlund, "Towards closing the sim-to-real gap in collaborative multi-robot deep reinforcement learning," 11 2020.

[11] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," pp. 1–12, 2017.

[12] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel, "Trust region policy optimization," *32nd International Conference on Machine Learning, ICML 2015*, vol. 3, pp. 1889–1897, 2015.

[13] V. Feinberg, A. Wan, I. Stoica, M. I. Jordan, J. E. Gonzalez, and S. Levine, "Model-Based Value Estimation for Efficient Model-Free Reinforcement Learning," 2018.

[14] Y. Fan and Y. Ming, "Efficient Exploration for Model-based Reinforcement Learning with Continuous States and Actions," nov 2020.

[15] A. Nagabandi, G. Kahn, R. S. Fearing, and S. Levine, "Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning," in *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 7579–7586, aug 2018.

[16] A. Plaat, W. Kosters, and M. Preuss, "Deep Model-Based Reinforcement Learning for High-Dimensional Problems, a Survey," aug 2020.

[17] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," *IEEE International Conference on Intelligent Robots and Systems*, pp. 5026–5033, 2012.

[18] M. Schwenzer, M. Ay, T. Bergs, and D. Abel, "Review on model predictive control: an engineering perspective," *International Journal of Advanced Manufacturing Technology*, vol. 117, no. 5-6, pp. 1327–1349, 2021.

[19] T. Wang, X. Bao, I. Clavera, J. Hoang, Y. Wen, E. Langlois, S. Zhang, G. Zhang, P. Abbeel, and J. Ba, "Benchmarking Model-Based Reinforcement Learning," pp. 1–25, 2019.

[20] M. Janner, J. Fu, M. Zhang, and S. Levine, "When to trust your model: Model-based policy optimization," *Advances in Neural Information Processing Systems*, vol. 32, no. NeurIPS, 2019.

[21] E. F. Morales and J. H. Zaragoza, "An introduction to reinforcement learning," *Decision Theory Models for Applications in Artificial Intelligence: Concepts and Solutions*, pp. 63–80, 2011.

[22] A. Toyama, K. Katahira, and H. Ohira, "Biases in estimating the balance between model-free and model-based learning systems due to model misspecification," *Journal of Mathematical Psychology*, vol. 91, pp. 88–102, 2019.

[23] L. Baird and A. Moore, "Gradient Descent for General Reinforcement Learning," in *Advances in Neural Information Processing Systems* (M. Kearns, S. Solla, and D. Cohn, eds.), vol. 11, MIT Press, 1998.

[24] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," *Society for Industrial and Applied Mathematics*, vol. 42, 04 2001.

[25] S. Ruder, "An overview of gradient descent optimization algorithms," pp. 1–14, 2016.

[26] C. Baroglio, A. Giordana, M. Kaiser, M. Nuttin, and R. Piola, "Learning controllers for industrial robots," *Machine Learning*, vol. 23, no. 2, pp. 221–249, 1996.

[27] K. S. Fu, *Learning Control Systems*, pp. 251–292. Boston, MA: Springer US, 1969.

[28] T. Wang and J. Ba, "Exploring Model-based Planning with Policy Networks," jun 2019.

[29] G. Williams, N. Wagener, B. Goldfain, P. Drews, J. M. Rehg, B. Boots, and E. A. Theodorou, "Information theoretic MPC for model-based reinforcement learning," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 1714–1721, 2017.

[30] P. Bakaráč and M. Kvasnica, "Fast nonlinear model predictive control of a chemical reactor: a random shooting approach," *Acta Chimica Slovaca*, vol. 11, no. 2, pp. 175–181, 2018.

[31] K. Fedorová, P. Bakaráč, and M. Kvasnica, "Agile manoeuvres using model predictive control," *Acta Chimica Slovaca*, vol. 12, no. 1, pp. 136–141, 2019.

[32] I. Kivi, "Online Planning Based Reinforcement Learning for Robotics Manipulation," 2019.

[33] OpenAI, "Openai baselines."

[34] W. Fedus, P. Ramachandran, R. Agarwal, Y. Bengio, H. Larochelle, M. Rowland, and W. Dabney, "Revisiting fundamentals of experience replay," *37th International Conference on Machine Learning, ICML 2020*, vol. PartF16814, pp. 3042–3052, 2020.

