TU WIEN Informatics

Modeling System Dynamics In Partially-Observable Environments Using Biologically-Inspired Recurrent Neural Networks

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Mihai-Teodor Stănușoiu, BSc.
Matrikelnummer 12118533

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.rer.nat. Dr.h.c. Radu Grosu

Mitwirkung: Mónika Farsang, MSc.

Wien, 5. Mai 2025

Mihai-Teodor Stănușoiu | Radu Grosu

TU WIEN Informatics

Modeling System Dynamics In Partially-Observable Environments Using Biologically-Inspired Recurrent Neural Networks

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Mihai-Teodor Stănușoiu, BSc.
Registration Number 12118533

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.rer.nat. Dr.h.c. Radu Grosu

Assistance: Mónika Farsang, MSc.

Vienna, May 5, 2025

Mihai-Teodor Stănușoiu	Radu Grosu

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Mihai-Teodor Stănușoiu, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 5. Mai 2025

Mihai-Teodor Stănușoiu

Acknowledgements

First and foremost, I would like to thank Prof. Radu Grosu for being an exemplary teacher. His lectures ignited my passion for Reinforcement Learning, and his relentless scientific curiosity continues to inspire me to always remain hungry for knowledge. I am also deeply grateful for Mónika Farsang's supervision throughout my thesis. Her knowledge of and insights into Machine Learning were tremendously valuable. Her assistance helped me persevere through the most challenging stages of the thesis process. It is hard to wish for a more attentive and genuinely helpful supervisor, and I hope to have as many future collaborations as possible.

I am also very grateful to be part of the Scuderia Segfault racing team. Integrating knowledge gathered throughout my studies for such a practical and entertaining application as autonomous racing has been a highlight of my Master's degree. Participating in international competitions has provided me with valuable practical experience and cherished memories with the team and participants from other universities. More importantly, I want to thank Andreas Brandstätter and Felix Resch for assembling the team, for fostering a collaborative environment, for inspiring me with their meticulousness and passion, and for representing the soul of the team. I extend the same gratitude to all present and former team members: Luigi Berducci, Agnes Poks, Moritz Christamentl, Philipp Mandl, Mónika Farsang, Mihaela-Larisa Clement, Elisa Di Cristo, Philipp Gratzer, Piet Kaul, Samuel Lechner, Nino Wegleitner.

I would also like to highlight my appreciation for the Master's studies offered by TU Wien. Each lecture has been insightful, and I am grateful for the organization and effort of all the professors, assistants, and organizational staff involved.

Last but not least, I would like to thank my family and Mihaela-Larisa Clement for being supportive of my studies and thesis work. And thank you, Larisa, for being by my side at every step of the way.

Kurzfassung

Selbstlernende End-to-End-Agenten sind eine beliebte Wahl für kontinuierliche Regelungsaufgaben. Anstatt eine Pipeline für Zustandsschätzung und Planung zu entwerfen, konzentriert sich ein Teilbereich der Reinforcement-Learning-Forschung auf die Entwicklung robuster, lernbasierter Regelungen. Moderne Ansätze haben sich bei der Bewältigung von Regelungs problemen in simulierten Umgebungen und einfachen realen Aufgaben bewährt. Diese Leistung erfordert jedoch häufig ausgewählte, geräuschfreie Beobachtungen und kontrollierte Laborbedingungen. Erfolgreiche, autonome Agenten, die in der realen Welt eingesetzt werden, müssen in der Lage sein, mit Unsicherheiten umzugehen, die durch partielle Beobachtbarkeit, Sensorrauschen und Verzögerungen entstehen.

Diese Arbeit untersucht einen Ansatz zur Modellierung von Zustandsrepräsentationen basierend auf aufgezeichneten Trajektorien für Agenten mit Reinforcement Learning, die in teilweise beobachtbaren Prozessen arbeiten. Liquid-Time-Constant Neural Networks (LTC) und Closed-form Continuous-time Neural Networks (CfC) verarbeiten die Sequenz vergangener Beobachtungen und Aktionen, um annähernd markovsche Zustände aus unzuverlässigen Eingangsdaten zu kodieren. Die Darstellungen werden in den modellbasierten TD-MPC2-Rahmen integriert, um kontinuierliche Regelungs probleme mit unvollständiger Information zu bewältigen. Die Arbeit führt drei verschiedene Weltmodellformulierungen ein, die den ursprünglichen TD-MPC2-Ansatz erweitern: (1) ein deterministisches Beobachtungs-Vorhersagemodell, (2) ein stochastisches Beobachtungs-Vorhersagemodell und (3) ein latentes Zustands-Vorhersagemodell

Die vorgeschlagenen Methoden werden anhand ausgewählter Standard-Benchmarks für kontinuierliche Regelungs simulationen (Cartpole, Acrobot, Walker) evaluiert, die so angepasst wurden, dass sie Verdeckungen, Sensorrauschen und Zeitverzögerungen induzieren. Die experimentellen Ergebnisse zeigen, dass die Ausstattung von TD-MPC2 mit History-Encodern, die in der Lage sind, komplexe Dynamiken zu erfassen, die Robustheit unter Sensorrauschen deutlich verbessert und eine optimale Leistung ohne Zugang zu Geschwindigkeitsmessungen erreicht. Insbesondere löst einer der vorgeschlagenen Ansätze die schwierige, unteraktuierte Akrobot-Schwenkaufgabe optimal, im Gegensatz zur TD-MPC2-Basislösung. Darüber hinaus deuten vorläufige Ergebnisse darauf hin, dass die Methoden das Potenzial haben, Zustände aus unregelmäßigen Beobachtungen zu erfassen, was ihr Potenzial für den Einsatz in der realen Welt mit stochastischen Zeitverzögerungen hervorhebt.

Abstract

End-to-end self-learning agents are a popular and desirable choice for continuous control settings. Instead of designing a pipeline of state estimation and planning, a subfield of Reinforcement Learning research focuses on designing robust, learning-based controllers. State of the art approaches have proven capable of tackling control problems in simulated environments and simple real-world tasks. However, such performance often requires curated, noise-free observations and controlled laboratory settings. Successful, autonomous agents deployed in the real world must be capable of dealing with uncertainty induced by partial observability, sensor noise and delays.

This work explores an approach to modeling history-based state representations for reinforcement learning agents operating in partially observable processes. Liquid Time-Constant Neural Networks (LTC) and Closed-form Continuous-time Neural Networks (CfC) process the sequence of past observations and actions to encode approximately Markovian states from unreliable input data. The representations are integrated into the model-based TD-MPC2 framework to tackle continuous control problems with incomplete information. The work builds on theoretical foundations on approximate information states and state abstractions from histories to introduce three different world model formulations: (1) a deterministic observation-predictive model, (2) a stochastic observation-predictive model, and (3) a latent state-predictive model, extending the original TD-MPC2 approach.

The proposed methods are evaluated on selected standard, continuous control simulation benchmarks (Cartpole, Acrobot, Walker), adapted to induce occlusions, sensor noise, and time delays. Experimental results show that equipping TD-MPC2 with history encoders capable of capturing complex dynamics significantly improves robustness under sensor noise and achieves optimal performance without access to velocity measurements. In particular, one of the proposed approaches optimally solves the challenging, underactuated Acrobot swingup task, unlike the baseline TD-MPC2. Furthermore, preliminary results suggest that the methods show potential in capturing states from irregularly timed observations, highlighting their potential for real-world deployment with stochastic time delays.

Contents

CHAPTER 1

Introduction

Designing controllers for tasks such as object manipulation and locomotion under complex system dynamics typically demands expert knowledge across various areas, including system modeling, state estimation, and trajectory optimization. Furthermore, the computational and memory constraints of robotics platforms limit solver runtime, requiring careful trade-offs between model complexity and optimization horizon length.

Recent state-of-the-art Deep Reinforcement Learning (DRL) approaches offer the potential for end-to-end control at higher frequencies than traditional feedback controllers like Model Predictive Control (MPC) can achieve, while avoiding manual feature engineering [LMA20], [WZW+23]. However, real-time control systems still face challenges such as unreliable sensor data, compute limitations, delays, and external disturbances. Thus, it is desirable to design algorithms that:

1. Reconstruct the system state from discrete, noisy perception data,
2. Train agents capable of data-efficient planning under real-world constraints.

This work studies the use of Liquid Neural Networks (LNN) to reconstruct a Markovian state space from partial observations. Liquid Time-Constant Networks (LTCs) [HLA+21] model input- and state-dependent adaptive time-constants at the neuron level, resulting in a coupled ODE system. Their architecture approximates the membrane potential dynamics of non-spiking neurons, inspired by the neural circuitry of small organisms such as *C. elegans*. Their expressivity and adaptability allow stable modeling of complex non-linear dynamics, a key challenge in the early stages of RL training. Moreover, their input-adaptive responses may confer robustness at inference, even in unseen conditions. As continuous-time recurrent networks, they enable agents to solve continuous-time control tasks with arbitrary timestep spacing, unlike traditional discrete-time RL algorithms, which struggle with time irregularities [YHL21]. Their closed-form variant, Closed-form

Continuous-time Neural Networks (CfC) [HLA+22], offers faster training and inference while preserving the expressivity of LTCs, whose ODE solver-based forward passes incur higher runtime costs.

Building on the theory of approximate information states [SSSM22] and abstract state representations from histories [LWL06, CPP09, NES+], this thesis extends the model-based RL algorithm TD-MPC2 [HSW], designed for continuous control tasks. TD-MPC2 employs a learned model of the agent and environment for short-horizon planning and long-term value estimation, but models state transitions with an MLP and does not explicitly address Partially Observable Markov Decision Processes (POMDP). This work leverages LNNs to construct Markovian state representations from sequences of noisy observations and past actions, proposing two abstraction models: an observation-trajectory predictive model and a latent-trajectory predictive model.

The thesis is structured as follows. Chapter 2 reviews relevant background, including an overview of Machine Learning (ML), neural networks for dynamical systems modeling (focusing on LTC and CfC), and reinforcement learning fundamentals from tabular methods to modern DRL, Model-Based Reinforcement Learning (MBRL), and POMDPs. Chapter 4 describes the proposed LNN-based extensions to TD-MPC2, detailing the problem settings, method architectures, and theoretical foundations for value estimation from encoded states. Chapter 5 defines hypotheses, methodology, and experimental setups, followed by evaluations of expressivity, robustness, sample efficiency, and overall performance in partially observable settings. Finally, Chapter 6 summarizes findings, discusses limitations, and outlines directions for future research.

CHAPTER 2

Background

The purpose of this chapter is to cover the theoretical concepts used throughout Chapter 4 and 5. Section 2.1 provides an introduction to relevant probability and information theory concepts. The following two sections, Section 2.2 and 2.3, cover machine learning basics, with a focus on continuous-time neural networks for modeling system dynamics such as LTC and CfC. Section 2.4 on Reinforcement Learning (RL) provides an overarching view of the theoretical foundation of self-learning agents, followed by specific approaches that learn an environment model from noisy data. Sections 2.4.1 and 2.4.2 are largely based on the seminal book on RL [Bar21], which defines much of the foundation of the RL field of research.

2.1 Probability and Statistics Overview

Real-time control of dynamical systems with learning-based approaches features uncertainty at multiple levels. These sources can be categorized as follows:

1. Observable data uncertainty: Controller inputs are typically derived from sensor data, which are inherently noisy.

2. Control uncertainty: A learned or identified model of the system is never exact and thus, it may not perfectly capture the system dynamics, introducing uncertainty in the control methods.

3. Inherent process stochasticity: Dynamical systems, or sequential processes, can be inherently stochastic in nature.

2.1.1 Information Theory for Random Variables and Their Probability Distribution

When dealing with uncertainty, it is useful to apply formal framework provided by *probability theory* in order to quantify randomness in data. A *random variable* is a variable whose possible values are determined by a probability distribution. For example, assume variable x can take values v_1, v_2. A probability distribution would indicate the likelihood of $x = v_1$ and the likelihood of $x = v_2$. This is an example of a *discrete* random variable, since the value domain over which it is defined is finite. However, a random variable defined over the set of natural numbers $\mathbb{N}$ would also be discrete, since $\mathbb{N}$ is countably infinite. When a random variable is defined over a real-valued domain $S^n \subseteq \mathbb{R}^{\kappa}, n \geq 0$, it is *continuous*.

A probability distribution over a discrete variable X with values in $\{x\}_{1:n} \coloneqq \mathcal{X}$ is defined using a *probability mass function* $\mathbb{P} : \mathcal{X} \to [0, 1]$. The probability mass function maps each possible value of X to the probability of that value. It can also be denoted as $P(X)$. The shorthand notation $\mathbb{P}(X = x) \coloneqq P(x)$ can also be used. Probabilities are valued in the range $[0, 1]$, with $\mathbb{P}(X = x) = 0$ meaning that X can never take the value x, and $\mathbb{P}(X = x) = 1$ meaning that X is always x. This implies that for any other $x' \neq x$, $\mathbb{P}(X = x') = 0$. It is also impossible for $\mathbb{P}(X = x) = 0$ for all $x \in \mathcal{X}$. It is thus intuitive that all probabilities sum to 1:

$$\sum_{x \in \mathcal{X}} \mathbb{P}(X = x) = 1 \tag{2.1}$$

A similar function can be defined for continuous variables. The *probability density function* of a probability distribution over a continuous variable X with values in range $[a, b]$ is defined as $\mathbb{P} : [a, b] \to [0, 1]$ and satisfies:

$$\int_a^b \mathbb{P}(X = x)dx = 1 \tag{2.2}$$

Probability distributions over multiple variables can also be defined. A set of independent random variables $\{X\}_{1:m}$ is arranged according to a *joint probability distribution* $\mathbb{P}(X_1, \ldots, X_m)$ and also satisfies (2.1) or (2.2), depending on the nature of the random variables. Joint probability distributions can be expressed as joint probability distributions over a subset of the variables. The relation is called *marginalization*. Let $\mathbb{P}(X_1, \ldots, X_m)$ be a joint probability distribution. The *marginal probability* distribution over subset $\{X_2, \ldots, X_m\}$ can be expressed in terms of $\mathbb{P}(X_1, \ldots, X_m)$ as:

$$\mathbb{P}(X_2, \ldots, X_m) = \sum_{x_1} \mathbb{P}(x_1, X_2, \ldots, X_m) \tag{2.3}$$

Values of random variables in the context of probability distributions are also called *states* of the random variable, and the evaluation of a random variable $X = x$ is also

called an *event*. An independence relation between two events $\perp \in \{(X, Y)\}$ is defined as:

$$X \perp Y \Leftrightarrow \mathbb{P}(X, Y) = \mathbb{P}(X) \cdot \mathbb{P}(Y) \tag{2.4}$$

Previously, $\{X_1, \ldots, X_m\}$ have been assumed to be pairwise *independent*. Let X be an event dependent on another event Y. *Conditional probability* distributions provide the likelihood of X given Y and is denoted as $\mathbb{P}(X|Y)$. For two random variables, conditional probability distributions are related to joint probability distributions by the following equality:

$$\mathbb{P}(X|Y) = \frac{\mathbb{P}(X, Y)}{\mathbb{P}(Y)} \tag{2.5}$$

A relation also exists for multiple random variables. The *chain rule of probability distributions* is a recursively-applying expression of a joint probability distribution as conditional probability distributions in the following manner:

$$\begin{aligned} \mathbb{P}(X_1, \ldots, X_m) &= \mathbb{P}(X_1|X_2, \ldots, X_m) \cdot \mathbb{P}(X_2, \ldots, X_m) \\ &= \mathbb{P}(X_1|X_2, \ldots, X_m) \cdot \mathbb{P}(X_2|X_3, \ldots, X_m) \cdot \mathbb{P}(X_3, \ldots, X_m) \\ &\ldots \\ &= \mathbb{P}(X_1|X_2 \ldots X_m) \cdot \mathbb{P}(X_2|X_3, \ldots, X_m) \cdot \cdots \cdot \mathbb{P}(X_m) \end{aligned} \tag{2.6}$$

The labeling order of the variables $1, \ldots, m$ does not matter.

In some cases, it is desirable to infer the probability of a hypothesis Y given some evidence $X\mathbb{P}(Y|X)$, and a prior distribution over the hypothesis $\mathbb{P}(Y)$. *Bayes' theorem* can be applied to infer the posterior distribution, given the evidence probability $\mathbb{P}(X)$ and the likelihood of observing the evidence if the hypothesis holds $\mathbb{P}(X|Y)$:

$$\mathbb{P}(Y|X) = \frac{\mathbb{P}(X|Y)\mathbb{P}(Y)}{\mathbb{P}(X)} \tag{2.7}$$

An important characteristic of random variables for quantifying uncertainty is their *entropy*. From the point of view of *information theory*, random variables carry information according to their probability distribution. A deterministic variable (where $\exists! x \in \mathcal{X} : X = x$) carries little information, whereas a truly random variable is more informative. In other words, the former has low entropy, while the latter has high entropy. The entropy of a discrete random variable is defined as:

$$H(X) = -\sum_{x \in \mathcal{X}} \mathbb{P}(x) \cdot \log \mathbb{P}(x) \tag{2.8}$$

where $-\log \mathbb{P}(x)$ is also called *self-information* function of an event. The entropy is similarly defined for continuous random variables:

$$H(X) = -\int_a^b \mathbb{P}(x) \cdot \log \mathbb{P}(x) dx \tag{2.9}$$

The *expected* value of a random variable is the mean of possible values, weighed by their likelihood. It is denoted as $\mathbb{E}[X]$ and defined as:

$$\mathbb{E}[X] = \sum_{x \in \mathcal{X}} x \cdot \mathbb{P}(X = x) \tag{2.10}$$

For continuous variables, it is defined as:

$$\mathbb{E}[X] = \int_a^b x \cdot \mathbb{P}(X = x) dx \tag{2.11}$$

More generally, the expectation can also be applied for a function of the random variable $f(X)$. It is then defined as:

$$\mathbb{E}[f(X)] = \sum_{x \in \mathcal{X}} f(x) \cdot \mathbb{P}(X = x) \tag{2.12}$$

The entropy can then be expressed as the expected information value over the probability distribution of the variable:

$$H(X) = -\sum_{x \in \mathcal{X}} \mathbb{P}(x) \cdot \log \mathbb{P}(x) = \mathbb{E}[\log \mathbb{P}(X)] \tag{2.13}$$

2.1.2 Statistical Distance Metrics Between Probability Distributions

Statistical distance metrics between probability distributions over the same random variable can be defined using the expectation operator. Given two distributions $\mathbb{P}$ and $\mathbb{Q}$ over X, a common distance metric is the *Kullback-Leibler divergence* (KL), defined as:

$$D_{KL}(\mathbb{P}||\mathbb{Q}) = \sum_{x \in \mathcal{X}} \mathbb{P}(x) \log \frac{\mathbb{P}(x)}{\mathbb{Q}(x)} = \mathbb{E}\left[\log \frac{\mathbb{P}(x)}{\mathbb{Q}(x)}\right] \tag{2.14}$$

KL can be used as an upper bound for the *total variational distance* (or *Wasserstein* distance) between two probability distributions [CK11]. The *Wasserstein distance* is denoted by $d_{TV}(\mathbb{P}, \mathbb{Q})$ and the upper bound in terms of KL is:

$$d_{TV}(\mathbb{P}, \mathbb{Q}) \leq \sqrt{2 D_{KL}(\mathbb{P}||\mathbb{Q})} \tag{2.15}$$

A similar distance metric is the *cross-entropy* between $\mathbb{P}$ and $\mathbb{Q}$, defined as:

$$D_{CE} = \sum_{x \in \mathcal{X}} \mathbb{P}(x) \log \mathbb{Q}(x) = \mathbb{E}_{x \sim \mathbb{P}}[\log \mathbb{Q}(x)] \tag{2.16}$$

2.1.3 Common Probability Distributions

Discrete random variables can be modeled either as a *multinomial* distribution, or as a *categorical* distribution, which are parametrized by a probability vector $p \in [0, 1]^n$, where n is the number of states the variable can take [BGC+17]. The difference lies in

the number of sampled values of the random variable that is modeled. The categorical distribution assumes one sample $X = k$, while the multinomial distribution requires n samples $X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n$. Let $\mathbb{P}$ be the probability mass function of the categorical, and $\mathbb{Q}$ be the probability mass function of the multinomial. They are defined as:

$$\mathbb{P}(X = k|p) = p_k \tag{2.17}$$

$$\mathbb{Q}(X_1 = x_1, \ldots, X_n = x_n|p) = \frac{n!}{x_1!x_2!\ldots x_n!} p_1^{x_1} \ldots p_n^{x_n} \tag{2.18}$$

Their expected values are:

$$\mathbb{E}_{x\sim\mathbb{P}}[x] = \sum_{i=1}^{n} ip_i \tag{2.19}$$

$$\mathbb{E}_{\{x_k\sim\mathbb{Q}\}}[x_k] = np_k, k \in [1, n] \tag{2.20}$$

Both distributions generalize distributions of discrete random variables with two possible states to n possible states (finite or countably infinite). The *multinomial* distribution generalizes the *binomial* distribution, while the *categorical* generalizes the *Bernoulli* distribution.

Continuous random variables are typically represented by *Gaussian* distributions, parametrized by a *mean* μ and a *standard deviation* σ. Its probability density function is denoted as $\mathcal{N}$ and is defined as:

$$\mathcal{N}(X = x|\mu, \sigma^2) = \sqrt{\frac{1}{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2} \tag{2.21}$$

The square of the standard deviation σ^2 is also called the *variance* of the distribution. The expected value of a random variable sampled from a *Gaussian* distribution is the mean:

$$\mathbb{E}_{x\sim\mathcal{N}}[x] = \mu \tag{2.22}$$

A side-by-side comparison of categorical and Gaussian distributions is shown in Figure 2.1. Binomial distributions are closely related to Gaussian distributions. For n samples of a discrete random variable, where n tends towards ∞, the shape of the distribution is a discretized Gaussian distribution, as shown in Figure 2.2.

Normal distributions can be extended to vectors of random variables $\mathbf{X} = [X]_n^\top$, where the mean is expressed as the expected value of $\mathbf{X}$, and the *covariance* matrix is computed as:

$$\Sigma_{i,j} = \mathbb{E}\left[(X_i - \mu_i)(X_j - \mu_j)\right], \; i, j \in [1, N] \tag{2.23}$$

Any arbitrary probability distribution over a random vector $\mathbf{X} = [X]_n^\top$ can be related to an associated multivariate Gaussian distribution by the *central limit theorem* [CB24], which states the following: given the expected value μ of the random vector $X_1, X_2, \ldots, X_n$ with mean $\bar{x}_n$ and covariance Σ, $(\bar{X}_n - u)\sqrt{n}$ is a multivariate normal distribution with mean 0 and covariance Σ.

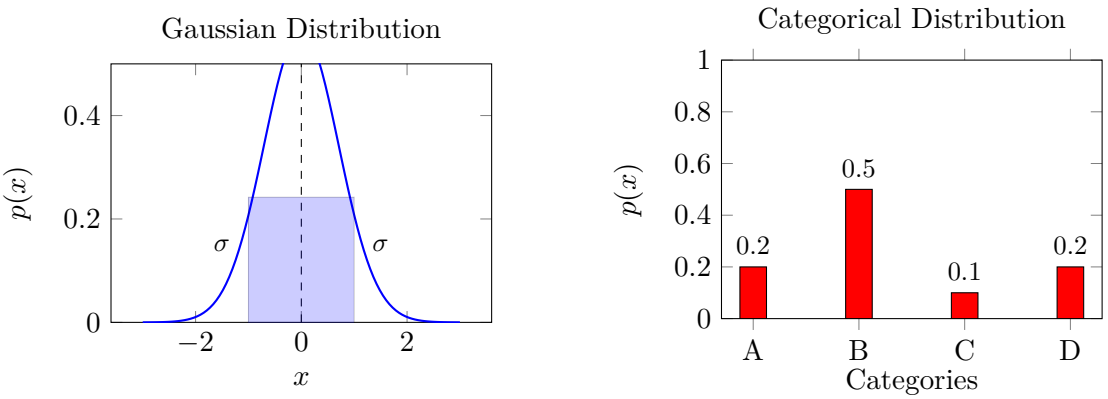


Figure 2.1: Comparison of a Gaussian (continuous) and a Categorical (discrete) distribution. The Gaussian highlights mean (μ) and standard deviation (σ), while the categorical shows the probability vector.

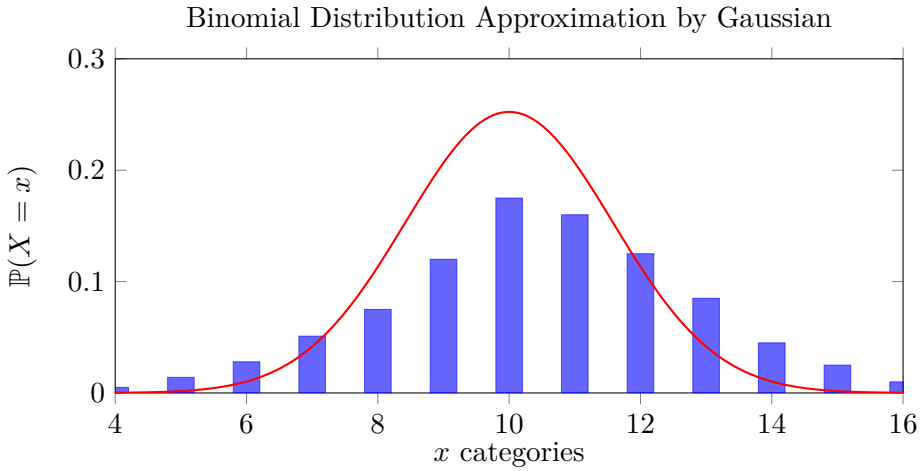


Figure 2.2: Samples from a Binomial Distribution $B(n = 20, p = 0.5)$ approximated by a Gaussian $\mathcal{N}(\mu = 10, \sigma^2 = 5)$.

2.2 Machine Learning Overview

Over the decades, the research field of *Artificial Intelligence* has advanced significantly at an accelerated pace. Starting from symbolic artificial intelligence, methods founded in high-order logic-based reasoning, the focus shifted towards *Machine Learning* and statistical methods that approximate arbitrary transfer functions that generalize to previously unseen data. Nowadays, the subfield of *Deep Learning* has and continues to give rise to algorithms and architectures that perform knowledge representation at high orders of abstraction and execute tasks without explicit instructions. The backbone of the field is represented by models inspired by neuroscience and algorithms that aim to replicate animal learning. These models are aptly titled *artificial neural networks* due to

their similarity to biological neural networks.

In general, artificial neural networks are formed by stacked layers of interconnected neurons with defined input and output layers and an arbitrary number of intermediate layers. The topology of the network varies, and the following subsection covers two important ones: *feedforward* and *recurrent* networks. Figure 2.3 shows an example of an ANN: a fully-connected feedforward neural network, which is described in the following section.

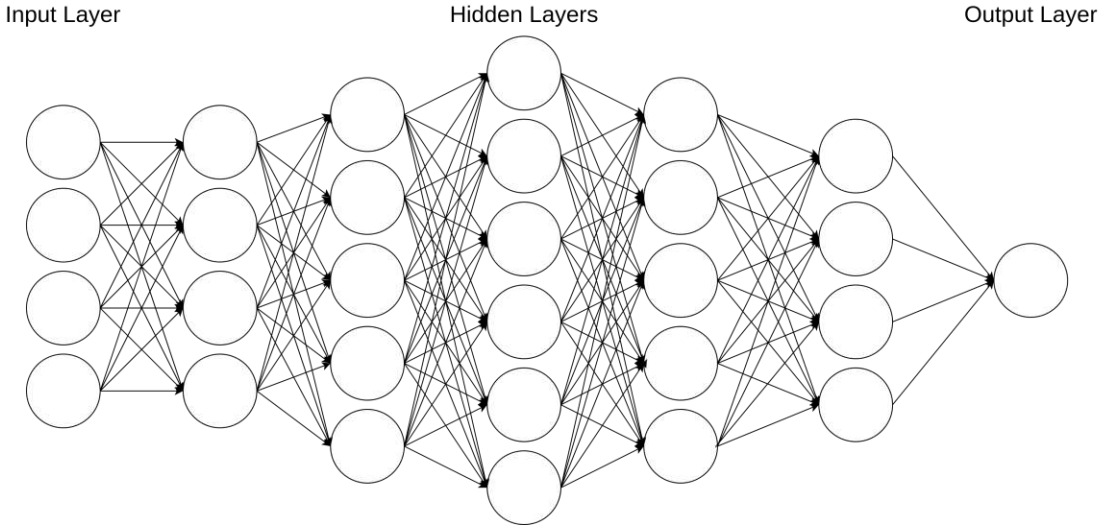


Figure 2.3: The architecture of a feedforward neural network.

2.2.1 Feedforward Neural Networks

The topology of an FNN is a weighted directed acyclical graph, with nodes representing artificial neurons (also called *cells*) and directed edges representing connections and the direction of the flow of information through the network (input → output). Each cell processes the signals received from the inward edges and outputs a signal along the outward edges. From a biological point of view, the cell represents the *nucleus* of a biological neuron, the *dendrites* represent the inward edges, and *axon* represents the outward edges. Biological neurons form *synapses* between dendrites and axons, the strength of which is abstracted by *weights* in ANNs. A biological neuron fires once the incoming signal exceeds its *synaptic threshold*, a mechanism abstracted by *activation functions*. An abstract view of the biological neuron model is included in Figure 2.4.

Assume an artificial neuron with n inward connections weighted by the column vector $w \in \mathbb{R}^n$. The neuron receives n inputs represented by the column vector $x \in \mathbb{R}^n$. The artificial neuron represents the following transfer function:

$$\hat{y} = \sigma(w^\top \cdot x + b) \tag{2.24}$$

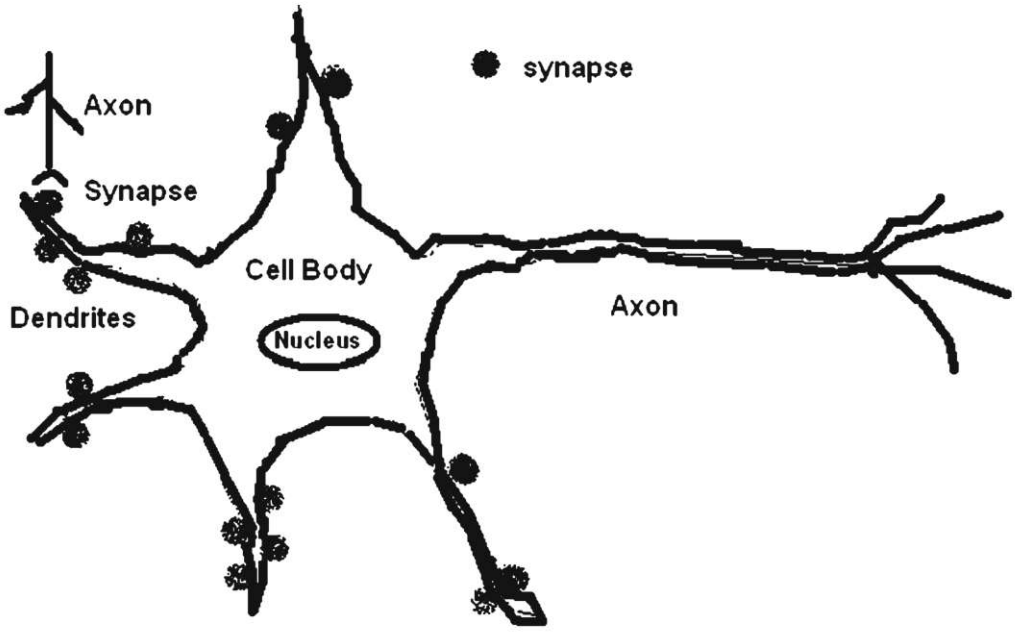


Figure 2.4: The biological neuron model, from [ZHS09].

σ represents the aforementioned activation function, which plays the role of the synaptic threshold. It can transform the neuron into a non-linear transfer function. In its absence, the neuron is simply a linear mapping shifted by an offset b called the *bias*. The choice of activation function depends on the desired codomain of the cell function. For a boolean output $y \in \top, \bot$, a binary step function can be chosen:

$$\sigma(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases} \tag{2.25}$$

Generally, activation functions are chosen to be continuously differentiable in order to enable gradient-based optimization methods for learning the weights and biases (the network parameters). This will be clearly illustrated in Subsection 2.2.2, where different continuously differentiable activation functions and their properties are also covered.

Multiple artificial neurons can be stacked "vertically" to form *layers* of neurons, which can further be stacked "horizontally" to form neural networks. The number of layers in a neural network, excluding the input layer, indicates the *depth* of the network. An FNN always contains an *input layer* (cells that receive network inputs) and an output layer (which emits the outputs of the network). When at least 1 *hidden layer* (intermediate layers) are present, when neurons in neighboring layers are fully connected and when non-linear activation functions are used in at least one of the hidden layers, the network is an MLP. Rosenblatt defined the MLP in [Ros58] for the purpose of approximating

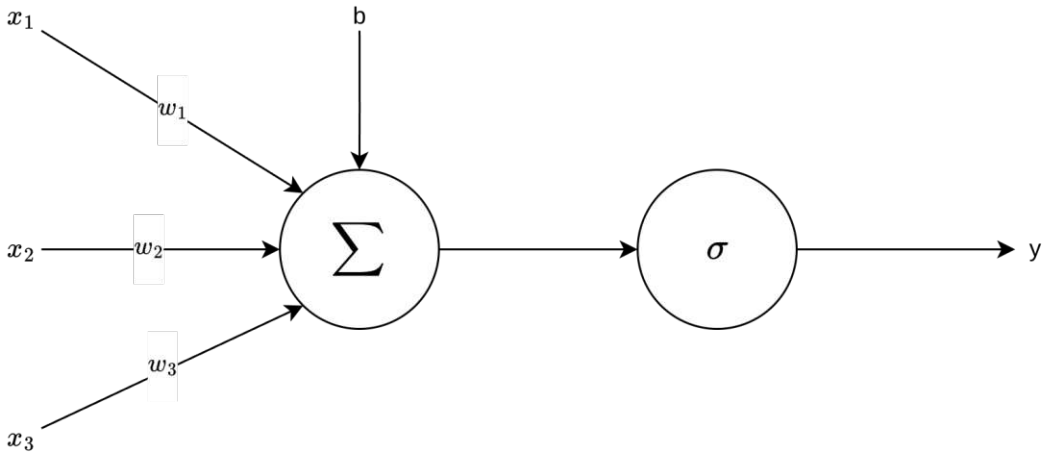


Figure 2.5: Model of an artificial neuron with 3 weights.

any arbitrary function. Previously, single-layer FNN had proven to be incapable of approximating the *exclusive-or* logic function $p \oplus u = (p \vee u) \wedge \neg(p \vee u)$ in [MP69]. MLPs are proven *universal function approximators* given enough hidden neurons, as proven by the *universal function approximation theorem*, which states that for any family of neural networks and any continuous function with domain and codomain as Euclidean spaces $f : \mathbb{R}^n \to \mathbb{R}^m$, there exists an instance of neural network of the family define over the function space such that, for any kernel $K \subseteq \mathbb{R}^n$, we have:

$$\sup_{x \in K} ||f(x) - \phi(x)|| < \epsilon \tag{2.26}$$

The proof for MLPs as universal approximators is sketched in [Cyb89]. Increasing the depth to a neural network allows for higher orders of abstractions, and networks that use more than three hidden layers are called *deep neural networks*.

2.2.2 Optimization Methods for Neural Networks

Assume a given function f, a neural network ϕ, and training and test datasets of the form $\mathcal{D} = \{(x, y)\}_{1:N}$ of input-output mappings according to f. The goal of an optimization process for learning ϕ is to find the right set of network parameters θ such that some objective function $\mathcal{L}$, applied over either of the datasets, is maximized/minimized. *Gradient-based optimization* methods make use of derivatives to guide the optimization process toward a minimum/maximum.

Let $f : \mathcal{X} \to \mathcal{Y}$ be a continuous differentiable function. The derivative of the function $f'(x) = \frac{dx}{dt}$ gives the slope of the function at the point x. Intuitively, this indicates the magnitude of influence a change of input has on the output. A straightforward gradient-based optimization method which uses the slope, or the gradient, of the differentiable

function is *gradient descent*. The method works by small-step updates of the parameters in the opposite direction of the gradient. For a step size α, the update rule is:

$$\theta \leftarrow \theta - \alpha \nabla f(\theta) \tag{2.27}$$

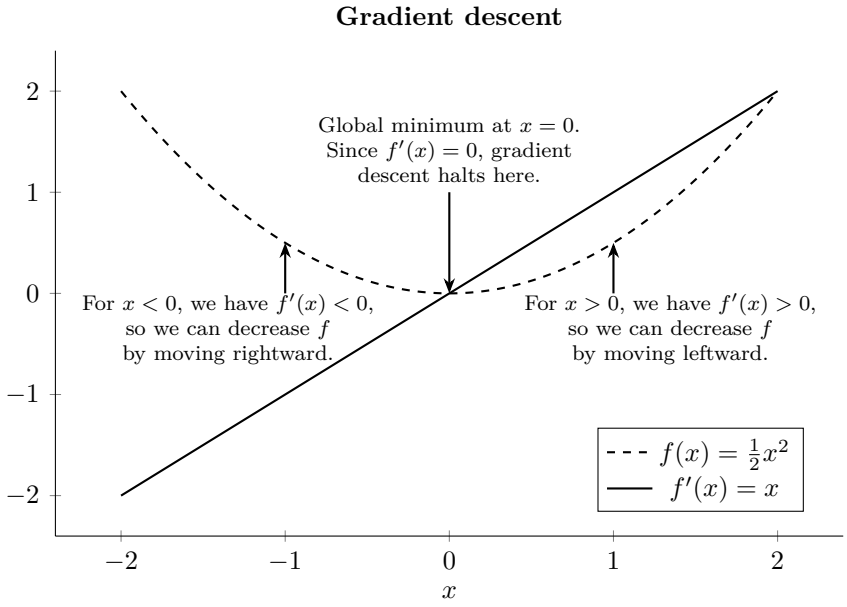


Figure 2.6: An illustration of a differentiable function f, its derivative f', its global minimum and case-by-case updates. Adapted from [BGC+17].

A useful example is included in Figure 2.6, adapted from [BGC+17]. Here, f is a strongly convex function with a unique local minimum at the point (0, 0), and with a linear derivative. In the region where the gradient is negative, the optimization process "shifts" the parameters rightwards. Where the gradient is positive, the parameters are updated leftwards.

Stronger assertions can be made regarding the nature of the function. ∇f is *Lipschitz-continuous* if it satisfies the following relation:

$$||\nabla f(x) - \nabla f(y)||_2 \leq L||x - y||_2, \forall x, y \in \mathcal{X} \tag{2.28}$$

L is the associated *Lipschitz constant*, and f is said to be *L-smooth*.

