World Models and Predictive Methods in Deep Reinforcement Learning: A Survey

St John M.M. Grimbly

GRMSTJ001
Supervisor: Dr Jonathan Shock
University of Cape Town

July 5, 2020

Abstract

This paper investigates and develops techniques and theory for implementing state-of-the-art model-based reinforcement learning techniques. First, neuroscientific ideas of prediction and inference are developed. The basics of deep learning and the reinforcement learning problem are then discussed. Deep model-free and model-based methods are developed in the context of later discussions. Theory of latent representation and predictive methods is presented before discussing advanced state-of-the-art models. Emphasis is placed on how these agents apply a broad range of the previously developed techniques. A discussion and implementation of a state-of-the-art agent (MuZero) in Python follows. Finally, the direction of the field as a whole is discussed.

Contents

1 Introduction **3**

2 Theory of Mind **3**
2.1 Predictive Coding & Predictive Processing . . . 4
2.2 Free Energy and Active Inference . . . 4

3 Theory of Learning **7**
3.1 The Reinforcement Learning Problem . . . 7
3.2 Bellman Equations . . . 8
3.3 Exploration and Exploitation . . . 9
3.4 Trade-offs . . . 10
3.5 Benchmarking Algorithms . . . 11
3.6 Deep Learning . . . 12

4 Model-Free Methods **13**
4.1 Policy-based Methods . . . 13
4.1.1 Direct Policy Differentiation . . . 14
4.1.2 REINFORCE . . . 14
4.1.3 Reducing Variance . . . 15
4.2 Actor-Critic Methods . . . 16
4.2.1 Proximal Policy Methods . . . 17
4.3 Value-Based Methods . . . 17
4.3.1 Dynamic Programming . . . 18
4.3.2 Q-Learning . . . 18

5 Model-Based Methods **19**
5.1 Planning without Derivatives . . . 20
5.2 Planning with Derivatives . . . 22
5.3 Learning without a Policy . . . 23
5.4 Learning with a Policy . . . 26

6 Representation & Advanced Predictive Methods **28**
6.1 Latent Representation in MBRL . . . 28
6.2 Contrastive Predictive Coding (CPC) & InfoNCE . . . 29
6.3 Predictive Coding for Sparse Rewards . . . 30

7 State-of-the-Art Models **31**
7.1 Learning By Imagination . . . 31
7.2 Free Energy of Expected Future . . . 35
7.3 SimPLe . . . 37
7.4 MuZero . . . 39

8 Simulations & Implementation **41**

9 Conclusions **45**

1 Introduction

The field of reinforcement learning (RL) is at a crossroads between optimal control, animal psychology, artificial intelligence and many other technical fields including game theory [1] and finance [2]. It has seen a surge of interest with high profile successes, such as the dominance of AlphaGo [3] over 18 time world Go champion, Lee Sedol. The idea of reinforcement learning emerged out of the study of animal psychology and the concept of trial-and-error learning. Edward Thorndike coined the *Law of Effect* in 1898, introducing the idea of reinforcement in the context of learning. He proposed that if a stimulus is followed by some successful response (reward in RL), then that stimulus-response connection will be strengthened [4]. Recent successes of the theory of reinforcement learning at solving complex computational tasks have seen a resurgence in the field. This paper develops and discusses advanced topics in reinforcement learning in the context of modern advancements and interest.

Reinforcement learning can be seen as a subset of machine learning alongside supervised and unsupervised methods. Whereas supervised and unsupervised learning deal with labelled and unlabelled static data respectively, reinforcement learning deals with how an agent should learn to make decisions in an abstract environment with a potentially changing landscape. More recently, the field has been subdivided into unsupervised RL and supervised RL. In supervised RL we explicitly program the reward structure from which the agent learns. In unsupervised RL the agent can learn from intrinsic motivations, such as curiosity. Most success in the field thus far has been in the supervised approach to reinforcement learning, and for this reason this paper will focus on these approaches.

RL is, for the most part, a formalism for the idea of trial-and-error learning. At each time-step an agent in an environment with a well defined state can perform some action. This action triggers a response from the environment in which the environment rewards the agent for taking this action and moves the agent to a new state. Naturally, the goal of the agent is to maximise the expected reward over time - the return. Further complexity arises in that the agent does not explicitly know how the environment will react and may only have partial information about what state it is currently in. For example, a human playing Montezuma's Revenge (an Atari game used for benchmarking RL agents) can only see a portion of the environment and cannot be sure about what exact state the surrounding environment is in. It must attempt to make optimal decisions with imperfect information.

Reinforcement learning algorithms are usually categorised as falling into one of two categories - model-free reinforcement learning (MFRL) or model-based reinforcement learning (MBRL). In MFRL an agent directly learns a value function or policy for interacting in the environment by observing rewards and state transitions, while in MBRL the agent uses its interactions with the environment to learn about the environment's dynamics, and thereby model it. Model-free methods have enjoyed success in areas such as robotics and computer games, however they are plagued by high sample inefficiencies [5]. These sample inefficiencies often limit the application of these methods to simulated environments which are less complex than real world dynamics. Learning a model of the environment allows an agent to significantly reduce the dimensionality and complexity of the environment it interacts with, allowing for much improved sample efficiency and performance in many simulated and real world scenarios. By learning a model of the environment, an agent can apply well known supervised learning techniques to plan an optimal strategy for maximising reward. Learning a good model of the environment is a non-trivial task as modelling errors can prove crippling to many tasks. This is the problem of model-bias.

This paper starts by developing the the theory of mind and reinforcement learning that will be used later in the paper. We then discuss popular model-free reinforcement learning methods and their pitfalls, especially those that make use of deep neural networks. From this we develop the theory of model-based reinforcement learning and develop critical notions of planning and learning. State-of-the-art methods which make use of latent representations, prediction, planning and learning with a model are then discussed. Finally a state-of-the-art agent which make use of some of these techniques will be implemented and discussed in detail.

2 Theory of Mind

As discussed in the introduction, the field of reinforcement learning is intimately tied to fields of psychology and computational neuroscience, among others. Modern theories argue prediction plays a vital role in human perception and inference, and is especially apparent in emerging theory of active inference [6]. It is argued that the active inference framework (AIF) makes it possible to bridge the fields of computational neuroscience and artificial intelligence by revealing common underpinnings of the theories, and using combined theory to overcome limitations. Using the principle of free energy minimisation, AIF provides a rigorous information-theoretic and naturalistic foundation for problems involving inference. Key to this formulation, a generative

model allows an agent to *generate* a model or predictions for novel environments based on past experiences of related environments. In other words, a generative model is a description of how an agent can leverage its prior experience to form a model of a foreign environment. This generated model can then be updated in the Bayesian sense as more information is acquired.

2.1 Predictive Coding & Predictive Processing

Predictive processing is a theory of mind in which the brain attempts to form and update a model of the environment. This model, as in model-based RL, can be used to make predictions of expected sensory input and can be compared to observed sensory input. This difference in prediction and true value can be used to revise and adapt the model, thereby improving future predictions. Effectively, this minimises the *surprise*. Predictive coding asserts that perception is largely constrained by prior predictions. That is, sensory signals only inform and shape perception insofar as they cause surprise to the system. In other words, signals inform perception given sensory input is different from sensory prediction.

Figure 1 shows an example of how the predictive coding model passes predictions down the network, and differences in prediction and reality are passed and filtered back up.

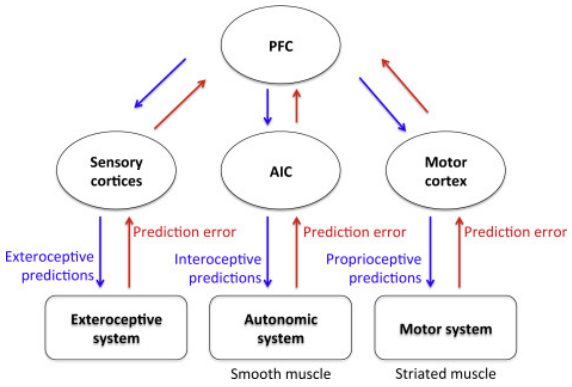


Figure 1: This diagram displays an example of the flow of information in the top-down theory of mind - predictive coding. Exteroceptive predictions are generated by the model. These predictions are compared to sensory input. Discrepancy between the model predictions and sensory input is passed back up in the form of a prediction error. Refer to [7].

For a theory of mind to be feasible, it must satisfy basic biological constraints. These constraints, as Bogacz [8] describes, can be summarised as:

1. **Local computation** - wherein a neuron only performs computation on in relation to the activity of neuron input and associated weights.
2. **Local plasticity** - synaptic plasticity is only a function of the activity of pre-synaptic and post-synaptic neurons.