[35] *Robotic Grasping Training Using Deep Reinforcement Learning With Policy Guidance Mechanism*, vol. Volume 2: of *International Manufacturing Science and Engineering Conference*, 2021.

[36] M. Kasaei, M. Abreu, N. Lau, A. Pereira, and L. P. Reis, "Robust biped locomotion using deep reinforcement learning on top of an analytical control approach," *Robotics and Autonomous Systems*, vol. 146, p. 103900, 2021.

[37] S. Delp, F. Anderson, A. Arnold, P. Loan, A. Habib, C. John, E. Guendelman, and D. Thelen, "OpenSim: Open-Source Software to Create and Analyze Dynamic Simulations of Movement," *Biomedical Engineering, IEEE Transactions on*, vol. 54, pp. 1940–1950, 2007.

[38] G. Chen, L. Pan, Y. Chen, P. Xu, Z. Wang, P. Wu, J. Ji, and X. Chen, "Robot Navigation with Map-Based Deep Reinforcement Learning," pp. 1–6, 2020.

[39] M. P. Deisenroth, "A Survey on Policy Search for Robotics," *Foundations and Trends in Robotics*, vol. 2, no. 1-2, pp. 1–142, 2011.

[40] M. P. Deisenroth and C. E. Rasmussen, "PILCO: A model-based and data-efficient approach to policy search," *Proceedings of the 28th International Conference on Machine Learning, ICML 2011*, pp. 465–472, 2011.

[41] K. J. Hunt, D. Sbarbaro, R. Zbikowski, and P. J. Gawthrop, "Neural networks for control systems-A survey," *Automatica*, vol. 28, no. 6, pp. 1083–1112, 1992.

[42] M. Watter, J. T. Springenberg, J. Boedecker, and M. Riedmiller, "Embed to control: A locally linear latent dynamics model for control from raw images," 2015.

[43] Y. Gal and Z. Ghahramani, "Dropout as a Bayesian approximation: Representing model uncertainty in deep learning," *33rd International Conference on Machine Learning, ICML 2016*, vol. 3, pp. 1651–1660, 2016.

[44] R. S. Sutton, "Integrated Modeling and Control Based on Reinforcement Learning and Dynamic Programming," in *Advances in Neural Information Processing Systems* (R. P. Lippmann, J. Moody, and D. Touretzky, eds.), vol. 3, Morgan-Kaufmann, 1990.

[45] R. S. Sutton, "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming," in *Machine Learning Proceedings 1990* (B. Porter and R. Mooney, eds.), pp. 216–224, San Francisco (CA): Morgan Kaufmann, 1990.

[46] W. Lee and F. A. Oliehoek, "Analog circuit design with dyna-style reinforcement learning," 2020.

[47] J. Peng and R. Williams, "Efficient Learning and Planning Within the Dyna Framework," *Adaptive Behavior*, vol. 1, 1998.

[48] K. Chua, R. Calandra, R. McAllister, and S. Levine, "Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models," in *Advances in Neural Information Processing Systems*, vol. 2018-Decem, pp. 4754–4765, 2018.

[49] H. Bharadhwaj, K. C. Xie, and F. Shkurti, "Model-Predictive Control via Cross-Entropy and Gradient-Based Optimization," no. 2016, pp. 1–11.

[50] E. Bøhn, S. Gros, S. Moe, and T. A. Johansen, "Optimization of the Model Predictive Control Update Interval Using Reinforcement Learning∗∗This work was financed by grants from the Research Council of Norway (PhD Scholarships at SINTEF grant no. 272402, and NTNU AMOS grant No. 223254).," *IFAC-PapersOnLine*, vol. 54, no. 14, pp. 257–262, 2021.

[51] M. Schwenzer, M. Ay, T. Bergs, and D. Abel, "Review on model predictive control: an engineering perspective," *The International Journal of Advanced Manufacturing Technology*, vol. 117, no. 5, pp. 1327–1349, 2021.

[52] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *35th International Conference on Machine Learning, ICML 2018*, vol. 5, pp. 2976–2989, 2018.