For convex, L-smooth functions, and a step-size satisfying $0 < \alpha \leq \frac{1}{L}$, it can be shown that *gradient descent* converges to the global minimum. A proof is included in Section 3.1 from [GG].

A machine learning problem learns a neural network ϕ that estimates an unknown joint probability distribution f, given a dataset of input-output samples $\mathcal{D} = \{(x, y)\}_{1:N}$ from the underlying joint probability x, y, and *loss* function $\mathcal{L}(\phi(x; \theta), y)$. The problem is

turned into a gradient-based optimization problem by minimizing the expected loss over the dataset [BGC+17]:

$$\mathbb{E}_{(x,y)\sim f(x,y)}\left[\mathcal{L}(\phi(x;\theta),y)\right] = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(\phi(x_i;\theta),y_i) \tag{2.29}$$

However, directly computing the gradient of (2.29) is intractable for $\mathcal{D}$ with high cardinality, as it requires evaluating all inputs in the dataset with the learned model. In practice, the gradient is statistically estimated, and there exist a variety of algorithms for deriving such statistics.

Stochastic gradient descent is applied on batches of input-output pairs sampled from $\mathcal{D}$ instead, following the gradient of the objective function $\mathcal{L}$ with respect to the network parameters. In that case, the gradient is *estimated* at each gradient descent iteration. The gradient of the objective function over inputs, wrt. the parameters is denoted as $\nabla_\theta\mathcal{L}(\phi(x;\theta),y)$. When there are more than one parameters $\theta\in\mathbb{R}^d, d>1$, the gradient $\nabla_\theta\mathcal{L}(x;\theta)$ is the column vector of partial derivatives:

$$\nabla_\theta\mathcal{L}(x;\theta) = \left[\frac{\partial\mathcal{L}(x)}{\partial\theta_1},\frac{\partial\mathcal{L}(x)}{\partial\theta_2},\ldots,\frac{\partial\mathcal{L}(x)}{\partial\theta_d}\right]^\top \tag{2.30}$$

Let $\{(x,y)_{1:n}\}\sim\mathcal{D}$ represent a sampled batch, g denote the true gradient and $\hat{g}$ its estimate. The update rules for the gradient estimate $\hat{g}$ and the network parameters θ are the following:

$$\hat{g}\leftarrow\sum\frac{\nabla_\theta\mathcal{L}(\phi(x;\theta),y)}{n} \tag{2.31}$$

$$\theta\leftarrow\theta-\alpha\hat{g} \tag{2.32}$$

Assuming i.i.d. samples, the gradient estimate is an *unbiased* estimate of the true gradient: $\mathbb{E}_{\{(x,y)_{1:n}\}_{1:\infty}\sim\mathcal{D}}[\hat{g}] = g$ [BGC+17]. Convergence proofs of SGD for convex, L-smooth functions (or derivations of bounds for non-convex but L-smooth functions) can be found in Section 5 of [GG].

An example of an alternative to the SGD algorithm is the *ADAM* optimizer introduced in [KB17]. *ADAM* uses estimates of the *first moment* (defined as the exponentially decaying average of previous gradients) and the second moment (the squares of the gradients) to dynamically compute individual learning rates for each parameter. The algorithm also includes bias correction to account for the initialization of these moment estimates. The update rules for the estimates, as well as a full description of the algorithm, can be found in Section 2 from [KB17].

All gradient-based optimization methods assume a continuously differentiable and L-smooth function. As a consequence, all functions in a neural network (the cell transfer function, the activation functions) need to satisfy this property. While it is possible to

apply gradient descent with differentiable functions that are non-differentiable at certain points (*ReLU* is non-differentiable in 0, as will be shown), continuous differentiability is desirable. The binary-step activation function (2.25), however, is outright excluded since it is non-differentiable at 0 and the derivative is always 0 otherwise. Gradient descent would not be able to perform any updates. For binary classification, an alternative activation function would be the *logistic function*, or the *sigmoid*:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}} \tag{2.33}$$

The logistic function is continuously differentiable and L-smooth over the range $(0, 1)$. As a result, its output can parametrize a Bernoulli distribution (2.17), which can be used to sample boolean variables.

Similarly, for predicting a discrete probability vector $[p]_{1:k}$ (the parameters of a categorical or a multinomial distribution (2.17)), the *softmax* activation function can be used. Assuming vector $\mathbf{X}$ of size n is the result of the output layer before activation, each vector element x_i is associated with a probability p_i by the following mapping:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{2.34}$$

It can easily be seen that the output probabilities sum up to 1 and are non-negative, making *softmax* a valid probability mass function:

$$\sum_{i=1}^{n} \text{softmax}(x_i) = \sum_{i=1}^{n} \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{2.35}$$

$$= \frac{1}{\sum_{j=1}^{n} e^{x_j}} \cdot \sum_{i=1}^{n} e^{x_i} \tag{2.36}$$

$$= 1 \tag{2.37}$$

The parameters of a Gaussian distribution μ, σ can also be estimated by a neural network. Assume $[x]_n$ is the output from the layer preceding the output layer. x can be sliced in two vectors $x^{[\mu]} \coloneqq [x]_{1:k-1}, x^{[\sigma]} \coloneqq [x]_{k:n}$, which can be used to parametrize the Gaussian distribution. The standard deviation is always a positive real value. Thus, an appropriate activation function is *softplus*, defined as:

$$\text{softplus}(x^{[\sigma]}) = \log 1 + e^{x^{[\sigma]}} \tag{2.38}$$

Typical activation functions for hidden layers are *Rectified Linear Unit* (ReLU), *Leaky Rectified Linear Unit* (LeakyReLU) and the *hyperbolic tangent* (tanh), with the following definitions:

$$\text{ReLU}(x) = \max(0, x) \tag{2.39}$$

$$\text{LeakyReLU}(x) = \max(0, x) + \alpha \min(0, x) \tag{2.40}$$

$$\tanh(x) = \frac{1}{1 + e^{-x}} \tag{2.41}$$

The hyperbolic tangent (2.41) and sigmoid (2.33) activation functions are *saturating* functions. A function is *saturating* if $\lim_{x\to\infty} |\nabla(f(x)| = 0$. As will be shown in Section 2.2.3, this can have a negative impact on the gradient computation.

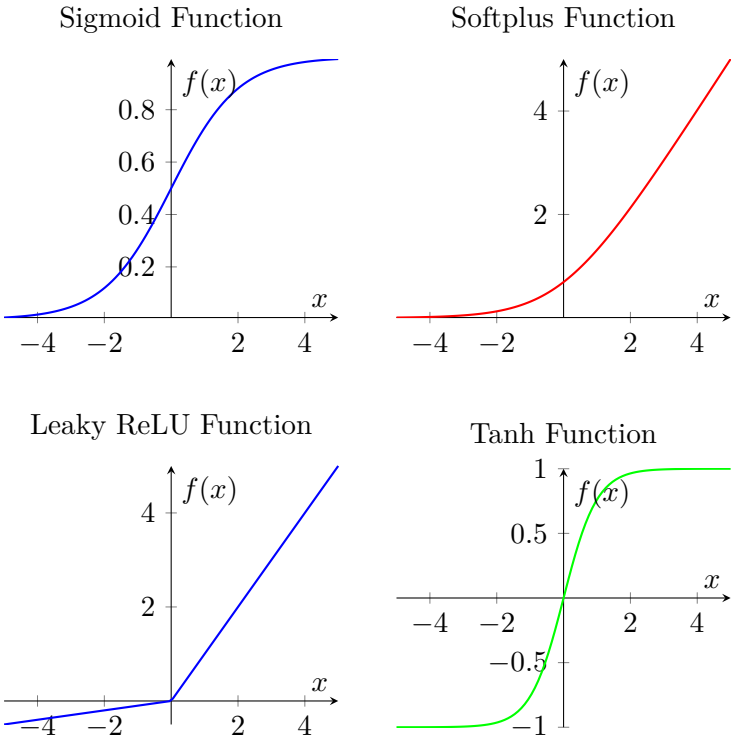


Figure 2.7: Comparison of Activation Functions: Sigmoid (2.33), Softplus (2.38), Leaky ReLU (2.40), and Tanh (2.41).

2.2.3 Backpropagation

The gradient from (2.30) is computed by *backpropagating* [RHW86] the evaluated loss function at the input x through the composition of functions in the network that relate θ to $\mathcal{L}(x;\theta)$. The mechanism used in backpropagation is the *chain rule* of calculus. Assume an MLP with one hidden layer and an activation function σ, of the following form:

$$\begin{aligned} \text{Input: } & x \in \mathbb{R}^d \\ \text{Hidden linear layer: } & f = Wx + b \\ \text{Activation function: } & \hat{y} = \sigma(f(x)) \end{aligned}$$

Starting from the partial derivative of the loss given the predicted output $\frac{\partial \mathcal{L}}{\partial \hat{y}}$, the partial derivative for each model parameter and the input can be computed by recursively applying the chain rule: $\left[\frac{\partial \mathcal{L}}{\partial W}, \frac{\partial \mathcal{L}}{\partial b}, \frac{\partial \mathcal{L}}{\partial x}\right]^\top$.

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial f} \cdot \frac{\partial f}{\partial W} \tag{2.42}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial f} \cdot \frac{\partial f}{\partial b} \tag{2.43}$$

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial f} \cdot \frac{\partial f}{\partial x} \tag{2.44}$$

2.2.4 Learning Algorithms for Neural Networks

Typically, in machine learning, a NN is optimized to generalize a target function for any inputs from the function's domain. In supervised learning, there exists a dataset of ground-truth input-output mappings, from which samples are used to compute a cost $\mathcal{J}$, which is then used to apply some gradient-descent method to adjust the function estimate parameters. The choise of cost function $\mathcal{J}$ is imporant for two reasons. Firstly, the function must be continuouly differentiable w.r.t. the function parameters, as mentioned before. Secondly, using an exact cost function perfectly fits the function estimate to the target function for the given training dataset, in the limit, after uniformly sampling the entire dataset. This phenomenon is known as *overfitting*, as the learned function would not fit the target function for inputs that are not in the training dataset. For these reasons, machine learning algorithms use *surrogate loss* functions that are continuously differentiable and guide the learning process towards the underlying goal.

The choice of surrogate loss depends on the task. Let $\mathcal{X}$ be the domain of the target function f and $\mathcal{Y}$ be the codomain. f_θ represents the NN function estimate. The general machine learning goal is to estimate a probability distribution over the outputs, conditioned on the inputs and the NN parameters: $\mathbb{P}(Y|X,\theta)$, as described in Section 2.2.1. Minimizing the expectation given in (2.29) is called *empirical risk minimization*, which applies to both probabilistic and deterministic function estimators. In the case of probability distribution estimation, *maximum likelihood estimation* optimizes θ to maximize the likelihood of the sampled ground-truth data, under the learned distribution.

One possibility is to assume that the target probability distribution is a Gaussian distribution, to fix its variance σ and to learn the mean of the distribution. In this case, a useful surrogate loss function is the MSE over the i.i.d. observed ground-truth outputs y and the outputs of the NN given the ground-truth inputs:

$$\mathcal{L}(\phi(x;\theta), y) = ||f_\theta(x) - y||^2 \tag{2.45}$$

The MSE is then averaged over the batch dimension.

A more general objective for any probability distribution is the KL divergence between the estimated distribution and the actual distribution (2.14). In practice, an appropriate surrogate loss for this objective Negative Log Likelihood (NLL) of the ground-truth outputs, given the learned distribution [BGC+17]. Let P_θ denote the distribution

parameterized by the outputs of f_θ. The NLL loss function is the following:

$$\mathcal{L}(\phi(x;\theta), y) = -\log \mathbb{P}_\theta(y|x) \tag{2.46}$$

2.2.5 Recurrent Neural Networks

Neural networks can be structured to share certain parameters across multiple neurons or layers. In *convolutional neural networks* for example, which are used for image inputs, the inputs as well as the layers are matrices. The weight associated with a unit is used multiple times during the convolution operation over the matrix input at each layer [LBBH98].

Another example are Recurrent Neural Network (RNN)s, which are used to process sequences, where the inputs have an extra time dimension. FNN are ill-posed to handle sequence inputs due to two important reasons:

1. Handling the time dimension would require adding separate parameters for each time index (timestep), in each layer of the network. However, the learned parameters would not be invariant to the time index of a given datapoint in an arbitrary input sequence. Furthermore, the time dimension would have to be fixed to the same value for both the network parameters and all inputs [BGC+17].
2. The space complexity of a FNN would increase linearly with the time dimension of the input sequences.

An important concept in RNNs is the explicit build-up of *memory* as a statistical summary of inputs from past timesteps. A good analogy are dynamical systems driven by external inputs, of the form:

$$s_t = f(x_t, s_{t-1}, t) \tag{2.47}$$

The state of the system at timestep t s_t depends on the previous state s_{t-1} and the external input applied to the system x_t. Evidently, the equation of a dynamical systems is a recurrent expression that is equivalent to the following, *unfolded* expression, given a mapping function g:

$$s_t = s_0 + \sum_{k=0}^{t} g(x_k) \tag{2.48}$$

The function g maps the sequence of past inputs to the current state. It can be said that g builds a statistical summary of the past inputs. In other words, it acts as a memory buildup.

RNNs architectures, in general, implement the same principle. The simplest architecture for an RNN is the one introduced in [Elm90].

For RNN, the general equation for a neuron in FNNs (2.24) is changed as follows: a state input is introduced to the neuron (analogous to the past internal state s_{t-1} of a

dynamical system), and the weight and bias vectors are shared across the time dimension:

$$s_t = \sigma_1(w_1^\top s_{t-1} + w_2^\top x + b) \tag{2.49}$$
$$\hat{y}_t = \sigma_2(s_t) \tag{2.50}$$

The lack of time indices for the parameter vectors w_1, w_2, b indicate that they are shared across timesteps. σ_1 is typically a bounded non-linear activation function such as the hyperbolic tangent function. Figure 2.8 illustrates an RNN in two ways: the recurrent view and the *unfolded*, spatial view, where each timestep is explicitly drawn.

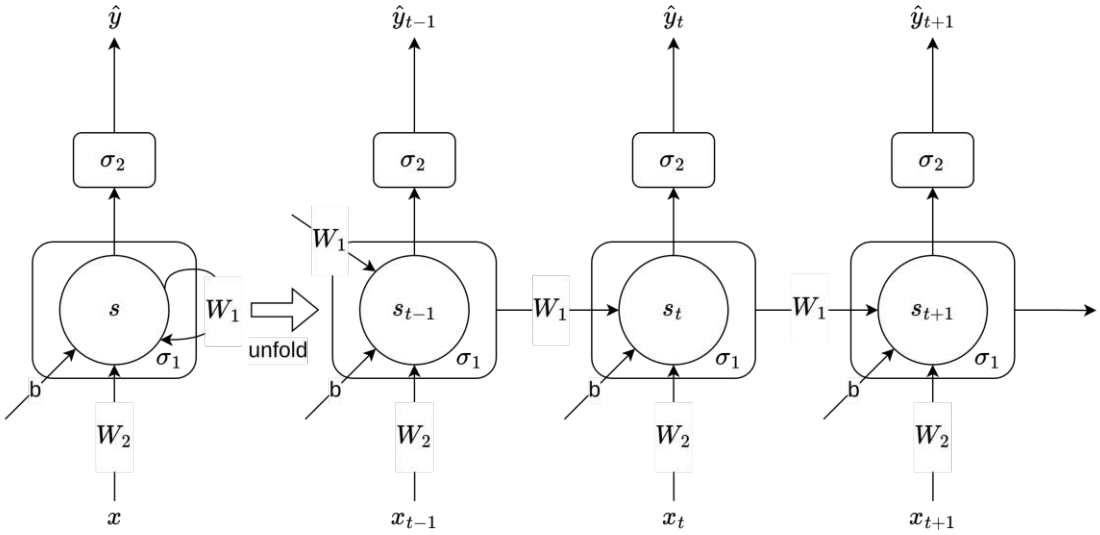


Figure 2.8: Recurrent and unfolded view of an RNN.

Any training algorithm for FNNs can be applied to train RNNs as well. Computing the gradient of a loss function w.r.t the network parameters is done by applying BPTT, which is essentially the same recursive application of the chain rule as before, but through the unfolded view of the network. However, BPTT introduces a few drawbacks. The composition of partial derivatives of non-linear functions at each unfolding step can lead to either collapsed values, or very inflated values for the final gradient. This is known as the *vanishing/exploding gradient* phenomenon and it introduces challenges in training standard RNNs on long sequences [PMB13]. Another problem is the sequential process of computing the gradient, which cannot be parallelized. This leads to training algorithms not scaling well with increasing sequence lengths.

Specific RNNs have been designed to deal with the *exploding/vanishing* gradient problem. LSTM recurrent neural networks, for example, are a type of *gated* RNN, where the information flow is controlled through learned gating mechanisms [HS97]. The inputs that pass the *input gate* are accumulated into the internal hidden state of the LSTM cell, which has a self-loop controlled by a *forget gate*. The output is also gated by

an *output gate*. The interplay between the learned gates allow the cell to selectively accumulate inputs, to learn how to *forget* past stale information and to control the outgoing information flow.

2.3 Dynamical Systems and Neural Network approximations

This section focuses on neural network architectures used to model dynamical systems.

2.3.1 Dynamical Systems

The evolution of dynamical systems over time can be described by a system of ordinary differential equations that define state updates. While autonomous systems depend only on the state of the system itself, external inputs are also typically modeled. An ODE expresses the n-order derivative of the system in terms of its state derivatives and the external input:

$$s^{(n)} = f(x, s, s^{(1)}, \ldots, s^{(n-1)}) \tag{2.51}$$

2.3.2 Neural Ordinary Differential Equations

Neural ODEs belong to a different class of NNs: continuous-time neural networks. Whereas the RNNs have been introduced as estimators for discrete-time sequential functions, neural ODEs directly estimate the derivative of a dynamical system in time domain [CRBD19]:

$$\frac{ds(t)}{dt} = f(s(t), x(t), t; \theta) \tag{2.52}$$

$$\tag{2.53}$$

Whereas inference (meaning, applying the learned function over some inputs) in a FNN is done by forwarding the input through sequentially applying the linear operations and non-linear activation functions in each layer, evaluating a neural ODE is done by integrating the function over time, starting from an initial state:

$$s(t) = s(t_0 + \Delta t) = s(t_0) + \int_{t_0}^{t} f(s(t), x(t), t)dt \tag{2.54}$$

The integral in (2.54) is approximated using numerical solvers such as the *forward Euler method* or *Runge-Kutta* [Run95] [Kut01]. The latter generalizes the former. For example, 4-th order Runge-Kutta (RK4) applies the following approximation steps, starting from

the initial state s_{t_0}:

$$\begin{aligned} k_1 &= f(s_{t_0}, x(t_0), t_0) \\ k_2 &= f\left(s_{t_0} + \frac{\Delta t}{2} k_1, x(t_0 + \frac{\Delta t}{2}), t_0 + \frac{\Delta t}{2}\right) \\ k_3 &= f\left(s_{t_0} + \frac{\Delta t}{2} k_2, x(t_0 + \frac{\Delta t}{2}), t_0 + \frac{\Delta t}{2}\right) \\ k_4 &= f\left(s_{t_0} + \Delta t \cdot k_3, x(t_0 + \Delta t), t_0 + \Delta t\right) \\ s_t &= s_0 + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{2.55}$$

The *explicit Euler* discretization method is identical to first-order *Runge-Kutta*:

$$s_t = s_{t_0} + \Delta t \cdot f(s_{t_0}, t_0) \tag{2.56}$$

2.3.3 Liquid Time-Constant Recurrent Neural Networks

Other approaches maintain the system dynamics estimation of neural ODEs of Section 2.3.2, while changing the formulation of the artificial neuron, as well as the inter-neuron synapses. Liquid Time-Constant Neural Network (LTC)s [HLA+21] represent the cell transfer function as an ODE that explicitly incorporates the synaptic transmission from neuroscience [KS98] into the neuron model, together with a varying time-constant. The synaptic currents are represented as sigmoidal functions dependent on the presynaptic nodes's states $\sigma(s_j)$ [LHA+]. The time-constant of the system is also modeled as a nonlinear function dependent on the presynaptic node modulated by a sigmoidal σ, and a learnable parameter representing the base time-constant of the system τ. In this manner, LTCs learn an adaptable system response to inputs that is also intrinsically self-coupled. The membrane capacitance of the neuron C_{m_i}, its resting potential $s_{\text{leak},i}$ and the synapse polarity E_{ji} are also modeled as learnable parameters of the network.

The transfer function for an LTC cell s_i with presynaptic neurons I_{in} is the following: [LHA+]:

$$\frac{ds_i}{dt} = -\left(\frac{1}{\tau_i} + \sum_{s_j \in I_{\text{in}}} \frac{w_{ij}}{C_{m_i}} \sigma_i(s_j)\right) s_i + \left(\frac{s_{\text{leak},i}}{\tau_i} + \sum_{s_j \in I_{\text{in}}} \frac{w_{ij}}{C_{m_i}} \sigma_i(s_j) E_{ij}\right) \tag{2.57}$$

The *liquid time-constant* of the neuron can be expressed as [Has20]:

$$\tau_{\text{system}} = \frac{1}{\frac{1}{\tau_i} + \sum_{s_j \in I_{\text{in}}} \frac{w_{ij}\sigma_i(s_j)}{C_{m_i}}} \tag{2.58}$$

where $\tau_i = \frac{C_{m_i}}{g_{l_i}}$ is the base time-constant paramtererized by the neuron's leakage conductance g_{l_i}.

LTC neurons can be wired together to form neural networks as systems of nonlinear ODEs with learnable time-constants. Similarly to neural ODEs (2.54), inference is done

by approximating the system of ODEs using numerical solvers over discrete timesteps. In the case of LTCs, the *hybrid Euler* solver is introduced [Has20] as a fast solver. The choice is motivated by the following reasons [LHA+]:

1. Real-time systems impose runtime constraints, so numerical solvers of high time complexity such as RK4 are avoided.
2. The system of ODEs of the form (2.57) is stiff, requiring fine-grained solver step sizes for stability, which further increases the inference time.
3. The exploding/vanishing gradient problem that occurs when computing partial derivatives via BPTT for RNNs must also be avoided.

The *hybrid Euler* solver combines the *explicit* Euler discretization (2.56) with the implicit Euler method:

$$s_t = s_{t_0} + \Delta t \cdot f(s_t, s_{t+1}) \tag{2.59}$$

A forward step using the hybrid Euler solver for LTC is expressed as the following [LHA+]:

$$s_i(t + \Delta t) := \frac{s_i(t) C_m/\Delta t + g_{l_i} s_{\text{leak}} \sum_{s_j \in I_{\text{in}}} w_{ij} \sigma(s_j(t)) E_{ij}}{C_m/\Delta t + g_{l_i} + \sum_{s_j \in I_{\text{in}}} w_{ij} \sigma(s_j(t))}, \tag{2.60}$$

The hybrid solver (2.60) is applied six times per LTC inference step.

2.3.4 Closed-Form Continuous-Time Neural Networks

Approximating the state of LTCs using numerical ODE solvers impose a high time complexity. An alternative, closed-form approximation called *closed-form continuous-time neural networks* (CfC) is proposed in [HLA+22]. Firstly, the synaptic currents in LTC cells are approximated by a non-linear function of the internal state and a neural ODE, leading to the following LTC formulation for a network [HLA+22]:

$$\frac{ds(t)}{dt} = -\left(\frac{1}{\tau} + f(\mathbf{s}(t), \mathbf{x}(t), t; \theta)\right) \mathbf{s}(t) + f(\mathbf{s}(t), \mathbf{x}(t), t; \theta) A \tag{2.61}$$

where $x(t)$ represents the network input. Theorem 1 from [HLA+22] defines an approximation for the closed-form of an LTC cell (2.57):

$$s(t) = (s_0 - A) e^{-(w_\tau + f(x(t);\theta))t} f(-x(t); \theta) + A \tag{2.62}$$

where $x(t)$ represents the neuron input at timestep t.

Starting from the closed-form approximation for a scalar neuron, [HLA+22] formulates a neural network architecture where the hidden state evolution of the network at timestep t is governed by the following equation:

$$\mathbf{s}(t) = B \odot e^{-(w_\tau + f(s(t), x(t); \theta)t)} \odot f(-s(t), -x(t); \theta) + A \tag{2.63}$$

where B, A, w_τ, θ are the learnable parameters of the network.

Since the formulation above, called Cf-S in [HLA+22], is prone to exhibit vanishing gradient issues when training via BPTT [HLA+22], the exponential decay term is replaced by a sigmoid function of the form $\sigma(-f(s(t), x(t), t; \theta_f))$, which has a slower decay over time and acts as a differentiable gating mechanism for the ODE. Furthermore, to enhance the flexibility of the model, the parameter vectors B and A are replaced by neural networks with separate learnable parameters $g(s(t), x(t); \theta_g)$ and $h(s(t), x(t); \theta_h)$. Finally, h is additionally multiplied with the reverse of the sigmoid $1 - \sigma(\cdot)$ to achieve a time-dependent gating mechanism that smoothly interpolates between f and g. Overall, the resulting CfC model is the following [HLA+22]:

$$\begin{aligned} s(t) = &\sigma(-f(s(t), x(t); \theta_f)t \odot g(s(t), x(t); \theta_g) \\ &+ [1 - \sigma\left(f\left(s(t), x(t); \theta_f\right) t\right)] \odot h(s(t), x(t); \theta_h) \end{aligned} \tag{2.64}$$

f, g and h are preceded by a *backbone* neural network that acts as shared feature extractor. Figure 2.9 illustrates the architecture of CfC:

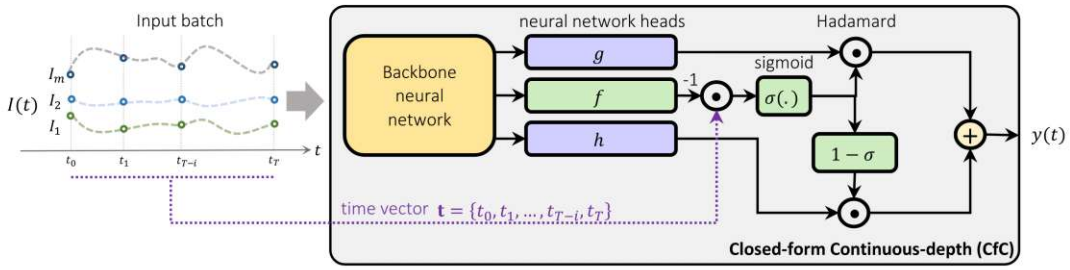


Figure 2.9: CfC architecture, taken from [HLA+22]. I represents the input matrix as batches of sequences of data, which is denoted as x in this section.

2.4 Reinforcement Learning

Reinforcement Learning (RL) encompasses a class of learning algorithms for solving a given problem, expressed as a sequential decision-making task with a defined goal. RL algorithms iteratively learns an *agent* that maps situations to decisions at each step, such that the defined goal is achieved. Unlike in supervised learning (in Section 2.2.4), ground-truth situation → decision pairs are not provided. Rather, learning is driven purely from maximizing a *reward* signal that reflects the objective of the task.

The study of RL is rooted in two distinct research fields, namely classical control theory and neuroscience. The former is concerned with designing optimal control policies for a dynamical system/plant in order to drive an actuating signal towards a reference signal, given a feedback signal. Regarding the latter, the theory is inspired from the study of animal behaviour, which is reinforced by rewards and punishments received.

In RL, an *agent* interacts with an *environment* by sampling *actions* from a learned *policy*, given the *observed* state of the environment/agent. The agent refines its policy based on the *rewards* received from the environment. With regards to control theory, the *agent* is analogous to the *controller/plant* and *observed state* to the *feedback signal*. While there is no direct analogy to the error between the reference and the actuating signals from control theory, the agent learns to choose optimal actions from the rewards received from the environment, in order to achieve the underlying objective of the task. The latter, which is inaccessible, is analogous to the *reference signal*, while the actions of the agent over time can be seen as the *actuating signal*. Meanwhile, the reward-driven learning process in RL is directly inspired from the *trial-and-error* learning process observed in neuroscience. Behavioral patterns have been observed to be *reinforced* by external positive/negative stimuli. Over time, animals are more likely to choose actions that have lead to positive stimuli. The positive stimuli is directly analogous to *rewards* in RL, while the reinforcement of behavioral patterns is analogous to the learning process of the agent's *policy*.

2.4.1 Fundamentals of Reinforcement Learning

Markov Decision Processes

RL problems are formulated as Markov Decision Process (MDP)s, which are extensions of Markov chains. A *Markov chain* is a stochastic process where the next event is independent of the history of events, given the current event. In other words, future outcomes of can predicted based solely on the current evidence. This is known as the *Markov property* and it represents the basic theoretical mechanism for RL approaches. The process is typically defined as having a state S as a random variable over time. The Markov property is expressed in (2.65):

$$\mathbb{P}(S_{t+1}|S_t, S_{t-1}, \ldots, S_1) = \mathbb{P}(S_{t+1}|S_t) \tag{2.65}$$

Markov chains are *fully-observable* Markov models, in the sense that the stochastic process is accessible. In Hidden Markov Model (HMM)s, the observable states O are conditioned on hidden, latent states S. The *Markov property* in HMM defines an independence of future observations from past latent states and observations, given the current state:

$$\mathbb{P}(O_{t+1}|S_t, S_{t-1}, \ldots, S_1, O_{t-1}, \ldots O_1) = \mathbb{P}(S_{t+1}|S_t) \tag{2.66}$$

MDPs extend Markov chains by conditioning the stochastic process on actions, and including rewards for state-action pairs. Whereas Markov chains and HMM are autonomous systems (the stocastic process is not conditioned on any inputs), MDPs are systems controlled by an *agent*.

Formally, an MDP is a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$, where:

1. $\mathcal{S}$ is the *state space*, or the domain the process state is defined over
2. $\mathcal{A}$ is the *action space* for the actions taken by the agent
3. $\mathcal{T}$ denotes the transition probability distribution over the next process state, given the current state and the action
4. $\mathcal{R}$ is the reward probability distribution, conditioned on states and actions
5. γ is a discount factor for future rewards

The transition probability distribution $\mathcal{T}$ of MDPs satisfies the Markov property. Future states of the process are independent on the past state-action sequence, given the current state and action.

Partially Observable Markov Decision Processes (POMDP) extend MDPs similarly to how HMMs extend Markov chains. Formally, POMDPs are represented as a 7-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{O}, \mathbb{P}_O, \gamma)$, with the following extension:

1. $\mathcal{O}$ is the *observation space*, over which observed states are defined
2. $\mathbb{P}_O$ is the probability distribution over observations, conditioned on the states, otherwise known as the *observation model*

The process state is hidden, and the process emits observations conditioned on the hidden states. The sequences of states and actions over time still form an MDP, and a POMDP can always be reduced to an MDP, where the stochastic process is represented by *belief states* - probability distributions over the underlying states, denoted $b(s), s \in \mathcal{S}$. This is called a *belief MDP*, and $\mathcal{B}$ denotes the space of belief distributions over the hidden states $s \in \mathcal{S}$ of the original POMDP.

All Markov models described above can be finite or infinite. Finite Markov models are defined by having an *absorbing state* s, such that $\mathbb{P}(S_{t+1} = s | S_t = s) = 1$. Figure 2.10 illustrates a simple example of each Markov model described in this subsection, with each having a starting and an absorbing state.

Bellman Equation and Dynamic Programming

In this subsection, for simplifying the definition of the introduced notions, environments are assumed to be finite Markov Decision Process (MDP)s. In other words, the Markov property is satisfied (2.65) and the state space $\mathcal{S}$, action space $\mathcal{A}$ and reward space $\mathcal{R}$ are finite sets. The goal of an MDP is to find a policy that maximizes the expected sum of rewards from the environment, up to the final absorbing state.

For finite MDPs, globally optimal policies can be found with *generalized policy iteration* methods, of which the main ones are *dynamic programming*, *Monte Carlo Prediction/Control* and *temporal-difference learning*. All of these methods feature a two-phase

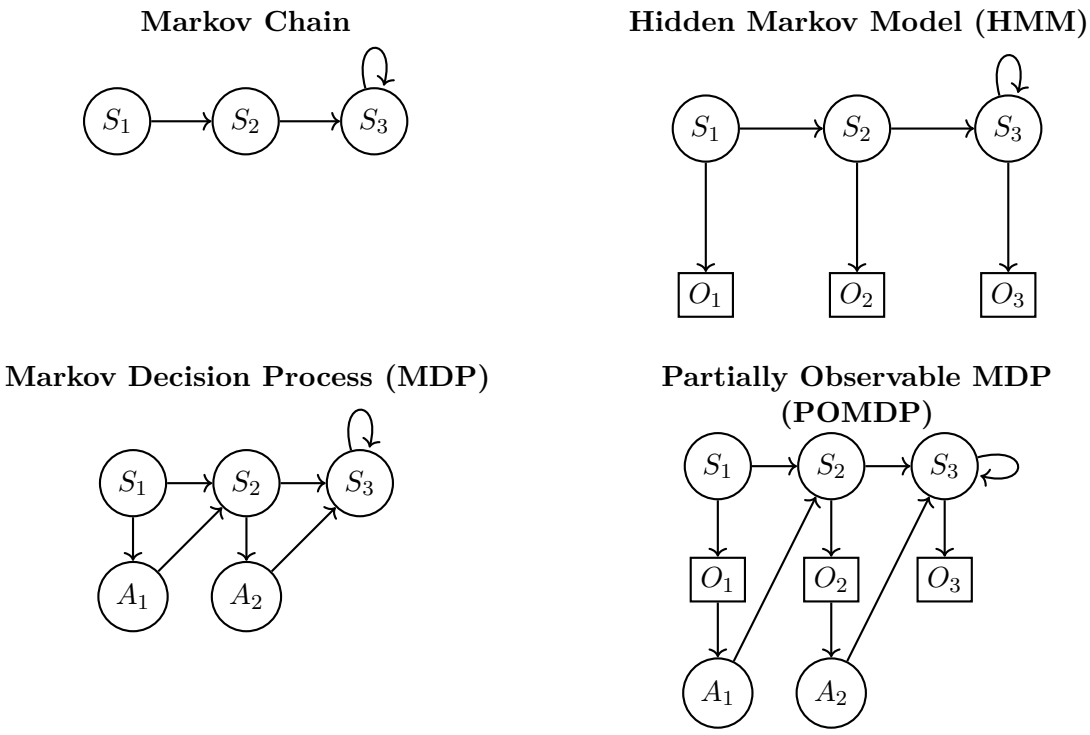


Figure 2.10: Simple examples of Markov models and decision processes.

iterative process of improving the evaluation of a policy, followed by improving the policy itself using the devised evaluation method.

A value function that maps the state space to real-scalar values $V : \mathcal{S} \to \mathbb{R}$, as well as a mapping from state-action to scalar values $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ can be defined. Their purpose is to quantify the utility of being in a certain state given the task at hand in the former and the utility of being in a state and choosing a specific action in the latter. The value function is recursively defined by the Bellman equation [BCC57] [Bar21]:

$$V_*(s) = \max_{a \in \mathcal{A}} \mathbb{E}[R_{t+1} + \gamma \cdot V_*(S_{t+1}) | S_t = s, A_t = a] \tag{2.67}$$

$$= \max_{a \in \mathcal{A}} \sum_{s', r} \mathcal{T}(s', r | s, a) \cdot [r + \gamma V_*(s')] \tag{2.68}$$

Here $*$ indicates optimality. As will be shown, there exists a total order over the possible value functions, as well a unique, optimal V_* such that $V_*(S) \geq V(S), \forall V$. The value function provides the expected discounted return $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$. γ is called the *discount factor* and is defined to be in range $(0, 1]$, and it is used to avoid infinite values for non-terminating MDP and to provide convergence guarantees for the optimal value function. If rewards are normalized in the range $(0, 1]$, then the bounds of the value function are $0 \leq V_\pi(S) \leq \frac{1}{1-\gamma}$.

The Bellman equation for the state-action quality function is [BCC57] [Bar21]:

$$Q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma \max_{a' \in \mathcal{A}} \cdot Q_*(S_{t+1}, a') | S_t = s, A_t = a] \tag{2.69}$$

$$= \sum_{s',r} \mathcal{T}(s', r|s, a) \cdot \left[r + \gamma \max_{a' \in \mathcal{A}} Q_*(s', a') \right] \tag{2.70}$$

As can be seen from the Bellman equation, the expected return over the possible actions is equivalent to summing the immediate reward R_t received by following the optimal action with the expected return starting from the next state after following said action.

Intuitively, value function $V_*(s_t)$ provides the expected returns G_t achieved by choosing the action that maximizes the value of each future state, whereas the quality function $Q_*(s_t, a_t)$ provides the expected G_t formed by the immediate reward R_t received by executing the conditioned action under the condition state, followed by the future returns under actions that maximize the future quality values.

By turning the Bellman optimality equation above into a recursive update rule, the MDP problem satisfies the principle of *optimal substructure* and thus, can be solved by DP [BCC57] [How60]. The expectation under $\max_{a \in \mathcal{A}}$ in the Bellman equation is first substituted with the expectation under the current policy π [How60]:

$$V_\pi(S_t) = \mathbb{E}_\pi \left[R_{t+1} + \gamma V_\pi(S_{t+1} | S_t = s) \right] \tag{2.71}$$

$$= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s',r} \mathcal{T}(s', r|s, a) \left[r + \gamma \cdot V_\pi(s') \right] \tag{2.72}$$

Starting from an arbitrary initial approximation V_0 of the true value function V_*, the approximation is refined at each iteration k using the Bellman equation for V_π as an update rule, called the *Bellman backup*:

$$V_k(S_t) = \mathbb{E}_\pi \left[R_{t+1} + \gamma V_{k-1}(S_{t+1} | S_t = s) \right] \tag{2.73}$$

The existence of the fixed point $V_k = V_\pi$ can be shown using the *contraction mapping theorem* [How60] [Bar21]. Let $\mathbb{F}$ be the space of functions on $\mathcal{S}$. Furthermore, let the *Bellman operator* be defined as $\Gamma : \mathbb{F} \to \mathbb{F}, \Gamma(V_\pi) = \mathbb{E}_\pi \left[R + \gamma \cdot \mathcal{T}_\pi(s'|s, \pi(s) \cdot V(s') \right]$. It can be shown that Γ is a contraction, i.e. $||\Gamma(V) - \Gamma(U)||_\infty \leq \gamma ||V - U||_\infty, \forall V, U \in \mathbb{F}$ [How60]. Thus, there exists a unique fixed point $V_k = \Gamma V_k$. The proof of applying the *contraction mapping theorem* for the value function space is included in [How60], and its application for value iteration convergence is described in [Bar21].