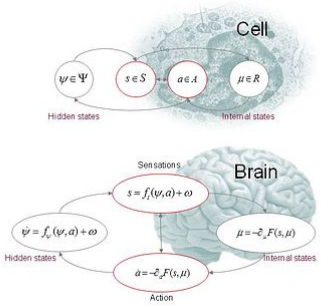
2.2 Free Energy and Active Inference

The free energy principle is an attempt at explaining how a self-organising system can maintain a non-equilibrium stead-state that is compatible with its existence - a phenomenon which seems to be in contradiction with the second law of thermodynamics [1]. Biological systems are said to minimise some *free energy function* of their internal states. Crucially, the theory uses this principle as just that - a *principle*. No evidence is used to promote the framework itself. Applications, however, of the framework require evidence and motivation for its use. Active inference is the notion of embodied perception in neuroscience as explained by the free energy principle. Active inference with a generative model ties together ideas of proprioception and interoception by telling us that agents must actively seek out sensations to increase their knowledge of the world. In some sense, the agent actively coordinates its sensors to control and obtain the sensations it seeks more knowledge of to further minimise its surprise. Formally, active inference occurs as a tuple $(\Omega, \Psi, S, A, R, q, p)$, where each of the parameters are defined as follows:

- Ω is the sample space of random fluctuations.

[1] The second law of thermodynamics states that the total entropy of an isolated system can never decrease over time. In fact, entropy tends to increase if any process within the system is non-reversible.

- $\Psi : \Psi \times A \times \Omega \to R$ is a hidden or external state which causes sensory states dependent on the action selected.
- $S : \Psi \times A \times \Omega \to R$ are the sensory states. This is a stochastic mapping from the action and hidden states.
- $R : R \times S \to R$ are the internal states that cause action and are dependent on sensory states.
- $p(s, \psi|m)$ is the generative density over sensory and hidden states conditioned on a generative model, m.
- $q(\psi|\mu)$ is the variational density over hidden states parameterised by an internal state μ.



Using this formalism we can formalise the objective of an inference agent. The theory of active inference tells us that the agent wishes to maximise its model evidence $p(s|m)$ or minimise the surprise $\log p(s|m)$. We can tractably calculate and apply an upper bound on this surprise - the variational free energy. Internal states thus minimise the free energy of the system. Interestingly, all Bayesian inference can be cast as a free energy minimisation problem. Minimising free energy with respect to the internal states corresponds to a optimisation over the Kullback-Leibler divergence between the variational and posterior densities over hidden states. Furthermore, free energy minimisation can be related to the *principle of minimum redundancy* in information theory to explain optimal behaviour. There are many further applications and related ideas in thermodynamics, neuroscience, optimal control and game theory for example. This generality is precisely why the principle is so powerful and fundamental.

One might start to wonder why all of this is so important in the context of reinforcement learning. The reason for this is the ideas of active inference can be applied to the classical studies of optimal control, which forms the foundation for much of the work in reinforcement learning. Predictive methods, as mentioned in the introduction, have recently proved to be highly performant in many model-based applications of reinforcement learning and artificial intelligence in general. Active inference relates to optimal control by replacing the value function (as we will discuss in detail in the context of RL) with a Bayesian prior belief function about the state transitions of the system,

$$f = \Gamma \cdot \nabla V + \nabla \times W,$$

where V is the scalar value function and W is the vector value function. This is in contrast to optimal control where the system is assumed to be in detailed balance (W has curl-free flow). The priors over the state transitions induce priors over the states themselves, which form a solution to the forward Kolmogorov equations. In *'what is optimal about motor control?'*[9], Friston argues that active inference may be a better way to understand and frame the problem of perception as forward models used in motor control are not the same as the generative models needed for perception in the brain.

We are now ready to work through an extended example to explain and explore some of the details of this method.

1D Perception Example

It is helpful and instructive to run through an motivating example presented in detail in Bogacz's tutorial on the free energy framework [8]. This example works through the mathematics of perception in the predictive coding and free energy framework, and uses this simplified approach to motivate the framework as viable and neurologically plausible.

We start by framing the problem. Let us consider the problem of an organism trying to infer the size of a nearby piece of food from the observation of light intensity. This observation is assumed to be 1-dimensional. That is, it is a single observation. Formally, we define the parameters of interest as follows:

i. Let v be the food size

ii. Let g be a nonlinear function relating size to light intensity

iii. Then u is a noisy estimate of the light intensity s.t. $u \sim N(g(v), \Sigma_u)$

To compute the food size explicitly, our organism would need to calculate $p(v|u) = \frac{p(v)p(u|v)}{p(u)}$. This involves calculating $p(u) = \int p(v)p(u|v)$. This integral could be difficult to compute since the posterior distribution of food size given some observation could be non-trivial. Since we cannot rely on summary statistics to describe the distribution, we turn to an approximate method.

Consider now computing the most likely food size, which we shall denote as ϕ. This maximises $p(v|u)$ - an approach which should be more tractable than finding the whole posterior distribution. The posterior density is then $p(\phi|u) = \frac{p(\phi)p(u|\phi)}{p(u)}$, where $p(u)$ does not depend on ϕ. To find ϕ that maximise the posterior we maximise the log of the distribution, since the logarithm is a strictly increasing function.

$$\begin{aligned} F &= \ln(p(\phi)p(u|\phi)) = \ln(p(\phi)) + \ln(p(u|\phi)) \\ &= \frac{1}{2}(-\ln(\Sigma_p) - \frac{(\phi - v_p)^2}{\Sigma_p} - \ln(\Sigma_u) - \frac{(u - g(\phi))^2}{\Sigma_u}) + C \end{aligned}$$

We then update ϕ proportionally to the direction of steepest ascent,

$$\frac{\partial F}{\partial \phi} = \frac{v_p - \phi}{\Sigma_p} + \frac{u - g(\phi)}{\Sigma_u} g'(\phi).$$

We notice $\epsilon_p = \frac{\phi - v_p}{\Sigma_p}$ and $\epsilon_u = \frac{u - g(\phi)}{\Sigma_u}$ are *prediction errors*. That is, they are simply weighted differences between expected values and observed values. Let us consider the feasibility of a neural implementation. Assume v_p, Σ_p, Σ_u are encoded in strength of synaptic connection, and that $\phi, \epsilon_p, \epsilon_u, u$ are encoded in activity of neurons. If we define the dynamics as follows,

$$\dot{\epsilon}_p = \phi - v_p - \Sigma_p \epsilon_p$$

$$\dot{\epsilon}_u = u - g(\phi) - \Sigma_u \epsilon_u$$

we can compute the prediction errors in a feasible, Hebbian manner. Considering $\dot{\epsilon}_p \to 0$ and $\dot{\epsilon}_u \to 0$ and notice the dynamics converge to the prediction errors themselves.

The question now is, how do we go about *learning*. As in 2.1, we know that least surprise $\iff$ most expected. Here, we want to maximise $p(u)$ but we said this was not feasible. It is simpler to maximise the joint distribution $p(u, \phi) = p(\phi)p(u|\phi)$ by maximising $F = \ln p(u, \phi)$. Consider some distribution $q(v)$ which approximates the posterior, $p(v|u)$. To ensure a good approximation, we want to minimise the difference between these distributions. The Kullback-Leibler divergence is one measure of the dissimilarity between distributions. Applying the definition of KL-divergence,

$$\begin{aligned} KL(q(v), p(v|u)) &= \int q(v) \log \frac{q(v)p(u)}{p(u, v)} dv \\ &= \int q(v) \log \frac{q(v)}{p(u, v)} dv + \int q(v) dv \ln p(u) \\ &= \int q(v) \log \frac{q(v)}{p(u, v)} dv + \ln p(u) \end{aligned}$$

Defining $-F = \int q(v) \log \frac{q(v)}{p(u,v)} dv$ as the free energy, we get

$$KL(q(v), p(v|u)) = -F + \ln p(u)$$

where $\ln p(u)$ is independent of ϕ. Therefore, maximising F (minimising -F) gives the desired result. Thus, minimising the free energy - and thus surprise - ensures the approximation is good.

$$\begin{aligned} F &= -\ln p(u) + KL(q(v), p(v|u) \\ &= \underbrace{-\ln p(u)}_{\text{surprise}} + \underbrace{\int q(v) \log \frac{q(v)p(u)}{p(u, v)} dv}_{\text{posterior distribution}} \end{aligned}$$

It should now be clear how minimising suprise can reveal information about the state a system is in. This ties in well with the theory of optimal control and reinforcement learning. These links should become more clear as we develop the theory of reinforcement learning.

