Evolutionary Planning on a Learned World Model

supervised by Sebastian Risi

Thor V.A.N. Olesen
Dennis Thinh Tan Nguyen

A thesis presented for the degree of
Master of Science, Computer Science

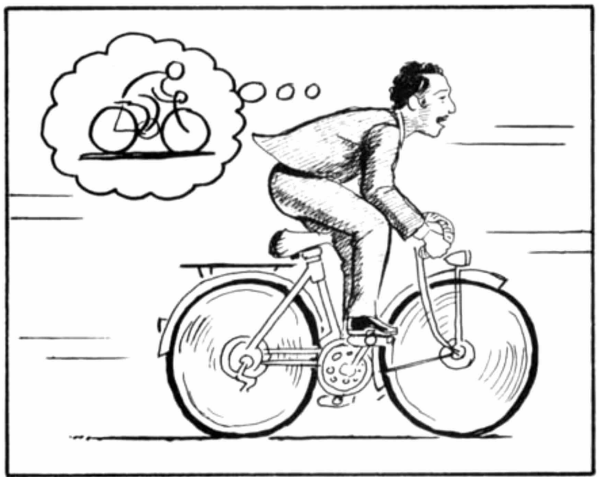


Figure 1: World Model, Scott McCloud's Comics (McCloud 1993).

Department of Computer Science
IT University of Copenhagen
Denmark
June 2020

Abstract

Artificial Intelligence (AI) and Machine Learning (ML) have shown an upward trend of growth in the 21st century. The journey to understand if machines can truly think and demonstrate some notion of "intelligence" has been a fascination in human society early on. The term "artificial intelligence" was coined in 1956 and is the science of creating machines that behave in a way considered intelligent, if it was a human being. However, expectations to AI seem to always outpace reality. Arguably, what most people think of as "true AI" has not yet been experienced, and most recent advances in AI are due to new hardware. This does not signify progress toward "general AI" and we find the hysteria to prevent any impactful progress from being made. Today is the age of implementation, and we would like to bring back the field to its age of discovery. In this pursuit, we present a small contribution to the discovery of AI.

The goal of this work is to use evolution to do planning on a learned model of the world. This is similar to how humans develop a mental model on the world based on what they can perceive with their senses. Given this internal model, humans are able to make decisions. We hope to show that machines can learn a similar model of the world to enable planning by using the model to simulate experiences and make good decisions. This is useful in real-world domains without access to a simulator, which means learning must be done using data from the real system. Recent progress has been made with model-free reinforcement learning where vast amounts of data are used to find a policy. However, this is not always possible or tractable in complex environments, which is why we direct our attention to sample efficient model-based reinforcement learning methods.

We focus our attention on the single-player, real-time "CarRacing" video game by Open AI. We use a Convolutional Variational Auto Encoder (ConvVAE) to learn a representation of the world and a Mixture Density Recurrent Neural Network (MDRNN) to learn a dynamics model of the world. The model is used as a Forward Model (FM) to simulate continuous actions with a Rolling Horizon Evolutionary Algorithm (RHEA), which encodes and evolves individuals as sequences of actions (EA) and repeats planning each time step until a limited look-ahead (RH) to obtain a good driving policy. To the best of our knowledge, this work is novel, since we show how to combine a Rolling Horizon Evolutionary Algorithm (RHEA) with a learned dynamics model (MDRNN) to do online policy-space planning with continuous actions in latent space (VAE).

Keywords: model-based reinforcement learning, mixture density recurrent neural network (MDRNN), convolutional variational auto encoder (ConvVAE), planning, rolling horizon evolutionary algorithm (RHEA)

Flow diagram of evolutionary planning agent

Figure 2 shows the flow diagram of our Agent model. The raw car racing observation is first compressed by a ConvVAE at each time step t to produce z_t. The latent vector z_t and the hidden state h_t of the MDRNN dynamics model is fed as input to the RHEA planning agent at each time step. The agent uses the dynamics model to do planning by simulating experiences in latent space and outputs a policy as an action vector a_t for motor control. The dynamics model takes the current z_t and action a_t as input to update its own hidden state and produce h_{t+1} used at time $t+1$.

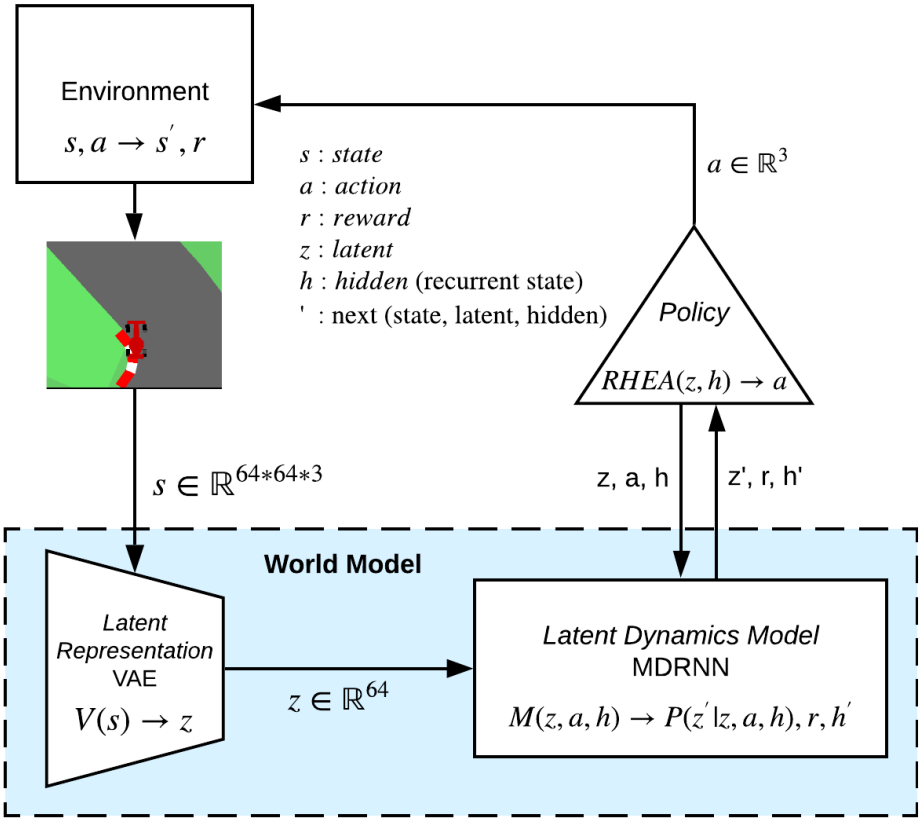


Figure 2: Simulated Online Evolutionary Planning (RHEA) in Latent Space (VAE) using a Learned Model (MDRNN) of the Car Racing game

Acknowledgements

We want to thank our supervisor Sebastian Risi for his valuable and constructive suggestions during the planning and development of this thesis research work. We have enjoyed our weekly meetings and his insightful advice as a pioneer in the field, especially during times of desperation. We would also like to thank Rasmus Berg Palm for his immense help in getting us started on the planning part of the thesis and his willingness to have feedback meetings to help us moving on the right path. Finally, we would like to thank Lars Reiter Nielsen and Jonas Hartmann Andersen for constructive criticism on the written work.

I (Thor) would like to thank Pieter Abbeel and Dan Klein, who taught my first introductory course in Artificial Intelligence and sparked my interest in exploring the technical and mathematical intricacies in Artificial Intelligence and Machine Learning. Further, I would like to thank my thesis partner Dennis for being patient, supportive, and a great friend. Finally, I would like to thank my family and Alexandra for all their love, support, and encouragement throughout my studies.

I (Dennis) would like to thank my thesis partner Thor Olesen for being supportive, inspiring, and a great friend throughout my years at ITU. He has pushed my academic limits in many ways, which has encouraged me to stay curious and explorative about the unknown. Further, I would like to thank my parents, sister, and Isabella for all their continuous love, support, and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The question of whether a computer can think is no more interesting than the question of whether a submarine can swim. - Edsger W. Dijkstra

The wish to start our studies at the IT University of Copenhagen was grounded in a deep desire to understand computers, programming and how it is possible to build anything in the virtual realm. Led by an intense curiosity, we set out to understand how it is possible to solve so many different problems in society through the use of machinery, computation and thinking. This has led to a new way of life: imagining something and bringing it alive as a manifestation in the real world through countless hours of thinking, programming and drinking coffee.

At some point, you learn enough about the machine and its mechanics that you no longer consider yourself a layman and start to question whether a machine can genuinely think for itself. As Dijkstra seems to put it, whether computers think or submarines swim, is entirely dependent on the definitions of "thinking" and "swimming, " which are ambiguous. Having demystified some parts of Computer Science, you realize that people often debate over the field of AI and ML but never settle to agree on a definition of "thinking", what AI is and when something is truly "intelligent".

As Computer Scientists, we realize that computers will most likely never "think" in the same way as a human does. Analogously, it makes no sense to discuss whether a submarine swims or not. Seemingly, a submarine does not swim but that does not make it an interesting topic of debate, since that is just a definition. Regardless, submarines are able to move through water just like fish and we can compare their methods, limitations and how well they each do it.

Similarly, we can still try to compare whether or not methods in AI can solve different human tasks, regardless of whether they actually exhibit any notion of "true intelligence". In our case, we will try to use a set of methods in the field of AI to solve the continuous control task of driving in a real-time car racing game.

1.1 AI in Games

The field of *Artificial Intelligence* (AI) in games is concerned with research in methods that produce 'intelligent' behavior in games. From our standpoint, the central problem in AI is the creation of a rational agent that, given some goals, tries to perform a series of actions that yield the best-expected outcome. This thesis explores whether AI planning algorithms can be used in a simulated model of a car racing game to enact well-behaved driving. For this purpose, a brief overview of Artificial Intelligence in games is presented. In the rest of the thesis, the terms "game AI" and "AI" will refer to the same research field. In this thesis, the focus will only be on methods used to control an agent to play a game (*Game as AI benchmarks*).

Games have always been used in AI research progress as a self-contained environment for testing ideas, solving complex problems and getting valuable feedback before applying them to real-world problems (e.g., medical diagnosis, robotics and finance). AI in games has been used for a long time and most "research in the field is concerned with constructing agents for playing games, with or without a learning component. Early pioneers in the field used game-playing programs to test whether computers could solve tasks that seemed to require "intelligence" (Yannakakis and Togelius 2018).

Initially, game AI focused on classic board games (Chess, Checkers), which seemed to challenge humans by their immense complexity despite the simple rules and were heavily based on mental models of the game. Alan Tuning used the Minimax algorithm to play Chess. In 1952, A.S. Douglas made a program that was able to play Tic-Tac-Toe. Later, Arthur Samuel invented the first form of *reinforcement learning* to learn to play Checkers by *self-play*. In 1992, Gerald Tesauro developed the TD-Gammon software that used an artificial neural network trained via temporal difference learning by playing backgammon against itself. After this, one of the most significant breakthroughs in the field of AI was IBM's Deep Blue program, which used the Minimax algorithm, Chess domain knowledge and a tuned board *evaluation function* to beat the world champion of chess, Garry Kasparov.

The latest major milestone happened in 2016 in the game of Go. The game has been a benchmark for game AI with a branching factor that approximates 250 and a much more extensive search space than Chess. The *branching factor* is the number of available actions in a turn and measures the complexity of a game. The *search space* denotes the set of possible *states* (the agent configuration within its environment) in the game. In 2016, Google DeepMind's AlphaGo used *Deep Reinforcement Learning* to beat Lee Sedol, one of the best Go players in the world. And in 2017, AlphaGo won against the world's best player, Ke Jie. These board games are known to have discrete turn-based mechanics and full board visibility.

Today, a lot of research in the field focuses on developing AI for other games such as video games. In 2014, Google DeepMind used model-free Deep Reinforcement Learning to play several classic Atari video games on a super-human level just from raw pixel inputs using a Deep Convolutional Neural Network trained with Q-learning (DQN) directly from pixels (Mnih et al. 2014). Most recently, the DeepMind team demonstrated how to use Monte Carlo Tree Search with a learned model to achieve super-human level in a range of challenging and visually complex Atari domains, without prior knowledge of the game dynamics and domain knowledge.

For this purpose, model-based Reinforcement Learning (RL) was used to learn a model of the environment dynamics and then do planning based on the learned model (Schrittwieser et al. 2020). As they say, "constructing agents with planning capabilities has long been one of the main challenges in the pursuit of artificial intelligence". One of the major applications of this approach is in real-world problems where a perfect simular is not available and the dynamics of the environment are too complex and unknown to navigate.

1.2 Real-Time Racing Games

There are many types of games that can be categorized accordingly:

- Action space: discrete or continuous actions
- Outcome: deterministic or stochastic (probabilistic)
- Player type: single-player or multi-player
- Environment: fully or partially observable
- Time: real-time or turn-based

Video games are becoming more diverse and complex and provide opportunities to work with visual perception of video data similar to how humans interpret and understand the visual world through vision. Lately, AI in video games has also been used for *procedural content generation* (Yannakakis and Togelius 2018, p. 9) where AI algorithms are used to create new game content.

From a pragmatic perspective, Open AI provides a nice interface for game environments and is often used in both industry and academia as a testbed for AI methods. In our case, we have devoted our attention to the real-time Open AI car racing video game. This has been chosen, since video games have increasingly become the new domain of choice for testing and showcasing new AI algorithms.

The car racing game is a single-player, *real-time* game unlike many of the previously described games, which are *turn-based*. Most classic board games are turn-based where players alternate between taking actions. Often, the time between turns is unimportant (although tournaments may enforce time limits). On the other hand, the car racing game is a real-time game where there is variation in how often an action can be taken.

Usually, this is limited by the screen update frequency of 60 frames per second (FPS) that ensures a perceived smooth movement, which may be decreased if rendering is too demanding. In practice however, the number of actions taken per second is usually limited to less than the FPS. A given *depth* of search (i.e., number of actions) means very different things depending on the time granularity of the game (i.e., limit of how far ahead agent can look).

From a research perspective, tree-based search methods like Monte Carlo Tree Search (MCTS) have been the most popular defacto choice for planning in games. Still, recent research has shown that rolling horizon evolutionary algorithms (RHEA) are a viable and competitive alternative to MCTS, since they work seamlessly on continuous action spaces like in video car racing games, unlike vanilla MCTS that only works with discrete action spaces. Being a real-time game, agents have limited time to respond to environmental signals. Thus, doing planning requires access to an efficient generative model that can be used to simulate driving in the car racing game with an efficient search method. For this purpose, it seems that little research has been done on verifying whether evolution-based search approaches can be used as a viable alternative to do planning on a learned model in complex video games with continuous action spaces.

1.3 Research Question

In the previous section, we presented a historical overview of the breakthroughs in AI history. We argued why video games are an under-explored type of game for doing evolutionary planning in the research field. The *car racing* game was chosen as the benchmark for this thesis and the goal is to explore AI methods for this game and examine the achieved playing score compared to other AI methods. The following research question was made based on these observations:

Research Question:

How can we design a game AI agent that can drive complete tracks in the car racing environment by using a learned model to do evolutionary online planning that copes seamlessly with the continuous action space?

Chapter 2

Background

This chapter defines and explains the theoretical concepts needed to understand the work of this thesis, which includes an understanding of artificial intelligence, machine learning, model-based reinforcement learning, evolutionary planning algorithms, neural networks and their architectures.

2.1 Artificial Intelligence (AI)

Artificial Intelligence may be defined as "the study of agents that receive percepts from the environment and perform actions accordingly". The central theme is the idea of a *rational agent*, which is something that perceives and acts in an environment to achieve the best outcome, or best-expected outcome given uncertainty (Russell and Norvig 2009, preface and p. 4). We adopt the view of intelligence concerned with *rational behavior*, meaning an *intelligent agent* takes the best possible action in a situation.

2.1.1 The World: an Agent and an Environment

Rational agents exist in an *environment*, which together creates a *world*. The environment in the world is everything that surrounds the agent but is not part of the agent. Intuitively, the environment represents situations, which the agent is able to perceive and act upon.

2.1.2 State Spaces and Search Problems

To represent a rational planning agent, we need to mathematically express the environment in which the agent lives, which is formally expressed as a *search problem*. The agent's current situation is called a *state*, which represents the configuration within the environment that the agent currently perceives. Given this current state, the agent wishes to know how it can arrive at a new state by acting in a way that satisfies its goals in the best possible way. Thus, a search problem is solved by finding a path from a start state to goal state and requires four components:

- A **state space**: set of possible states (situations) in the given world
- A **successor function**: takes a state-action pair and outputs the cost of performing the action and a state
- A **start state**: the state in which an agent exists at first
- A **goal test**: a function that takes a state as input and determines whether it is a goal state

2.1.3 CarRacing Problem Formulation

The objective in the *CarRacing-v0* environment is to drive a car as fast as possible through a randomly generated two-dimensional racetrack containing grass and boundaries. Thus, well-behaved driving is defined by driving on the road while avoiding grass and staying within the boundaries. If the car has a high velocity at a corner or on the grass, it may lose traction and spin out of control. Consequently, this results in much time spent regaining control and getting back on the road, leading to a worse game score.

The game is considered solved when the agent consistently gets an average score of 900 points in 100 randomly generated tracks. The total score in the game is defined by equation 2.1 where t is the number of game ticks it takes the car to complete the track and 1000 is the maximum possible score. Every game tick, the score is -0.1 and $\frac{1000}{N}$ for every track tile visited where N is the total number of tiles in a given track.

$$score(t) = 1000 - 0.1t \tag{2.1}$$

The state space is the set of all possible frames in the car racing environment. The start state is the initial frame showing the start position of the car on the track before starting to drive. The goal test returns true if all the tiles on the road have been visited (track is completed) or the car leaves the playfield. The Open AI gym interface provides the successor 'step' function, which returns the score and successor state (frame) given the current state (frame) and an action.

State Space

In the car racing environment, each state is represented as a frame, which consists of a $96 \cdot 96 \cdot 3$ grid of 256 RGB values where 96 denotes the height and width of the screen and 3 denotes the number of image channels. The size of the state space is $256^{3 \cdot 96^2}$, which is considered huge and requires some preprocessing to reduce it to something more manageable.

Firstly, one may resize and crop the frame by reducing the resolution and removing the image borders, which can reduce the number of possible states to around $256^{3 \cdot 48 \cdot 36}$.

Secondly, one could represent each pixel with a value of 1 (road) or 0 (grass), instead of using the full RGB spectrum across three channels.

As a result, the cardinality (i.e., size) of the state space could be reduced down to $2^{48 \cdot 36}$. Regardless, the state space remains very large and doing this preprocessing poses a computational cost, which must be taken into account if frames are received and preprocessed in real time.

Action Space

In the CarRacing environment, an action is represented as a vector of three components used to control the gas, brake and steering of the car. An action is considered to be a triple $(s, g, b) \in [-1, 1] \times [0, 1] \times [0, 1]$ where steering (s) ranges from hard left (-1) to hard right (+1) and acceleration (g) and braking (b) both range from doing nothing (0) to full acceleration or braking (1).

The complexity of the game search space is defined by the branching factor of all possible actions at any state. When the domain is bigger, the branching factor increases and the search space explodes. The continuous action space is infinitely large and requires informed heuristics or discretization to overcome the combinatorial explosion.

Characteristics

The environment is *partially observable*, meaning the agent cannot perceive all aspects of the environment directly relevant to its choice of action. Namely, the agent can only observe the current frame, which shows a segment of the track, instead of the whole track. This is similar to self-driving vehicle scenarios, which deal with partial information to solve the problem of autonomous driving. If the environment was *fully observable*, the agent should be able to observe the full track, since this might be relevant to its task. Namely, it might be driving fast on a forward road segment and needs to know when a sharp turn is approaching to reduce speed in time.

By default, the environment is *deterministic*, meaning the next state (frame) of the environment is completely predictable from the current state and the action executed by the agent. Notice, if one uses a model to represent the environment, the next state might have some uncertainty associated with it coming from randomness or a lack of a good environment model. In such *stochastic* games, the successor function is not deterministic, meaning it does not map an action and state (frame) to one particular state but instead to a distribution over next future states along with the cost. This may be represented as a function $s_t, a_t \rightarrow P(s_{t+1})$, $c(s_t, a_t, s_{t+1})$ where t denotes the current time step, s_t denotes the current state, a_t denotes the current action taken, $P(s_{t+1})$ denotes a distribution over future states s_{t+1} and $c(s_t, a_t, s_{t+1})$ denotes the cost of taking action a_t from state s_t to successor state s_{t+1}.

The environment is *sequential*, meaning current and past decisions affect future decisions. Namely, the way the agent drives now or drove earlier will affect the consequences of its future driving. For example, the agent may be driving very fast in the past, resulting in a high current speed, which may ultimately prevent the agent from completing a sharp turn. Thus, the agent needs to consider the temporal dependency of the vehicle's speed when making decisions. On the other hand, an AI that does some medical image analysis may decide if a person is sick from a single image, which means it is an *episodic* environment, as one image is independent to the next. In car racing, the sequence of frames are highly correlated and exhibit a temporal aspect, in which future driving actions depend on previous actions.

Further, it is a *single-agent* environment, since only one agent is present to drive a car, unlike systems with multiple agents interacting with each other. Thus, the environment is not a *competitive* or *cooperative multi-agent* environment, in which the behaviors of others matter. The environment is *static*, meaning it does not change while the agent is "thinking". Finally, the state space is *discrete* (very large but finite space) and the action space is *continuous*.

2.2 Machine Learning (ML)

Historically, AI was mainly dominated by a symbolic (explicit) and logical approach between the 1950s and 1980s where knowledge-based expert systems became popular (Russell and Norvig 2009, p. 17-24). These models encoded human domain knowledge explicitly as symbolic facts (predicates) in a "knowledge base" database of rules. The system would then solve human problems using AI methods such as logic to reason from database facts and search to find solutions to problems. *Symbolic AI* are methods that use such an explicit representation of a problem with formulas or rules. In contrast, *subsymbolic AI* (machine learning - neural networks) are methods that use an implicit representation learned from experience without rules, mainly inspired by the brain (Bolander 2019).

Machine learning is a subbranch of AI that investigates how algorithms can improve automatically through experience or data. Arthur Samuel coined the term in 1959 as "the field of study that gives computers the ability to learn without being explicitly programmed". We prefer the more nuanced formal definition by Tom Mitchell: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E" (Mitchell 1997, p. 2). Machine learning (also known as pattern recognition) grew into its own field, shifting focus away from symbolic approaches toward statistical and probability theory methods with a focus on prediction and finding common patterns.

2.2.1 Supervised Learning

The canonical example of a machine learning problem is that of handwritten digit recognition. We are given a 28 by 28 greyscale image represented by an input or feature vector $\boldsymbol{x} \in \mathbb{R}^{28\times 28}$ with $d = 28{\cdot}28$ dimensions and need to *predict* the digit it represents called the *target*, output or label (vector) $t \in \{0, 1, \ldots, 9\}$. The input feature vector and output target constitute a single *example* and usually we are given n such examples that constitute a *training set*:

$$\mathcal{D} = \{(\boldsymbol{x_1}, t_1), \ldots, (\boldsymbol{x_n}, t_n)\} \tag{2.2}$$

The goal is to find a *model* $f(\boldsymbol{x}, \boldsymbol{\theta})$ described by k parameters $\boldsymbol{\theta} \in \mathbb{R}^k$, which is an idealized mathematical representation of a real-world phenomena. To do this, we need to *learn* or *train* such model, which means tuning (i.e., changing) the parameters $\boldsymbol{\theta}$ such that $f_\theta(\boldsymbol{x}) = t$. This requires a *loss function* $L(t_n, f(\boldsymbol{x_n}, \boldsymbol{\theta}))$, which quantifies the difference between the actual target and model predictions. The *error* is then the total loss:

$$E(\theta) = \sum_n L(t_n, f(x_n, \theta)) \tag{2.3}$$

We use the error to guide in which direction the parameters should be tuned to minimize the difference between model predictions and targets:

$$\theta^* = argmin_\theta E(\theta) \tag{2.4}$$

The parameters are typically updated using an *optimization procedure* such as *gradient descent*, which minimizes the average error across all examples by iteratively moving in the direction of the steepest descent as defined by the negative of the gradient in *Calculus* (study of change), meaning the direction in which the difference between the actual target and model predictions is reduced maximally across the whole training set:

$$\theta_{t+1} \leftarrow \theta_t - \eta \frac{1}{n} \sum_{i=1}^{n} \frac{d}{d\theta_t} L(t_i, f(\boldsymbol{x_i})) \tag{2.5}$$

Notice we just use the derivative if there is only one parameter. Otherwise, given multiple parameters, we reduce the difference between the actual target and model predictions across all k parameter dimensions by repeating this process for a gradient vector of k partial derivatives:

$$\boldsymbol{\nabla f} = [\frac{\partial}{\partial \theta_1} \cdots \frac{\partial}{\partial \theta_k}]^T \tag{2.6}$$

where each component $j \in [1, k]$ represents the derivative (rate of change) of the loss function with respect to parameter j:

$$\theta_{t+1}^j \leftarrow \theta_t^j - \eta \frac{1}{n} \sum_{i=1}^{n} \frac{\partial}{\partial \theta_t^j} L(t_i, f(\boldsymbol{x_i})) \tag{2.7}$$

When the model is learned, we can evaluate it on a *test set* to see how well the model is able to *generalize*, meaning its ability to correctly categorize new examples not in the training set. Similar to how we may have multiple parameters, we may also generalize the representation of our input data from a set of input vectors into a data matrix containing all n examples in the rows with d features each: $X = [x_1 \cdots x_n]^T$ where each feature vector $(x_i)^T \in \mathbb{R}^{1 \times d}$ so $X \in \mathbb{R}^{n \times d}$.

For example, we might use a linear model where the number of parameters correspond to the number of features in the input: $f_\theta(\boldsymbol{x}) = \boldsymbol{x}^T \cdot \boldsymbol{\theta} = \hat{t}$ where $x^T \in \mathbb{R}^{1 \times d}$, $\theta \in \mathbb{R}^{d \times 1}$ and $\hat{t} \in \mathbb{R}$ (one example prediction). Linear Algebra is used to model such linear sums in multiple dimensions using matrix vector notation: $f(X) = X \cdot \boldsymbol{\theta} + \boldsymbol{b}$ where $X \in \mathbb{R}^{n \times d}$, $\boldsymbol{\theta} \in \mathbb{R}^{d \times 1}$ and $\boldsymbol{b} \in \mathbb{R}^{n \times 1}$ and $\hat{\boldsymbol{t}} \in \mathbb{R}^{n \times 1}$ (example prediction vector). *Probability* may be used to measure uncertainty in the real world as part of the model when making decisions with incomplete information.

2.2.2 Categories

In general, there exist three categories of Machine Learning, including *Supervised learning*, *Unsupervised learning* and *Reinforcement learning*. Supervised learning is where you are given a data set $\mathcal{D} = \{(\boldsymbol{x_1}, t_1), \ldots, (\boldsymbol{x_n}, t_n)\}$ and learning is similar to having a teacher train you for a test where he is able to provide you the correct answers (targets).

The example we described is a *Supervised* Machine Learning problem in which we are given a training set of input-target example pairs and the task is to learn a function that match each input to an output (prediction) such that the output matches the corresponding target. This fundamentally requires a model, loss function and optimization procedure.

Unsupervised learning is where we are not given the targets (labels):

$$\mathcal{D} = \{\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}\} \tag{2.8}$$

Now, we have to uncover patterns in the data using *clustering* or *density estimation*. Clustering is the task of dividing examples into groups such that examples within a group are similar but dissimilar to examples in other groups. Density estimation is the task of estimating the distribution from which examples were drawn. *Reinforcement learning* is the task of finding actions in a situation that maximize total expected future reward.

2.2.3 Bias-Variance trade-off

Ultimately, we wish to obtain a model that generalizes well to new unseen test examples of the phenomena we try to model. This generalization ability depends on the complexity of the model, which is determined by whether the model is too complex and overfitting the noise in the training data, or too simple, making it unable to capture the underlying pattern inherent in the data.

Bias is the degree to which the model oversimplifies the relationships inherent in the data. Variance is the degree to which the model follows the patterns inherent in the data too closely. Thus, overfitting happens when the model has low bias, but a high variance and underfitting occurs when the model has a high bias but low variance. Neither models are likely to generalize well, so striking a balance between overfitting and underfitting is crucial in Machine Learning to capture the pattern and not the noise inherent in the data.

The *bias-variance trade-off* describes this tension between the prediction error introduced by the bias and variance that we can control, unlike the error coming from noise. The generalization (test or prediction) error is the sum of the bias squared, variance and the irreducible error due to noise in the data. Ideally, we want a model with low bias and low variance, but these are bipolar, so we strive for a middle ground. Thus, we experiment with different levels of complexity and choose the one that minimizes the total error to find a model that generalizes well.

The bias-variance trade-off is shown in figure 2.1 where y is the variable we are trying to predict, $\boldsymbol{x}$ is our input vector and it is assumed that

$$y = f(x) + \epsilon \tag{2.9}$$

where $\epsilon \sim N(0, \sigma_\epsilon)$ is the error assumed normally distributed with mean zero. The model $\hat{f}(x)$ is a prediction of the true target y and the mean squared error is the expectation of the squared difference between the prediction and target (Bishop 2006, eq. 3.41, p. 149):

$$\begin{aligned}
&\text{Mean Squared Error (MSE)} : L(y, \hat{f}(x)) = E[(y - \hat{f}(x))^2] \\
&\text{Bias Variance Decomposition} : (E[\hat{f}(x)] - y)^2 + E[(\hat{f}(x) - E[\hat{f}(x)])^2] + \sigma_\epsilon^2 \\
&\text{Expected total loss} = \text{bias}^2 + \text{variance} + \text{noise}
\end{aligned} \tag{2.10}$$

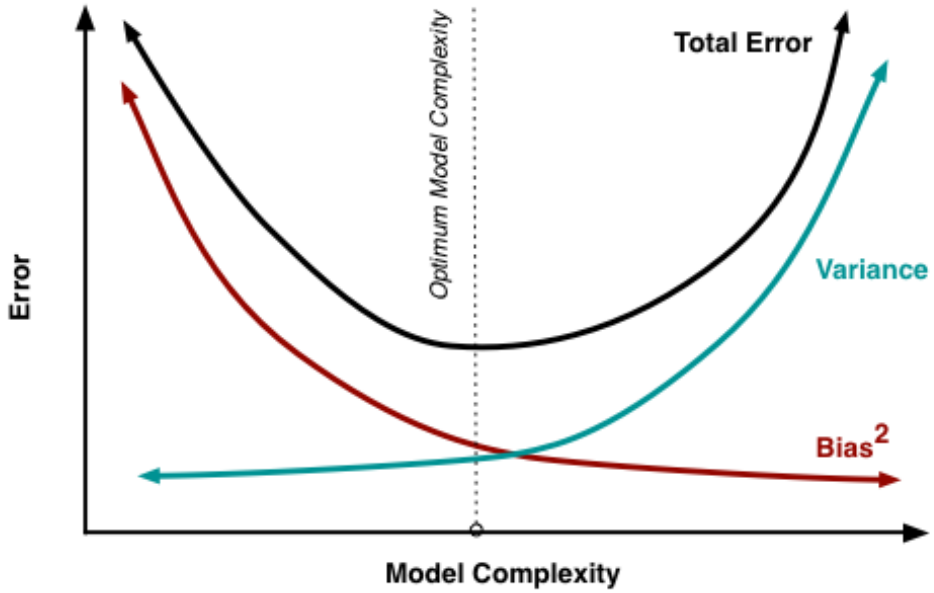


Figure 2.1: Bias and variance in total error (Fortmann-Roe 2012)

2.3 Reinforcement Learning (RL)

Previously, we described car racing as a traditional search problem where we attempt to find a path from a starting point to some goal state using a deterministic successor function that maps a state-action pair to a single successor state and the cost of taking an action in a given state.

However, there is usually a degree of *uncertainty* in the real world. Thus, we now consider a different environment setting where the successor function maps a state-action pair to a distribution over successor states and a reward. As such, the cost that is usually minimized is replaced with a reward to be maximized.

Thus, we now take into account the dynamics of the world by considering an environment with an agent whose actions are *nondeterministic*, which means there are multiple future successor states that can happen from an action taken in a given state. In games, this kind of environment setting is referred to as a *stochastic game*.

Such problems where the world is uncertain are classified as *nondeterministic search problems* and can be solved with models known as *Markov Decision Processes* (MDP) where the environment is *fully observable*.

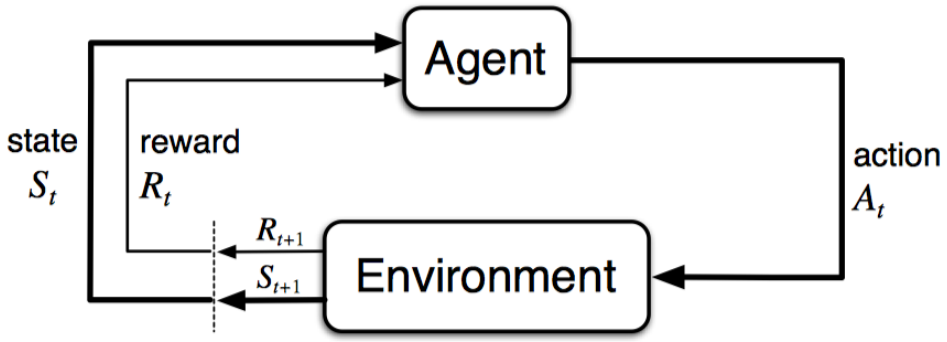


Figure 2.2: Agent-Environment Feedback Loop (Kapoor 2018)

In *Reinforcement Learning* (RL), an agent makes *observations* (perceive) and takes *actions* (act) in an *environment* (world) yielding *rewards* (feedback) that incentivize certain behavior as shown in figure 2.2. The objective is to learn a policy (action given state) to act in a way that maximize future expected reward.

Thus, "Reinforcement learning is learning what to do through *experience* - how to map situations to actions - so as to maximize a numerical reward signal". The learner is not told which actions to take, but must instead discover which actions yield the most reward by exploring the unknown transition dynamics of the environment. *Trial-and-error* search and *delayed rewards* are what defines it (Sutton and Barto 2018, p.1).

2.3.1 Markov Decision Processes

A Markov Decision Process is a model used to solve *sequential decision-making problems* in a stochastic environment represented by a tuple $\langle S, A, T, R \rangle$ where S is the set of states, A is the set of actions, T is the transition (dynamics) function and R is the reward function. Notice the set of states (S) and actions (A) are represented the same way as in traditional search problems. However, the deterministic successor function $s, a \rightarrow s', c(s, a, s')$ is now represented by a transition function $T : s, a \rightarrow P(s'|s, a)$, which models the probability of moving to state s' at time $t+1$ by taking action a in state s at time t. The cost is replaced by a separate reward function that describes the immediate scalar reward signal attained by taking an action in a given state at time t: $R : s, a \rightarrow \mathbb{R}$. An MDP also consists of a set of starting states S_0, a *horizon* T and a discount factor $\gamma \in [0, 1]$.

The discount factor models an exponential decay in the value of rewards over time. Intuitively, the discount factor helps determine how much the agent cares about rewards in the future where $\gamma = 0$ favors immediate reward and $\gamma = 1$ is indifferent to future rewards. In general, solving an MDP means finding an optimal *policy* $\pi^* : S \rightarrow p(A = a|S)$ such that the expected reward is maximized. If the MDP is *episodic*, the state is reset after each episode (game) of length T and the episode is is said to be a *trajectory* or *rollout* $E = \langle e_1, \ldots, e_T \rangle$ of *experiences* $e_1 = \langle (s_t, a_t, r_t) \rangle$ represented by the sequences of states, actions and rewards. The *return* is the total accumulated reward obtained by following a policy in the environment until the end, either determined by a fixed horizon or an infinite horizon with a discount factor of $y < 1$:

$$R = \sum_{t=0}^{T-1} \gamma^t r_{t+1} \tag{2.11}$$

The goal of RL is to find an optimal policy, π^*, which achieves the maximum expected return across all states:

$$\pi^* = argmax_\pi E[R|\pi] \tag{2.12}$$

Assuming the MDP is non-episodic, meaning it has an *infinite horizon* of $T = \infty$, $\gamma < 1$ is used as mathematical convenience to prevent an infinite sum of rewards. Unlike supervised learning in which feedback given as targets are immediately received, reinforcement learning is subject to the *credit assignment problem*, which is the problem of determining which action(s) that lead to a certain outcome. Namely, the reward feedback is often *delayed* (sparse) so it does not depend on a single decision performed in the current state but, rather, on the whole sequence of agent actions. This is why a *finite horizon* is used to place constraints on the number of time steps for which the agent can take actions and collect rewards. Hence, the agent is given a fixed number of time steps or episodes (games) as a "lifetime" to accumulate as much reward as possible before being automatically terminated.

A main assumption held by MDPs is the **Markov property**, which states that the future and past are conditionally independent, given the present:

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, a_0, s_0) = P(s_{t+1}|s_t, a_t), \tag{2.13}$$

where $s_0 \ldots s_t, a_0 \ldots a_t$ are state action sequences up to time t. These probabilities are encoded by the transition function $T(s, a, s') = P(s'|s, a)$, which represents that probability that an agent taking action $a \in A$ from a state $s \in S$ ends up in state $s' \in S$ in time $t+1$, denoted $'$ (apostrophe).

The Markov assumption is popular amongst RL algorithms but is often considered unrealistic, since it requires states to be *fully observable*. Real world environments are often partially observable and can instead be modelled with a *Partially Observable Markov Decision Process* (POMDP). A POMDP may be seen as a generalized MDP model of the environment dynamics where the agent is unable to observe everything, like in most real problems, so it has to maintain a belief (distribution over states) instead where $b(s_t) = P(s_t)$.

Thus, a POMDP can be seen as an MDP with a sensor model $P(o_t|s_t, a)$ that describes the probability of making a noisy observation o given a state-action pair. The agent receives a noisy observation o_t, instead of the unobservable hidden state s_t from the sensor *model*, which is used to update its *belief* $b(s_{t+1})$ about the next hidden state s_{t+1}. The agent then computes the new belief state $b_{t+1} = P(s_{t+1}|o_t, a_t, b_t)$ based on a current belief state b_t, action a_t and observation o_t (Kochenderfer 2014).

2.3.2 Model-Based Reinforcement Learning

"*Model-based* methods rely on planning as their primary component, while model-free methods primarily rely on learning a policy through trial-and-error directly from experience" (Sutton and Barto 2018, p. 159). Refer to appendix A.1.2 for more details on model-free RL. It may be time-consuming to learn a policy directly from experience, since converging to an optimal policy requires sufficient exploration of the state space.

This may take a long time when using deep neural networks as non-linear function approximators of state values (i.e., quality of a state). *Model-based* Reinforcement Learning solves RL problems by learning a model directly from experience and use that model to enable planning to construct a policy (plan or action). *Planning is then any method (usually a special type of search) that uses a model in a simulated environment to produce or improve a policy used to select actions.* Commonly, a different representation of the state space is used, which contains information more relevant to the task at hand.

Traditional search algorithms employ *state-space planning*, which is a search through the state space for an optimal policy or an optimal path to a goal. Actions cause transitions from state to state and value functions are computed over states. One may also do *plan-space planning*, in which you search through the space of plans (Sutton and Barto 2018, p. 160).

"A model of the environment is anything that an agent can use to predict how the environment will respond to its actions" (Sutton and Barto 2018, p. 159). Thus, the model $M = \langle T_\theta, R_\theta \rangle$ is a representation of an MDP $\langle S, A, T, R \rangle$ parameterized by θ that estimates the transition and reward function. The estimation is usually done using a nonlinear function approximator, such as a deep neural network that can recognize patterns inherent in a complex environment. The state-space S and action space A are assumed to be known.

Planning is then using the model to do lookahead of what value or policy (action) to select in an environment based on the return in equation 2.11. Thus, real interactions are replaced by *simulated* interactions in a simulated environment (the model). Planning is all done internally without taking steps in the real environment, which allows us to think.

Model-based RL is an iterative procedure where the agent *acts* in an environment that generates experiences *perceived* by the agent. These experiences are used to *learn* a model of the environment. The model is then used to enable *planning* (thinking) and constructing or improving the value or policy function before acting again. Thus, we learn a model and use planning to solve the MDP, instead of using dynamic programming to figure out the optimal value or policy function. Each time we get new experiences, we may update our estimates of the MDP represented by our model $M_\theta = \langle T_\theta, R_\theta \rangle$ shown in figure 2.3.

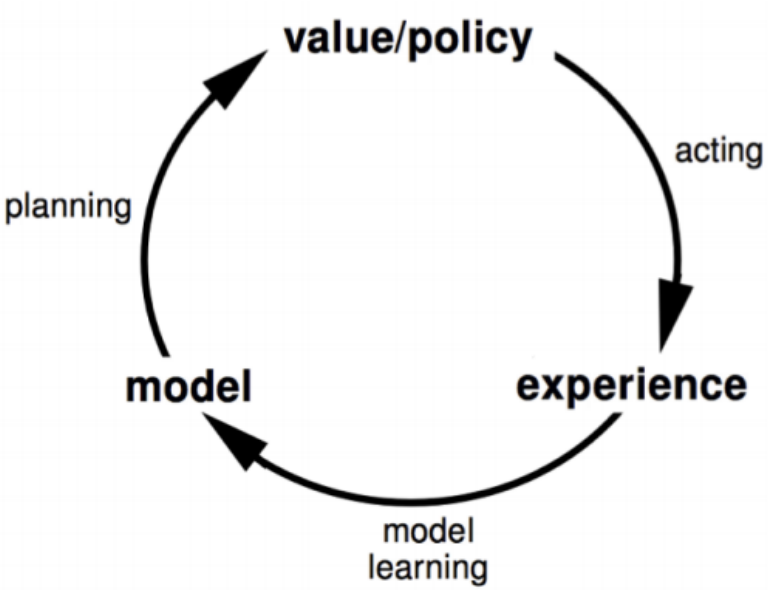


Figure 2.3: Model-Based RL Loop (Silver 2020)

Model-based vs Model-free RL

Fundamentally, both model-based and model-free RL uses the computation of value functions to estimate how good states in the environment are to guide the policy search (see A.1). However, learning a value or policy function may be hard. For example, chess has a sharp value function, since a particular move can suddenly change the odds of winning. In chess, the value function seeks to learn about states where its not a terminal position. This, requires a true understanding of the game mechanics to quantify how likely you are of winning. Yet, the reward function is an easy learning problem, since the reward is just 0 unless checkmate, so one simply needs to classify positions as checkmate or not. In chess, the model is the rules used to do lookahead to estimate the value function (based on the reward signal).

Instead of estimating the value function by learning, this may be done by planning using tree search. Tree search is powerful in games due to its ability to construct a more accurate value function when it might otherwise be hard to estimate it directly from experience. In this regard, sometimes a model provides a more compact and useful representation about the information of the environment, rather than the policy or value function itself.