The iterative DP algorithm for finding V_π is called *policy evaluation*, and the output is the exact value function for the MDP state space. This is then used to iteratively improve the policy in *policy iteration*. The quality function for taking an action in a certain state and thereafter following the policy π provides an evaluation for the current policy:

$$Q_\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma \cdot V_\pi(S_{t+1}) | S_t = s, A_t = a] \tag{2.74}$$

The *policy improvement theorem* [Bar21] states that if another policy π' provides actions that improve the quality over the valued current policy $Q_\pi(s, \pi'(s)) \geq V_\pi(s), \forall s \in \mathcal{S}$, then the policy π' is an improvement over π because it provides greater or equal expected returns: $V'_\pi(s) \geq V_\pi(s)$. The policy iteration algorithm proceeds as follows. Starting from an initial random policy π, a policy π' is chosen such that $\exists s \in \mathcal{S} : \pi'(s) \neq \pi(s)$. For all other states, the condition for the policy theorem (2.4.1) holds. π' is considered a *refinement* if $Q_\pi(s, \pi'(s)) > V_\pi(s)$. By the *policy improvement* theorem, the refined policy is an improvement over the previous policy. Since there exists an optimal value function $V_*(S)$, in the limit, the iterative process converges to an optimal policy π such that $Q_\pi(s, \pi(s)) = V_\pi(s)$.

The evaluation of the policy at each iteration (which requires multiple sweeps through the state space) can be bypassed by directly iterating over the value function using the Bellman optimality equation (2.67) as an update rule:

$$V_{k+1}(S) \leftarrow \max_{a \in \mathcal{A}} \sum_{s',r} \mathcal{T}(s', r|s, a) \cdot [r + \gamma V_k(s')] \tag{2.75}$$

where k denotes the iteration counter. Instead of evaluating under the current policy at each iteration, the value function is directly updated using the action that yields the highest expectation. In this manner, convergence of the value function update implies convergence of the policy. The algorithm is called *value iteration*. While convergence still occurs in the limit, in practice, the iterative algorithm is stopped when a termination condition is satisfied (typically $V_{k+1}(S) - V_k(S) < \epsilon$ for all $S \in \mathcal{S}$.

All these approaches are grouped under the *general policy iteration* term, as the two-phase process of evaluation and improvement is present in all of them. Figure 2.11 illustrates how the process alternates between satisfying the goals of finding a better estimate for evaluating the quality of a derived policy and improving it, a process which converges in the limit to the optimal value function and policy V_*, π_*.

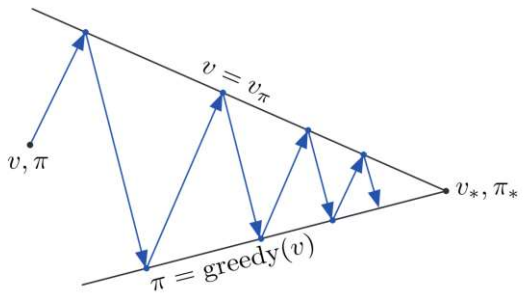


Figure 2.11: Policy iteration steps towards convergence, from [Bar21].

Monte Carlo Methods

The *DP* methods described in the previous section require the model of the environment to be known. For the Bellman update rule, for example (2.75), computing the expected value requires knowledge of the probability distribution $\mathcal{T}(s', r|s, a)$. What if this is not readily available? Fortunately, an optimal policy can still be found (at least in the tabular case assumed until now) by learning only from samples provided by the environment model. There is a spectrum of sampling-based approaches that ranges from *Monte Carlo* to *Temporal Difference* learning.

Firstly, the *Monte Carlo* approaches are covered. As in the previous section, which defined two-phase processes of evaluation and improvement, Monte Carlo methods can also present two distinct phases: *prediction* and *control*. In fact, this terminology is more common in RL literature, beyond the fundamentals of *policy/value evaluation/iteration*. This distinct two-phase process is desirable because it makes the problem stationary: as the *prediction* step depends on the actions taken in future iterations, which are yielded by a policy that undergoes active learning. Thus, it can be said that the *Monte Carlo* methods covered below also form a General Policy Iteration (GPI).

Monte Carlo works by averaging over sampled returns. Typically, episode rollouts are executed under the policy and rewards are collected for each state. Then, to estimate the expected value of each state, the collected returns starting from each state are averaged. In theory, the optimal value function V_* is computed in the limit, when each state has been visited infinitely many times. For the *first-visit Monte Carlo* algorithm, returns following the first visit to each state are averaged. If all rollouts have the same probability distribution, then returns are independent and identically distributed (i.i.d.) estimates of the value of each state [Bar21], thus, in the limit, their averages converge to the expected returns, which are equal to $V_*(S)$.

The same process can also be applied to find the optimal state-action value function $Q_*(S, A)$. State value functions are insufficient in the case of *Monte Carlo* techniques because, in the absence of the state transition probability distribution, one-step lookahead cannot be executed (crucial for the Bellman update rule). Considerations have to be made in the case of deterministic policies, as the entire state-action space cannot be covered by following a deterministic π. If π is not defined as a stochastic policy, then *exploration* has to be artificially introduced, in order to satisfy the convergence guarantees mentioned in the case of V_* estimation. Essentially, even in deterministic cases, non-zero probabilities have to be assigned to all possible actions at each environment execution step, a technique termed *exploring starts* [Bar21].

Assuming the optimal $Q_\pi(S, A)$ has been found, the policy improvement is trivially done by the following update [Bar21]:

$$\pi_k(s) \leftarrow \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q_{\pi_k}(s, a), \forall s \in \mathcal{S} \tag{2.76}$$

In practice, the assumption of exploring starts (that each state-action pair is visited

enough times) cannot hold except in the most trivial of MDPs. One way to sidestep the assumption is to do policy improvement towards an $\epsilon-greedy$ policy, instead of the greedy policy as above. An ϵ-greedy policy is defined as having a $\frac{\epsilon}{|\mathcal{A}(s)|}$ probability of selecting a *suboptimal* action ($\mathcal{A}(s)$ the set of possible actions from the state s). In *Monte Carlo control*, first-visit returns are averaged for the estimation of $Q(S, A)$ [Bar21], and the ϵ-greedy stochastic policy is updated such that the actions that yield the highest estimated returns are assigned the highest probability, while the rest are assigned $\frac{\epsilon}{|\mathcal{A}(s)|}$. [Bar21] includes a derivation showing that the *policy improvement* theorem is satisfied for any iteration of an ϵ-greedy policy.

Another way of dealing with the *exploration/exploitation* dilemma is to employ *off-policy* algorithms, where the policy used to interact with the environment (and thus collect experience) is different (but not entirely) from the learned policy. The former is exploratory and stochastic, while the latter is the target optimal (possibly deterministic) policy. A first requirement is that all actions taken under the target policy π must always have a non-zero probability of being chosen under the exploratory policy $\bar{\pi}$. This is crucial because the target policy distribution (or deterministic function) is learned using samples from $\bar{\pi}$. Furthermore, the evaluation of the policy at each iteration depends on the expected returns, for which only samples from $\bar{\pi}$ are provided. To estimate the expected returns under π, the Monte Carlo returns are weighed by the relative probability of trajectories under the two policies, a value termed the *importance sampling ratio* ρ.

The probability of a trajectory $\tau = (A_t, S_{t+1}, \dots A_{T-1}, S_T)$ conditioned on S_t and the policy π is [Bar21]:

$$\mathbb{P}(\tau | S_t, A_{t:T-1} \sim \pi) = \prod_{k=t}^{T-1} \pi(A_k|S_k) \cdot \mathbb{P}(S_{k+1}|S_k, A_k) \tag{2.77}$$

At first glance, computing the *importance sampling ratio* between trajectories under π and $\bar{\pi}$ depends on the state transition probability distribution, which is assumed to be inaccessible (in the context of Monte Carlo methods). However, the state transition probability factors cancel out when computing the ratio, as they are identical in the denominator and the numerator. Thus, the final form for ρ is [Bar21]:

$$\rho = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{\bar{\pi}(A_k|S_k)} \tag{2.78}$$

Finally, a method for dealing with the non-stationarity of the value estimates during learning is α-step value function updates. For each sample episode, the difference between the achieved return G_t and the value estimate under state S_t forms a *target* for updating the estimate. The value function estimate is then "moved" towards the target in increments based on a constant parameter α. The update rule for the value estimate is [Bar21]:

$$V_\pi(S_t) \leftarrow V_\pi(S_t) + \alpha \left[G_t - V_\pi\right] \tag{2.79}$$

Naturally, an analogous update rule for a state-action value function $Q_\pi(S, A)$ is defined identically.

Temporal-Difference Learning

Until now, two approaches of policy evaluation and improvement have been presented. In DP methods, the environment model is accessible, and the value estimate under the policy V_π is updated based on transitioning to the next state and then summing the immediate reward with the expected future return by applying the current value function estimate over the next state $V_\pi(S_{t+1})$. In MC methods, on the other hand, the environment model is assumed to be inaccessible, and thus, policy iteration occurs purely from experienced returns. In the former, returns from the next step onwards are estimated. In the latter, expected returns are estimated by averaging sample returns from environment interaction. *Temporal-Difference learning* is a middle ground between the two, because both the model and future returns are inaccessible, learning occurs purely from samples (like in MC), and updates are done based on learned estimate of future returns (like in DP). Despite this, it is surprisingly a more efficient reinforcement learning method, for reasons that will be shown in this section.

Firstly, recall that in MC, the state(-action) value estimate is updated by averaging over returns from sampled episodes, or by shifting the estimate by an α-weighed *target* determined by the error between sampled returns and estimates. Similar targets can also be determined for TD, but they assume a form that is closer to DP targets, which are in turn informed by the Bellman equation. Looking at the one-step Bellman update rule used in *value iteration* (2.75), the value function is *bootstrapped* by estimates from future states, which are also *Bellman backups*. While those targets are computed based on expectations under the environment model, in TD learning, targets are formed directly from encountered reward R_{t+1} and next-state S_{t+1}. Similarly to how target updates for MC were defined in (2.79), the simplest TD target update rule for the value function, based on one-step rewards, is the following [Bar21]:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_t + 1) - V(S_t)\right] \tag{2.80}$$

The similarity to DP targets can be immediately seen in TD target $R_{t+1} + \gamma V(S_t + 1)$. Immediately, this update rule seems to be more advantageous than both DP update rule variants (which require the environment model) and the ones that underline MC methods (which require reaching the end of episodes to compute exact returns). TD updates are done incrementally and online. It is enough to wait for the execution of at least one step before updating the value function estimate. Luckily, convergence can also be shown to hold in TD learning (at least in the tabular case), but proofs vary between the different TD-based algorithms.

As was the case with MC control, TD control algorithm can be *on-policy* and *off-policy*. In both cases, learning the state action value function $Q(S, A)$ is, as stated before, required for evaluating the behavior policy. To this end, an update rule for the state-action value

function can be trivially defined as [Bar21]:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right] \tag{2.81}$$

Again, updates are done purely based on next-step samples $(R_{t+1}, A_{t+1}, O_{t+1})$, starting from S_t and executing A_t, The *SARSA* algorithm is based on this one-step update mechanism, which is reflected in its namesake acronym.

Q-learning, on the other hand, is an *off-policy* algorithm that does not require following the target policy for retrieving the next state-action pair S_{t+1}, A_{t+1}. Instead, the targets for the TD-update are the greedy action (action that yields the highest $Q(S_{t+1}, a)$). In other words, the algorithm directly iterates Q towards the optimal Q_*.

Another approach similar to *SARSA* is to compute the targets as the expected state-action value under the policy, instead of samples from the policy like in *SARSA*. As shown in [Bar21], this variant generally converges faster than *SARSA* due to eliminating the variance from the distribution The algorithm is called *Expected SARSA* and the update rule is [Bar21]:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathbb{E}_\pi \left[Q(S_{t+1}, A_{t+1} | S_{t+1})\right] - Q(S_t, A_t)\right] \tag{2.82}$$

The backups in update rules shown above are all computed by incorporating the encountered next-step reward, thus they are called *TD(0)* . Temporal difference learning updates also work by summing n-step rewards with the future estimate. The Bellman equation (2.67) defines a recursive relation between future returns and value estimates. In other words, $G_t = \mathbb{E}\left[R_{t+1} + \gamma V(S_{t+1})\right] = \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2})\right]$. This implies that the TD-update rule can also incorporate multi-step rewards. If all rewards until the end of the episode (in the episodic case) are incorporated, then the setup is identical to Monte Carlo. Figure 2.12 (taken from [Bar21]) illustrates the spectrum of possible TD backups, ranging from the introduced one-step TD to Monte Carlo, also named ∞*-step TD* in this case.

Updating the value estimate at timestep t using n-step targets requires executing n steps, collecting rewards along the way, and then updating the estimate. Thus, the estimate update actually occurs after $t + n$ steps, using the previous estimate and the return $G_{t:t+n} \coloneqq \sum_{k=t+1}^{t+n} \gamma^{k-t-1} \cdot R_k$ [Bar21]:

$$V_{t+n} \leftarrow V_{t+n-1} + \alpha \left[G_{t:t+n} - V_{t+n-1}(S_t)\right] \tag{2.83}$$

It can be shown that the n-step TD error of estimating V_π is upper bound by the error induced by the estimation at the previous timestep. This is called the *error reduction property* and shows that the learning the value function via TD actually learns V_π, converging in the limit towards V_* [Bar21]. The *error reduction property* is the following:

$$\max_s \left|\mathbb{E}_\pi \left[G_{t:t+n} | S_t = s\right] - V_\pi(s)\right| \leq \gamma^n \max_s \left|V_{t+n-1}(s) - V_\pi(s)\right| \tag{2.84}$$

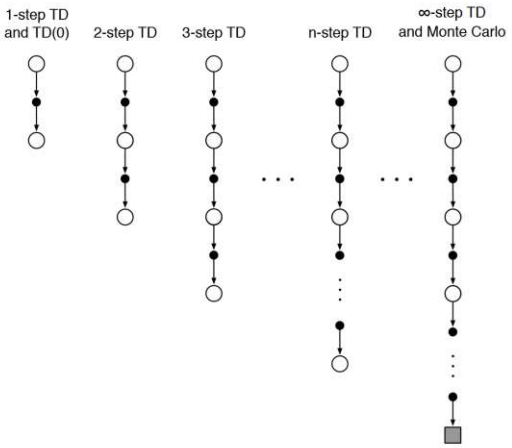


Figure 2.12: The spectrum of n-step TD backups, from [Bar21].

That is, the target error is always less or equal to the previous iteration error for all states.

Substituting the value function in the n-step TD update described above with the state-action value function $Q(S, A)$ yields the update rule for *n-step SARSA*, an on-policy control algorithm similar to the one-step SARSA. The difference is that Q for the encountered state and chosen action at timestep $t + n$ is updated by (2.84) based on the collected return $G_{t:t+n}$. The policy is then updated such that it is ϵ-greedy wrt. Q, as in the case of MC control algorithms, to ensure that the state-action space is covered in the limit.

Similarly, for an *off-policy* TD control algorithm, the TD target can be weighted by the *importance sampling ratio* ρ, taken as a product of ρ over the range $t : t+n$ in the general n-step case [Bar21].

One last important lemma related to TD learning is that targets can also be formed by averaging multi-step returns of variable n. Any number of variable-length multi-step returns can be summed to form a target, as long as they are weighted such that their sum is 1. For example, $0.5 \cdot G_{t:t+1} + 0.5 \cdot G_{t:T}$, where T is the episode length in episodic MDPs, is a valid target that combines a one-step TD target with a Monte Carlo target. Different combinations can yield different results (in terms of convergence rate), depending on the decision process.

The most common combination is the one that incorporates all n-step updates for a given n. In particular, the targets are weighed by λ^{n-1}, and the sum is multiplied by $(1 - \lambda)$

for normalization. The resulting target is the $\lambda - return$ and is defined as [Bar21]:

$$G_t^\lambda := (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}\cdot G_{t:t+n} \tag{2.85}$$

$$:= (1-\lambda)\sum_{n=1}^{H-t-1}\lambda^{n-1}\cdot G_{t:t+n} + \lambda^{H-t-1}\cdot G_t \tag{2.86}$$

The choice of λ again yields a spectrum of targets ranging from one-step TD to MC. It is worth looking at the second formulation of the $\lambda - return$, as it can better illustrate this effect. When $\lambda = 1$, the truncated sum on the left collapses to 0 due to the $1-\lambda$ factor in front, leaving G_t. Consequently, the target is the MC target. On the other hand, if $\lambda = 0$, then the last term in the summation cancels out, together with all terms in the expanded left sum except the one associated with $n = 1$, because its weight is $0^0 = 1$. That specific term is $G_{t:t+1}$, which is equivalent to the one-step TD target (2.80). Because of this, the latter is also called $TD(0)$, while the former is $TD(1)$. In general, $\lambda - returns$ are targets for the general $TD(\lambda)$ algorithm.

The $\lambda - return$ presented above depends on the observed returns $(G_{t:t+k})_{k=1}^n$. This is called the *forward view*. There exists an alternative formulation for the λ targets, called the *backward view*, that can be computed online. In other words, instead of waiting for the next n timesteps to occur, the update is done using returns from the past n timesteps. This introduces a problem known as the *credit assignment* problem. In the *forward view*, n-step returns were weighted by an exponential decay based on λ. Steps further in the future can be said to be assigned a smaller credit, meaning they contribute less towards the error. This can be easily seen in Figure 2.13, taken from [Bar21].

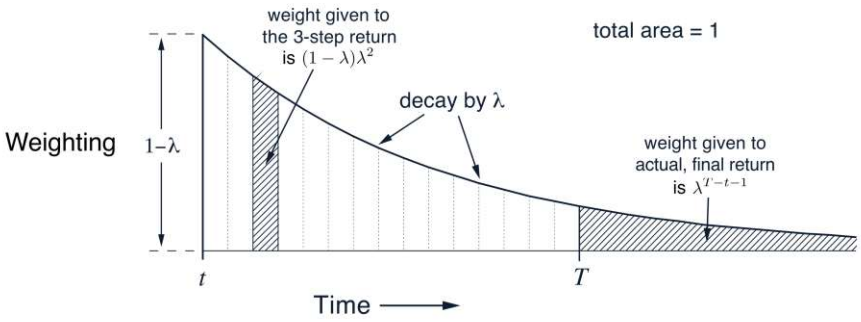


Figure 2.13: λ-controlled exponentially-decaying weighted returns, from [Bar21].

In the *backward view*, however, at least in on-policy control algorithms, such a *credit assignment* is not readily available. To update the current estimate based on returns in the past with $TD(\lambda)$ targets, *eligibility traces* have traditionally been used as an extra weight vector that indicates both the frequency and the recency of each state. In the

tabular setting, the *eligibility* trace of a state is computed as [Bar21]:

$$e_t(S) = \begin{cases} \lambda \cdot \gamma \cdot e_{t-1}(S) & \text{if } S \neq s \\ \lambda \cdot \gamma \cdot e_{t-1}(S) + 1 & \text{if } S = s \end{cases} \tag{2.87}$$

When an update is done, the TD error is proportionally assigned to past states depending on the computed eligibility.

While the *backward view* is provably equivalent to the *forward view* in terms of convergence in the limit (and is included in [Bar21]), it is not covered here, as the methods in this thesis make use of function approximation methods (described in the following Section 2.4.1), for which the credit assignment is done with a different mechanism. Thus, the nuances of eligibility traces for $TD(\lambda)$ in the tabular case are not elaborated here.

Function Approximation Methods

So far, the decision process state, action and reward spaces $\mathcal{S}, \mathcal{A}, \mathcal{R}$ have been assumed to be sufficiently small to explicitly store all possible mappings of the state transition probability, reward, policy and value functions $\mathbb{P}(S'|S, A), R(\cdot|S, A), \pi(S), V(S)$ in memory. In this case, space and time complexities are polynomial at worst. However, there are settings where spaces are infinitely large. For example, the gripper of a robot manipulator is located in its configuration space, a 3D Euclidean space with real values. If this position is included in the state, then $\mathcal{S}$ is continuous. Naturally, a decision process in this case cannot be stored in the memory, and an optimal policy cannot be found in finite time. However, even in cases of discrete-valued states/actions defined on large sets that can be stored in a sufficiently large memory buffer, there is no guarantee that the entire space is explored in the limit, a strict requirement for convergence in the methods covered until now.

To address both issues, the MDP functions have to be approximated using a limited sample set, in a manner that generalizes the entire function domains $\mathcal{S}$, $\mathcal{S} \times \mathcal{A}$. For example, a linear function can be approximated by parameterising the function by a weight vector w in the number of dimensions of the function. In Deep RL, functions are represented as ANNs, trained via supervised learning on targets. In general, function approximations are represented by the weight vectors/matrices, the sizes of which are usually smaller than the size of the state space. A change in one weight influences multiple underlying states.

The state-of-art approach for function approximation is to use deep neural networks as estimators. For example, the value function in *Monte Carlo control* can be represented by a deep neural network parametrized by θ (and denoted as V_θ), which can be learned by SGD. The target in the update rule (2.79) can be used as the basis for the loss function to be used in backpropagation.

Let the target be denoted as U_t in general function approximation for V_π. Under the assumption that samples are i.i.d. random variables, the parameters θ can be

incrementally updated in the direction of the gradient of the loss function wrt θ [Bar21]:

$$\theta \leftarrow \theta - \frac{1}{2}\alpha \left[U_t - V_\theta(S_t)\right] \Delta V_\theta(S_t) \tag{2.88}$$

Under the independent and identically distributed (i.i.d.) assumption of the targets, SGD converges the approximator V_θ to V_π, under the stochastic approximation conditions for the learning rate α:

$$\sum_{t=1}^{\infty} a_t = \infty \tag{2.89}$$

$$\sum_{t=1}^{\infty} a_t^2 < 1 \tag{2.90}$$

For *Monte Carlo control*, the targets are $U_t \coloneqq G_t$ are unbiased estimates ($\mathbb{E}\left[U_t|S_t\right] = V_\pi(S_t)$) and thus, SGD converges to V_π. In the case of $TD(0)$, on the other hand, $U_t \coloneqq R_{t+1} + \gamma \mathbb{E}_\pi \left[Q_\theta(S_{t+1}, A_{t+1}|S_{t+1})\right]$ are biased estimates, since $Q_\theta(S_{t+1}, A_{t+1}|S_{t+1})$ depends on the values of θ at timestep t. The update process is thus prone to divergence. To avoid this, targets are computed using a Q estimate parameterized by a *delayed* and detached copy of θ, denoted as $\bar{\theta}$, as shown in [Lil15] and [MKS+13]. The delayed parameters are updated using the update rule known as *Polyak averaging*, introduced in [PJ92]:

$$\bar{\theta} \leftarrow \tau\theta + (1-\tau)\bar{\theta}, \tau \in [0, 1) \tag{2.91}$$

The concept is also known in literature as having an *exponential moving average* (EMA) of the learned parameters. The gradient is taken wrt the estimate $Q_\theta(S_t, A_t)$, but changes in θ have no effect on the targets, since they are computed with detached parameters. In this case, the optimization process is called *semi-gradient descent* and it does not offer the same convergence guarantees as SGD under the assumption of *iid* examples in the general case.

Furthermore, *off-policy* algorithms with *function approximation* methods introduce further issues that arise from distribution shifts. In *off-policy* algorithms, updates on the target policy π are done using examples (or unbiased returns) under the behaviour policy $\bar{\pi}$. Approximating the two policies increases the variance of estimates and induces mismatched distributions, which can lead to divergence in the worst case.

A better approach when using function approximation methods is to parametrize the policy directly, and use an estimated state-action value function for computing gradients. The policy parameters are then trained via *gradient ascent* by updating them in the positive direction of the gradient of an objective function $\mathcal{J}(\theta)$, which is typically chosen to maximize the excepted return under the policy distribution. The expected return can be provided by a learned parametrized $Q_\zeta(S, \pi(S))$. In this case, the policy is called the *actor*, while the value function is called the *critic*, since it is used to evaluate the policy. All methods that parametrize the policy such that the policy is differentiable wrt its

parameters ($\Delta\pi_\theta(a|s)$ is well defined for all $a \in \mathcal{A}$ and $s \in \mathcal{S}$) fall under the umbrella of *policy gradient methods*.

Computing the gradient of the objective function $\mathcal{J}(\theta)$ requires, at first glance, to compute the gradient of the value function that provides the estimated returns. To illustrate is, let the objective function for the episodic case be the following:

$$\mathcal{J}(\theta) = V_{\pi_\theta}(s_0) \tag{2.92}$$

In the episodic case, this is an appropriate objective, since the policy should maximize the episodic return. In the general case, the estimated return to maximize can be provided by the state-action value function conditioned on expected actions under the policy:

$$V_{\pi_\theta}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \cdot Q_\pi(s, a) \tag{2.93}$$

However, the *policy gradient theorem* states that the gradient of (2.93) is proportional to the expected value of the gradient of the policy distribution times the expected return (which in this case is provided by Q_{π_θ}) under the on-policy state distribution. Thus, the gradient is not propagated through either the on-policy state distribution nor the value function. According to the *policy gradient theorem*, the gradient of (2.93) is:

$$\Delta\mathcal{J}(\theta) \sum_s \mu(s) \sum_a Q_{\pi_\theta}(s, a) \Delta\pi_\theta(a|s) + C \tag{2.94}$$

C is a constant term. $\mu(s)$ is the *on-policy state distribution*, defined as:

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')} \tag{2.95}$$

$\eta(s)$ represents the number of visits of state s in an episode. The proof for the *policy gradient theorem* is sketched in [Bar21].

Any policy distribution can be approximated by parametrization, and the choice depends on the action domain. If the action space is discrete, then one example of distribution is the *categorical distribution*, where each discrete value in the domain set of size K is associated with a probability bucket a random variable can fall in. The parametrized policy distribution can then be modeled as a Gumbel-Softmax distribution [JGP]. This provides a differentiable parametrization for otherwise non-differentiable categorical samples. For discrete probabilities $p_1, p_2, \ldots, p_k$ and k i.i.d. samples drawn from Gumbel(0, 1) $g_1, g_2, \ldots, g_k$, the policy can be defined as

$$\pi_\theta(a_i|s) = \frac{e^{\frac{\log(p_i)+g_i}{\tau}}}{\sum_{j=1}^{k} e^{\frac{\log(p_j)+g_j}{\tau}}} \tag{2.96}$$

Here, τ is the softmax temperature parameter that controls where the Gumbel-Softmax distribution lies on a range from the modeled categorical distribution to a uniform

distribution. As $\tau \to 0$, the former is achieved, while as $\tau \to \infty$, the distribution becomes a uniform distribution. The temperature is typically *annealed* during training, starting from a higher value to reduce variance and annealing towards 0.

The softmax representation of the policy provides smoother gradients than using an ϵ-greedy policy, while keeping the policy stochastic. Thus, in policy gradient methods, it is the preferred approach for representing discrete action spaces.

For continuous action spaces, one assumption that can be made is that actions are sampled from a Gaussian distribution, which has the following *probability density function*:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.97}$$

The parametrized policy can be defined as the pdf of the Gaussian distribution. Its parameters can be split such that one part parameterized an approximator for μ and the other an approximator for σ: $\theta = [\theta_\mu, \theta_\sigma]^\top$. In the case of the latter, the function's codomain must be positive, as $\sigma > 0$ for a Gaussian distribution.

If the action space is of the form $\mathcal{A} = \mathbb{R}^n$, the policy can be parametrized as a multivariate Gaussian distribution $\mathcal{N}_n(\mu, \Sigma)$ instead. Σ denotes the covariance matrix $\mathbb{R}^{n\times n}$ defined as $\Sigma_{i,j} = \mathbb{E}\left[(a_i - \mu_i)(a_j - \mu_j)\right], i, j \in [1, n]$. Σ must be a symmetric, positive definite matrix.

The convergence properties of *policy gradient* under different MDPs are still under study. Perhaps the most comprehensive study in this direction to date is [AKLM21], which provides computational and sample complexity analysis, approximation errors and convergence rates (or lack thereof) for policy gradient with different parametrization methods, applied to tabular and large-space MDPs.

In general in deep RL, the optimization process is generally non-convex due to a variety of reasons. Firstly, MDP functions are represented as deep neural networks, which are typically non-linear. Secondly, the variance induced by the value estimation during gradient ascent for the policy can also lead to saddle points, making the problem non-convex. Finally, the reward signal can be non-continuous, leading to a rugged optimization landscape. For this reason, adaptive gradient-based optimizers are used. An example is the Adam optimizer [KB17], which adapts the learning rate based on estimates of the first and second moments of the gradient.

2.4.2 Model-Based Reinforcement Learning

So far, all reinforcement learning methods presented aimed to find an optimal policy that maximizes the expected returns received from the environment, which were provided by a computed or learned value function. In the case of MC and TD learning, the value estimate was learned from sampled transitions $(S_t, A_t, R_{t+1}, S_{t+1}$ in the case of TD, and sampled episodes in the case of MC. State transition and reward probability distributions were assumed to be inaccessible. These methods fall under the umbrella

of *model-free reinforcement learning*, since the environment model is not used. In DP, however, the environment model was explicitly used to compute an exact value function at each iteration, using the available state-transition and reward distributions. Thus, DP can be categorized as a *model-based reinforcement learning* algorithm.

In model-based RL, the known model is used to look ahead at what will happen in the future, given a certain state and a policy. Updates are done based on the collected information from the look-ahead process. In DP, the target for V_π was the expected return under π, which can be exactly computed given $\mathbb{P}(S_{t+1}|S_t, A_t)$. In this case, the model is used as a *distribution model*, since it generates all possible trajectories weighed by their probability of occurring. The model can also be used to sample trajectories. Given a state s_t and a policy π, a trajectory can be computed by sampling $a_t \sim \pi(\cdot|s_t)$, $s_{t+1} \sim \mathbb{P}(\cdot|s_t, a_t)$, $r_{t+1} \sim R(s_t, a_t)$ at each timestep and concatenating the samples, resulting in the trajectory $\tau = (s_t, a_t, r_{t+1}, s_{t+1}, \dots s_{T-1}, a_{T-1}, r_T, s_T)$. Sampling a trajectory using the environment model is also called a *rollout* in the context of model-based RL.

For conciseness, *model-free* RL methods can also be called *learning* methods, since the value and the policy are learned from environment interactions, whereas *model-based* RL are also called *planning* methods, since learning is done on model interactions (*simulated experience*). A range of algorithms can be constructed by combining *planning* with *learning*, ranging from one side of the spectrum (pure learning, like $TD(\lambda)$) to the other (for example, DP). Out of these, the class of *model-based RL* algorithms has, as a minimum requirement, a *planning* phase. Figure 2.14 illustrates the training sequence for *Dyna-Q*, a model-based algorithm for tabular MDPs. The model is learned using trajectories collected by interacting with the environment. In the next phase, the policy and value estimates are improved using data provided by the environment, the model, or both, using any reinforcement learning technique. The two training phases can occur alternatively or in parallel.

The benefits of augmenting the learning process of the value function with model samples are twofold. Firstly, it can greatly improve sample efficiency. Secondly, it can help reduce variance in V or Q estimation. Since a model is available, the value function at each iteration can be computed by averaging over an arbitrary number of rollouts with the model.

A learned model can also be used for planning at inference-time by rolling out trajectories from the current environment state, a technique called *decision-time planning*. In the general case, an action is chosen by averaging the returns over simulated trajectories and then selecting the action that yields the highest return. Such decision-time planning methods are an instance of MC sampling since they produce Monte Carlo estimates for state-action pairs. Given a learned environment model $\mathcal{T}(S'|S, A)$, $R(S, A)$, and learned policy $\pi(A|S)$ and state-action value function $Q_\pi(S, A)$, starting from a state s_t, a rollout can be executed in the following manner [LXL+]:

$$\max_{a_{t:t+H}} \mathbb{E}_{s_{t+1}\sim\mathbb{P}(s_{t+1}|s_t,a_t)} \left[\sum_{k=t}^{t+H} R(s_k, a_k) \right] \tag{2.98}$$

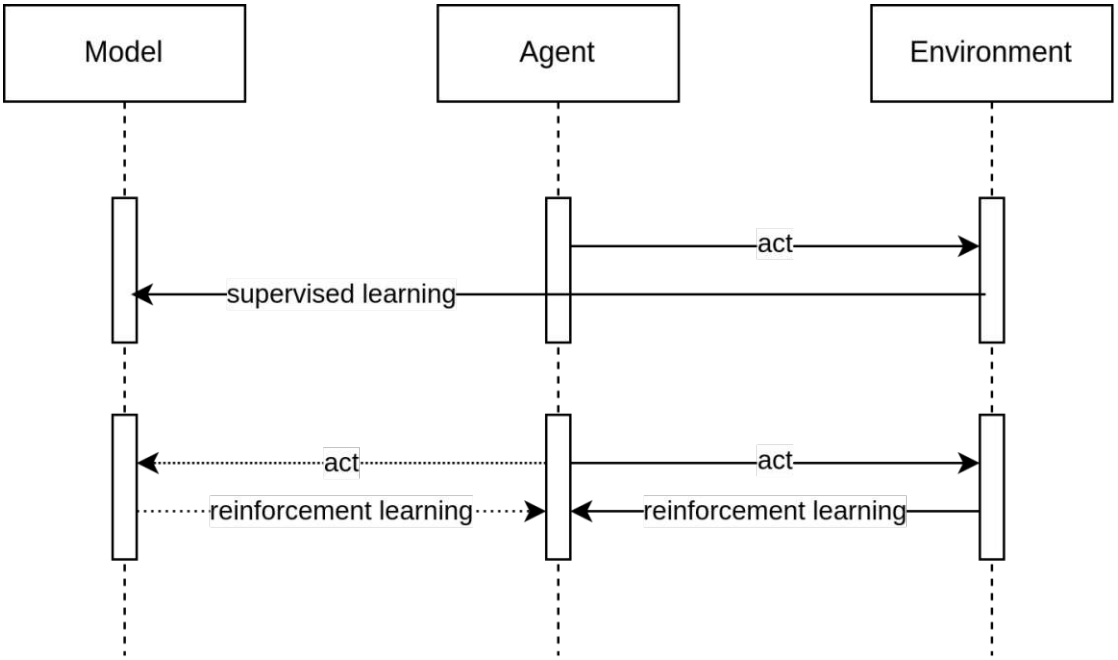


Figure 2.14: Dyna-Q training sequence. The model is trained via supervised learning from experience gathered from interaction with the environment. The agent policy and value functions are trained on targets computed from environment interaction and (optionally) from interacting with the learned model.

One of the fundamental challenges in *model-based RL* is that reinforcement learning from data provided by a learned model can be suboptimal, given that the model is trained from a limited amount of samples from the environment. Suboptimality of the learned policy can then lead to a flawed exploration of the state space, which increases the bias of the learned model. This bias has to either be explicitly accounted for in *model-based RL* algorithms (such as in trajectory-predictive models [LS01]) [AMKL], or analytically bounded [AHL16]. In [AMKL], for example, the training objective is minimizing the multi-step error between predicted future observations and ground-truth observation sequence from the dataset, as opposed to the one-step observation prediction goal.

Upper bounds for a learned model can be derived using the *simulation lemma* from [KS02]. Let $\mathcal{T}_*(S'|S,A)$ be the transition distribution for the environment and $\mathcal{T}_*(S,A)$ be the mode. Similarly, let $\mathcal{T}_\theta(S'|S,A)$ be the learned model and $\mathcal{T}_\theta(S,A)$ its mode. As stated previously in Section 2.4.1, the value estimate error in DP (with a known model) is $\frac{1}{1-\gamma}$. With an approximate model where $\max_{s,a} ||\mathcal{T}_\theta(s,a) - \mathcal{T}_*(s,a)|| \leq \epsilon_\mathcal{T}$ and $||R_\theta(s,a) - R_*(s,a)|| \leq \epsilon_R$, the value estimate bias has the following upper bound [LXL+] [FLRP+19]:

$$\max_s ||V^\pi_\theta(s) - V^\pi_*(s)|| \leq \frac{\gamma\epsilon_\mathcal{T} \max_{r\in\mathcal{R}}}{2(1-\gamma)^2} + \frac{\epsilon_R}{1-\gamma} \tag{2.99}$$

Trajectory Sampling

A learned model of the environment can be used in a variety of ways. As seen in *Dyna-Q* [Sut91] and illustrated in Figure 2.14, the model can be used to provide *simulated experience* as a dataset for value/policy learning. Such a procedure is called **background planning** in [Bar21]. The model can also, however, be used during inference. Instead of acting under the learned policy, an action can be selected by a trajectory optimization procedure that uses the state transition model to compute rollouts and the reward model to compute a cost function. Using a learned model in this way is also called *decision-time planning* [Bar21].

One class of algorithms that plan based on simulated trajectories is called *rollout algorithms*. Rollout algorithms can be seen as computing Monte-Carlo $Q_\pi(s,a)$ estimates for a given state and action s_t, a_t by averaging over returns from simulated trajectories starting from s_t and executing a_t. The estimates can then be used to choose the action a' that maximises $Q_\pi(s,a')$, as in Section 2.4.1. Trajectories are rolled out by sampling actions from a distribution at each time step.

For discrete action spaces, a popular *trajectory sampling* approach is *Monte Carlo Tree Search*, an instance of a heuristic search algorithm. Starting from the current state s_t, MCTS incrementally builds a search tree by sampling actions at each step from a *tree policy* (either randomly, from a learned policy prior or by choosing the action according to expectation maximization of the return), up until a leaf node $s_l, l > t$. A rollout $\tau = (s_l, a_l, s_{l+1}, a_{l+1}, \dots, s_T)$ is then generated starting from s_l, executing $a_l \sim \pi(s_l)$, using the learned transition model to sample $s_{l+1} \sim (P)(\cdot|s_l, a_l)$ and repeating the process until the terminal state s_T is encountered. π can be represented by a uniform distribution or a learned distribution via policy iteration and is called the *rollout policy*. The rollout return approximated by the rewards provided by the learned reward model $\sum_{k=t}^{T} \gamma^{k-1} R(s_k, a_k)$ is used as a target update for all state-action values along the trajectory.

For continuous action spaces, approximate rewards provided by a learned reward model can be used to form a cost function for sampled trajectories. A rollout policy π together with the transition model can be used to generate $\tau_1, \tau_2, \dots, \tau_k$ trajectory samples. The trajectories can be evaluated by a cost function formed by accumulated estimated rewards received:

$$\mathcal{J}(\tau) = \sum_{i=t}^{T} R(s_i, a_i), (s_i, a_i) = \tau_i \tag{2.100}$$

The action that maximises the above cost function is selected:

$$i \leftarrow \underset{i=\bar{1:k}}{\operatorname{argmax}} \mathcal{J}(\tau_i) \tag{2.101}$$

$$a_t \leftarrow \tau_{i_0} \tag{2.102}$$

When the rollout policy π is uniform, the trajectory sampling method is also called *random shooting*. The rollout policy can also be a learned policy using GPI algorithms,

optionally with *background planning*. In that case, simulated trajectories and returns with the model can help correct the bias of a learned policy. The inverse can also be true, however, in that the bias of the learned model can lead to a suboptimal choice of (2.101).