3 Theory of Learning

We are now ready to discuss the problem of how an agent can *learn* to solve some assigned problem. There are various approaches that shall be discussed in this paper, including model-based and predictive methods. In this section some important definitions and basic theory in the study of learning is introduced. Since this paper focuses mainly on modern reinforcement learning, the notation used sticks to the conventional style (in contrast to classic studies of optimal control for example) which will present itself in almost all modern research and practical implementations.

3.1 The Reinforcement Learning Problem

The problem of reinforcement learning (RL) is that of how to map states to actions so as to maximise some numerically encoded reward signal [10]. As discussed in the introduction, the reinforcement learning agent experiences a changing landscape based on the actions it takes and attempts to learn an optimal sequence of actions by applying the concept of trial-and-error with reinforcement theory. This problem can be formally stated in terms of Markov decision processes (MDP) within what Sutton and Barto [10] refer to as the agent-environment interface. MDPs are an appropriate formalism for the RL problem since many (especially classical) reinforcement learning algorithms make use of dynamic programming techniques where the exact mathematical model of the problem is available. In some sense, reinforcement learning is the study of approximate dynamic programming.

The agent-environment interface describes how an agent and the environment the agent is interacting with communicate and interplay. It is cyclical in nature, as shown in figure 2.

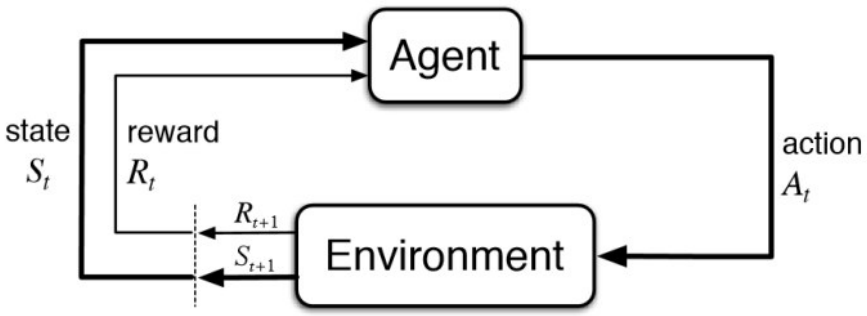


Figure 2: Diagram showing the flow of components in an MDP. An agent selects action A_t. The environment returns the next state and an associated reward to the agent, dependent on the selected action. The process then repeats [10].

We can begin to formalise the RL problem by first noting that the reward an agent receives is only dependent on the current state it is in, as well as the action it decides to take. In other words, the history of how the agent got to the current state is irrelevant. In statistical terms, we can say it has the Markov property. Of course, we can formulate problems in which the environment has some memory of the agent's decisions [11], but this is not the common approach in RL and shall not be discussed further.

Definition 3.1 *A state has the Markov property if for* $t = 1, 2, \ldots$, $P(X_{t+1} = i_{t+1} | X_t = i_t, \ldots, X_1 = i_1, X_0 = i_0) = P(X_{t+1} = i_{t+1} | X_t = i_t)$.

The agent exists in an environment of states which all have the Markov property. The states thus form a Markov chain.

Definition 3.2 *A discrete-time stochastic process is a Markov chain if every state has the Markov property. Formally, it is a tuple* (S, P) *of states* S *and transition probabilities* P.

Further, since moving from one state to another results in a reward, we can define a Markov reward process.

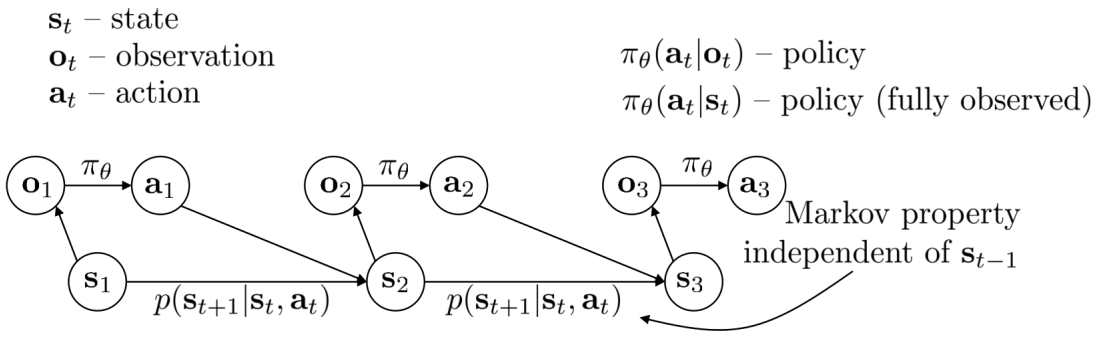
Definition 3.3 *A Markov reward process (MRP) is a Markov chain where rewards are associated with each state. Formally, it is a 4-tuple* (S, P, R, γ) *of states* S, *transition probabilities* P, *rewards* R, *and a discount factor* γ.

Further, the agent exists in the MRP and can make decisions. This leads to the definition of a Markov decision process (MDP) with which most RL problems can be formulated.

Definition 3.4 *A Markov decision process (MDP) is a Markov reward process with actions that can be taken. Formally, it is a 5-tuple (S, A, P, R, γ) of states S, actions A, transition probabilities P, rewards R, and a discount factor γ.*

In many modern problems an agent can only observe part or some representation of states in the state space. This requires the further abstraction of a partially observed Markov decision process (POMDP).

Definition 3.5 *A partially observed Markov decision process (POMDP) is a Markov decision process in which the agent cannot observe the underlying state. Instead, there are observations and associated observation probabilities which allow inference about the underlying state. Formally, it is a 7-tuple $(S, A, P, R, \Omega, O, \gamma)$ of states S, actions A, transition probabilities P, rewards R, observations Ω, observation probabilities O, and a discount factor γ.*



Objective Function

As stated in section 3.1, the aim in reinforcement learning is to maximise the expected return over some time horizon. Given some policy parameterised by some vector θ, we can reformulate this by finding the θ which maximises some *objective function*, $J(\theta)$, defined as follows.

$$J(\theta) = \mathrm{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right] \simeq \frac{1}{N} \sum_i \sum_t r(s_{i,t}, a_{i,t})$$

Thus, the goal is to find the parameterisation of our policy such that,

$$\theta^* = \arg\max_\theta \underbrace{\mathrm{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right]}_{J(\theta)}.$$

If we now consider the task of maximising this objective, taking the gradient seems like the natureal next step. This leads to the formulation of policy based methods which we discuss in section 4.

3.2 Bellman Equations

The Bellman equation is a fundamental necessary result that says the *value* of a decision problem at some discrete time step is given in terms of the payoff from the current step and the value of the remaining decisions. In other words, the Bellman equation breaks the optimisation of the value of a sequence of actions into a sequence of subproblems that are simpler to solve. This is done using the Bellman *principle of optimality* which states that:

Principle 3.1 *The Principle of Optimality states that an optimal policy has the property that no matter the initial state and decision, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*

This principle motivates the derivation of the Bellman Expectation Equation,

$$v_\pi(s) = \mathrm{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s \right].$$

We can similarly derive an expression for the state-action value,

$$q_\pi(s, a) = \mathrm{E}\left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a \right].$$

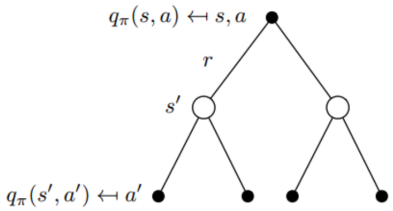


Figure 3: Backup diagram for state-action value.

This equation tells us that, under some policy π, we can determine the value of some state-action pair by observing the reward after taking the current action and then averaging over the expected reward from future actions. This allows us to define Optimal Value Functions for an MDP. The Optimal Value is simply the value function that yields the highest expected reward,

$$v_*(s) = \max_\pi v_\pi(s) \quad \text{and} \quad q_*(s,a) = \max_\pi q_\pi(s,a).$$

This, in turn, defines the Optimal Policy. An optimal policy is one which yields the optimal value function. Of course there can be more than one such policy. A policy π is said to be better than another policy π' if

$$\pi \geq \pi' \quad \text{if} \quad v_\pi(s) \geq v_{\pi'}(s), \forall s.$$

Then, formally, the optimal policy is

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a \in A} q_*(s,a) \\ 0 & \text{otherwise} \end{cases}.$$

Finally, we can define the Bellman Optimality Equation as the Bellman Expectation Equation where instead of taking the average of the actions, our agent selects the action with the max value. This is exactly what many reinforcement learning methods do to solve the reinforcement learning problem in terms of a defined MDP.