Planning sometimes helps because trying to learn a value function directly is hard in a complex environment like Go, Chess, or video games (e.g., car racing). Intuitively, planning provides the means to roll forward in the state space to see what might happen. If the rules of the game are not provided, the agent must figure out those rules first and then plan accordingly. Alternatively, one may disregard the RL problem and instead consider the planning problem where the model tells the rules (MDP) used to figure out an optimal move. Thus, the motivation of using model-based RL, besides sample efficiency, is when modelling complex MDPs where it may be hard to estimate the value function and do well without planning.

Pros and Cons

The advantages of model-based LR are twofold. Firstly, one can efficiently learn a model by supervised learning (see 2.2.1). This is like having a teacher tell a pupil the right thing; given a state and an action, what is the target next state and reward. Thus, it is not necessary know the state after a full trajectory (simulation) of steps but only one step. Secondly, one can use the model to reason about uncertainty. In general, we want to take actions that make us understand the world better. Hence, it is not always ideal to take actions that maximize reward from the current state (i.e., view of the world). Namely, ones understanding of the world might be wrong. Ideally, the model tells everything required to know and the uncertainty inherent to the model can be used to encourage exploration. The downside is that there are two main sources of approximation error from the rewards and the dynamics (next state predictions given state-action pair). This means planning will only be as good as the model.

Model Learning

The goal is to estimate a model M_θ from experience $\{S_1, A_1, R_2, \ldots, S_T\}$, which is a supervised learning problem:

$$\begin{aligned} S_1, A_1 &\to R_2, S_2 \\ S_2, A_2 &\to R_3, S_3 \\ &\vdots \\ S_{T-1}, A_{T-1} &\to R_T, S_T \end{aligned}$$

Learning the reward function $R : s, a \to r$ is a regression problem where we can use Mean Squared Error to quantify the difference between predicted rewards of the model and actual rewards in the real environment. Learning the dynamics function $T : s, a \to P(s'|s, a)$ is a *density estimation* problem, which might use Kl divergence loss to minimize the difference between the true and predicted distribution over next states. The goal is then to find the parameters θ that minimize the total loss on experiences in our training data.

Planning with an inaccurate model

The environment model learned may be imperfect: $M_\theta = \langle T_\theta, R_\theta \rangle \not\approx M = \langle T, R \rangle$. Thus, the model-based RL agent's performance is limited to the optimal play it can achieve in the approximated MDP. Hence, the agent is only as good as the estimated model. When the model is inaccurate, the planning process will compute a suboptimal policy. One solution is to use model-free RL methods when the model is unable to promote further exploration.

Another solution is to use some Bayesian approach, in which one explicitly reasons about model uncertainty where deviations on reward and next state predictions are used to determine if one should plan. Nonetheless, in the continuous state-action setting, one may never revisit the exact same state or retake the same action. In these scenarios, we use function approximation like neural networks and not search methods relying on discrete state-actions. In summary, the model is not perfect, so we need ways of exploring uncertain parts of the world.

Planning problem: simulation-based search

The planning problem is finding a good policy using a learned environment model. This can be solved by taking samples of imagined trajectories and use those to do efficient planning. The key idea of this is based on *sampling* and *forward search*. Namely, since many real environments have large state spaces, one may not efficiently explore the whole state space. Thus, we need to resort to ways of sampling experiences in the environment such that the world is sufficiently explored to model its dynamics accordingly.

Forward search

Forward search algorithms select the best action by *lookahead* and they traditionally build a *search tree* starting from the current state as the root. The search algorithm then uses the model of the MDP to look ahead. Thus, there is no need to solve the whole MDP but only a sub-MDP starting from "now". This makes sense, since we often care more about understanding our current state (situation) than the state further into the future.

By analogy, one might be climbing a mountain and for this task the next step is more important to get up the mountain. Similarly, forward search focuses on what is likely to happen next in the short term future. Thus, one may not need to solve the whole MDP (i.e, finding a complete path like traditional search algorithms do) and instead focus on what is now represented by a sub-MDP (part of the environment search space).

Simulated-Based Search is a method that uses forward search combined with sample-based planning. Thus, it simulates k episodes (rollouts) of T experiences (sequence length) from time t (now) using its environment model: $\{s_t^k, A_t^k, R_{t+1}^k, \ldots, S_T^k\} \sim M_\theta$. These simulated experiences are then used to either learn or plan what to do next. Forwarch search rests on the assumption that planning always starts from some notion of "now". Intuitively, one starts from "now" and imagine what may happen next (trajectory experience sampled from model) and learn from that imagined experience. This is analogous to imagining how one will climb the mountain before actually doing it.

Sampling

Sampling helps us focus on what matters by sampling promising and revealing actions with high probability from the environment. For this purpose, a simulation policy π is used as a way of picking actions in the simulation model M_θ. For example, MCTS simulates k episodes from the current (real) state and evaluates actions by their mean return (i.e, average score on all simulations of doing action a into some state s). Using such simulation policy to rollout (i.e, unfold) games is a simple way of estimating the value of how good a state it.

Often this turns out to be a viable alternative to complicated value function approximations that may be hard to get right with complex ML methods. In practice, this simulation-based approach (also called MC rollouts) works very well, since it dynamically probes how good positions are from the current state (i.e, now) onwards, while ignoring all other irrelevant environment scenarios. The (weak) *law of large numbers* (LLN) roughly states that you obtain the expected value if you repeat an experiment independently a larger number of times. By the law of large numbers, given enough independent, simulated experiences, such model should approximate the real transition and reward function, meaning it gradually converges to an optimal policy given enough representative samples.

Recall that one resorts to informed, sample-based, approximation search methods to cope with the large search space in realistic complex environments. Tree-based search methods like MCTS is just one example of an informed search algorithm that uses a UCT heuristic to strike a balance between exploration of new places and exploitation of knowledge about the world. This boils down to the *exploration exploitation dilemma* of deciding whether to explore the action space or exploit current knowledge at any given time in the simulation.

Real and Simulated Experience

There are now two sources of experience. Real experiences are sampled from the environment (true MDP) using $T(s, a, s') = P(s'|s, a)$:

$$S' \sim P(s'|s, a) \tag{2.14}$$

$$R = R(s, a) \tag{2.15}$$

Simulated experiences are sampled from the model (approximate MDP):

$$S' \sim P_\theta(s'|s, a) \tag{2.16}$$

$$R = R_\theta(R|s, a) \tag{2.17}$$

Car Racing as a Model-based RL Planning Problem

Previously, the car racing environment was interpreted as a search problem (see 2.1.2) that is solved using a successor function. However, solving it as a planning problem requires access to a *Forward Model* (FM), which is a model that can simulate experiences to enable planning. This is not currently available, since the car racing environment does not support simulating outcomes by rolling forward and backward in time. For this purpose, it is not possible to make copies of the environment (e.g. using Python deepcopy), since it relies on a mutable Box2D object to maintain its own internal state (Github 2019).

Thus, we need access to a *model* (i.e, simulator) that enables simulated search, which is made possible by learning a model of the environment like in model-based RL. We can then either learn a policy offline or plan online using the model. Since the environment is partially observable and sequential, we will need to use a model that can capture the spatial (velocity) and temporal (time) dynamics of the environment. We also need to extend it with the reward needed for planning the next best action available in the current state. To disregard the Markov property, we may use RNNs (e.g. with LSTM units) to keep track of long-term dependencies.

2.4 Planning

In this section, we assume that a forward model is given, which estimates the transition dynamics and rewards of an otherwise unknown RL environment to enable planning.

Planning is any method that uses such model to produce or improve a policy. Thus, the type of agents we are concerned with are *planning agents*, which maintain a model of the world and use it to simulate actions. Based on this, the agent can simulate hypothesized consequences and decide which action is best to choose. This branch of AI is concerned with simulated "intelligence", which is similar to what humans do when trying to determine the best possible course of action in any situation by thinking (i.e, planning) ahead.

Arguably, using a model to do planning with simulated search is similar to search in symbolic AI where the rules (i.e, dynamics) that govern the world are learned, instead of using hard coded formulas or rules based on human domain knowledge and logic.

In this section, we justify not using classical uninformed AI search methods. Instead, we advocate for either using informed heuristic search methods (e.g., Monte Carlo Tree Search with UCT) or informed genetic local search methods based on hill-climbing and evolutionary algorithms.

2.4.1 Planning with Uninformed Classical Search

Given a model, search is one way of finding a policy (action) that will achieve a goal. In classical planning, we assume the environment to be fully observable, deterministic, finite, static and discrete (in time, action and states). Otherwise, searching for a plan may be overwhelmed by irrelevant actions when exploring the vast state and action search space.

However, when the number of states is large, its applicability is limited by the requirement to search for only one state at a time. In our case, the input state space is very large ($256^{3 \cdot 96 \cdot 96}$) and the output is a continuous action. Typically, this makes the planning problem more difficult, since the search space of state possibilities that need to be explored by the planning algorithms grows exponentially with the domain.

Assuming we have a model of the world, the agent has everything it needs to know that is relevant to planning. Now, the main challenges of AI and planning are twofold. Firstly, the states (frames in video games or complex real-world situations) need to be represented in a computationally tractable way; otherwise, one needs an efficient way of exploring this state space. Secondly, the model that is planned on might be *imperfect*, meaning it cannot unfold the future reliably from the start to a goal state.

2.4.2 Planning with Informed Heuristic Tree Search: Simulated Forward Search with Sampling (MCTS)

Since the state space is very large, we do not use traditional search methods. Instead, we resort to informed search methods like Monte Carlo Tree Search (MCTS) with a UCT heuristic to explore the most promising actions only. Sampling is used in MCTS to cope with the large state space by using random simulations (i.e, MC rollouts) to explore the state space sufficiently, resulting in better sample efficiency (i.e, fewer experiences needed to find good policy). Forward search is used in MCTS to select the best action by searching from the current time step and only looking into the near short term future.

Simulation-based search is the combination of doing forward search (i.e, lookahead) into the near future and doing sampling through random simulations. This is based on the idea that we do not need to solve the full MDP but only the sub MDP by employing a *divide and conquer* approach where we solve *sub goals* of planning. Instead of trying to find a trajectory that completes a whole track in car racing, one may find a trajectory that completes segments of the track. Please refer to appendix A.1.3 for a detailed explanation of how MCTS works.

The reason we spend so much effort understanding and distinguishing between different AI methods for problem-solving is to be able to cherish and defend the final choice of AI methods. In reality, there are many similarities when treating an environment as either a search, reinforcement learning, or planning problem. Namely, they all stem from using some search mechanism for problem-solving, which can be viewed as a search through a huge set of options (search space) to find a desired solution.

Ultimately, we need to understand the big picture of how basic concepts turned into the foundation of AI today to appreciate the fact that one may not solve today's problems using yesterday's solutions. Given that CarRacing has a continuous action space, that is not directly compatible with vanilla tree-based search methods like MCTS with UCT, which is why we now look into evolutionary planning methods.

Evolutionary Computation (EC)

Evolutionary Computing (EC) is a branch of AI, which is concerned with optimization algorithms inspired by natural evolution as a strategy for finding solutions to a problem. Natural evolution is described in a given environment with a population of individuals that strive for survival and reproduction. The fitness is determined by the environment and is used to decide how well individuals succeed in their goals. In a way, it defines their chances of survival and multiplying as part of further candidate solutions (Eiben and Smith 2015, p. 13). The idea of doing automated problem solving using Darwinian principles goes back to the 1940s where Turing suggested "genetical or evolutionary search".

There are four popular implementations ('dialects') of the EC idea, including *evolutionary programming* (EP), *genetic algorithms* (GA), *evolution strategies* (ES) and *genetic programming* (GP). Contemporary terminology refers to the whole field by evolutionary computing and the involved algorithms by *evolutionary algorithms* with each their subarea and *genetic algorithms* being the most popular (Eiben and Smith 2015, p. 14).

Evolutionary Algorithms (EA)

There are many variants of evolutionary algorithms but the common idea is the same: given a population of individuals in some environment with limited resources (e.g., rewards), competition for those resources causes natural selection (i.e, survival of the fittest) (Eiben and Smith 2015, p. 25). As such, EA can be seen as a general procedure that evolves individuals of a population. Often, the way the different methods of EA differ are in their representation of individuals (e.g. string of bits or vector) and choice of biological operators included (e.g. mutation or crossover or both). In this work, we will refer to all of these methods by EAs that consist of a set of required components shown in list 2.4.2.

In EAs, a population of individuals are initially created, which represent candidate solutions (chromosomes) to a problem. Each solution (phenotype) has an encoding (genotype). In order to find the best solution, the population is evolved in a number of generations where each individual is evaluated using a fitness function. Individuals with low fitness are replaced by offspring of the most fit individuals (elites). This is done using *crossover* and *mutation* operators. As a result, promising genes from fit individuals remain in the population, whereas suboptimal genes are discarded. Similar to evolution, the goal is to evolve individuals with genes that make them survive in the environment. We seek an optimal solution or plan represented as a sequence of actions (genes) in an environment (game) by exploring a subspace of policies (policy search) using evolution. An optimal plan can either be *learned* or *planned*. *Neuroevolution* is a branch of AI that uses evolution to train artificial neural networks (ANN) that can approximate the policy function. Alternatively, one can use a model and do online planning with evolutionary search methods.

Evolutionary Algorithm Components:

- Representation: definition of individuals (e.g. bit string or vector)
- Evaluation function: fitness (e.g. reward)
- Population: set of possible solutions (e.g. action space)
- Parent selection mechanism (e.g. tournament-based)
- Variation: crossover (recombination) and mutation (e.g. uniform)
- Survivor selection mechanism (e.g. replacement by elitism)

2.4.3 Planning with Informed Genetic Local Search: Rolling Horizon Evolutionary Algorithms (RHEA)

In real-time games, agents have limited time to respond to the environment. Thus, this requires the presence of a learned policy (*offline learning*) or a model to enable planning with a rolling horizon search procedure (*online learning*). For the latter purpose, evolutionary algorithms have shown to be a viable alternative to MCTS that can cope seamlessly with continuous action spaces (Perez, Simon M. Lucas, et al. 2013).

For this purpose, Rolling Horizon Evolutionary Algorithms (RHEA) may be used to do policy search in a subspace of policies by planning on a model while coping with a continuous action space. These are algorithms, which "encode individuals as sequences of actions". *Rolling Horizon* (RH) refers to how the first action of a plan is executed in one game step before the plan is reevaluated and adjusted, looking one step further into the future and slowly expanding the 'horizon' (Gaina, Simon M. Lucas, and Pérez-Liébana 2017, p. 2). *Evolutionary Algorithms* (EA) refers to algorithms inspired from biology that encode solutions to problems as individuals that are part of a population evolved over several generations until a good solution is found or a budget is reached.

RHEA uses such population but we also consider the simplest RHEA variant with a population of one, which is the single-agent Random Mutation Hill Climber (RMHC) that uses a hill-climb optimization procedure. Individuals are evaluated by performing actions using a dynamics function in a forward model (simulation) and using the reward function to determine their fitness. The RHEA algorithm continuously evolves plans for a limited time frame in the simulated world model before transferring its best policy (plan) back into the real environment. This approach is interesting because most popular RL methods today learn a policy offline in video games. In contrast, this idea relies solely on a model and evolution to do simulated-based search that can cope with continuous action spaces.

RHEA evolves a population of individuals encoded as sequences of actions over multiple generations during a limited time budget. First, it initializes a population of uniformly random individuals. Initially, it may choose to do *elitism* by promoting one or two of the individuals with the highest fitness (here considered to be the total reward achieved in the simulated environment during planning). Subsequently, it uses tournament selection where a random subset of the population is used to select the parents of the offspring, such as the elites of the subset that have the highest total reward. Two parents are repeatedly sampled this way and used to produce offsprings (children) by (uniform) crossover (recombination). Finally, the children may be mutated by uniformly replacing an arbitrary action in its sequence of actions. The individuals in the evolved population are then evaluated by executing their sequence of actions with a model in the simulated environment, which is known as a simulated rollout (i.e, imagined future trajectory). The whole procedure is shown in figure 2.4.

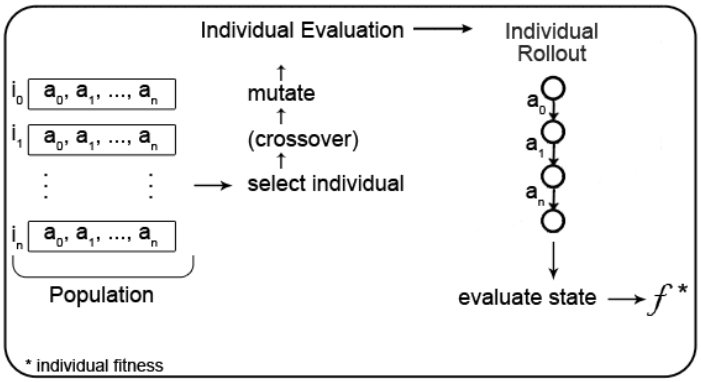


Figure 2.4: RHEA Flow Diagram (Gaina, Simon M. Lucas, Pérez-Liébana, et al. 2017)

Performance

Typically, evaluating the fitness of all individuals in the population is the most time-consuming part. Thus, one may run the fitness evaluation across multiple individuals concurrently. Another problem is that of *Premature convergence* in genetic algorithms that often converge too early into a suboptimal solution due to insufficient exploration of the search space. In this regard, it is crucial to use genetic operators (e.g. mutation) that encourage diversity because crossing over a homogeneous population may not yield new solutions. Diversity may be encouraged by doing uniform crossover, increasing the population size, or using a good mutation operator. RHEA also depends on the initial population used, which may be initialized with other search methods like MCTS, instead of random search. Considering these approaches may help ensure that RHEA does not get stuck in a local optima space.

Random Mutation Hill Climbing (RMHC)

Random Mutation Hill-Climb (RMHC) is a simple and effective type of evolutionary algorithm that repeats the process of randomly selecting a neighbour of a best-so-far solution and accepts the neighbour if it is better than or equal to the current individual. Thus, it is a RHEA variant with a single individual and a random hill-climb procedure to search in the policy subspace (i.e, action space). This *local search* method starts with a complete solution and iteratively tries to improve it by taking random steps or restarting with another assignment. Thus, RMHC is also referred to as Stochastic Hill Climbing (Deepa 2008, p. 27). The method is very efficient for problems with well-behaved continuous fitness functions that use gradients to guide the search. It will converge towards the optimal solution if the fitness function is continuous and is unimodal (i.e, only has one peak). However, in complex environments, the fitness function may have many peaks, making the algorithm likely to stop on the first peak it finds even if it is suboptimal. To avoid a local optimum, one may repeat multiple hill climbs at each time starting from different random positions. Otherwise, the method relies on a good mutation operator.

Pseudocode

Algorithm 1: Random Mutation Hill Climb (RMHC)

Input : $l \in \mathbb{N}$: individual length (horizon)
$d \in \mathbb{N}$: dimension of search space (discrete for illustration)
Output: First action of best individual (elite plan)

Randomly initialise an individual $\boldsymbol{x} \in \mathbb{R}^{l \times d}$ (uniform);
while *time not elapsed* **do**
  Select the element $i \in x$ to mutate;
  $\boldsymbol{y} \leftarrow \boldsymbol{x}$ after mutating $i \in \boldsymbol{x}$ (random mutation uniform);
  $Fit_{\boldsymbol{y}} \leftarrow fitness(\boldsymbol{y})$;
  if $Fit_{\boldsymbol{y}} \geq bestFitSoFar$ **then**
    $\boldsymbol{x} \leftarrow \boldsymbol{y}$ Update the best-so-far individual;
    $bestFitSoFar \leftarrow Fit_{\boldsymbol{y}}$
  end
end
return $\boldsymbol{x}[0]$

Algorithm 2: Rolling Horizon Evolutionary Algorithm (RHEA)

Input : $l \in \mathbb{N}$: individual length (horizon);
$d \in \mathbb{N}$: dimension of search space (discrete);
$\mu \in \mathbb{N} - \{1\}$: population size ($\mu = 1$ is RMHC)
Output: First action of best individual (elite plan)

`// population =` $\{x_0, \ldots, ..., x_{\mu-1}\}$ `where` $|population| = \mu$
population $\leftarrow \{\}$, **elite** $\leftarrow \{\}$;
for $i \leftarrow 0$ **to** $\mu - 1$ **do**
  $\boldsymbol{x_i} \leftarrow$ Uniform random individual $\boldsymbol{x_i} \in R^{l \times d}$ `// initialize`
  population $\leftarrow population \cup \{\boldsymbol{x_i}\}$ `// add individual to population`
end
while *time not elapsed* **do**
  for $i \leftarrow 0$ **to** $\mu - 1$ **do**
    $Fit_{x_i} \leftarrow$ fitness(x_i) `// evaluate individuals`
  end
  `// elitism: promote best individual to next population`
  elite $\leftarrow \{e \in population | Fit_e = max\{Fit_x | x \in population\}\}$;
  for $i \leftarrow 0$ **to** $\mu - 1$ **do**
    `// random subset:` $pop \subseteq population, |pop| = \frac{|population|}{2}$
    $\boldsymbol{pop} \leftarrow$ tournament(population) `// selection`
    $Fit_{\boldsymbol{a}} \leftarrow max\{Fit_x | x \in pop\}$;
    $\boldsymbol{parent_a} \leftarrow \{x | x \in pop, fitness(x) = Fit_a\}$;
    $Fit_{\boldsymbol{b}} \leftarrow max\{Fit_x | x \in pop - \{parent_a\}\}$;
    $\boldsymbol{parent_b} \leftarrow \{x | x \in pop - \{parent_a\}, Fit_x = Fit_b\}$;
    $\boldsymbol{child} \leftarrow$ crossover$(parent_a, parent_b)$ `// uniform recombination`
    $\boldsymbol{child} \leftarrow$ mutate(child);
  end
end
return $\boldsymbol{elite}[0]$ `// NB: after evaluating final population and elite`

2.4.4 N-Tuple Bandit Evolutionary Algorithm

The parameter configuration in a planning algorithm may have a vital influence on how well the agent can plan towards an optimal trajectory. Thus, we now consider the task of how to tune these parameters.

Grid search is an exhaustive approach where each possible parameter configuration is a point in a grid space. The grid is exhaustively searched until the optimal parameter configuration is found. While grid search may find the optimal parameter configuration, this approach is very computationally expensive. The parameter space defined for the planning algorithms may yield an enormous number of possible configurations, making grid search infeasible. Instead, the N-Tuple Bandit Evolutionary Algorithm (NTBEA) is used to tune the parameters of RHEA and RMHC.

NTBEA is an evolutionary algorithm that uses a model to estimate the fitness of new parameter configurations in the search space, while using a bandit approach to balance exploration and exploitation in the search space. NTBEA was developed and applied to RMHC by (Kunanusont, Gaina, and et.al. 2017) and also applied to RHEA (Simon M Lucas, Liu, and Perez-Liebana 2018) with promising results. Its ability to balance the trade-off between exploring and exploiting parameter configurations is beneficial, as it provides a way to avoid being trapped in a local optima of the parameter space, while still exploiting promising configurations. This makes NTBEA an effective approach for finding a somewhat promising configuration when tuning the parameters, since it does not exhaustively explore the parameter search space, but instead focuses its search in regions with promising parameter configurations only.

Key Components of NTBEA

NTBEA consists of three main components; a bandit landscape model, an evolutionary algorithm and a fitness evaluator.

The bandit landscape model is a model of the fitness landscape that is built and updated iteratively using the evaluations of solutions (i.e., parameter configurations). It uses a n-tuple system to model this fitness landscape that works as follows; provided a d-dimensional parameter search space (i.e., all possible d parameters), the parameter search space is then sub-sampled by its dimensions with several n-tuples where n ranges from 1 to d dimensions. For example, given a 2-dimensional search space with two parameters that each take two values, $p_1 \in (1, 2)$ and $p_2 \in (a, b)$, then the 1-tuple would be set of all possible parameter values for each of the two parameters: $\{(1), (2), (a), (b)\}$. Likewise, the 2-tuple would be the set of all possible parameter value combinations across both parameters: $\{(1, a), (1, b), (2, a), (2, b)\}$. This generalizes to any n-tuple ranging from 1 to n. Together, these n-tuples constitute the points (i.e, parameter configuration solutions) in the fitness landscape. Each n-tuple has a look-up table that contains the frequency at which a given parameter configuration in the n-tuple is encountered and the sum of the fitness associated with the evaluation of that particular parameter configuration.

The evolutionary algorithm component is responsible for finding the next promising point (i.e., parameter configuration) to be evaluated. Due to the large search space, it may be impracticable to evaluate each point. Instead, we model the relationship between the points and sample accordingly. Given some arbitrary current point, the algorithm will create k neighboring points, in which the proximity is defined by a mutation operator that uniformly mutates the current parameter configuration. Once the neighborhood is created, an Upper Confidence Bound (UCB) value is calculated for each neighbor point. Each parameter dimension in the search space is represented as an independent, multi-armed bandit where each arm represents a parameter value. The UCB of any arm i in a given parameter dimension d is calculated with equation 2.18 (Simon M Lucas, Liu, and Perez-Liebana 2018):

$$UCB_i = \hat{X}_i + k\sqrt{\frac{\ln n}{n_i + \epsilon}} \tag{2.18}$$

where $\hat{X}_i$ is the mean fitness obtained for picking value i in the evaluation of parameter d, k is the exploration factor, n is the frequency of evaluations with a particular parameter d and n_i is the frequency at which it is evaluated with a particular value i.

The mean fitness $\hat{X}_i$ acts as an exploitation term and $\sqrt{\frac{\ln n}{n_i+\epsilon}}$ as en exploration term. High k values promote exploration, whereas low k values promote exploitation. ϵ controls whether each parameter value should be used at least once. If ϵ is 0, then each parameter value is visited, which results in exhaustive exploration of the search space that is infeasible.

The combinations of parameter values across dimensions are modelled as super-bandits, which can test combinations of different parameters. Namely, in a d-dimensional parameter space where each parameter can take on n possible values, the super bandit would have n^d arms (i.e., parameter combinations). Hence, the final UCB value of a given point uses a modification of equation 2.18 where all n-tuples are aggregated and the computed UCB value is an unweighted arithmetic average as shown in equation 2.19 (Simon M Lucas, Liu, and Perez-Liebana 2018):

$$UCB(x) = \frac{1}{d}\sum_{j=1}^{d} UCB_{N_j(x)} \tag{2.19}$$

where x is a point in the parameter search space, d is the number of dimensions, $N_j(x)$ is the n-tuple indexing function that indexes the j^th bandit (i.e., parameter) in point x (i.e., configuration). Finally, the evolutionary algorithm will return the point with the highest UCB value.

Evaluator function is the final component that takes the point returned from the evolutionary algorithm and evaluates it on the problem domain. The reward obtained from the evaluation is used as the fitness value of the point and added to the fitness landscape bandit model.

Algorithm 3: N-Tuple Bandit Evolutionary Algorithm (NTBEA)

Input : $\mathbb{S}$: parameter search space
$fitness$: evaluator function
$n \in \mathbb{N}$: number of neighbours
$p \in [0, 1]$: mutation probability
$iterations$: maximum number of evaluations

Output: most promising parameter configuration

```
1  i = 0 // parameter configuration evaluation counter
2  model ← {} // initially empty fitness landscape model
3  x ← random point x ∈ 𝕊
4  while i < iterations do
5  |  Fit_x ← fitness(x) // evaluate parameter configuration
6  |  model.add(x, Fit_x) // add point to model landscape
7  |  neighbours ← neighbours(model, x, n, p) // mutation
8  |  x ← arg max_{x'∈neighbours} UCB(x') // find most promising point
9  |  i ← i + 1
10 end
11 return evaluate(model) // point with highest mean fitness in model
12
13 Function neighbours(model, x, n, p):
14 |  neighbours ← {}
15 |  while (|neighbours| <= n) do
16 |  |  x' ← mutate(x, p)
17 |  |  if x' ∉ neighbours then
18 |  |  |  neighbours.add(x')
19 |  end
```

The algorithm - The NTBEA algorithm is demonstrated in algorithm 3 and works as follows. An initially random parameter configuration is selected from the parameter search space and set as the current point. The algorithm then proceeds by repeating the following steps, until the maximum number of iterations has been reached:

- Evaluate current point to get its fitness (line 5)
- Add current point and its fitness to the n-tuple landscape model (line 6)
- Get n distinct neighbour points by using a mutation operator on the current point (line 7 and 13)
- For each point in the neighbourhood, calculate the UCB values and return the point with highest the UCB value (line 8)
- The highest UCB point is set as the current point

Finally, the best point is returned by retrieving the point with the highest mean fitness in the n-tuple landscape model. This point represents the most promising parameter configuration the algorithm was able to find.

2.5 Artificial Neural Networks (ANN)

In this section, we consider a branch of Machine Learning called Artificial Neural Networks (ANN), which are inspired by the human brain and used to capture complex non-linear patterns. Neural networks produce an output from an input and belong under the realm of *subsymbolic AI*, which is concerned with implicitly represented models that are learned from experience (Bolander 2019). We will consider a range of neural network architectures to learn a model (dynamics, reward) of a complex video game environment that enables planning with simulation-based search.

2.5.1 Feed-Forward Neural Network (FNN)

Feedforward neural networks (FNN) are artificial neural networks (ANN) where the connections (weights) between neurons (units) do not form a cycle and information only travels forward in the network. FNNs are mainly used for supervised learning when the data to be learned is not sequential or time-dependent. That is, FNNs try to compute a model function f on input x such that $f(x) \approx y$ for training pairs (x, y).

Single-layer Perceptron (SLP) - Linear Feed-Forward

The *perceptron* was invented by Frank Rosenblatt (1957) and is the most basic processing element in an ANN, best described as a single-layer neural network used for linear classification (binary) in supervised learning. The perceptron receives inputs $\boldsymbol{x} \in R^d$ from an environment that are all connected by a weight $w_j \in R$ and the output, y, is a weighted sum of the inputs:

$$y = \sum_{j=1}^{d} w_j x_j + w_0 = w_0 + w_1 x_1 + \ldots w_d x_d \tag{2.20}$$

where w_0 is the intercept (bias). Assuming w_0 is a weight coming from an extra bias unit, x_0, we can rewrite the output of the perceptron as a dot product using general matrix-vector notation:

$$y = \boldsymbol{w}^T \boldsymbol{x} \tag{2.21}$$

where $\boldsymbol{w} = [w_0, w_1, \ldots, w_d]^T$ and $\boldsymbol{x} = [1, x_1, \ldots, x_d]^T$ are augmented to include bias, weight and input. Notice, for $d = 1$ we have $y = wx + w_0$, which is the equation of a line with w_0 intercept and w as slope. The perceptron defines a linear hyperplane that divides the input space into two: a positive and negative half-space. This is known as a linear discriminant function that can separate two classes by the sign of the output using a threshold function. The perceptron uses a step activation function to determine the output $H(x) = 1$ if $x \geq 0$ else 0. This decides if each neuron should be activated, depending on whether each neuron's input is relevant to the prediction. The activation is then used as output for binary classification: $o = 1$ if $\boldsymbol{w}^T \boldsymbol{x} \geq 0$ else 0.

Training: Gradient Descent Backpropagation (Chain Rule)

An error function is used to measure how well our model predictions o are compared to the true values y across n training examples

$$X = \{(\boldsymbol{x_1}, y_1), \ldots, (\boldsymbol{x_n}, y_n)\} \tag{2.22}$$

using a loss function $L(o_i, y_i) = (o_i - y_i)^2$ s.t.

$$E(X) = \frac{1}{n}\sum_{i=1}^{N}(o_i - y_i)^2 = \frac{1}{n}(g(\boldsymbol{w}^T\boldsymbol{x}) - y_i)^2 \tag{2.23}$$

where $g(x)$ is the particular activation function used by the perceptron as ouput $o = g(\boldsymbol{w}^T\boldsymbol{x})$. Learning corresponds to iteratively updating the values of the weights and biases denoted $\theta = \langle \boldsymbol{w}, b \rangle$ of our model, using an optimization procedure like gradient descent to minimize the error function $E(X)$ by adjusting the model parameters in the direction of the negative gradient of the error function:

$$\theta_{t+1} = \theta_t - \alpha\frac{\partial E(X)}{\partial \theta} \tag{2.24}$$

θ_i are the values of the model parameters after the i^{th} iteration of gradient descent, $\frac{\partial f}{\partial x}$ is the partial derivative of f with respect to x and α is the learning rate, which controls the step size gradient descent takes each iteration. Values of α that are too large cause learning to be suboptimal (failing to converge), while too small values make learning slow.

Backpropagation ("backward propagation of errors") calculates the gradient of the error function with respect to each weight in the neural network. The "backwards" part is because one finds the gradient of the final layer of weights, which is propagated back to the preceding layer so that partial computations of the gradient from one layer are reused for computing the gradient of the previous layer. The backpropagation algorithm is derived using the chain rule (calculus) on the partial derivatives of the error function:

$$\frac{\partial E}{\partial w_{ij}^l} = \frac{\partial E}{\partial a_j^l}\frac{\partial a_j^l}{\partial w_{ij}^l} \tag{2.25}$$

, where a_j^l is the product-sum output of neuron j in layer l with input i before it is passed to a (non-linear) activation function (e.g. sigmoid) to generate an output. This decomposition of the partial derivative states that the change in the error function due to a weight is a product of the change in E due to the activation a_j^l times the change in the activation due to the weight w_{ij}^l. Backpropagation uses the chain rule to find the derivative of the composite function $o = g(\boldsymbol{w}^T\boldsymbol{x})$ and propagates the error back to adjust each weight based on their error contribution.

Multilayer Perceptron (MLP) - Non-Linear Feed-Forward

A perceptron with a single layer of weights can only approximate linear functions of the input, so we need to consider a non-linear discriminant instead to capture the dynamics of a complex environment. Hence, we consider another ANN called a Multilayer Perceptron, which is a feedforward network with intermediate *hidden* layers that uses non-linear sigmoid activation functions between the input and output layers to enable non-linear function approximation. MLPs are organized into *layers* with an *input lyayer*, zero or more *hidden layers* and an *output layer*. Input $\boldsymbol{x}$ is fed to the input layer, the (linear) "activation" $\boldsymbol{w}_h^T \boldsymbol{x}$ propagates forward and the values of the hidden units z_h are perceptrons, in which we apply a non-linear sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ to its weighted sum:

$$z_h = \sigma(\boldsymbol{w}_h^T \boldsymbol{x}) = \frac{1}{1 + exp(-(\sum_{j=1}^{d} w_{hj} x_j + w_{h0}))}, h = 1, \ldots, H \quad (2.26)$$

The output y is a perceptron in the 2nd layer taking hiddens as inputs:

$$y = \boldsymbol{w}^T \boldsymbol{z} = \sum_{h=1}^{H} w_h z_h + w_0, \quad (2.27)$$

where there is a bias unit in the hidden layer (z_0) and w_0 are the bias weights. The input layer of x_j is not counted due to its lack of computation so when there is a single hidden layer and an output layer, it is called a two-layer network (see 2.5). w_{hj} means weight from input j to unit h.

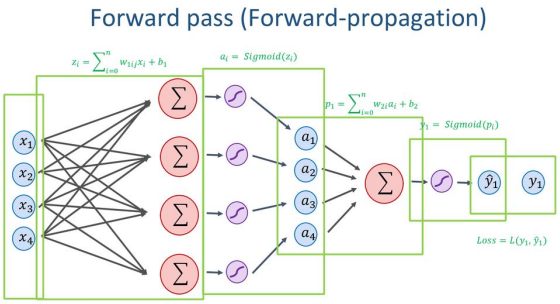


Figure 2.5: MLP Forward Pass (Dalgaard 2019, p. 16)

The most important contribution of a feed-forward network with a single layer and a non-linear activation, stems from the fact that an MLP with one hidden layer can learn any non-linear function of the input (Alpaydin 2014, p. 283). This is known at the *universal approximation theorem*, which states any continuous function can be approximated with a multilayer perceptron, given enough hidden units that use non-linear activation functions. Stacking many hidden units with non-linear outputs allows us to make piecewise approximations of a non-linear function

2.5.2 Reconstruction - Auto Encoders (AE)

A vanilla *Autoencoder* (AE) is an MLP-like ANN with a feed-forward architecture consisting of an input layer, an output layer and one or more hidden layers. Unlike an MLP, it is an unsupervised learning model that tries to reconstruct its inputs instead of predicting a target value (supervised). Thus, an *autoencoder* is a neural network trained to attempt to copy its input to its output (Goodfellow, Bengio, and Courville 2016, p. 493). Its goal is to learn a compressed latent representation (encoding or code) of the input data that may be used to efficiently reconstruct (i.e., decode) the input. Internally, it has a hidden layer h that describes a latent *code* $\boldsymbol{z}$ used to represent the input. The network consists of two neural networks; an encoder function $h = f(x)$ and a decoder that produces a reconstruction $r = g(h)$. Notice, it does not learn to predict $g(f(x)) = x$ everywhere, since such a model would not be useful. Instead, the model is forced to determine, which parts of the input should be copied to learn useful properties of the data that minimize a reconstruction error (e.g., MSE) between the actual and predicted input shown in figure 2.6.

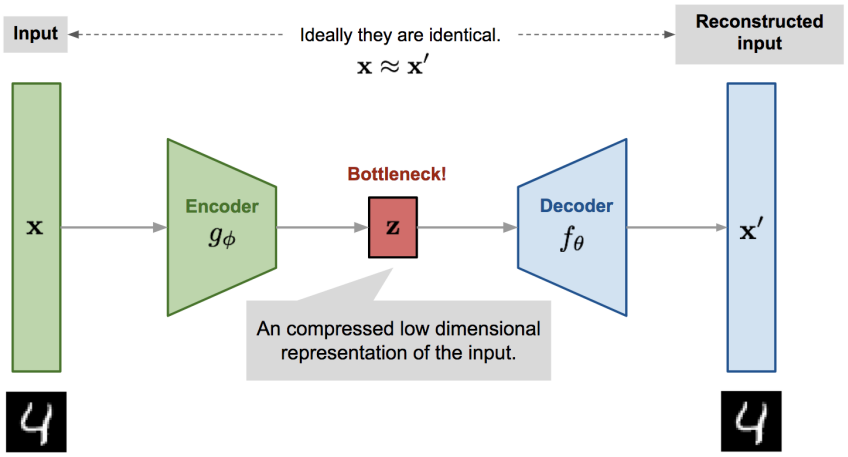


Figure 2.6: Autoencoder Architecture (Weng 2018a)

Traditionally, autoencoders were mainly used for dimensionality reduction or learning useful features. Today, theoretical breakthroughs in connecting autoencoders and latent variable models have made it possible to use autoencoders for generative modelling. Modern autoencoders have generalized the idea of an encoder and decoder beyond deterministic functions to stochastic functions $P_{encoder}(z|x)$ and $P_{decoder}(x|z)$ (Goodfellow, Bengio, and Courville 2016, p. 493). This is called a *generative* model that learns a distribution of the input data using unsupervised learning. The main purpose of the vanilla AE is deterministic input reconstruction, so it is not a generative model. The two most popular approaches for generative models are Generative Adversarial Networks (GANs) and Variational Auto Encoders (VAE). The Variational Auto Encoder (VAE) is a generative extension to the AE, making it possible to predict a distribution over the input that better captures the uncertainty in our reconstruction errors.

2.5.3 Deep Neural Networks (DNN)

A *Deep Neural Networks* (DNN) is an artificial neural network with multiple hidden layers between the input and output layers. Thus, an MLP with multiple hidden layers is an example of a DNN where each hidden layer has its own weights and apply the sigmoid function to its weighted sum. This is just a composed function with multiple nested hidden activations z_i in layers $i \in [1, \ldots, l]$ and output $y(z_l(z_{l-1}(\ldots z_1(x))$. Convolutional Neural Networks (CNN) are DNNs that use local information in computer vision for image recognition. Recurrent neural networks (RNNs) are DNNs that model sequential data with feedback loops (back to the past), capturing temporal dynamic behavior. Auto Encoders (AE) are similar to deep feed-forward MLPs, except they are unsupervised and aim to learn to reconstruct input. *Mixture Density Networks* (MDN) are neural networks that models the distribution of data as a mixture model.

2.5.4 Representation - Convolutional Networks (CNN)

"Convolutional neural networks (LeCun, 1989) are a specialized kind of neural network for processing data that has a known grid-like topology (e.g., 2D images). They use a linear *convolution* operation in place of general matrix multiplication, in at least one of their layers" (Goodfellow, Bengio, and Courville 2016, p. 321). CNNs are typically used in computer vision tasks where the image input has a *local structure*. Namely, nearby pixels are correlated and images have local features like edges, corners and objects. They solve the problem of a large parameter space in DNNs with fully connected layers through the use of *partially connected layers* and *weight sharing*.

A CNN can be seen as a structured MLP where each unit is connected to a local group of units because not all inputs are correlated. Instead, only a set of hidden units define a window (filter or kernel matrix) over the input space and are only connected to a local subset of the inputs. This decreases the total number of connections and reduces the number of free parameters. This may be repeated across multiple layers. Each layer is only connected to a small number of local units and checks for increasingly more complicated features by combining past features in a larger part of the input space until the output layer is reached. (**tutorial**) The inputs are the pixels and the first layer may learn to detect edges, whereas the second hidden layer may use the first hidden layer to learn more combinations of edges. Doing this across multiple layers serves as a kind of *hierarchical learning* where fewer features get more complex and abstract, until reaching the output of the network. The work of each hidden unit is called a *convolution* of its input with its weight vector (filter or kernel). The number of parameters can be reduced further by *weight sharing*, in which different units have connections to different inputs while sharing weight values. Thus, CNNs learn local structures of the data with fewer parameters than a fully-connected MLP without structure (Alpaydin 2014, p. 295).

The layers that consist of hidden neurons that are not connected to every input neuron (image pixel) but only to a local region (receptive field)are called the *convolutional layers*. Each neuron in the second convolutional layer is only connected to neurons in a local region of hidden units from the first layer. Consequently, the CNN focuses on learning low-level features in early layers and high-level features in the final hidden layers. This is useful to learn a visual representation of 2D video game environments whose patterns are inherently non-linear and hierarchical. For this purpose, a deep convolutional neural network can be used with convolutions to detect regional patterns and non-linear activation functions (e.g., ReLu $\sigma(x) = max(0, x)$) to capture non-linearities.