MBRL algorithms that also learn a policy π via MFRL also (implicitly or explicitly) learn a state(-action) value function $V(S)$ or $Q(S, A)$. Let's assume both π and Q_π are learned from environment interactions, simulated experience, or both. Furthermore, assume an infinite horizon. In that case, rollouts are truncated up to a given horizon T and the cost function $\mathcal{J}$ is evaluated only on truncated sampled trajectories. To compute Monte Carlo estimates, $\mathcal{J}$ is extended by adding a terminal value cost term associated with the return accumulated from states at the end of the trajectories onwards. This can be termed as the *long-term return*, and an approximation can be provided by Q. Assume s_T is the last state in a sampled trajectory τ. The long-term return is provided by $Q_\pi(s_T, a_T)$, where $a_T \sim \pi(\cdot|s_T)$. The cost function then becomes:

$$\mathcal{J}(\tau) = \sum_{i=t}^{T} R(s_i, a_i), + Q_\pi(s_T, a_T), (s_i, a_i) = \tau_i, a_T \sim \pi(\cdot|s_T) \tag{2.103}$$

Such planning procedures can be reformulated as instances of learning-based MPC, with the optimization goal being:

$$\max_{a_{1:T}} \mathbb{E}_{\tau=(s_1, a_{1:T}, (s_{t+1} \sim \mathbb{P}(\cdot|s_t, a_t))_{t=1}^{T}} \mathcal{J}(\tau) \tag{2.104}$$

Other decision-time planning approaches aim to mix the benefits of Monte Carlo estimation from rollouts and having a learned policy prior. The former reduces trajectory model and action sampling biases, while the latter guides rollouts toward higher returns. Assume action sequences can be sampled from a vector of multivariate normal distributions $\mathcal{N}(\mu_t, \Sigma_t), t \in [1, T]$, with μ and Σ initialized at each start of the planning procedure. The MPC cost function can be reformulated as finding the distribution parameters that maximise the following objective:

$$\mu, \Sigma = \underset{\mu,\Sigma}{\operatorname{argmax}} \mathbb{E}_{a_{1:T} \sim \mathcal{N}(\mu,\Sigma), \tau=(s_1, a_{1:T}, (s_{t+1} \sim \mathbb{P}(\cdot|s_t, a_t))_{t=1}^{T}} J(\tau) \tag{2.105}$$

An iterative, derivative-free optimization method called the *Cross-Entropy Method* can fit μ and Σ based on evaluating resulting trajectories according to $\mathcal{J}$, as shown in [Rub97]. Let $\{(a_1, a_2, \ldots, a_T)_i\}_{i=1}^{n} \sim \mathcal{N}(\mu, \Sigma)$ be n sampled action sequences of length T. Each sample $(a_1, a_2, \ldots, a_T)_i)$ is evaluated by $\mathcal{J}$ over the trajectory τ_i induced by the sequence and the transition model, and the top k elite samples according to the order induced by $\mathcal{J}$ are selected ($k < n$). The multivariate normal distribution parameters are updated based on the elite samples by the following update rules:

$$\mu \leftarrow \frac{\sum_{i=1}^{k} a_i}{k} \tag{2.106}$$

$$\Sigma \leftarrow \frac{\sum_{i=1}^{k} (a_i - \mu_t)(a_i - \mu_t)^\top}{k} \tag{2.107}$$

Another derivative-free optimization method for fitting μ, Σ is MPPI [WAT]. Assume a family of multivariate normal distribution with parameters initialized at the start of the planning procedure, this time for each sequence timestep $\mathcal{N}(\mu_t, \Sigma_t)$. MPPI is, again, an iterative process that adjusts μ, Σ, this time for each distribution. Let $\{v_i = \mathcal{J}(\tau_i)\}_{i=1}^k$ be the values associated with the elite samples. n samples are then weighted by:

$$w_i = \frac{e^{v_i - \max_{v \in \{v_j\}_{j=1}^k}}}{\sum_{l=1}^k e^{v_l - \max_{v \in \{v_j\}_{j=1}^k}}} \tag{2.108}$$

The multivariate normal distribution parameters can then be adjusted according to the weighted samples, such that action sequences that lead to high-cost trajectories are more likely to be sampled:

$$\mu_t \leftarrow \frac{\sum_{i=1}^k w_i \cdot a_{t,i}}{\sum_{i=1}^k w_i + \epsilon} \tag{2.109}$$

$$\Sigma_t \leftarrow \sqrt{\frac{\sum_{i=1}^k w_i (a_{t,i} - \mu_t)^2}{\sum_{i=1}^k + \epsilon}} \tag{2.110}$$

The convergence of the planning-time optimization of the action distribution parameters can be guided by the learned policy prior. In [HWS] and [HSW], the MPPI sample trajectories set $\{\tau_{1:n}\}$ is formed by a set of trajectories $\{\tau_{1:m}\}$ induced by action sequences sampled from the policy prior π, and $\{\tau_{1:n-m}\}$ trajectories with actions from the family of multivariate normal distribution. Updating the distribution parameters according to (2.109) is done as before. The MC sampling reduces variance, the samples originating from the learned policy prior accelerate convergence of the optimization process, and the locally-optimized distributions correct the policy prior bias.

2.4.3 Partially Observable Markov Decision Processes

Up until now, the presented RL methods assumed a MDP, where the *Markov property* of $S_{t+1} \perp\!\!\!\perp S_{t-1} | S_t$ is always satisfied for all received states. The agent is provided with all the information necessary for decision-making at each timestep, and no *sensor model* is required for state estimation. This assumption does not, however, hold in almost all real-world scenarios. It is said that the environment is not *fully observable*. The underlying state of the environment is not accessible, and the environment emits *observations* O instead of states S.

More formally, MDP are extended with the observation set $\mathcal{O}$, and a conditional observation probability distribution $\mathbb{O}(\cdot|S, A)$, resulting in POMDPs, a 7-tuple $\langle \mathcal{S}, \mathcal{A}, \mathbb{P}, \mathcal{R}, \mathcal{O}, \mathbb{O}, \gamma \rangle$.

There are many approaches to dealing with partial observability, and all of them rely on estimating the state of the process given the past observable information and actions take, which is then used as the state in value/policy estimation. An accurate estimation of the state can then be used for value iteration. This is also termed a *history abstraction*.

There are cases where a history abstraction can precisely determine the state of the process. One such trivial case is an episodic POMDP, where the alternate sequence of past observations and actions $o_1, a_1, o_2, \ldots, a_{t-1}, o_t$ can be concatenated to form a state representation that satisfies the *Markov property*. Let $\bar{S}_t \coloneqq (o_{1:t}, a_{1:t-1})$. It can be immediately seen that $\bar{S}_{t+1} \perp\!\!\!\perp \bar{S}_{t-1} | \bar{S}_t$. However, this approach is not trivial in most cases, as the space complexity of storing past information grows linearly with the history length (which can be ∞ in *infinite-horizon* MDP). Even in episodic cases, the observations can be high dimensional, and it is not feasible to apply value iteration over long sequences of images, for example. Thus, the computational complexity grows exponentially with the state space dimension, as well as the length of the observation history, two phenomena coined as the *curse of dimensionality* and the *curse of history*, respectively [BCC57, PT87]. For the general case, there are two main approaches for recovering Markov states from partial observations, and while both are briefly described in this section, the latter is explored in more detail, being the class the methods presented in this thesis belong to. The spectrum of state-of-the-art approaches is, however, much larger (and they are covered in the comprehensive study [NES+]).

The first category uses Bayesian inference to reduce the POMDP to a MDP by modeling belief distributions over possible states, as described in Section 2.4.1. The *Bayes theorem* (2.7) places a central role in updating the belief states, given incoming observations (which are textitevidence from a Bayesian perspective). Given the transition model $\mathcal{T}(\cdot|S, A)$ and a sensor model $\mathbb{O}(O|S, A)$, the belief states can be inferred by recursively applying Bayes's theorem:

$$b(s') = \mathbb{P}(s'|o, a, b) \tag{2.111}$$

$$= \frac{\mathbb{P}(o|s', a, b) \cdot \mathbb{P}(s'|a, b)}{\mathbb{P}(o|a, b)} \tag{2.112}$$

$$= \frac{\mathbb{P}(o|s', a) \cdot \sum_{s \in \mathcal{S}} \mathbb{P}(s'|s, a) \cdot \mathbb{P}(s|a, b)}{\mathbb{P}(o|a, b)} \tag{2.113}$$

$$= \frac{\mathbb{O}(o|s', a) \cdot \sum_{s \in \mathcal{S}} \mathbb{P}(s'|s, a) \cdot b(s)}{\mathbb{P}(o|a, b)} \tag{2.114}$$

Function approximation methods can be used to either parametrize the *sensor model* and the *transition model*, or $\mathbb{P}(S'|O, A, B)$ directly. $b(s)$ can also be inferred via *variational inference* by using $\mathbb{P}(S_t|O_{1:t}, A_{1:t-1})$ as an approximate variational posterior Q. The parameters of Q can be trained by minimizing the KL divergence (2.14) between the approximate posterior $Q(S_t|O_{1:t}, A_{1:t-1})$ and a temporal prior $P(S_t|S_{t-1}, A_{t-1})$. Since the true posterior is intractable, Evidence Lower Bound Objective (ELBO) is used as a surrogate loss:

$$\log \mathbb{P}(o_{1:t} \mid a_{1:t-1}) \geq \mathbb{E}_{Q(s_{1:t})} \left[\sum_{t=1}^{T} \log \mathbb{O}(o_t \mid s_t, a_{t-1}) \right] - D_{\mathrm{KL}}(Q(s_{1:T}) \parallel \mathbb{P}(s_{1:T} \mid a_{1:T-1})) \tag{2.115}$$

P is typically initialized as a Dirichlet distribution (commonly used as prior distributions), and updated by conditioning over the entire history of observations and actions. In practice, the history $(o_{1:t}, a_{1:t})$ is typically encoded by an RNN, resulting in latent states z_t.

The second category tries to approximate the underlying state space as a latent state space encoded by a history compression function. Formally, let $h_t \coloneqq (o_{1:t}, a_{1:t-1})$ be the sequence of past observations and actions concatenated with the current observation, and let $\mathcal{H}$ be the space of possible histories for a given POMDP. As stated before, the Markov property is satisfied if states are represented by complete histories and thus, POMDPs can be treated as MDP where $\mathcal{S} = \mathcal{H}$. In practice, this is not tractable. Furthermore, let $\phi : \mathcal{H} \to \mathcal{Z}$ be a *history compression* function that maps histories to latent variables called *information states*. For a given history h, $\phi(h)$ represents a *sufficient statistic* for h if, informally, its latent encoding is informationally equivalent to h. The general criteria for a sufficient statistic is stated in [Bar21] as satisfying:

$$\phi(h) = \phi(h') \Rightarrow \mathbb{P}(\tau|h) = \mathbb{P}(\tau|h'), \forall h, h' \in \mathcal{H}, \tau \in \{\mathcal{A} \times \mathcal{O}\}^* \tag{2.116}$$

In other words, the probabilities for trajectories conditioned on two histories are equal if their associated information states are equal. An equivalent definition is provided in [CPP09] as Definition 4.2. Under the latter definition, $\phi(h)$ and $\phi(h')$ are *belief-trajectory equivalent*. The trajectories τ are also defined as *tests* in [LS01], since they are used to test equivalence relations between information states. Theorem 5 of [SSSM22] implies any GPI can be applied on information states. Definition 3 from [SSSM22] formalizes the notion of information states and states necessary and sufficient criteria for σ mappings to be informationally equivalent to history inputs. The sequence of functions $(\phi_t : \mathcal{H}_t \to \mathcal{Z}_t)_{t=1}^T$ (the sequence is introduced due to the iterative updates in GPI) is an *information state generator* if for any realizations $h_t \in \mathcal{H}$, $a_t \in \mathcal{A}$ at timestep t, it satisfies the following *self-predictive abstraction* conditions [NES+]:

(RP) Sufficient for reward prediction:

$$\mathbb{E}\left[R_t|h_t, a_t\right] = \mathbb{E}\left[R_t|\sigma(h_t), a_t)\right] \tag{2.117}$$

(ZP) Self-predictive:

$$\mathbb{P}(Z'|h, a) = \mathbb{P}(z'|\phi(h), a) \tag{2.118}$$

(EZP) Alternatively, predictive of the expected next latent state:

$$\mathbb{E}(Z'|h, a) = \mathbb{E}(z'|\phi(h), a) \tag{2.119}$$

Satisfying RP is a hard requirement. Otherwise, there exist trivial mappings that satisfy ZP without being informative of the history. For example, any constant mapping $\phi(h) = c$ yields a uniform distribution over actions.

The *self-predictive abstraction* conditions above can be replaced by the following set of stronger conditions, where sufficient properties are defined in terms of observation realizations o_t, o_{t+1}:

(Rec) Satisfies a recurrence relation:

$$\exists \varphi : \mathcal{Z} \times \mathcal{A} \times \mathcal{O} \to \mathcal{Z} \text{ s.t. } \phi_{t+1}(h_{t+1}) = \varphi_t(\phi_t(h_t), o_t, a_t) \tag{2.120}$$

(OP) Sufficient for observation prediction:

$$\mathbb{P}(o_{t+1}|h_t, a_t) = \mathbb{P}(o_{t+1}|\phi(h_t), a_t) \tag{2.121}$$

While Rec and OP are stronger conditions (Rec $\wedge$ OP $\to$ ZP, they are easier to verify in some cases. To satisfy Rec, for example, σ can be modeled as an RNN, with φ being RNN's internal state update rule. In POMDPs with high-dimensional observations, an additional *observation reconstruction* condition is introduced:

(OR) Observations can be reconstructed from information states

$$\exists \psi_t : \mathcal{Z}_t \to \mathcal{O} \text{ s.t. } \psi_t(\phi_t(h)) = o_t \tag{2.122}$$

Definition 3 from [SSSM22] is also equivalent to Definition 4.5 of *weak belief bisimulation relation* from [CPP09]

The properties notation used here (ZP, RP, Rec, OP and OR) is borrowed from [NES+], which presents a unified view of RL algorithms for POMDPs in terms of satisfied *self-predictive* properties and implication relations between them.

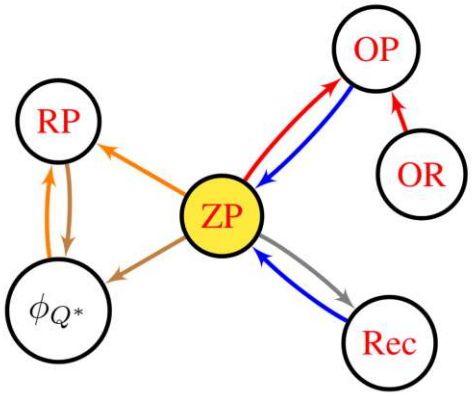


Figure 2.15: Implication graph for information state generator conditions, from [NES+]. Each coloring for the edges denotes a separate implication. ϕ_{Q^*} is equivalent to the optimal information state-action value function that safisfies (2.128).

Recall the *belief trajectory equivalence* tests (2.116), which are formulated as equivalence checks for trajectory rollouts given two identical histories. The predictive conditions for an *information state generator* OP and ZP are instead formulated as probability distribution equivalences for one-step predictions. However, the tests can still be expressed as a recursive application of OP conditioned on the history and action at the previous step, followed by Rec.

$$\begin{aligned}
\mathbb{P}(o_{t:t+k}|h_t, a_{t:t+k}) &= \mathbb{P}(o_{t+k}|h_{t+k-1}, a_{t+k-1}) \cdot \mathbb{P}(h_{t+k-1}) && (2.123)\\
&= \mathbb{P}(o_{t+k}|h_{t+k-1}, a_{t+k-1}) \cdot \mathbb{P}(h_{t+k-2}, o_{t+k-1}, a_{t+k-1}) && (2.124)\\
&= \mathbb{P}(o_{t+k}|\phi(h_{t+k-1}), a_{t+k-1}) \cdot \mathbb{P}(\phi(h_{t+k-1}) && (2.125)\\
&= \mathbb{P}(o_{t+k}|\phi(h_{t+k-1}), a_{t+k-1}) \cdot \mathbb{P}(\varphi(\phi(h_{t+k-2}), o_{t+k-1}, a_{t+k-1}) && (2.126)\\
& && (2.127)
\end{aligned}$$

This leads to the result that *belief trajectory equivalence* tests of arbitrary length can be satisfied as long as OP holds (one-step predictions are accurate). Stronger assertions can be made regarding multi-step predictions. Let $\text{ZP}^* := \mathbb{P}(z_{t+k}|h_t, a_{t:t+k-1})$ denote multi-step ZP. Similarly, let $\text{OP}^* := \mathbb{P}(o_{t+k}|h_t, a_{t:t+k-1})$ denote multi-step OP. It can be shown $\text{ZP} \rightarrow \text{ZP}^*$ for POMDPs, and the proof is sketched in [NES+]. The same cannot be said for OP, however. While $\text{OP} \rightarrow \text{OP}^*$ holds for MDPs (again, the proof can be found in [NES+]), it does not generally hold for POMDPs. The MDP in Figure 2.16 (taken from [CPP09]) serves as a counterexample. In fact, it is a counterexample Markov Model, since transitions are not conditioned on actions. On the left, we have $\mathbb{P}(u_1|s) = \mathbb{P}(u_1|t_1) \cdot \mathbb{P}(t_1|s) = 1 \cdot 0.5 = 0.5$. On the right, we have $\mathbb{P}(u_1'|s') = \mathbb{P}(u_1'|t') \cdot \mathbb{P}(t'|s') = 0.5 \cdot 1 = 0.5$. So we have $\mathbb{P}(u_1|s) = \mathbb{P}(u_1|s')$. However, $\mathbb{P}(t_1|s) \neq \mathbb{P}(t'|s')$.

Theorem 5 from [SSSM22] states that, given state-action value function over the information space $Q_\pi : \mathcal{Z} \times \mathcal{A} \rightarrow \mathbb{R}$, defined as the expected return given action a and thereafter following policy π, we have $Q_\pi(h, a) = Q_\pi(\sigma(h), a)$. Furthermore, there exists an optimal history-action value function $Q_*(h, a) = Q_*(\sigma(h), a)$ that satisfies:

$$\sigma(h) = \sigma(h') \Rightarrow Q_*(h, a) = Q_*(h', a) \tag{2.128}$$

Q_* can be computed (or approximated) by any model-free method described in the previous section.

Delayed Markov Decision Processes

MDPs have so far been assumed to be time-invariant discrete piecewise sequential decision processes. While state-action transitions of the form $(s_t, a_t, s_{t+1})_{t=1}^T$ are sequentially defined, the time delta between two states is always one unit, and the agent is assumed to execute an action at the same time as the environment emits its state. Many tasks that can be approached with RL methods, however, operate in continuous time, and the dynamics of the system are expressed as systems of ordinary differential equations. Closed loop control systems can exhibit two delay types: feedback delay and control delay. From the view of the agent-environment interaction loop in MDPs as a closed-loop control

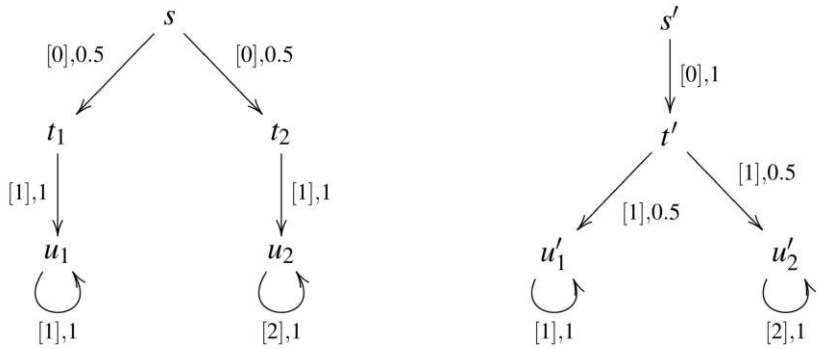


Figure 2.16: Counterexample MDP for OP*, taken from [CPP09]. Edge weights (values without surrounding brackets) indicate transition probabilities.

system, the former type of delay can be analogous to delayed observations (feedback from the environment), while the latter can be related to a delayed effect of the policy on the environment.

[WNLL09] formulates a *constant-delay* MDP (CDMDP) $\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma, k$, where k is the constant state feedback delay. Furthermore, the work explores different strategies to handle state feedback delay, with the most significant (and widely used) being *the augmented approach*. Here, the constant-delay is reduced to an MDP by mapping the state-space to $\mathbf{I}_k \coloneqq \mathcal{S} \times \mathcal{A}^k$, where the state at timestep t I_t is constructed from the received state s_t and the following k actions $\{a_{t:t+k}\}$.

[KE03] covers Markov decision processes with both constant and stochastic observation, action, or reward delays. Similarly to *CDMDP*, an MDP structure can be extended with constants k_o, k_a, k_c, representing the constant state feedback delay, the constant action delay (from action emission by the agent until its effect on the environment) and the constant reward delay. Generalizing the reduction to an MDP applied in the former case, the state space can be redefined as $\mathcal{S}^{k_o} \times \mathcal{A}^{k_a}$, meaning each state is mapped to the last k state-action transitions. For the stochastic delay process (SDMDP), k_o, k_a, k_c are random natural numbers. [KE03] proceeds to provide reductions from SDMDP to MDP.

2.4.4 Continuous-Time Reinforcement Learning

Finally, a fundamentally different approach that implicitly deals with delayed timesteps is the continuous-time model-based reinforcement learning framework proposed in [YHL21]. The work formulates a continuous-time decision process, with the state transition function is an ordinary differential equation of the form $\dot{s}(t) = \frac{ds(t)}{dt} = \mathcal{T}(s(t), a(t))$. The state at timestep t can be explicitly computed as $s(t) = s_0 + \int_0^t f(s(k), a(k))dk$. The continuous-time value function is defined as [YHL21]:

$$V(s(t)) = \int_0^T e^{\frac{-k-t}{\gamma}} R(s(k), a(k)) \tag{2.129}$$

CHAPTER 3

Related Work

This chapter presents an overview of the state of the art in MBRL in the first Section 3.1, while the second Section 3.2 covers related approaches to solving POMDPs, with a focus on partially-observable continuous control tasks.

3.1 State of the Art in Continuous Control Agents

Up until recently, the state of the art in RL for continuous control had been defined by actor-critic model-free algorithms. Since this group of tasks is defined by continuous state spaces (and usually continuous action spaces), the research focus has been on function approximation methods using deep neural networks (in other words, policy gradient methods). Since the importance of safety and sample efficiency in control tasks in robotics shifted the focus to *off-policy* approaches, where samples from the experience collected by the behaviour policy can be used multiple times during training the target policy.

The first groundbreaking off-policy policy gradient algorithm designed for continuous action spaces has been Deep Deterministic Policy Gradient (DDPG) [Lil15], which trains a deterministic policy and a state-action value function Q as the critic via policy gradient from offline data. To encourage exploration, noise is added to the deterministic actions from the behavior policy at train-time. Importantly, the targets for Q are computed using an Exponential Moving Average (EMA) of the network parameters for training stability, a notion introduced in [MKS+13]. *TD3* [FHM18] extended DDPG by introducing the notion of multiple critic networks, to avoid the overestimation bias induced by off-policy learning. Soft Actor Critic (SAC) trains a stochastic policy instead, which is a better fit for robotics tasks. The work introduced *entropy regularization* as an additional surrogate loss term for training both the actor and the critic. Thus, the policy is encouraged to balance a trade-off between exploiting increasing returns and exploring the state space.

Early MBRL methods focused on using a learned model (trained on short sequences) to augment the training dataset for the actor and critic models [JFZL] (learning in imagination). However, the myopic rollout horizon of the learned model can populate the dataset with noisy samples, which in turn leads to an overestimation bias for the actor/critic networks. The quality of the learned models started to improve with Planning in Latent Space (PlaNet) [SRD+20]. The work introduced learning latent dynamics models (Recurrent State-Space Model (RSSM)), which enable longer rollouts for learning in imagination, as well as introducing robustness to noise by using a probabilistic model. Planning at inference-time is then done using CEM. The first Dreamer version [HLBN] learns a Recurrent State-Space Model (RSSM) [DDS+] trained via variational inference, similarly to PlaNet. The approach also trains actor and critic networks using the simulated data, which is significantly more performant than the CEM planer. The model is then used for generating trajectories for actor/critic training as before. The second version [HLNB] focuses on visual inputs, and uses categorical distributions for the latent state space instead. Finally, DreamerV3 [HPBL] is a hyperparemeter-free MBRL approach that is performant across a variety of tasks. Importantly, it is the first model-based approach that shows significant improvements in performance and sample efficiency over previous state-of-art model-free algorithms in continuous control problems. It is, however, not a sample-efficient algorithm, and the large number of learnable parameters can make the approach unfeasible for hardware deployment.

A state-of-art MBRL algorithm that is designed for control tasks is TD-MPC2 [HSW], which is extended in this thesis. In *TD-MPC2* and similar approaches [DR11] [SRD+20], the learned model is used in planning by generating multiple trajectories in latent space, which are then used to derive the best action using optimization methods such as CEM and MPC. Instead of learning an RSSM, TD-MPC2 learns a self-predictive latent dynamics model, which is more efficient in terms of space complexity. Furthermore, to estimate long-term returns beyond the rollout horizon during planning, the learned model also includes a state-action value estimator, which is then used to learn a policy prior via policy gradient and entropy regularization. The policy prior is used to bootstrap the MPPI optimizer employed during planning by providing quality action samples. TD-MPC2 is currently state of the art in continuous control tasks, outperforming traditional model-free choices.

3.2 Related Works on Partially Observable Continuous Control Problems

Various works have explored encoding statistically sufficient representations for dynamical system states, given unreliable sensor data. One line of work has emerged from the theoretical foundations on state abstractions [LWL06] and bisimulation relations in MDPs [GDG03] [CPP09]. Predictive state representations [LS01] defines a maximal subset of core tests for latent states equivalence as multi-step observation-action prediction equivalence. States are represented as probabilities of predicted trajectories satisfying

the core tests. Thus, states are not represented as encoded observation-action histories, but rather as probabilities of predicting certain trajectories.

Approximate information states [SSSM22] define statistically-sufficient representations of the observation-action history, in terms of next-state predictions. The approximate states are encoded by *information state generators*, and they are capable of next-state and next-reward predictions. [LBE24] implement an approximate state generator using an RSSM trained via variational inference, similarly to Dreamer [HPBL].

[ZLP+21] unifies the theory on predictive state representations (for multi-step predictions) and bisimulation relations (for state abstraction equivalence relations) by extending the latter to POMDPs and history spaces. The work introduces the concept of *causal states*, which are predictive state representations on histories encoded by RNNs. [HPBL] and [HDT19] learn belief states via variational inference with an ELBO surrogate objective. On the other hand, [ZMC+21] uses bisimulation metrics in the actual state and latent state spaces, in order to train a self-predictive and reward-predictive encoder. SAC is then applied on the latent state space.

Other works simply (but effectively) augment classical model-free architectures with *memory* components. [KOD+19] extends the Deep Q-Network algorithm (Q-learning with neural networks) by incorporating RNNs, and studies the challenges of training RNNs via T-BPTT on sequences sampled from a replay buffer (the experience dataset). [MGK] extends the TD3 model-free algorithm to POMDPs by adding LSTM-based history encoders to both the actor and the critic networks. Its effectiveness is validated on partially observable tasks that served as inspiration to the current work, such as no velocity states and Gaussian sensor noise.

Another related line of research focuses on using continuous-time neural networks for modeling the policy or the transition dynamics. In [ZZH+23], a GRU-ODE based history encoder is employed within an actor-critic framework to address the challenges posed by continuous-time environments with irregularly-sampled observations. This approach integrates policy gradient training with KL divergence as a loss function for latent state alignment. The Neural ODE-based encoder captures the underlying system dynamics from past observations and actions, even at irregular timesteps, and is evaluated on MuJoCo classical control simulations with stochastic observation delays.

A more straightforward, but principled model-based approach models both the transition dynamics and the policy using neural ODEs [Chi], which are trained end-to-end to minimize trajectory errors. The approach is principled because it closely models dynamical systems as a system of coupled ODEs: one for the state evolution, the other for the controller. A similar approach is studied in [Sch08], but the coupled neural ODEs from the former work are replaced by traditional RNNs. The coupled architecture is trained in phases instead: the first phase trains the state dynamics RNN via regression on predicted observation; the second phase trains the control RNN via policy gradient ascent.

CHAPTER 4

Method

4.1 Problem Statement

The focus of this thesis is to explore ways of modeling different components of in common reinforcement learning system using continuous-time neural networks. In particular, Liquid Time-constant Neural Networks (LTC) [HLA+21] and its closed form, Continuous-time Neural Networks (CfC) [HLA+22], are used. These networks fall under the umbrella of recurrent neural networks, and there is a wide existing body of research into employing this family of neural networks in RL pipelines. The two main methods described in this chapter are built upon existing end-to-end solutions that solve the general reinforcement learning problem class. The focus, however, is on the subclass of single-agent continuous control problems in partially-observable environments, and the methods covered in this thesis are evaluated on specific instances of continuous control problems, the choice of which is motivated in Chapter 5.

The single-agent reinforcement learning problem can be formulated as a discrete-time sequential decision-making problem: given an agent placed in an environment, as well as the state of the agent at the current timestep and observation from the environment, find the sequence of actions the agent should take in order to reach a goal state. Formally, RL algorithms are solutions to Markov Decision Processes, previously described in Section 2.4.1. The methods described in this work are oriented towards solving continuous control problems expressed as POMDPs instead, which are formalized in Section 2.4.3.

Given that LTC and CfC networks efficiently model dynamic systems with varying time-constants, they are especially suitable for robotics applications with perception noise and latency as constraints. As such, relevant RL problem instances to solve using the methods covered are motion control tasks with noisy sensor input, in the absence of a known dynamical model of the agent. Since the lack of a derived model of the robot (and consequently, forward and inverse kinematics/dynamics) is assumed, task-space motion

control is excluded from the scope. Thus, three problem instances with distinct dynamic systems of the form (2.47) (where $x \in \mathbb{R}^n$ is the system state and $u \in \mathbb{R}^m$ is the control signal) are chosen. Here, they are briefly formalized. Implementation details are covered in Chapter 5.

The methods proposed in this work approximately reduce POMDPs to an MDP, and the chosen approach is to map the history space of observations and actions $\mathcal{H}$ to a Markovian state space $\mathcal{Z}$ using AIS defined as LNNs. AIS are defined in [SSSM22] and covered in Section 2.4.3 of this work. Throughout this chapter, the POMDP is denoted by M, and the estimated MDP (also called an *augmented MDP* in literature [FLRP+19] is denoted by $\hat{M}$.

Cartpole

Cartpole represents an interesting control problem for RL approaches because the system is inherently unstable (due to gravity), is underactuated and the only actuator present is discrete. Nevertheless, it is a well-understood control problem with known solutions and transparent equations of motion/system dynamics.

The setup for *cartpole* used is defined in [BSA83] as a cart that is able to move on a restricted 2D plane, and a pole that is hinged on top of the cart in the middle. The system can be controlled by applying a fixed linear force F to the left or the right side of the cart. No other actuator is present.

Formally, the state of the system is defined by the following components:

1. χ: position of the cart on the 2D plane
2. $\dot{\chi}$: linear velocity of the cart
3. θ: angle of the pole with the vertical axis
4. $\dot{\theta}$: angular velocity of the pole

The system is further parameterized by the masses of the cart M and the pole m in kilograms and the half-length of the pole l in meters. The general control problem is defined as: starting from an initial state of the system (with $\dot{\chi} = 0$ and $\dot{\theta} = 0$), the goal is to balance the pole vertically ($\theta \sim 0$) for as long as possible, while keeping the cart in the restricted space on the plane. The system is illustrated in Figure 4.1, taken from [BSA83].

The system dynamics at timestep t is defined by the following equation in matrix form:

$$\begin{bmatrix} M+m & ml\cos\theta_t \\ ml\cos\theta_t & \frac{3}{4}ml^2 \end{bmatrix} \begin{bmatrix} \ddot{\chi}_t \\ \ddot{\theta}_t \end{bmatrix} - \begin{bmatrix} ml\dot{\theta}_t^2 \sin\theta_t \\ mgl\sin\theta_t \end{bmatrix} = \begin{bmatrix} F \\ 0 \end{bmatrix} \tag{4.1}$$

g is the gravitational acceleration. The physical parameters of the system conform to [BSA83].

In the context of this work, MDP formulations for the cartpole swingup control problem can have the two reward functions: a dense, continuous reward signal, or sparse rewards. The definition of the former is the one used in [TTM+]:

$$R_{\text{dense}}(s_t, a_t) = R_{\text{upright}}(s_t, a_t) \cdot R_{\text{control}}(s_t, a_t) \cdot R_{\text{vel}}(s_t, a_t) \cdot R_{\text{centered}}(s_t, a_t)$$

where a_t is the constant force F_x applied to the cart at timestep t:

$$\begin{cases} R_{\text{upright}}(s_t, a_t) & = \frac{1+\cos\theta_{t+1}}{2} \\ R_{\text{control}}(s_t, a_t) & = \frac{4+F_{x_t}}{5} \\ R_{\text{vel}}(s_t, a_t) & = \frac{1+\dot{\theta}_{t+1}}{2} \\ R_{\text{centered}}(s_t, a_t) & = \frac{1+\chi_{t+1}}{2} \end{cases} \tag{4.2}$$

The reward signal above includes defines the *subgoals* of keeping the pole in an upwards orientation, of using as little actuation over time as possible, of constraining the angular velocity of the pole, and of bringing the cart to the center position on the plane.

A sparse reward function returns a reward of 1 for every timestep where the angle of the pole is in a tolerance range around the upright position, and the cart position is in a tolerance range of $[-0.25m, 0.25m]$:

$$R_{sparse}(s_t, a_t) = \begin{cases} 1, & \text{if } \chi_{t+1} \in [-0.25, 0.25] \wedge |\theta_{t+1}| < \epsilon_{\text{upright}} \\ 0, & \text{otherwise} \end{cases} \tag{4.3}$$

Variations on the control problem (such as starting state, state bounds, control objective) have been previously used. For example, if the initial angle of the pole is such that the pole is oriented "downwards", the control objective includes swinging the pole up and then balancing it. The specific details used in this work are described in Chapter 5.

Acrobot

The *acrobot* system, initially defined in [Spo02], is a double pendulum chain comprised of two links, with one end of the chain fixed to the x-axis. Only the joint that connects the two links is actuated. A classical control problem is to actuate the system such that the free end of the chain is balanced upwards, similar to the *cartpole swingup* control problem. This is a hard continuous control problem, as the system is underactuated.

The system state is defined as:

1. θ_1: angle of fixed joint w.r.t. the vertical axis
2. θ_2: angle of joint between the chain links

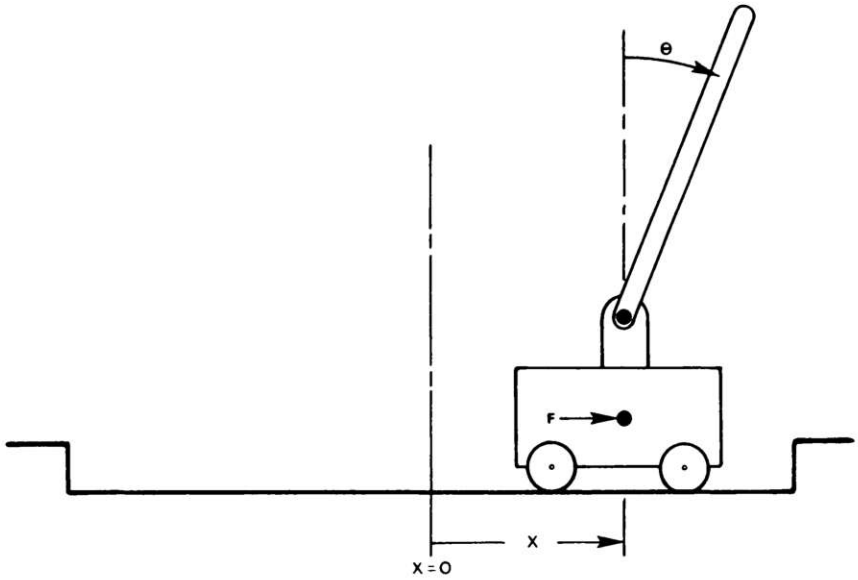


Figure 4.1: Cartpole system, from [BSA83]. x is equivalent to χ, while F represents the constant force to be applied the side of the cart.

3. $\dot{\theta_1}$: angular velocity of the fixed joint
4. $\dot{theta_2}$: angular velocity of the joint between the chain links

The system dynamics can be described by the following equation in matrix form [Cou02]:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{4.4}$$

where:

$$\begin{cases} a_{11} = \left(\frac{4}{3}m_1 + 4m_2\right) l_1^2, \\ a_{22} = \frac{4}{3}m_2 l_2^2, \\ a_{12} = a_{21} = 2m_2 l_1 l_2 \cos(\theta_1 - \theta_2), \\ b_1 = 2m_2 l_2 l_1 \dot{\theta}_2^2 \sin(\theta_2 - \theta_1) + (m_1 + 2m_2) l_1 g \sin\theta_1 - \mu_1 \dot{\theta}_1 - u, \\ b_2 = 2m_2 l_1 l_2 \dot{\theta}_1^2 \sin(\theta_1 - \theta_2) + m_2 l_2 g \sin\theta_2 - \mu_2 \dot{\theta}_2 + u. \end{cases} \tag{4.5}$$

The reward function defined for MDP formulations of the *acrobot swingup* control task is the following:

$$R(s_t, a_t) = 0.5 \cdot \text{clip}(|\cos\theta_{1_{t+1}}|, \epsilon_{\text{radius}}) + 0.5 \cdot \text{clip}(|\cos\theta_{2_{t+1}}|, \epsilon_{\text{radius}}) \tag{4.6}$$

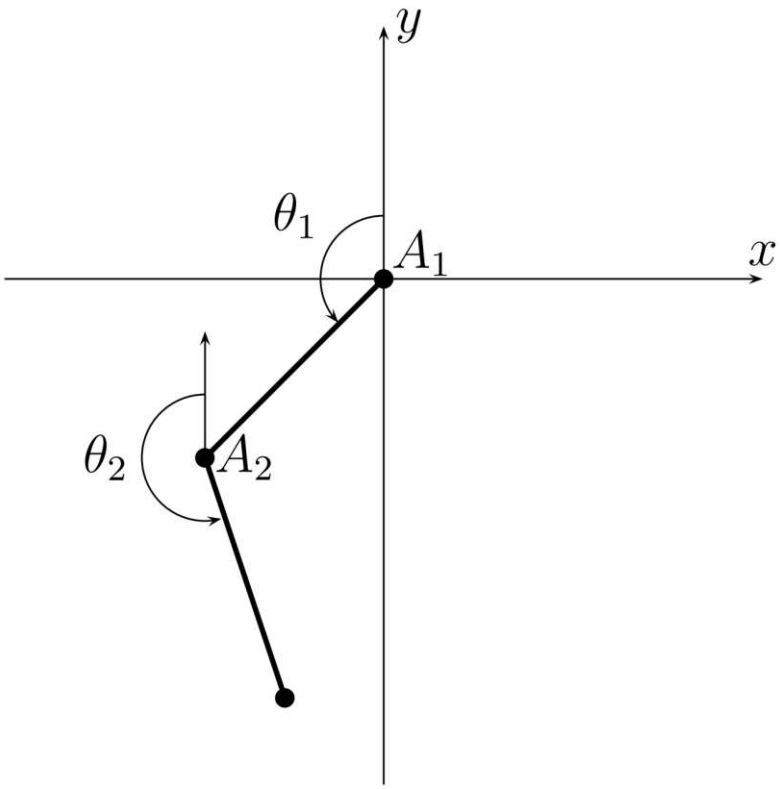


Figure 4.2: The acrobot system, from [Cou02].