3.3 Exploration and Exploitation

One of the inherent problems an agent faces in some arbitrary environment is how to decide whether to explore and discover more of the world around it, or to rather apply what it already knows and exploit this knowledge to maximise the expected return. There is a trade-off since exploiting what we already know will likely lead to some amount of reward, but with imperfect information about the world around the agent there is a possibility that the agent is missing out on some large reward it does not yet know about. There is thus a problem of ensuring good exploration of the state-space while also ensuring at some point we apply what we have learned to ensure good reward. This problem becomes even harder when the environments dynamics are themselves a function of time, and so what we have learned in previous time-steps may become invalid as we progress. This is an active area of research and some possible solutions are presented and discussed in the context of model-based reinforcement learning in later sections.

Multi-armed bandit problems consist of of choosing among multiple discrete arms, each yielding some specific reward with some unknown probability. The common example is that of a slot machine. The reward function of the bandit is unknown, so an agent must interact with it to *learn* and approximate its reward function. At each point it must make the decision of whether to learn more about the machine or to exploit the knowledge it has gained. Three such strategies for how an agent can decide on an action are now discussed.

ϵ-greedy

Epsilon-greedy methods are the simplest methods for deciding on an action with some trade-off between exploring and exploiting. A greedy method is one in which the agent always, *greedily*, chooses the actions it believes will yield the highest reward. ϵ-greedy methods extend this idea to incorporate some non-greedy (exploratory) decisions with some probability $1 - \epsilon$. In practice these methods are surprisingly robust and performant, and show up in state-of-the-art methods. Some problems with ϵ-greedy methods is that it chooses the action to explore indiscriminately. That is, there is no weighting or rule for how to choose what action to explore. The exploration strategy itself may be poor in scenarios where random exploration is not ideal. In practise $\epsilon = 0.99$ often performs well. Decaying epsilon values are further simple modification which allows an agent to exploits more of its knowledge over time as it gains knowledge and it has explored more of the state space.

Upper Confidence Bound

One of the problems with ϵ-greedy strategies was the indiscriminate choice of actions when exploring. One approach is to maintain some prior distribution encoding what we know about the reward function for each state-action pair. This Bayesian strategy allows us to explore possibilities which have high uncertainty in our knowledge - ones which may have high reward. UCB says we should choose the action in which the associated expected reward plus one standard deviation (from our prior of that option) is the highest. Formally, $\mu + \beta\sigma$ is the *upper confidence bound* where β is a trade-off parameter which shrinks over time.

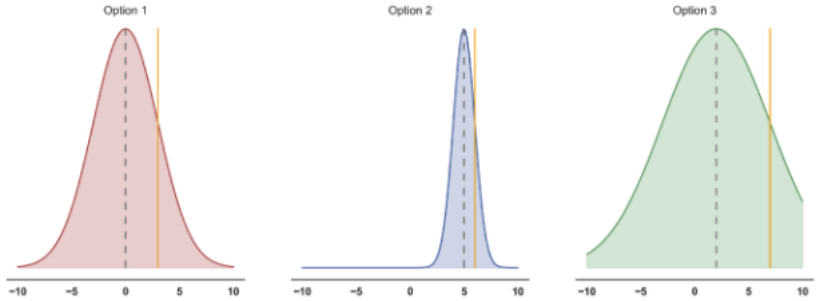


Figure 4: Three prior distributions encoding our knowledge about the reward for each option available to the agent. The option with the highest $\mu + \beta\sigma$ (in yellow) is selected for exploration.

Thompson Sampling

The last strategy to be discussed is that of Thompson sampling - a method which is theoretically similar to UCB. In Thompson sampling we 'draw' from each of our prior distributions and order the the resulting rewards. The option which yields the highest reward is selected for exploration. This has the effect that options with high mean rewards μ are selected more often, but uncertain events can also be selected since there is large variance in the prior distribution. This method, in contrast to UCB, never *gives up* on any option. Rather, as we become more certain about the reward priors, choosing options with smaller expected rewards becomes less likely.

3.4 Trade-offs

Model-free or model-based? Supervised or unsupervised? Gradient or policy methods? Off vs on-policy approaches? The options for different algorithmic structures in RL seem endless. So why are there so many different choices? The answer lies in the trade-offs we make when choosing which approach to take. Let's look at sample efficiency, for example. We would like to learn as much as possible from every bit of data we have gathered. So why not choose the most sample efficient method every time? In the current state of RL, many of the less sample efficient algorithms turn out to be more performant in the long run, and so having more compute plus a less efficient algorithm turns out to be a better solution than having a more complex or more sample efficient algorithm. The goal, of course, is to develop algorithms that are highly sample efficient and performant. State-of-the-art model-based methods are models which aim to do just this, and this is what is develop in this paper.

Another common way to distinguish between reinforcement learning algorithms is by observing whether or not they are *on-policy* or *off-policy*. Every reinforcement learning agent makes decisions within the framework of some MDP, and applies some policy to make these decisions. To ensure the agent explores the state-space the agent must at times make sub-optimal decisions. The policy the agent follows is therefore not necessarily optimal. The critical difference between an off and on-policy agent is that an off-policy learns a *target policy* while enacting some *behaviour policy* during learning. The agent thus learns *off* the target policy. An on-policy agent, on the other hand, learns the policy it is enacting. By their nature, on-policy approaches are generally easier to implement than their off-policy counterparts. Off-policy approaches, however, allow more flexibility. For example, an off-policy approach could even learn from a human-expert. In this paper we explore algorithms which use different approaches and we shall discuss the trade-offs.

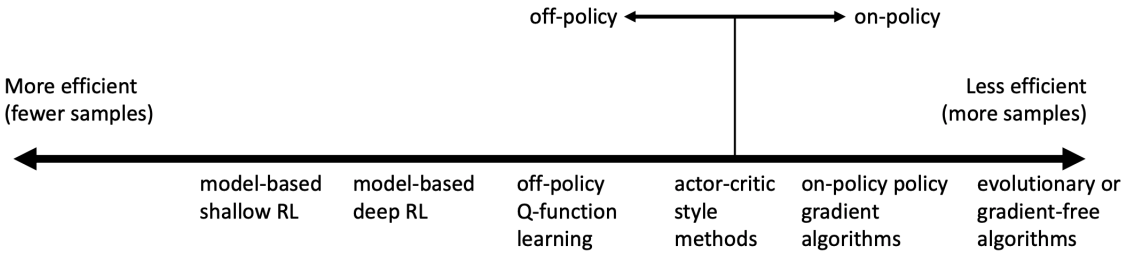


Figure 5: Diagram showing different types of algorithms and their sample efficiencies.

3.5 Benchmarking Algorithms

Environments

In practical applications of RL the environment the agent is placed in determines how well the agent performs. Testing algorithms on a wide variety of environments becomes very important. Key factors of an environment include, but are not limited to:

1. How sparse the rewards are. Very sparse rewards could require immense exploration.
2. Whether or not states are fully observable. It may be hard for an agent to estimate the value of the state it is in with limited information.
3. The stochastic nature of the environment - are rewards highly variable? This makes learning the value of states more difficult.
4. Multi-objective, unspecified or risk sensitive reward functions. Agents struggle to navigate reward functions as they may 'learn' states that lead to high reward as being undesirable.
5. Real time inference or decision making. High pace environments limit the computation agents can perform before making a decision.
6. High dimensionality puts pressure on the agents as it is forced to approximate or sample from the environment to make decisions.
7. Delay in rewards means the agent may not know what decision to attribute success to. This makes repeating good decisions harder.

These problems mean that different agents can perform well in different types of environments. Benchmarking these agents in tasks known to contain different challenges thus becomes important in evaluating the performance of an agent. Further, how to measure the 'intelligence' of an decision making agent is an open question, making quantifying intelligence all the more challenging. Many state-of-the-art model-based algorithms have been recently benchmarked [5]. The authors conclude that "across this very substantial benchmarking, there is no clear consistent best MBRL algorithm, suggesting lots of opportunities for future work bringing together the strengths of different approaches." Some of the common environments in which modern algorithms are tested include

1. The Arcade Learning Environment (ALE) for classic control tasks with (often) partial observability.
2. MuJoCo for continuous control and physics based tasks.
3. VizDoom for partial observability and high-risk training.
4. Board games such as Chess, Go, and other high dimensional, well defined rule-based games.