A *filter* is the small set of learnable weights that store a single pattern represented by a small image, the size of the receptive field, which corresponds to the input space that affects a particular unit of the network. A *convolution* is the process by which a filter moves over an image to find the most essential features. A set of filters (convolution kernels) is used to slide over the original image input to learn different hierarchical structures inherent in the data. A convolution works by applying such *filter* repeatedly to the input, which *convolves* (slides) over every $S \times S$ block of the input and computes the dot product of the filter matrix with the input pixel block values as the output. The output is called a *feature map* and serves as a compressed highlight of the features in the image that activates the filter the most. This is repeated across the layers into filters that activate on increasingly complex patterns. The *stride* is the number of pixels by which the filter moves over the input (e.g., 1 below). The *padding* is the number of pixels added to the input image when it is processed by the filter (kernel) of the CNN. Using *zero padding* will retain the dimensions of the input in the convolution output by adding zeros around the inputs. Typically, this is done before the convolution and may be followed by *max pooling*, which further reduces the dimensionality of the input space by using a max filter.

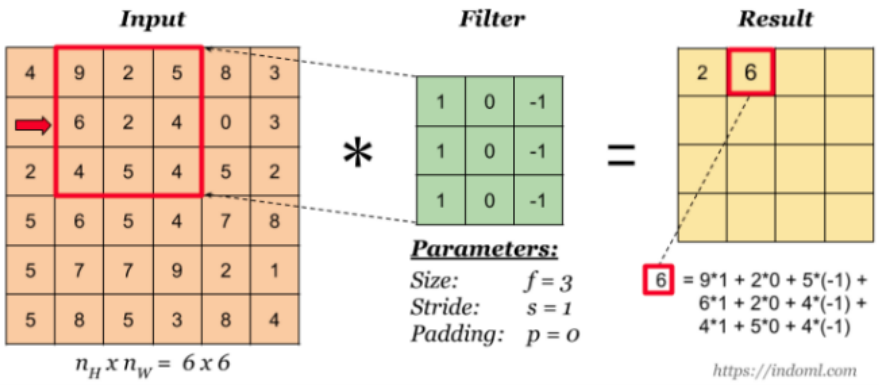


Figure 2.7: Convolution (Weng 2018a)

Deep RL agents use CNNs to approximate a representation and value function of a complex environment directly from raw high dimensional visual inputs. CNNs are also used in AEs to reconstruct input (ConvAE)

2.5.5 Sequential - Recurrent Neural Networks (RNN)

"Recurrent neural networks (RNNs) are a family of neural networks for processing sequential data" (Goodfellow, Bengio, and Courville 2016, p. 364). CNNs are specialized for processing images, whereas RNNs are good at processing a sequence of values by learning a function f on variable-length input $X_t = \{x_1, \ldots, x_t\}$ such that $f(X_t) \approx y_k$ for training pairs (X_n, Y_n) $\forall$ $1 \leq t \leq n$ where n denotes the total number of minibatches of such sequences of length t.

To go from MLPs to RNNs, we use the idea of *parameter sharing* across the model like the CNN. RNNs are recurrent due to cycles (feedback loops) in the network, which represent the influence of the present value of a variable on its own value in a future time step. Such a computational graph is represented by *unfolding* a recursive or recurrent computation with a repetitive structure (chain of events). This is used to represent a dynamical system in equation 2.28, which describes the state of a system $s^{(t)}$ as a function of the past $s^{(t-1)}$ parameterized by θ and is recurrent because the definition of s at time t refers back to the same definition at time $t-1$ (Goodfellow, Bengio, and Courville 2016, p. 365):

$$\boldsymbol{s}^{(t)} = f(s^{(t-1)}; \boldsymbol{\theta}) \tag{2.28}$$

The dynamical system in equation 2.28 may be extended with an external input signal $x^{(t)}$ to capture information about the past sequence:

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}) \tag{2.29}$$

Equation 2.30 is usually employed to define the values of hidden units in RNNs where $\boldsymbol{h}$ is used to represent the state of the RNN:

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}) \tag{2.30}$$

An RNN adds an output layer that read information of the hidden state h to make predictions. The RNN is then trained to perform a task that requires predicting the future from the past where $\boldsymbol{h}^{(t)}$ is used as a lossy summary of relevant information in the past sequence of inputs up to time t. Equation 2.30 shows that RNN can retain past information by tracking the state of the world (h) and updating it as it moves forward (unrolls). The unfolded recurrence after t steps is represented with a function $g^{(t)}$.

$$\boldsymbol{h}^{(t)} = g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(1)}) = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}, \boldsymbol{\theta}) \tag{2.31}$$

The function $g^{(t)}$ takes the sequence $(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(1)})$ as input to produce the current state and the unfolded recurrent (recursive) structure is used to factorize $g^{(t)}$ into repeated application of a function f. The unfolding ensures the model always has the same input size, regardless of the sequence length. This is defined in terms of transitions from one state to another, instead of the variable length of the history of states. Further, it allows us to use the same transition function f with the same parameters at each step. As a result, a RNN can learn a single model f that operates on all-time steps and sequence lengths. Thus, the RNN can be used on sequence lengths that were not present in the training data, as it does not need to learn a separate model $g^{(t)}$ for all time steps. Intuitively, the recurrent graph conveys the idea of information flowing forward in time (outputs and loss) and backward in time (gradients) in a succinct way.

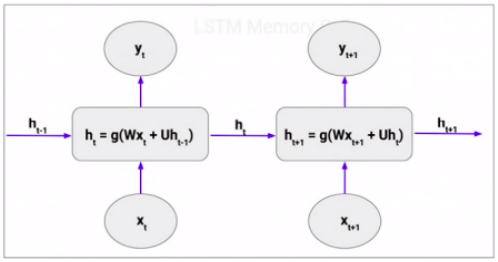


Figure 2.8: Vanilla RNN (Yau 2018)

The RNN is trained by repeatedly unfolding the network one time step and then applying the backpropagation algorithm ("Backpropagation through time"). *Teacher forcing* can be performed during training where the model receives the ground truth output $y^{(t)}$ as input at time $t+1$. The sequences can be rather long, so the unrolled RNN tends to be very deep. As a result, the partial derivatives of the gradient may tend to get smaller, as we move backward through the hidden layers, meaning the earlier layer neurons learn more slowly than later layers. This is called the *vanishing gradient problem* where the product of the chain of partial derivatives in the early layers becomes increasingly small when the partial derivatives are smaller than 1, until the weights reach zero, making them inactive. Likewise, the product of partial derivatives may explode if they are all beyond 1, which is just solved by clipping the gradients to below 1.

The problems occur in deep neural networks with gradient-based methods using backpropagation. The weights are updated proportional to the partial derivative of the error with respect to the current weight value in each training iteration. For DNNs, we end up multiplying partial derivatives many times. This may either lead to infinitely small or large partial derivatives used to update the weights. For small partial derivatives (< 1), the weight updates become insignificant, making them unable to change. In general, this happens because we use non-linear activation functions (e.g. $\sigma(x) = \frac{1}{1+e^{-1}}$) to capture non-linear patterns, which squeeze the inputs into very small output ranges (e.g., sigmoid maps $x \in [0, 1$ and tanh to $[-1, 1)$. Hence, a large region of the input space is mapped to small ranges, so the gradients become increasingly small.

This is worsened by the fact that we stack many layers of non-linearities in DNNs. To solve this, we may use activation functions that do not squash the input like the Rectified Linear Unit (ReLU) $\sigma(x) = max(0, x)$. However, this kind of activation function is subject to the *dying ReLU* problem where a neuron that is initially negative is unlikely to recover.

Long Short-Term Networks (LSTM)

The Long Short Term Memory Networks (LSTM) is a *gated RNN* based on the idea of a *gated recurrent unit*" (Goodfellow, Bengio, and Courville 2016, p. 397). Unlike RNN, LSTMs are capable of learning long-term dependencies and cope with short-term memory due to vanishing gradients.

Compared to the RNN, both use Backpropagation through time (BPTT) for training and both use a recurrent structure with repeating modules. In standard RNNs, this repeating module is typically a single *tanh* layer. In contrast, the LSTM uses three sigmoid neural networks (gates) and point-wise operations (e.g., vector addition) to learn what information should be included, forgotten and output, making short-term memory less of an issue for the LSTM. The sigmoid gates output a number between 0 and 1 that denotes how much information is let through each component where 0 means nothing and 1 means everything. Another key component of the LSTM is the cell state C_t, which acts as a small buffer of information passed through the chain of LSTM layers. As the cell state passes through each of the four LSTM layers, it may be modified by the gates, depending on the current input x_t and previous hidden state h_{t-1}.

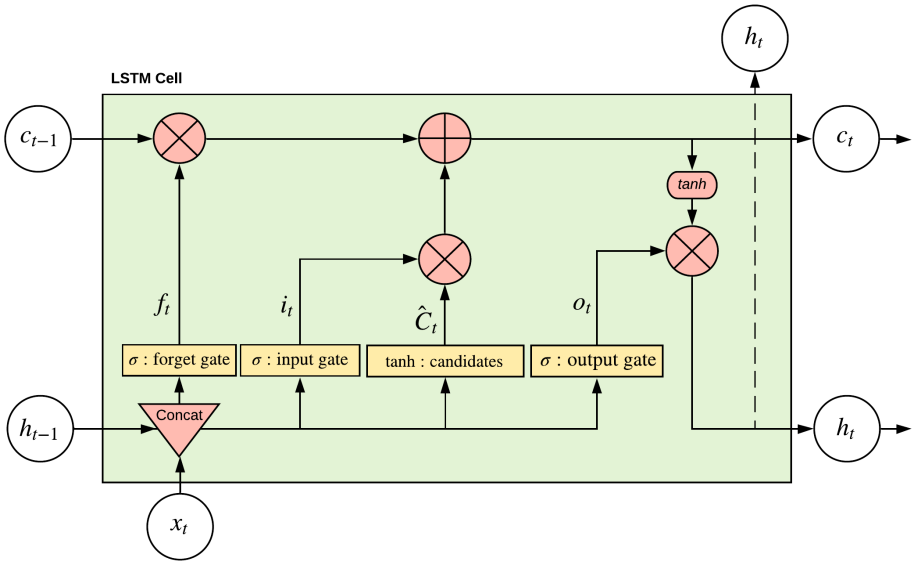


Figure 2.9: LSTM Memory Cell and Gating flow

The core flow of a single LSTM cell is shown in figure 2.9 and works as follows: Given some input vector x_t and previous hidden state h_{t-1}, the LSTM concatenates both vectors and passes the result through the first *forget gate*. Based on h_{t_1} and x_t, the forget gate decides what information is kept or removed from the previous cell state c_{t-1}. The output of the forget gate $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$ is then a filter vector that is multiplied to the cell state C_{t-1}.

The next step for the LSTM is to decide what new information is to be added to the cell state, which is a two-fold operation. As previously, the LSTM inputs the concatenated vector to the second *input gate* and a *tanh layer*. The input gate decides, which values are updated; $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$, while the tanh layer outputs a vector of new candidate values $\hat{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$. Both i_t and $\hat{C}_t$ are multiplied together to create a vector of new information to include in the cell state. The cell state C_t is then a linear combination of the forget gate and input gate $C_t = f_t \cdot C_{t-1} + i_t \cdot \hat{C}_t$.

Finally, the LSTM computes its current hidden state h_t, which is also a two-fold operation that is based on the updated cell state C_t and concatenated vector (h_{t-1} and x). Recall, the hidden state contains all information on past inputs. As previously, the LSTM inputs the concatenated vector into the final *output gate*. The output gate decides what information in the cell state C_t is used to form the current hidden state h_t by outputting a filter vector $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$. Simultaneously, the LSTM passes the cell state C_t through a tanh operator to squeeze the values between -1 and 1. Finally, the LSTM forms the current hidden state through multiplication $h_t = o_t \cdot \tanh(C_t)$. Both the current cell state C_t and hidden state h_t are then passed to the next cell in the LSTM where the same process is repeated. The equations are shown together below:

$$
\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \ \text{(forget gate)} \\
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \ \text{(input gate)} \\
\hat{C}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \ \text{(candidates)} \\
C_t &= f_t \cdot C_{t-1} + i_t \cdot \hat{C}_t \ \ \text{(cell state)} \\
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \ \text{(output gate)} \\
h_t &= o_t \cdot \tanh(C_t) \ \text{(hidden state)}
\end{aligned}
\tag{2.32}
$$

2.5.6 Uncertainty - Mixture Density Networks (MDN)

In general, the most popular output types in a neural network are the linear, sigmoid and softmax units. Normally, we consider neural networks that represent a function $f(\boldsymbol{x};\boldsymbol{\theta})$ where the outputs are predictions of the target value y. Now, instead we may also consider to approximate a conditional distribution $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ where maximum likelihood suggests we use the negative log-likelihood $-log\, p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ as loss function.

As a result, the outputs of such function are not the direct predictions of the target $\boldsymbol{y}$. Instead, the outputs are the parameters of a distribution over y. The goal of supervised learning is then to model a conditional distribution $p(\boldsymbol{y}|\boldsymbol{x})$, which is often modelled using a Gaussian distribution. However, data in most real-world domains does not simply admit a unimodal (i.e., single peak) Gaussian distribution but, rather, follows a multimodal non-Gaussian distribution. Thus, we seek a way to model complex conditional probability distributions in the real world.

This can be achieved by using a semi-parametric mixture model that describes $p(\boldsymbol{y}|\boldsymbol{x})$ as a linear combination of a mixture of parametric densities where each component k has its own density function $p(\boldsymbol{y}|\theta_k(\boldsymbol{x}))$, proportional to its mixing coefficient π_k. A *Mixture Density Network* (MDN) is a neural network that is typically used as the output in flexible neural networks (e.g., RNN) to approximate any arbitrary conditional density function $p(\boldsymbol{y}|\boldsymbol{x})$ of which the Gaussian mixture model is the most popular (2.33):

$$p(\boldsymbol{y}|\boldsymbol{x}) = \sum_{k=1}^{K} \pi_k(\boldsymbol{x}) N(\boldsymbol{y}|\boldsymbol{\mu}_k(\boldsymbol{x}), \boldsymbol{\sigma}_k^2(\boldsymbol{x})) \tag{2.33}$$

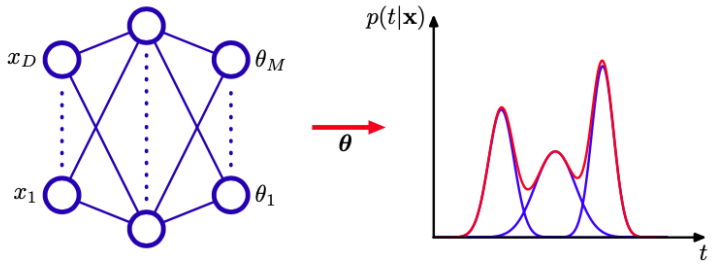


Figure 5.20 The *mixture density network* can represent general conditional probability densities $p(\mathbf{t}|\mathbf{x})$ by considering a parametric mixture model for the distribution of $\mathbf{t}$ whose parameters are determined by the outputs of a neural network that takes $\mathbf{x}$ as its input vector.

Figure 2.10: Mixture Density Network (Bishop 2006, p. 274)

The parameters of the mixture model are the mixing coefficients $\pi_k(x)$, means $\mu_k(x)$ and the variances $\sigma_k^2(x)$, which the Mixture Density Network uses to approximate a conditional distribution as a mixture model, which is typically used as output in a conventional neural network that takes $\boldsymbol{x}$ as its input. The neural network in figure 2.10 could be a two-layer network of hidden units with non-linear sigmoidal (e.g. 'tanh') activation functions.

Given K components of the mixture model and L target components in y, the MDN will have K output unit activations a_k^π that determine the mixing coefficients $\pi_k(x)$, K outputs a_h^σ that determine the kernel widths $\sigma_k(x)$ (variance) and $K \times L$ outputs a_{kj}^μ that determine the components $\mu_{kj}(x)$ of the kernel centres $\mu_k(x)$ (mean). The mixing coefficients satisfy the probability axioms (Equation 2.34):

$$\sum_{k=1}^{K} \pi_k(x) = 1\,,\; 0 \leq \pi_k(x) \leq 1 \tag{2.34}$$

This is achieved by using the softmax activation as output to normalize each of the mixture components (Equation 2.35)

$$\pi_k(x) = \frac{exp(a_k^\pi)}{\sum_{l=1}^{K} exp(a_l^k)} \tag{2.35}$$

Further, the variances must also be non-negative $\sigma_k^2(x) \geq 0$ so they are represented by the exponential activations (Equation 2.36)

$$\sigma_k(x) = exp(a_k^\sigma) \tag{2.36}$$

Finally, the means $\mu_k(x)$ are represented by the network output activations, since they are real components (Equation 2.37):

$$\mu_{kj}(x) = a_{kj}^\mu \tag{2.37}$$

The MDN is trained by using the maximum likelihood of minimizing the error function, defined by the negative logarithm of the likelihood (Equation 2.38) under the IID assumption where w are the weights and biases:

$$E(w) = -\sum_{n=1}^{N} log\left\{ \sum_{k=1}^{K} \pi_k(x_n, w) N(y_n | \mu_k(x_n, w), \sigma_k^2(x_n, w) \right\} \tag{2.38}$$

Chapter 3

Related Work

This chapter surveys previous work in learning models of the environment that facilitate planning (model-based RL). Recall a model $M = \langle T, R \rangle$ consists of a transition dynamics function and a reward function. In this regard, there are many ways of learning these components and most approaches approximate these functions using deep neural networks. They are either trained component-wise or end-to-end using gradient-free methods such as evolution (*neuroevolution*) or using gradients-based methods such as backpropagation and gradient descent. Given the setting where an agent, with an internal model, needs to continually act and learn in the world, a common theme in this work is a choice between two approaches to decide how to act: the model-free *learned* approach or the model-based *planning* approach.

In general, the learned approach uses a model of the environment and then learns a policy by training a deep neural network to approximate a distribution over actions given states $\pi^*(a|s) = argmax_\pi V_\pi(s)$. The goal is to pick the action that maximizes the return (expected total reward) $\sum_{t=0}^{T} r_t$ where r_t is the reward at time t and T is the horizon.

The *planning* approach uses the model to learn or search for a policy. The latter (i.e. planning with search) is usually done with Monte Carlo Tree Search (MCTS) to do simulation-based search by forward search and sampling. Alternatively, recent work shows Rolling Horizon Evolutionary Algorithms (RHEA) are a viable competition to tree-based search methods, such as MCTS and they support continuous action spaces by default.

Very little work has been done in showing how to do online planning on a learned model using evolution, which copes seamlessly with continuous action spaces that emulate the dynamics of realistic and complex real-time environments with a limited decision time budget. Ultimately, this section should help clarify and compare different contemporary methods that may be used to learn a world model and do planning. Please refer to appendix A.1.1 for a very brief summary of model-based RL and planning concepts required to understand the main ideas of this chapter if you have not read section 2.3.2 in the background chapter.

3.1 Learning Generative Models

In *Learning and Querying Fast Generative Models for Reinforcement Learning* (Buesing et al. 2018), the authors argue for the need of efficient and accurate environment models in model-based RL. They suggest using generative models that learn and operate on compact state representations, which they call *state-space models*. State-space models (SSMs) are described as models that find a compact state representation s_t, which captures all essential aspects of the environment on an abstract level. They assume that the next compact state s_{t+1} can be predicted from the previous state s_t and action a_t alone, without using previous pixels or actions.

They assume the compact state s_t is sufficient to predict o_t where o_t is the real state. As a result, latent states are sufficient to generate predictions of states into some future horizon. They consider both deterministic transition models (dSSMs) given a state-action pair at time t:

$$s_{t+1} = g(s_t, a_t) \tag{3.1}$$

and stochastic transition models (sSSMs):

$$p(s_{t+1}|s_t, a_t) \tag{3.2}$$

that explicitly model uncertainty over the future next state s_{t+1}. The latter is parameterized by introducing a latent variable z_t for every time step t whose distribution depends on s_{t-1} and a_{t-1}, by making the state a deterministic function of the past state, action and latent variable:

$$\begin{aligned} z_{t+1} &\sim p(z_{t+1}|s_t, a_t) \\ s_{t+1} &= g(s_t, a_t, z_{t+1}) \end{aligned} \tag{3.3}$$

In addition to the transition function, they use an observation model (i.e, decoder) to sample real observations:

$$o_t \sim P(o_t|s_t, z_t) \tag{3.4}$$

Latent variables are constrained to be normal with diagonal covariances.

The state-space models (SSMs) are implemented using a Residual Networks (*ResNets*), which are ANNs that skip connections or use shortcuts to jump over layers. This avoids vanishing gradients when training deep convolutional neural networks, by reusing activations from a previous layer, since gradients are propagated through fewer layers, which also speeds up learning dramatically.

The paper is based on recent work exploring models from standard recurrent neural networks (RNNs) to fully stochastic models with uncertainty and is one of the first examples demonstrating it is possible to learn a model of the world in Atari video games. However, they are limited to a discrete action setting and do not show planning is possible on the learned model. Further, they assume access to a pre-trained environment model via Monte-Carlo rollouts under a rollout policy so additional work would be to drop this assumption and jointly learn a model and agent.

3.2 World Models

In *World Models* (Ha and Schmidhuber 2018a), the authors explore building generative neural network models of popular RL environments. They call the model of the environment a *world model*, which is trained in an unsupervised manner to learn a compressed spatial and temporal representation of the environment. They compare this model to how "Humans develop a mental model of the world based on what they can perceive with their limited senses" (Ha and Schmidhuber 2018a, p. 1). Using this model, they demonstrate that it is possible to train a very compact and simple policy to solve the task of driving in the car racing environment and obtaining a high score when playing Doom in the Vizdoom environment. Thus, unlike the previous paper, they manage to show it is possible to jointly learn a model and an agent. Besides, they show it is possible to train the agent entirely inside the simulated environment (model) and successfully transfer back the learned policy into the actual environment. This seems to be one of the first studies that shows it is possible to learn a model of the environment and then do planning. In this case, they do *state-space planning* where a policy is learned from the latent state space provided by the model. Their model is represented by a visual component, memory component and a decision-making component.

Large neural networks are used to model the visual and memory component, which are trained separately using Backpropagation and an Adaptive moment estimation (Adam) optimizer, which is a variant of stochastic gradient descent that computes adaptive learning rates for all parameters. The visual component compresses visual frames $s \in R^{64 \cdot 64 \cdot 3}$ into a latent low-dimensional representation $z \in R^{N^z}$ where $N_z = 32$ in car racing and $N_z = 64$ in doom. The memory component makes predictions $P(z_{t+1}|a_t, z_t, h_t)$ about future codes (i.e., latent states) z_{t+1} when given an action a_t, current latent state z_t and recurrent hidden state h_t of historical information. The decision-making component decides which actions to take, based on z_t, h_t and rewards r_t from the real environment.

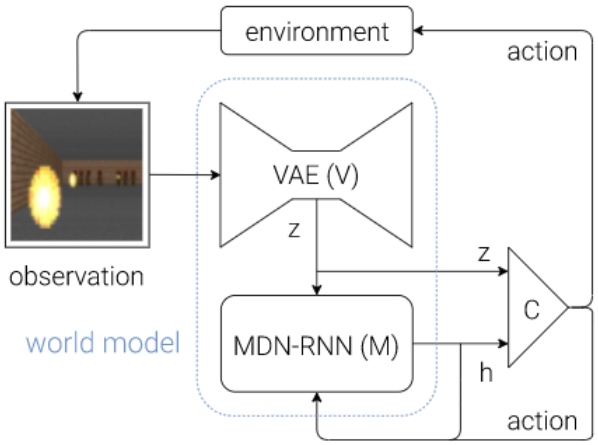


Figure 3.1: Agent model flow diagram (Ha and Schmidhuber 2018b)

They use a Convolutional Variational Autoencoder (CONVVAE) as the View component to compress each frame s_t at time t into a low dimensional latent vector z_t that can be used to reconstruct the original image (figure 3.2):

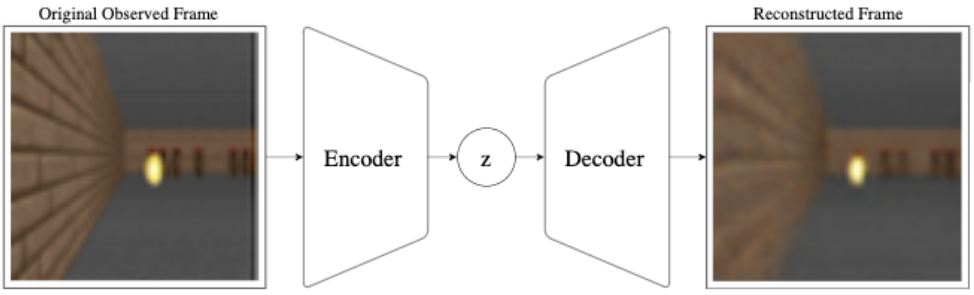


Figure 3.2: VAE Flow (Ha and Schmidhuber 2018b)

They use a Mixture Density Recurrent Neural Network (MDRNN) in the Memory (M) component to predict the future (figure 3.3):

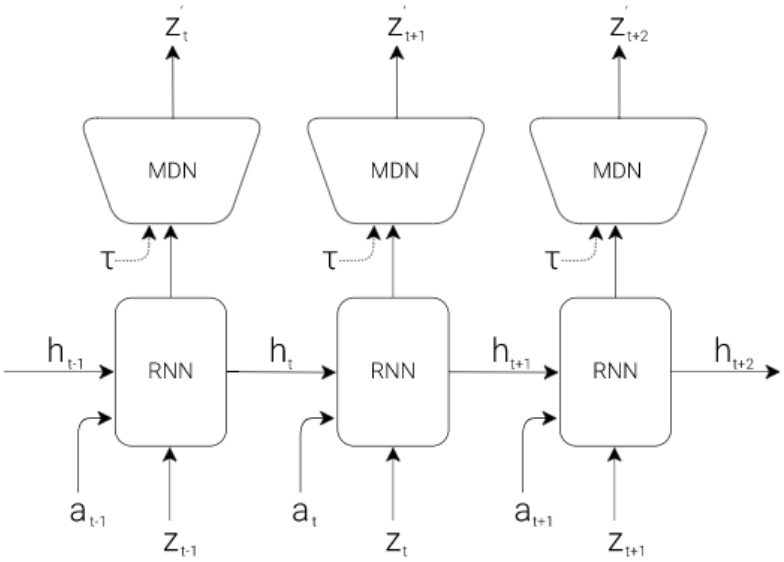


Figure 3.3: RNN with MDN mixture of Gaussian output used to sample prediction of next latent vector (Ha and Schmidhuber 2018b)

Figure 3.3 shows how the MDRNN is a predictive model of future latent z vectors that the View (V) component is expected to produce. The RNN outputs a probability density function $p(z)$, instead of a deterministic prediction of z, since many complex and realistic environments are stochastic. For this purpose, they approximate $p(z)$ as a mixture of Gaussians and train the RNN to output the probability distribution of the next latent vector z_{t+1} given the current and past information. In practice, the RNN models the parameters of $P(z_{t+1}|a_t, z_t, h_t)$, which is used to form the distribution in an MDN as output where a_t is the action taken at time t and h_t is the hidden state of the RNN at time t.

Arguably, it may seem excessive for the MDRNN to use a mixture of Gaussians to model the future, since the VAE is simply a single diagonal Gaussian distribution. However, the authors mention the discrete modes (peaks) in a mixture density model are useful for environments with random discrete events (e.g., a binary variable if a monster shoots a fireball). The VAE is trained to minimize $L_2\,(MSE)$ distance between the actual frames and predicted reconstructions (reconstruction error). It also minimizes Kl Divergence as a measure of difference between the estimated and true probability distribution of $P(z|s)$, since the goal of VAEs is to infer $P(z)$ from $P(z|s)$. This posterior is intractable so we estimate it using the encoder output, such that it is as close as possible to a standard normal distribution. For the MDRNN, we maximize the likelihood of the Gaussian Mixture Model from the MDN by minimizing the negative log-likelihood.

The Controller (C) model is used to determine a policy by finding an action that maximize the expected cumulative reward of the agent during a rollout in the environment. The controller is a simple single-layer linear model that maps z_t and h_t directly to action a_t at each time step: $a_t = W_c[z_t h_t] + b_c$ where W_c and b_C are the weight matrix. The weight matrix and bias vector maps the concatenated input vector $[z_t h_t]$ to the output action a_t. Notice, the authors fail to mention that they do not learn the reward function but use the rewards of the real environment to train their controller. This can be regarded as less puritan, since they do not use a complete model to do planning, which includes learning a reward function.

In *World Models*, the authors first collect 10,000 rollouts from a random policy that encourages exploration of the real state-space. Notice, their MDRNN and VAE use 512 hidden units and a latent size of 64 in doom and 256 hidden units and a latent size of 32 in car racing. Secondly, they train the VAE (V) to encode each frame into a latent vector $z \in R^{32}$. Thirdly, they train an MDRNN (M) to model $P(z_{t+1}|a_t, z_t, h_t)$. They evolve a controller (C) that maximizes expected cumulative reward of a rollout. This is done using a gradient-free *Covariance-Matrix Adaptation Evolution Strategy (CMA-ES)* to evolve C's weights. They use a temperature parameter to control the uncertainty of the simulated environment to avoid the agent from exploiting imperfections in it.

In summary, the authors in *World Models* have demonstrated planning in latent space using a learned model. The model uses a VAE to obtain a compressed representation. It uses an MDRNN to predict the future to capture the spatial and temporal traits of the environment. Thus, one may treat the model as an MDP that does not rely on the Markov property, since the current recurrent hidden state contains all past relevant information needed along with the current latent state to predict future latent states. The output of the MDRNN is an MDN to model the transition function as a stochastic mixture of Gaussians, which enables us to represent a partially observable environment and its dynamics with a stochastic transition function.

The paper shows it is possible to learn a model of the environment and use it, instead of the actual environment to learn a policy. Interestingly, they can model the future and come up with hypothetical scenarios using the MDRNN. Thus, the Controller can also train inside the simulated environment. Further, they show the Controller trained in the simulated environment (model) of doom is successfully able to transfer back its policy into the actual environment. According to the authors, their method is the first reported solution to solve the task in car racing and doom, by using a world model to take in raw pixel images and directly learn a spatial-temporal representation.

In our view, we have identified the following issues and rooms for improvement in the paper. Firstly, it may be fine for simple tasks to train the model on a dataset collected from a random policy, but we need an *iterative training* approach for complex environments, as they suggest. Namely, the agent should be able to explore its world and collect new experiences that can be used to retrain and improve the world model over time. A simple procedure is to initialize the model and Controller with random parameters, then do N rollouts (game episodes) where the agent learns and stores the generated actions, observations and rewards. These may then be used to train the MDRNN to model the future better and train the Controller to maximize expected rewards inside the model. This cycle can be repeated until the task is done.

Secondly, if the MDRNN does a poor job at predicting the future, that means the agent has encountered parts of the world it is unfamiliar with. For this purpose, one could look into ways of encouraging *exploration* of unfamiliar parts of the world, which could serve as new data collected to improve the world model. According to the paper, one may flip the sign of the MDRNN loss function to encourage exploration, since this would cause previously bad future predictions to be explored more, leading the agent into new territory.

Further, they use the actual *reward* function of the real environment to train their Controller instead of a model. In addition, the author train each component separately, although *end-to-end training* of the components might lead to a representation more relevant to planning. For example, the VAE may learn to encode information irrelevant to the task at hand because it is not trained in combination with the MDRNN and Controller. However, the authors mention the benefit is that the VAE can then be reused for different tasks, unlike a model optimized for one task only in the environment.

Finally, the paper only shows how to do planning using a learned approach by combining the benefits of model-based and model-free approaches (learning a policy in a model). However, they do not show how to do online *plan-space planning* through search ("thinking"). Thus, one may look into iterative end-to-end training, exploration methods, reward modeling and online planning.

3.3 Deep Neuroevolution of World Models

In *Deep Neuroevolution of Recurrent and Discrete World Models* (Risi and Stanley 2019), the authors address the issue of world models that rely on multiple different neural components responsible for visual processing, memory and decision-making with inspiration from the *World Models* paper (Ha and Schmidhuber 2018a). They critique the need to train the components separately with different specialized training methods. Thus, they demonstrate that models with the same parts can be trained efficiently end-to-end through a genetic algorithm (GA). Interestingly, the paper shows that the evolved visual and memory system obtains an effective representation, similar to the system trained through gradient descent in the Ha and Schmidhuber (2018a) paper. In general, gradient-descent methods rely on a differentiable loss function, which means it must be continuous (i.e., have no breaks) and smooth (i.e., derivatives are defined).

However, the authors use evolution as their optimization technique, which also works well with nonsmooth optimization problems involving discrete variables (e.g., minimizing $f(x)$ subject to $x \in R^n$) so their approach open up opportunities for classical planning in latent space. The approach is commonly referred to as *Neuroevolution*, which is using evolutionary algorithms to train the parameters and topologies (i.e, structures) of ANNs. Neuroevolution is called *Deep Neuroevolution* when done on DNNs. The paper mainly focuses on exploring whether a GA can optimize a multi-component system end-to-end without the need for separate training phases. Moreover, they argue most approaches to learning dynamical systems mainly focus on gradient descent methods that work on RNNs, which is a popular choice of model used to capture a spatial and temporal representation of a dynamical environment. In their approach, they evolve DNNs with a simple genetic algorithm that adds Gaussian noise to the network's parameter vectors in the mutation operator. Further, they observe the importance of having a memory component to predict potential futures that allow the agent to take sharp corners in the car racing environment.

They also investigate whether their evolved memory model is able to do the same by using *t-SNE dimensionality reduction* to gain insight into the inner workings of the MDRNN latent state predictions. The authors show that the recurrent network changes drastically in situations where the agent needs to react quickly to changes in the environment (e.g. sharp turns). Further, the authors argue that the benefit of an end-to-end training approach might be more clear in complicated tasks for which data collected with random rollouts is insufficient. Finally, they have not looked into whether the agent can train and improve in the simulated environment like in the original world model paper. Nonetheless, they also use the rewards of the actual environment to train their controller and do not show whether it is possible to do online planning on a fully learned environment model.

3.4 Learning Latent Dynamics for Planning from Pixels (PlaNet)

In *Learning Latent Dynamics for Planning from Pixels* (Ha, Hafner, et al. 2019), the authors propose a Deep Planning Network (PlaNet), which is a purely model-based agent that learns the environment dynamics from frames and picks continuous actions through fast online planning in latent space. The authors argue that learning latent dynamics models for planning are subject to problems such as model inaccuracies, accumulating errors of multi-step predictions, failure to capture multiple futures and overconfident predictions outside training. For this purpose, they rely on a dynamics model that represents the world as a compact sequence of hidden states that can accurately predict the rewards multiple time steps ahead. They do this by using a latent dynamics model with both deterministic and stochastic transition components.

This paper demonstrates a method that solves the task of planning entirely inside latent spaces. This is achieved by using a recurrent state-space model with both deterministic and stochastic components. Furthermore, *latent overshooting* is used, which trains multi-step predictions in latent space to improve the accuracy of long term future predictions used for planning. This minimizes compounding prediction errors that occur when modeling one-step lookahead. Besides, they model the environment dynamics from experience by doing iterative training of the dynamics model.

Moreover, a model-predictive control (MPC) is used to allow the agent to adapt its plan based on new observations. MPC replan at each step, similar to the *rolling horizon* planning approach. However, given latent states, it outputs a probabilistic distribution of actions instead of deterministic actions. Further, they do not use any policy or value network in contrast to model-free RL or previous model-based RL methods that predominantly learn a policy in the simulated environment. When collecting episodes for training data, a small Gaussian exploration noise is added to the action, to encourage exploration of the state-space environment and capture the various dynamics of the environment.

The *cross entropy (CE)* used is a planning algorithm to search for the best action sequence under the model. CE is a gradient-free, adaptive, randomized algorithm used for policy search optimization to infer a distribution over action sequences that maximize the objective. The main idea is that the probability of locating a somewhat optimal solution to the action-policy distribution with random search is a rare-event probability. Thus, the CE method is used to gradually change the action distribution of random search so that the rare-event is more likely to occur. It uses *cross-entropy* or *Kullback-Leibler divergence* as a measure of closeness between two distributions to minimize the difference between the initially random action-policy distribution and the optimal action-policy distribution.

The method estimates a sequence of sampling distributions that converge to a distribution with probability densities concentrated in near-optimal solution regions. The CE method employs an iterative procedure where each iteration switches between generating a random action-policy trajectory and updating the action distribution parameters. The switching is based on the reward obtained by following the action-policy samples for a particular state in the learned model. Ultimately, this helps produce a "better" trajectory sample in the next iteration. Before starting the search, they initialize a time-dependent diagonal Gaussian belief over optimal action sequences:

$$a_{t:t+H} \sim N(\mu_{t:t+H}, \sigma^2_{t:t+H} I), \tag{3.5}$$

where t is the current time step of the agent and H is the planning horizon. Starting from zero mean and unit variance $N(0, 1)$, they repeatedly sample J candidate action sequences, evaluate them under the model and re-fit the belief to the top K action sequences.

After i iterations, the planner returns the mean of the belief for the current time step, μ_t. Notice, the belief over action sequences resets from zero mean and unit variance again to avoid a local optima after receiving the next observation. A candidate action sequence is evaluated under the learned model by sampling a state trajectory starting from the current state belief and summing the mean rewards predicted along the sequence. They only evaluate a single trajectory per action sequence to focus the time budget on evaluating a larger "population" of different sequences. They use a reward function modeled as a function of the latent state without generating images to speed up the evaluation of large batches of action sequences. The latent dynamics model is a *recurrent state-space model (RSSM)* that can predict forward purely in latent space.

The Recurrent State-Space Model (RSSM) combines state-space models (SSMs) to model sequences and dynamical systems with RNNs to resolve the problem of capturing nonlinear, non-Markovian long-term dependencies (state transition depends on all past latents $z_{<t}$). The PlaNet paper highlights two findings to guide future designs of dynamic models: stochastic and deterministic paths in the transition model are crucial for successful planning. The purely stochastic transitions make it difficult for the transition model to reliably remember information in multiple time steps despite its generality. Thus, a deterministic sequence of hidden vectors $\{h_t\}_{t=1}^{T}$ are used to allow the model to access not just the last state but all previous states deterministically.

The recurrent state-space model (RSSM) is given by four components:

$$\begin{aligned} h_t &= f_{RNN}(h_{t-1}, z_{t-1}, a_{t-1}) \text{ (deterministic state model: RNN)} \\ z_t &\sim p(z_t|h_t) \text{ (stochastic state model: SSM)} \\ s_t &\sim p(s_t|h_t, z_t) \text{ (observation state model)} \\ r_t &\sim p(r_t|h_t, z_t) \text{ (reward model)} \end{aligned} \tag{3.6}$$

The deterministic state model is implemented as a recurrent neural network (RNN). This model splits the state into a stochastic part z_t (denoted s_t in original paper figure 3.4) and a deterministic part h_t. Both depend on the stochastic and deterministic parts at the previous time step through the RNN. In the RSSM, transitions depend on all past latents $z_{<t}$ so it is non-Markovian (i.e., given the present, the future is not independent of the past).

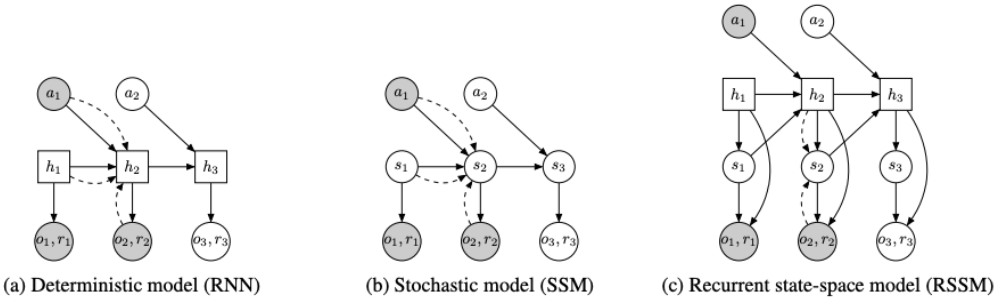


Figure 3.4: Latent Dynamics Model Design (Ha, Hafner, et al. 2019)

PlaNet use a recurrent state-space model (latent dynamics model) with both deterministic and stochastic components to predict a variety of possible futures as needed for robust planning, while remembering information over many time steps. This is combined with latent overshooting, which is generalizing the training objective for latent dynamics models to train multi-step predictions, by enforcing consistency between one-step and multi-step predictions in latent space. This helps improve long-term predictions required to plan with long horizons in latent space. Similarly, they only predict future rewards and not states to evaluate an action sequence, which speeds up planning.

Compared to preceding work on world models, PlaNet works without a policy network, since it chooses actions purely by planning so it benefits from model improvements on the spot. They evaluated it on continuous control tasks where the agent is only given image observations and rewards. By doing planning with a learned model, PlaNet was able to outperform model-free methods like DeepMind's *Asynchronous Advantage Actor-Critic (AC3)* algorithm released in 2016. For context, the basic idea behind an actor-critic agent is to combine policy and value based function methods using function approximators where an actor produces the best action for a given state (policy) and the critic receives the environment (state) and action by the actor as input and outputs the action value (Q-value) to create an actor-critic synergy.

3.5 Planning with a Learned Model (MuZero)

In *Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model* (Schrittwieser et al. 2020), the DeepMind team proposes the *MuZero* algorithm, which combines tree-based search (MCTS) with a learned model to do planning. They show MuZero can achieve superhuman performance in complex board games (Go, Chess and Shogi), as well as in visually complex Atari video games. MuZero learns a model iteratively that focuses on predicting the quantities most directly relevant to planning: the reward, the action-selection policy and the value function. They explain how planning algorithms based on lookahead rely on knowledge about the environment dynamics, which might not be available in real world domains like robotics. For this purpose, they suggest using Model-based RL to learn a model of the environment dynamics and then plan with respect to the learned model. They also mention that most successful methods today are based on model-free RL that estimates the optimal policy and value function directly from interactions with the environment.

MuZero is an extension of their previous *AlphaGo, AlphaGo Zero and AlphaZero* search algorithms that also do lookahead planning using MCTS but assume access to a perfect world model. In short, AlphaGo uses MCTS with multiple deep convolutional policy networks that have access to human expert moves. They then show that DNNs can play Go by predicting a policy (mapping from state to action) and value estimate (probability of winning in a given state) that are used by MCTS to select promising actions. AlphaGo Zero is an extension that does not use data from human games but plays against itself (i.e., self-play). It combines the value and policy network into a single Residual Network (ResNet) and uses the action distribution produced by MCTS lookaheads as target to train the policy network in a supervised manner, instead of using the policy network's predictions of the action distribution.

AlphaZero is an extension to AlphaGo Zero that can also play Chess and Shogi. MuZero is the current state-of-the-art algorithm from DeepMind that relaxes the assumption of having access to the environment dynamics and supports single-agent domains with discounted rewards, unlike its predecessors that work in two-player games with undiscounted terminal rewards of ± 1. Chess, Shogi and Go are all examples of games with a perfect simulator, since the dynamics are just the rules of the game. However, MuZero demonstrates it is possible to learn a model in Atari games from raw pixels and use that for successful planning using MCTS and a policy network. Figure 3.5 shows the MuZero pipeline.