The clip function clips the value of the first parameter to 0 if it is larger than the second argument. Intuitively, the desired orientation for both joints is upwards, and rewards are only received if the respective orientation angles fall in a radius tolerance of ϵ_{radius} around 0° from the vertical axis.

Planar walker

To explore the ability of the proposed methods to solve locomotion tasks, a bipedal robot is chosen as the agent. The robot is modeled as a 7-link, 6-joint biped robot, where the legs are two open chains connected at the hip with one torso link.

This system is specifically chosen as an instance of an underactuated locomotion system, which requires more challenging control design and system analysis.

The task under test requires maintaining a running gait, which is defined as an alternating stance phase/flight phase cycle. During the stance phase, one leg is in no-slip contact with the ground, while the other swings freely and during flight, both legs are in the air [GK17]. The starting state is a double support phase, where both legs are in contact

with the ground. During running, the dynamics are described by a spring-loaded inverted pendulum system, as shown in Figure 4.3.

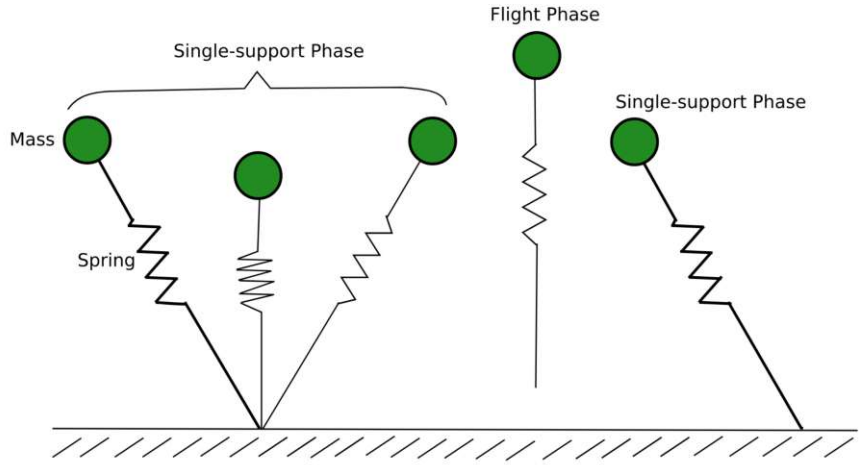


Figure 4.3: Spring-loaded inverted pendulum model of biped during running [GK17].

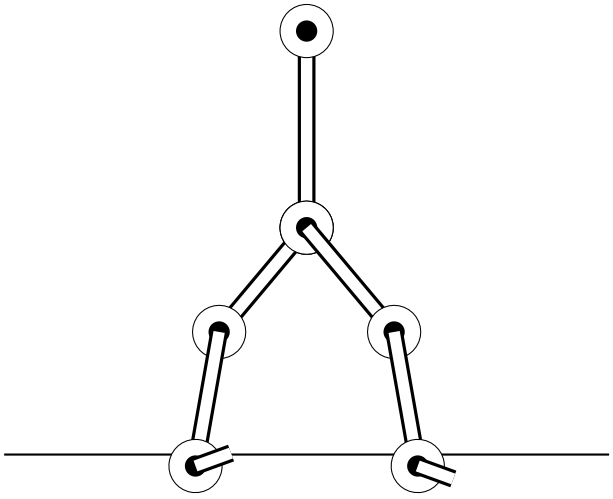


Figure 4.4: Kinematics of the 2D bipedal walker.

4.2 TD-MPC2

TD-MPC2 is a highly scalable, hyperparameter-free state-of-the-art Model-Based Reinforcement Learning algorithm (Section 2.4.2) designed to solve continuous control tasks. It jointly trains a world model with surrogate loss terms for latent state transition, reward and state-action value predictions, as well as a stochastic maximum entropy policy prior, used to guide an MPPI for local trajectory optimization. The optimization cost is the

expected discounted sum of rewards for the optimization horizon, plus the long-term return, both of which are provided by the learned world model [HSW].

The overall learning goal of the world model is to learn state representations for model-based planning with trajectory sampling, a method described in Section 2.4.2. Specifically, an MPPI optimizes the parameters of a multivariate Gaussian $\mathcal{N}(\mu, \Sigma)$ according to (2.109), using the objective function (2.104). The learned world model is used to generate the sample trajectories $\{\tau_{t:t+H}\}_{1:K}$ conditioned on sampled actions from a learned policy prior, and to compute the cost (2.104) with the reward model for timesteps $[t, t+H)$ and by estimating the cost-to-go from timestep $t+H$ onwards, for each trajectory.

Since planning in TD-MPC2 is done in the latent space, its world model does not reconstruct observations as in [HPBL], but rather predicts next-step latent states. The model's architecture is the following (task embeddings are ignored and some notations differ from the original paper for consistency reasons):

- $z = \phi_\theta(o)$ current observation encoder
- $\hat{z} = \mathcal{T}_\theta(z, a)$ next latent state predictor given current latent state, action
- $\hat{r} = R_\theta(z, a)$ reward categorical distribution parameters predictor for taking given action under given latent state
- $\hat{q} = Q_\theta(z, a)$ latent state-action value categorical distribution parameters predictor
- $\hat{\mu}_\pi, \hat{\sigma}_\pi = \pi_\xi(z_t)$ policy prior normal distribution parameters

where θ are the parameters of the ϕ, $\mathcal{T}$, R and Q networks and ξ are the policy prior network parameters.

The world model in TD-MPC2 is comprised of jointly-trained MLPs, including the state-transition model. The encoder maps each observation at each incoming timestep to a latent state. Furthermore, the state-transition function g outputs deterministic states, rather than parameterizing a probability distribution over latent states. Finally, initial empirical results for the algorithm (as per [HSW]) are derived from evaluating the algorithm on MDPs only. As a consequence, it can be hypothesized that TD-MPC2 is not sufficient for solving POMDPs.

A feasible approach, as previously discussed in 2.4.3, is to use an encoder function $\phi : \mathcal{H} \to \mathcal{Z}$ that "summarizes" the information contained in the history of observations and actions. Such encoders are called *approximate information state generators* if they satisfy (AP1) and (AP2), or (AP1), (AP2a) and (AP2b) defined in [SSSM22], as covered in 2.4.3. The properties relate to one-step reward prediction (AP1) and self-prediction of future latent states (AP2), or the weaker property of observation prediction with recurrence (AP2a) and (AP2b). Equivalent properties are *self-predictive abstraction (RP + ZP / EZP* and the *observation-predictive abstraction* properties (RP + Rec + OP)

property sets from [NES+]. Throughout this chapter, the unified view on state and history representations presented in [NES+] is used, since characteristics TD-MPC2, the variant presented in this thesis and other related works can be contrasted under this view.

In the context of MDPs, the encoder ϕ used in TD-MPC2 represents an AIS trained to satisfy (AP1) and (AP2) for Markov states (or RP + EZP). Since the history information is fully contained in the current observation, there exists a surjective mapping from encoders defined over the space of histories in a given MDP $\phi_\theta : \mathcal{H} \to \mathcal{Z}$ to encoders over the space of observations $\mathcal{O}$. Furthermore, the algorithm trains a state-action value estimator $Q_\theta : \mathcal{O} \times \mathcal{A} \to \mathbb{R}$ for policy optimization and cost-to-go estimation during rollouts via $TD(0)$ learning with bootstrapped targets 2.4.1, the learning goal being (2.128). Overall, from the point of view presented in [NES+], TD-MPC2 is a deterministic *self-predictive state abstraction* with return prediction RP + EZP + (2.128).

Proposition 1 from [NES+] gives a surrogate objective function as an upper bound for EZP in deterministic environments. Given a parameterized state/history encoder $\phi_\theta : \mathcal{O} \to \mathcal{Z}$ (where $\mathcal{O} = \mathcal{H}$ in MDPs as previously stated) and a deterministic parameterized next-state predictor $g_\xi : \mathcal{Z} \times \mathcal{A} \to \mathcal{Z}$, the surrogate loss is, according to [NES+], continuous regression with l_2 norm:

$$\mathcal{L}(\theta, \xi) = \mathbb{E}_{o' \sim P(\cdot|o,a)} \left[||g_\xi(\phi_\theta(o), a) - \phi_{\bar{\theta}}(o')||_2^2 \right] \tag{4.7}$$

where $\bar{\theta}$ indicates *detached* parameters. Proposition 3 from [NES+] shows that targets in the surrogate loss associated with EZP (meaning, encoded next-step observation with detached parameters) ensure stationary points for estimating the expectation, which is under the next-state distribution. This objective is used in TD-MPC2 for learning the transition model, with the difference being that the parameters of the encoder and the next-state predictor are shared under the *world model* parameters θ.

The surrogate loss associated with learning the latent states transition model is:

$$\mathcal{L}_{\text{cons}}(\theta) = \mathbb{E}_{\{\{o_t, a_t, r_t, o_{t+1}\}_{t=1}^H\}_1^B} \left[\sum_{t=0}^{H} \rho^t \left(||\bar{z}_t - \phi_{\bar{\theta}}(o_t)||_2^2 \right) \right] \tag{4.8}$$

The reward predictor in TD-MPC2 parameterizes a categorical distribution with K_R bins, as shown in (2.34). In order to target RP, TD-MPC2 uses a soft cross-entropy surrogate loss function for discrete regression by turning the real/target rewards into a soft label distribution using a Gaussian centered at the actual scalar reward. The objective is then (2.16) with *soft targets*. The associated surrogate loss function is:

$$\mathcal{L}_{\text{rew}}(\theta) = \mathbb{E}_{\{\{o_t, a_t, r_t, o_{t+1}\}_{t=1}^H\}_1^B} \left[\sum_{t=0}^{H} \rho^t \left(D_{CE}(R_\theta(\bar{z}_t, r_t)) \right) \right] \tag{4.9}$$

The state-action value predictor Q_θ is trained on the same sampled minibatch of sequences via one-step $TD(0)$ learning, as seen in Section 2.4.1. The targets $R_t = \bar{Q}_\theta(z_t, a_t)$ are

computed using the rewards sampled from the experience dataset for the current timestep r_t (the batch dimension is implicitly assumed here) plus the state-action value from the next step onwards $\bar{Q}_{\bar{\theta}}(\bar{z}_{t+1}, \pi(z_{t+1}))$, using encoded ground-truth observations from the next step $z_{t+1} = \phi_{\bar{\theta}}(o_{t+1})$ and actions sampled from a learned policy prior. $\bar{Q}_\theta$ is an EMA of Q_θ, with parameters updated using the *Polyak averaging* rule (2.91). Similar to the reward model, Q parameterizes a categorical distribution over discretized values, and the surrogate objective associated with learning Q is (2.16) with *soft targets*, as before. The associated loss function is:

$$\mathcal{L}_{\mathrm{Q}}(\theta) = \mathbb{E}_{\{\{o_t,a_t,r_t,o_{t+1}\}_{t=1}^{H}\}_1^B}\left[\sum_{t=0}^{H}\rho^t\left(D_{CE}(Q_\theta(\bar{z}_t,a_t), r_t + \gamma\bar{Q_\theta}(\bar{\phi_\theta}(o_{t+1}), \pi_\xi(\phi_{\bar{\theta}}(o_t))))\right)\right] \tag{4.10}$$

To estimate expected values for rewards and next-state predictions, the training loop in TD-MPC2 uniformly samples minibatches of short-length trajectories from an experience buffer, with starting observations $\{o_0 \bigcup (o_t, a_t, r_t, o_{t+1})_{t=1}^{H}\}_1^B \sim \mathcal{B}$. Latent trajectories, conditioned on the ground-truth actions, of the form $\tau = \hat{z}_{t-1}, a_t, \hat{z}_t, \ldots, a_{t+H}, \hat{z}_{t+H}$, are generated using the transition model $\hat{z_{t+1}} = g_\theta(\hat{z}_t, a_t)$ ($\hat{z}$ denotes model outputs), starting from the first encoded ground-truth observation $\phi_\theta(o_0)$. Supervised learning with stochastic gradient descent is then applied (as shown in Section 2.2.2), with the surrogate loss associated with estimating the gradient being composed of the objectives mentioned above. The reward and value outputs are conditioned on the predicted latent states sequences. Thus, the loss function for training the world model is the expected value of a weighted sum of the loss terms above, under the sampled minibatches (α_{cons}, α_{rew}, α_{Q} weigh each cost term):

$$\mathcal{L}(\theta) = \mathbb{E}_{\{\{o_t,a_t,r_t,o_{t+1}\}_{t=1}^{H}\}_1^B}\left[\alpha_{\mathrm{cons}}\cdot\mathcal{L}_{\mathrm{cons}}(\theta) + \alpha_{\mathrm{rew}}\cdot\mathcal{L}_{\mathrm{rew}}(\theta) + \alpha_{\mathrm{Q}}\cdot\mathcal{L}_{\mathrm{Q}}(\theta)\right] \tag{4.11}$$

The policy prior does not share parameters with the rest of the world model. Its parameters ξ are trained using a *stochastic maximum entropy* objective [HSW] [ZMB+08], which adds a weighed entropy loss term to the policy gradient ascent objective (2.93). This guides the optimization process for the policy parameters towards exploration (thus implicitly dealing with the exploration/exploitation dilemma mentioned throughout Section 2.4.1 without introducing stochasticity during planning as in ϵ-*greedy*, which is non-differentiable as mentioned in Section 2.4.1. The surrogate loss for optimizing ξ is:

$$\mathcal{L}(\xi) = \mathbb{E}_{\{\{o_t,a_t\}_{t=1}^{H}\}_1^B}\left[\sum_{t=0}^{H}\rho^t\left(\alpha Q_\theta(z_t, \pi_\xi(z_t)) - \beta\mathcal{H}(\pi_x i(\cdot|z_t))\right)\right] \tag{4.12}$$

Here, as well as in (4.11), $\pi_\xi(z_t)$ is shorthand for $\hat{a}_t \sim \pi_\xi(\cdot|z_t)$. $\mathcal{H}(\pi_x i(\cdot|z_t))$ represents the entropy of the policy distribution, mathematically defined as (2.8).

θ and ξ are learned using the *ADAM* optimizer [KB17], described briefly in Section 2.2.2.

4.3 LNN-TD-MPC2

Several methods for dealing with POMDPs have been covered in Section 2.4.2, and all approaches try to reduce POMDP to an glsMDP that satisfies the Markovian independence property. Here, the AIS approach presented in [SSSM22] is chosen, due to three main reasons:

1. Using history encoders to generate statistically sufficient latent states preserves the overall TD-MPC2 algorithm (only the state representation is changed), avoiding the complexity of variational-inference-based MBRL algorithms such as Dreamer [HPBL].
2. LNNs can be efficient history encoders for an AIS (as shown in this work), enabling the derivation of analytical properties beyond empirical results.
3. The unified view of state representations defined in [NES$^+$] provides a common analysis framework for both TD-MPC2 and this work.

Recall that TD-MPC2 uses state representations that satisfy RP and EZP in deterministic, fully observable MDPs. Many continuous control tasks, however, are neither Markovian nor deterministic. To handle such tasks, a variant of the TD-MPC2 world model can be build, using the theoretical foundations of [SSSM22], [NES$^+$] and [LXL$^+$] with LNNs such as LTC [HLA$^+$21] and CfC [HLA$^+$22] to encode efficient state representations in partially observable environments.

The state representations in TD-MPC2 are replaced with representations provided by an LNN that encodes histories of states $H_t = o_0, a_0, \ldots, a_{t-1}, o_t$. Let $\phi : \mathcal{H} \to \mathcal{Z}$ denote such an encoder. There are two approaches to using an LNN as a history encoder that satisfies the properties of an AIS:

1. The *self-predictive abstraction* approach learns a latent state dynamics model $g_\phi : \mathcal{Z} \times \mathcal{A} \to \mathcal{Z}$ that satisfies RP and (4.7) as an upper bound for ZP, as in TD-MPC2. Ground-truth latent states are encoded using the LNN ϕ as a history encoder during learning. The latent state dynamics model g_ϕ is then used during planning to generate rollouts.
2. The *observation-predictive abstraction* approach parameterizes a one-step observation probability distribution to satisfy OP together with RP. The recurrent update condition Rec is implicitly satisfied by the unfolding nature of the LNN state, as an instance of an RNN. During planning, forward histories are recursively constructed by appending observations sampled from the next-observation distribution model and actions sampled from the policy prior distribution.

The choice of state representations depends on the task. Section 5 of [NES$^+$] provides extensive empirical results that highlight the strengths of each compared to the other

in different environments and tasks. In POMDPs with perturbed observations, the *self-predictive abstraction* yields better results, confirming the authors' hypothesis that modeling observation distributions for noisy observations is a harder learning objective than learning self-predictive state representations. In environments with sparse rewards, however, the approach is weaker, since learning expressive latent state dynamics without using explicit observations relies on a strong reward signal. In such cases, the *observation-predictive abstraction* is more appropriate. These hypotheses also hold for the world models presented here, as will be shown in Chapter 5.

The history encoder ϕ_θ is now represented by an LNN that captures the underlying continuous-time state of the system, given the past observations and associated timesteps, as well as the past actions. Given the LTC's formulation as a nonlinear first-order dynamical system with adaptive time-constant, and the CfC's distillation of liquid time-constant models to a closed-form, such an encoder is (hypothetically) capable of capturing both the system dynamics and the system response to inputs. Lastly, as instances of RNNs, they explicitly build up memory through their internal state feedback mechanism. As a consequence, three hypotheses regarding LNN-based AIS ϕ can be formulated:

1. ϕ is a *self-predictive abstraction* or an *observation-predictive abstraction* in POMDPs.
2. ϕ the same holds for SDMDP with randomly-delayed observations of Section 2.4.3.
3. ϕ continuous-time RL tasks of Section 2.4.4 can be solved with LNN-TD-MPC2.

4.3.1 Self-Predictive Abstraction Model

The first approach is to target the EZP and the RP conditions for the LNN-based history encoder. The consistency surrogate loss used in TD-MPC2 (4.8) already targets an upper bound for EZP. The main difference is that the latent states z_t used in the loss function are provided by the LNN encoder.

4.3.2 Observation-Predictive Abstraction Model

Let $\phi_\theta : \mathcal{H}' \to \mathcal{Z}$ be an encoder represented by an LNN, defined over the space of histories. The history space is extended with the space of real numbers $\mathcal{H}' \coloneqq \mathcal{O}^n \times \mathcal{A}^{n-1} \times \mathbb{R}^n$, and the piece-wise time delta Δt between observation timesteps are added to the history. φ in Rec is represented by the internal state update rule of the LNN (2.57) (2.64). Furthermore, the latent state transition model in TD-MPC2 is replaced with a model that parameterizes a distribution over the observation space conditioned on latent states and actions: $g_\theta : \mathcal{Z} \times \mathcal{A} \to \Delta\mathcal{O}$. Intuitively, this transition model expresses the probabilities of observing certain information, after taking a certain action given a certain latent state. The reward and value models, as well as the policy prior, are kept unchanged.

In order to target OP, a forward KL-divergence surrogate loss (2.14) is used to match the underlying observation distribution of the POMDP. For ease of readability, let $Q_\theta(o)$ denote the posterior distribution parameterized by g_θ conditioned on latent states and

actions, and $P(o)$ be the underlying observation distribution. Given the observation samples from minibatches $o_{1:H}$ (batch dimension is implicitly assumed), which have been observed by taking the same actions that are used as inputs to g in the supervised learning loop, minimizing the KL-divergence can be reformulated as minimizing the negative log likelihood of the observed samples under the learned distribution (2.46). The proof of the equivalence is the following:

$$\begin{aligned} D_{KL}(P(o)||Q_\theta(o)) &= \mathbb{E}_{o\sim P}\left[\log \frac{P(x)}{Q_\theta(x)}\right] \\ &= \mathbb{E}_{o\sim P}\left[-\log Q_\theta(x)\right] + \mathbb{E}_{o\sim P}\left[\log P(x)\right] \end{aligned} \tag{4.13}$$

Since the last term does not depend on the learned parameters θ, it can be assumed as a constant during optimization. With this simplification, the above equation can be used as a surrogate loss term for learning the next-observation distribution model g_θ.

The learned next-step observation distribution is represented as multivariate diagonal normal distribution (described in Section 2.1.3, with the following parameters:

1. mean μ_N, where N is the observation dimension $|\mathcal{O}|$, and B is the batch dimension
2. covariance as a diagonal matrix $diag(\Sigma_{N\times N})$

The choice is supported by the *central limit theorem* [CB24], described in Section 2.1.3. Given enough i.i.d samples, the mean of the samples converges in distribution to a multivariate normal distribution.

To achieve this, the output dimension of the transition model $\mathcal{T}_\theta$ is $2|\mathcal{O}|$. Each output of the model is split in two even chunks, and the first is interpreted as the mean of the distribution by applying a hyperbolic tangent function (2.41) scaled by the observation space bounds. The standard deviation results from applying the differentiable softplus activation function (2.38), clipped by hyperparameters $\min_\Sigma$ and $\max_\Sigma$.

For deterministic environments, however, the requirement can be relaxed. Since deterministic environments imply a deterministic observation transition model, for any two given histories h_t and h'_t such that $\phi(h_t) = \phi(h'_t)$, the transition model must return the same observation:

$$\tau(\phi(h_t), a_t) = \tau(\phi(h'_t), a_t), \forall h_t, h'_t \in \mathcal{H} \text{ s.t. } \phi(h_t) = \phi(h'_t) \tag{4.14}$$

Seeing as when targeting ZP in deterministic environments, a surrogate loss that enables continuous regression over exact predictions is sufficient, the same surrogate loss (4.8) is applied for observation-predictive abstractions when environments are deterministic.

Supervised learning for RNNs using SGD or similar stochastic gradient methods is challenging. Due to the sequential nature of the models and the BPTT training algorithms

applied over contiguous sequences of input-output pairs, the i.i.d. property is typically not satisfied. Since the training algorithm relies on sampled sequences from the experience dataset, BPTT cannot be used to train the LNN. Instead, T-BPTT is used, and the choice of the initial hidden state for each sequence greatly influences the optimization processes. [HS15] studied the effects of using zero-valued initial hidden states during training, as opposed to storing the hidden state in the dataset. Zero-value initial hidden states preserve the i.i.d. property of samples, while using stored hidden states inevitably introduces correlation between samples and can destabilize the learning process due to the distribution shift caused by the RNN parameters used during environment interaction, which are outdated once a learning iteration occurs [AKLM21]. Learning with zero-valued initial hidden states, however, does not exploit the temporal information contained in the history of the sampled sequences, which is required for learning an AIS.

[KOD+19] introduces an approach called *burn-in*, where constant-length slices from the beginning of the sampled sequences are used to initialize the hidden state for training. Gradients are not allowed to flow through unfolding over the burn-in sequences. The work also introduces a hybrid approach, where the burn-in of the RNN internal state starts from stored hidden states. While this approach reintroduces the correlation and distribution shift issues, the latter is mitigated by burning in the state of the network using the same parameters as the ones used during the training iteration. All the approaches are empirically compared using metrics such as recurrent state staleness and sample efficiency.

Using the empirical results of [KOD+19] as a starting point, the hidden states of the LNN computed during environment interaction are stored in the experience dataset. During training, minibatches of sequences are sampled from the dataset $\{a_o \bigcup (o_t, a_t, \Delta t_t, r_t, o_{t+1})_{t=1}^{H}\}_1^B \sim \mathcal{D}$, together with an initial hidden state $z_1 = \phi(h_1)$. Constant-length sub-sequences are sliced from the beginning of the sampled sequences $\{(o_t, a_t, \Delta t_t, r_t, o_{t+1})_{t=1}^{T}\}_1^B, T < H$. The observations and actions, together with the starting hidden state, are used as sequence inputs to the encoder with detached parameters:

$$z_T = \bar{\phi_\theta}(h_T) = z_1 \bigcup \bar{\phi_\theta}(o_{1:T}, a_{0:T}, \Delta t_{1:T}) \tag{4.15}$$

$\bar{\theta}$ indicates that the parameters are detached from the computation graph.

Starting from z_T, the sampled sequences $o_{T+1:H}, a_{T:H}, \Delta t_{T+1:H}$ are fed through the LNN encoder. The next-step observation model g_θ outputs the parameters of the observation distribution Q given $z_t, a_t, t \in [T : H]$. The negative log likelihood surrogate objective (4.13) is then used to achieve OP. Following variational inference-based algorithms such as *Dreamer* [HPBL], *free bits* are employed to avoid the forward KL-divergence's collapse towards 0. This is achieved by clipping the surrogate loss below a certain threshold.

As a concrete example, let g_θ parameterize a multivariate Gaussian distribution over the observation space $\Delta(\mathcal{O}) = \mathcal{N}(\mu_\theta, \Sigma_\theta) := Q(O)$. The final surrogate loss for training the

next-step observation distribution model is:

$$\mathcal{L}_{\text{pred}}(\theta) = \mathbb{E}_{\{\{o_t,a_t,\Delta t_t,r_t,o_{t+1}\}_{t=T}^{H}\}_1^B}\left[\sum_{t=T}^{H}\rho^t\left(\max(1,\log Q(\bar{\phi_\theta}(o_t))\right)\right] \tag{4.16}$$

The targets for the update associated with the value function are also changed to $TD(\lambda)$ targets, computed using recursive bootstrapped λ-returns over the sampled sequences. Eligibility traces are not required (unlike the online setting in $TD(\lambda)$ method described in Section 2.4.1) for credit assignment, since complete sequences are sampled. Let $\bar{Q_\theta^\lambda}(z_t, a_t)$ denote the λ targets, where a_t are actions sampled from the policy prior and z_t is the history at timestep t. Given a batch of sequences as before, targets are recursively computed starting from the timestep H backwards using the following rule:

$$R_t^\lambda = r_t + \gamma\left((1-\lambda)Q_{\bar{\theta}}(z_{t+1},a_{t+1}) + \lambda R_{t+1}^\lambda\right), t \in [T,H] \tag{4.17}$$

The surrogate loss associated with learning the value function remains identical to TD-MPC2, but the targets are changed to λ targets computed as above:

$$\mathcal{L}_{\text{Q}}(\theta) = \mathbb{E}_{\{\{o_t,a_t,\Delta t_t,r_t,o_{t+1}\}_{t=1}^{H}\}_1^B}\left[\sum_{t=0}^{H}\rho^t\left(D_{CE}(Q_\theta(\bar{z}_t,a_t), Q_{\bar{\theta}}^\lambda(\bar{z}_t,\pi_{\bar{\xi}}(\bar{z}_t))\right)\right] \tag{4.18}$$

where $\bar{z}_t = \phi_\theta(h_t)$.

The final loss function for the observation-predictive LNN-TD-MPC2 is:

$$\mathcal{L}(\theta) = \mathbb{E}_{\{\{o_t,a_t,\Delta t_t,r_t,o_{t+1}\}_{t=1}^{H}\}_1^B}\left[\alpha_{\text{pred}}\cdot\mathcal{L}_{\text{pred}}(\theta) + \alpha_{\text{Q}}\cdot\mathcal{L}_{\text{Q}}(\theta) + \alpha_{\text{rew}}\cdot\mathcal{L}_{\text{rew}}(\theta)\right] \tag{4.19}$$

Proposition 4 of [SSSM22], and the equivalent Proposition 7 of [NES+], show that observation-predictive abstractions imply self-predictive abstractions. However, matching the observation distribution (OP) is a stronger condition to satisfy, especially given that sampled minibatches are not i.i.d. when training RNNs. Depending on the task, targeting OP might be infeasible.

4.3.3 Training Procedure and Inference Function

The training procedure is similar to the default TD-MPC2 procedure: adaptive optimization of the world model and policy prior parameters using the Adam optimizer [KB17], given surrogate losses computed on minibatches of trajectories uniformly sampled from an experience dataset. The world model parameters θ, as well as the policy prior parameters ξ are updated once per training procedure call. The changes stem from the surrogate loss functions used for training the proposed models, the fixed sequence slices used for burning in the encoder hidden state and the λ-targets used for training the value function. The update procedure for the observation-predictive architecture is described in Algorithm 1, whereas the update for the self-predictive approach is described in Algorithm 2.

The inference step using the planner is identical to the TD-MPC2, and the planning procedure is described in Algorithm 3. The changes stem only from how the world model executes a forward step using the transition model $\mathcal{T}_\theta$, given an action. The forward step for the observation-predictive architecture is described in Algorithm 5, and for the self-predictive approach in Algorithm 4. Figure 4.5 illustrates the forwards step for both models.

Algorithm 1 Observation-predictive LNN-TD-MPC2 update iteration

Require: LNN encoder ϕ_θ, next-step observation model f_θ, reward model R_θ, value model Q_θ, policy prior π_ξ, dataset $\mathcal{D}$, batch size B, burn-in index T, sequence end H, learning rate η, loss weights $\lambda_{\text{pred}}, \lambda_{\text{rew}}, \lambda_{\text{Q}}$, discount γ, TD factor λ, policy weights α^π, β^π

Sample minibatch $\{z_1, (o, a, r, \Delta t)_{1:H}\}^B \sim \mathcal{D}$ ▷ Sample sequences
$z_T \leftarrow \phi_{\bar{\theta}}(o_{1:T}, a_{1:T-1}, \Delta t_{1:T}, z_1)$ ▷ Burn-in with frozen encoder
$z_{T+1:H} \leftarrow \phi_\theta(o_{T+1:H}, a_{T:H}, \Delta t_{T:H}, z_T)$ ▷ Encode full sequence
$\hat{o}_{T+1:H} \leftarrow f_\theta(z_{T:H-1}, a_{T:H-1})$ ▷ Predicted observations
$\hat{\mu}^f_{T+1:H}, \hat{\Sigma}^f_{T+1:H} \leftarrow f_\theta(z_{T:H-1}, a_{T:H-1})$ ▷ Predict observation distribution
$\hat{r}_{T+1:H} \sim R_\theta(z_{T:H-1}, a_{T:H-1})$ ▷ Reward prediction
$\hat{Q}_{T+1:H} \sim Q_\theta(z_{T:H-1}, a_{T:H-1})$ ▷ Value prediction
$\mathcal{L}_{\text{pred}}(\theta) \leftarrow \|\hat{o}_{T+1:H} - o_{T+1:H}\|_2^2$ ▷ Observation loss (deterministic)
$\mathcal{L}_{\text{pred}}(\theta) \leftarrow -\log p(o_{T+1:H} \mid \mathcal{N}(\hat{\mu}^f_{T+1:H}, \hat{\Sigma}^f_{T+1:H}))$ ▷ Observation loss (stochastic)
$\mathcal{L}_{\text{rew}}(\theta) \leftarrow D_{\text{CE}}(\hat{r}_{T+1:H}, r_{T+1:H})$ ▷ Reward loss
$\mathcal{L}_{\text{Q}}(\theta) \leftarrow D_{\text{CE}}(\hat{Q}_{T+1:H}, R^\lambda_{T+1:H})$ ▷ Value loss
$\mathcal{L}(\theta) \leftarrow \lambda_{\text{pred}}\mathcal{L}_{\text{pred}}(\theta) + \lambda_{\text{rew}}\mathcal{L}_{\text{rew}}(\theta) + \lambda_{\text{Q}}\mathcal{L}_{\text{Q}}(\theta)$ ▷ Total model loss
$\theta \leftarrow \text{ADAM}(\nabla_\theta \mathcal{L}(\theta), \alpha)$ ▷ Update world model
$\mu^\pi_{T:H-1}, \sigma^\pi_{T:H-1} \leftarrow \pi_\xi(\text{sg}(z_{T:H-1}))$ ▷ Policy prior dist
$\hat{a}_{T:H-1} \sim \mathcal{N}(\mu^\pi_{T:H-1}, \sigma^\pi_{T:H-1})$ ▷ Sample actions
$\mathcal{L}(\xi) \leftarrow \alpha^\pi Q_{\bar{\theta}}(\text{sg}(z_{T:H-1}), \hat{a}_{T:H-1}) - \beta^\pi \mathcal{H}(\mathcal{N}(\mu^\pi_{T:H-1}, \sigma^\pi_{T:H-1}))$ ▷ Policy loss
$\xi \leftarrow \text{ADAM}(\nabla_\xi \mathcal{L}(\xi), \alpha)$ ▷ Update policy prior

Algorithm 2 Self-predictive LNN-TD-MPC2 update iteration

Require: LNN encoder ϕ_θ, dynamics model f_θ, reward model R_θ, value model Q_θ, policy prior π_ξ, dataset $\mathcal{D}$, batch size B, burn-in index T, sequence end H, learning rate η, loss weights $\lambda_{\text{pred}}, \lambda_{\text{rew}}, \lambda_{\text{Q}}$, discount γ, TD factor λ, policy weights α^π, β^π

$\hat{z}_{T+1:H} \leftarrow f_\theta(z_{T:H-1}, a_{T:H-1})$ ▷ Predicted next-states
$\hat{\mu}^f_{T+1:H}, \hat{\Sigma}^f_{T+1:H} \leftarrow f_\theta(z_{T:H-1}, a_{T:H-1})$ ▷ Predict next-state distributions
$\mathcal{L}_{\text{cons}}(\theta) \leftarrow \|\hat{z}_{T+1:H} - z_{T+1:H}\|_2^2$ ▷ Latent state consistency loss (deterministic)
$\mathcal{L}_{\text{cons}}(\theta) \leftarrow -\log p(z_{T+1:H} \mid \mathcal{N}(\hat{\mu}^f_{T+1:H}, \hat{\Sigma}^f_{T+1:H}))$ ▷ Latent state cons. loss (stoch.)

Algorithm 3 Planning/inference step for TD-MPC2/LNN-TD-MPC2

Require: Starting latent state z_t, forward dynamics model $\mathcal{T}_\theta$, reward model R_θ, value model Q_θ, policy prior π_ξ, trajectory rollout length H_{plan}, number of sampled trajectories K, number of MPPI samples S, Number of MPPI iterations I, previous MPPI solution mean μ^S_{t-1}

$z_t \leftarrow \phi_\theta(o_t, a_{t-1}, z_{t-1})$ ▷ Encode current observation-action pair
$z_t^K \leftarrow [z_t]_K^\top$ ▷ Initialize K trajectories
for $i = t$ to $t + H_\Pi - 1$ **do**
  $a_i^K \sim \pi_\xi(\cdot|z_i)$ ▷ Sample actions from policy prior
  $z_{i+1} \leftarrow \mathcal{T}_\theta(z_i, a_i)$ ▷ Forward dynamics step
end for
$a_{t+H_\Pi} \leftarrow \pi_\xi(\cdot|z_{t+H_\Pi})$
$A^{1:K} \leftarrow a^{1:K}$ ▷ Bootstrap MPPI with policy prior actions
$z_t^S \leftarrow [z_t]_S^\top$ ▷ Repeat start state for S samples
$\mu^S \leftarrow \mu^S_{t-1}$, $\Sigma^S \leftarrow 0$ ▷ Initialize MPPI distribution
for $i = 1$ to I **do**
  $A^{K+1:S} \sim \mathcal{N}(\mu^S, \Sigma^S)$ ▷ Sample actions from MPPI distributions
  $G^{1:S} \leftarrow [0]_S$ ▷ Initialize returns
  for $j = t$ to $t + H_\Pi - 1$ **do**
    $r_j^S \sim R_\theta(z_j^S, A_j^S)$ ▷ Compute reward
    $G^S \leftarrow G^S + r_j^S$ ▷ Add to return
    $z_{j+1}^S \leftarrow \mathcal{T}_\theta(z_j^S, A_j^S)$ ▷ Forward dynamics
  end for
  $G^S \leftarrow G^S + Q_\theta(z^S_{t+H_\Pi}, A^S_{t+H_\Pi})$ ▷ Add estimated return-to-go
  $w^s \leftarrow (2.108)(G^S)$ ▷ Weigh samples according to G^S
  $\mu^S, \Sigma^S \leftarrow (2.109)(w^S)$ ▷ Update MPPI distribution
end for
return $\text{argmax}_{a \in A^S} G^s$ ▷ Return action that leads to the best trajectory

Algorithm 4 Transition model $\mathcal{T}_\theta$ for self-predictive LNN-TD-MPC2

Require: Current information state z_t, current action a_t, LNN encoder ϕ_θ, dynamics model f_θ

return $z_{t+1} \sim f_\theta(\cdot|z_t, a_t)$ ▷ Sample next-step information state

Algorithm 5 Transition model $\mathcal{T}_\theta$ for observation-predictive LNN-TD-MPC2

Require: Current information state z_t, current action a_t, LNN encoder ϕ_θ, observation predictor model f_θ

$\hat{o}_{t+1} \sim f_\theta(\cdot|z_t, a_t)$ ▷ Sample next-step observation prediction
return $z_{t+1} \leftarrow \phi_\theta(\hat{o}_{t+1}, a_t, z_t)$ ▷ Encode predicted observation and input action

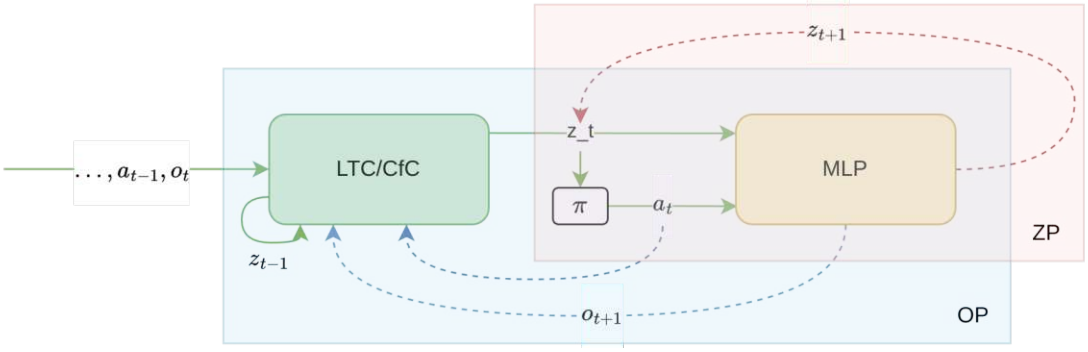


Figure 4.5: Forward step for LNN-TD-MPC2. Blue indicates the observation-predictive model, while red indicates the self-predictive model. Green represents a common data path, while dashed lines denote one timestep forward.