Finally, the challenges of real-world applications of RL are often not present in popular environments for testing RL. A new environment for testing all of these challenges simultaneously is *Humanoid* [12]. Future testing and comparison should be performed on similar, diverse environments.

Measuring Intelligence and ARC

Expanding on the question of how to measure artificial intelligence, François Chollet argues we need an explicit goal to measure, not just anecdotal performance measures [13]. Classical tests, such as the Turing test, is not a sufficient measure as it outsources the measure of intelligence to humans. Measuring intelligence this way breaks down into a question of "does this agent seem human to you?" This is a biased in the way of humans and leads to the *AI effect* wherein each time a new achievement in AI is reached (e.g. Deep Blue Chess performance)

the benchmark for intelligence is 'moved'. Chollet raises the point that intelligence should measure both skill and adaptivity of an agent, and so testing agents on a wide range of environments is crucial. Chollet raises the point that current machine learning agents are locally intelligent in that they know their 'unknowns'. The next stage of AI is broad intelligence in which agents will be able to solve a broad category of related tasks. Arguably some of the model-based methods presented later in this paper, such as MuZero, satisfy some of these conditions. The ultimate goals of AI research is to create an extreme intelligence agent. This level of intelligence would be the ability to solve open-ended and abstract problems, similar to how humans can solve previously unknown problems. He is also argues that intelligence should be measured in a different manner to performance because task performance can be memorised or 'bought' by throwing data at the problem. Chollet goes on to model a measure of intelligence on psychometric (IQ) testing done on humans. The Abstraction and Reasoning Corpus (ARC) provides a benchmark to measure skill-acquisition of AI agents on unknown tasks. The agents is constrained to learning from a limited number of demonstrations of complex tasks. Solving tasks with limited demonstrations requires extreme sample efficiency - something model-based approaches are relatively good at.

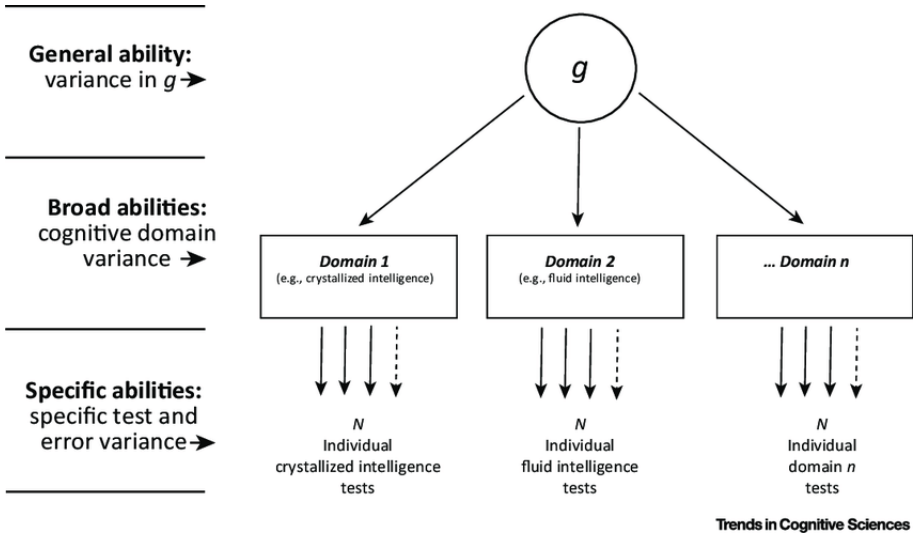
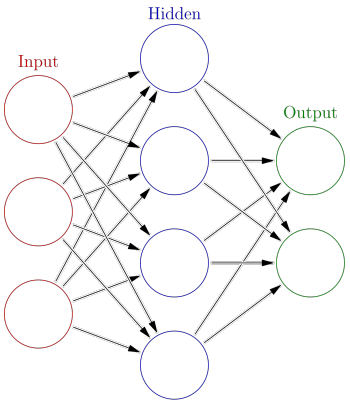


Figure 6: Diagram showing the hierarchical structure of general intelligence in humans. Obtained from [14].

3.6 Deep Learning

This paper focuses on modern and highly performant approaches to reinforcement learning. As with all of machine learning, the trend has been towards applying deep learning approaches for function approximation to boost performance. This section will briefly discuss what neural networks are and some of the important results which underpin the impressive performance these methods provide. The focus of this paper, however, is not the mathematics of deep learning and so the details will not be dwelled on.

An artificial neural network (ANN) is a computing system inspired by biological neural networks in an animal brain. An ANN is a collection of connected nodes called *neurons* joined by *synapses* having associated weights. This structure forms a weighted directed acyclic graph, and leads to some remarkable results which are vital in deep learning.



Theorem 3.1 (Universal Approximation Theorem) *A feed-forward network with a single hidden-layer*

containing a finite number of neurons can approximate arbitrary well real-valued continuous functions on compact subsets of R^n.

This theorem tells us that the construction of a simple neural network to represent a wide range of possible functions is possible when given the appropriate parameters and weights. A neural network can thus be used to approximate any function given some mild assumptions. This is especially useful for approximating functions in reinforcement learning and machine learning in general.

The structure also allows for a the backpropagation algorithm. This algorithms provides a method for evaluating the expression for the derivative of the cost function as a product of derivatives between layers of a neural network from left to right by. The algorithm can be formulated and efficiently computed using matrix multiplication. This is vital as, combined with the universal approximation theorem, it provides a platform for evaluating rates-of-change of arbitrarily complex functions!

Though inspired by the biological brain, and originally designed to imitate it, over time the focus of research in neural networks has been in designing networks for specific tasks. For example, state-of-the-art models in object detection and image classification applies convolutional neural networks (CNNs) in which data is sent through a series of *convolutions* which modifies the data in such a way so as to highlight features of the image. In fact, CNNs are especially important in visual reinforcement learning tasks. Further important advances include recurrent neural networks (RNNs) which can deal with sequential data such as speech or text. Examples include long short-term memory (LSTM) models, which tackle the problem of vanishing gradients in networks. All these models are important in the context of RL because input data to an RL model often has to be processed by these methods to obtain state-of-the-art performance. This will be apparent when we look at modern agents such as MuZero and Dreamer.

4 Model-Free Methods

Model-free reinforcement learning algorithms are a class of algorithms which do not use the transition probability information to train and make decisions. In a sense, they are a class of trial-and-error learning algorithms which purely use experience of reward to choose and learn an optimal policy. There are three major classes of such algorithms in modern literature and can be classified as:

1. Policy-based methods,
2. Aactor-critic methods, and
3. value based methods.

State-of-the-art methods often use approximate methods. As is common in modern machine learning, neural networks are often used as the function approximators. With this in mind, we now develop the model-free approaches to deep reinforcement learning - model free methods with deep neural networks as the function approximators.

4.1 Policy-based Methods

Policy-based methods seek to optimise and learn a policy directly without the need for consultation of a value function (discussed in section 4.3). The policy parameter is usually learned by maximising some performance measure, $J(\theta)$ - the objective function. Naturally, we consider maximising this function directly by moving in the direction of steepest ascent.

Theorem 4.1 (Policy Gradient) *For any Markov decision process (MDP) we have that, in both the average-reward and start-state formulations,*

$$\nabla_\theta J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) = \mathrm{E}\left[\sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta})\right],$$

where the constant of proportionality is 1 for continuous case and the average length of an episode for the episodic case. Here $J(\theta)$ is the performance measure parameterised by θ, and $\mu(s)$ is the on-policy state distribution - the time spent in each state under policy π [10].

4.1.1 Direct Policy Differentiation

As is common in machine learning, we now consider trying to maximise the objective function by directly computing the gradient.

$$\begin{aligned}\nabla J(\theta) &= \int \nabla_\theta \pi_\theta(\tau) r(\tau) d\tau \\ &= \int \pi_\theta(\tau) \frac{\nabla_\theta \pi_\theta(\tau)}{\pi_\theta(\tau)} r(\tau) d\tau \\ &= \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) r(\tau) d\tau \\ &= E_{\tau \sim \pi_\theta(\tau)} \left[\nabla_\theta \log \pi_\theta(\tau) r(\tau) \right]\end{aligned}$$

Let us now consider the finite-horizon case.

$$\underbrace{\pi_\theta(s_1, a1, \ldots, s_T, a_T)}_{\pi_\theta(\tau)} = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

$$\log \pi_\theta(\tau) = \log p(s_1) + \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) + p(s_{t+1}|s_t, a_t)$$

$$\nabla_\theta \log \pi_\theta(\tau) = \underbrace{\nabla_\theta \log p(s_1)}_{0} + \nabla_\theta \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) + \underbrace{\nabla_\theta p(s_{t+1}|s_t, a_t)}_{0}$$

Thus we get the useful result,

$$\begin{aligned}\nabla J(\theta) &= \mathrm{E}_{\tau \sim \pi_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right) \right] \\ &\simeq \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)) (\sum_{t=1}^{T} r(s_t, a_t) \right)\end{aligned}$$

That is, we can sample trajectories to get an estimate for the gradient of the objective function. This gives us a tractable method of updating our parameter of the policy, θ.

$$\theta \longleftarrow \theta + \alpha \nabla J(\theta),$$

where α is some learning parameter. Notice how we did not use the Markovian assumption in this derivation of the update rule. This means that the gradient applies even in the context of a partially observed MDP.