Diagram A shows how MuZero uses its model to plan. The model consists of three connected components for representation, dynamics and prediction. First, observations (e.g. Go board or Atari screen) are passed into a *representation* function $h : o \rightarrow s$ to obtain the initial hidden state s, which is similar to the latent state representation z from $V : o \rightarrow z$ in *World Models*. The hidden states are used by a *prediction* function $f : s_k \rightarrow p_k, v_k$ to predict the policy p_k and value function v_k.

In *World Models*, there is no value function, since the authors make world models of video games that rely on a dense reward signal only. This is unlike MuZero that plays on board games where a value function is needed to determine the value of a particular state of the game, since the agent only receives a reward at the end of the game when it has won or lost, making the reward signal very sparse. Further, the policy is a stochastic distribution over actions $\pi = P(a|s)$ in MuZero, whereas it is just a deterministic function $\pi(s) = a$ in *World Models*. Finally, a deterministic *dynamics* function is used to produce an immediate reward r_k and a new hidden state s_k given a previous hidden state s_{k-1} and candidate action a_k. This is different to *World Models* that uses a stochastic dynamics function: $z, a \rightarrow p(z'|z, a), r$, to produce a reward and next latent state sampled from a mixture Gaussian distribution over latent states.

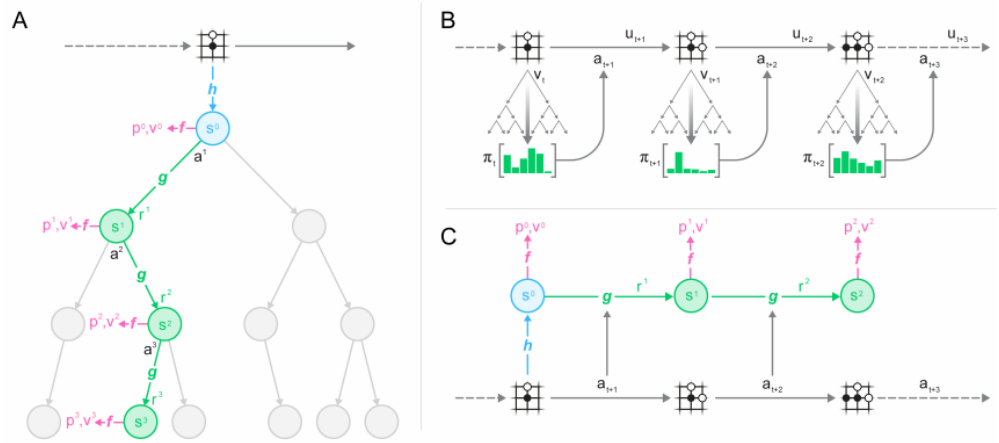


Figure 3.5: Planning, acting and training with a learned model (Schrittwieser et al. 2020)

Diagram B shows how MuZero acts in the environment. Monte-Carlo Tree Search (MCTS) is performed at each timestep t to do lookahead search of promising actions. The basic algorithm involves iteratively building a search tree until some budget is reached, upon which the best-performing root action is returned. Each node in the search tree represents a state of the game and edges to child nodes represent actions leading to subsequent states. An action a_{t+1} is sampled from the search policy $\pi_t = P(a_{t+1}|s_t)$, which is proportional to the visit count for each action from the root node. The environment uses the action a_{t+1} and the current observation o_t to generate a new observation o_{t+1} and reward u_{t+1}. At the end of the episode, the trajectory experience generated by MCTS planning is stored into a memory replay buffer. Refer to A.1.3 for the MCTS algorithm.

Diagram C shows how MuZero trains its model. Trajectories (rollouts of rewards, actions and observations) generated by MCTS planning are sampled randomly from the memory replay buffer. The representation function h is used to convert the past observations $o_1, \ldots, o_t$ from the trajectory into hidden states $s_1, \ldots, s_t$ latent space. At each step k, the model is unrolled recurrently for K steps by taking the hidden state s_{k-1} and real action a_{t+k} to predict the next hidden state s_k and reward u_k.

The parameters of the three components (representation, dynamics, prediction functions) are jointly trained, end-to-end by backpropagation-through-time to predict three quantities: the policy $p_k \approx \pi_{t+k}$, the value function $v_k \approx z_{t+k}$ and reward $r_{t+k} \approx u_{t+k}$ where z_{t+k} is a sample return (total expected reward), which is either the final reward (board game) or n-step return (Atari).

The loss function is composed of a reward, value, policy and regularization term. The reward targets are the observed rewards r_t^k that are compared to the predicted rewards u_{t+k} from the g dynamics function. The improved policy targets are generated by an MCTS search and used by the policy network that tries to minimize the error between the predicted policy p_t^k and the search policy $\pi_t + k$. The improved value targets are generated by playing the game or MDP like in AlphaZero. Unlike AlphaZero, MuZero supports long episodes with discounting and intermediate rewards. This is done by *bootstrapping*, which is estimating quantities based on other estimated quantities. In this case, the return z_t (total expected discounted reward) is estimated using the predicted rewards n steps into the future from the search value:

$$z_t = u_{t+1} + \gamma u_{t+2} + \ldots + \gamma^{n-1} u_{t+n} \tag{3.7}$$

Thus, the value function objective is to minimize the error between the predicted value $v_(t + k)$ and the value target z_{t+k}. An L2 regularization term (i.e., Ridge Regression) is added to reduce overfitting by keeping weights and biases small:

$$l_t(\theta) = \sum_{k=0}^{K} l^r(u_{t+k}, r_t^k) + l^v(z_{t+k}, v_t^k) + l^p(\pi_{t+k}, p_t^k) + c||\theta||^2 \tag{3.8}$$

,

where l^r, l^v and l^p are loss functions for the reward, value and policy and K denotes the horizon of future trajectories starting from time t.

In model-based RL, you can use simulations of the dynamics offline to improve the current iterations of the control policy. This is similar to how MuZero trains a policy network to learn offline from game trajectories generated by planning online with MCTS. MuZero cleverly uses MCTS to update the policy and value targets used to train the policy and value network. This is an example of *imitation learning*, which is the process of training a policy to mimic the actions of another policy.

Muzero trains the policy network to reflect the distribution of actions selected during a state in Monte Carlo Tree Search. The policy and value network is then used to influence action selection in future tree searches as done in AlphaZero and MuZero. As a result, planning is improved in large action spaces because the agent focuses on more promising choices based on previous experiences. Deep Neural Networks (DNN) are used to create trained policies, which can predict the actions of other agents and aid an individual agent's own search during planning.

The approach in *MuZero* is very similar to that of *World Models* except *World Models* do not employ an iterative end-to-end training procedure like in MuZero and in *Deep Neuroevolution*. Instead, the model components (representation view V and dynamics model M) are trained separately using gradient descent method and backpropagation. Finally, they train the controller with evolution to produce a deterministic action given a latent state, whereas MuZero trains a policy network to predict a distribution over actions given latent states.

Also, MuZero uses a value network, since they use the algorithm for board games, which *World Models* does not need to do for video games. In *World Models* they use a stochastic transition model, instead of a deterministic one like in MuZero. Interestingly, MuZero also seems to use *multi-step predictions* as indicated by the horizon K in the loss function. Yet, *World Models* does single-step predictions, meaning the latent dynamics model is only evaluated by its ability to predict the next latent state, given the current latent state $z_{t+1} \sim p(z_{t+1}|z_t, h_t, a_t)$ and not trained to minimize compounding errors as a result of consecutive future predictions. Hence, the *World Models* dynamics model is not evaluated by its ability to predict multiple steps ahead and relies on the assumption that the near future can be predicted from the immediate past.

On the other hand, the authors in *MuZero* argue the immediate future s_{t+1} may be accurately predicted from past observations $s_{t-k:t}$, which are summarized in *World Models* by the recurrent hidden state of the MD-RNN dynamics model. However, *World Models* fails to address the issue of compounding prediction errors when doing single-step predictions of the future, which may be improved by overshooting (predicting multiple steps ahead) to predict the consecutive steps into the future. Finally, the authors in *MuZero* paper mention extending their deterministic dynamics function to stochastic transitions as future work.

In our opinion, the main problem in MuZero is the immense amount of training data and hardware resources required. Namely, they use 20 billion frames in Atari and train 12 hours using 8 TPUS. In comparison, *World Models* only trains on 10,000 random rollouts that each contain 500 frames, which corresponds to 5 million frames that can be trained on a single Nvidia 1080 GTX in roughly the same time, although it only has a normalized 30% GPU score of a single TPU, according to a Deep Learning Benchmark in PyTorch (see A.2).

3.6 Dream to Control (Dreamer)

In *Dream to Control: Learning Behaviors By Latent Imagination* (Hafner et al. 2020), the authors show the possibility to learn long-horizon behaviors by latent imagination using a learned actor-critic approach. They argue it is feasible to learn world models from high-dimensional sensory inputs through deep learning but there are many potential ways to derive behaviors (policies) from them. Behaviors are often derived from dynamics models by either maximizing imagined rewards with a nonparametric policy (e.g., *World Models*) or by online planning (e.g., *PlaNet*).

Their main critique is that many world models are limited in how far they can accurately predict, rendering many model-based agents shortsighted. The paper is an extension to *Learning Latent Dynamics for Planning from Pixels* that proves the possibility of learning accurate world models from images. They use the PlaNet world model and focus on the behavioral aspect of planning that have held back model-based RL. For this purpose, they present Dreamer, a RL agent that solves long-horizon tasks from images purely by latent imagination. Dreamer learns a world model, given a sequence of observations, actions and rewards from the agent's past experience. It leverages the PlaNet world model to predict ahead using compact model states instead of images, enabling the model to predict many latent future trajectories on a single GPU. The three processes of Dreamer that are commonly used across model-based methods are: learning the world model (PlaNet), learning behaviors from predictions (i.e., policy) made by the world model and executing the learned behaviors in the environment to collect new experience. To learn behaviors, Dreamer uses a learned value network that takes into account rewards beyond the planning horizon and an actor (policy) network to compute actions. These three processes are repeated in parallel until the agent has achieved its goals. This is similar to the iterative training approach advocated by some of the other methods described previously.

Dreamer overcomes the problem of shortsighted planning agents by learning a value network and actor-network via backpropagation through predictions of its world model. The actor network learns to predict optimal actions by propagating gradients of rewards backward through predicted state sequences, which is not possible in model-free approaches. Thus, the agent knows how small changes to its actions affect, which rewards are predicted in the future and will refine the actor-network accordingly to maximize the reward. For rewards beyond the prediction horizon, the value network estimates the sum of future rewards for each model state. The rewards and values are backpropagated to refine the actor-network and gradually improve selected actions. Thus, the agent backpropagates value estimates through trajectories imagined in latent space of a learned model. In summary, Dreamer learns policy and value models in its latent space. The value model optimizes Bellman consistency of imagined trajectories (latent returns). The action model maximizes value estimates by propagating gradients back through imagined trajectories.

When interacting with the real environment, it executes the action model learned in the simulated environment. Thus, their planning approach is similar to *World Models* where a policy is learned inside the model and successfully transferred back to the real environment. However, Dreamer uses a multi-step prediction objective and a value network to model the future beyond the horizon. The latent dynamics world model is represented by three components shown in equations 3.9. The representation model encodes observations o_t and actions a_{t-1} into a latent state s_t, using the convolutional VAE (ConvVAE) encoder and decoder from *World Models* to create a continuous vector-valued latent model state s_t with Markovian transitions. The transition model predicts future model states without seeing the corresponding observations that cause them. This is the RSSM latent dynamics model used in PlaNet that has a stochastic and deterministic component and use multi-step prediction. The reward model predicts the rewards given the latent model states and is a three-layer dense neural network.

$$\begin{aligned} &\text{Representation model} : p(s_t|o_t, a_{t-1}) \\ &\text{Transition model} : q(s_{t+1}|s_t, a_t) \\ &\text{Reward model} : q(r_t|s_t) \end{aligned} \tag{3.9}$$

The distribution p is used to generate samples in the real environment and q for their approximations that enable latent imagination. The model mimics a non-linear Kalman filter, latent state-space model or HMM with real-valued states. However, it is conditioned on stochastic actions and rewards, allowing the agent to imagine outcomes of potential action sequences without executing them in the environment. The latent dynamics define an MDP that is fully observable, since the compact model states (s_t) are Markovian where future latent states are independent of past latent states. Imagined trajectories start in the true latent model state s_t drawn from the agent's past experience.

$$\begin{aligned} &\text{Action model} : a_t \sim q_\theta(a_t|s_t) \\ &\text{Value model} : v_t(s_t) \approx E_{q(\cdot|s_t)}(\sum_{t=0}^{t+H} \gamma^t r_t) \end{aligned} \tag{3.10}$$

The simulated trajectories follow predictions from the transition model $s_t \sim q(s_t|s_{t-1}, a_{t-1})$, reward model $r_t \sim q(r_t|s_t)$ and a policy $a_t \sim q(a_t|s_t)$. The aim is to maximize expected imagined rewards $E_q(\sum_{t=0}^{\infty} \gamma^t r_t)$ with respect to the policy. The actor-critic approach learns an action model that implements the policy and aims to predict actions that solve the imagined environment. The value model estimates the expected imagined rewards that the action model achieves from each state s_t. They are trained together as in policy iteration: the action model aims to maximize an estimate of the state value, while the value model aims to match an estimate of the value that changes as the action model changes. Both are parameterized as dense neural networks.

3.7 Summary

In *Learning Latent Dynamics for Planning from Pixels*, the authors have shown it is possible to do online planning in latent space using an adaptive random algorithm on a recurrent state-space model (SSM) with a deterministic and stochastic component and a multi-step prediction objective. In *World Models*, the authors showed it is possible to learn a stochastic latent MDRNN as a model of the world with a single-step objective and train a learned policy with evolution inside that same model. In *Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model* (MuZero), the authors show how to do planning with a learned model in board and video games using a combination of tree-based search (MCTS) and a learned policy network to do imitation learning.

In our opinion, *Learning Latent Dynamics for Planning from Pixels* (PlaNet) is the approach that is most similar to that in our thesis (i.e., online planning on a learned model). However, it uses a fairly complicated dynamics model and planning algorithm. In *Dream to Control: Learning Behaviors By Latent Imagination*, the authors use the PlaNet world model but no longer do online planning. Instead, their Dreamer agent uses an actor-critic approach to learn behaviors that consider rewards beyond the horizon, by learning an action model and value model in the latent space of the world model. Thus, their approach is similar to *World Models* where they plan on a learned model by training a policy inside the simulated environment with backpropagation and gradient descent, instead of evolution. The novel part is using a value network to estimate rewards beyond a finite imagination horizon. Also, *World Models* does not show how to do planning on a fully learned model, since the reward signal is not learned in their model. Finally, MuZero relies on extensive training data and access to unrealistic GPU resources, which may not be feasible in practice.

Altogether, many of the approaches seem to agree on an iterative, end-to-end training approach of the model. The components are directly relevant to planning (representation, dynamics, reward) and the trajectories are used to improve the world model iteratively. This is based on a multi-step prediction objective to ensure the model can predict multiple steps into the future. These approaches all demonstrate the possibility to plan on a learned model, yet very few show it with online planning. Further, none of the methods show planning on a learned model using an evolutionary online planning algorithm that may cope with continuous action spaces. Finally, most successful methods rely on large training data and GPU resources.

For this purpose, we will strive to show that it is possible to do online planning with a simple evolutionary algorithm on a learned model of the world trained using a reasonable amount of data in the visual domain of the 2D car racing environment.

Chapter 4

Approach

This section will explain our approach of implementing AI planning agents for car racing using the theory from the background chapter and the methods in the related work chapter. The chapter will describe the main methods used by the agents and include descriptions of the implementation and any enhancements made. The chapter will not reexplain the definitions, concepts and theory behind our approach (see background and related work for this). Instead, we focus on how to solve the continuous control task of learning how to drive whole random tracks from pixels by doing planning on a learned model of the car racing environment.

4.1 Planning: Online Policy Search (RHEA) in Latent Space (VAE) with Model (MDRNN)

We use model-based Reinforcement Learning to solve the continuous control task in car racing. This is done by online evolutionary planning on a learned model of the environment. Our solution is based on the model used in *World Models* (Ha and Schmidhuber 2018a) and the planning agent presented in *Rolling Horizon Evolution versus Tree Search for Navigation in Single-Player Real-Time Games* (Perez, Simon M. Lucas, et al. 2013). We use a latent representation of states in the environment and a learned model of the environment, which consists of an estimated transition dynamics and reward function that operates on the latent representation. The latent representation and model (dynamics, reward) constitute our "world model" and is used to obtain an efficient representation and a model of the environment. The world model enables us to do planning, which is using the environment model to find or improve a policy. In our case, we wish to find the best course of action given a state in latent space. Ultimately, our goal is to find such mapping where we can transfer our policy back to the real environment successfully. Namely, an optimal "plan" in latent space should also be optimal in the real environment. Today, most popular deep model-based RL methods do this by learning a model and doing (offline) state-space planning on it, which is learning an optimal policy from the latent state space. This is typically done using a policy network trained with gradient-based backpropagation (e.g. *MuZero*) or gradient-free evolution (e.g. *World Models*). Unlike model-free RL that learns a policy directly from real experiences, the policy is now learned from a model in latent space with compressed and efficient simulated experiences. We use the same latent space and model but, instead of doing state-space planning, we do plan-space (online) planning, which is searching through a subspace of plans to find an optimal policy. We use RHEA to do online planning (like *PlaNet*) where an agent evolves a plan (i.e., sequence of actions) in the imaginary model for some time, acts on the real world by performing the first action of its plan and then evolves a new plan repeatedly in a simulation until the game is over. Figure 4.1 shows the entire reproduction flow of our work. We generate 20,000 rollouts with a random and good policy. We then train the VAE to obtain a latent representation, which is used to train a MDRNN dynamics model. The world model (VAE, MDRNN) is tested to verify the VAE reconstructions and that the MDRNN respects the rank order of the rewards in the real environment. The best world model is picked to do NTBEA parameter tuning of our RHEA planning agent. Finally, the best world model is used to benchmark the tuned RHEA agent by planning in latent space and transferring its policy back to the real environment. It has already been shown in related work that it is possible to learn a dynamics model and efficiently plan in its latent space (*PlaNet*). *Our research is novel, since it shows how to do efficient online planning using a Rolling Horizon Evolutionary Algorithm (RHEA) for policy search with a learned dynamics model (MDRNN) in latent space (VAE).*

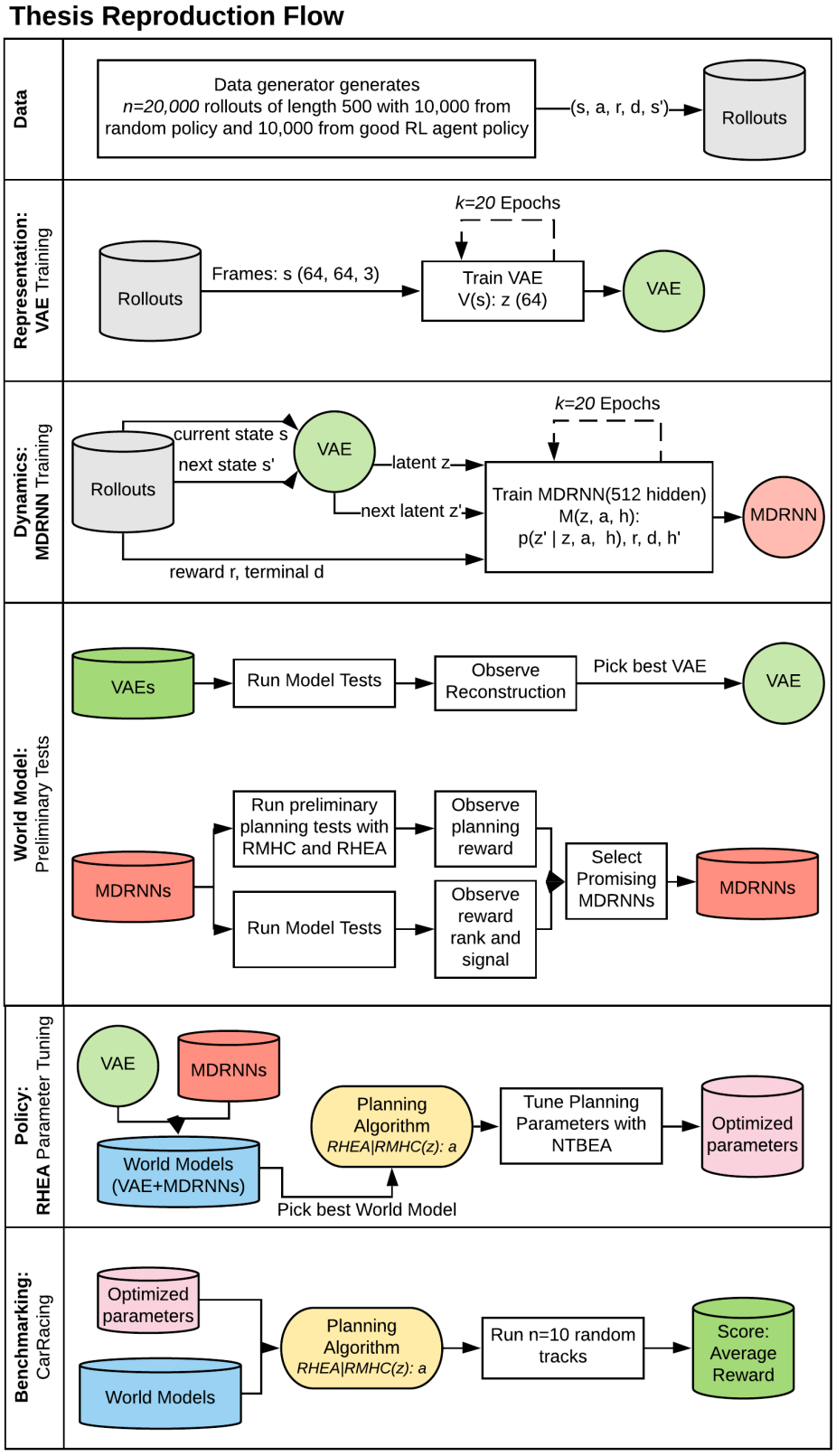


Figure 4.1: Approach: Data Generation, VAE Compression, MDRNN Dynamics, World Model Tests, NTBEA Parameter Tuning, RHEA Policy

4.2 System Architecture

This section briefly presents the technology stack system architecture.

Technologies

The final system is implemented in **Python 3.7**, which is an interpreted and high-level programming language widely used for machine learning purposes. The machine learning operations in the system are supported by **PyTorch 1.5**, an open-source machine learning library developed by Facebook. Alternatively, one may use Tensorflow 2 to implement our models. PyTorch and Tensorflow 2 are both Pythonic, which means they both follow the syntax and conventions in Python. Both use dynamic computation graphs that handle the model computations such that we only need to focus on writing the models, defining loss functions, selecting optimization procedures and writing the train-test loop. PyTorch was chosen, since it is the defacto ML library used in the research community and because our implementation is based on an existing implementation in PyTorch (Tallec, Blier, and Kalainathan 2018). To log the training of the VAE and MDRNN, we use **TensorBoard**, which is a library that that helps visualize and inspect the loss during experimentation. **NumPy** is used to support operations on large, multi-dimensional arrays of data and **Matplotlib** is used for general visualization purposes such as plotting planning trajectories and VAE reconstructions. Finally, **git** is used to version control the code base and **github** is the primary platform for hosting the thesis repository.

System Components

To ensure that the system is modular, maintainable and easy to understand, it is based on object-oriented programming design principles where the system components are represented by separate objects.

Each component is encapsulated and designed with the single-responsibility principle in mind, such that it only exposes the interface and is responsible for a single part of the system. This allowed us to quickly identify issues, implement extensions and work on different components simultaneously.

The final system architecture and its components are illustrated in figure 4.2 where the primary components are:

- **Models**: VAE and MDRNN
- **VAE/MDRNN-Trainers**: responsible for training the models
- **DataHandler**: generate rollouts from good or random policy
- **RHEA / RMHC**: implementation of the planning algorithms
- **EvolutionHandler**: delegate evolution operations for RHEA and RMHC

- **Simulated Environment**: act as an interface for the world model to enable efficient simulation-based search
- **Environment Wrapper**: custom interface for the open gym environment, which allowed us to decouple from the library
- **Testers**: planning tester benchmarks and tests the agents, while Model Tester inspects MDRNN reward predictions and VAE reconstructions
- **Parameter Tuner**: NTBEA Wrapper acts as an interface to the NTBEA algorithm based on (Bamford 2019)

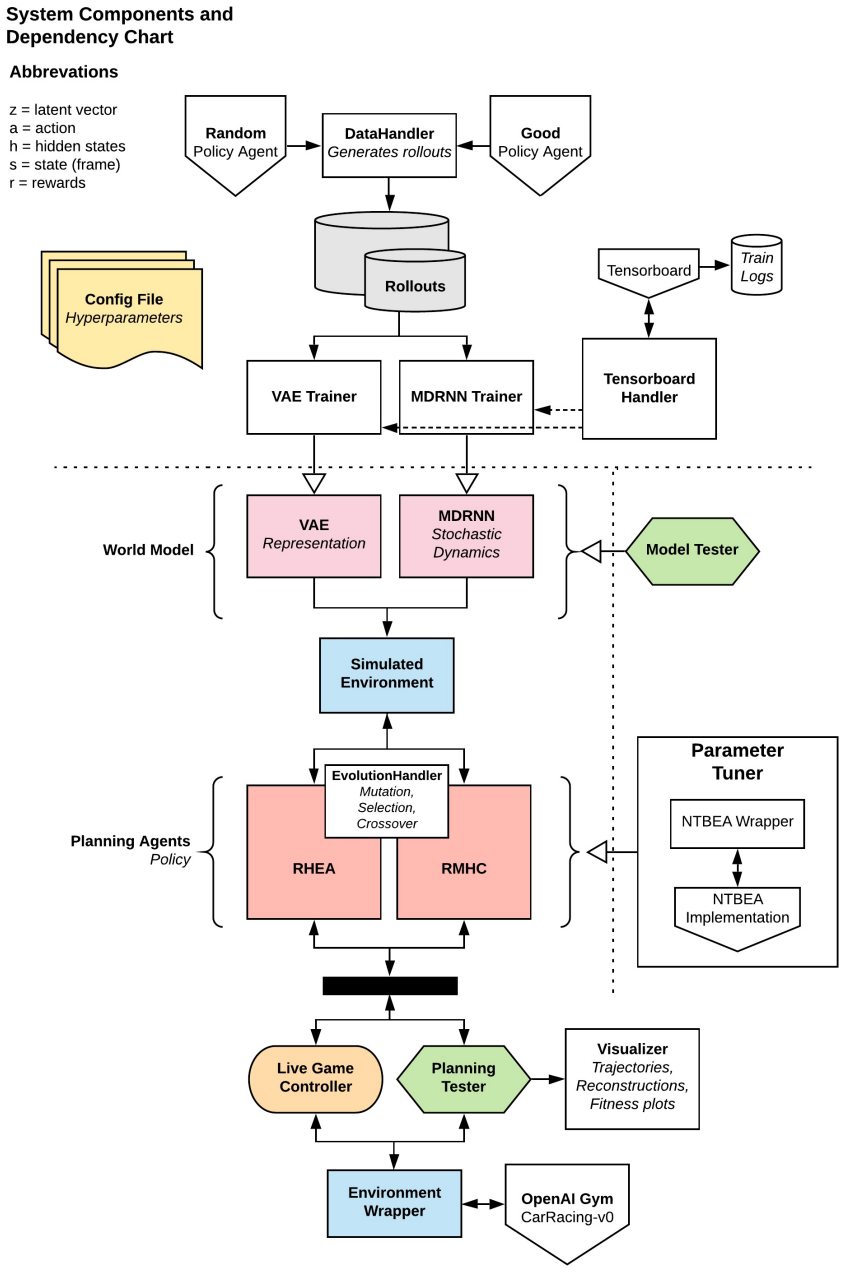


Figure 4.2: System Component Dependencies

Centralized Hyperparameters

Due to the large number of components in the system, one may quickly lose overview of all adjustable hyperparameters. To avoid this, a configuration file is used to centralize all hyperparameters used in the system. The hyperparameters are structured with JSON, an open standard file format that stores data based on key-value pairs (JSON objects) and array types (JSON lists). The hyperparameters of each component are categorized within their JSON objects and are dynamically loaded when the system is executed. Figure 4.3 shows a small sample of how the hyperparameters are structured. Please refer to the *config.json* file in the source code to see all hyperparameters and how they are configured.

```
2   "vae_trainer": {
3       "max_epochs": 20,
4       "train_buffer_size": 50,
5       "test_buffer_size": 50,
6       ,...,
7   },
```

Figure 4.3: Shortened example of how hyperparameters are centralized and stored in a JSON file

4.3 Data Generation

The MDRNN and VAE are trained through supervised learning, which relies on access to a representative dataset of environment experiences for training and testing the models. We have implemented a *Data Handler* component that is responsible for the data generation. We refer to a sample as a rollout of experiences that represents a single game. A full dataset consists of $n = 10000$ rollouts where each rollout has a sequence length of $j = 1000$. A rollout consists of a sequence of j experiences where each experience x_i is represented by an ordered tuple *(state, action, reward, terminal)* or as $x_i = \{(s_i, a_i, r_i, d_i), j \in \{1, \ldots, j\}$ where the state represents the frame generated after applying an action.

With the vanilla implementation, the rollouts are initialized by placing the car on random segments of the track (except on grass) to avoid the same starting segment of the track. We use a random policy on the environment while recording states, rewards, actions and terminals until the sequence length is satisfied or the game has reached a terminal state as illustrated in algorithm 4. Moreover, since the rollouts are independent we also parallelize the whole procedure to speed up the data generation. The final implementation of the data handler can be found the *datahandler.py* file in package *utility*.

Algorithm 4: Data generation procedure

Input : *maxRoll*: number of rollouts to generate
maxRollSize: rollout length
Output: generated rollouts

```
rollouts ← {}
environment ← car racing game
policy ← random or good policy
i = 0
while i < maxRoll do
    sequence ← {}
    state ← environment.reset()
    j ← 0
    while j < maxRollSize do
        action ← policy(state)
        state, reward, done ← environment.step(action)
        sequence.add(state, action, reward, done)
        j ← j + 1
    end
    rollouts.add(sequence)
    i ← i + 1
end
return rollouts
```

Good Policy and Random Policy

The policy used to generate rollouts may have a significant influence on how the VAE and MDRNN behave once trained. The VAE needs a dataset that contains a diverse set of frames such that it can encode different states (i.e., different road shapes) into latent states with minimal loss of information. This is necessary to ensure the MDRNN continuously receives well-represented latent encodings of real states. Likewise, the MDRNN needs a diverse set of rollout sequences so that it can capture the dynamics of the environment (i.e. reward signal mapping between actions and states).

To experiment with different policies, a good policy and a random policy has been implemented. The good policy uses the controller agent from the *World Models* paper (Ha and Schmidhuber 2018a) to generate rollout sequences that exhibit good driving behavior, which we use in our experiments to see how sequences with a good exploration of the environment state space may affect the performance of the planning agents.

The random policy is sampled as a Brownian sampling motion where a steer, gas and brake component in the action vector is sampled as $a_{t+1} = a_t + \sqrt{dt} \cdot N(0, 1)$. For each action component (steer, gas, brake), we generate a delta value that increases or decreases the previous action component.

The delta value is sampled from a standard normal distribution and multiplied with a *dt-factor* that controls how much influence the delta value has on the next action component. A higher *dt-factor* increases the delta value's influence, while a smaller *dt-factor* decreases the delta value's influence.

Arguably, one might simply replace the whole action with a new random action, but this may introduce large fluctuations in actions between time step and result in very inconsistent driving sequences. For instance, if the car is alternating a lot between steering left and right ("jaggy driving") while driving on a straight road, the car might still collect tiles to get positive rewards. However, if the rollout sequences mainly demonstrate "jaggy driving", feeding these to the MDRNN during training may force the MDRNN to associate rewards with jaggy driving. Thus, when the agent plans on a straight road, it may produce strong positive reward signals during "jaggy driving", instead of driving somewhat straight due to an inductive sampling bias where certain driving behavior is overrepresented in the rollouts.

Instead, a Brownian motion is used for random rollout sampling, since it uses controlled delta adjustments on previous actions to produce new action points with similar proximity to each other, thus reducing large action fluctuations between time steps. This may potentially help the MDRNN to capture dynamics associated with well-behaved driving and help encourage sufficient exploration during driving, whereas a uniformly random policy will often alternate between gassing and braking and remain stuck.

Corner Detection

It is evident that the rollout sequences are dominated by an inductive bias towards the presence of straight roads and left turns, since the tracks mostly consist of these segments. This may introduce imbalanced data where rare track segments (e.g., right corners, s-corners and other complex parts) might be underrepresented and not captured by the model.

For this purpose, we implemented a component that enables the generation of rollouts, which only contain sequences of corners. The data handler searches trough the track for corners by manually placing the car on different track segments until it detects a corner based on color codes. When a corner is detected on a frame, the car is deployed with a given policy and the rollout is recorded. This was done to circumvent the presence of imbalanced rollouts that favor forward track segments and help capture the environment dynamics better by exposing the model to a variety of driving scenarios on different road segments.

4.4 World Model

Please refer to 3.2 in the related work chapter for a detailed explanation of the *World Models* paper. We have chosen their approach to learn a world model, since it has been shown to work when doing planning in latent space using the same Car racing environment. Also, the components they use have successfully been employed to do planning in latent space in many of the other works presented, including *Deep Neuroevolution* (see 3.3) and *Dreamer* (see 3.6) .

Unlike them, we do not do offline state-space planning by evolving a policy controller to learn a good policy. Instead, we do online plan-space planning by using RHEA to search for good policies in a local subspace of policies during the game. Further, we augment a reward and terminal output in a MDRNN so we can use it for online planning, which requires a model with both dynamics and rewards. Thus, we learn a full model of the environment by not using rewards from the real environment in our policy search, which is arguably more puritan.

However, using random rollouts to train our MDRNN proved inssuficient, since the MDRRN was unable to respect the relative rank order of rewards in the environment. Namely, rewards should be higher when driving fast on the road and lower when either driving on grass or not driving at all. Thus, we chose to use a good RL agent (e.g. the controller in *World Models*) to generate 10,000 additional rollouts, which helped sufficiently explore the real environment state-space to learn a good reward function with a rank order similar to the that of the real environment.

Finally, we implemented an iterative trainer that initializes a random world model and repeatedly alternates between doing planning on it to generate "planning rollouts" and then using those rollouts to retrain an improved world model for the next iteration. However, this proved to be very slow and we have not had the time to show it is possible to learn a good world model starting from random rollouts only, which is left for future work. Thus, we currently assume access to rollouts obtained from a good policy where the environment is sufficiently explored to train a reward function that respects the rank order of rewards for online planning.

4.4.1 Vision: Variational Auto Encoder (VAE)

A Variational Autoencoder (VAE) is one of the most popular generative models that uses backpropagation-based function approximators (neural networks) to a build generative model. A VAE is often used in the research community because of its quick rise in popularity, mainly explained by the fact that its assumptions are weak, training is fast with backpropagation and its approximation has a small error. The idea behind a Variational Autoencoder (VAE) was invented in 2014 (Kingma 2014). Unlike traditional autoencoders that map data X into a fixed deterministic encoding Z, the VAE is a generative model since it maps data X into a probabilistic distribution over a latent representation $P_\theta(Z|X)$ parameterized by θ.

The reason we use a VAE is to provide a compressed representation z by learning to encode and decode in latent space. This lower dimensional latent space is more efficient for our memory (MDRNN) and planning agent (RHEA) to work with. A major benefit of a generative model is the ability to generate new samples x'. Importantly, unlike traditional autoencoders, we do not wish to learn spread out latent spaces. Instead, we strive for a *meaningful grouping*, which means similar observations exist in the same region of the latent space. Thus, samples that are close in the latent space may produce similar images when decoded. A useful latent encoding is obtained by using the KL divergence on the latent space to keep the latent distribution close to that of a standard normal. Ultimately, a generative model allows us to generate new samples efficiently in the simulated environment. This addresses the sample inefficiency problem and enables us to do planning by training a controller or doing policy search in latent space, which is ideal in the car racing environment that has a high dimensional observation space and continuous action space.

Architecture: Convolutional Variational Auto-Encoder (ConvVAE)

We use a Convolutional Variational Auto Encoder (ConvVAE) as our visual component (V) to learn an abstract, compressed representation $z_t \in R^{64}$ into of states (frames) $s_t \in R^{64 \cdot 64 \cdot 3}$. Since the environment gives observations as high dimensional pixel images, these are first resized to 64×64 pixels. The resized images are used as observations in our world model where each pixel is stored as three floating-point values between 0 and 1 that represent each of the RGB channels. Arguably, one might crop the image further into the region relevant to planning and use greyscale to further compress the frames. The dimension of our latent space is 64 similar to the Doom task in the original paper, since this proved to yield better reconstructions than using $N_z = 32$ as in the Car racing task of the paper. The ConvVAE architecture consists of inputs, layers, activations and outputs, as shown in figure 4.4.

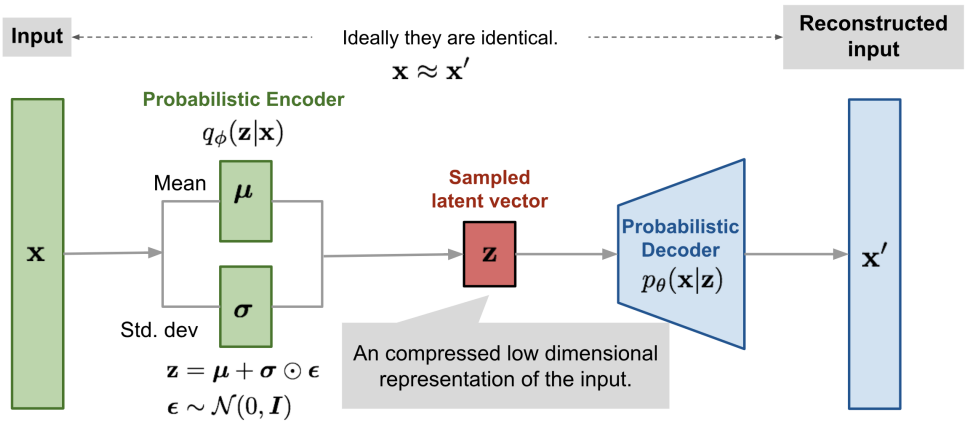


Figure 4.4: VAE Flow Diagram

The encoder is a neural network that outputs a compressed representation of a (preprocessed) state s (frame) using a deep CNN of four stacked convolutional layers and non-linear relu activations to compress the frame and two fully-connected (dense) layers that encode the convolutional output into low dimension vectors μ_z and σ_z:

$$encoder : s \in R^{64 \cdot 64 \cdot 3} \rightarrow \mu_z \in R^{64}, \sigma_z \in R^{64} \tag{4.1}$$

The means μ_z and standard deviations σ_z are used to sample a latent state z from a univariate Gaussian $N(\mu_z, \sigma_z I)$ with diagonal variance that does not model the correlation parameter p between each element of z :

$$z \in R^{64} \sim P(z|s) = N(\mu_z, \sigma_z I) \tag{4.2}$$

Thus, unlike a traditional autoencoder that deterministically maps real states into latent space $s \rightarrow z$, we enforce a Gaussian prior belief over the latent vector, which limits the information capacity for compressing each frame, but makes the world model more robust to unrealistic $z \in R^{64}$ vectors generated by the memory model (M). The decoder is a neural network that learns to decode and reconstruct the state (frame) s given the latent state z using a deep CNN of four stacked deconvolution layers:

$$decoder : z \in R^{64} \rightarrow s' \in R^{64 \cdot 64 \cdot 3}, \; s' \approx s \tag{4.3}$$

Each convolution and deconvolution layer uses a stride of two and the layers are shown in appendix figure A.4 as *Activation-type Output Channels x Filter Size*. All convolutional and deconvolutional layers use relu activations except for the output layer, since the output has to be between 0 and 1 in images. We trained the model for 20 epochs over 10,000 rollouts collected from a random policy. For this, the Mean Squared Error (MSE or L_2 distance) between the input image s and the reconstruction s' (reconstruction loss) is used, in addition to the KL loss, which measures how much one probability distribution differs from another probability distribution. The KL-divergence term acts as a regularizer of the model parameters θ, encouraging the approximate posterior $q_\theta(z|x)$ to be close to a tractable standard normal prior $p(z)$ (see (Kingma 2014, p. 11)).

VAE KL Divergence Loss

We assume latent states follow a standard normal distribution $z \sim N(0, I)$, the decoder reconstructions follow a multivariate normal distribution $p_\theta(x|z) = N(x|\mu, \Sigma)$ and the latent encodings follow a multivariate Gaussian distribution with diagonal covariance: $q_\phi(z|x) = N(z|\mu, \sigma^2 I)$.

We wish to maximize the lower bound $\mathbb{L}$ in equation A.18 by maximizing $E_{z\sim q_\phi(z|x)}[logp_\theta(x|z)]$ and minimizing $\mathbb{KL}(q_\phi(z|x)||p(z))$. Assuming $q_\phi(z|x)$ is Gaussian and $p(z)$ is a standard normal, then $KL(q_\phi(z|x)||N(\cdot))$ has a closed form solution.

The KL divergence between the multivariate Gaussian encoder with diagonal covariance in R^n: $q_\phi(z|x) = N(z|\mu, \sigma^2 I)$ and prior $z \sim N(z|0, I)$:

$$KL(q_\phi(z|x)||p(z)) = -\frac{1}{2}\sum_{i=1}^{n} log\sigma_i^2 + 1 - \sigma_i^2 - \mu_i^2 \tag{4.4}$$

The derivation of the KL divergence between two multivariate Gaussian distributions is found in appendix A.26. This used used to find the above closed form when z is assumed to be standard normal in appendix A.31.

VAE Reconstruction Loss

The decoder $p_\theta(x|z)$ is assumed to be an isotropic Gaussian $N(x|\mu, \Sigma = \sigma^2 I)$, which makes the negative log likelihood proportional to the L_2 squared Euclidean distance (MSE) between p(z) and x (Doersch 2016).

We assume the components of x are independent and $cov(x_i, x_j) = 0$ for $i \neq j$ and $var(x_i) = \sigma_i^2, \forall i$. Then $dist_{L_2}(x, \mu)^2 = ||x - \mu||_2^2 = \sum_{i=1}^{d}(x_i - \mu_i)^2$.

Thus, we can simply use the pixel-wise distance between the original and reconstructed frames that are normalized with a sigmoid function. The standard reconstruction loss is the squared euclidean (L2) distance between the original and reconstructed images, also referred to as mean squared error or MSE. Thus, to evaluate how well our VAE reconstructs training frames, we will do a pixel-to-pixel comparison of images using the MSE.