4.3.4 Theoretical Bounds

The world model upper bound analysis referenced in Section 2.4.2 is extended to world models with AIS encoder in [AHL16] and [FLRP+19]. Let ϵ_θ^P represent the upper bound on the prediction error, which can be formulated as turning the history compression conditions of ZP and OP into l_1-norms bounded by a tolerance ϵ_θ^P:

$$\max_{H \in \mathcal{H}} ||\mathcal{T}(\cdot|h,a) - \mathcal{T}(\cdot|\phi(h),a)||_1 \leq \epsilon_\phi^{\mathcal{T}} \tag{4.20}$$

where the probability distributions are over the latent space $\mathcal{Z}$ when ϕ_θ is a *self-predictive* abstraction parameterized by θ, or the observation space $\mathcal{O}$, when ϕ_θ is an *observation-predictive* abstraction parameterized by θ. In the context of learning θ from a dataset $\mathcal{D}$ using the LNN-TD-MPC2 training procedure of Algorithm 1, ϵ_ϕ^P is an upper bound for the expected value of the prediction error under sampled trajectories from $\mathcal{D}$. In this case, ϕ_θ is also called an ϵ-*sufficient* history mapping, denoted as ϕ_ϵ. Let ϕ_* denote the sufficient mapping that satisfies OP and ZP exactly.

Proposition 2 from [FLRP+19] states the following: for any two given histories such that $\phi_\theta(h_1) = \phi_\theta(h_2)$, the maximum difference in state-action value evaluation over the action space, and given the learned state-action value model, can be bounded by the predictive bound ϵ_ϕ^P:

$$\max_{a \in \mathcal{A}} |Q_\theta(\phi_*(h_1),a) - Q_\theta(\phi_*(h_2),a)| \leq \epsilon_\phi^{\mathcal{T}} \frac{R_{\max}}{1-\gamma} := \epsilon_\theta^Q \tag{4.21}$$

where $R_{\max} = \max \mathcal{R}$. The proposition is proven in [FLRP+19].

Let $\mathcal{D}$ denote in this context, the set of sampled sequences during training, and $\mathcal{D}_\infty$ represent the sampled dataset in the limit (which is equivalent to the entire experience dataset). Let $\pi_\xi^{\mathcal{D}}$ denote the policy learned from the limited dataset $\mathcal{D}$, and $\pi_\xi^{\mathcal{D}_\infty}$ represent the policy learned by sampling the entire experience dataset in the limit. Finally, let $V_\theta^{\pi_*}$ represent the theoretical optimal value function for the abstract MDP encoded by

ϕ_θ, under the optimal policy π_*, and $V_\theta^{\pi_\xi^{\mathcal{D}_\infty}}$ a possible value function, using the policy prior π_ξ learned from the asymptotical dataset $\mathcal{D}_\infty$. Lemma 1 by [AHL16] provides the following bound for the value estimate, in terms of the state-action value error ϵ_θ^Q from above:

$$\max_{h\in\mathcal{H}} |V_\theta^{\pi_\xi^{\mathcal{D}_\infty}}(\phi_\theta(h)) - V_\theta^{\pi_\xi^{\mathcal{D}_\infty}}(\phi_\theta(h))| \leq \frac{2\epsilon_\theta^Q}{(1-\gamma)^2} \tag{4.22}$$

Plugging ϵ_θ^Q from (4.21) [FLRP+19] in (4.22) [AHL16], Theorem 1 from [FLRP+19] gives an upper bound for the asymptotic bias of the theoretical value function associated with the learned AIS, under the learned policy π_ξ, in terms of the theoretical optimal policy:

$$\max_{h\in\mathcal{H}} |V_\theta^{\pi_\xi^{\mathcal{D}_\infty}}(\phi_\theta(h)) - V_\theta^{\pi_*}(\phi_\theta(h))| \leq \frac{2\epsilon_\phi^{\mathcal{T}} R_{\max}}{(1-\gamma)^3} \tag{4.23}$$

Theorem 3 from the same work [FLRP+19] also provides a bound for the overfitting of the learned model on the sampled experience. The overfitting of the model on the limited dataset $\mathcal{D}$ is bounded by:

$$\begin{aligned}\max_{H\in\mathcal{H}}(Q_\theta^{\pi_\xi^{\mathcal{D}_\infty}}(\phi_\theta(H), \pi_\xi(\cdot|\phi_\theta(H))) - Q_\theta^{\pi_\xi^{\mathcal{D}}}(\phi_\theta(H), \pi_\xi(\cdot|\phi_\theta(H))) \leq \\ \leq \frac{2R_{\max}}{(1-\gamma)^2}\sqrt{\frac{1}{2n}\ln\left(\frac{2|\phi(\mathcal{H})||\mathcal{A}|^{1+|\phi(\mathcal{H})|}}{\delta}\right)}\end{aligned} \tag{4.24}$$

with a probability $\geq 1-\delta$. The proof is in Appendix A.2. of [FLRP+19].

4.3.5 Nomenclature and Hyperparameter Choice

The base TD-MPC2 algorithm is designed to be hyperparameter-free, including batch size, MPPI population size, replay buffer size and learning rate. Some of the hyperparameters are automatically chosen using task-dependent heuristics. The choices and the motivation behind them are described in Appendix H from [HSW]. The practitioner is given a choice of hyperparameter configurations, which includes the dimensions and the number of hidden layers for the MLPs, as well as the number of Q-functions in the Q-ensemble. Here, an additional architecture size is defined, which has fewer parameters than the smallest TD-MPC2 configuration. The architecture configuration is denoted as *light*. The architecture proposed in this chapter replaces the default encoder with the LNN encoder, which only requires defining the number of hidden neurons and the size of the latent vectors. Furthermore, the default values of some TD-MPC2 hyperparameters are changed, to acommodate the proposed algorithm variants. The default length of the sampled trajectories for training $H = 3$ is extended to $H = 32$ to facilitate training the LNN with T-BPTT on longer sequences. The decaying term ρ used in the initial surrogate loss functions (4.8), (4.9), (4.10), used to decay the error terms further along

the sequences and focus on immediate accurate predictions, is now set to 1. The proposed *light configuration* is summarized in Table 4.1.

Hyperparameter	Fixed value
LNN hidden size	64
MLP hidden size	192
# Q-functions	2
Latent vector size	64

Table 4.1: *Light* architecture configuration.

LNN-TD-MPC2 defines extra hyperparameters. The planning horizon H_Π is decoupled from the world model training horizon H. Furthermore, one can choose between the LNN models to use: LTC or CfC. Finally, a choice between the *observation-predictive* and the *self-predictive* models, as well as deterministic or stochastic predictions, can be made by the user. As will be shown in Chapter 5, the choice of architecture depends on the task. $\min_\Sigma$ and $\max_\Sigma$ are hyperparameters representing the bounds for the learned covariance of the next-step observation multivariate normal distribution in the stochastic *observation-predictive* model.

CHAPTER 5

Evaluation

This chapter focuses on establishing empirical tests for the following *hypotheses*:

1. LNNs are effective and sample-efficient history and state dynamics encoders in MDPs and POMDPs. There is a high correlation between the internal state of the liquid time-constant models and the underlying state of the environment.
2. Training LNNs using LNN-TD-MPC2 is stable and converges to returns comparable to the ones achieved by TD-MPC2 for the given MDPs.
3. LNN-TD-MPC2 models solve and outperform standard TD-MPC2 in POMDPs and SDMDPs.
4. LNN-TD-MPC2 performance scales with planning horizon, whereas TD-MPC2 does not.
5. The stochastic observation-predictive world model perform better in POMDPs with perturbed observations or sparse rewards. The self-predictive model performs better in fully-observable MDPs with complex dynamics.
6. The history encoder in LNN-TD-MPC2 correctly processes irregularly sampled observations (see Section 2.4.3) and achieves better performance on task variants with stochastic observation delay, compared to standard TD-MPC2.

5.1 Methodology

To test the above hypotheses in an empirical manner, three different robotics systems are chosen, as mentioned in Section 4.1: Cartpole, Acrobot and Walker. Similarly, to test the LNN-TD-MPC2 models' capabilities of learning a model of the environment to control these systems towards solving specific tasks, even in partially observable scenarios, a

variety of scenarios are implemented. For validating the first hypothesis, fully observable and partially observable variants of *Cartpole* are chosen, since the dynamical system presents easily-derivable equations of motion, but controlling the system without a given model is still challenging. The small state space further enables analysis of the correlation hypothesis. For all other hypotheses, a suite of environments is chosen, and multiple trials with different seeds for the random number generators are run to derive performance statistics and rule out the hypothesis of the model overfitting to a specific randomness source. All LNN-TD-MPC2 world models presented in Section 4.3 are evaluated for all scenarios.

For each task, the following partially observable variants are defined:

1. $\text{perturbed}^{<\sigma>}$: observation noise sampled from a normal distribution with σ^2 variance
2. $\text{delay}^{<N>}$ observations are delayed by t steps, where t is sampled from a discrete uniform distribution $\mathcal{U}[0, N)$

Since *cartpole* is a simpler control problem to solve, the following additional partially observable variants are defined:

1. remove-vel: velocity states are not included in the observations
2. $\text{perturbed-flickering}^{<\sigma>}$: observation noise sampled from a normal distribution with σ^2 variance, as before; furthermore, a probability of *flickering* for each state (observations are zeroed out for a timestep) is sampled from a normal distribution with $\sigma = 0.2$ standard deviation

For tasks with observation noise, all models are trained on the task variant with $\sigma = 0.1$. This includes the $\text{cartpole-swingup}^{<\sigma>}_{\text{perturbed-flickering}}$ task, where the models are trained with observation noise sampled from a normal distribution with $\sigma = 0.1$ and flickering probability sampled from a normal distribution with $\sigma = 0.2$. The trained models are then evaluated on observation spaces with increasing noise levels $\sigma \in \{0.1, 0.2, 0.3\}$.

To facilitate benchmarking, environment rewards are normalized to be in the range of $[0, 1]$, and the infinite-horizon MDPs are truncated after 1000 steps, leading to episodic returns being defined in the range $[0, 1000]$.

5.2 Setup

The simulation setup relies on *dm-control* [TTM+], a standardized benchmarking Python library for continuous-control tasks. *dm-control* is an abstraction layer over environments simulated using *MuJoCo* (Multi-Joint dynamics with Contact) [TET12], an open-source C/C++ physics simulation engine. The structure of a Mujoco simulation is the following:

- MJCF - XML extension used to define models
- mjModel - static data structure defined using MJCF, which represents the hierarchical structure of a robot, in terms of links, joints, sensors, actuators and physical parameters.
- mjData - runtime instances of entities defined by a *mjModel*
- 3D interactive scene visualizer, rendered using OpenGL [SR07]
- Differentiation engine for forward and inverse dynamics [Tod14], with single-step Euler integrator (2.56), 4^{th} order Runge-Kutta integrator (2.55) and others

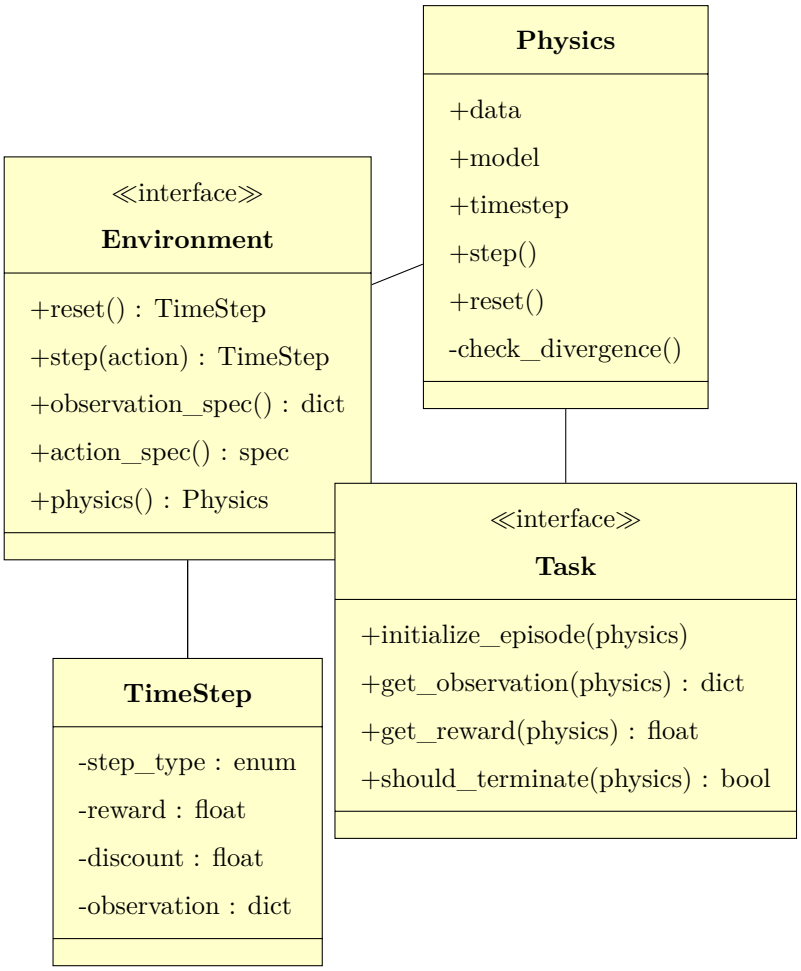


Figure 5.1: UML class diagram of the dm_control RL environment interface.

The *dm-control* offers a Python wrapper over the MuJoCo engine, which provides Python bindings for the C/C++ MuJoCo API. Environments in *dm-control* implement the

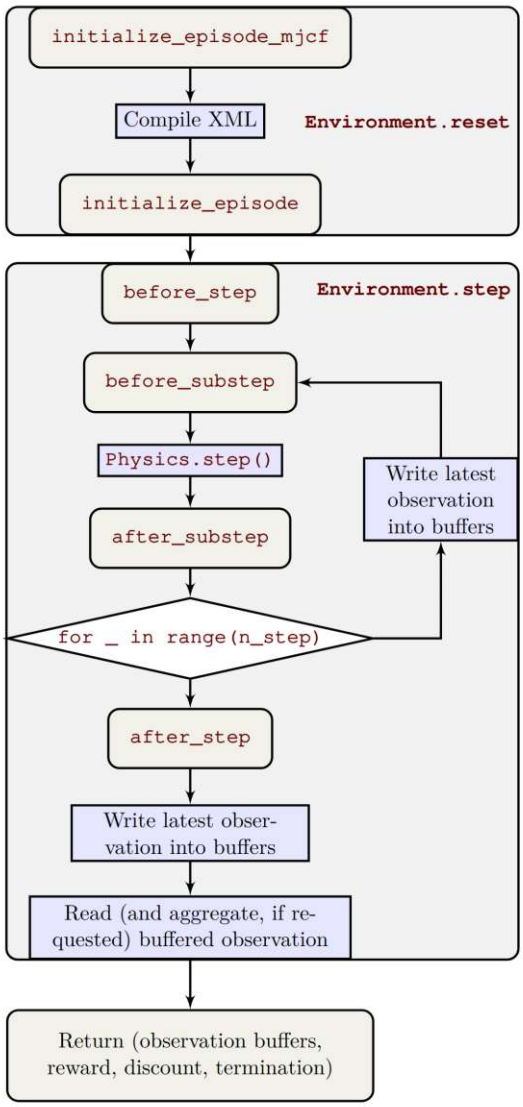


Figure 5.2: Execution flow of **reset** and **step** methods of the dm-control implementation of the dm-env *environment* interface. Taken from [TTM+].

dm_env interface, an API that was designed to standardize RL environment implementations [MDA+19]. Figure 5.1 represents a UML class diagram of the main *dm_env* interfaces and data structures. The description of the *dm-control* implementation of the interface as a MuJoCo wrapper is defined in section 2 of [TTM+]. Crucially, the separation between the *physics* layer (which interfaces with the Mujoco Python bindings) and the higher-level *environment* layer (exposed to the practitioner) is maintained, and the physics stepping (which forwards integrates the MuJoCo simulation) and the

environment stepping are decoupled. One environment step call can invoke multiple physics step calls, which allows the physics timestep to be kept small and the integration to be numerically stable. The flowchart in Figure 5.2 illustrates the execution flow of calls to the *reset* and *step* methods. The former initializes the system, the physics and environment-specific data structures, while the latter forwards the simulation by one execution step, given an action command. The library further offers modules for defining custom systems, environments and tasks, as well as extensions such as variations with corrupted observations or reward shaping.

The most relevant module of the *dm-control* library in the context of this chapter is the *control suite* module, which implements a benchmark suite that standardizes popular MuJoCo RL environments, such as those from *Gymnasium* [TKT+24] (including *cartpole*, *acrobot*, and *walker*), as well as extra variations that are more difficult to solve. Rewards are standardized to be in the range of $[0, 1]$, and sparse rewards are implemented using a smooth tolerance function around the discrete values of $\{0, 1\}$, which keeps the reward function differentiable. Infinite-horizon tasks are truncated after 1000 steps, leading to episodic returns of $R \in [0, 1000]$. This fits the environment definition proposed in the methodology section in Section 5.1. A solution is considered *optimal* by the authors of [TTM+] if $R \geq 800$.

To test *Hypothesis 6*, task variants of the type $\text{delay}^{<N>}$ are created by implementing a wrapper over the *dm-control* environment. This is facilitated by the *Gymnasium wrapper* class [Fou25], which allows extending *Gymnasium* environments, including the class's members and the *step* and *reset* methods. The observations emitted by the environment, together with their timestamps, are stored in a double-ended queue of capacity N. At each call of the *step* method, the environment returns the i-th observation-timestamp pair from the queue, with $i \sim \mathcal{U}[1, 10]$. This effectively implements SDMDPs with delayed observations (see Section 2.4.3). The described environment is illustrated in Figure 5.3.

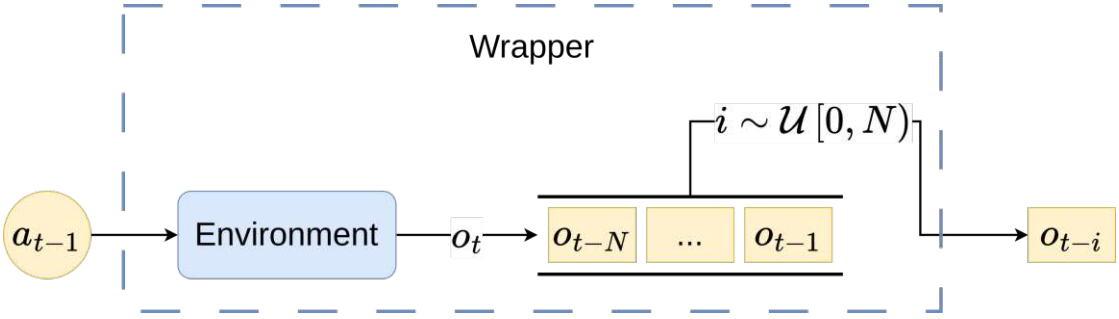


Figure 5.3: Environment wrapper where observation history is stored in a double-ended queue. Observations are uniformly sampled from the queue, implementing stochastic observation delay

The library also provides a benchmarking suite for standardized continuous-control agents, with normalized action intervals of $[-1, 1]$ and truncated episodes with fixed duration. The aim is to provide uniform durations and returns across environments and tasks. All

episodes are truncated after 1000 steps. Given that rewards are in the range $[0, 1]$, the theoretical maximum return achievable for any given environment is also 1000, which matches the proposed reward and return ranges in Section 5.1. The authors of [TTM+] define an optimality criterion for task solutions as achieving returns higher than 800. While the library provides both image and proprioceptive observations, only the latter are used in this evaluation. The control suite is documented in section 6 of [TTM+].

Evaluation scenarios are devised on the *cartpole*, *acrobot* and *walker*. In *dm-control*, a distinction is made between *domains* (which represent the physical model) and the *tasks* (complete instance of an MDP). For *cartpole*, the tasks associated with the formulation in Section 4.1 are the *swingup* and *swingup-sparse*, with the initial orientation of the pole being downwards. The latter task provides sparse rewards only: a reward of 1 per timestep is received only when the pole angle is approximately upwards. Both tasks will be included in the evaluation. For *walker*, the *run* task is used, which includes a weighed forward velocity term in the reward function.

The algorithm variants are defined in terms of the world model used, the model size and the planning horizon. The naming scheme used to distinguish the world models is $< \text{LNN-variant} >^{<\text{OP/ZP}>}_{<\text{det/stoch}>}$, where LNN-variant $\in \{\text{LTC}, \text{CfC}\}$.

Table 6.1 in the Appendix lists the environments and tasks used in the scenario, their state, observation and action spaces, the reward type (sparse or dense) and references to the corresponding reward function, for dense rewards.

All models evaluated in this chapter have been trained on a local workstation equipped with an NVIDIA RTX 5080 16 GB VRAM GPU.

5.3 Evaluating State Representation

This section focuses on validating *Hypothesis 1*, which means establishing metrics for evaluating the similarity between the learned state space encoded by the LNN and the actual state space of the environment/system. For this analysis, the *cartpole* environment is used, as the state space is small and equations of motion are well understood. Following the learned-actual state space correlation analysis done in section 4.2.2 in from [HS18], the correlation between each state of the system $\chi, \dot{\chi}, \theta, \dot{\theta}$ and each quadratic combination of the latent vector variables is computed. Whereas the latent vector size in [HS18] is equal to the number of states of the *cartpole* environment (4), this section aims to apply the same analysis to arbitrary latent space sizes.

Given that the states of the system can be expressed as non-linear combinations of the other states, the cart linear acceleration $\ddot{\chi}$ and pole angular acceleration $\ddot{\theta}$, finding the quadratic combination of latent states which yields the highest correlation coefficient for each (unobserved) state variable is sufficient as a non-linear combination that can capture second-order dependencies. In practice, this is done by taking the building the matrix $Z_{T\times D}$ where each line is of the form $[z_1, \ldots, z_n, z_1^2, \ldots, z_n^2]$, reducing the method

to finding the best linear mapping between Z and the state matrix $S = [\chi, \dot{\chi}, \theta, \dot{\theta}]_{1:T}$. T represents the episode length, and D the size of the latent space.

At first hand, Ordinary Least Squares regression seems appropriate. Ordinary Least Squares finds the weight matrix that minimizes the l_2-norm between z and s in the following manner:

$$\min_W ||WZ - S||_2^2 \tag{5.1}$$

However, since all model sizes used in evaluation have latent space sizes larger than 4, finding such a mapping is trivial (using least squares regression, for example), since Z becomes close to singular. To avoid this, Ridge regression is used instead, which adds a regularization term that penalizes large weights:

$$\min_W ||WZ - S||_2^2 + \alpha||W||_2^2 \tag{5.2}$$

W is then used to predict the four states of the system $\hat{S}$, and the Pearson correlation coefficient is computed using the following formula:

$$r = \frac{\sum_i \left(S_i - \bar{S}\right)\left(\hat{S}_i - \bar{\hat{S}}\right)}{\sqrt{\sum_i \left(S_i - \bar{S}\right)^2}\sqrt{\sum_i \left(\hat{S}_i - \bar{\hat{S}}\right)^2}} \tag{5.3}$$

$\bar{S}$ represents the mean of vector S.

The experiment scenario is $\text{CfC}_{\text{det}}^{\text{OP}}$ model of size 0.5 (meaning 64 hidden units), trained using a data set of 50K environment steps, after which the algorithm converges towards saturated returns. The $\text{cartpole}_{\text{remove-vel}}$ task is used, in order to test the hypothesis that the CfC-based encoder captures the dynamics of the system only from partially observable state histories. The episode is split into two successive phases: the *swing-up phase*, where the system is controlled to swing the pole upwards, and the *balance phase*, where the pole is kept balanced in the upward orientation. The results are summarized for the initialized and trained model in Figure 5.4a, 5.5a and 5.4b, 5.5b for the swing-up and balanced state, respectively.

Another validation method is to use dimensionality reduction techniques to project the latent state vectors and the actual state vectors to the 2D and plot the projected data points. Following the analysis in section 4.1 from [ZZH+23], Uniform Manifold Approximation and Projection (UMAP) [HM24] is used to project the matrices $Z_{T\times N}$ and $S_{T\times 4}$ to the 2D space. The data points are then colored according to the natural logarithm $\dot{\chi}$ and $\dot{\theta}$. If the hypothesis holds, then point clusters should be colored similarly. Furthermore, the topology of the projected spaces should be similar (rotation invariant). The results are shown for the initialized and trained model in Figure 5.6a, 5.7a and 5.6b, 5.7b for the swing-up and balanced state, respectively.

Finally, it is important to assess the one-step prediction error. In det_{OP} model variants, the loss function (4.7) is an upper bound for the surrogate loss used for deterministic observation predictions. This property holds in deterministic environments. Furthermore, as discussed in Section 2.4.2, and since the planner relies on trajectory sampling using the predictive model, it is important to check the one-step prediction error, as it accumulates along the rollouts. For the fully observable version of *cartpole*, Figure 5.8 and 5.9 show the states at each timestep compared to the one-step state predictions in the first subfigure, while the second subfigure plots predicted state rollouts of length $H = 10$ at each H timesteps. It can be seen that the predicted states are close to the actual states of the environment, indicating a small one-step prediction error. Results for one-step and multi-step prediction error statistics over 10 episodes with different Random Number Generator (rng) seeds are also included in Table 5.1 and 5.2.

Metric	One-step prediction mean/standard deviation	Multi-step prediction mean/standard deviation
$\text{CfC}^{\text{OP}}_{\text{det}}$	2e-05 ± 8.8e-8	1.22e-3 ± 1e-05
$\text{LTC}^{\text{OP}}_{\text{det}}$	6e-4 ± 1e-05	1e-2 ± 7e-05

Table 5.1: Prediction error metrics for $\text{CfC}^{\text{OP}}_{\text{det}}$ and $\text{LTC}^{\text{OP}}_{\text{det}}$ on the fully observable cartpole task.

Metric	One-step prediction mean/standard deviation	Multi-step prediction mean/standard deviation
$\text{CfC}^{\text{OP}}_{\text{det}}$	2.4e-5 ± 8.8e-8	1.2e-3 ± 8e-6
$\text{LTC}^{\text{OP}}_{\text{det}}$	1e-4 ± 2.7e-7	1.3e-3 ± 7e-6

Table 5.2: Prediction error metrics for $\text{CfC}^{\text{OP}}_{\text{det}}$ and $\text{LTC}^{\text{OP}}_{\text{det}}$ on the partially observable $\text{cartpole}_{\text{remove-vel}}$ task.

The same does not hold for $\text{cartpole}_{\text{remove-vel}}$, since it is stochastic. Applying the same force to the cart given two identical position observations can lead to different observation states/targets, since the velocity states are not observable. If ϕ is an unbiased encoder, the problem reduces to a deterministic MDP. As shown above, the LNN encoder manages to generate latent embeddings that are highly correlated with the true environment states. The upper bound analysis described in Section 4.3.4 does not hold here. Thus, the prediction error is empirically analyzed.

Firstly, the time-series of observation prediction loss (4.19) associated with this scenario is included in Figure 5.10, providing an estimate for the expected n-step error under the true observation distribution. While not a true estimate (samples from the experience dataset

are not i.i.d.), the figure at least shows that the mean squared error for next-observation predictions given minibatches of sampled sequences is minimized, and the bias is small (the error converges to ~ 0.02).

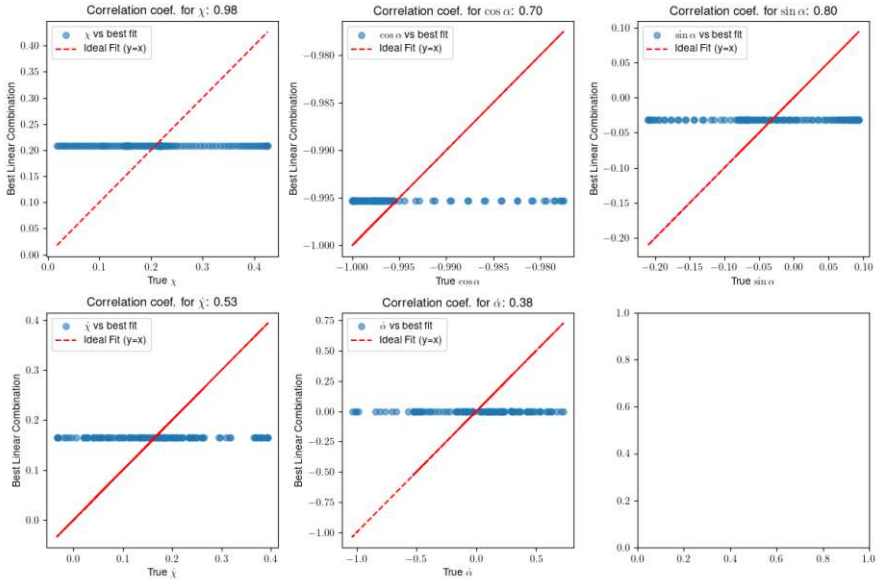
The choice of the number of neurons used by the LNN encoder also influences the state representations. To determine the minimum number of neurons required to model the underlying state of the system, the sparsity of the hidden states is analyzed by computing an *effective rank* for the hidden state matrices $Z_{T\times D}$ during training. The singular values of the matrix Z are computed using Singular Value Decomposition (SVD), and a distribution is defined over the singular values $\sigma_{1:Q}, Q = \min(T, D)$ as follows:

$$p_k = \frac{\sigma_k}{||\sigma||_1}, k \in [1, Q] \tag{5.4}$$

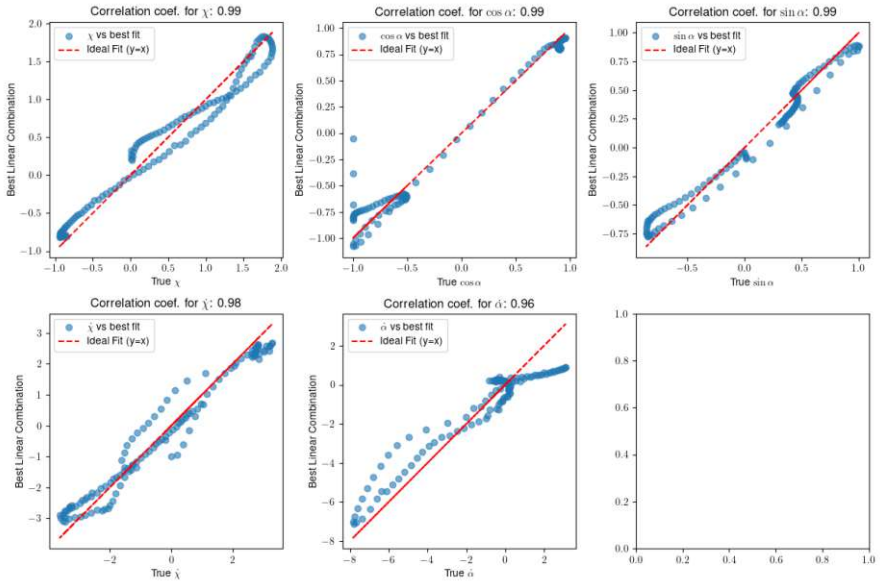
where $||\cdot||_1$ denotes the l_1 norm. The *effective rank* of Z is then [RV07] computed as the exponential of the entropy of the distribution:

$$erank(Z) = e^{\sum_{k=1}^{Q} p_k \log p_k} \tag{5.5}$$

To analyze the impact of the number of LNN on learned representations, the effective rank of Z is logged at each training timestep with different hidden neuron counts, using the $\text{CfC}^{\text{OP}}_{\text{det}}$ architecture variant, on multiple environments $\text{cartpole}_{\text{remove-vel}}$, $\text{cartpole}^{\sigma=0.2}_{\text{perturbed}}$ and acrobot. The results are summarized in Figure 5.11 and 5.12.

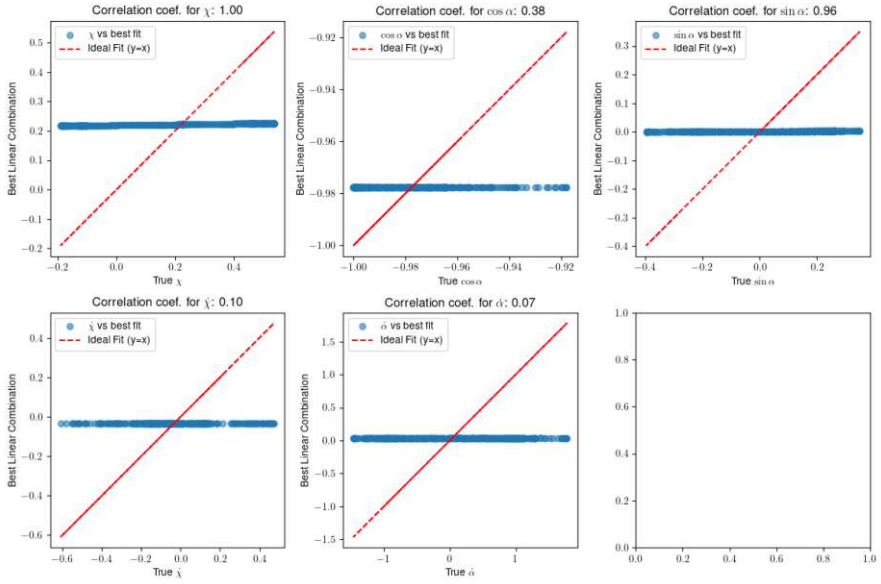


(a) Initialized model.

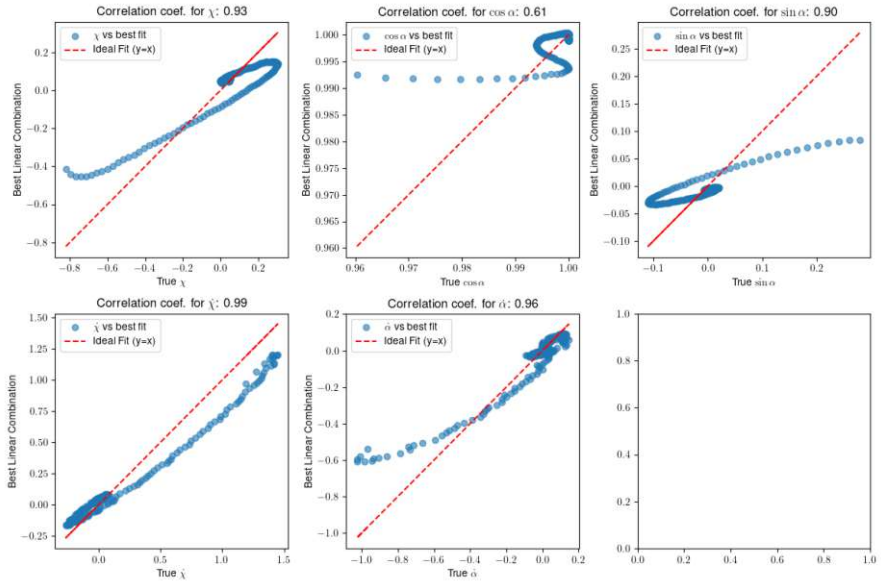


(b) After 120k training steps. Each plot represents the correlation between the best quadratic combination of learned hidden states and the corresponding state of the system. During the swingup phase, forces are applied to the sides of the cart to swing the pole upwards, so the cart traverses the 2D plane in range $[-1, 2]$, as shown in the χ plot. The velocity of the cart $\dot{\chi}$ also varies in range $[-4, 4]$. The pole is swung towards 0°, starting from 180°. Thus, both $\cos\alpha$ and $\sin\alpha$ are scattered across the $[-1, 1]$ range. All correlation coefficients are close to 1, indicating the capability of the LNN-based world model of learning the underlying system dynamics, even with no velocity information.

Figure 5.4: Swingup phase of $\text{cartpole}_{\text{remove-vel}}$.

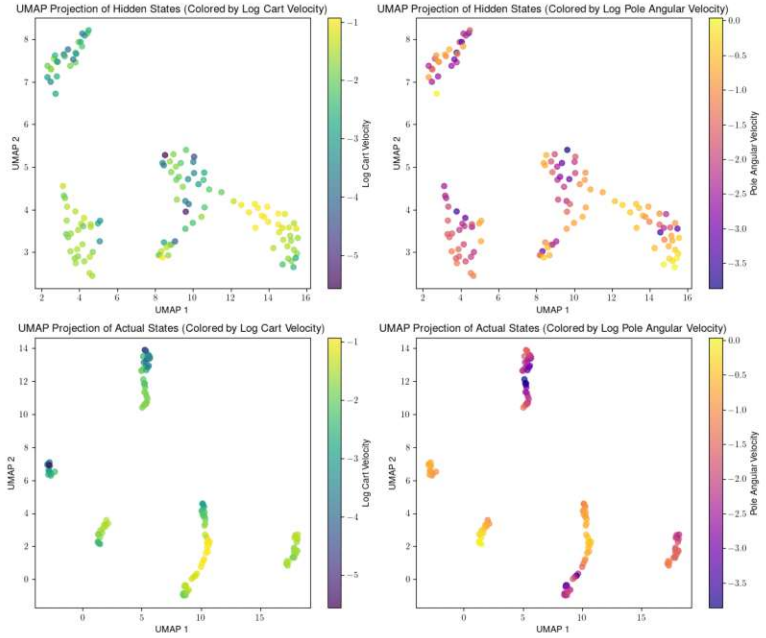


(a) Initialized model.

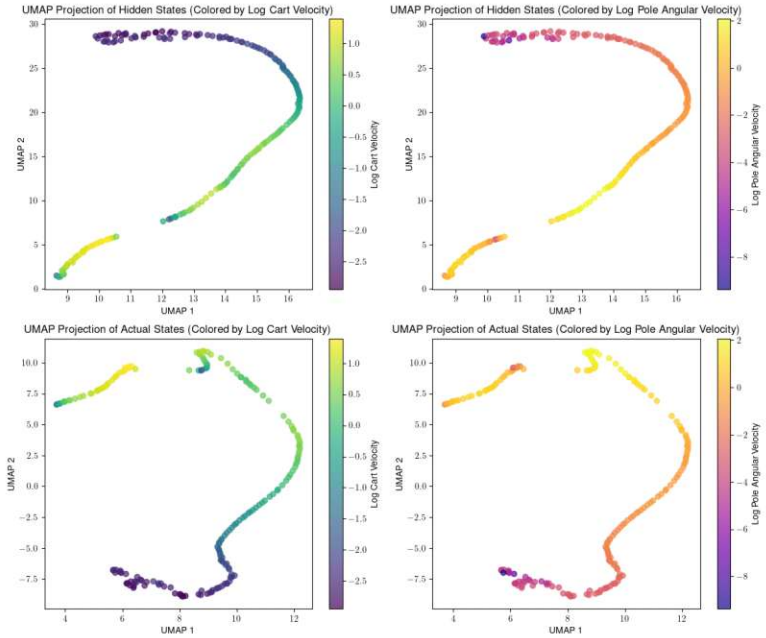


(b) After 120k training steps. Again, correlation is high for all system states. This time, $\cos\alpha$ is clustered around 1, while $\sin\alpha$ is clustered around 0, showing that the pole is balanced around the angle $\alpha = 0°$. The angular velocity $\dot{\alpha}$ and cart velocity $\dot{\chi}$ are also clustered around 0.