4.1.2 REINFORCE

REINFORCE (Monte-Carlo Policy-Gradient Control) can be seen as the most basic form of a family of policy gradient methods. These policy-based methods aim to exploit the properties of the policy gradient theorem to efficiently improve the policy directly.

Effectively, REINFORCE is the obvious algorithm which arises from direct policy differentiation. First, we sample some trajectories $\{\tau^i\}$ by running our policy π_θ. We compute the gradient of the objective function using these samples, and then update our parameter θ. The algorithm is given in pseudocode below 4.1.2.

Algorithm 1 REINFORCE

Input: differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \epsilon \mathbb{R}^{d'}$
for `each episode` **do**
  Generate an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_{T-1}$ following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  for each step of episode t = 0,1,...,T-1 **do**
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \log \pi(A_t|S_t, \theta)$
  end for
end for

4.1.3 Reducing Variance

One problem with the policy methods presented so far is that they are very sensitive to difference in sampled trajectories. This means the gradients we calculate from the approximation suffer from large variance - that is, they are very noisy.

Applying the concept of causality, we note that the action we take at time-step t does not change the reward we observe at $t' < t$. Therefore, we can reduce the magnitude of the summation by only considering future rewards, called the *reward to go*, $\hat{Q}_{i,t}$.

$$\nabla J(\theta) \simeq \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \underbrace{\left(\sum_{t'=t}^{T} r(s_t, a_t) \right)}_{\hat{Q}_{i,t})}.$$

Another approach to reducing variance is by considering baselines. The intuition here is that we would like to make selection of actions that result in above average reward, more likely. We would also like to decrease the probability of choosing actions that lead to below average reward. Essentially, we want to focus on the *relative success* of following some trajectory, τ.

$$\nabla J(\theta) \simeq \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) - b \right),$$

$$b = \frac{1}{N} \sum_{i=1}^{N} r(\tau).$$

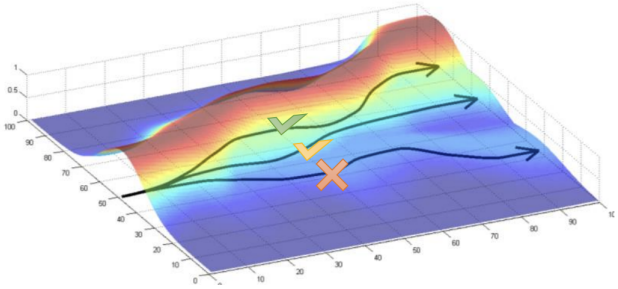


Figure 7: This figure providing intuition for why using baselines is a good idea. The height of the surface gives the amount of reward as a function of trajectory. We want to make the trajectories along the peaks of the surface more likely [15].

Subtracting a baseline is unbiased in expectation, $\mathrm{E}\left[\nabla_\theta \log \pi_\theta(\tau) b\right] = 0$, but not unbiased in variance. This is the exact behaviour we want. The optimal baseline can be derived by considering the baseline that minimises this variance. It turns out the optimal baseline is the expected reward weighted by the magnitude of the gradients. Often in practice, we just use the expected return - the value function.

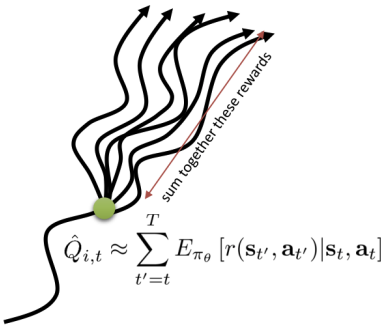


Figure 8: Figure displaying how the reward to go can be approximated by the Q-value [15].

4.2 Actor-Critic Methods

One way we can improve upon policy-based methods is by introducing a *critic* to indicate how well the agent is doing at any point along some trajectory. The critic can be thought of as a state dependent baseline. Note, the fact that the baseline is now state depenedent has no affect on the expectation - it remains unbiased.

$$\nabla J(\theta) \simeq \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t))(\underbrace{Q(s_{i,t},a_{i,t}) - V(s_{i,t})}_{A(s_{i,t},a_{i,t})}\right)$$

Given this approach, we now have three different value functions we can approximate. The theory of actor-critics is focused on how we can best approximate these value functions. The choice of what to approximate depends on what behaviour is deemed desirable.

1. $\mathbf{A}^\pi(\mathbf{s_t}, \mathbf{a_t})$: The advantage, by definition, will always be a much smaller quantity that Q^π or V^π. It will thus lead to a smaller variance in theory. It is, however, strongly dependent on policy and so might be highly variable with respect to choice of π.

2. $\mathbf{Q}^\pi(\mathbf{s_t}, \mathbf{a_t})$: Dependent on state and action. Not as sensitive to π and A^π.

3. $\mathbf{V}^\pi(\mathbf{s_t})$: Only dependent on state.

Let us consider the problem of trying to estimate the advantage function, $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$. How can we estimate $Q^\pi(s_t, a_t)$ given that it is dependent on states and actions? Let us start with the single sample approximation for the expectation as follows.

$$\begin{aligned} Q^\pi(s_t, a_t) &= r(s_t, a_t) + \mathrm{E}_{s_{t+1}\sim p(s_{t+1}|s_t,a_t)}\left[V^\pi(s_{t+1})\right] \\ &\simeq r(s_t, a_t) + V^\pi(s_{t+1}) \end{aligned}$$

This leads nicely to an approximation is terms of the difference in state-value between two temporally adjacent states.

$$A^\pi(s_t, a_t) \simeq r(s_t, a_t) + V^\pi(s_{t+1}) - V^\pi(s_t)$$

Then, to get an estimate for V^π, we can use Monte-Carlo estimation with neural network function approximation. We apply supervised learning to training data $\{s_{i,t}, y_{i,t}\}$ where $y_{i,t}$ is the target. This target can be calculated approximated in two main ways:

1. **Monte-Carlo Estimate**: This is a single sample estimate approach, which may lead to high variance but is unbiased:
$$y_{i,t} = \sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'}).$$

2. **Bootstrapped Estimate**: This is an approximation for the ideal target, which is biased with low variance:
$$y_{i,t} = \sum_{t'=t}^{T} \mathrm{E}_{\pi_\theta}[r(s_{i,t'}, a_{i,t'})] \simeq r(s_{i,t}, a_{i,t}) + V^\pi(s_{i,t+1}).$$

We apply supervised learning with the following loss given by

$$L(\phi) = \frac{1}{2}\sum_i ||\hat{V}^\pi_\phi(s_i) - y_i||^2$$

From here we can start to develop algorithms for evaluating the policy.

Algorithm 2 Batch Actor-Critic

for k steps **do**
  Sample $\{s_i, a_i\}$ from $\pi_\theta(a|s)$.
  Fit $\hat{V}^\pi_\phi(s)$ to sampled reward sums.
  Evaluate $\hat{A}^\pi(s_i, a_i) = r(s_i, a_i) + \gamma\hat{V}^\pi_\phi(s'_i) - \hat{V}^\pi_\phi(s_i)$.
  $\nabla_\theta J(\theta) \simeq \sum_{i\,\theta} \log \pi_\theta(a_i|s_i)\hat{A}^\pi(s_i, a_i)$.
  $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$.
end for

Extending the idea of batch actor-critic, we can implement an online actor-critic algorithm or even a one-step actor-critic as implemented below.

Algorithm 3 One-step Actor-Critic

for `each episode` **do**
  Initialize S (first state of episode)
  $I \leftarrow 1$
  while S is not terminal state **do**
    $A \sim \pi(\cdot|S, \boldsymbol{\theta})$
    Take action A, observe state S' and reward R
    $\delta \leftarrow R + \gamma \hat{v}(S', \boldsymbol{w}) - \hat{v}(S, \boldsymbol{w})$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}} \delta \nabla \hat{v}(S, \boldsymbol{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \boldsymbol{I} \delta \nabla \log \pi(A|S, \boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$
  end while
end for

Further extending these algorithms is possible. State-of-the-art model-free actor-critic methods include Advantage Actor Critic (A2C) and Asynchronous Advantage Actor Critic (A3C) [16], which apply and extend these principles of approximating the advantage and minimising the loss function directly to optimise the parameters.