VAE Total Loss (MSE + KL)

$$\begin{aligned} L_{VAE} &= E_{z\sim q_\phi(z|x)}[-logp_\theta(x|z)] + \mathbb{KL}(q_\phi(z|x)\,||\,p(z)) \\ &\propto MSE(x, \hat{x}) + KL(q_\phi(z|x)\,||\,p(z)) \\ &= \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{x}_i)^2 - \frac{1}{2}\sum_{i=1}^{n} log\sigma_i^2 + 1 - \sigma_i^2 - \mu_i^2 \end{aligned} \tag{4.5}$$

VAE training: Reparameterization Trick

The expectation term in the loss function invokes generating samples from $z \sim q_\phi(z|x)$. Sampling is a stochastic process, which means we cannot backpropagate gradients. To make the VAE trainable, the random variable z is expressed as a deterministic variable $z = \mu + \sigma \cdot \epsilon, \epsilon \sim N(0, 1)$. This reparameterization trick makes the z sampling process trainable, since the stochasticity is transferred into a random variable $\epsilon \sim N(0, I)$ while the model is trained by learning the mean μ and variance σ^2 of the distribution.

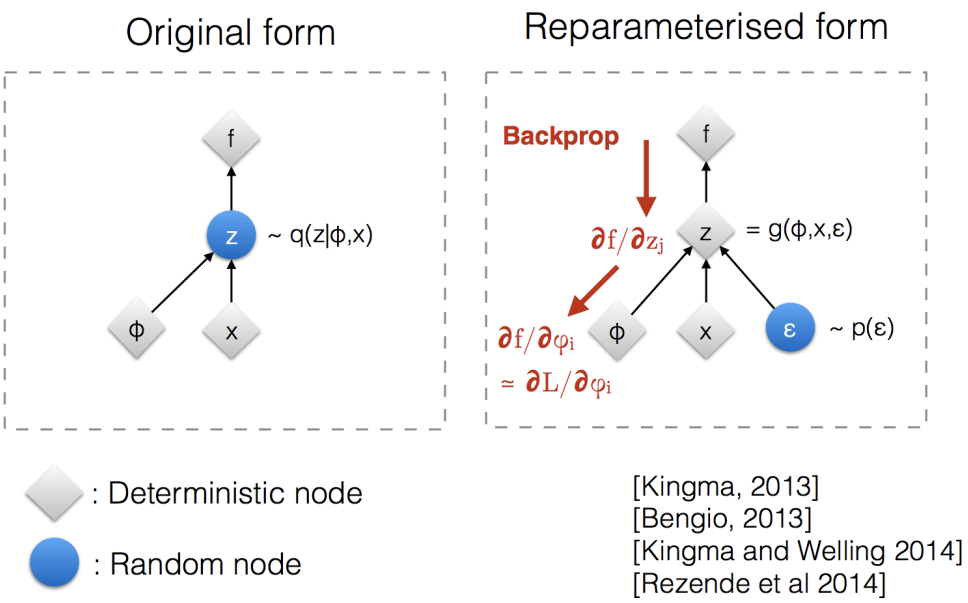


Figure 4.5: Reparameterization trick makes z sampling process trainable (Weng 2018b)

The theory behind the VAE

Please refer to appendix A.3.1 for an explanation of generative modelling and its purpose. Also, refer to appendix A.3.2 for a detailed derivation of the theory behind the Variational Auto Encoder. This includes the interpretation of a VAE as a Probabilistic Graphical Model (PGM) and the underlying theoretical concepts, such as, Bayesian Inference, Approximate Inference, Variational Inference and Kl Divergence and Maximum Likelihood Estimation. In particular, it explains how to approach generative modelling by approximation of a posterior from a variational inference perspective. Finally, it contains all derivations used in the final VAE objective loss function, which includes an explanation of the KL divergence score and its derivation. Most importantly, we wish to find a model that is representative of our data X: $P(X) = \int_z P(X|z;\theta)P(z)dz$. However, computing the integral over our latent z is intractable, which is why we direct our attention to approximate solutions, as described in the appendix.

4.4.2 Memory: Mixture Density Recurrent Neural Network (MDRNN)

For the Memory component, we use an LSTM with 512 hidden units combined with a Mixture Density Network (MDN) with five Gaussian mixture components as the output layer. The *Mixture-Density Recurrent Neural Network (MD-RNN)* is used to predict the latent future as a Gaussian mixture model. The VAE vision component is used to compress what the agent sees at each time frame, whereas the MDRNN memory component is used to compress what happens over time by predicting the future latent vectors z that the VAE is expected to produce. Most complex environments are stochastic in nature so the RNN is trained to output a probability density function $p(z)$, instead of a deterministic prediction of z. For this purpose, a Mixture Density Network (MDN) is used to transform the output of an LSTM to form the parameters of a mixture distribution with Gaussian mixture components. Thus, $p(z)$ is approximated as a mixture of Gaussian distribution and the RNN is trained to output the distribution of the next latent vector z_{t+1} given past information: $P(z_{t+1}|a_t, z_t, h_t)$ where a_t is the action taken at time t and h_t is the hidden state of the RNN at time t.

By default, standard RNNs are inherently one dimensional and therefore poorly suited to multidimensional data like images. Thus, in order to use RNNs on multi-dimensional tasks, the data must be pre-processed to one dimension. The standard LSTM is also one-dimensional, since the cell contains a single self connnection, whose activation is controlled by a single forget gate (Graves, Fernández, and Schmidbauer 2013, p. 6). Thus, we use the VAE vision component to compress 2D frames into a 1D latent state in $N_z = 64$ dimensions, which enables us to use a standard LSTM to unroll the future in latent space. The choice of the number of hidden units in the LSTM was based on our experiment results and 512 units was complex enough to capture the spatial and temporal representations of frames in latent space. We extend the output of the MDRNN with the expected (mean) reward to obtain a fully learned world model that may be used for planning online entirely in latent space. We do not model the correlation parameter between each element in z and instead have the MDRNN output a diagonal covariance matrix of a factored multivariate Gausssian distribution.

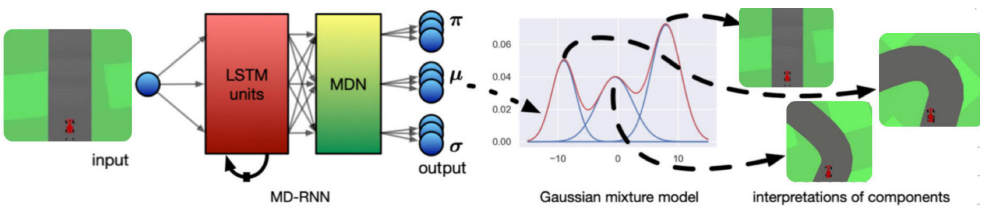


Figure 4.6: MDRNN model data with probability distributions composed of several components parameterized by π, μ and σ. A Gaussian mixture is used to model a variety of futures (Ellefsen, Martin, and Torresen 2019)

Gaussian Mixture Model (GMM)

The output of the MDRNN are the parameters π, μ, σ of a parametric Gaussian mixture model where π is used to represent the mixture probabilities:

$$p(z|\theta) = p(z|\pi, \mu, \Sigma) = \sum_{k=1}^{k} \pi_k \cdot N(z|\mu_k, \Sigma_k) \tag{4.6}$$

Each mixture component is represented by a multivariate Gaussian distribution with diagonal covariance $\Sigma_k = \sigma^2 I$ where $z \in R^d$:

$$N(z|\mu_k, \Sigma_k) = \frac{1}{(2\pi)^{d/2} \cdot |\Sigma_k|^{1/2}} exp(-\frac{1}{2}(x - \mu_k)^T \Sigma_k^{-1}(x - \mu_k)) \tag{4.7}$$

A diagonal covariance is used to reduce the parameter space from d^2 to d at the cost of flexibility, since this means we ignore the correlation between individual latent values. However, intuitively the compressed latent vector should mainly contain uncorrelated values that are "uniquely" important for reconstructing the original observations, since they are constructed using the VAE, which does a kind of dimensionality reduction when reducing the original observations with convolution filters.

By using a GMM with diagonal covariance, we can approximate complex probability distributions. The mixture probabilities are parameterized as $log\pi$ in PyTorch and recovered by taking the exponential. The mixture coefficients function as priors that describe the probability of the future target latent vector z_{t+1} having been generated by that particular Gaussian mixture distribution. The mixture coefficients are transformed via softmax to ensure the normalization axiom is satisfied, meaning $\sum_{k=1}^{k} \pi_k = 1$. Further, in order to interpret the mixture components as probabilities, they are ensured to be non-negative $0 \leq \pi_k \leq 1$.

Negative Log Likelihood Loss and LogSumExp Trick

Similar to the VAE, the parameters $\theta = \{\pi, \mu, \sigma\}$ of the Gaussian mixture model are found with maximum likelihood estimation. Thus, we wish to find the parameters θ that maximize the likelihood of the data, which is equivalent to minimizing the negative log likelihood:

$$\theta^*_{ML} = argmax_\theta p(z|\theta) = argmin_{\theta=\{\pi,\mu,\Sigma\}} - logp(z|\pi, \mu, \Sigma) \tag{4.8}$$

The likelihood function is given by (under the IID assumption):

$$\mathbb{L}(\theta|Z) = f(Z|\theta) = f(z_1, \ldots, z_n|\theta) = \prod_{i=1}^{n} p(z_i|\theta) \tag{4.9}$$

The negative log likelihood function is given by:

$$-\mathbb{LL}(\theta) = -log\mathbb{L}(\theta|Z) = -log(\prod_{i=1}^{n} p(z_i|\theta)) = -\sum_{i=1}^{n} logp(z_i|\theta) \tag{4.10}$$

This loss function is hard to optimize because the sum over the components appears inside the log, thus coupling all the parameters:

$$-\mathbb{LL}(\theta) = -\sum_{i=1}^{n} logp(z_i|\theta) = -\sum_{i=1}^{n} log(\sum_{k=1}^{k} \pi_k \cdot N(z_|\mu_k, \Sigma_k)) \quad (4.11)$$

Thus, we cannot obtain a closed-form analytical solution but need to use an iterative method to find a solution, which is typically done using either a *Stochastic Gradient Descent (SGD)* algorithm or an *Expectation Maximization (EM)* algorithm. We use a variant of SGD called *Adaptive Moment Estimation (Adam)*, which is another optimization method used to iteratively update the MDRNN weights by computing individual adaptive learning rates for different parameters.

Another drawback of the GMM is that there are typically many of parameters to learn so it may require lots of data and iterations to get good results. Namely, an unconstrained model with K mixtures (or simply K clusters) and $D-$dimensional latent data involves fitting $D \times D \times K + D \times K + K$ parameters, since there are K covariance matrices each of size $D \times D$, K mean vectors of length D and K mixture coefficient weights. This may pose a problem for datasets with a large number of dimensions (e.g. images), since the number of parameters grows roughly as the square of the dimension: $O(K \times D \times D) = O(D^2), D > K$. Thus, it is common to make assumptions to simplify the problem like fixing the covariance matrix of each component to be diagonal so the number of parameters grows roughly linearly in the number of dimensions: $O(D)$.

Finally, there is a significant problem associated with the Maximum Likelihood framework applied to Gaussian mixture models due to the presence of singularities (i.e., point at which the likelihood function takes an infinite value). Namely, assume one of the mixture components j has its mean μ_j equal to one of the data points so $\mu = z_n$. This data point will contribute a term in the likelihood function of the form:

$$N(z_n|z_n, \sigma_j^2 I) = \frac{1}{(\pi^{1/2})\sigma_j} \quad (4.12)$$

The limit $\sigma_j \to 0$ will make this term go to infinity so the log likelihood will also go to infinity. Thus, the maximization of the log likelihood function is not a well posed problem due to the presence of such singularities, which occur whenever a Gaussian component 'collapses' onto a specific data point. Thus, we need to either reset or throw away Gaussian mixture components approaching infinite likelihood. Alternatively, the EM algorithm can be used to find the maximum a posteriori (MAP) estimate, which uses a prior distribution as a regularization of the ML estimation.

Notice, the problem only occurs in a mixture Gaussian, whereas a single Gaussian will have multiplicative factors $\lim_{\sigma \to 0} e^{\frac{(z_n-\mu)^2}{2\sigma^2}} = 0$ multiplied onto the normalization term $\lim_{\sigma \to 0} \frac{1}{\sqrt{2\pi\sigma}} = \infty$ so the overall likelihood goes to zero. However, with two or more mixture components, one of them can collapse onto a point and contribute an ever increasing additive value to the log likelihood.

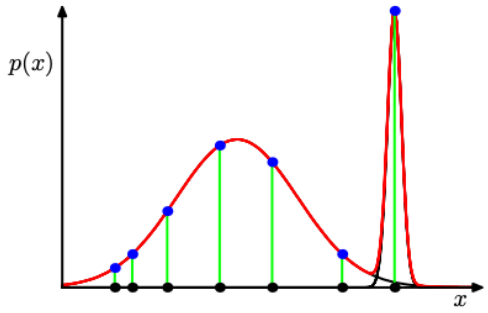


Figure 4.7: Singularity in likelihood function due to mixture (Bishop 2006, p. 434)

Finally, one needs to worry about the problem of *underflow*. Underflow is the condition where the result of a calculation is a number of smaller absolute value than the computer can actually represent on its CPU. Typically, likelihoods are too small to be represented as floating-point numbers, which is solved by using log-likelihoods instead. This works great when likelihoods are multiplied so the condensed product becomes an expanded addition of log values. However, in a Gaussian mixture, we take the log of a sum over mixture components whose likelihoods are frequently small enough to lead to underflow. For this purpose, we use the *"logsumexp" trick* to calculate $log(\sum_{k=1}^{k} \pi_k \cdot N(z|\mu_k, \Sigma_k))$ in equation 4.11 (GMM negative log likelihood) where each mixture component k is denoted $A_k = \pi_k \cdot N(z|\mu_k, \Sigma_k)$. The standard "logsumexp" trick computes $log(A_1 + \ldots + A_k)$ from log values $logA_1$ to $logA_k$ by dividing by the largest term A_m and converting the scaled terms to linear domain:

$$\begin{aligned} &A_k = \pi_k N(z_n|\pi_k, \Sigma_k),\ A_m = max\ A_k\ ,\ k \in 1, \ldots, K \\ &LSE[logA.] = log(\sum_{k=1}^{K} A_k) = logA_m + log(\sum_{k=1}^{K} e^{logA_k - logA_m}) \end{aligned} \tag{4.13}$$

This changes the GMM log likelihood function (see A.34 for derivation) such that the final loss function is this expected negative log likelihood:

$$-L(Z|\pi, \mu, \Sigma) = -\frac{1}{n} \sum_{i=1}^{N} logA_m + log(\sum_{k=1}^{K} exp(logA_k - logA_m)) \tag{4.14}$$

The "log-sum-exp" trick is used to avoid numeric underflow and helps improve the numeric stability when computing the log-likelihood of the GMM whose Gaussian mixture components easily underflow.

Training: Teacher Forcing

The MDRNN was trained for 20 epochs like the VAE, but the training data consisted of rollouts from both a random policy and the good RL controller agent in the original paper. *Teacher forcing* is used to quickly and efficiently train the LSTM by using ground truth latent observations from a prior time step as input. Formally, teacher forcing is a procedure in which the model receives the ground truth output z_t as input at time $t + 1$, instead of using the previously predicted output $\hat{z}_t$. Unlike the original authors, we do not store a precomputed set of μ_t and σ_t for each of the frames when training the MDRNN for efficiency. We use the original frames in our training data and use the VAE during training to encode the frames into the parameters of our latent space distribution and sample $z_t \sim N(\mu_t, \sigma_t^2 I)$ each time we construct a training batch, which helps prevent overfitting the MDRNN to a specific sample z_t.

4.5 Control: Plan with RHEA Policy Search

This section will describe components and extensions used to enable planning in our world model approach. We take point of departure in the vanilla RHEA algorithm described in the background chapter. Please refer to section 2.4.3 for a detailed description of the RHEA and RMHC planning algorithms.

4.5.1 Simulated Environment

One of the critical components in planning is the usage of a forward-model that enables evaluation of simulated trajectories. Thus, to enable planning in the model, we also need to support the following operations:

- consecutive transitions in latent space (step function)
- rollbacks to earlier latent states (evaluation)
- consecutive synchronization of world model and real environment

The step function is supported by the MDRNN and is a critical component, as it enables the agent to perform state transitions in the latent space and get feedback based on a given trajectory. Without it, this may prevent the agent from planning altogether. *consecutive synchronization* and *rollbacks* are mechanisms that are not supported out-of-the-box but are two critical operations necessary to do planning in latent space. To ensure that the model is continuously synchronized with the real environment, a procedure is needed to update the current latent space as we step in the real environment. If the latent space is not synchronized, the agent may falsely do planning on a latent space representation that does not capture the current state. Consequently, the trajectory produced may not apply well to the real environment and result in poor planning.

In our case, *continuous latent state rollbacks* is necessary for planning, as it enables continuous evaluations of simulated planning trajectories. If this is unsupported, the agents may at most evaluate one trajectory per game tick, since it is unable to rollback to an initial state and evaluate other planning trajectories. Consequently, being limited to a single planning trajectory may lead to poor planning, as the agent cannot sufficiently explore the policy search space for promising simulated plans. Thus, a simulated environment was implemented to support the above operations by encapsulating the MDRNN and VAE in a simulated environment.

The simulated environment acts as an interface to the world model that enables online planning in latent space. It has a step function $a, z, h \rightarrow z', h', r, d$ that takes an action, *latent state* and *hidden state* and returns a predicted next latent state, next hidden state, reward and terminal z', h', r, d. By default, the simulated environment does not keep track of the next latent z_{t+1} and hidden states h_{t+1} unless the simulated environment acts as the real environment. Instead, they are tracked by the callers of the step function. Since they are self-contained, the simulated environment has all it needs to generate next latent state and reward predictions, which enables the support of latent space synchronization and rollbacks.

Latent space synchronization

For *latent space synchronization*, a *game controller* starts by defining the initial latent and hidden state $\{z_0, h_0\}$, consisting of a zero vector for the hidden state and a VAE encoded latent state of the initial game state (frame). The game controller then proceeds to keep track of the latent and hidden state, by simultaneously stepping the real environment and simulated environment with the same action together with the current states at time t, $\{z_t, h_t\}$ for the simulated environment. The returned $\{z_{t+1}, h_{t+1}\}$ from the simulated environment overwrites the current $\{z_t, h_t\}$ and this loop repeats until the game ends. The persisted latent and hidden state $\{z_t, h_t\}$ in the game controller is then provided to the planning algorithm as the initial state.

Latent state rollbacks and planning flow

Similar to the game controller, RHEA uses the simulated environment. Instead of synchronization, the planning algorithm uses it as a forward model to simulate trajectories and perform rollbacks on $\{z, h\}$. When RHEA evaluates a population, the initial $\{z_0, h_0\}$ from the game controller is copied to each individual and assigned as current $\{z_t, h_t\}$. When evaluating an individual, each of its planned actions is stepped in the simulated environment. While stepping, the current individual states $\{z_t, h_t\}$ are updated with the ones returned from the simulated environment until all actions are exhausted. A rollback is performed when all individuals are evaluated (and evolved) where the initial $\{z_0, h_0\}$ is copied and assigned as current $\{z_t, h_t\}$ for the next-generation population. This process repeats until the number of generations is satisfied (see figure 4.8).

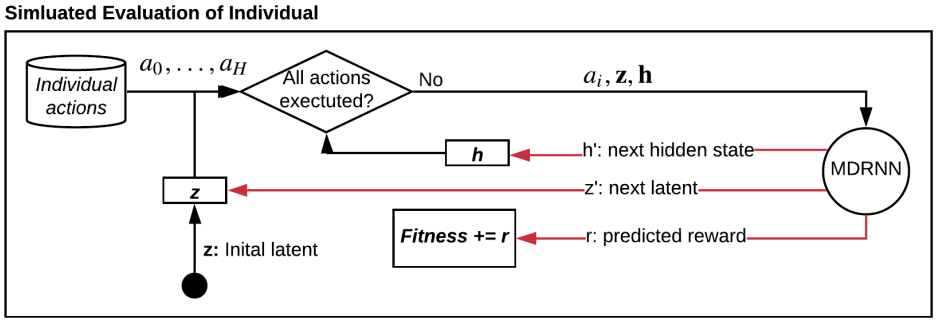


Figure 4.8: Planning with independent states and rollbacks. Each RHEA individual is given an initial $\{z, h\}$ and is updated at each time step up to a horizon H. The process is repeated at each generation

Concurrent Evaluations

Since the latent and hidden states are independent and fully decoupled from the simulated environment, it is possible to use the step function concurrently without the next latent state, reward predictions and hidden state being affected by concurrency issues. Thus, to speed up the planning process of RHEA, we parallelize the evaluation step such that each individual in the populated is evaluated concurrently, which led to significant planning time reductions.

Custom Action Sampling

In order to avoid accelerating and breaking at the same time, we modified the sampling function such that the acceleration and gas is determined by a number uniformly sampled between -1 and 1. If the number is between -1 and 0, the absolute value of that corresponds to the braking component and acceleration is set to 0. If the number is between 0 and 1, the number corresponds to the acceleration and braking is set to 0. Further, we cap the acceleration at some threshold (e.g. [0, 0.3-0.5] instead of [0, 1]) to avoid the car from quickly reaching very high speeds where it has less road grip and becomes more likely to drive into the grass.

4.5.2 Extensions to RHEA and RMHC

Shift Buffer

Shift buffer is a population management technique used to avoid repeating the entire search process from scratch at every new game tick. For this, we keep the final population evolved during one game tick to the next. Since the first action of the best elite is executed, all first actions from the population are removed and a new random action is appended at the end. Arguably, this may help retain planned policies from previously instead of starting from scratch. Thus, it is interesting to see how shift buffer affects the planning performance in car racing.

EvolutionHandler

An important extension to RHEA and RMHC is the EvolutionHandler. Instead of hardcoding the concrete evolution operators in the planning algorithms (i.e., selection, crossover and mutation), the operators are standardized in the evolution handler. This allowed us to write common evolution operators such as mutation once for RMHC and RHEA, to avoid code duplication. Secondly, by abstracting the evolution operators away from the planning algorithms, one may easily replace and extend with other operators, which allowed us to quickly see the effects of different operators.

Finally and most importantly, the ability to dynamically switch between different evolutionary operators was a crucial feature, since this made it possible to do automated NTBEA parameter tuning. Without the ability to dynamically interchange evolution operations, we cannot use NTBEA for efficient parameter tuning, since it depends on the ability to pick different parameter configurations. Namely, it is limited to simple typed parameters such as the horizon length, generations, or population size, thus reducing the likelihood of finding a somewhat optimal parameter configuration. The final implementation of the evolution handler can be found in *evolutionhandler.py* file in package *tuning*.

Macro Actions and MC Rollouts - Discontinued Extensions

Initially, we included Monte Carlo rollouts (MC rollouts) and macro actions as additional extensions but these were discontinued due to poor results. The motivation for using MC rollouts was to approximate the expected reward beyond the planning horizon by random sampling. However, MC rollouts yielded poor reward predictions, which might be due to the model not being able to look into the future beyond the horizon. Also, MC rollouts made the evaluation process much slower.

Macro actions were implemented to speed up gameplay and because the agent most likely wants to be able to repeat its previous policy without replanning in certain scenarios (e.g. keep driving forward on a straight road). The motivation was to repeat a planned action n number of times before executing another planning step. While the approach seems to speed up the game, the agent's performance dropped significantly when using a large macro action. For instance, if the planned action was to turn left, repeating this n many times would result in the car drifting out of control. Noticeably, the agent performed best with small macro actions (i.e., 2-3), which defeated the purpose of using macro actions although it might be interesting for future work.

Chapter 5

Experiments

This chapter will describe the primary experiments and the results produced in this thesis. All experiments were executed on a desktop machine with an AMD Ryzen-7-3700X CPU with 8 x 4.2 GHz cores and up to 16 logical processors. The machine also has 32 GB RAM and used Nvidia RTX 2070 Super with 8 GB VRAM as GPU.

The experiments are grouped into 3 groups with the ultimate purpose of finding a world model and planning algorithm configuration that together forms an agent that may perform well in the car racing game.

The first group of experiments finds a preliminary set of world wodels that are representative of the dynamics in the car racing environment. This is crucial, since a world model is used as a forward model in the planning algorithms and it strongly affects their overall planning performance.

The second group of experiments uses a subset of the world models based previous group of experiment. Each world model is then combined with a planning algorithm where the parameters are tuned with the N-tuple bandit evolutionary algorithm (NTBEA). Each combination of world model and planning algorithm (agent) are then benchmarked with the preliminary and tuned parameters. We run the agents 10 times on a simple track with no extreme turns and 10 times on random tracks where the performance of an agent is the mean reward of the runs. The choice of tracks enables the ability to compare how well the agents perform on easier tracks and how well they perform on difficult tracks, thus showcasing the overall capabilities of the agents.

The third and final group of experiments investigates how different horizon sizes, shift buffers and using the parameters from the best model on other models affects the planning performance in the car racing game. Each configuration is then evaluated 10 times on the car racings tracks where the agent's performance is the mean of the runs.

5.1 Preliminary Planning Experiments

This section describes the experiments conducted to find a preliminary set of world models useful for planning and describes how they are evaluated.

Setup

Before running any experiments, it is critical to define a set of repeatable and consistent test cases with well-defined measurement factors that describe the model's ability to represent the essential dynamics of the car racing game that the planning algorithms need during planning.

In this regard, 3 measurement factors are established:

- VAE encoding of observation and how well information is preserved.
- Rank order of reward signal
- Reward signal strength

The VAE represents the sensory input of the model by encoding an observation into a compact latent vector that the MDRNN uses to predict how state transitions behave. However, when compressing an observation, some information is lost due to the reduction of dimensions. The loss of information may negatively impact the model's ability to model the dynamics of the real environment. Hence, the model's performance is very likely dependent on the VAE's ability to maximize the information retained when encoding an observation, which is done by minimizing the reconstruction error during training. When testing a model, we observe the quality of the reconstructions next to actual frames, to give an overall estimate on the VAE's ability to retain information.

The rank order and strength of the reward signal produced by the model has a significant impact on the final trajectory produced by the planning algorithms. Even though the rewards predicted by the model may differ from the rewards of the real environment, the rank order of the rewards must be preserved by the model. Incorrect rank order of rewards may produce very poor planning trajectories such as driving off-road due to false-positive reward signals. The strength of the reward signal may help reinforce the rank order, such as perfect driving, producing larger positive rewards in comparison to semi-perfect driving.

The measurement factors are observed in 13 consistent and repeatable test cases that represent different driving segment scenarios during car racing:

- Forward drive on road: slow, medium, fast
- Drive on grass
- No movement
- Perfect turns: left, right, u-turn, s-turn
- Failed turns: left, right, u-turn, s-turn

Arguably, since the tracks are randomly generated, these test cases may not cover all different scenarios a car may be exposed to. However, the test cases do provide a general approximation of the scenarios, which allow us to measure the quality of our MDRNN dynamics model. The forward drive tests measures its ability to represent the reward signal when driving forward at different speeds and the reward rank order when driving slow and fast. We want to ensure that driving on the road yields a positive reward signal and that the magnitude of the reward signal is greater when driving fast as opposed to driving slow. The drive-on-grass tests measure the model's ability to punish bad driving by producing negative rewards when driving off-road. The no-movement test measures if the model punishes the agent for staying still and the turn tests measure the model's reward signal when completing turns or failing to complete them.

Finally, the model tests do not use any planning agent to control the vehicle but instead use hardcoded action sequences. This ensures that the test results may be successfully reproduced, since the car driving is made deterministic and repeatable.

Preliminary planning tests

The above tests partly describe the quality of the different models. However, what finally matters is how they perform as forward models in a planning algorithm. Due to time constraints, we did not perform exhaustive parameter tuning and planning tests for all the models produced. Instead, we ran preliminary planning tests based on the same forward and corner test scenarios with RHEA and RMHC. The total sum of rewards across these tests is used as an estimate of the model performance. The most promising models are then selected for full parameter-tuning and benchmarking. Below are the planning parameters used for preliminary planning tests. The parameters are set such that both RHEA and RMHC take approximately 1 second per planning step.

Preliminary Planning Algorithm Parameters

- Horizon: 10 (RHEA) - 20 (RMHC)
- Generation: 10 (RHEA) - 15 (RMHC)
- Mutation type: uniform
- Shift buffer: True
- Population: 8 (RHEA) - 1 (RMHC)
- Selection type (RHEA): tournament
- Crossover type (RHEA): uniform
- Genetic operator (RHEA): crossover + mutation

Preliminary Results

VAE Reconstructions Results

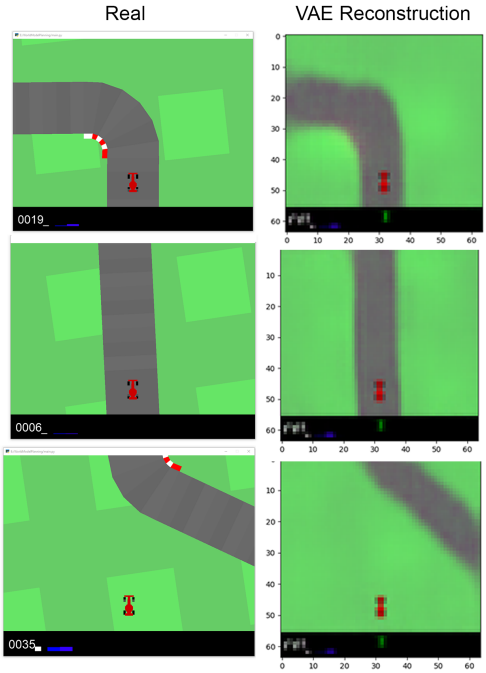


Figure 5.1: VAE reconstructions of frames from the real environment from VAE trained on random policy only. VAEs were in general able to reconstruct the original frames but the quality of the reconstructions highly depends on the samples used to train

Almost all VAEs trained were able to reconstruct the input frames with close similarity whenever the car was on the road and off-road, as seen in figure 5.1. However, using rollouts solely based on a good policy resulted in the VAE not being able to reconstruct scenarios when driving off-road. Hence the car was always somewhat on the road, as seen in figure 5.2. This is likely due to the lack of observations where the car was driving off-road and this may impact how well the agent plans.

Alternatively, using rollouts solely based on random rollouts, the VAE was able to reconstruct off-road an on-road scenarios with close similarity to the input. Using a random policy to sample the rollouts may result in higher diversity in the rollout data, as the car may drive on and off the road.

The samples used to train the VAE seem to lay the foundation of how well it can encode and reconstruct different scenarios in the environment. Thus, having a diverse set of rollouts that contain different scenarios of car driving yielded the best reconstruction quality.

By observing the loss of the VAE with respect to the different rollout datasets in figure 5.3, the general loss progression, as well as the final loss after 20 epochs, were very similar when using rollouts solely based on a random policy or rollouts based on a good policy.

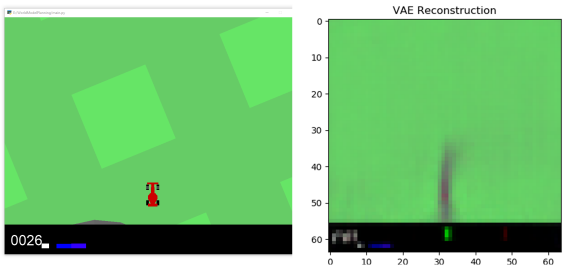


Figure 5.2: VAE trained on samples from a good policy only was unable to reconstruct frames during bad driving. Observe how the car is still on the road on the VAE decoded image (right) even though it is on the grass in the real environment (left). This is due to the lack of off-road driving observation. Hence a good VAE is trained on a diverse rollout dataset based on different driving behavior and scenarios

The VAE trained on random policy rollouts has a slightly smaller final loss of 23.3 compared to the VAE trained on good policy rollouts with a loss of 25.3. Training a VAE with both rollout datasets combined, resulted in a much higher test loss, which is likely due to the increased diversity of the data being used. Surprisingly, the overall reconstruction quality did not improve much when comparing it with the VAE trained on random policy rollouts. Thus, using 10k random policy rollouts to train the VAE was sufficient in representing the different scenarios of the car racing environment.

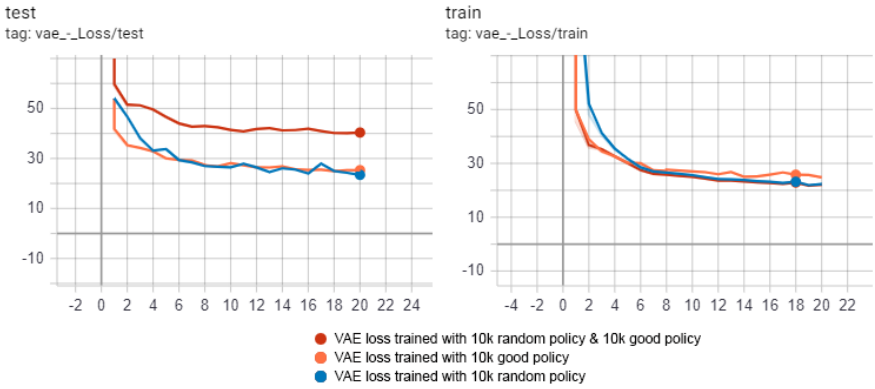


Figure 5.3: VAE test and train loss between VAEs trained with different datasets. The loss when using solely good or random policy rollouts was very similar. Combining both random and good policy data resulted in higher test loss, which might be due to the increased diversity of the data yielding a higher generalization error. Training a VAE with random policy rollouts was sufficient in representing different scenarios

MDRNN Parameters and Preliminary Planning Results

Model	Units	Sequence	Latent	Rollout Data
A	256	64	32	10k Random Policy
B	256	64	32	10k Good Policy
C	256	64	32	20k Good+Random
D	256	64	32	10k Random+5K Passive
E	256	64	32	20k Good+Random+5k Passive
F	256	64	32	10k Good+Random
G	256	64	64	20k Good+Random
H	512	64	32	20k Good+Random
I	512	64	64	20k Good+Random
J	512	64	64	10k Good+Random+10k Turns
K	512	64	64	20k Good+Random+10k R-Turns
L	512	500	64	20k Good+Random
M	512	500	64	20k Random
N	1024	500	64	20k Good+Random

Table 5.1: MDRNN Parameters (number of hidden units, sequence length and latent dimensions) and data set used for each world model. Model L performed best in the preliminary planning tests as well as in the model tests. Surprisingly, model F, which was less complex and trained on less data also showed promising tests results. The final models selected for benchmarking were model F, L and M

Model A-F

The initial model experiments (A-F) focused on using the same parameters presented in the original *world model* paper. Each model used 256 hidden units and a latent vector size of 32, with the only difference being that that we used a sequence length of 64 instead of 500 for faster training.

Model A (initial MDRNN) was trained on 10,000 random rollouts, similar to the MDRNN trained in the *World Model* paper. Disappointingly, we got poor results after running the preliminary planning and model tests with RHEA and RMHC. Model A was not performing well on any of the test cases. The minimum accumulated reward of completing all test cases is 205. Using model A, the total reward with RHEA was only 56 and surprisingly RMHC managed to get a total reward of 91, since it managed to get into road turns before ending in the grass.

One explanation of model A's poor planning results may be due to the fact that using rollouts based on a random policy does not sufficiently explore the state space so that the MDRNN may not properly capture the environment dynamics and rewards.

For example, good driving behavior may generate experiences required to produce strong reward signals within the world model such that the MDRNN learns to capture the fact that it is preferable to stay on the road. Also, the car usually ended up driving quickly into the grass rather than completing a turn or staying on the road in most of the rollout sequences. Thus, it was interesting to investigate an MDRNN model based solely on rollouts from a good policy (model B) to better explore the environment.

Model B was trained on good policy rollouts and yielded 120 in total test reward with RHEA, which is a 140% increase compared to model A. RMHC achieved a total test reward of 136, which is a 45% increase in performance. Both RHEA and RMHC did well on the forward drive tests but experienced problems on turns, which resulted in the agent driving into the grass. By visual inspection of the MDRNN reconstructions, it was obvious that the MDRNN still believed that it was on the the road in the simulated environment as shown on figure 5.4, despite it being on grass. A further inspection into the model tests revealed that the reward signal when driving on the grass near the road was similar to the reward signal when the car was on the road. This incorrect rank order of rewards may confuse the planning agent in finding a suboptimal trajectory and consequently lead it to belief that driving into grass is locally optimal.

Figure 5.4: Comparison between model B MDRNN reconstruction against real frame. The model still believes the car is on the road, despite it being on grass. This is very likely due to bad driving samples being underrepresented when using good policy rollouts during MDRNN training

A possible explanation to this anomaly is due to the under representation of rollouts that exhibit bad driving behavior, similarly to the problem with model A, that was trained solely on random data. Training the MDRNN on rollouts from only a good policy seemed to render it unable to produce a strong negative reward signal, which yielded poor planning results.

Model C investigates if the combination of random and good policy rollouts improves the planning performance of the agent. Model C was trained on 10,000 random and 10,000 good policy rollouts and the results seemed promising. Both RHEA and RMHC saw an increase in performance. RMHC only had a 3% increase in performance, while RHEA had a 17% increase in performance.

Interestingly, RHEA outperformed RMHC in many test cases, especially during turns where RMHC seemed to struggle in completing the turns. The poor performance of RMHC is likely due to it getting stuck in a local maxima when exploring its policy subspace, considering that only one individual is evaluated and randomly mutated in RMHC. RHEA uses a population to better cope with local maximas, which helps promote continuous diversity in the policy subspace of the population. Despite the improvements with model C, the overall planning was still not sufficiently good as both RMHC and RHEA were having trouble in completing turns and manually adjusting the planning parameters did not seem to yield any noticeable improvements compared to the baseline parameters.

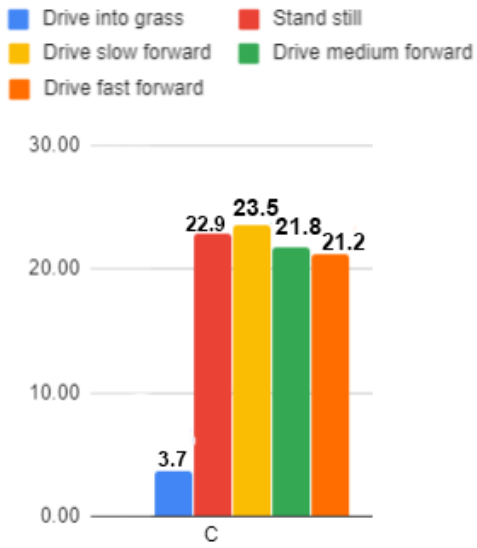


Figure 5.5: Histogram of model C average reward signal when driving with different speeds. Notice how the rank order is reversed such that driving slow had a higher reward signal compared to slow driving and staying still yields a high positive reward signal. This is a clear indication that something is wrong with the MDRNNs reward predictions

Further inspection of the reward signal showed that the model was unable to capture the rank order of the rewards, as shown in figure 5.5. The rank order of the speed was reversed, which means that driving slow would yield a higher reward signal than driving fast. Additionally, staying still produced a higher reward signal than driving fast. This was a clear indication that something was wrong with the reward signal and might be why the agent was still struggling to stay on the road.

Model F was trained with 5000 random and 5000 good policy rollouts, which is half the amount of data used for model C. Surprisingly, the reduction yielded a slight increase in planning performance for RHEA with a total average test reward of 175, compared to model C with 169 total average test reward. RMHC did also see minor improvements in planning performance with a total average test reward of 139, as opposed to model C with an average test reward of 148. Further inspection of the reward rank order showed that the reduction seemed to have corrected the incorrect rank order from model C as shown on figure 5.6.

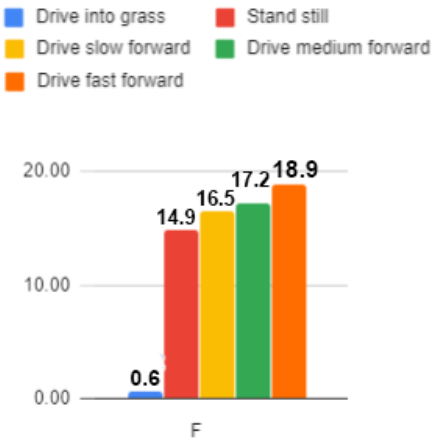


Figure 5.6: Histogram of model F average reward signal when driving with different speeds. The rank order of speed is in correct order (e.g. driving fast yields a higher reward than driving slow) by using half the rollout (5k instead of 10k). The corrected rank order explain the significant planning improvements in model F compared to model C whose rank order was reversed

Data: Diversity and Size

It seems that the rank order and quality of the latent predictions generated by the MDRNN highly depended on the diversity and size of the rollout data. Namely, we saw significant planning improvements when combining rollouts based on both a random policy and good policy. Intuitively, this makes sense, since it allows the MDRNN to learn the dynamics across more diverse rollouts that exhibit both good and bad driving behavior. Using only one of the two yielded poor results, since our MDRNN was unable to capture the full reward spectrum yielded by different driving behaviors.

Moreover, it was expected that using more rollouts (20,000) would allow the MDRNN to better capture the reward signal in the environment than using fewer rollouts (10,000). However, this was not the case in model F when compared to model C, as the increased number of rollouts failed to capture the reward rank order, as shown in figure 5.5. We were not sure why this was the case but suspected that the reason might be twofold.

Firstly, the increased number of rollouts might have reinforced the already imbalanced data by introducing more frequently occurring observations (e.g. straight roads and left turns) and undermining some of the infrequent observations (e.g. s-turns and right-turns). Secondly, the model might be insufficiently complex to benefit from more data to capture the reward signal accurately. However, we were not certain and ended up using 20,000 rollouts in our final model that was more complex and were successfully able to capture the reward rank order, although 10,000 rollouts might have been sufficient.

Model G-N - Adjusting complexity of MDRNN

Based on the previous results, model F was the best performing model amongst the initial models (A-F). Still, the overall planning results (particularly during turns) begged for improvements. Thus, it was interesting to investigate whether increasing the complexity of the MDRNN through its number of hidden units, latent size and sequence length would help improve the planning performance of the agent.

Latent size and Hidden Units

Model G - The premise of increasing the latent vector size from 32 to 64 was to investigate whether the increased amount of information retained in the latent vector would help improve planning. This proved to be true for RHEA that saw a 15 percent increase in performance (from 145 to 170), while RMHC did not see any improvements, despite correct rank rank order.