Figure 5.5: Balance phase of the cartpole$_{\text{remove-vel}}$.

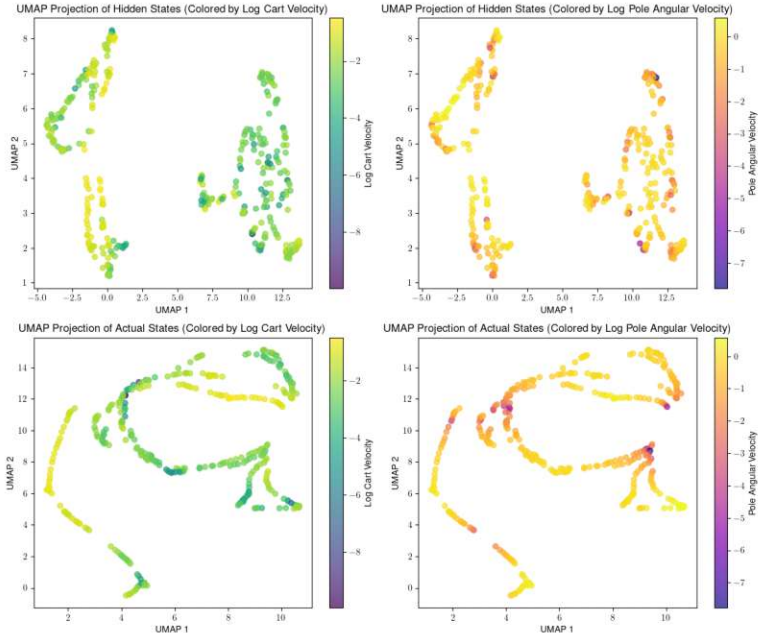


(a) Initial model.

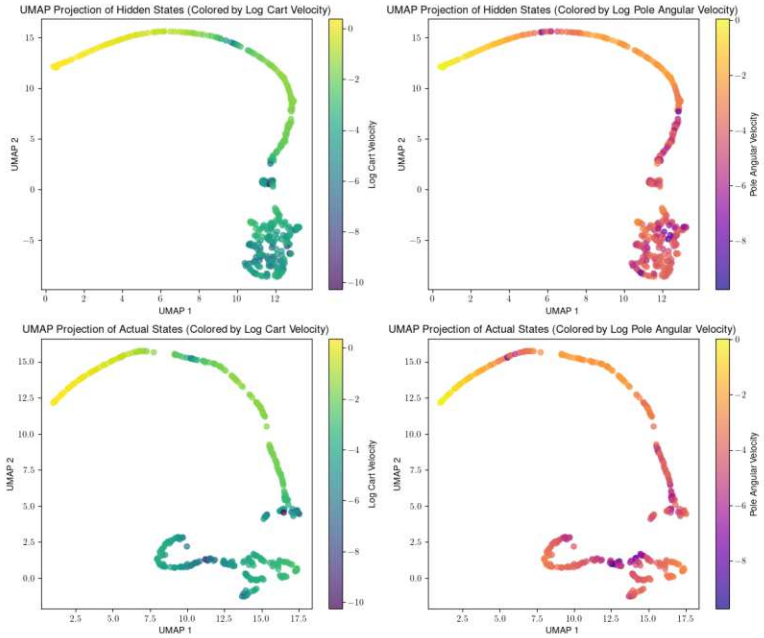


(b) After 120k training steps. On the left: points are colored according to the logarithm of the cart velocity. On the right: points are colored according to the logarithm of the pole angular velocity.

Figure 5.6: UMAP projection to 2D for hidden states and actual states, swingup phase of $\text{cartpole}_{\text{remove-vel}}$.

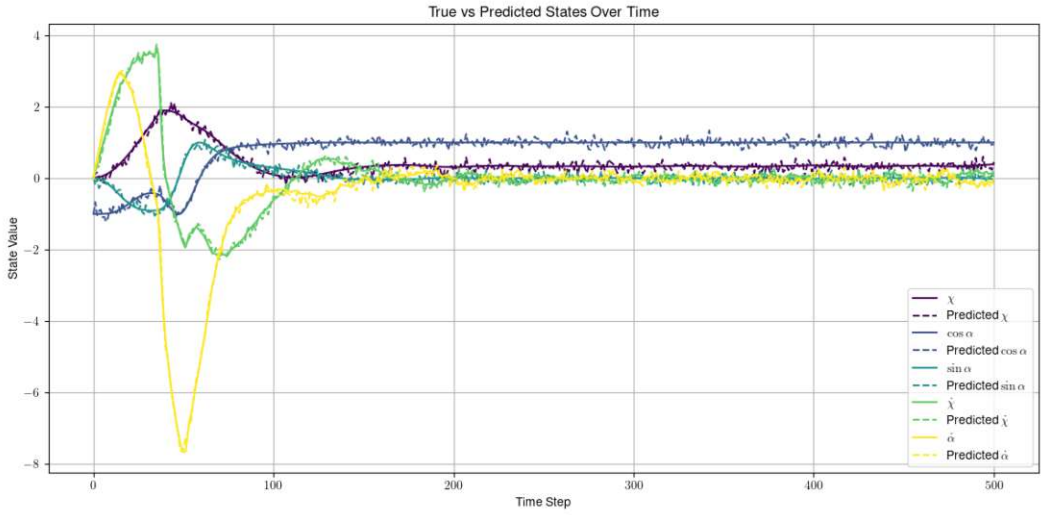


(a) Initial model.

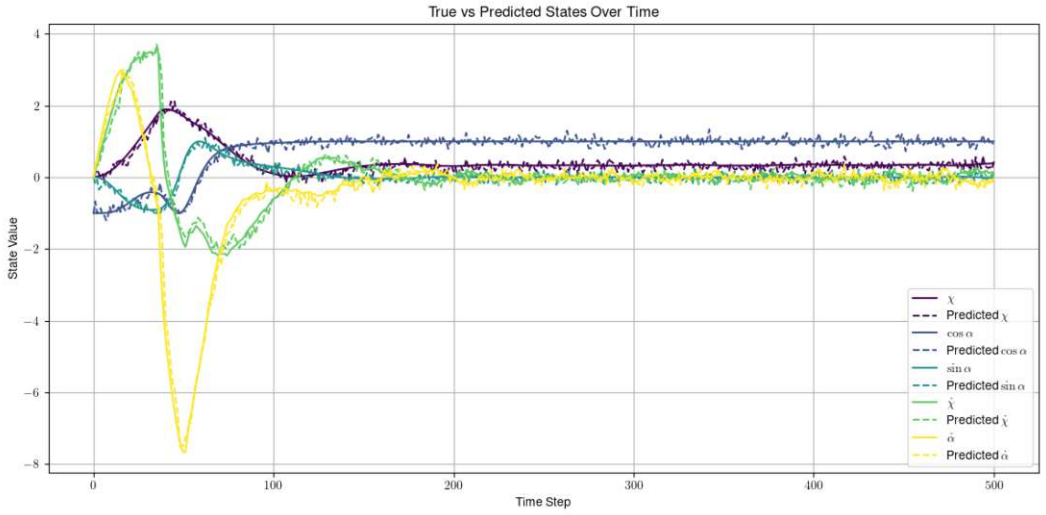


(b) After 120k training steps.

Figure 5.7: UMAP projection to 2D for hidden states and actual states, balance phase of $\text{cartpole}_{\text{remove-vel}}$.

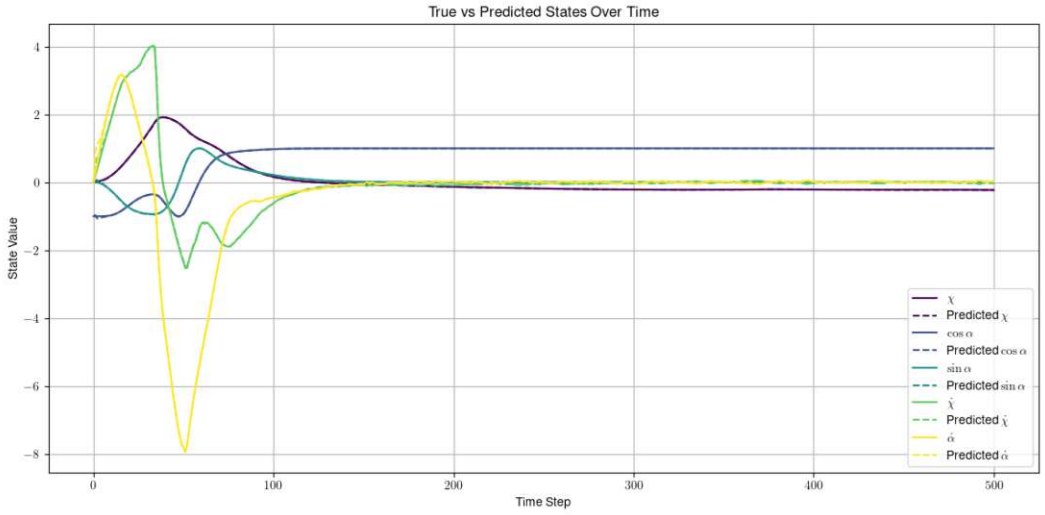


(a) Plot of one-step state predictions vs. actual states.

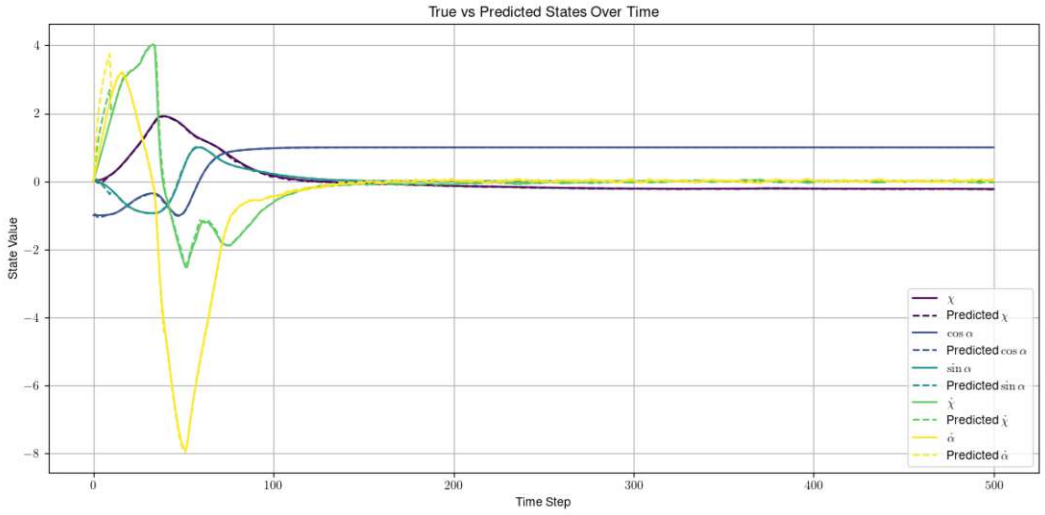


(b) Plot of 10-step state predictions vs. actual states.

Figure 5.8: State predictions vs. actual states over the runtime of an episode of cartpole using an $\text{CfC}^{\text{OP}}_{\text{det}}$ model trained over 50K environment steps. Corresponding states and predicted states are colored identically, and the latter are represented as dashed lines.



(a) Plot of one-step state predictions vs. actual states.



(b) Plot of 10-step state predictions vs. actual states.

Figure 5.9: State predictions vs. actual states over the runtime of an episode of cartpole using an $\text{LTC}^{\text{OP}}_{\text{det}}$ model trained over 50K environment steps. Corresponding states and predicted states are colored identically, and the latter are represented as dashed lines.

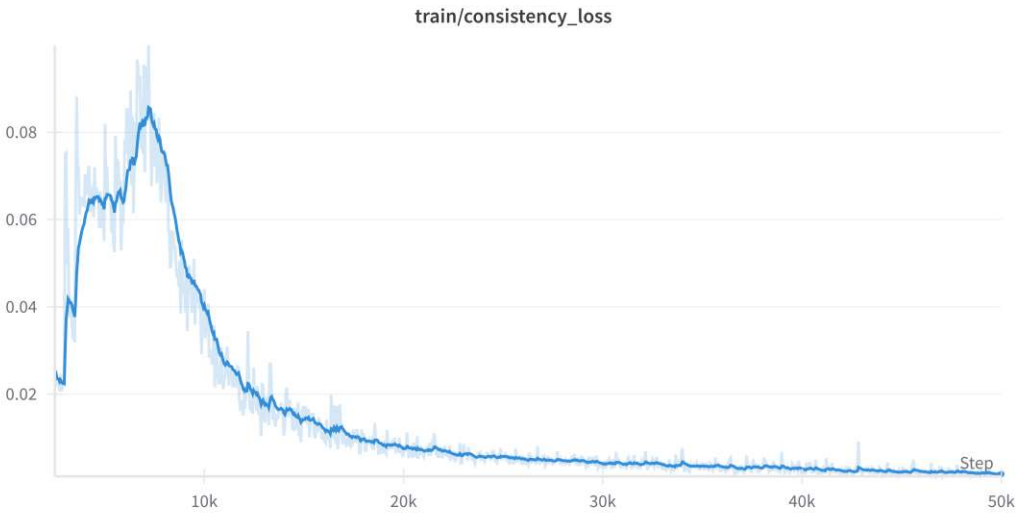
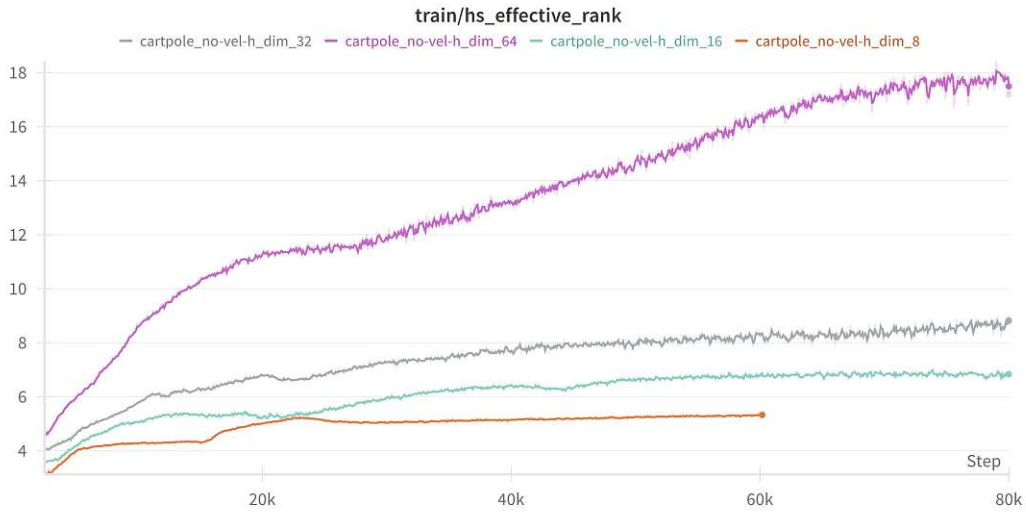
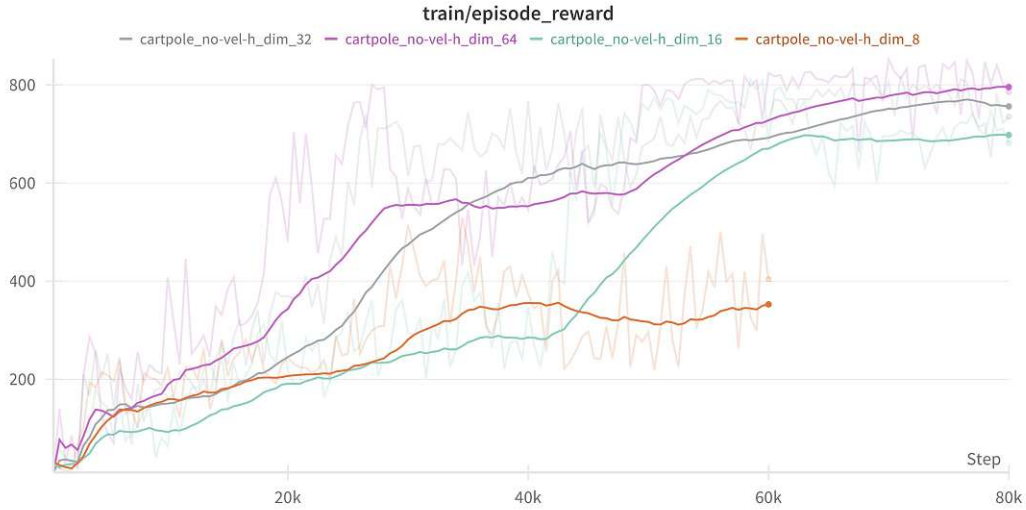


Figure 5.10: Observation prediction loss for $\text{CfC}^{\text{OP}}_{\text{det}}$ on $\text{cartpole}_{\text{remove-vel}}$ over 50K environment steps.

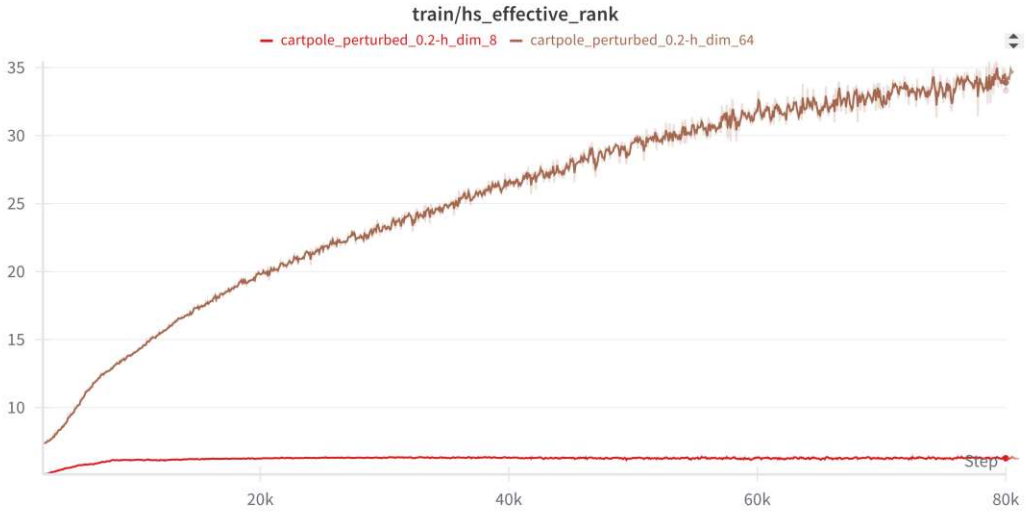


(a) Effective rank of hidden state matrix of dimensions sequence length × number of neurons over time during training. Using 64 neurons saturates the effective rank at ∼ 18, indicating sparse, highly-correlated learned states.



(b) Episodic return during training for sizes 8, 16, 32 and 64. The 32 and 64 neuron configurations yield the highest returns. Given that the effective rank associated with 32 neurons saturates at ∼ 9 early during training, a size larger than 32 is redundant for this task.

Figure 5.11: The effective rank of Z at each training timestep with different hidden neuron counts, using the $\text{CfC}^{\text{OP}}_{\text{det}}$ architecture variant.



(a) Effective rank of hidden state matrix of dimensions sequence length × number of neurons over time during training. Using 8 neurons saturates the effective rank at ∼ 9 earlier, indicating condensed representations that are close to the number of modelled states: 4 states for the agent, mean and standard deviation for both noise Gaussian and flickering Gaussian.

(b) Episodic return during training for sizes 8 and 64 neurons. Using only 8 neurons leads to convergence and more consistent returns during training.

Figure 5.12: The effective rank of Z at each training timestep with different hidden neuron counts, using the $\text{CfC}^{\text{OP}}_{\text{det}}$ architecture variant with noise and flickering.

5.4 Regression Testing

In this section, the stability training process of all model variants on a challenging MDP is analysed. This connects back to *Hypothesis 2*. The acrobot-swingup task is chosen for this scenario. Since TD-MPC2 cannot learn an optimal solution for this task, the chances of instability in the training process for the introduced models are higher in this case. The hypothesis that learning a TD-MPC2 world model which uses an LNN-based history encoder does not degrade the agent performance is assessed. In other words, no regression is introduced.

Figure 5.13 illustrate relevant training plots for stability analysis. Oscillation in the value loss associated with (4.18) would indicate high variance, caused by the following: the value targets (4.17) computing using fresh samples from the experience dataset $\mathcal{D}$ are out-of-distribution with respect to the estimated Q_θ distribution (overfitting). Looking at Equation 1 from [FLRP$^+$19], the estimated value error is proportional to an overfitting term and a bias term. Unless $Q_\theta = Q^*$ (the value estimate perfectly matches the optimal value function), which is only theoretically possible in the limit after covering the entire $\mathcal{S} \times \mathcal{A}$ space, an approximately constant value loss over time must indicate the bias, which is upper bounded by the next-state prediction error [FLRP$^+$19] (2.99).

The consistency loss is expected to have two distinct phases, similar to the two-phase nature of an episode of each of the tasks selected for evaluation: the swingup-balance phases in *cartpole* and *acrobot*, and the stand-gait phases in *walker*. Only first phase is initially observed during environment interaction. As the behavior policy improves, the experience dataset is diversified by the addition of transitions which include observations from the second phase. Thus, for the deterministic models, the MSE surrogate consistency loss (4.19) initially increases, and then proceeds to decay over time. Similarly, for the stoch-OP models, the negative log likelihood loss (4.13) is expected to decay over time, indicating that the learned distribution approaches the underlying next-observation distribution. The continuous regression is not necessarily converging asymptotically to 0 for det-ZP, but rather to the asymptotic bias of the next-latent state predictor given samples from the history encoder ϕ.

The *pi_scale* plot illustrates the running scale over the estimated Q values for the batches of encoded ground-truth history sequences and sampled action sequences from the policy prior. A rising plot indicates a high credit assigned by the model. A sudden decrease indicates that the model accounts for the former *overconfidence* in the estimated returns. A stagnant curve indicates a lack of further learning.

Finally, the norm of the gradients of the world model loss (4.11) w.r.t its parameters θ is illustrated, in order to assess whether the *exploding/vanishing gradients* phenomena occur during training. Similarly, the norm of the gradients associated with the policy prior parameters ξ is also shown.

Figure 6.3 illustrates an episode of a solved variant of the cartpole-swingup task, with frames sampled at different timesteps. Similarly, Figure 6.1 illustrates an episode of the

Figure 5.13: Training plots for TD-MPC2, $\text{CfC}^{\text{OP}}_{\text{det}}$ and $\text{CfC}^{\text{ZP}}_{\text{det}}$ on the acrobot_swingup task. The y-axis in the consistency_loss and grad_norm plots is log-scale. The gradients are stable for all models, and the steadiness of the value loss indicates an absence of overfitting. The smooth decrease in policy loss indicates that the entropy is slowly minimized, as the training of the policy parameters converges towards maximizing return estimates. The running scale of the Q estimates is also increasing over time, especially for the best-performing det-OP model.

solved acrobot-swingup task.

5.5 Agent Performance Comparison on All Control Tasks With Proprioceptive Inputs

This section presents performance scores for all model variants on the tasks described in Section 5.1, providing evidence for *Hypothesis 3*. Due to limited resources for training, all models presented in this work are trained over 200K environment steps only. As shown in the plots showing the episodic return for all tasks during training, however, 200K steps already lead to convergence. The models are tested against two baselines: TD-MPC2 and DreamerV3 [HPBL], as two state-of-the-art MBRL algorithms. The scores for the baselines are chosen as the scores reported in [HMa] by the authors of [HSW], as they include achieved episodic returns for trained TD-MPC2 and DreamerV3 models after each 100 000 steps. To make the comparison as fair as possible, the 200 K-step checkpoints are chosen. Worth noting is that the baseline TD-MPC2 model has approximately 5M learnable parameters and the DreamerV3 model has around 20M, whereas the proposed models are much smaller (below 500K learnable parameters, as described in Section 5.2).

Episodic returns achieved during training the proposed models on the fully-observable tasks are shown in Figure 5.14, while the mean and standard deviation of returns over 10 episodes for all models and baselines, with unique rng seeds, are reported in Table 5.3 (shared seeds across models). Due to resource constraints, training runs that converge early are stopped before reaching 200K training steps. The standard, 5M parameters TD-MPC2 model still achieves returns on cartpole-swingup and walker-run, although results with some of the proposed approaches are comparable. Notably, performance on acrobot-swingup is improved with all LNN-TD-MPC2 models besides the $^{OP}_{\text{stoch}}$ models, with an optimal performance achieved using $\text{CfC}^{ZP}_{\text{det}}$.

The benefits of using the LNN-based history encoders are highlighted by the results on the partially observable task variants included in Table 5.4, 5.5 and 5.6. While not all models reach the optimal scores for all tasks, the results nevertheless indicate that LNN-TD-MPC2 is more robust. The stochastic model $^{OP}_{\text{stoch}}$ tends to decay in performance the least, with increasing sensor noise variance. However, it also typically achieves the lowest performance ceiling out of all LNN-TD-MPC2 variants. The difference in performance is more clearly illustrated in Figure 5.15, which illustrate the robustness added by the LNN encoder.

Another highlight is the sample-efficiency of the LTC-based models cartpole-sparse. The authors of [HSW],[HMa] report a score of 1.0 after 100K training steps, whereas $\text{LTC}^{OP}_{\text{det}}$ converges after approximately 50K steps even earlier, while $\text{LTC}^{ZP}_{\text{det}}$ converges even earlier - after just 50K steps.

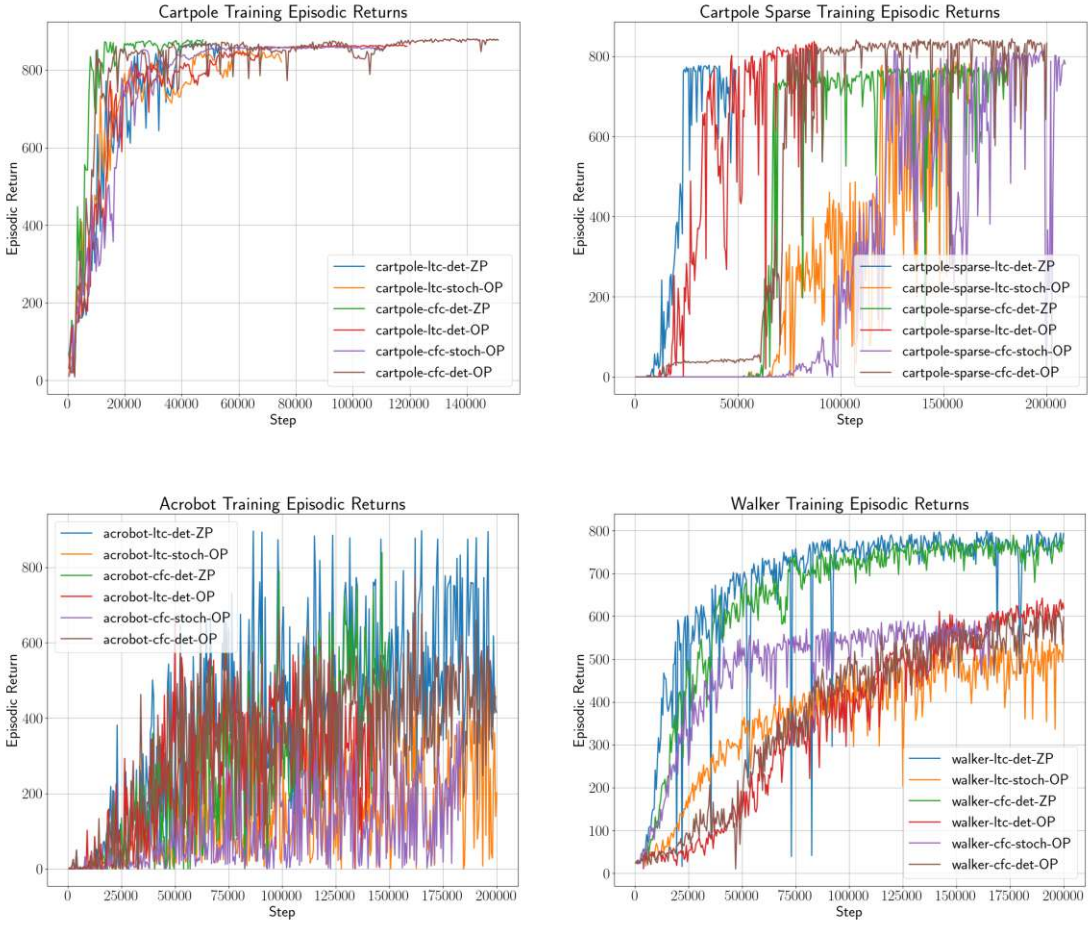


Figure 5.14: Episodic returns over time during training the LNN-TD-MPC2 variants, on the fully observable tasks.

Model	cartpole	cartpole-sparse	$\text{acrobot}_{\text{swingup}}$	walker-run
TD-MPC2	863.9	784.8	329.0	830.9
DreamerV3	906.5	258.6 (735.0 after 300K)	154.5	632.7
$\text{CfC}^{\text{ZP}}_{\text{det}}$	856.0	769.4±0.92	883.1±0.2	785.98 ± 16.14
$\text{CfC}^{\text{OP}}_{\text{det}}$	880.0 ± 6.7	799.5±3.14	544.7±6.73	632.7 ± 34.12
$\text{CfC}^{\text{OP}}_{\text{stoch}}$	856.0 ± 7.4	805.90 ± 5.20	263.26 ± 61.96	724.32 ± 16.03
$\text{LTC}^{\text{ZP}}_{\text{det}}$	822.04 ± 0.90	768.40 ± 0.97	769.29 ± 0.62	767.53 ± 24.48
$\text{LTC}^{\text{OP}}_{\text{det}}$	860.0± 9.04	726.85 ± 82.04	284.80 ± 102.12	653.81 ± 13.59
$\text{LTC}^{\text{OP}}_{\text{stoch}}$	824.35 ± 12.08	752.40 ± 23.98	536.36 ± 2.54	633.60 ± 40.63

Table 5.3: Performance across four key environments: *cartpole*, *cartpole-sparse*, *acrobot-swingup*, and *walker-run*.

Model	$\text{cartpole}_{\text{remove-vel}}$	$\text{cartpole}^{\sigma=0.1}_{\text{perturbed-flickering}}$	$\text{cartpole}^{\sigma=0.2}_{\text{perturbed-flickering}}$	$\text{cartpole}^{\sigma=0.3}_{\text{perturbed-flickering}}$
TD-MPC2	193.22 ± 13.68	378.82 ± 45.43	353.63 ± 76.27	365.71 ± 47.37
$\text{CfC}^{\text{ZP}}_{\text{det}}$	603.0	798.89 ± 50.64	742.19 ± 52.02	550.69 ± 76.55
$\text{CfC}^{\text{OP}}_{\text{det}}$	868.27 ± 2.12	878.0 ± 34.07	756±70.0	645.49 ± 92.07
$\text{CfC}^{\text{OP}}_{\text{stoch}}$	705.80 ± 103.31	825 ± 25.45	693.08 ± 54.02	600.22 ± 79.86
$\text{LTC}^{\text{ZP}}_{\text{det}}$	726.85 ± 82.04	780.85 ± 29.36	736.24 ± 74.14	677.99 ± 52.82
$\text{LTC}^{\text{OP}}_{\text{det}}$	846.07 ± 2.65	824.35 ± 16.02	744.14 ± 36.91	471.47 ± 104.39
$\text{LTC}^{\text{OP}}_{\text{stoch}}$	724.50 ± 63.09	804.30 ± 54.22	763.06 ± 55.91	721.52 ± 78.38

Table 5.4: Performance on *cartpole-swingup* variants: no velocity and with varying sensor noise levels and sensor flickering.

Model	$\text{acrobot}^{\sigma=0.1}_{\text{swingup}}$	$\text{acrobot}^{\sigma=0.2}_{\text{swingup}}$	$\text{acrobot}^{\sigma=0.3}_{\text{swingup}}$
TD-MPC2	90.50 ± 104.67	57.74 ± 55.10	74.80 ± 41.12
$\text{CfC}^{\text{ZP}}_{\text{det}}$	559.78 ± 28.61	448.90 ± 75.03	322.62 ± 113.33
$\text{CfC}^{\text{OP}}_{\text{det}}$	564.55 ± 35.14	495.90 ± 48.82	374.15 ± 47.69
$\text{CfC}^{\text{OP}}_{\text{stoch}}$	272.56 ± 93.26	285.83 ± 56.31	239.59 ± 161.22
$\text{LTC}^{\text{ZP}}_{\text{det}}$	468.84 ± 44.98	370.50 ± 51.94	281.79 ± 64.72
$\text{LTC}^{\text{OP}}_{\text{det}}$	326.23 ± 72.63	307.21 ± 76.20	224.97 ± 121.64
$\text{LTC}^{\text{OP}}_{\text{stoch}}$	263.19 ± 71.27	254.91 ± 111.96	201.62 ± 95.31

Table 5.5: Performance on *acrobot-swingup* under perturbations of varying noise levels.

Model	$\text{walker-run}^{\sigma=0.1}_{\text{perturbed}}$	$\text{walker-run}^{\sigma=0.2}_{\text{perturbed}}$	$\text{walker-run}^{\sigma=0.3}_{\text{perturbed}}$
TD-MPC2	581.92 ± 37.27	495.75 ± 38.88	387.69 ± 38.61
$\text{CfC}^{\text{ZP}}_{\text{det}}$	764.30 ± 15.06	734.35 ± 22.53	647.65 ± 38.56
$\text{CfC}^{\text{OP}}_{\text{det}}$	589.87 ± 20.48	569.53 ± 36.58	519.73 ± 26.55
$\text{CfC}^{\text{OP}}_{\text{stoch}}$	697.32 ± 17.28	672.14 ± 12.43	630.33 ± 27.61
$\text{LTC}^{\text{ZP}}_{\text{det}}$	714.06 ± 11.12	607.04 ± 19.83	546.43 ± 47.25
$\text{LTC}^{\text{OP}}_{\text{det}}$	570.15 ± 20.48	454.38 ± 49.65	501.66 ± 39.56
$\text{LTC}^{\text{OP}}_{\text{stoch}}$	642.81 ± 17.38	597.67 ± 32.84	485.32 ± 64.43

Table 5.6: Performance on *walker-run* under perturbations of varying noise levels.

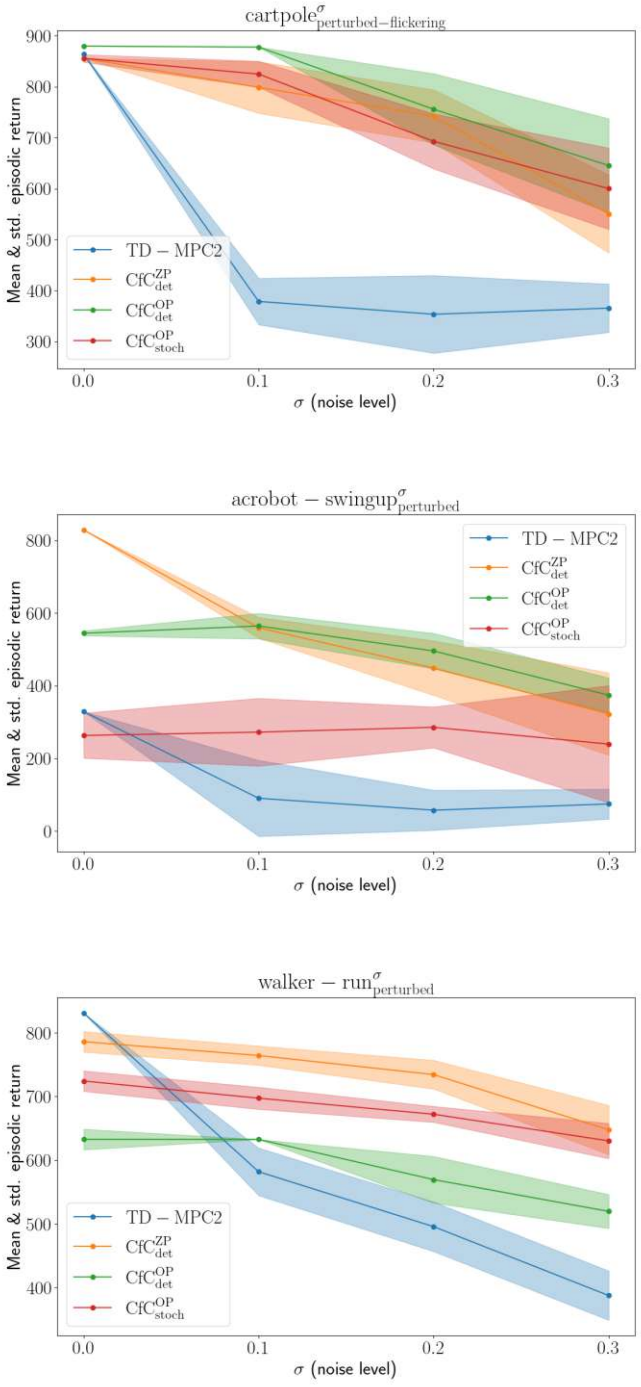


Figure 5.15: Mean and standard deviation of returns achieved by TD-MPC2 and CfC-based models, on each partially observable task with sensor noise.

5.6 Comparison of Self-Predictive and Observation-Predictive Models

Hypothesis 5 is validated by the results reported in Table 5.3, 5.4, 5.5 and 5.6. Observation-predictive models perform better on tasks with sparse rewards (see cartpole-sparse scores) and tasks with specific states omitted from observations, such as velocity (see $\text{cartpole}_{\text{remove-vel}}$ scores). The episodic returns during training for the sparse reward task also reflect the advantage of the observation-predictive model in the absence of a continuous reward signal. Furthermore, the stochastic models stoch-OP deal better with the uncertainty introduced by sensor noise or flickering (see $\text{cartpole}^{\sigma}_{\text{perturbed}}$ results), and prove more robust under increasing noise variance. Figure 5.17 illustrates relevant metrics during training the stoch-OP model on task $\text{cartpole}^{\sigma=0.1}_{\text{perturbed}}$. Specifically, the variance of the parameterized next-step observation distribution decays during learning, towards a value of ~ 0.25. While this variance is higher than the variance of the normal distribution, which provides noise samples for the observations, the task also artificially *flickers* observations with probabilities sampled from a normal distribution of $\sigma^2 = 0.2$. This can explain the higher uncertainty of the predictive model. Similarly, the NLL of the ground-truth observations under the learned distribution also decays over time, showing that the learned distributions fit the underlying distributions. Figure 5.16 shows more training plots for the stochastic model on the walker-run task, highlighting the stability and convergence of the learning process.