4.2.1 Proximal Policy Methods

In 2017 a new family of policy methods was introduced in [**?**] (PPO) [17] which performed better than previous state-of-the-art algorithms while being easier to implement and tune. These methods alternate between sampling data from the environment by interacting with it, and optimizing a some other objective function (the 'surrogate') using stochastic gradient ascenent. The authors found that PPO strikes a 'favourable balance between sample complexity, simplicity, and wall-time'. The pseudocode is included for completeness.

Algorithm 4 Proximal Policy Optimisation with Clipped Objective

Input: initial policy parameters θ_0, clipping threshold ϵ
for k = 0,1,2,... **do**
  Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
  Estimate advantages $\hat{A}_t^{\pi_k}$ using advantage estimation algorithm
  Compute policy update

$$\theta_{k+1} = \underset{\theta}{\operatorname{argmax}} \mathcal{L}_{\theta_k}^{CLIP}(\theta)$$

  by taking K steps of minibatch SGD (via Adam), where

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = E_{\tau \sim \pi_k}\left[\sum_{t=0}^{T}\left[min(r_t(\theta)\hat{A}_t^{\pi_k}, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t^{\pi_k})\right]\right]$$

end for

4.3 Value-Based Methods

The question now arises, can we omit the policy gradient? Consider the advantage function, $A^{\pi}(s_t, a_t)$, which tells us how much better some action a_t is than the average performance of any action when following some policy π. If we then represent this policy as simply finding the action that maximises the advantage at any stage we are guaranteed to improve (at least in the short term). This policy can be written as

$$\pi'(a_t|s_t) = \begin{cases} 1 & \text{if } a_t = \arg\max_{a_t} A^{\pi}(s_t, a_t) \\ 0 & \text{otherwise} \end{cases}$$

The guarentee of improvement stems from the policy improvement theorem, as presented by Sutton and Barto [10].

Theorem 4.2 (Policy Improvement Theorem) *Let π and π' be any pair of deterministic policies such that $Q^{\pi}(s, \pi'(s)) \geq V^{\pi}(s) \quad \forall s \in S$. Then the policy π' must be at least as good as π.*

Proof 4.1

$$
\begin{aligned}
v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\
&= \mathrm{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = \pi'(s)\right] \\
&= \mathrm{E}_{\pi'}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s\right] \\
&\leq \mathrm{E}_{\pi'}\left[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s\right] \\
&= \mathrm{E}_{\pi'}\left[R_{t+1} + \gamma\, \mathrm{E}_{\pi'}\left[R_{t+2} + \gamma v_\pi(S_{t+2})\right] | S_t = s\right] \\
&\vdots \\
&\leq \mathrm{E}_{\pi'}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s\right] \\
&= v_{\pi'}(s)
\end{aligned}
$$

With this in mind, we can develop the classical methods of reinforcement learning which follow from optimal control - and often form the introduction to RL. These are the value based methods.

4.3.1 Dynamic Programming

Consider now the situation where we have a discrete, finite state space with discrete, finite actions and we know everything about the environement and its dynamics. That is, we have a fully observed MDP. Representing this state-action space as a table, we can store the full, exact value function (e.g. GRIDWORLD). We can notice that the policy given by the $\arg\max_{a_t} A^\pi$ is deterministic in the sense that given any state, the action the agent takes is fully specified as,

$$
\pi'(a_t|s_t) = \begin{cases} 1 & \text{if } a_t = \arg\max_{a_t} A^\pi(s_t, a_t) \\ 0 & \text{otherwise} \end{cases} \quad \rightarrow \pi(s) = a.
$$

Our bootstrapped update method for the value function is then,

$$
V^\pi(s) \leftarrow r(s, \pi(s)) + \gamma\, \mathrm{E}_{s' \sim p(s'|s,\pi(s))}[V^\pi(s')].
$$

Simply iterating and applying the policy improvement theorem, we develop a method for finding the optimal policy (and value function).

1. Evaluate the (preferred) value function
2. Set $\pi \longleftarrow \pi'$

Noticing that we can directly improve the value function without finding a policy, we arrive at the **value iteration** algorithm.

Algorithm 5 Policy Iteration

Initialise $V(s)$ and $Q(s, a)$ to arbitrary values.
while V(s) has not converged **do**
  for every state s **do**
    for every action a **do**
      $Q(s, a) \leftarrow r(s, a) + \gamma\, \mathrm{E}[V(s')]$
    end for
    $V(s) \leftarrow \max_a Q(s, a)$
  end for
end while

We can also evaluate a policy continually to calculate its value functions. We can do this randomly to obtain an estimate for this value. As the iteration, our value function will converge and we can apply the value function to determine the optimal policy. Finally we arrive at the classic policy iteration procedure. Here policy iteration and policy improvement is performed alternatively, iteratively, until convergence upon a (near) optimal policy.

4.3.2 Q-Learning

These ideas of iteratively updating and improving the policy leads directly to a family of fitted Q-value iteration. The most popular of which is Q-learning. Interestingly, we can notice that Q-iteration is actually *off-policy* since it approximates the value of the new policy π' at the next state s'_i. This algorithm is essentially learning some pool of values for transitions after taking some action in the state-action space.

Algorithm 6 Q-Learning

Collect dataset $\{s_i, a_i, s'_i, r_i\}$ using some policy π.
for K steps **do**
  Set $y_i \leftarrow r(s_i, a_i) + \gamma \max_{a'_i} Q_\phi(s'_i, a'_i)$.
  Set $\phi \leftarrow \arg\min_\phi \frac{1}{2} \sum_i ||Q_\phi(s_i, a_i) - y_i||^2$.
end for

Notice when discussing dynamic programming, it was specified that the state-action space was discrete. The *curse of dimensionality* applies here in the sense that most interesting and real scenarios in which we want to apply value learning has state-action spaces that are intractably large for any computational device to represent the value function in a *table*. Once again, we can apply deep learning methods to approximate the value function. Applied to value iteration methods, we arrive at the *fitted value iteration* algorithm. One example is deep Q-learning (DQN) [18] which uses a Huber loss function, where the loss is quadratic for small values and linear for large values. This allows for less dramatic changes in the approximation of the Q-value while learning, which can often negatively impact learning performance. The full pseudocode is presented for completeness.

Algorithm 7 Deep Q-Learning

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode=1,M **do**
  Initialize sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for t=1,T **do**
    Select random action a_t with probability ϵ
    else select $a_t = \text{argmax}_a \, Q(\phi(s_t), a; \theta)$
    Observe r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random subsample of transitions from D
    Set

$$y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$$

    Perform gradient descent step on $(y_i - Q(\phi_j, a_j; \theta)^2$ w.r.t. network parameters θ
    Reset $\hat{Q} = Q$ every C steps
  end for
end for

5 Model-Based Methods

Up to this point we have discussed methods primarily relying on the *learning* of value functions, usually approximating these with some neural network. That is, our focus has been on trying to learn how much each state is 'worth' according to the expected return of being in a specific state. Model-based methods add to this by adding a *model* of the state dynamics. The dynamics of the system can be:

1. **Deterministic:** Here the agent knows how the world works. Thus, given some state, the agent can plan with assured knowledge of how the environment will react to a sequence of actions. Formally, we have

$$a_1, \ldots, a_T = \arg\max_{a_1, \ldots, a_T} \sum_{t=1}^{T} r(s_t, a_t) \text{ s.t. } a_{t+1} = f(s_t, a_t).$$

2. **Stochastic Open-Loop:** In this scenario the environment dynamics are stochastic, but an agent must still execute a plan from start to finish whether or not the optimal scenario it has planned for has played out. That is, it aims to achieve maximum return based on the dynamics of the system conditioned on the entire plan. The task is then to find actions such that

$$a_1, \ldots, a_T = \arg\max_{a_1, \ldots, a_T} \text{E} \left[\sum_{t=1}^{T} r(s_t, a_t) \right].$$

3. **Stochastic Closed-Loop:** In this scenario the agent has the advantage of being able to change its plan based on the dynamics it observes. The task is then to find a policy such that

$$\pi = \arg\max_{\pi} \mathrm{E}_{\tau \sim p(\tau)} \left[\sum_{t=1}^{T} r(s_t, a_t) \right].$$

The aim is to learn how the environment 'reacts' to specific actions, giving us the ability to simulate experience and predict the best actions to take from a particular state. The focus here is on *state-space planning* in which value functions are computed over all possible states. Notice, there are two distinct ideas here:

1. Learning a model of the environment (*learning*); and
2. using a model to improve some value function or policy (*planning*).