Model H - The premise of increasing the number of hidden units was to investigate whether a more complex MDRRN model would help better capture the dynamics and reward signal of the environment than our baseline MDRNN. Thus, we doubled the number of hidden units in model H from 256 to 512. The preliminary tests for RHEA yielded an average test reward of 178, which was 18% increase in performance compared to model C. Moreover, comparing model H to model F with an average test reward of 171, model H had a slightly higher average test reward of 178. RMHC also showed performance improvements when compared to model C and model F. Thus, by only increasing the number of hidden units, a substantial performance gain was obtained when compared to using a simpler model, which might indicate that we had previously used an insufficiently complex model that did not capture the environment dynamics and rewards.

Model I - significant performance improvements were seen with model I, which used the increased latent vector size and number of hidden units from model G and H. The RHEA average test reward improved by 30% from model C and 15% from model F while RMHC average test reward improved by 23% from model C and 22% from model F as showed on figure 5.7. By inspecting the rank order and reward signal on figure 5.8, shows that the reward rank order is correct after increasing the complexity of the model and the rank order is well separated (I and H).

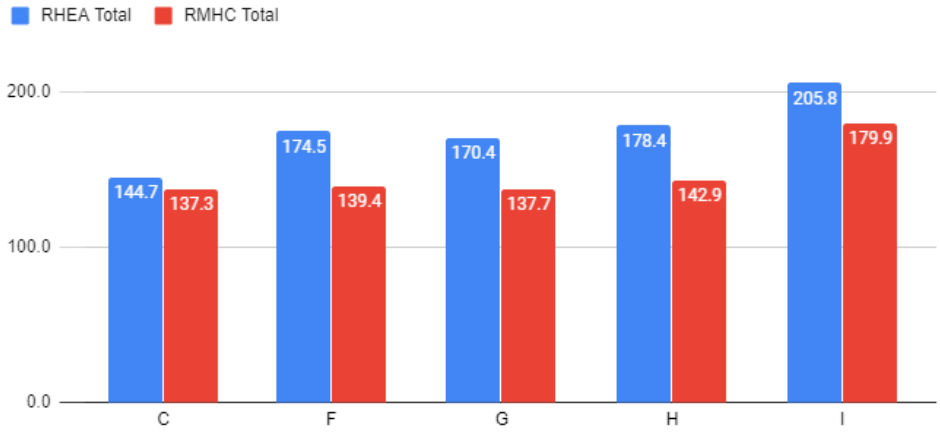


Figure 5.7: Histogram of model C, F, G, H, I: total average preliminary planning test reward. Increasing either latent vector size (G), hidden units (H) or both (I) in the MDRNN proved to increase planning performance when compared to the baseline models (C and F).

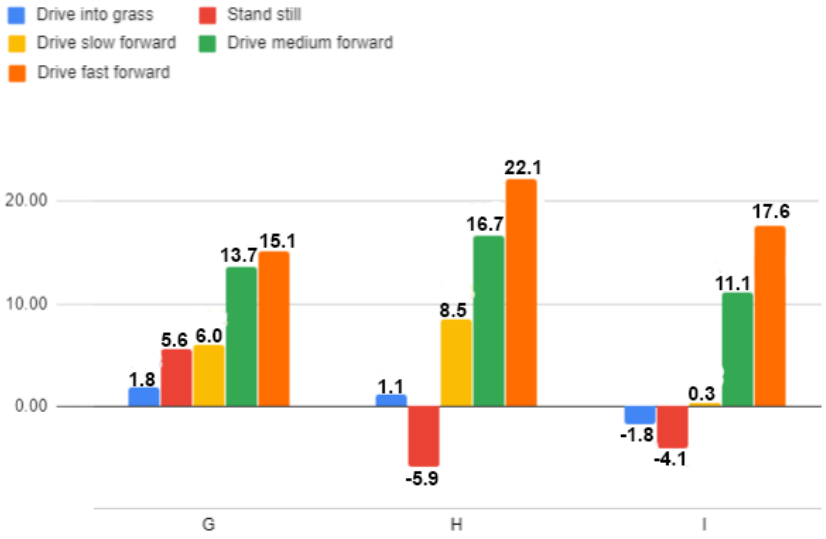


Figure 5.8: Histogram of model G-I average reward signal when driving at different speeds. Increasing either latent size (G) or doubling the number of hidden units (H) resulted in correct rank order of the rewards compared to model C, whose rank order was reversed. Increasing the number of hidden units resulted in a greater reward signal difference between the orders. Model I that combined both adjustments of latent size and number of hidden units yielded the overall best tests results amongst the three models

Model N - Too complex MDRNN

While 512 hidden units in the MDRNN proved to be a sweet spot with respect to planning performance, increasing the hidden units to 1024 (model N) resulted in a catastrophic 71% drop in planning performance for RHEA and 45% drop for RMHC when compared to model C and failed all tests. Further inspection of the reward signal showed an incorrect rank order of the rewards. Arguably, the performance drop is likely due to the MDRNN becoming too complex and hence having overfitted to the reward observations of the training data. Thus, increasing the complexity of the MDRNN may improve the planning performance of the agent, since it allows the MDRNN to capture the dynamics and correlation of the environment and reward better than a less complex MDRNN. However, if the MDRNN becomes too complex and is not somehow penalized, the MDRNN is more likely to overfit to the training data and generalize poorly, which is why we had to carefully inspect the test loss.

Model L-M - Increasing Sequence Length

All previous models (A-K) use a sequence length of 64, but further inspection revealed that a sequence length of 64 constitutes a very short road segment, as shown in figure 5.9 where we compare it with a sequence length of 500. Thus, we investigated the effect of using a short versus a long sequence length. The hypothesis was that a long sequence of frames should enable the MDRNN to better capture long-term temporal dependencies than using a short sequence. Namely, a long sequence may cover a full complete turn compared to a short sequence, which may only cover half a turn.

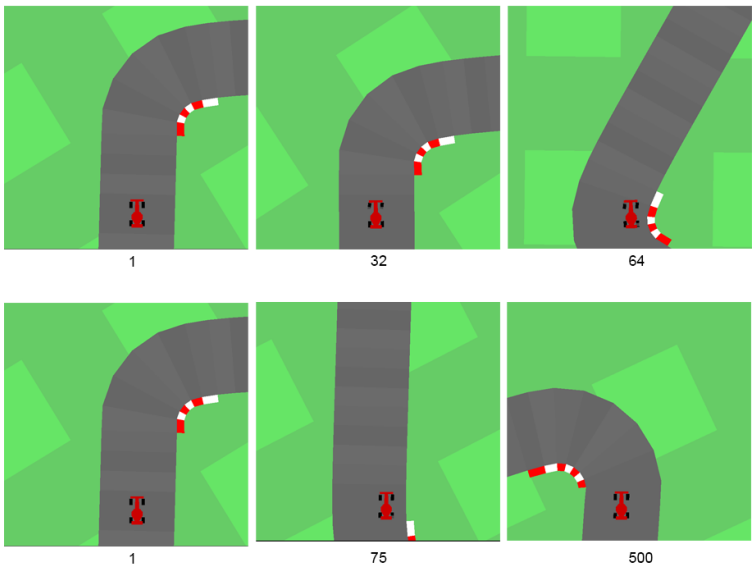


Figure 5.9: Car Position Image Sequence vs MDRNN Sequence Length. Increasing the sequence length allows the MDRNN to better capture long term temporal dependencies compared to a short sequence. This was demonstrated with model L, which yielded the best preliminary tests results amongst all MDRNN models

Hence, model L was trained similar to model I with 20,000 good and random policy rollouts, a latent vector size of 64 and 512 hidden units. The main difference was the increased sequence length of 500 instead of a sequence length of 64, which was the default in the PyTorch implementation we used. The increase in sequence length proved to yield astounding planning results. Both RMHC and REAH managed to complete all tests with minor errors. Recall, the minimum completion test reward was 205. With model L, the average test reward for RHEA was 273, which is a 47% increase in performance compared to model C and a 35% increase compared to model F. RMHC got an average test reward of 223, which is a 38% increase in performance compared to model C and model F. Thus, model L was the best performing MDRNN model amongst all the other models trained (A-N). The increase in sequence length seemed to help the MDRRN capture the temporal dependencies of the state transitions and reward signal, thereby explaing the significant increase in planning performance.

Model M - Random Trained MDRNN Based on all the previous findings, model M was trained with all the same parameters as the best performing model L but purely random policy rollouts only.

The preliminary results yielded poor results compared to all other models, which used a combination of random and good policy rollouts. However, the new MDRNN parameters did yield better results when comparing to the model A that was also trained on random rollouts as shown on figure 5.10. This might be due to insufficient exploration of the environment.

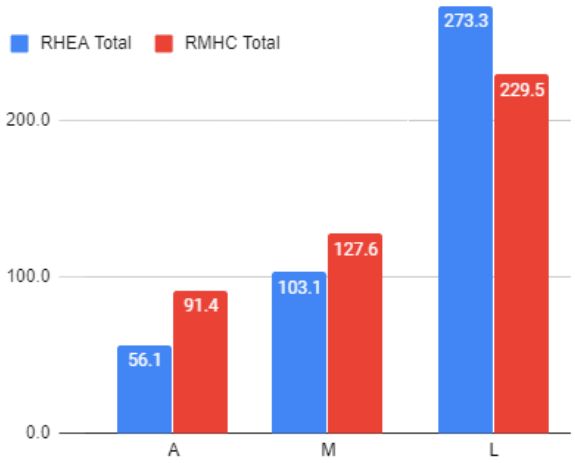


Figure 5.10: Histogram of model A, M, L: total average preliminary planning test reward. Despite using the MDRNN parameters based on the best model L, model M yielded poor results when trained on random rollouts compared to models trained with combined policy rollouts. However, model M still performed better than model A, which was also trained on random rollouts

Model Selection for tuning and benchmarking

Model	A	B	**C**	D	E	**F**	G	H	**I**	J	K	**L**	M	N
RHEA Total Reward	53	120	**148**	152	81	**175**	170	178	**206**	181	149	**273**	104	43
RMHC Total Reward	91	135	**137**	70	23	**140**	138	142	**180**	146	140	**230**	127	76

Table 5.2: Complete preliminary planning test results for each world model and planning algorithm (RHEA and RMHC). The table shows the total average accumulated reward across the test cases executed (forward drive, left turn, right turn, u-turn, s-turn). Models with 205 reward or more passed the test cases. Great improvements were seen when combining random and good policy rollouts (model C), increasing the latent dimensions and number of hidden units (model I) and increasing the sequence length from 64 to 500 (model L).

Based on all preliminary findings and test results in table 5.2, model **F**, **L** and **M** were selected for full parameter tuning and further benchmarking to see how well they would perform as forward models once optimized. model F was selected due to the good preliminary test results, despite it using a less complex MDRNN. model L was selected due to its top performance. Finally, model M was selected, despite its poor performance, since it was interesting to see how well a model trained solely on rollouts from a random policy would perform against a model that had access to rollouts that sufficiently explore the environment. The results are shown in figure 5.11.

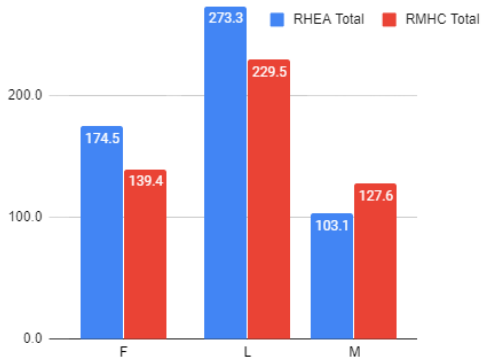


Figure 5.11: Histogram of average total preliminary test reward across model F, L and M. These are the models used later in the benchmark experiments. F was selected due to good preliminary tests results, despite having a less complex model. Model L was selected due to best test results. Model M was selected to test an MDRNN trained on random data only

5.2 NTBEA Parameter Tuning

The preliminary tests were used to obtain a set of candidate models that showed promising planning results. Using these models, this section discusses how the parameters of RHEA and RMHC planning agents are tuned in an efficient way, since the number of parameter configurations is infeasible to exhaust. Please refer to appendix A.4.1 for a detailed explanation of the parameters shown in table 5.3.

Parameter Search space

The parameter search space describes all configurable hyperparameters used in our implementation of RHEA and RMHC as presented in table 5.3.

Parameter	**Values**	**#Options**
Horizon	5, 10, 15, 20	4
Generations	1, 5, 10, 15, 20	5
Shift buffer	true, false	2
Fitness assignment	Total	1
Mutation type	uniform-1, uniform-All, uniform-subset	3
Population	1 (RMHC), 2, 4, 8, 16	5
(RHEA) Genetic operator	crossover, mutation, crossover+mutation	3
(RHEA) Selection type	uniform, tournament, rank, roulette	4
(RHEA) Crossover type	uniform, 1-point, 2-point	3

Table 5.3: Parameter search space for RMHC and RHEA. Note that last 4 parameter types are only applicable for RHEA while the rest is applicable for both RHEA and RMHC.

Tuning Setup

This section describes the final setup for tuning RHEA and RMHC with NTBEA.

We run NTBEA with the following parameters for each selected world model (F, L, M) for both the RMHC and RHEA planning agent:

- Evaluations: 100
- Mutation probability: 0.5
- k-factor (explore factor): 2.0
- n neighbours: 50

Compared to the number evaluations used in the paper (Gaina, Devlin, et al. 2020), the authors ran NTBEA on RHEA for 1500. We only perform 100 evaluations due to the time it takes for a single evaluation to complete.

When exploring configurations with large horizons, generations and population sizes (RHEA), an evaluation takes a couple of minutes to complete. We estimated that it takes approximately 3 hours to complete 100 evaluations for one world model on our local desktop. Hence, running 1500 iterations would take 2 days to complete, which sums up to 12 days when done for all models, which was not feasible given the time frame of the thesis. Arguably, the smaller number of evaluations reduces the likelihood of NTBEA converging to an optimal parameter configuration. However, according to the authors, it is still possible for NTBEA to converge in under 100 evaluations for some games. Thus, we found that 100 might be the bare minimum number of evaluations, although this should be verified.

Moreover, we do not run evaluations on full random tracks due to the immense time it takes for an agent to complete a single game when doing online planning (e.g. 15 min assuming each step takes 1 second in 900 steps). Instead, we reuse test cases from preliminary planning tests that expose the agent to scenarios of driving forward, left turns, right turns, u-turns and s-turns. Each case is executed in parallel and the final reward is the total sum of rewards achieved across the different test cases. This method may not provide the actual performance of a given parameter configuration. However, the test cases help reveal how well the agent is able to drive under a variety of scenarios, based on different segments of the track without taking too much time. Thus, the fitness returned per evaluation may be a simple approximation of how well each parameter configuration performs, which might be sufficient.

Tuned Configurations

Parameter	F	L	M
Horizon	10	10	5
Generations	20	15	15
Genetic operator	mutation	crossover + mutation	mutation
Shift buffer	false	false	false
Mutation type	all-uniform	subset-mutation	single-uniform
Population	16	8	16
Selection type	uniform	tournament	roulette
Crossover type	N/A	1 point	N/A

Table 5.4: Tuned Parameters for RHEA with NTBEA

Parameter	F	L	M
Horizon	5	20	5
Generations	20	15	15
Shift Buffer	false	false	false
Mutation Type	subset mutation	subset mutation	subset mutation

Table 5.5: Tuned Parameters for RMHC with NTBEA

Final parameters for each world model is presented in table 5.5 for RMHC and in table 5.4 for RHEA.

By observing the horizon parameter of RMHC and RHEA, it seems that both algorithms prefer to use a somewhat small horizon between 5 and 10 across all world models. A possible explanation would be that a smaller horizon allows the agent to exploit near-future trajectories that are more certain than exploiting a long uncertain horizon. Moreover, both RMHC and RHEA prefer to use a large number of generations (15-20). This makes sense, since it allows the agent to spend more time planning by refining its planning trajectories in the local policy subspace, thus increasing the likelihood of finding a good planning trajectory.

Surprisingly, both RHEA and RMHC prefer not to use shift buffer, which we initially assumed to be opposite. Shift buffer allows the agent to retain information gained from previous planning steps. However, it seems that the agents prefer to start from scratch at each planning step. A possible explanation is that previous planning steps may yield sub-optimal trajectories, which are propagated forward into future planning trajectories when using shift buffer, since it retains past plans. Thus, this may result in the agent getting stuck in the same sub-optimal trajectory space during the next planning step. By disabling shift buffer, all individuals are randomly initialized at each planning step, which may allow the agent to start at a different location in the trajectory space.

Another interesting finding is the choice of genetic operator in RHEA. Both model F and M only preferred mutation, whereas model L prefers crossover and mutation. Our initial expectation was that all models preferred crossover and mutation, since it may provide a more diverse population than just using either mutation or crossover alone. However, only using the mutation operator to encourage diversity may be sufficient for the agent in model F and M.

Finally, all models preferred uniform subset-mutation as a mutation operator for RMHC. uniform subset-mutation mutates a random subset of actions. This choice of mutation operator makes sense, since RMHC only has one individual mutated for each generation. Thus, RMHC prefers to evaluate mutated individuals that differ a lot from the current elite trajectory to increase exploration and promote diversity in the policy subspace, which increasing the likelihood of finding a promising trajectory, while avoiding getting stuck in a sub-optimal space.

Compared to RHEA, each model differs with its final choice of mutation operator, which is most likely because of the increased number of genetic operators at disposal and the diversity derived from using a population.

5.3 Planning Benchmarks

This section presents a final set of experiments conducted, in which the selected models **F** (least complex), **L** (best in preliminary tests) and **M** (trained on random policy) are benchmarked with RMHC and RHEA, based on different models and choices of NTBEA tuned parameter configurations.

Speed capping - Pre-Condition for all benchmarks

It was necessary to cap the maximum gas (acceleration) that can be sampled in our action vector from 1 to 0.3 across all benchmarks. Based on initial benchmark runs, the agents tended to prefer high gas values, which at first seems very good (i.e. driving fast is better). However, the agents would not brake when reaching a corner, which led to the car starting to drift and lose control, ultimately ending up in the grass due to sustaining a too high speed. A possible explanation is that the model is unable to capture the dynamics of the environment. Namely, we have not used any rollouts with an agent that drives sufficiently fast while reaching the grass and starting to drift so this physical phenomena is not captured by our MDRNN dynamics model. Consequently, when the agent plans a trajectory in the world model, the agent would believe that driving into a corner with high speed would not result in getting out of control, as shown in figure 5.12. Arguably, setting a cap on the speed may bias the experiments in favor of the agent but, not doing so, might prevent us from measuring the agent's planning ability when given a suboptimal dynamics model that does not capture this aspect of the physical laws that govern the real environment.

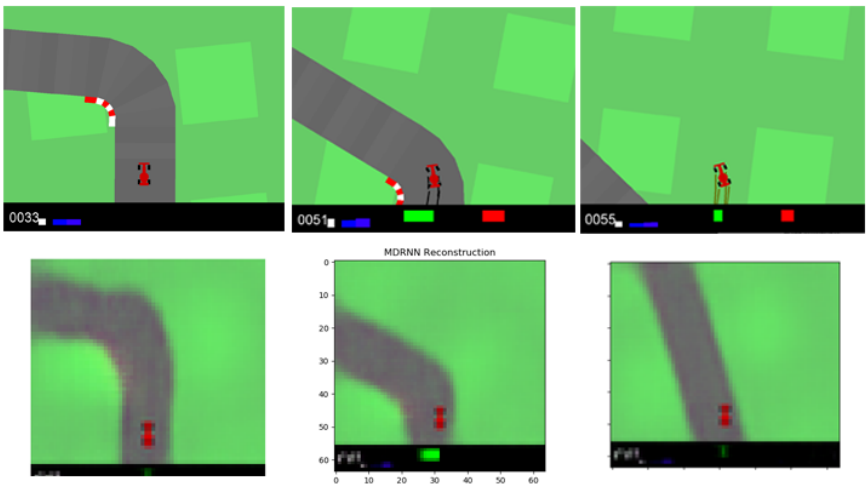


Figure 5.12: Agent (model L, RHEA) not braking at corner despite high velocity end up in grass (uncapped gas). The top row shows the actual driving in the real environment and the bottom shows the planned trajectory in the simulated environment. The car is still on the road within the model despite the high velocity. This points to the assumption that the model has not captured this dynamic

5.3.1 Tuned vs Preliminary Parameters

The first benchmark compares how well each agent (planning algorithm and world model) performs on a full track with the tuned planning parameters against the preliminary planning parameters. Each agent is run 10 times on a simple unchanged track and 10 times on random tracks.

5.3.2 Model F - Least Complex Model

Based on the results in figure 5.13, it is evident that the tuned parameters improved the planning performance for both RHEA and RMHC when driving on the simple track versus the preliminary parameters. The average reward for RHEA was increased by 13% when using the tuned parameters, whereas RMHC's average reward was increased by 40%.

However, when model F was deployed on random tracks, the preliminary parameters performed slightly better for both RHEA and RMHC. Using the tuned parameters, RMHC saw a slight decrease in average reward by 5%, while RHEA saw a decrease in average reward by 15%. Due to the nature of random tracks, it is difficult to control how the tracks are generated. Thus, the slight decrease in performance with tuned parameters may be due to generated tracks being somewhat more complex when tested with tuned parameters than during the preliminary parameters. However, when they are both deployed on the same simple track, it is clear that the tuned parameters improve the performance of RHEA and RMHC.

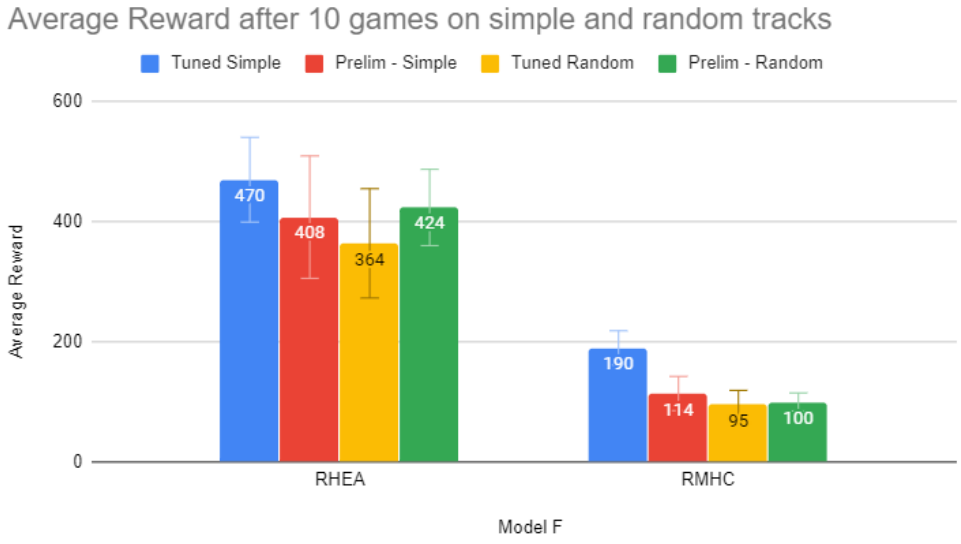


Figure 5.13: Model F histogram of tuned vs preliminary parameters. The error bars show the variance. The tuned parameters increased the performance of RHEA and RMHC when deployed on a simple track but saw a slight decrease when deployed on random tracks

When we observe the general performance of model F, RMHC was performing much worse than RHEA. We noted that RMHC had trouble exploring the trajectory space when the car was about to end up in grass. This is likely due RMHC getting stuck in a local optima trajectory space and failing to sufficiently correct its trajectory towards the road when it is at the edge of the road as shown on figure 5.14. Once the car was in the grass, the agent seemed to stay on the path and was unable to get back on the road, as it could not escape the local policy subspace in the simulated environment.

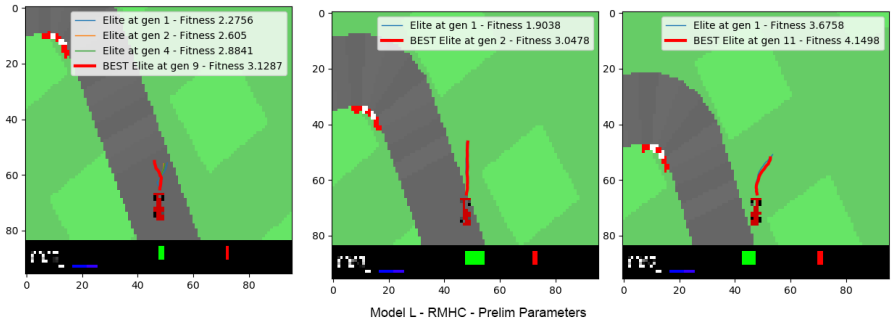


Figure 5.14: RMHC Car Trajectory with model F. Note that the car fails to correct the trajectory when it is about to drive into the grass. This is likely due to RMHC getting stuck in a local optima policy subspace

When using RHEA, the car was somewhat able to drive around the track and get around 300-400 in total reward out of 900. However, the agent usually failed when reaching complex parts of the track and ended up in the grass. Since the MDRNN is less complex (i.e., only 256 hidden units instead of 512) and only uses a short sequence of 64 (instead of 500) when trained, it may have trouble representing complex dynamics of the game. A long sequence length was shown to help the model capture long term temporal dependencies. Thus, when the agent plans within the simulated environment, it may have trouble representing and completing a trajectory that succesfully transfers back to the real environment.

The reason why model F yielded promising results in the preliminary tests cases may be due to the simplistic representation of the car racing game. The agent was only required to drive for short amount of steps to complete a given test, such as completing a left turn, which reduced the likelihood of driving poorly. The test cases only serve as an evaluation of agent driving behavior on simple hand-picked parts of the game tracks and it may be possible for an agent to get an excellent performance. However, when the agent is finally deployed on real tracks, the agent's weaknesses are made clear, despite it doing well in the predefined tests - it seemingly fails to generalize. model F may not complete whole tracks but it still proves that it can drive significantly better using RHEA than a random policy, despite its low complexity. However, the overall driving is far from human-level driving.

5.3.3 Model L - Best in Preliminary Tests

Based on the results in figure 5.15, the tuned parameters resulted in massive improvements in planning performance. The average reward of RHEA was increased by 47% on simple tracks and 67% on random tracks compared to preliminary parameters.

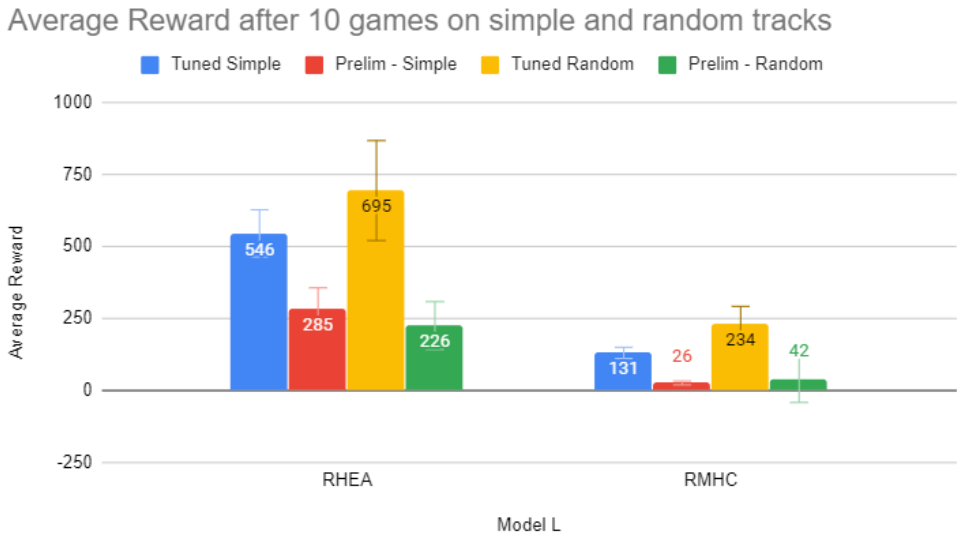


Figure 5.15: Model L Histogram: Tuned vs Preliminary Parameters. The error bars show the variance. The tuned parameters significantly increased the performance for RHEA and RMHC and the agent was sometimes able to complete tracks using model L with RHEA

RMHC's average reward was increased by 80% on random and simple tracks. Despite the great improvements, its performance was still poor, as it only got at most 234 in average reward when driving on random tracks. The agent failed mostly when driving into complex parts of the game (e.g. corners). Again, this is likely due to RMHC getting stuck in local optima policy subspaces and hence failing to escape a poor trajectory. RHEA performed remarkably well with an average reward of 695 out of 900 and managed to get a maximum score of 827. The agent was, at times, able to complete whole tracks if the population was initialized from a good starting point in the local policy subspace and the car was well positioned on the road throughout the game (i.e., not driving into the corner with bad initial trajectory), which is demonstrated in figure 5.16.

However, we still saw occasions where the agent would lose control when entering turns due to the high speeds it attains or when it ends up taking a bad turn and is unable to escape back onto the track. model L is far from a perfect world model and it does not reach human-level performance, nor does it par with the learned controller agent showcased in the world models paper. Regardless, the general performance of model L is promising and points towards evidence that it is possible to use RHEA for online planning on a learned forward model.

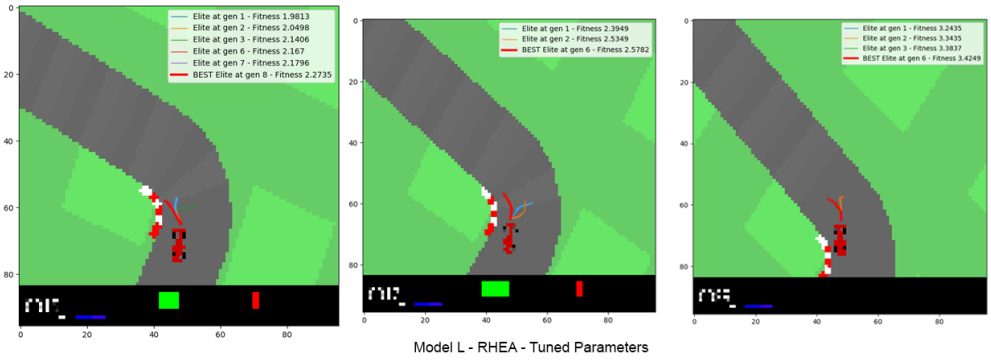


Figure 5.16: RHEA with model L completing left turn with shown trajectories. The best trajectory is marked with red.

5.3.4 Model M - Random Rollouts Only

Based on the results in figure 5.17, the tuned parameters did not seem to improve the planning performance when using model M. Surprisingly, using the tuned RHEA parameters saw a decrease in average reward of 30% on the simple track but a slight 18% increase on random tracks. RMHC did not see any performance gains with the tuned parameters.

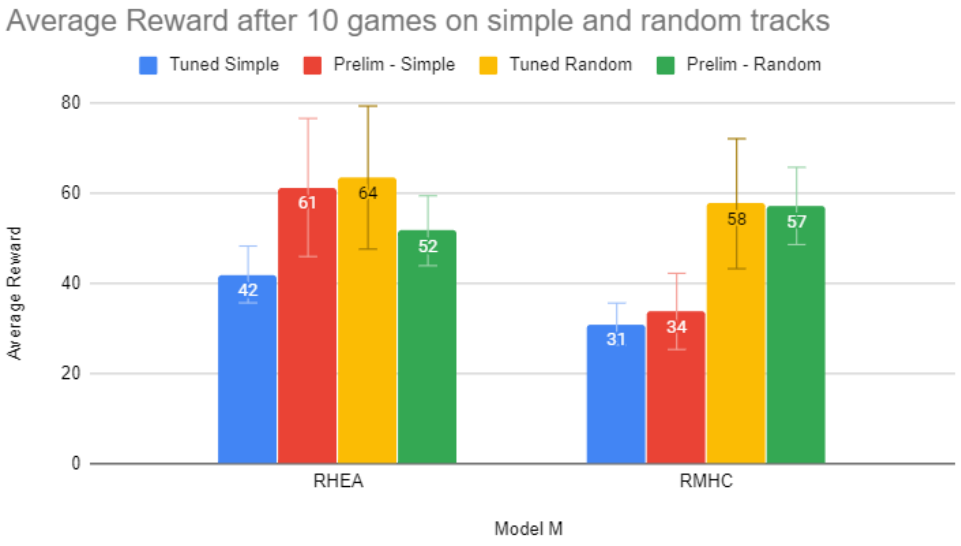


Figure 5.17: Model M Histogram: Tuned vs Preliminary Parameters. The error bars show the variance. The tuned parameters increased the performance but the model still failed as a forward model

Arguably, the two main reasons that help explain the abysmal performance of model M are the use of random rollouts only and the limited number of NTBEA iterations. The authors from (Simon M Lucas, Liu, and Perez-Liebana 2018) ran 1500 iterations when tuning and they mentioned that some games required the full iteration budget to converge to an optimal parameter configuration despite some games converging below 100 iterations.

Firstly, since we only run NTBEA for 100 iterations, the algorithm may not have converged to a somewhat optimal parameter configuration for model M. Thus, we are unable to see any significant performance gains with the tuned parameters. Hence, it would be interesting to see whether the parameter configurations suggested by NTBEA change for model M if run ran it with 1500 iterations. The second explanation relates back to the random rollouts used for training. As mentioned in the preliminary experiments section, the random policy only exposes the world model to random driving. Thus, good driving behavior might be underrepresented, making it hard for the MDRNN to produce an accurate reward signal required for planning. This limits the overall planning capacity of the model, despite tuning the planning parameters. If the model only represents the dynamics and reward signal from random driving, then it may only be able to produce reward predictions accordingly. The connection was somewhat clear when the agent decided to drive into the grass, despite driving on a straight road, as shown in figure 5.18. This was the exact same behavior observed when the rollouts were generated using a random policy where the car would suddenly turn straight into the grass. While doing so, the car was still able to collect tiles. Hence, a positive reward is obtained before ending in the grass. Thus, the positive reward may be associated with inappropriate actions such as turning hard left when there are no turns on the track.

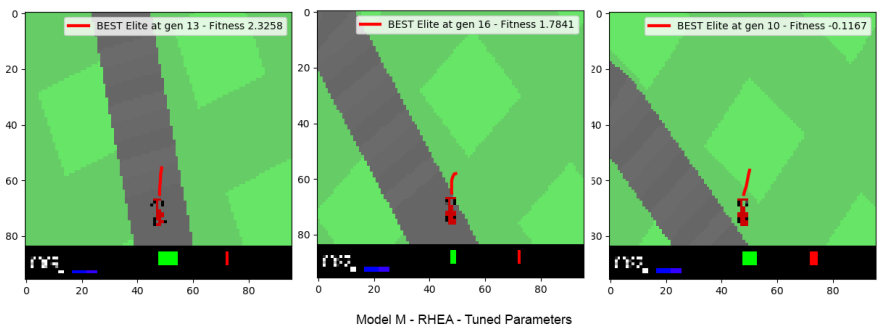


Figure 5.18: Using model M, we saw the agent having sudden urges to drive into the grass despite, driving on a straight road. There is a clear connection to how well the agent performs versus the rollouts used for training. Random rollouts may limit the planning capacity of a model and its ability to approximate real rewards

One way to improve model M may be to train the MDRNN with random rollouts and then use an iterative training procedure with the planning algorithm. Thus, the agent continuously drives and stores rollouts and use those rollouts to retrain the model. Assuming the agent can drive slightly better than the random policy, it may help generate new rollouts that improve the MDRNN and its ability to capture the environment dynamics. Thus, the MDRNN may slowly improve its reward signals from a gradually-improving planning agent.

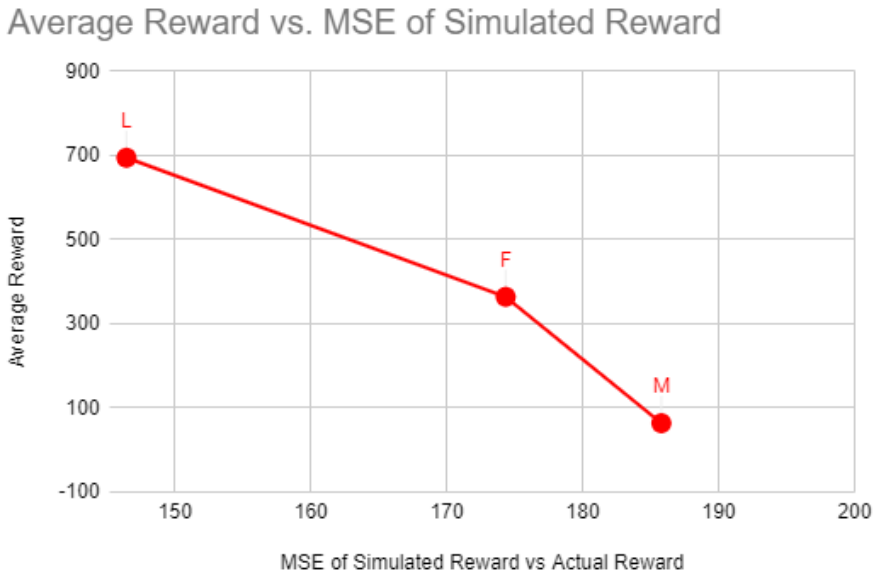


Figure 5.19: Total Average Reward vs Reward MSE: There is a strong correlation between the average total reward and the model's ability to predict the actual rewards. A smaller MSE yields a higher average reward (model L) compared to a high MSE that results in abysmal performance with model M

5.3.5 Total Reward vs Reward MSE

Based on the above planning results, it seems evident that model L is still the best forward model for RMHC and RHEA.

There is a clear correlation on how well a model performs as a forward model when comparing the mean squared error (MSE) of the models reward predictions versus the actual rewards as shown in figure 5.19. model L that performed the best also has the smallest MSE, whereas model M that performed the worst has the highest MSE. This was partially used to access the model quality based on how well it predicts the actual rewards. This is crucial for planning, as the predicted rewards are used by the agents to determine the planned trajectories, which are transferred back to the real environment. Poor rewards prediction may lead to poor reward signals and incorrect rank orders that result in poor planning as shown with model M.

5.3.6 Different Horizons - L and M

This section investigates how the agent performs with different horizons. Recall that the horizon defines how far an agent plans. We ran this experiment with model L and model M for RHEA and RMHC 10 times on random tracks. The results are shown in figure 5.20 where the top graph shows the average reward with different horizons for model L and the bottom figures shows it for model M.

It seems evident that model L with RHEA performs best with a horizon of

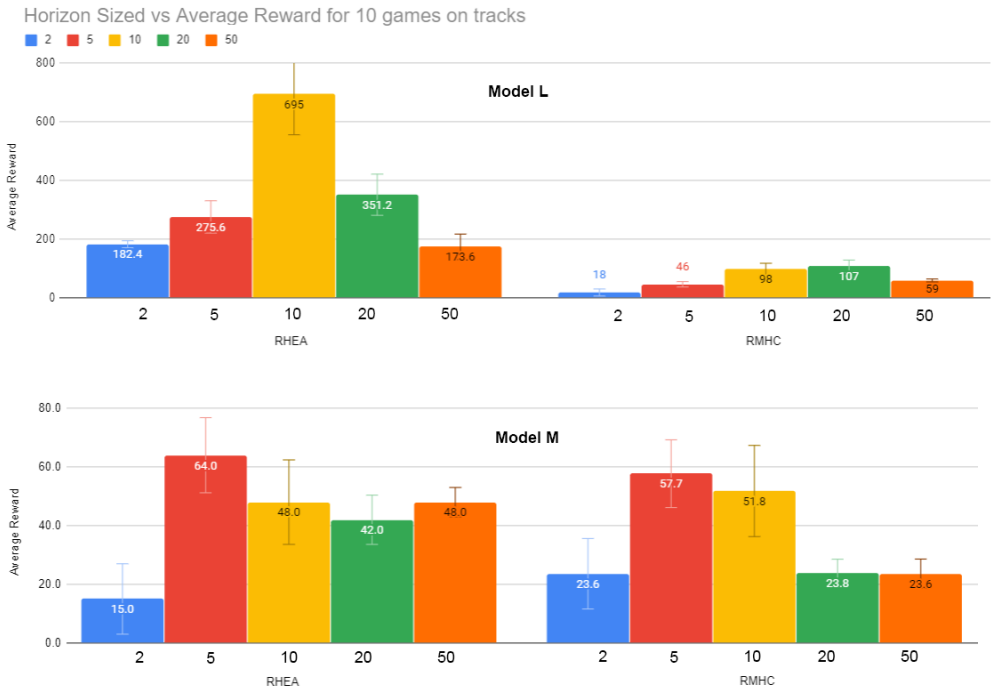


Figure 5.20: Histogram of average reward vs horizons for model M and L. Both models prefer a horizon between 10 and 20. A high horizon produce very uncertain trajectories due to noisy rewards and a small horizon does not sufficiently capture the direction of the car. Thus, we need to find a sweet spot that allows the agent to look far ahead, while being confident about the planned trajectory

10, whereas RMHC has a sweet spot between 10 and 20. Similarly, model M prefers a horizon of 5 with RHEA and between 5 and 10 in RMHC. Unfortunately, adjusting this horizon did not seem to improve the agent's planning performance, which is likely due to the limited planning capacity of model M as mentioned in the previous section.

Both models prefer a somewhat small horizon, which makes sense because using large horizons (e.g., 50) may result in very uncertain trajectories. The model only approximates how the future may unfold to a certain extent. Hence, unfolding far into the future may provide noisy rewards due to uncertainty, which we currently do not capture, since we only model the average reward signal. Using a small horizon (e.g., 2) would exploit a more certain near-future and thus produce less noisy rewards. Yet, a small horizon does not bring much information about whether the trajectory is on a good or bad path. This is similar to having an agent trying to plan as far as the car's tip. Consequently, the short-sighted agent may not be able to act in time before driving into grass.

Hence, we need to find a sweet spot where the agent can look far enough into the future, while still being confident that the trajectory is likely to occur. For model M and L, it is between 10-20 but this is not a fixed value, as other models may support larger horizons.

5.3.7 Model L Parameters in Model M

This section investigates whether it is possible to improve the poor planning performance of model M by using the tuned planning parameters in model L with model M. A hypothesis is that NTBEA may not have converged to an optimal parameter configuration for model M. Thus, using the best parameters in the best performing model may be closer to an optimal parameter configuration and improves model M's performance.