However, as noted before, the stochastic observation-predictive model tends to hit a performance ceiling on tasks with more complex dynamics. Furthermore, as reflected in the training plots in Figure 5.14, the model requires more training samples to improve, compared to all other approaches. Regarding the more complex dynamics of acrobot and walker, training the LNN encoders for future observation predictions seems to hinder the model performance, in contrast to the simpler goal of latent state consistency in the self-predictive approaches. These results reflect and strengthen the argument from [HSW], which states that reconstructive models are not required to solve continuous control tasks.

On the other hand, the self-predictive models handle fully observable complex dynamics better, as reflected by results for harder continuous control problems such as *acrobot-swingup* (which is unsolved by the baselines) and *walker-run*, included in Table 5.3.

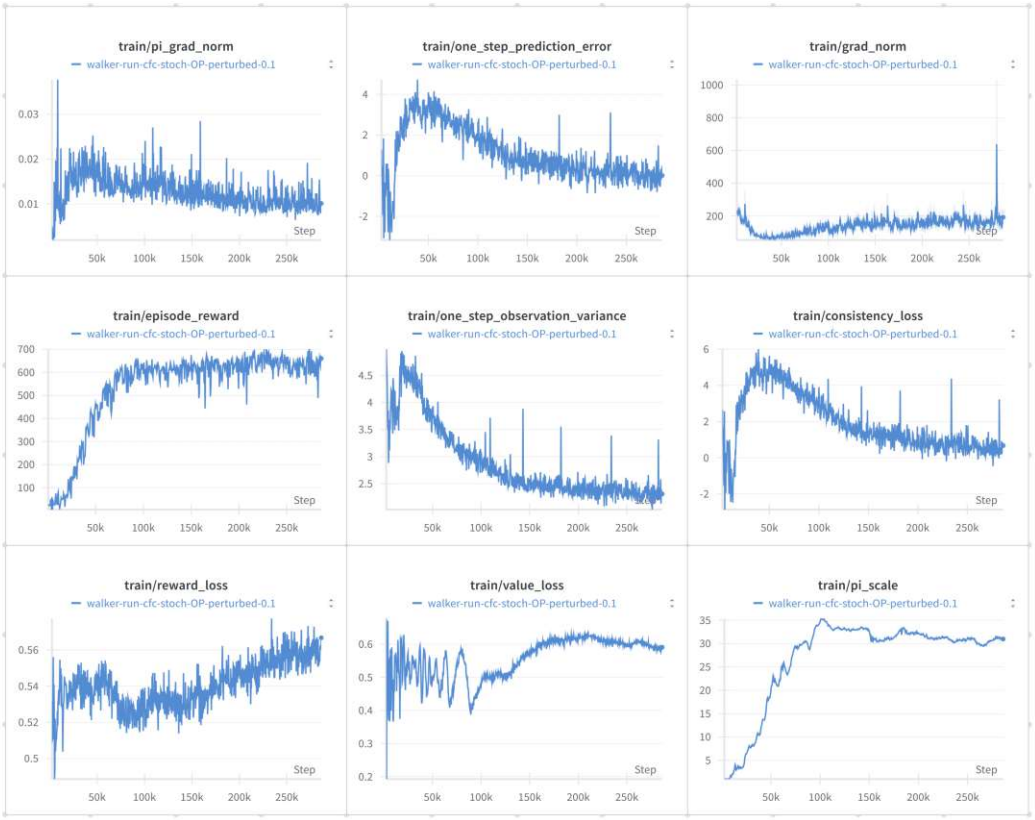
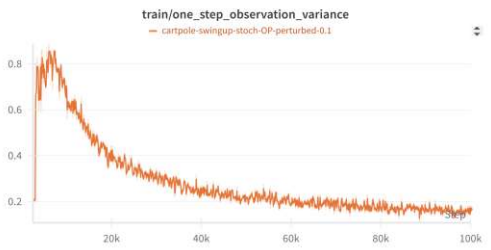
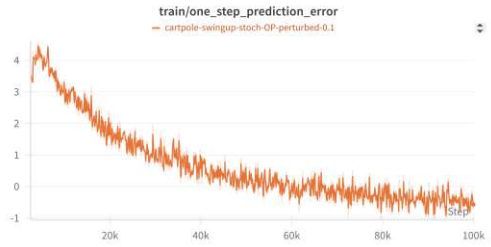


Figure 5.16: Training plots for $\text{CfC}^{\text{OP}}_{\text{stoch}}$ on $\text{walker}^{\sigma=0.1}_{\text{perturbed}}$.



(a) One-step observation distribution variance averaged over sampled batch during training.



(b) One-step NLL of sampled ground-truth observation batch over time. Values are interpreted as densities instead of probabilities, so values are allowed to be negative.

Figure 5.17: Relevant training plots for $\text{CfC}^{\text{OP}}_{\text{stoch}}$ on $\text{cartpole}^{\sigma=0.1}_{\text{perturbed}}$ over 100K steps.

5.7 Performance Comparison on Tasks with Irregularly Sampled Observations

To evaluate the performance of LNN-TD-MPC2 on tasks with irregular time delays in observations, the cartpole-swingup and walker-run tasks are adapted by introducing stochastic observation delays (see Section 5.1) at different levels. For the former, the following maximum values for the discrete random observation delay are used: $3, 6, 10$. Since the state space of the cartpole-swingup task is small, these larger steps are used to highlight the difference in performance across the tested models. The walker-run is comparatively more complex, in terms of state space dimensionality. Thus, smaller increments in maximum random delays are used: $1, 2, 3$.

The mean and standard deviation of episodic returns across 10 trials, achieved by the baseline TD-MPC2 and $\text{LTC}^{\text{ZP}}_{\text{det}}$ are highlighted in Figure 5.18. The stability of TD-MPC2 collapses on the walker-run, with increasing observation delays, while it maintains a decent performance on cartpole-swingup. The LTC-based model efficiently constructs state representations from irregularly-sampled observations, leading to more consistent and stable returns, especially on the more complex walker-run task.

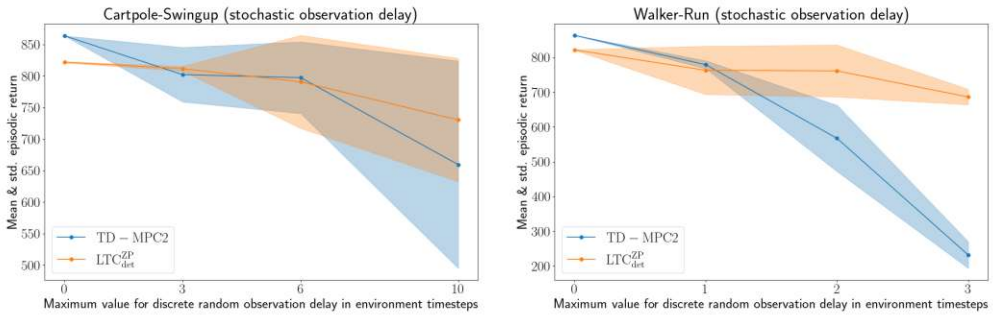
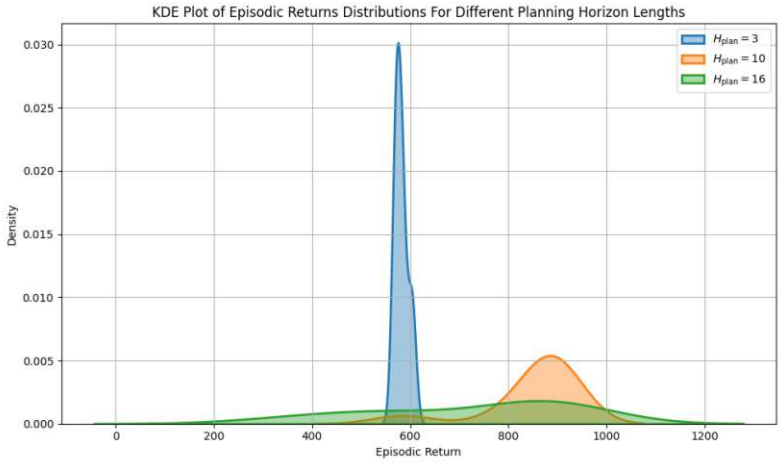


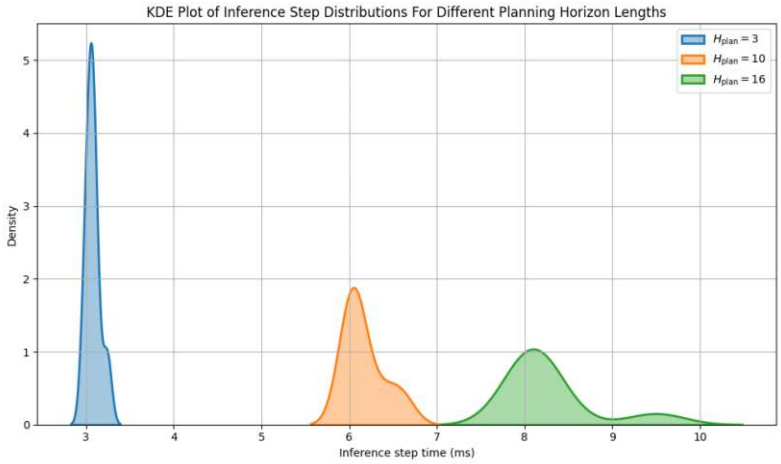
Figure 5.18: Mean and standard deviation of returns achieved by TD-MPC2 and $\text{LTC}^{\text{ZP}}_{\text{det}}$, on cartpole-swingup and walker-run with various upper bounds on the introduced discrete stochastic observation delay.

5.8 Impact of Planning Horizon Length

To analyze the effect of increasing the planning horizon on agent performance, the $\text{CfC}^{\text{det}}_{ZP}$ model that solves the acrobot-swingup task (with the default planning horizon value of 10), is tested using different horizon lengths for 10 trials. Each trial is initialized with a unique rng seed. The results are shown in Figure 5.19. While using a shorter planning horizon greatly decreases the execution time required by the planner, the achieved returns are also greatly diminished. On the other hand, using a H_{plan} value higher than the default value of 10, such as 16, increases the variance of the collected returns across trials, making the overall agent more inconsistent at solving the task. Furthermore, the execution/inference time also increases, a further drawback for real-time systems.



(a) Probability density functions of episodic returns for 10 episodes of acrobot-swingup, using $\text{CfC}^{\text{det}}_{ZP}$ and different planning horizon lengths. Extending the planning horizon to 10 clearly leads to increased consistent performance, even with an outlier return of ~ 600. Increasing the planning horizon beyond 10 leads to diminishing returns, as returns achieved with $H_{\text{plan}} = 16$ present a much higher variance.



(b) Probability density functions of planner execution time for 10 episodes of acrobot-swingup, using $\text{CfC}^{\text{det}}_{ZP}$ and different planning horizon lengths. Execution time increases linearly with the planning horizon.

Figure 5.19: Performance comparison of $\text{CfC}^{\text{det}}_{ZP}$ on the acrobot-swingup task, using different planning horizon length. Performance is defined in this case as achieved episodic returns and the time complexity of the planner. The space complexity is identical, since the number of model parameters and MPPI samples is identical across runs. Plotted probability density functions are computed using kernel density estimation.

CHAPTER 6

Conclusions

6.1 Summary of Findings

Designing end-to-end self-learning agents poses a great challenge for solving continuous control tasks under noisy or flickering sensor data, especially when data arrives at irregular timesteps. TD-MPC2, the state of the art MBRL algorithm, efficiently solves such tasks under the assumption of fully observable states, and outperforms other algorithms on standard control benchmarks in simulation. However, as shown in Chapter 5, Section 5.6, this performance does not necessarily extend to partially observable task variants. This thesis leverages LNNs to build sufficiently predictive state representations by encoding the full history of observations and actions of the agent throughout an episode run, and improves TD-MPC2 to tackle more challenging robotics tasks.

As shown in Section 5.3, the state representations of the proposed LNN-based models have the potential to closely resemble the underlying states of the partially-observed process. This is further reflected by the results on one-step and multi-step state prediction tests, which illustrate a predicted trajectory that is close to the actual trajectory of the controlled dynamical system.

General benchmarks on the fully observable tasks described in Section 4.1, 5.1, and 5.2 show that the proposed extensions to the TD-MPC2 algorithm achieve similar performance to the baseline algorithm, which already outperforms the previous state of the art DreamerV3. A key finding here is that the proposed self-predictive model achieves optimal performance (as defined by the creators of the tasks) on acrobot-swingup, which is unsolved by the baseline algorithms. Judging by results reported in [HPBL] and [HSW], acrobot-swingup represents the most challenging task from the *dm-control* suite. The time-constant adaptivity of LNNs allows the model to efficiently recover the state dynamics that contribute to achieving higher rewards. Notably, the observation-predictive model does not reach the same performance, even though it performs better than the

baseline. Since the *acrobot* system is underactuated, one hypothesis is that predicting the states is not as important for solving the task as is modeling the dynamics relevant to properly control the system to reach the defined objective. In this sense, the TD-MPC2's intentional design of learning reward-centric latent models, the use of MPPI for planning, as well as the expressivity of LNNs for capturing relevant dynamics, contribute towards solving this particularly challenging task.

The findings reported in Section 5.6 suggest that the proposed extensions are more equipped to tackle control tasks with sensor noise flickering. The performance ceiling on the chosen tasks is higher than the one achieved by standard TD-MPC2, and the proposed models are much more robust at increasing noise levels.

Finally, the capability of LNNs at processing inputs that arrive at irregular timesteps allows the proposed approaches to achieve good performance on SDMDP variants of the chosen tasks, whereas the performance of the standard TD-MPC2 collapses once observations are delayed.

6.2 Future Work

While the performance of LNN-TD-MPC2 reported in this work is promising, the evaluation scope is limited. Due to the scope induced by multiple model variants, hyperparameter choice, time and computing resources, and number of trials required for evaluating the hypotheses defined in Section 5.1, Chapter 5 focuses on assessing performance on selected simulation environments under diverse scenarios. However, the end-goal of deploying the proposed methods on hardware remains delegated to potential future research. For this to occur, the *sim-to-real* gap has to be addressed - a non-trivial challenge. Nevertheless, the sample efficiency and the relatively-high level of robustness to uncertainty (in comparison to baselines) shown by LNN-TD-MPC2 in the limited evaluation scope of Chapter 5 could extend to actual robotics systems - an exciting potential avenue of research.

Some of the scores reported in Section 5.5, 5.6 suggest there is room for improvement in the design and/or in the implementation of the proposed architectures. For example, the stochastic observation-predictive model seems perfectly capable of capturing the randomness induced by the artificially-added sensor noise in the *cartpole-swingup* and *walker-run* tasks. However, the algorithm converges to suboptimal returns, below the *optimal* scores of 800. While the achieved score is still higher than the baseline, the simultaneous finding of successful convergence and suboptimal returns indicate possible issues in the implementation. This will be investigated and addressed in future work.

Finally, the theoretical analysis hinted at in Section 4.3.4 can be extended, as it was entirely based on previous analysis on bounds on approximate information state generators [SSSM22] [AHL16] [FLRP+19] [LXL+] [NES+]. For example, the deterministic observation-predictive model($<\text{LNN}>^{\text{OP}}_{\text{det}}$) has no theoretical grounding, and only empirical results validate the efficiency of the model in some scenarios. In fact, [NES+],

[SSSM22] directly suggest that the next-step deterministic observation prediction is an insufficient condition for efficient approximate information state generators, even in fully-observable contexts. Furthermore, expected next-latent state predictions ($<\text{LNN}>^{\text{ZP}}_{\text{det}}$) are also not theoretically founded when applied to POMDPs. This remains an open challenge in the domain, not just within the scope of this thesis.

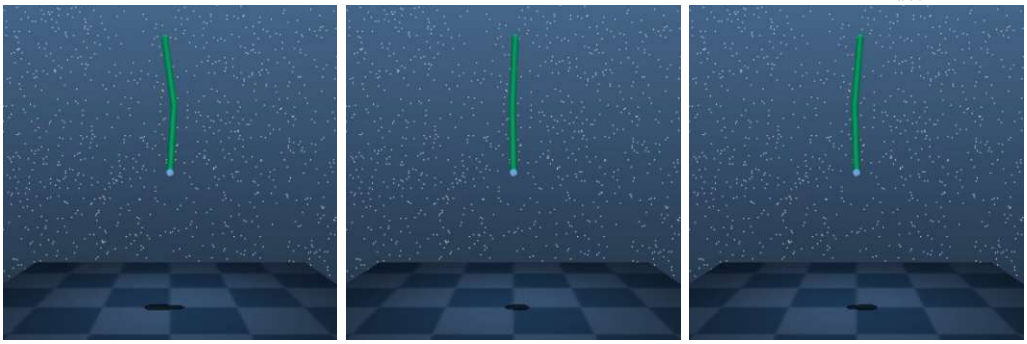
Appendix

Task	State Space	Observation Space	Action Space	Reward Type	Reward Function (if Dense)
cartpole swingup	4D (χ, θ, $\dot{\chi}$, $\dot{\theta}$ $\sin(\theta)$)	5D (χ, $\cos\theta$, $\sin\theta$, $\dot{\chi}$, $\dot{\theta}$)	1D (torque)	Dense	Pole upright angle, small control effort, low angular velocity, and cart centering
cartpole swingup sparse	4D (χ, θ, $\dot{\chi}$, $\dot{\theta}$ $\sin(\theta)$)	5D (χ, $\cos\theta$, $\sin\theta$, $\dot{\chi}$, $\dot{\theta}$)	1D (torque)	Sparse	r = 1 if $\cos(\theta) > 0.95$, else r = 0
cartpole swingup no velocity	4D (χ, θ, $\dot{\chi}$, $\dot{\theta}$ $\sin(\theta)$)	3D (χ, $\cos\theta$, $\sin\theta$)	1D (torque)	Dense	Pole upright angle, small control effort, low angular velocity, and cart centering
acrobot swingup	4D $(\theta_{1,2}, \dot{\theta}_{1,2})$	6D ($\cos\theta_{1,2}$, $\sin\theta_{1,2}$, $\dot{\theta}_{1,2}$)	2D Joint torques	Dense	End effector in upright position with tolerance
walker run	$\sim$ 18D ($\theta_{1:7}$, $\dot{\theta}_{1:7}$, top velocities v_x, v_z, top coord. z_{top}, torso coord. x_{torso})	24D same as state, joint angles replaced by cos and sin	6D (joint torques)	Dense	Positive signal for forward velocity, negative signal for colliding with ground and control cost

Table 6.1: State, observation, and action spaces along with reward specifications for selected `dm_control` tasks.

t=0.0 t=2.0 t=4.0

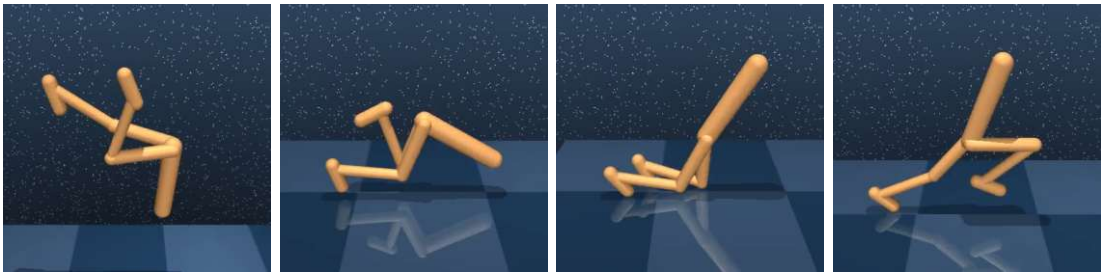
(a) Swingup phase of an episode of acrobot-swingup solved with $\text{CfC}^{\text{ZP}}_{\text{det}}$.



t=8.0 t=10.0 t=12.0

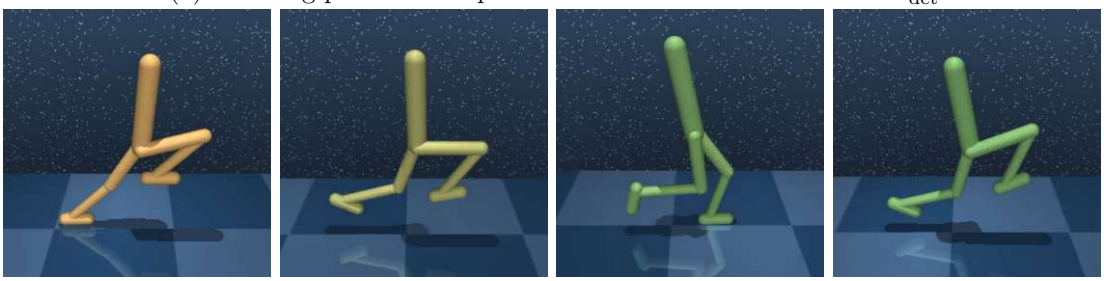
(b) Balance phase of an episode of acrobot-swingup solved with $\text{CfC}^{\text{ZP}}_{\text{det}}$.

Figure 6.1: Frames captured from an episode of the acrobot-swingup simulation.



t=0.5 t=1.0 t=1.5 t=2.0

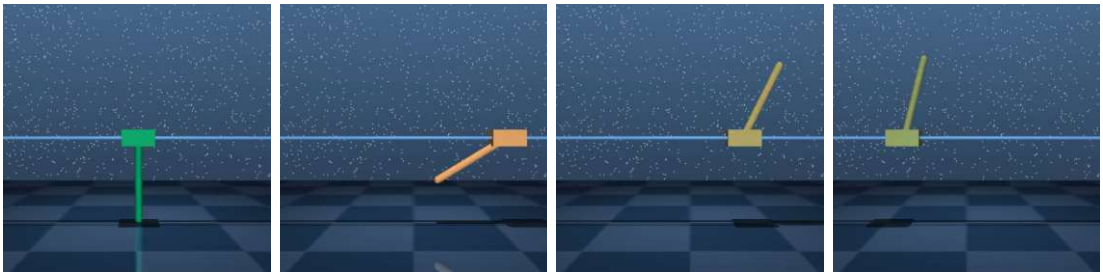
(a) Standing phase of an episode of walker-run solved with $\text{CfC}^{\text{ZP}}_{\text{det}}$.



t=2.5 t=3.0 t=7.0 t=7.5

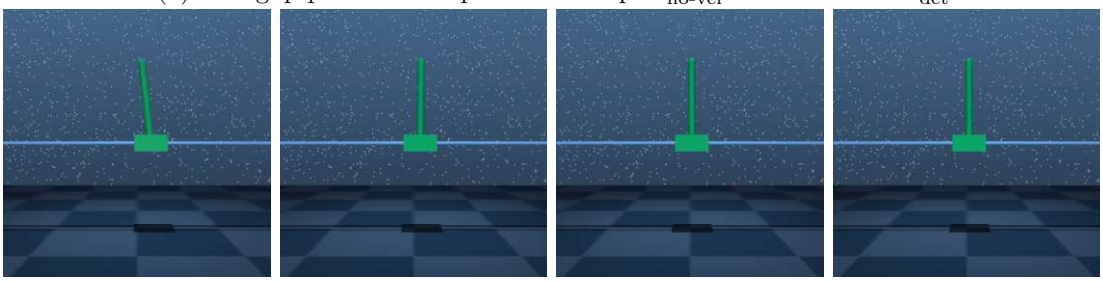
(b) Running phase (alternate single-support/flight phases) of an episode of walker-run solved with $\text{CfC}^{\text{ZP}}_{\text{det}}$.

Figure 6.2: Frames captured from an episode of the walker-run simulation.



t=0.0 t=2.0 t=4.0 t=6.0

(a) Swingup phase of an episode of $\text{cartpole}_{\text{no-vel}}$ solved with $\text{CfC}^{\text{OP}}_{\text{det}}$.



t=8.0 t=10.0 t=12.0 t=14.0

(b) Balance phase of an episode of $\text{cartpole}_{\text{no-vel}}$ solved with $\text{CfC}^{\text{OP}}_{\text{det}}$.

Figure 6.3: Frames captured from an episode of the $\text{cartpole}_{\text{no-vel}}$ simulation.

List of Figures

List of Tables

List of Algorithms

Glossary

AIS Approximate Information State Generator. 54, 60, 62, 63, 65, 69, 70

ANN Artificial Neural Network. 9, 34

BPTT Backpropagation Through Time. 18, 21, 22, 64, 65

CEM Cross-Entropy Method. 50

CfC Closed-form Continuous-time Neural Network. 2, 3, 21, 22, 62, 63, 79, 113

DDPG Deep Deterministic Policy Gradient. 49

DP Dynamic Programming. 26, 28, 30, 38, 39

DRL Deep Reinforcement Learning. 2

ELBO Evidence Lower Bound Objective. 51

EMA Exponential Moving Average. 35, 61

FNN Feedforward Neural Network. 9–11, 17–19

GPI General Policy Iteration. 40, 44

HMM Hidden Markov Model. 23, 24

i.i.d. independent and identically distributed. 13, 16, 34, 36, 65, 66, 81

KL Kullback–Leibler. 6, 16, 43

LNN Liquid Neural Network. 2, 54, 62, 63, 65, 70, 71, 73, 78, 80–82, 91, 93, 98, 103, 104

LSTM Long-Short-Term-Memory. 18

LTC Liquid Time-Constant Neural Network. 2, 3, 20, 21, 62, 63, 100

MBRL Model-Based Reinforcement Learning. 41, 49, 50, 62, 93, 103

MC Monte Carlo. 30, 32, 33, 37, 38, 42

MCTS Monte Carlo Tree Search. 40

MDP Markov Decision Process. 23–26, 29, 32, 34, 37, 38, 42–44, 46, 47, 50, 54–56, 59, 60, 62, 69, 73, 74, 78, 80, 91, 113

MFRL Model-Free Reinforcement Learning. 41

MLP Multilayer Perceptron. 10, 11, 15, 59, 70

MPC Model Predictive Control. 41, 50

MPPI Model Predictive Path Integral Control. 42, 50, 58, 59, 68, 104

MSE Mean Squared Error. 16, 91

NLL Negative Log Likelihood. 17, 98, 99

NN Neural Network. 16, 19

ODE Ordinary Differential Equation. 19–22, 51

pdf Probability Density Function. 37

PlaNet Planning in Latent Space. 50

POMDP Partially Observable Markov Decision Process. 2, 24, 42–46, 49, 51, 53, 54, 59, 62, 63, 73, 105

RL Reinforcement Learning. 3, 22, 23, 28, 37–39, 42, 45, 46, 49, 54, 63, 76, 77

rng Random Number Generator. 93, 101

RNN Recurrent Neural Network. 17–19, 21, 44, 45, 51, 62–66, 113

RSSM Recurrent State-Space Model. 50, 51

SAC Soft Actor Critic. 51

SDMDP Stochastic Delay Markov Decision Process. 63, 73, 77, 104

SGD Stochastic Gradient Descent. 13, 34, 35, 64

T-BPTT Truncated Backpropagation Through Time. 51, 65, 70

TD Temporal Difference. 30–34, 37

Bibliography

[AHL16] David Abel, David Hershkowitz, and Michael Littman. Near optimal behavior via approximate state abstraction. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2915–2923, New York, New York, USA, 20–22 Jun 2016. PMLR.

[AKLM21] Alekh Agarwal, Sham M. Kakade, Jason D. Lee, and Gaurav Mahajan. On the theory of policy gradient methods: Optimality, approximation, and distribution shift. *Journal of Machine Learning Research*, 22(98):1–76, 2021.

[AMKL] Kavosh Asadi, Dipendra Misra, Seungchan Kim, and Michel L. Littman. Combating the compounding-error problem with a multi-step model.

[Bar21] Andrew G Barto. Reinforcement learning: An introduction. by richard's sutton. *SIAM Rev*, 6(2):423, 2021.

[BCC57] R. Bellman, Rand Corporation, and Karreman Mathematics Research Collection. *Dynamic Programming*. Rand Corporation research study. Princeton University Press, 1957.

[BGC+17] Yoshua Bengio, Ian Goodfellow, Aaron Courville, et al. *Deep learning*, volume 1. MIT press Cambridge, MA, USA, 2017.

[BSA83] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983. Conference Name: IEEE Transactions on Systems, Man, and Cybernetics.

[CB24] George Casella and Roger Berger. *Statistical inference*. CRC press, 2024.

[Chi] Cheng Chi. NODEC: Neural ODE for optimal control of unknown dynamical systems. version: 1.

[CK11] Imre Csiszár and János Körner. *Information theory: coding theorems for discrete memoryless systems*. Cambridge University Press, 2011.

[Cou02] Rémi Coulom. *Reinforcement learning using neural networks, with applications to motor control*. PhD thesis, Institut National Polytechnique de Grenoble-INPG, 2002.

[CPP09] Pablo Samuel Castro, Prakash Panangaden, and Doina Precup. Equivalence relations in fully and partially observable markov decision processes. In *IJCAI*, volume 9, pages 1653–1658, 2009.

[CRBD19] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations, 2019.

[Cyb89] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[DDS+] Andreas Doerr, Christian Daniel, Martin Schiegg, Duy Nguyen-Tuong, Stefan Schaal, Marc Toussaint, and Sebastian Trimpe. Probabilistic recurrent state-space models.

[DR11] Marc Deisenroth and Carl E Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pages 465–472, 2011.

[Elm90] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[FHM18] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

[FLRP+19] Vincent Francois-Lavet, Guillaume Rabusseau, Joelle Pineau, Damien Ernst, and Raphael Fonteneau. On overfitting and asymptotic bias in batch reinforcement learning with partial observability. *Journal of Artificial Intelligence Research*, 65:1–30, 2019.

[Fou25] Farama Foundation. Gymnasium documentation v1.1.1, 2025.

[GDG03] Robert Givan, Thomas Dean, and Matthew Greig. Equivalence notions and model minimization in markov decision processes. *Artificial intelligence*, 147(1-2):163–223, 2003.

[GG] Guillaume Garrigos and Robert M. Gower. Handbook of convergence theorems for (stochastic) gradient methods.

[GK17] Surbhi Gupta and Amod Kumar. A brief review of dynamics and control of underactuated biped robots. *Advanced Robotics*, 31(12):607–623, 2017.

[Has20] Ramin Hasani. *Interpretable recurrent neural networks in continuous-time control environments*. PhD thesis, Technische Universität Wien, 2020.

[HDT19] Dongqi Han, Kenji Doya, and Jun Tani. Variational recurrent models for solving partially observable control tasks, 2019.

[HLA+21] Ramin Hasani, Mathias Lechner, Alexander Amini, Daniela Rus, and Radu Grosu. Liquid time-constant networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(9):7657–7666, May 2021.

[HLA+22] Ramin Hasani, Mathias Lechner, Alexander Amini, Lucas Liebenwein, Aaron Ray, Max Tschaikowski, Gerald Teschl, and Daniela Rus. Closed-form continuous-time neural networks. *Nature Machine Intelligence*, 4(11):992–1003, 2022. Publisher: Nature Publishing Group.

[HLBN] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination.

[HLNB] Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models.

[HM24] John Healy and Leland McInnes. Uniform manifold approximation and projection. *Nature Reviews Methods Primers*, 4(1):82, 2024.

[HMa] Nicklas Hansen, Vincent Moens, and asmith26. Td-mpc2. `https://github.com/nicklashansen/tdmpc2/tree/main`.

[How60] Ronald A Howard. Dynamic programming and markov processes. 1960.

[HPBL] Danijar Hafner, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. Mastering diverse domains through world models.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[HS15] Matthew J. Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527, 2015.

[HS18] David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[HSW] Nicklas Hansen, Hao Su, and Xiaolong Wang. TD-MPC2: Scalable, robust world models for continuous control.

[HWS] Nicklas Hansen, Xiaolong Wang, and Hao Su. Temporal difference learning for model predictive control.

[JFZL] Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. When to trust your model: Model-based policy optimization. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

[JGP] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax.

[KB17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[KE03] K.V. Katsikopoulos and S.E. Engelbrecht. Markov decision processes with delays and asynchronous cost collection. *IEEE Transactions on Automatic Control*, 48(4):568–574, 2003. Conference Name: IEEE Transactions on Automatic Control.

[KOD+19] Steven Kapturowski, Georg Ostrovski, Will Dabney, John Quan, and Remi Munos. Recurrent experience replay in distributed reinforcement learning. In *International Conference on Learning Representations*, 2019.

[KS98] Christof Koch and Idan Segev. *Methods in neuronal modeling: from ions to networks*. MIT press, 1998.

[KS02] Michael Kearns and Satinder Singh. Near-optimal reinforcement learning in polynomial time. *Machine learning*, 49:209–232, 2002.

[Kut01] Wilhelm Kutta. *Beitrag zur näherungsweisen Integration totaler Differentialgleichungen*. Teubner, 1901.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LBE24] Gaspard Lambrechts, Adrien Bolland, and Damien Ernst. Informed pomdp: Leveraging additional information in model-based rl, 2024.

[LHA+] Mathias Lechner, Ramin Hasani, Alexander Amini, Thomas A. Henzinger, Daniela Rus, and Radu Grosu. Neural circuit policies enabling auditable autonomy. 2(10):642–652.

[Lil15] TP Lillicrap. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[LMA20] Yuan Lin, John McPhee, and Nasser L Azad. Comparison of deep reinforcement learning and model predictive control for adaptive cruise control. *IEEE Transactions on Intelligent Vehicles*, 6(2):221–231, 2020.

[LS01] Michael Littman and Richard S Sutton. Predictive representations of state. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14. MIT Press, 2001.

[LWL06] Lihong Li, Thomas J Walsh, and Michael L Littman. Towards a unified theory of state abstraction for mdps. *AI&M*, 1(2):3, 2006.

[LXL+] Fan-Ming Luo, Tian Xu, Hang Lai, Xiong-Hui Chen, Weinan Zhang, and Yang Yu. A survey on model-based reinforcement learning.

[MDA+19] Alistair Muldal, Yotam Doron, John Aslanides, Tim Harley, Tom Ward, and Siqi Liu. dm_env: A python interface for reinforcement learning environments, 2019.

[MGK] Lingheng Meng, Rob Gorbet, and Dana Kulić. Memory-based deep reinforcement learning for POMDPs.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[MP69] Marvin Minsky and Seymour Papert. An introduction to computational geometry. *Cambridge tiass., HIT*, 479(480):104, 1969.

[NES+] Tianwei Ni, Benjamin Eysenbach, Erfan Seyedsalehi, Michel Ma, Clement Gehring, Aditya Mahajan, and Pierre-Luc Bacon. Bridging state and history representations: Understanding self-predictive RL.

[PJ92] Boris T Polyak and Anatoli B Juditsky. Acceleration of stochastic approximation by averaging. *SIAM journal on control and optimization*, 30(4):838–855, 1992.

[PMB13] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. Pmlr, 2013.

[PT87] Christos H. Papadimitriou and John N. Tsitsiklis. The complexity of markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, 1987.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[Ros58] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[Rub97] Reuven Y Rubinstein. Optimization of computer simulation models with rare events. *European Journal of Operational Research*, 99(1):89–112, 1997.

[Run95] Carl Runge. Über die numerische auflösung von differentialgleichungen. *Mathematische Annalen*, 46(2):167–178, 1895.

[RV07] Olivier Roy and Martin Vetterli. The effective rank: A measure of effective dimensionality. In *2007 15th European signal processing conference*, pages 606–610. IEEE, 2007.

[Sch08] Anton Maximilian Schäfer. *Reinforcement learning with recurrent neural networks*. PhD thesis, Osnabrück, Univ., Diss., 2008, 2008.

[Spo02] Mark W Spong. The swing up control problem for the acrobot. *IEEE control systems magazine*, 15(1):49–55, 2002.

[SR07] Dave Shreiner and Randi J Rost. *OpenGL (R) Library*. Addison-Wesley Professional, 2007.

[SRD+20] Ramanan Sekar, Oleh Rybkin, Kostas Daniilidis, Pieter Abbeel, Danijar Hafner, and Deepak Pathak. Planning to explore via self-supervised world models. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 8583–8592. PMLR, 13–18 Jul 2020.

[SSSM22] Jayakumar Subramanian, Amit Sinha, Raihan Seraj, and Aditya Mahajan. Approximate information state for approximate planning and reinforcement learning in partially observed systems. *Journal of Machine Learning Research*, 23(12):1–83, 2022.

[Sut91] Richard S Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *ACM Sigart Bulletin*, 2(4):160–163, 1991.

[TET12] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[TKT+24] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.

[Tod14] Emanuel Todorov. Convex and analytically-invertible dynamics with contacts and constraints: Theory and implementation in mujoco. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6054–6061. IEEE, 2014.

[TTM+] Yuval Tassa, Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Piotr Trochim, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, and Nicolas Heess. dm_control: Software and tasks for continuous control.

[WAT] Grady Williams, Andrew Aldrich, and Evangelos Theodorou. Model predictive path integral control using covariance variable importance sampling.

[WNLL09] Thomas J Walsh, Ali Nouri, Lihong Li, and Michael L Littman. Learning and planning in environments with delayed feedback. *Autonomous Agents and Multi-Agent Systems*, 18:83–105, 2009.

[WZW+23] Dan Wang, Wanfu Zheng, Zhe Wang, Yaran Wang, Xiufeng Pang, and Wei Wang. Comparison of reinforcement learning and model predictive control for building energy system optimization. *Applied Thermal Engineering*, 228:120430, 2023.

[YHL21] Cagatay Yildiz, Markus Heinonen, and Harri Lähdesmäki. Continuous-time model-based reinforcement learning. In *Proceedings of the 38th International Conference on Machine Learning*, pages 12009–12018. PMLR, 2021. ISSN: 2640-3498.

[ZHS09] Jinming Zou, Yi Han, and Sung-Sau So. Overview of artificial neural networks. In David J. Livingstone, editor, *Artificial Neural Networks: Methods and Applications*, pages 14–22. Humana Press, 2009.

[ZLP+21] Amy Zhang, Zachary C. Lipton, Luis Pineda, Kamyar Azizzadenesheli, Anima Anandkumar, Laurent Itti, Joelle Pineau, and Tommaso Furlanello. Learning causal state representations of partially observable environments, 2021.

[ZMB+08] Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, Anind K Dey, et al. Maximum entropy inverse reinforcement learning. In *Aaai*, volume 8, pages 1433–1438. Chicago, IL, USA, 2008.

[ZMC+21] Amy Zhang, Rowan McAllister, Roberto Calandra, Yarin Gal, and Sergey Levine. Learning invariant representations for reinforcement learning without reconstruction, 2021.

[ZZH+23] Xuanle Zhao, Duzhen Zhang, Liyuan Han, Tielin Zhang, and Bo Xu. Ode-based recurrent model-free reinforcement learning for pomdps, 2023.