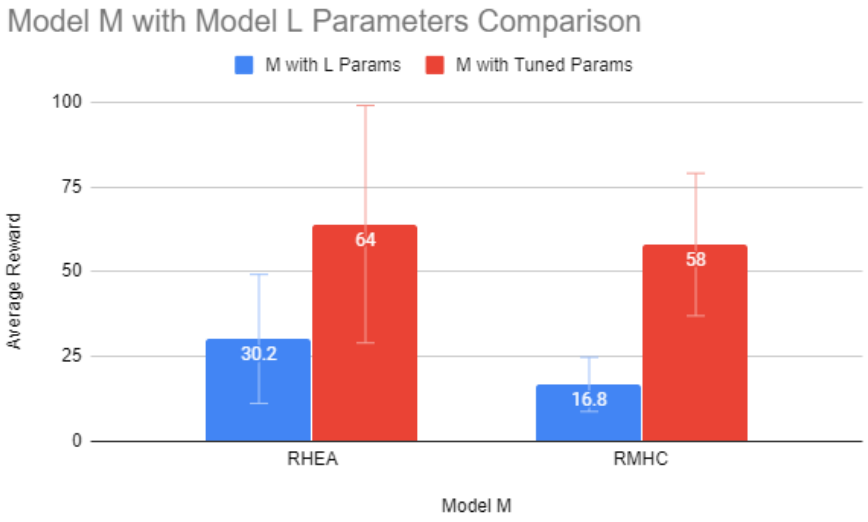


Figure 5.21: The histogram compares the average reward achieved with model M by using tuned planning parameters from model L for RMHC and RHEA. The error bars show the variance. Using the parameters from model L significantly decreased the average reward for model M

Based on the results in figure 5.21, using the planning parameters from model L on model M severely decreased the performance of model M. The average reward of RHEA was decreased by 53% and 69% for RMHC. Unsurprisingly, the tuned planning parameters for model M were expected to perform better, as they were tuned explicitly for model M. Using a parameter configuration tuned for another model that performs better, may not be a silver bullet that can drastically improve the model.

5.3.8 Shift Buffer - L and M

Recall that NTBEA prefers to disable shift buffer in all tuned models. We investigate the effect of enabling shift buffer and compare the results with model M and L that use tuned parameters. The experiments are run 10 times on random tracks for both world models and planning algorithms. The results are presented in figure 5.22 that shows the average reward when enabling or disabling shift buffer. The top graph shows the comparison for model M and the bottom graph shows the comparison for model L. It seems that enabling shift buffer reduces the performance of the planning algorithms for both model M and L. The performance drop is most significant when using RHEA compared to RMHC. When shift buffer is enabled with RHEA, the average reward is decreased by 30% for model M and 61% for model L. This reduced planning performance may help indicate why NTBEA preferred to disable shift buffer for all agents. Finally, the results may confirm our hypothesis that disabling shift buffer allows the agent to start at sufficiently random locations in the policy subspace. Thus, it avoids getting stuck in a sub-optimal space due to previously retained trajectories and increases the likelihood of finding a somewhat optimal trajectory.

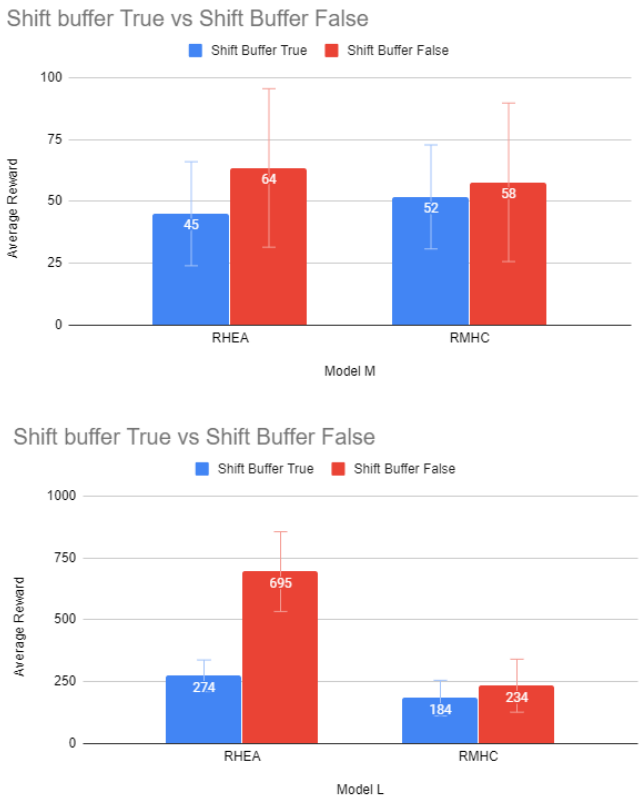


Figure 5.22: Histograms that compare average rewards with shift buffer enabled and disabled for RMHC and RHEA. Top figure shows comparison with model M and bottom figure shows comparison with model L where error bars denote variance. Disabling shift buffer yields a higher average reward than enabling it

Recent work in (Gaina, Devlin, et al. 2020) showed a preference towards enabling shift buffer in 17 out of 19 games. However, all these games are grid-based with discrete action spaces and is significantly less complex than the car racing game that has an infinitely large continuous action space. Thus, the agent may find it easier to plan in a discrete grid-based game, as the trajectory space is significantly smaller, which enables exploitations of past plans using shift buffer without the need to encourage more diversity to escape bad policies.

The car racing policy search space introduces a huge trajectory search landscape with local optimas. Consequently, if the agent always starts from the same local optima due to shift buffer, it will most likely stay in the same space of plans given the population is unable to explore other policy regions through sufficiently upheld diversity. However, this is not a certain conclusion, since the above only applies to the car racing game that is represented in a model where parts of the landscape may be non-representative or uncertain. Thus, it would be interesting to see whether enabling shift buffer would result in poor planning when using a perfect model of the game.

5.3.9 Summary

Based on the above experiments, we found that using an effective tuning approach, such as NTBEA, to tune the planning parameter configurations helped sufficiently improve the planning abilities of the agents to complete whole tracks. This was shown with model L using 512 hidden units, a sequence length of 500 and a latent size of 64 trained on a combination of 20,000 good and random rollouts.

Moreover, a simpler model F with 256 hidden units was trained on a combination of 10,000 good and random rollouts and achieved excellent preliminary test results but demonstrated mediocre driving skills. This makes sense, since the preliminary tests only represented a tiny segment of real tracks, which may not fully expose the weakness of a given agent's driving behavior.

Model M, which was trained with random rollouts only, failed drastically as a forward model, despite tuning the planning parameters, playing with different horizons and shift buffer settings. This is likely due to the capacity of the dynamics model being limited by the random policy, which insufficiently explores the environment, as argued with model A in the preliminary experiments.

Moreover, the MSE of rewards proved to give a reasonable quality estimate of the models. This was indicated to be true, since model L that performed the best had the smallest MSE, while model M, which performed the worst, had the highest MSE.

Further, the horizon benchmark showed that the models prefer to use a small horizon between 10 and 20. Horizons beyond this yielded poor planning trajectories due to the compounding errors from consecutive reward predictions. In contrast, short horizons fail to provide enough information on where the car is heading. Thus, a sweet spot is needed, such that the agent may plan far enough into the future, while still being confident about the planned trajectory.

Shift buffer showed promising results in the EA paper by (Gaina, Devlin, et al. 2020) but ended up significantly worsen our planning results. Arguably, this was likely due to the agent getting stuck in the same sub-optimal planning trajectory space from the previous trajectory retained by shift buffer.

Comparing RHEA and RMHC, RHEA managed to achieve a much higher average reward than RMHC. This makes sense, since RMHC only uses a single individual and is more prone to getting stuck in a local policy subspace. In contrast, RHEA exploits a population and may better cope with sub-optimal trajectory spaces than RMHC. Despite that, their overall planning performance is strictly dependent on the model's ability to capture the dynamics and rewards of the real environment and the planning parameters used.

A final comparison of the average reward when driving random tracks is shown in figure 5.6. The figure includes all benchmarked models with RHEA and other methods for comparison in the car racing game. It is important to note that we only performed 10 trials across the models due to time constraints, whereas the official game benchmark uses 100 trials.

Our best model L does not beat the model from the world model paper, nor does it beat human-level play. However, it does achieve a relatively high average reward and managed to beat traditional learned Deep RL methods (A3C, DQN). The findings prove that there is much potential in using RHEA for online planning on a learned forward model.

Method	Average Score
Model M (Random policy rollouts)	64 ± 5
DQN	343 ± 18
Model F (random/good policy rollouts, 256 hidden)	364 ± 101
A3C (continous)	591 ± 45
A3C (discrete)	652 ± 10
Model L (random/good policy rollouts, 512 hidden)	695 ± 13(**77%**)
Risi world model	903 ± 72
Ha world model	906 ± 21(100%)

Table 5.6: Our RHEA agent is able to achieve an average score of 695 ± 13 across 10 random trials compared to other methods with 100 trials. We do not solve the task of +900 average score but are able to beat traditional Deep RL methods (DQN, AC3) with an average score of 591-652 where model L achieved an average score of 695

Chapter 6

Discussion

This section will critically assess and discuss some of the shortcomings of our approach and experiments along with the lessons learned in our work.

6.1 Model (MDRNN)

In this thesis, we have realized that a planning agent depends strongly on access to a somewhat perfect world model. Namely, the model used by the planning agent needs to model the reward so that the rank order is respected. In our case, the modelled reward signal must be higher when the agent drives fast on the road, than driving slowly or driving on the grass. Thus, a planning agent needs access to a world model that has been trained on representative observations from the environment. This begs the question of how one may use an alternative rollout policy than a random policy to sufficiently explore the real environment state-space.

Interestingly, the full world model in the original approach of *World Models* (Ha and Schmidhuber 2018a) uses the actual reward signals to train their controller agent, which we find to be less puritan, since that means they do not in fact train a fully learned model. Recall, in model-based RL, a model is composed of an estimated transition dynamics function and reward function. In our approach, we use a fully learned model by extending the MDRNN transition function with a reward signal. However, we realized that the original MDRNN does not work, since it is trained on random rollouts only. Namely, evidence points to the fact that the original MDRNN is able to predict future latent states when inspecting the reconstructions but the reward rank order seems to be entirely off.

For this purpose, we relied on their pre-trained controller agent that exhibits good driving behavior, as the policy used to generate additional rollouts that sufficiently explore the environment state-space to learn a good reward function. Thus, we are successfully able to respect the rank order of rewards in the simulated environment when compared to rewards in the real environment, which means our MDRNN facilitates online planning in latent space.

Another concern is that the MDRNN seems to struggle with predicting multiple steps into the future, which may be improved using multi-step predictions. The MDRNN is only evaluated for its ability to predict the next latent state given the current latent state (i.e., single-step prediction objective). Thus, we do not directly measure its ability to predict multiple steps into the future in our objective function, which may lead to compounding future prediction errors during planning.

A learned lesson comes from *PlaNet* that suggests using both a stochastic (e.g., a latent state) and a deterministic component (e.g., a recurrent hidden state) in the dynamics model for successful planning. They argue that purely stochastic transitions make it difficult for the transition model to reliably remember information across multiple time steps. Thus, a deterministic sequence of hidden vectors $\{h_t\}_{t=1}^{T}$ is used to allow the model to access, not just the last state, but all previous states deterministically. Arguably, our MDRNN has such a hidden state that summarizes the past, but information that goes further back into the past may be forgotten as it unrolls forward in time. Thus, we also assume the model to be non-

Markovian, since we predict z_{t+1} from z_t and h_t that we assume roughly "summarizes" prior latent states $z_1, ..., z_{t-1}$ like in their non-Markovian RSSM model. This assumption could influence the planning results in our work when looking at Markovian alternatives to RNNs.

6.2 Planning (RHEA)

We found that both the RHEA and RMHC planning algorithms were able to find policies in the continuous action space of the car racing environment that made our agent exhibit relatively good driving behavior. However, both algorithms are subject to local optima, regardless of whether we use a hill climb or evolutionary algorithm procedure to maximize the fitness obtained from different sequences of policies in the local policy subspace. The hill climb procedure was especially prone to getting stuck in a local maxima when trying to find a good policy. In contrast, the evolutionary algorithm procedure was able to escape this by using a population with multiple individuals scattered around the random policy search space.

It was very noticeable that the RMHC planning agent depended strongly on the initial seed of the first random plan generated. However, it was able to drive pretty well when starting in a good subspace of policies and was very efficient, since it only uses one individual that is mutated and replaced repeatedly. On the other hand, the RHEA agent was able to obtain more consistent scores with fewer deviations, since it is less prone to getting stuck in a locally optimal policy subspace due to its large population. For both of them, it proved very important to use a mutation operator that encourages enough diversity to avoid premature convergence. Finally, their performance greatly fluctuates, depending on the choice of model used to do simulated planning, which stems well with the fact that successful planning depends on access to a good model.

However, our world model does still not entirely capture the dynamics of the environment. For example, we currently have to set a threshold of 30-50% on the gas used by the planning agent. This is because both planning agents exhibit a preference towards driving at such high speed that they end up drifting at corners. Arguably, our world model has not been trained on rollouts, in which the rollout policy triggers these kinds of scenarios. Namely, the policy we use exhibits good driving behavior, meaning it stays on the road and maintains a decent speed at corners. Thus, our world model does not capture traits like the physics of the environment. Arguably, this begs the question of how to obtain a better exploration of the state space.

In terms of parameters, we realized that our agent is somewhat short-sighted, since we maximally use a future horizon of 15 steps. According to related work, short-sighted agents are typically an issue in most model-based approaches. *Dreamer* suggests using a learned value network to help approximate the value of future states beyond the horizon.

Finally, it was a revelatory experience that our NTBEA parameter tuning showed it was worse to use shift buffer in our planning agents. Normally, this is highly advocated in most RHEA papers but they mainly focus on grid-based games, which use discrete action spaces. Arguably, we rely on as much diversity as possible in our trajectories to help escape bad planning policies in the infinitely large continuous action space of the car racing environment. Thus, by using shift buffer, the agent may get stuck in the sub-optimal planning trajectory space from the previous trajectory retained by shift buffer.

6.3 Challenges

We struggled a lot with training our MDRNN, which experienced negative loss values during training of the GMM output that predicts the distribution of future latent states as a Gaussian mixture. Namely, we had issues with likelihood probabilities underflowing and loss values going towards negative infinity due to the presence of singularities. Thus, we had to carefully reset any mixtures of the distribution that collapsed towards infinite values with a small variance. Further, we had to employ a mathematical "logsumexp" trick to keep the Gaussian mixture values from underflowing. These analytical objective functions proved to be very hard to work with when doing gradient-based optimization of the negative log-likelihood function. We spent a significant amount of time troubleshooting the loss of our Gaussian mixture model output, which might have been avoided if we had used a gradient-free evolutionary training procedure.

Secondly, we spent a lot of time tuning the complexity of the MDRNN model and using different rollout training data sets to obtain a model that respects the rank order of real rewards to do successful planning. Arguably, we rely on a good reward signal in the simulated environment when doing online planning, as opposed to learning approaches that often seem to use the real reward signal directly when training a policy network. The default MDRNN model was by no means able to predict future rewards very well. This makes sense, since the original work in *World Models* focused mainly on the model's ability to make good predictions of the next latent state.

However, it is crucial to learn a good reward signal when doing online planning and this is arguably more important than learning to predict good future latent states and make accurate reconstructions from latent space. This is why we use a good RL agent policy to obtain a representative set of observations that sufficiently explores the dynamics and rewards of the environment. Arguably, we could have avoided this and saved a lot of time if we had just employed an iterative end-to-end training approach where we mainly optimize the reward signal and keep improving our model over time.

Chapter 7

Conclusion

We have successfully implemented a game AI agent that can complete whole random tracks in the car racing environment using the Rolling Horizon Evolutionary Algorithm (RHEA) for online planning on a learned model. Thus, we have answered our research question, although several challenges beg for future work. Mainly, we do not consistently solve the continuous control task in the car racing environment of obtaining an average score of 900 across 100 random trials and we rely on access to a representative data set of rollouts generated by a pre-trained RL agent.

A model-based Reinforcement Learning approach is adopted that enables planning. A ConvVAE is used to compress observations into a lower-dimensional latent space. The compressed latent space representation is used to efficiently learn an MDRNN model of the environment dynamics and rewards to enable planning. The model uses a gradient-based training procedure, which requires a rollout exploration policy that sufficiently explores the environment state space. The MDRNN captures long term temporal dependencies using an LSTM and uses an MDN to predict the future as a stochastic Gaussian mixture model distribution over latent states along with the expected reward signal.

The size of the latent vector was increased from 32 to 64 so the MDRNN dynamics model has sufficient information to predict future latent states. Further, the complexity of the MDRNN was increased from 256 to 512 hidden units so that it better captures the rewards and their rank order in the real environment. Also, the sequence length of rollouts used to train the MDRNN was increased from 64 to 500 so that it better captures the long term temporal dependencies of the environment.

To do planning in latent space, we use a simulated environment that synchronizes the MDRNN hidden state with the real environment and supports rolling back to previous hidden states when simulating planning trajectories. We do evolutionary *plan-space planning* by using RHEA to search through a random subspace of continuous plans in the simulated environment. Each individual has its copy of the hidden and latent states to enable parallel simulation and rollback of planning trajectories.

A custom sampling procedure was implemented to avoid simultaneous acceleration and braking, as this was not taken into account in the original environment. A gas threshold of 0.3 down from 1.0 was enforced to avoid the planning agents from driving too fast and losing control, since our dynamics model does not currently capture such driving behaviour.

Our best planning agent uses a population of 8, horizon of 10, 15 generations, 1-bit crossover, tournament-based parent selection, uniform subset mutation and total simulated reward as fitness. NTBEA parameter tuning is used to efficiently find this parameter configuration. RHEA performs better than RMHC, which is most likely due to RMHC getting stuck in a local optima of the policy subspace. In contrast, RHEA uses a population to encourage more diversity across the policy subspace and avoid premature convergence. A random subset of uniformly selected actions is used as mutation operator to ensure sufficient diversity, which is required to avoid individuals from getting stuck in a locally optimal policy subspace.

To the best of our knowledge, this work is novel, since we show how to do efficient online planning using an evolutionary planning algorithm (RHEA) for policy search with a learned dynamics model (MDRNN) in latent space (VAE), which has not been done before. Ultimately, we show that it is possible to do planning with evolution and beat traditional Deep RL methods (A3C, DQN) when given a world model based on sufficient exploration of the environment dynamics. However, new approaches are needed to solve the car racing task, which is described in the next section.

7.1 Future work

This section explores possible extensions recommended in future work. Firstly, a *good rollout exploration policy* is required to sufficiently explore the environment state-space when training a dynamics model using gradient-based methods. Secondly, a *multistep-prediction objective* may be used in the MDRNN to see if this improves the accuracy of multiple consecutive predictions into the future of the latent space, similar to *PlaNet*. Also, the MDRNN struggles with predicting far into the horizon, resulting in a short-sighted agent, which might be addressed by using a learned value network like in *Dreamer* to look beyond the future horizon.

Thirdly, an *end-to-end training* approach, similar to *MuZero*, should help learn only what is relevant to planning, such as the reward signal. In extension to this, it would be interesting to explore a gradient-free evolutionary training procedure like *neuroevolution*, since this would avoid the need for a fixed training set of representative rollouts. Another benefit is that it works in discrete domains and does not assume the objective is continuous and smooth like gradient-based methods. Finally, this avoids having to deal with complex, analytical objective functions derived by Maximum Likelihood estimation. It might also be interesting to add a *deviation to the reward* signal and use this model uncertainty to guide agent behavior and dynamically decide on a planning horizon.

A simple improvement in our approach would be to do more extensive parameter tuning of both our model and planning agent. Namely, we have not spent much time playing with the *number of mixtures* used in our dynamics model. Further, we used a fixed *temperature* parameter to control the stochasticity of the simulated environment, which proved important to making stable future latent predictions but also affects the reward. In terms of planning, *NTBEA tuning* was only run for 100 iterations but the original paper uses 1500 iterations, so it might be interesting to run more iterations of tuning the planning agents on better hardware. Moreover, *alternative fitness functions and evolution strategies* may be explored like self-adaption of parameters where the horizon, macro action and mutation probability are included in individuals and co-evolved dynamically with their planned action solutions (Eiben and Smith 2015, p. 101).

Finally and most importantly, future work would be to learn the reward signal in the MDRNN, starting from random rollouts only, instead of also relying on a good RL policy to generate the initial training rollouts. Arguably, we may obtain similar results by using an iterative trainer that initializes a random world model and does RHEA planning on it to generate new rollouts. The generated planning rollouts could then be used to retrain an improved world model iteratively, until the task of getting an average score of 900 across 100 random trials is completed. Ultimately, we strongly believe that such an *iterative end-to-end training approach* on a simpler domain may help reveal that online evolutionary planning in latent space is possible without assuming access to a representative data set, while also solving the continuous car racing control task.

Appendix A

Appendix

A.1 Background

A.1.1 Model-based RL and Planning Summary

In model-based RL we strive to learn a model of the environment represented by a the transition dynamics function $T(s_t, a_t, s_{t+1}) = P(s_{t+1}|s_t, a_t))$ and a reward function $R(s_t, a_t, s_{t+1}) = E[R_{t+1}|S_t = s_t, A_t = a_t]$. Planning is then using this model to decide a course of action (policy) for interacting with the modeled environment by considering possible future situations before they are actually experienced. This is either done using *state-space planning*, where planning is a search through the state space for an optimal policy or *plan-space planning* where planning is a search through the space of plans for a plan that maximize expected return.

Model-free learning is similar to state-space planning in computing value functions as a key step toward improving the policy and do so by updates or backup operations to simulated experience. Traditionally, we use Dynamic programming or to search through the state space and generate updated targets of our state's estimated value: $V(S_t) \leftarrow E_\pi(R_{t+1} + \gamma V(S_{t+1})$ (approximated by DNNs). However, the difference between model-free learning and model-based planning methods is that the planning method uses simulated experiences in the simulated environment (model) to estimate the value function by backing-up update operations, whereas learning methods use real experience generated by the true environment.

The other way to do planning is to begin and complete the search for a policy after encountering each new state s_t by outputting a single action only and then repeat the planning for the next state s_{t+1} to produce action a_{t+1}, and so on. This is called *decision time planning*, where simulated experiences are used to select an action for the current state with a limited decision time budget instead of gradually improving a policy or value function through a learned approach. This ambiguous presence of learning a policy in both the model-free and model-based planning approach may be confusing. That is, one may learn a policy directly from real experiences (model-free) or use simulated experience in a model.

Planning may be done on the state-space by learning a policy from state-values or in the planning-space by searching for a policy (e.g. MCTS or RHEA).

A.1.2 Reinforcement Learning: Model-Free Approach

Model-free RL is used where we learn a policy directly from experience. The MDP is typically solved using the *Bellman equation*, which recursively describes the expected rewards. It is a *dynamic programming* equation that decomposes a problem into smaller subproblems through its inherent recursive structure with overlapping subproblems. It is used to describe the optimal value of state and a q-state. The optimal value of a state s, $V^*(s)$, is the maximum expected value of the future discounted reward (return) that an optimally-acting agent starting from s will receive over its lifetime where. The optimal value of a q-state (s, a), $Q(s, a)$, (also known as the q-value of an action-state), is the maximum expected value of the future discounted reward an agent will receive after starting in s, having taken action a, then acting optimally. Recall $T(s, a, s') = P(s'|s, a)$.

$$V^*(s) = max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^*(s')] \tag{A.1}$$

$$\begin{aligned} Q^*(s, a) &= E[R(s, a, s') + \gamma max_{a'} Q^*(s', a')|s, a] && \text{(A.2)} \\ &= \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma max_{a'} Q^*(s', a')] && \text{(A.3)} \end{aligned}$$

$$V^*(s) = max_a Q^*(s, a) \tag{A.4}$$

Hence, if the environment is stochastic, the reward the agent can expect to receive is an expectation, which is the average reward weighted by the probability of future states conditioned on taking action a in state s. Otherwise, in a deterministic environment $T(s, a, s') = 1$ we just get a sum of rewards.

Q-learning is a popular example of an active feedback-based model-free RL algorithm that learns an optimal policy by watching the agent play (e.g. randomly) and gradually improving its estimates of the above Q-values computed from the Bellman equation that maximize the expected future reward (cumulative, discounted) starting from the current state. Thus, model-free learning attempts to estimate the q-values of states (how good a state is based on the reward I expect to get) directly, without any memory to construct a model of the rewards and transitions in the MDP. Q-learning solves the dynamic programming problem by iteratively updating the action q-values function using the Bellman equation to compute Q_{t+1} for all states s in the current time step t in the environment. Q-learning will acquire *q-value samples* $R(s, a, s') + \gamma max_{a'} Q(s', a')$ during agent play and do value iteration updates, using the weighted average of the old value and new learned value in equation A.5.

$$sample = R(s, a, s') \cdot \gamma \cdot Q(s', a')$$
$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \cdot sample \tag{A.5}$$

Assuming this procedure spends enough time exploring the state space while decreasing the learning rate α appropriately, Q-learning will learn the optimal q-values for every q-state even with suboptimal or random actions (*off-policy learning*): $Q_t \rightarrow Q^*$ as $t \rightarrow \infty$. However, this is done for each episode of experiences (sequence of actions and states), which is computationally intractable given a very large state space. Thus, a popular method is to use *Approximate Q-learning* by combining Q-learning with a deep convolutional neural network that works as a *nonlinear function approximator* used to estimate q-values of any state-action pair (s, a): $Q^*(s, a) \approx Q(s, a, \theta)$, where θ is a vector of model parameters. This is an example of *Deep Q-Learning* (DQN), which is how Deep Mind used a Deep Neural Network (DNN) called a Deep-Q-Network (DQN) to estimate Q-values. Hence, a learned function approximator may be used to cope with a large state space by estimating the value of states and thereby learn a good policy directly from raw video game pixels.

A.1.3 Planning: Monte Carlo Tree Search(MCTS)

Monte Carlo Tree Search is an example of an informed search method that uses UCT as a heuristic for exploration and uses simulation-based rollouts to estimate the states. Instead of computing or approximating value function ($V^*(s)$ or $Q^*(s, a)$, Monte Carlo methods estimate the expected return of a state by averaging the return across multiple rollouts of a policy. Thus, this works in non-Markovian environments too, but can only be used in episodic MDPs, since the rollouts have to terminate to compute its return. The idea is to expand the current state and run rullouts until a terminal is reached. Values are then filled in the search tree by backpropagation of estimated values. Iteratively, you go back to the root in the search tree of visited nodes and run a new simulation on a new node using a default policy (e.g., random). Gradually, given the knowledge in the current search tree, you start to guide the search towards more promising nodes using a tree policy. Effectively, you end up ignoring useless parts of the search tree while still exploring them enough to be sure they are useless.

Alternatively, you could use other informed search algorithms but they often rely on heuristics that are hard to capture in complex environments (hard value function to learn). Instead, the novel idea is to use random simulations to guide search to promising actions. This assumes that the simulation policy is acting as a reasonable proxy for deeper search. This should be the case, since the search tree is developed with rich information further over time. In a way, the simulation policy is just acting as a heuristic guiding search towards best parts in the search space. Eventually, the random policy is replaced by knowledge in the tree.

Thus, any short-term bias should be removed asymptotically even when using a random policy. The benefits of MCTS (Tree search with MC rollouts) is a highly selective best-first search (pick promising nodes). Also, states are evaluated dynamically unlike in Dynamic Programming and offline learning an approximation of the entire state space. Further, sampling helps break the curse of dimensionality (search space increases exponentially in the number of dimensions), since we do not have to consider all the information in the environment. Also, simulations work even for black box models, since they only require a model to do sampling. Finally, a procedure like MCTS is computationally efficient (parallelizable rollouts) and anytime (query real-time).

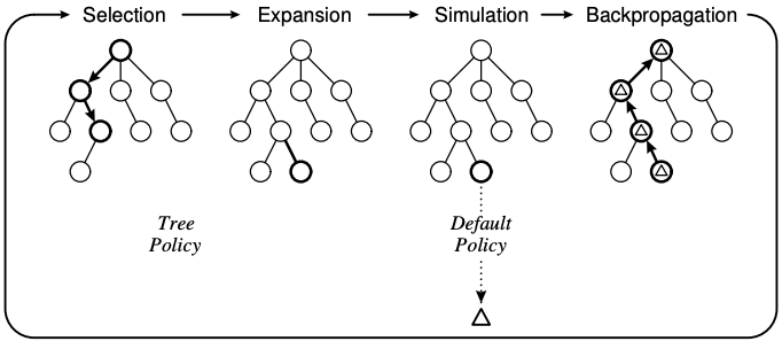


Figure A.1: One iteration of general MCTS approach (Perez, S. Lucas, et al. 2012)

The goal of MCTS is to approximate the (true) value of the actions that may be taken from the current state, which is achieved by iteratively building a partial search tree using the above procedure. Four steps are used for each search iteration:

- *Selection*: a child selection policy is recursively applied from root note to descent through the tree until a leaf node is reached. A node is *expandable* if it is not a terminal state and has unvisited children
- *Expansion*: One (or more) child nodes are created to expand the tree based on valid actions.
- Simulation: a simulation is run from the new node(s) according to a default policy (random) until a terminal node is reached (play-out/rollout).
- *Backpropagation*: The simulation result (e.g. total reward) is "backed up" through the selected nodes to update their statistics

Monte Carlo methods do sampling to approximate the Q-value of a state-action pair, which is simply the expected reward of that action where $N(s,a)$ is the number of times action a has been selected from state s, $N(s)$ is the number of times a game has been played out through state s, z_i is the result of the ith simulation played out from s, and $I_i(s,a)$ is 1 if action a was selected from state s on the ith play-out from state s or 0.

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N} (s) I_i(s,a) z_i \tag{A.6}$$

The way the tree is built depends on how nodes in the tree are selected. The success of MCTS is primarily due to its choice of tree policy used to select and expand nodes that are more promising. The most popular choice is the Upper Confidence Bound (UCB) heuristic as tree policy that treats the choice of child node as a multi-armed bandit problem where the value of a child node is the expected reward approximated by the Monte Carlo simulations. It is used to address the exploration-exploitation dilemma in MCTS by selecting child nodes that maximize a combination of exploitation of high reward nodes and exploration of less visited nodes. MCTS that uses UCB as tree policy is called MCTS with Upper Confidence Bounds for Trees (UCT) where a child node j is selected to maximize:

$$UCT = v_j + C \times \sqrt{\frac{lnn}{n_j}} \tag{A.7}$$

,

where v_j is the estimated value of the node (exploitation term, e.g. $Q(s,a)$) and the 2nd term is used for exploration where n_j is the number of times the node has been visited, n is the total number of times that its parent has been visited and C is a non-negative temperature hyperparameter used to control the degree of exploration typically set to $C = \sqrt{2}$.

Normalized performance of GPUs

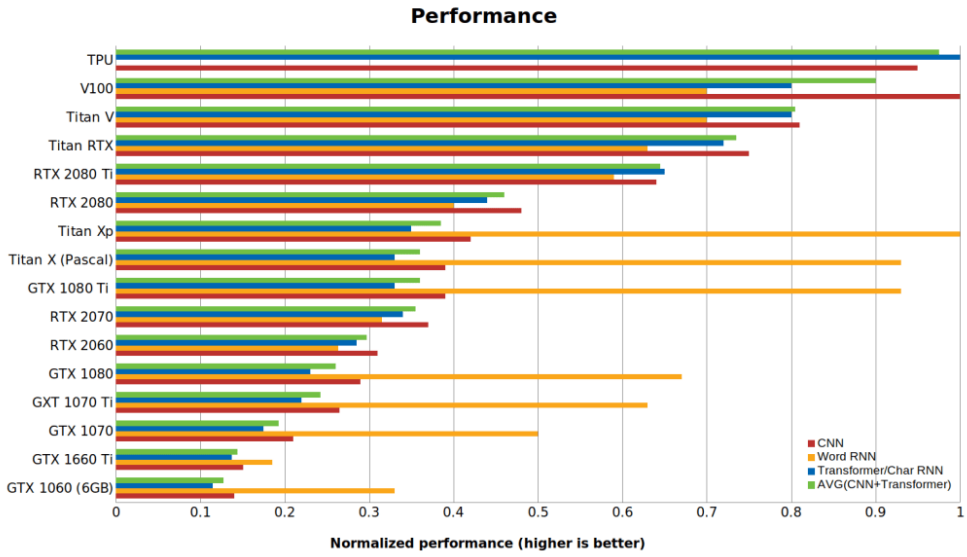


Figure A.2: Normalized Performance of GPUs and TPU in PyTorch 1.0.1 and CUDA 10. Higher is better. (Dettmers 2019)

A.2 Related Work

A.2.1 Shaping Belief States with Generative Environment Models for RL

In *Shaping Belief States with Generative Environment Models for RL* (Gregor et al. 2019), the authors investigate how agents form and maintain beliefs about aspects of the environment relevant to planning. They propose a way to efficiently train expressive generative models in complex 3D environments to form stable belief-states. Previously, the stochastic state-space models (latent dynamics) seen so far only cope with 2D environments. Thus, this paper is a useful starting point for planning in more realistic 3D environments. An example of this could be a simulated environment for autonomous vehicle driving where self-driving cars have a 3D view of their environment through the use of cameras or light-based radar sensors (LIDAR). The authors show it is possible to capture the layout of the environment and the position and orientation of the agent. Similar to *Learning Latent Dynamics for Planning from Pixels*, they also mention the importance of predicting multiple steps into the future *overshooting* in combination with a generative model to obtain stable representations. The authors hypothesize that "an ideal unsupervised learning algorithm should use past observations to create a stable representation of the environment", which includes "capturing the variation of the environment in a temporally coherent way" (Gregor et al. 2019, p. 1). They formalize the problem as a partially observable environment using a POMDP to model scenarios where the agent cannot observe the world state fully. A POMDP agent either acts to change the world state (as in MDP) or to obtain new evidence sensed in the environment to update its belief (probability) about being in the current world state. This approach to planning relies on summarizing previous observations $o_{1:t}$ into a belief distribution $b_t(s_t) = p(s_t|o_{1:t})$ about the current unobserved hidden state s_t, which means the belief distribution is a sufficient statistic of past observations: $b_t \equiv o_{1:t}$. Thus, in previous settings, we assumed access to the environment state s_t, but here we replace s_t with b_t to go from an MDP to a "belief MDP" (POMDP). We do not know the actual state but use noisy observations to infer what state may be as a distribution. The belief state captures the same information as past observations because it is a distribution over states that condition on those same observations. This provides a way to use all past observations in a tractable way since a long sequence of observations with information is retained within the current belief state. Intuitively, we can think of this as selecting actions in a "belief MDP", which is like an MDP except s_t is replaced with b_t.

The fundamental problem in building useful environment models is *long-term consistency*. Namely, most models are unable to perform coherent long-term predictions while performing accurate short-term predictions, even in simple, partially observed environments. The authors argue that this has nothing to do with the capacity of these models but instead is due to failure in model conditioning and a weak objective.

They propose a belief-state architecture based on the LSTM (RNN) augmented with a *Kanerva Memory model* to remember a map of the world, which is useful for representing the current belief of situations in realistic driving environments. The Kanerva memory model is a *Sparse Distributed Memory (SDM)* model of human long-term memory (Kanerva 1988). It is a generalized random-access memory (RAM) that consists of a fixed table of addresses A pointing to modifiable memory M, which supports fast reads and writes, and the size/capacity of the SDM store is independent of the input dimension. This is useful for model-free RL methods that often employ a memory buffer of experiences to train a learned policy.

A simple way to form a belief state b_t is to train a next-step prediction model $p(s_{t+1}|s_1, \ldots, s_t) = p(s_{t+1}|b_t)$, where $b_t = RNN(b_{t-1}, s_t, a_t)$ summarizes the past and s denotes the state of the environment. This is analogous to the *World Models* paper where we use a latent representation z_t of the real states s_t (denoted x_t in the original paper) and an RNN with a hidden state h_t to capture the past except an RNN is deterministic so the hidden state of an RNN is equivalent to a deterministic distribution over hidden states such that $p(z_{t+1}|z_{1:t}) = p(z_{t+1}|h_t)$ where $h_t = RNN(h_{t+1}, z_t, a_t)$. Unlike *World Models*, this paper relaxes the assumption that the real states are observable and instead represent states by a stochastic belief distribution over hidden states. Under an optimal solution, it contains all information required to predict the future $p(s_{t+1:H}|b_t)$ with horizon H, since any joint distribution can be factorized as a product of conditionals: $p(s_{t+1}, s_{t+2}, \ldots, s_{t+H}|b_t) = p(s_{t+1}|b_t) \times p(s_{t+2}|b_{t+1}) \times \ldots \times p(s_{t+H}|b_{t+H}) = f(s_{t+1}, b_t) \times \ldots \times f(s_{t+H}, b_{t+H})$, which is why most research use next-step prediction in RL. The authors show that it is possible to predict the immediate future s_{t+1} with high accuracy as an almost deterministic function of the immediate k past observations $s_{t-k:t}$. Intuitively, this means $p(s_{t+1}|s_{t-k:t}, a_{t-k-1:t-1}, b_{t-k}) \approx P(s_{t+1}|s_{t-k:t}, a_{t-k-1:t-1})$ where the immediate past weakens the need of the belief state. However, the authors argue that predicting the distant future requires a better knowledge of the environment dynamics, encouraging the need of belief-state with such information. In this regard, overshooting (predicting multiple steps ahead) seems to improve the ability to predict the long-term future.

The authors use a single-layer MLP to predict the discretized position and orientation of the agent and a convolutional network to predict the top-down view (map decoder). The belief state b_t is learned by an LSTM and is composed of the LSTM hidden state h_t and the LSTM cell state c_t. This is used as the belief state to decode the map and find the current location of the agent. The paper does not look into planning but focuses mainly on the effect of model choices on the learned representations and belief states. They use a fixed handcrafted policy that chooses random locations for generating training data and a path planning policy to move between locations used to analyze the model and belief state (so no RL).

Arguably, their approach is less relevant to this thesis, since they rely on a very complicated architecture (Kanerva Sparse Distributed Memory) to enable efficient model-free RL on learned 3D environment models and they do not do any online planning as we wish to do in our 2D driving environment. However, the paper serves as a good reminder for what future work may be done using POMDPs to model realistic 3D driving environments where the world is not fully observable and we only observe noisy sensor observations $o_t \sim O(o_t|s_t, a_t)$ as evidence to update our current belief $b_t = P(s_t|o_{1:t-1}, a_{1:t-1})$ about the world into a new belief $b(s_{t+1}) = P(s_{t+1}|o_t, a_t, b_t) \propto O(o_t, s_{t+1}, a_t) \sum_{s_t} T(s_{t+1}|s_t, a_t) b_t(s_t)$.

A.3 Approach

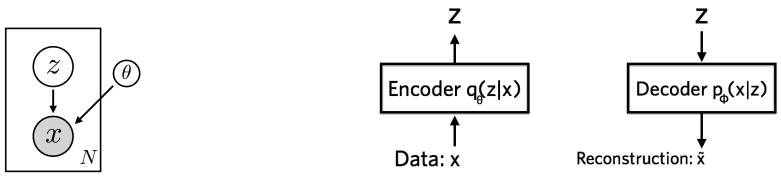
A.3.1 Generative Modelling

"Generative modelling" is an area of Machine Learning that deals with models of distribution $P(X)$ defined over data observations X in some potentially large dimensional space d. In our case, states are represented by high-dimensional images, for which we would like to find a compressed representation that is more efficient to plan in using a forward (dynamics) model on it. Thus, the job of our generative model is to somehow capture the dependencies between pixels by assigning real images high probability and images of random noise with low probability. Formally, our goal is to learn a model P_θ which we can sample from, such that P_θ is as similar as possible to $P(X)$ where $P(X)$ is the true distribution of examples X. Training these kinds of models may be hard for an number of reasons. Firstly, they might require strong assumptions about the structure of the data. Second, they might make poor approximations, leading to poor models. Third, they might rely on computationally intractable inference procedures like Markov Chain Monte Carlo (MCMC). Today, a popular alternative is training neural networks as powerful universal function approximators through backpropagation (see 2.5.1).

A.3.2 VAE as a Probabilistic Graphical Model (PGM)

Interestingly, a Variational Autoencoder (VAE) can be understood from both a deep learning and graphical model perspective. So far, we have described a variational autoencoder from a neural net perspective as a generative model that consists of an encoder, a latent space and a decoder as shown in figure A.3b. From a Bayesian perspective, this is originally described with a probabilistic graphical model (PGM), which is a graph of conditional probabilities (edges) between random variables (nodes) as shown in figure A.3a.

Our goal is to find a model that is representative of our dataset X (in our case states s). Formally, we wish to obtain a vector of latent variables z in a high-dimensional space $\mathbb{Z} \in R^{64}$, which we can easily sample from according to a probability density function (PDF) $P(z)$ defined over Z.



(a) VAE as a graphical model. Sample from z and X N times while model parameters are fixed. (Doersch 2016)

(b) VAE as neural network. Encoder compresses data X into latent Z and decoder reconstructs X from Z (Altosaar 2020)

Figure A.3: VAE as Graphical Model (left) and Neural Network (right)

Given a deterministic function $f(z;\theta)$ parameterized by a vector θ in a deep neural network, we wish to optimize θ such that we can sample z from $P(z)$ and with high probability, $f(z;\theta)$ will be like the observations in our dataset. Thus, we aim to maximize the probability of each X in our training set under the generative process as follows:

$$P(X) = \int P(X|z;\theta)P(z)dz \tag{A.8}$$

Notice our function approximator $f(z;\theta)$ parameterized by θ as a DNN has been replaced by a distribution $P(x|z;\theta)$ to show explicitly that X depends on z using the total probability theorem. Ultimately, our goal is to do *Maximum Likelihood Estimation* (MLE), which is finding the parameter values in θ that maximize the likelihood of making the observations X given the parameters θ and latent z.

In VAEs, the choice of output distribution is often the continuous Gaussian PDF like in our case: $P(X|z;\theta) = N(X|f(z;\theta), \sigma^2 \cdot I)$ with mean $f(z;\theta)$ and covariance equal to the identity matrix I times a scalar deviation σ. By using a Gaussian distribution, we can use gradient descent to increase $P(X)$ by making $f(z;\theta)$ approach X for some z.

This corresponds to gradually making the training data more likely under the generative model. The important assumption is that $P(x|z)$ is computable and continuous in the model parameters θ, since the gradient descent optimization procedure requires a function that is continuous and smooth for gradients to exist as needed in backpropagation during training.

To maximize Equation A.8, VAEs need to define how information is represented with latent variables z and how to deal with the integral over z. Interestingly, the VAE assumes that samples of z can be drawn from a simple normal distribution $z \sim N(0, I)$, where I is the identity matrix. This is possible because any distribution in d dimensions can be generated by taking a set of d variables that are normally distributed and mapping them with a sufficiently complex function, which is made possible by deep neural networks as universal function approximators.

That is why a deep neural network is used as a function approximator $f(z;\theta)$ where normally distributed $z's$ are mapped to latent values, which are ultimately useful for reconstructing the original image data. Interestingly, we do not need to worry about whether such latent structure exists, since the model will learn such structure if it helps accurately reproduce (i.e., maximize likelihood of) the training data. Hence, we want to maximize the likelihood of generating our real training observations indexed by i:

$$\theta^*_{ML} = argmax_\theta \, p_\theta(X;\theta) = argmax_\theta \prod_{i=1}^{n} p_\theta(x_i;\theta) \tag{A.9}$$

Maximizing the likelihood of our training data (generating real data samples) is equivalent to minimizing the log likelihood of our data as a loss function with gradient descent.

Bayesian Inference

Looking at the VAE from a Bayesian perspective as a PGM (see A.3a), we need to find $p(z|x)$ where x represents observable data and z represents a hidden variable. This is a *Bayesian inference* problem where we use Bayes theorem to update our beliefs upon observing the data. This is done by computing a posterior belief $p(z|x)$ based on a prior belief $p(z)$, likelihood of our data $p(x|z)$ and observations (data) $p(x)$, using Bayes rule, the multiplication rule (numerator) and the marginalization rule (denominator):

$$p(z|x) = \frac{p(z,x)}{p(x)} = \frac{p(x|z)\cdot p(z)}{\int_z p(x|z)\cdot p(z)dz} \tag{A.10}$$

However, the problem is computing the marginal $p(x)$, which is intractable, since $z \in R^d$ is higher dimensional so the integral will have a form $\underbrace{\int \cdots \int}_{d \text{ times}}$ Instead, we will approximate $p(z|x)$ to find $p(x)$:

$$p(x) = \frac{p(x|z)\cdot p(z)}{p(z|x)} \tag{A.11}$$

Approximate Inference

For this purpose, one may do *approximate inference* by either using *Markov Chain Monte Carlo (MCMC)* sampling methods or *Variational Inference (VI)* optimization methods to approximate the posterior $p(z|x)$. Traditionally, MMCMC has been most widely used and is based on a stochastic Markov chain model with random variable nodes and transition probability edges subject to the Markov property. In MCMC, a Markov chain is constructed on z whose stationary distribution (i.e. remains unchanged as time progresses) is the posterior $p(z|x)$. Sampling is performed from the chain to collect samples from the stationary distribution. The posterior is approximated with an empirical estimate constructed from the collected samples (Blei, Kucukelbir, and McAuliffe 2018, p. 2).

In short, MCMC uses sampling to approximate the posterior and is non-parametric and asymptotically exact given enough time and an appropriate sampling procedure but this is not feasible in practice given the finite amount of time available and potentially large amount of data.

Variational Inference (VI)

Rather than use sampling, the main idea behind VI is to use optimization, which is faster and easier to scale to large data at the potential cost of a less accurate approximation. VI methods approximate the intractable integral of the posterior $p(Z|X)$ with a parameterized posterior approximation $q_\theta(Z|X)$ that is close to a family of tractable densities such as the Gaussian $z \sim N(0, I)$. Thus, VI finds a tractable approximate posterior $q_\theta(Z|X)$ by minimizing the KL divergence between a tractable distribution $q(Z)$ and the exact posterior $P(Z|X)$. Formally, we wish to approximate $P(Z|X)$ with a tractable (e.g. Normal or Exponential) distribution $q(Z)$ by playing with the parameters of the distribution s.t. $q(Z|X) \approx P(Z|X)$. Put simply, we avoid the intractable problem by using a parameterized distribution that we know how to deal with and make it as similar as possible to the original distribution.

How do we make q close to p?

In order to make our approximate posterior $q(z|x) \approx p(z|x)$, we will use the Kullback-Leibler (KL) Divergence score, which quantifies how much one probability distribution differs from another probability distribution.

This takes point of departure in *information theory (entropy)*, which is the level of information, surprise or uncertainty inherent in a random variable's possible outcomes. The premise is that the "informational value" of an event depends on the degree to which it is surprising. Namely, if there is a high probability of something occuring (e.g. the sun rising tomorrow), then there is little information because it is certain. On the other hand, if there is a low probability of something happening (e.g. snow during summer), then this event contains large information.

This is quantified with the *Information* equation I:

$$I(x) = -logp(x) \tag{A.12}$$

In this regard, *entropy* denoted H measures the expected (average) amount of information conveyed by knowing the outcome of a random event:

$$H(x) = E[I(x)] = -E[-logp(x)] = -\sum p(x) \cdot logp(x) \tag{A.13}$$

Finally, the KL divergence $\mathbb{KL}(p||q)$ measures dissimilarity between two different distributions p, q quantified almost by the difference in their entropy except it is with respect to the first distribution p:

$$\begin{aligned}
\mathbb{KL}(p||q) = & \\
&= H(q)_{q\sim p(x)} - H(p) \\
&= E_{q\sim p(x)}[-logq(x)] - E[-logp(x)] \\
&= -\sum p(x) \cdot log(q) + \sum p(x) \cdot logp(x) \\
&= \sum p(x) \cdot (log(p(x) - logq(x)) \\
&= \sum p(x) \cdot log\frac{p(x)}{q(x)} \\
&= E[log\frac{p(x)}{q(z)}] \\
&= E[-log\frac{q(x)}{p(x)}]
\end{aligned} \tag{A.14}$$

, where $KL \geq 0$ and $KL(p||q) \neq KL(q||p)$, meaning the KL measure is antisymmetric or divergent and not a symmetric distance measure.

Now, in order to make our approximate posterior distribution $q(z)$ close to the true distribution p, we need to minimize their KL divergence with respect to q:

$$min\mathbb{KL}(q||p) \tag{A.15}$$

Interestingly, we can decompose the KL divergence between the approximate posterior $q(z)$ and the true posterior $p(z|x) = \frac{p(x|z)\cdot p(z)}{p(x)} = \frac{p(x,z)}{p(x)}$ such that we obtain the log likelihood $logp(x)$:

$$\begin{aligned}
\mathbb{KL}(q(z)||p(z|x) = & \\
&= -\sum_z q(z) \cdot log\frac{\frac{p(x,z)}{p(x)}}{q(z)} \\
&= -\sum_z q(z) \cdot log(\frac{p(x,z)}{q(z)} \cdot \frac{1}{p(x)}) \\
&= -\sum_z q(z) \cdot [log\frac{p(x,z)}{q(z)} + log\frac{1}{p(x)}] \\
&= -\sum_z q(z) \cdot log\frac{p(x,z)}{q(z)} + \sum_z q(z) \cdot logp(x) \\
&= -\sum_z q(z) \cdot log\frac{p(x,z)}{q(z)} + logp(x) \cdot \sum_z q(z) \\
&= -\sum_z q(z) \cdot log\frac{p(x,z)}{q(z)} + logp(x) \\
&= E_{z\sim q(z)}[-log\frac{p(x,z)}{q(z)}] + logp(x)
\end{aligned} \tag{A.16}$$

Notice, we used the following rules: $\frac{\frac{a}{b}}{\frac{c}{d}} = \frac{ad}{bc}$, $log(ab) = log(a) + log(b)$, $log\frac{a}{b} = log(a) - log(b)$, $log(1) = 0$, $\sum_z q(z) = 1$ and $\sum_z zx = x \cdot \sum_z z$.

Thus, the likelihood of the data can be decomposed into a KL divergence term $\mathbb{KL}(q(z)||p(z|x))$ to be minimized and a *variational lower bound* term $\mathbb{L} = E_{z\sim q(z)}[log\frac{p(x,z)}{q(z)}]$ to be maximized:

$$logp(x) = \mathbb{KL} + \mathbb{L} = \mathbb{KL}(q(z)||p(z|x)) + E_{z\sim q(z)}[log\frac{p(x,z)}{q(z)}] \tag{A.17}$$

The latter term is called "lower bound" because $\mathbb{KL}(q(z)||p(z|x) \geq 0$ and $logp(x) = \mathbb{KL} + \mathbb{L}$ so $\mathbb{L} \leq logp(x) - \mathbb{KL} \leq logp(x)$, meaning $\mathbb{L} \leq logp(x)$. Hence, maximizing the lower bound $\mathbb{L}$ of another function $logp(x)$, also maximizes that function $logp(x)$. As a result, we realize that maximizing the log likelihood of the data $logp(x)$ is achieved by minimizing the KL divergence $\mathbb{KL}$ and maximizing the lower bound $\mathbb{L}$. Recall $p(z|x) = \frac{p(x,z)}{p(x)} = \frac{p(x|z)\cdot p(z)}{\int_z p(x|z)\cdot p(z)}$ is intractable. Instead, we may rewrite the joint $p(x, z)$ as $p(x|z) \cdot p(z)$ in a further decomposition of the lower bound:

$$\begin{aligned}
\mathbb{L} &= E_{z\sim q(z)}[log\frac{p(x,z)}{q(z)}] \\
&= E_{z\sim q(z)}[log\frac{p(x|z)\cdot p(z)}{q(z)}] \\
&= E_{z\sim q(z)}[logp(x|z) + log\frac{p(z)}{q(z)}] \\
&= E_{z\sim q(z)}[logp(x|z)] + E_{z\sim q(z)}[\frac{p(z)}{q(z)}] \\
&= E_{z\sim q(z)}[logp(x|z)] - \mathbb{KL}(q(z)||p(z))
\end{aligned} \tag{A.18}$$

Interestingly, this decomposition of the lower bound shows that maximizing the lower bound is equivalent to maximizing the log likelihood $logp(x|z)$ and mimimizing KL divergence between our approximate posterior $q(z)$ and a prior $p(z)$. By revelation, we can pick a tractable distribution from a known family of distributions (e.g. Gaussian) as our prior $p(z)$ and make our approximate posterior $q(z|x)$ close to $p(z)$, which is tractable and thus enables us to efficient optimization.

From graphical model to autoencoder

In the graphical model shown previously, we have two random variables $x \sim p(x|z)$ and $z \sim p(z)$. Now, we will assume that a distribution $q(z|x)$ exists. We already know that the posterior $p(z|x)$ is hard to compute. Thus, we instead want to compute $q(z|x) \approx p(z|x)$ by minimizing $\mathbb{KL}(q(z|x)||p(z|x))$ and maximizing the variational lower bound $\mathbb{L}$ in equation A.17. We showed that maximizing this lower bound is equivalent to minimizing KL divergence between our approximate posterior $q(z|x)$ and a tractable prior $p(z)$ and maximizing the log likelihood of the data in equation A.18.

Now, we can assume $p(x|z)$ and $q(z|x)$ are approximated with neural networks. The approximate posterior is parameterized with an encoder neural network $q_\phi(z|x)$ and the likelihood $p(x|z)$ is parameterized with a decoder neural network $p\theta(x|z)$. Now, the encoder objective is to find $q(z|x) \approx p(z|x)$ via KL divergence and the decoder objective is to maximize the log likelihood of the data, which together maximize the lower bound L.

Interestingly, we can pick a distribution of our choice to represent $p(z)$. This is usually taken to be a tractable standard normal (noise):

$$z \sim N(0, I) \tag{A.19}$$

The encoder will then try to make the distribution in latent space similar to the standard normal distribution just picked.

For the decoder, we may observe that $p(x|z)$ is a deterministic function that maps a sampled latent vector z to a predicted reconstruction $\hat{x}$ such that $\hat{x} \approx x$. Normally, we assume the actual observations follow a Gaussian distribution when given the reconstructions:

$$p(x|\hat{x}) = e^{-|x-\hat{x}|^2} \iff logp(x|\hat{x}) = |x - \hat{x}|^2 \tag{A.20}$$

Consequently, maximizing the expected log likelihood $E_{z\sim q(z)}[logp(x|z]$ is equivalent to minimizing the L_2 squared euclidean reconstruction error $|x - \hat{x}|^2$.

Compared to a traditional autoencoder that minimize $|x - \hat{x}|^2$, we now minimize this, in addition to the KL divergence term $K(q_\phi(z|x)|p_\theta(z|x))$. Thus, the KL divergence term functions as a regularizer to the autoencoder such that the distribution of latent variables has a certain form similar to that of a tractable prior distribution $p(z)$ picked by us.

Thus, we need to design the encoder network $_\phi(z|x)$ such that it does not generate a latent code $z \in R^d$ but instead the parameters of the distribution $p(z)$. We have assumed z follows a normal distribution in equation A.19 parameterized by means $\mu \in R^d$ and standard deviations $\sigma \in R^d$. A common choice of the form of $q_\phi(z|x)$ is a multivariate Gaussian with a diagonal covariance:

$$z \sim q_\phi(z|x) = N(z|\mu, \Sigma = \sigma^2 I) \tag{A.21}$$

The covariance matrix of the encoder distribution is diagonal to reduce its parameter space from d^2 to d parameters:

$$\Sigma = \begin{bmatrix} \sigma_1 & \cdots & 0 \\ 0 & \ddots & 0 \\ 0 & \cdots & \sigma_d \end{bmatrix} \tag{A.22}$$

Thus, the idea is to use the encoder $q_\phi(z|x)$ to generate the means μ and standard deviations σ of our tractable prior target distribution $p(z)$ and sample $N_z = 64$ latent vectors from $z \sim N(\mu, \sigma I)$ as input to the decoder (see figure 4.4). The reason why it is a generative model is because we sample latent vectors z and feed the decoder $P(x|z)$ with these to generate reconstructions similar to real observations.

Maximum Likelihood Estimation

In our case, we have a probabilistic decoder model $p_\theta(x|z)$ controlled by a set of parameters θ. The decoder model provides a probability distribution over example observations x (frames), which are assumed independent and identically distributed (i.i.d.). The *Maximum Likelihood (ML)* estimation procedure is used to pick the parameters θ that maximize the probability that the decoder model will generate the training data. One usually exploits the monotonically increasing property of the logarithm and instead optimizes the *log likelihood* that is more convenient to work with than the likelihood, since it can convert a product of factored distributions into a sum that may analytically manipulated or enable parallel loss computations. In cases where little data is available, the ML estimation of parametric models performs poorly compared to Bayesian inference (in which one makes new predictions by integrating over all possible parameter valeus of θ).

Intuitively, we normally want do to maximum likelihood estimation in which we find the parameters θ such that the likelihood of our observations is maximized, meaning the model $p_\theta(X)$ will likely generate the training data $p(X)$. The expected likelihood of our data for a single sample is:

$$\begin{aligned} &\mathbb{L}(\theta|x) = p_\theta(x) \text{ (general form)} \\ &\mathbb{L}_i(\theta, \phi|x_i) = -E_{x_i \in X}[E_{z \sim q_\phi(z|x)}[p_\theta(\hat{x_i}|z)]] \end{aligned} \tag{A.23}$$

The expected log likelihood for a single sample is given by:

$$\begin{aligned} &\mathbb{LL}(\theta|x) = logp_\theta(x) \text{ (general form)} \\ &\mathbb{L}_i(\theta, \phi|x_i) = E_{x_i \in X}[E_{z \sim q_\phi(z|x)}[logp_\theta(\hat{x_i}|z)]] \end{aligned} \tag{A.24}$$

The goal is to maximize likelihood function, which is similar to maximizing the expected log likelihood function. In order to use this objective as a loss function to be optimized with gradient descent, we use the expected negative log likelihood to be minimized:

$$\begin{aligned} \theta^*_{ML} &= argmax_\theta \mathbb{LL}(\theta|X) \text{ (general form)} \\ &= argmin_\theta - \mathbb{LL}(\theta|X) \\ &= argmin_\theta E_{x_i \in X}[E_{z \sim q_\phi(z|x)}[-logp_\theta(\hat{x_i}|z)]] \end{aligned} \tag{A.25}$$

Notice, our VAE loss function (see 4.5) strives to minimize $\mathbb{KL}(q_\phi(z|x)||p(z))$ and $E_{q\phi(z|x)}[-logp_\theta(x|z)]$, which is the exact equivalent to minimizing the above, meaning we maximize the likelihood of the data. Ultimately, this verifies the choice of objective function when trying to learn a generative VAE model that is able to create images similar to real training examples.

A.3.3 Derivation of KL Divergence between two Multivariate Gaussian Distributions

Recall the KL divergence between two distributions P and Q is defined as

$$D_{KL}(P||Q) = E_P[log\frac{P}{Q}] \tag{A.26}$$

Also, the probability density function for a multivariate Gaussian (normal) distribution with mean μ and covariance matrix Σ is

$$p(x) = \frac{1}{(2\pi)^{n/2}|\Sigma^{1/2}}exp(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)) \tag{A.27}$$

The KL divergence for two multivariate Gaussians in R^n, $P_1 = N(x|\mu_1, \Sigma_1)$ and $P_2 = N(x|\mu_2, \Sigma_2)$ is

$$\begin{aligned}
&D_{KL}(P_1||P_2) \\
&\quad = E_{P_1}[logP_1 - logP_2] \\
&\quad = \frac{1}{2}E_{P_1}[-log|\Sigma_1| - (x-\mu_1)^T\Sigma_1^{-1}(x-\mu_1) + log|\Sigma_2| + (x-\mu_2)^T\Sigma_2^{-1}(x-\mu_2)] \\
&\quad = \frac{1}{2}log\frac{|\Sigma_2|}{|\Sigma_1|} + \frac{1}{2}E_{P_1}[-(x-\mu_1)^T\Sigma_1^{-1}(x-\mu_1) + (x-\mu_2)^T\Sigma_2^{-1}(x-\mu_2)] \\
&\quad = \frac{1}{2}log\frac{|\Sigma_2|}{|\Sigma_1|} + \frac{1}{2}E_{P_1}[-tr(\Sigma_1^{-1}(x-\mu_1)(x-\mu_1)^T + tr(\Sigma_2^{-1}(x-\mu_2)(x-\mu_2)^T)] \\
&\quad = \frac{1}{2}log\frac{|\Sigma_2|}{|\Sigma_1|} + \frac{1}{2}E_{P_1}[-tr(\Sigma_1^{-1}\Sigma_1) + tr(\Sigma_2^{-1}(xx^T - 2x\mu_2^T + \mu_2\mu_2^T))] \\
&\quad = \frac{1}{2}log\frac{|\Sigma_2|}{|\Sigma_1|} - \frac{1}{2}n + \frac{1}{2}tr(\Sigma_2^{-1}(\Sigma_1 + \mu_1\mu_1^T - 2\mu_2\mu_1^T + \mu_2\mu_2^T)) \\
&\quad = \frac{1}{2}(log\frac{|\Sigma_2|}{|\Sigma_1|} - n + tr(\Sigma_2^{-1}\Sigma_1) + tr(\mu_1^T\Sigma_2^{-1}\mu_1 - 2\mu_1^T\Sigma_2^{-1}\mu_2 + \mu_2^T\Sigma_2^{-1}\mu_2)) \\
&\quad = \frac{1}{2}(log\frac{|\Sigma_2|}{|\Sigma_1|} - n + tr(\Sigma_2^{-1}\Sigma_1) + (\mu_2-\mu_1)^T\Sigma_2^{-1}(\mu_2-\mu_1)
\end{aligned} \tag{A.28}$$

These are the rules used in the derivation:

$$
\begin{aligned}
&log(ab) = log(a) + log(b)\\
&log(e^x) = x\\
&log\frac{a}{b} = loga - logb\\
&(a-b)^2 = a^2 + b^2 - 2ab\\
&|AB| = |A||B| \implies |A^n| = |A|^n\\
&cov(x,x) = var(x)E[(x-\mu)(x-\mu)^T] = E[xx^T] - \mu\mu^T\\
&|\Sigma|^{1/2} = |\Sigma^{1/2}| = \frac{1}{2}|\Sigma| \text{ since } \Sigma = \sigma^2 I \text{ is diagonal}\\
&tr(A) = \sum_{i=0}^{n} a_{ii} \qquad \text{(A.29)}\\
&tr(c) = c, c \in R\\
&tr(I) = n\\
&tr(A) = tr(A^T)\\
&tr(AB) = tr(BA)\\
&tr(A+B) = tr(A) + tr(B)\\
&tr(ABC) = tr(BCA) = tr(CAB)\\
&tr(a^Ta) = tr(aa^T)\\
&x^TAx = tr(x^TAx) = tr(xx^TA) = tr(Axx^T) \text{ (”trace trick”)}
\end{aligned}
$$

A.3.4 Closed Form Derivation of KL Divergence between Multivariate Gaussian and Standard Normal

We showed the following derivation of KL divergence between two multivariate Gaussian distributions:

$$KL(P_1||P_2) = E[logP_1 - logP_2] = \frac{1}{2}(log\frac{|\Sigma_2|}{|\Sigma_1|} - n + tr(\Sigma_2^{-1}\Sigma_1) + (\mu_2-\mu_1)^T\Sigma_2^{-1}(\mu_2-\mu_1) \qquad \text{(A.30)}$$

Suppose $P_1 = q(z|x) = N(z|\mu, \Sigma = \sigma^2 I)$ and $P_2 = P(z) = N(0, I)$ so our VAE uses $\mu_1 = \mu, \Sigma_1 = \Sigma, \mu_2 = \vec{0}, \Sigma_2 = I$.

The KL divergence between such a multivariate Gaussian distribution and standard normal distribution has a closed form:

$$
\begin{aligned}
&KL(q(z|x)||p(z|x)) \\
&= \frac{1}{2}(log\frac{|\Sigma_2|}{|\Sigma_1|} - n + tr(\Sigma_2^{-1}\Sigma_1) + (\mu_2 - \mu_1)^T\Sigma_2^{-1}(\mu_2 - \mu_1) \\
&= \frac{1}{2}[log\frac{|I|}{|\Sigma|} - n + tr(I^{-1}\Sigma) + (\vec{0} - \mu)^T I^{-1}(\vec{0} - \mu)] \\
&= \frac{1}{2}[log\frac{1}{|\Sigma|} - n + tr(\Sigma) + \mu^T\mu] \\
&= \frac{1}{2}[-log|\Sigma| - n + tr(\Sigma) + \mu^T\mu] \\
&= \frac{1}{2}[-log\prod_i \sigma_i^2 - n + \sum_i \sigma_i^2 + \sum_i \mu_i^2] \\
&= \frac{1}{2}[-\sum_i^n log\sigma_i^2 - n + \sum_i \sigma_i^2 + \sum_i \mu_i^2] \\
&= \frac{1}{2}[-(\sum_i log\sigma_i^2 + 1) + \sum_i \sigma_i^2 + \sum_i \mu_i^2] \\
&= -\frac{1}{2}[\sum_{i=1}^n log\sigma_i^2 + 1 - \sum_i \sigma_i^2 - \sum_i \mu_i^2] \\
&= -\frac{1}{2}\sum_{i=1}^n log\sigma_i^2 + 1 - \sigma_i^2 - \mu_i^2
\end{aligned}
\tag{A.31}
$$

In PyTorch, that corresponds to the loss function:

$$
-\frac{1}{2}\ \text{torch.sum(logvar + 1 - logvar.exp() - mean.pow(2))} \tag{A.32}
$$

The following rules were used:

$$
\begin{aligned}
&I = I^{-1} \\
&log(1) = 0 \\
&log\frac{a}{b} = loga - logb \\
&|\Sigma| = \prod_i \sigma_i^2 \text{ diagonal covariance matrix} \\
&\sum_i = 1^n 1 = n
\end{aligned}
\tag{A.33}
$$

A.3.5 MDRNN Loss: GMM Log Likelihood Function with logsumexp trick

$$
\begin{aligned}
&\mathbb{LL}(\theta|Z) \\
&\quad = \mathbb{LL}(\pi, \mu, \Sigma|z_1, \ldots, z_n) \\
&\quad = logp(z_1, \ldots, z_n|\pi, \mu, \Sigma) \\
&\quad = log(\prod_{i=1}^{n} p(z_i|\pi, \mu, \Sigma)(\text{ IID }) \\
&\quad = \sum_{i=1}^{n} logp(z_i|\pi, \mu, \Sigma) \\
&\quad = \sum_{i=1}^{n} log(\sum_{k=1}^{K} \pi_k \cdot N(z_i|\mu_k, \Sigma_k)) \\
&\quad = \sum_{i=1}^{n} log(\sum_{k=1}^{K} A_k),\ A_k = \pi_k N(z_n|\pi_k, \Sigma_k) \\
&\quad = \sum_{i=1}^{n} log(\sum_{k=1}^{K} exp(logA_k)) \\
&\quad = \sum_{i=1}^{n} [logA_m + log(\sum_{k=1}^{K} exp(logA_k - logA_m)],\ A_m = maxA_k, k \in 1, \ldots, K \\
&\quad = \sum_{i=1}^{n} [log\pi_m - \frac{D}{2} log 2\pi - \frac{1}{2} log|\Sigma_m| - \frac{1}{2} \sum_{i=1}^{n} (x_i^m - \mu_m)^T \Sigma_m^{-1} (x_i^m - \mu_m)] + \cdots \\
&\quad + \sum_{i=1}^{n} [log(\sum_{k=1}^{K} exp(logA_k - logA_m)]
\end{aligned}
\tag{A.34}
$$

A.4 Experiments

A.4.1 NTBEA Tuning: Planning Parameter Explanations

Horizon - A fixed value that defines how far the planning algorithms plans ahead. We decided to cap the upper bound to 20 as it would provide a somewhat far lookahead while maintaining a low run time during tuning.

Generations - A fixed value that defines the number of iterations a population (RHEA) or individual (RMHC) is evolved and evaluated. Similarly to the horizon parameter, the upper bound is capped to 20 generations to maintain a feasible run time during tuning.

Shift Buffer - Population management technique introduced by (Gaina, Simon M. Lucas, and Pérez-Liébana 2013). If Shift Buffer is enabled, each individual's final genome at timestep t is retained at timestep $t + 1$ where the first action is popped, and a new random action is appended to the end of the genome. Otherwise, if Shift Buffer is disabled, the population (RHEA) or individual (RMHC) is randomly initialized for timestep $t + 1$. Shift Buffer's purpose is to minimize loss of information gained from previous planning steps by avoiding to repeat the entire search process from scratch at each game step.

Fitness assignment - The undiscounted sum of simulated reward when evaluating an individual is assigned as the fitness. Arguably, discounting the rewards while prioritizing immediate rewards may likewise yield promising results since the world model becomes more uncertain when looking further into the future. We did run some manual tests with discounted rewards but did not see any significant improvements. Thus we decided to use the undiscounted implementation due to simplicity.

Mutation Type - There are 3 types of mutation operators that have been implemented.

Uniform-1 picks uniformly one single action from the genome and mutate it with another random action.

Uniform-All goes through each action with a mutation probability of $1/L$ where L is the length of the horizon (number of actions).

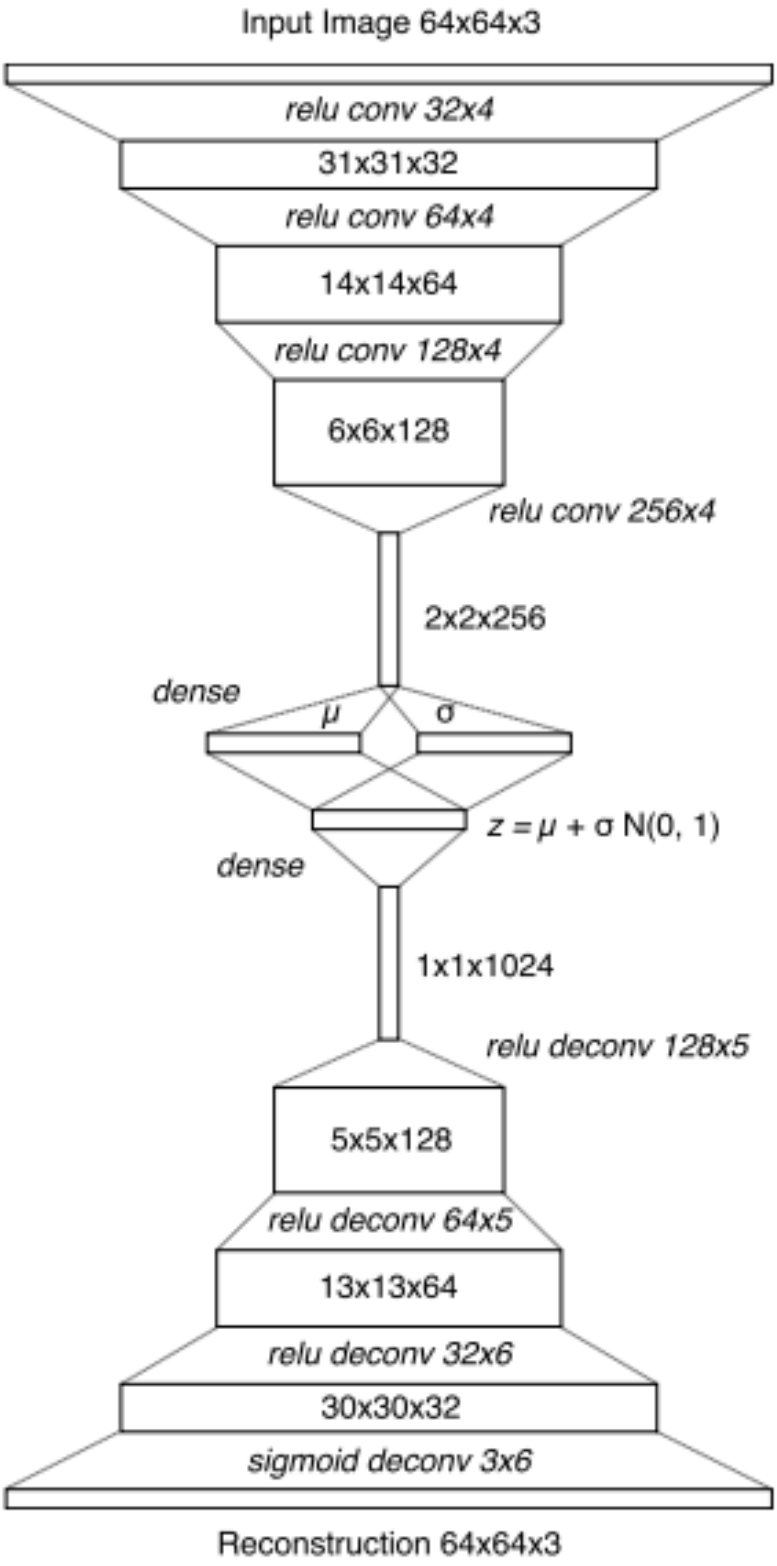


Figure A.4: ConvVAE Architecture (Ha and Schmidhuber 2018a)

Uniform-Subset selects uniformly N number of actions to mutate where N is a random integer between 1 and L that defines the size of the subset. The mutation operator is only applicable for RHEA if the genetic operator is *Mutation* or *Crossover + Mutation*; otherwise the mutation operator will not be in use.

Population - RHEA - Fixed value that defines the number of individuals within a population that is evaluated and evolved with RHEA. Since each individual is evaluated in parallel, the upper bound is set to 16, which is the number of logical threads our system has to avoid CPU scheduling.

Genetic Operator - RHEA - There are 3 genetic operators that are implemented.

Crossover will only use crossover to introduce diversity in the population for each generation.

Mutation will only use mutations to introduce diversity in the population for each generation.

Crossover + Mutation uses both mutation and crossover to introduce diversity in the population.

Selection Type - RHEA - The Selection operator defines how two individuals are selected for crossover to produce an offspring.

Uniform does not put any pressure on fitness but only selects two parents uniformly amongst the population for crossover.

Tournament selects a percentage of the population ($t = 0.5$) uniformly without replacement and chooses the two best individuals as parents.

Roulette selection chooses individuals with probabilities equal to their fitness. Individuals with high fitness have an increased probability of being selected. However, the fitness of an individual can be negative due to negative rewards; thus, we normalize all individuals' fitness to be nonnegative by summing the absolute value of the smallest fitness to all the other individuals' fitness. Yet, the worst performing individual would have a fitness of zero after normalization, hence by adding a small positive value $y = 0.01$ to all individuals' fitness ensures that there is a slight chance for the worst performing individual of being selected.

Rank selection sorts the individuals according to their fitness. Each individual is then assigned a rank such that the lowest fitness individual would have rank 1, second lowest would have rank 2 and up to n where n is the rank of the individual with the highest fitness. Individuals are then chosen based on the probabilities of their ranks and not by their fitness unlike in roulette selection. This reduces the selection pressure by minimizing the differences in fitness. Individuals in roulette selection with extremely high fitness values are most likely getting constantly picked, thus undermining the rest of the population. Rank selection ignores extreme fitness values due to ranks being used as probabilities instead of fitness.

Crossover Type - RHEA - The Crossover operator defines how two parents' genomes are combined where the outcome is the offspring's genome.

Uniform crossover selects for each action either of the parents' actions with equal probability.

n-point (1, 2-point) crossover randomly selects n fixed points, which would split the parents' genomes into n+1 subsections. The genome of the offspring is then formed by alternating between the parents' subsections. We use 1 and 2 points as options with n-point crossover. Finally the crossover operator is only applicable when the genetic operator is either *Crossover* or *Crossover-Mutation*.

Preliminary Tests

Model D and E Passive Rollouts

Model D and **Model E** was an initial attempt to improve the planning by adding 5k passive rollouts to see if it was possible to reinforce the negative reward signal of standing still due to the high reward signal of passiveness seen in Model C.

Disappointingly, the additional passive data dropped the performance of RMHC and RHEA. RMHC saw up to 85 percent performance drop and RHEA had up to 42 performance drop. Further inspection showed that both Model D and E still had an incorrect rank order of rewards which is likely the reason of the performance drop as shown on figure A.5.

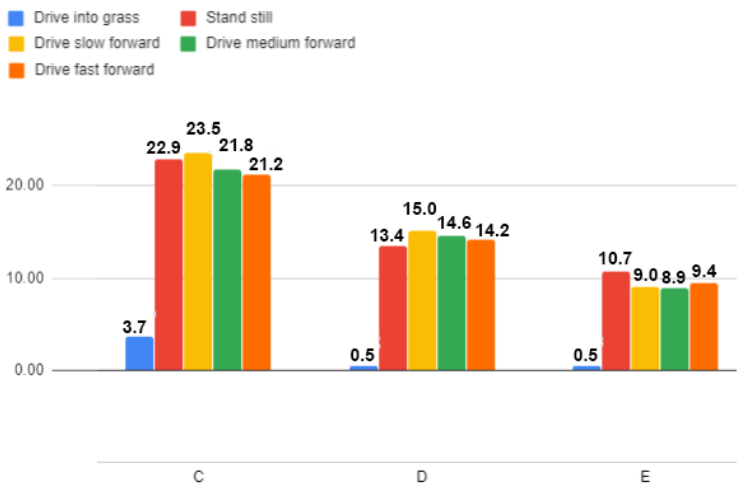


Figure A.5: Histogram of model C-E average reward signal when driving at different speeds. The additional passive data (D and E) proved to not mitigate the incorrect rank order from model C but had worsened the overall performance of the agent

Model J-K: Additional turn data

Based on the previous results, model I performed best in the preliminary results. However, the agent still struggled with right turns, and S-turns, which is a combination of left and right turns as it seemed to consistently turn left. Hence we tried to improve Model I by including additional rollouts that only contained turn sequences with the expectation that it would improve the reward signal during turns. Disappointingly, the additional turn did not improve anything but instead dropped the planning performance. Manually parameter tuning of the planning algorithms with Model I did not yield any significant results.

However, later beyond the preliminary experiments, we implemented subset mutation which mutates a random subset of genes. Using the new mutation method with RHEA, the agent was able to complete the right turns with Model I.

Unsurprisingly, this proved that the preliminary parameter configuration for RHEA led to the agent getting frequently stuck in a local maxima space with Model I. The change in mutation type allowed the agent to escape the local maxima space by using a mutation method that may have introduced greater diversity within the population. Thus the preliminary planning parameters are not a silver bullet configuration that is optimally applicable to all the models which is later discussed in the next section.

Bibliography

McCloud, Scott (1993). *McCloud: Understanding Comics*. URL: https://en.wikipedia.org/wiki/Understanding_Comics.

Yannakakis, Georgios N. and Julian Togelius (2018). *Artificial Intelligence and Games*. URL: http://gameaibook.org/book.pdf.

Mnih, Volodymyr et al. (2014). *Playing Atari with Deep Reinforcement Learning*. URL: https://arxiv.org/pdf/1312.5602v1.pdf.

Schrittwieser, Julian et al. (2020). *Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model*. URL: https://arxiv.org/pdf/1911.08265.pdf.

Russell, Stuart J. and Peter Norvig (2009). *Artificial Intelligence - A Modern Approach (3rd Edition)*. Harlow, Essex: Pearson.

Bolander, Thomas (2019). *What is AI - and where is it heading? Part II: Symbolic and subsymbolic AI*. URL: http://www.imm.dtu.dk/~tobo/dighumlab2.pdf.

Mitchell, Tom M. (1997). *Machine learning*. New York, United States: McGraw-Hill.

Bishop, Cristopher M. (2006). *Pattern Recognition and Machine Learning*. Manhattan, New York City: Springer.

Fortmann-Roe, Scott (2012). *Understanding the Bias-Variance Tradeoff*. URL: http://scott.fortmann-roe.com/docs/BiasVariance.html.

Kapoor, Sanyam (2018). *Policy Gradients in a Nutshell*. URL: https://towardsdatascience.com/policy-gradients-in-a-nutshell-8b72f9743c5d.

Sutton, Richard S. and Andrew G. Barto (2018). *Reinforcement Learning - An Introduction (2nd Edition)*. Cambridge, Massachusetts: MIT Press.

Kochenderfer, Mykel J. (2014). *Decision Making uner Uncertainty*. URL: https://web.stanford.edu/~mykel/pomdps.pdf.

Silver, David (2020). *Lecture 8: Integrating Learning and Planning*. URL: https://www.davidsilver.uk/wp-content/uploads/2020/03/dyna.pdf.

Github, Open AI Gym (2019). *Make a copy of current env*. URL: https://github.com/openai/gym/issues/1292.

Eiben, A.E. and J.E. Smith (2015). *Introduction to Evolutionary Computing*. Manhattan, New York City: Springer.

Perez, Diego, Simon M. Lucas, et al. (2013). *Rolling Horizon Evolution versus Tree Search for Navigation in Single-Player Real-Time Games*. URL: https://dl.acm.org/doi/pdf/10.1145/2463372.2463413.

Gaina, Raluca D., Simon M. Lucas, and Diego Pérez-Liébana (2017). *Population Seeding Techniques for Rolling Horizon Evolution in General Video Game Playing*. URL: https://arxiv.org/pdf/1704.06942.pdf.

Gaina, Raluca D., Simon M. Lucas, Diego Pérez-Liébana, et al. (2017). *Introducing real world physics and macro-actions to general video game ei*. URL: https://www.researchgate.net/publication/320745695_Introducing_real_world_physics_and_macro-actions_to_general_video_game_ai.

Deepa, S.N. Sivanandam S.N. (2008). *Introduction to Genetic Algorithms*. Manhattan, New York City: Springer.

Kunanusont, Kamolwan, Raluca D. Gaina, and et.al. (2017). *The N-Tuple Bandit Evolutionary Algorithm for Automatic Game Improvement*. URL: https://arxiv.org/pdf/1705.01080.pdf.

Lucas, Simon M, Jialin Liu, and Diego Perez-Liebana (2018). *The N-Tuple Bandit Evolutionary Algorithm for Game Agent Optimisation*. URL: https://arxiv.org/pdf/1802.05991.pdf.

Dalgaard, Randi (2019). *CS6501: Deep Learning for Visual Recognition - Neural Networks (University of Virginia)*. URL: https://slideplayer.com/slide/16362305/.

Alpaydin, Ethem (2014). *Introduction to Machine Learning*. Cambridge, Massachusetts: MIT Press.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. Cambridge, Massachusetts: MIT Press.

Weng, Lilian (2018a). *From Autoencoder to Beta-VAE*. URL: https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html.

Yau, Jeffrey (2018). *Vanilla Reccurrent Neural Network (RNN)*. URL: https://www.slideshare.net/databricks/time-series-forecasting-using-recurrent-neural-network-and-vector-autoregressive-model-when-and-how-with-jeffrey-yau.

Buesing, Lars et al. (2018). *Learning and Querying Fast Generative Models for Reinforcement Learning*. URL: https://arxiv.org/pdf/1802.03006.pdf.

Ha, David and Jürgen Schmidhuber (2018a). *World Models*. URL: https://arxiv.org/pdf/1803.10122.pdf.

– (2018b). *World Models (Github)*. URL: https://worldmodels.github.io/.

Risi, Sebastian and Kenneth O. Stanley (2019). *Deep Neuroevolution of Recurrent and Discrete World Models*. URL: https://arxiv.org/pdf/1906.08857.pdf.

Ha, David, Danijar Hafner, et al. (2019). *Learning Latent Dynamics for Planning from Pixels*. URL: https://arxiv.org/abs/1811.04551.

Hafner, Danijar et al. (2020). *Dream to Control: Learning Behaviors By Latent Imagination*. URL: https://arxiv.org/pdf/1912.01603.pdf.

Tallec, Corentin, Léonard Blier, and Diviyan Kalainathan (2018). *Pytorch implementation of the "WorldModels"*. URL: https://github.com/ctallec/world-models.

Bamford, Chris (2019). *NTBEA Implementation in python*. URL: https://github.com/bam4d/NTBEA.

Kingma, Diederik P. (2014). *Auto-Encoding Variational Bayes*. URL: https://arxiv.org/pdf/1312.6114.pdf.

Doersch, Carl (2016). *Tutorial on Variational Autoencoders*. URL: https://arxiv.org/pdf/1606.05908.pdf.

Weng, Lilian (2018b). *From Autoencoder to Beta-VAE*. URL: https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html.

Graves, Alex, Santiago Fernández, and Jürgen Schmidbauer (2013). *Multi-Dimensional Recurrent Neural Networks*. URL: https://arxiv.org/pdf/0705.2011.pdf.

Ellefsen, Kai Olav, Charles Patrick Martin, and Jim Torresen (2019). *How do MD-RNNs predict the future?* URL: https://arxiv.org/pdf/1901.07859.pdf.

Gaina, Raluca D., Sam Devlin, et al. (2020). *Rolling Horizon Evolutionary Algorithms for General Video Game Playing*. URL: https://arxiv.org/pdf/2003.12331.pdf.

Perez, Diego, Simon Lucas, et al. (2012). *A Survey of Monte Carlo Tree Search Methods*. URL: http://www.diego-perez.net/papers/MCTSSurvey.pdf.

Dettmers, Tim (2019). *Which GPU(s) to Get for Deep Learning*. URL: https://timdettmers.com/2019/04/03/which-gpu-for-deep-learning/.

Gregor, Karol et al. (2019). *Shaping Belief States with Generative Environment Models for RL*. URL: https://arxiv.org/pdf/1906.09237.pdf.

Kanerva, Pentti (1988). *Sparse Distributed Memory*. Cambridge, Massachusetts: MIT Press.

Altosaar, Jaan (2020). *Tutorial - What is a variational autoencoder?* URL: https://jaan.io/what-is-variational-autoencoder-vae-tutorial/.

Blei, David M., Alp Kucukelbir, and Jon D. McAuliffe (2018). *Variational Inference: A Review for Statisticians*. URL: https://arxiv.org/pdf/1601.00670.pdf.

Gaina, Raluca D., Simon M. Lucas, and Diego Pérez-Liébana (2013). *Rolling Horizon Evolutionary Algorithms for General Video Game Playing*. URL: https://www.researchgate.net/publication/340271118_Rolling_Horizon_Evolutionary_Algorithms_for_General_Video_Game_Playing.