Real experience can thus be used in two ways for planning. It can improve the model of the environment and it can be used to directly improve upon a value function or policy. This relationship is encapsulated in figure 9 below.

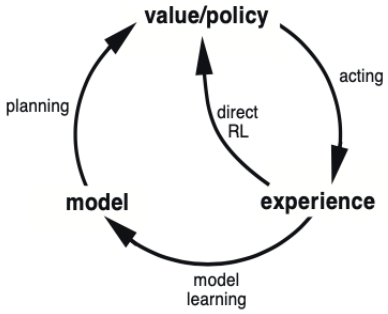


Figure 9: Diagram showing how each component of the RL process fits into a complete algorithm. See Chapter 8 [10].

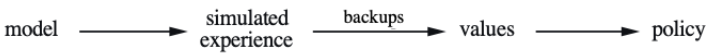
There are clearly advantages to both methods. Model-based (indirect) methods can make fuller use of limited data collected (better sample efficiency), allowing for fast learning of more optimal strategies. The models, however, can be limited by design and biases. This makes model-free (direct) methods preferable in some scenarios. Later we shall see that advancements in model-based methods are quickly closing the gap wherever there seems to be an advantage in using model-free methods (e.g. Atari).

Figure 9 gives rise to a natural algorithm structure in which is transition between acting, learning a model and planning happens continually. We shall first discuss *planning* in the context of reinforcement learning - a field which borrows heavily from optimal control theory.

5.1 Planning without Derivatives

In the context of model-based RL *planning* refers to the process of taking a model as an input and producing an improved policy as output. Since the model is *known* and all the state dynamics are encapsulated in the model, the task of improving the policy becomes a task of finding actions that lead an agent along the optimal (in terms of reward) path in the state-space. This is the *state-space planning* as presented in Sutton & Barto [10]. *Plan-space planning* is another approach in which planning is instead seen as a search through the space of all plans. Since RL deals with sequential decision making, plan-space planning is not efficient in this context and is not considered further.

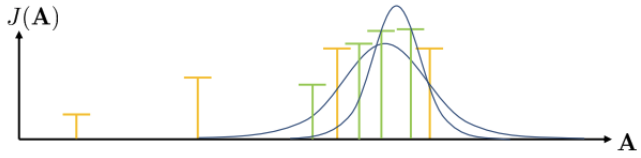
The methods now explored for state-space planning all share a common structure. This involves computing some value function by simulating experience to improve the policy. There is clear similarity between dynamic programming methods often used in the context of tabular state-action spaces and this general planning structure. This motivates some of the methods we discuss later, such as Dyna-Q.



One simple method for performing derivative-free optimisation of a sequence of actions, $A = a_1, \ldots, a_T$, is to select a random set of actions and simply pick the 'best' one. There are many ways of improving upon this idea, and we now discuss some of these.

Cross-Entropy Method

One such method for derivative-free optimisation of a sequence of actions, $A = a_1, \ldots, a_T$, is the Cross-Entropy Method (CEM). This method of optimisation is not the best optimiser, but it is extremely easy to implement and is highly parallelisable. It tends to work well on problems that don't have large dimeniionsality (e.g. $d \geq 100$) and with a low number of time-steps. The intuition behind this method is to randomly sample from some distribution (commonly Gaussian) of trajectories, select the best trajectories, and then bias this distribution towards the highly performant trajectories. Repeating this process leads to a situation where we are selecting from a distribution of high value trajectories. A simple visualisation of this shifting distribution is given below.



Notice that since we optimising entire trajectories, this method is an open-loop planner. It does not make use of the ability to account for replanning after selecting an action. The model is used in the evaluation of the value of the trajectory.

Algorithm 8 Cross-Entropy Optimisation

Pick trajectories $A_1, \ldots, A_N$ from some distribution, $p(A)$.
Evaluate value of each sequence, $J(A_1), \ldots, J(A_N)$.
Select *elites* $A_{i_1}, \ldots, A_{i_M}$ with highest value, $M < N$.
Refit $p(A)$ using only these elites.

This method can be generalised and improved. For example, one such generalisation (CMA-ES [19]) applies the idea of shifting the distribution towards the direction of the 'momentum' of the distribution. That is, we shift the distribution towards the direction the overall trend of improvement seems to be going.

Rollout & MCTS

Rollout algorithms are a set of decision time planning algorithms which apply Monte Carlo control to simulated trajectory sampling. They aim to accurately estimate the action-value of a particular state by averaging the returns over trajectories observed by taking different actions. Once an action-value is deemed accurate enough, the action with the highest estimated value is selected and executed. Rollout algorithms do not aim to estimate an optimal policy. Rather, they attempt to select the best action from the current state given some rollout policy. The policy improvement theorem assures us that selecting the action with the highest estimated value should result in an improvement of the rollout policy. Of course, whether this works depends on how accurate your estimates are and how far into the future one can roll out action selection.

Monte Carlo Tree Search (MCTS) is a decision time planning algorithm which combines a rollout algorithm with the ability to accumulate value estimates. This allows for directing of action selection and simulation towards trajectories with higher estimated reward. MCTS has been astoundingly successful in improving performance of many state-of-the-art algorithms to date, including many of the model based algorithms presented in later sections.

Each iteration of the MCTS algorithm consists of four distinct steps. These can be categorised as follows:

1. **Selection:** Starting at the root node, a general tree policy is used to select the appropriate leaf to expand.
2. **Expansion:** At implementation specific points, the selected leaf node is expnded by selecting unexplored actions.
3. **Simulation:** A rollout policy is applied to the selected leaf node select the estimated best action.
4. **Backup:** The newest estimate for the action values are backed up the tree. Action-values for states beyond the tree policy are not maintained.

These four steps are repeated as many times as constraints of the computational system allow. Once action selection is required, some policy is used to select the appropriate action. For example, this may be the most visited or the highest value state. Once an action is selected the sub-tree with the current state as the root node can be reused from the previous MCTS procedure.

Algorithm 9 Generic MCTS Algorithm

for K steps **do**
  Find a leaf s_l using TreePolicy(s_1)
  Evaluate the leaf using RolloutPolicy(s_l)
  Update all values in tree between s_1 and s_l.
end for
Select best action from s_1

The selection of an appropriate tree policy for leaf selection can have a large impact on performance. A popular choice of a score function for selecting a child in the tree is given by

$$\text{Score}(s_t) = \frac{Q(s_t)}{N(s_t)} + 2C\sqrt{\frac{2\ln N(s_{t-1})}{N(s_t)}},$$

where $N(s_t)$ is the number of times node s_t has been visited. This scoring is based on the UCB method, as presented in section 3.3.

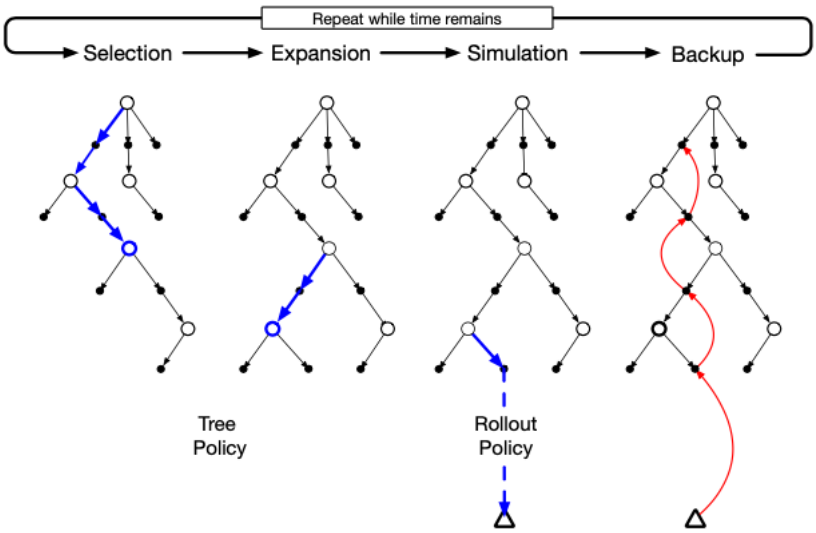


Figure 10: Diagram showing the 4 steps of Monte Carlo Tree Search as explained in Sutton & Barto [10].

5.2 Planning with Derivatives

In the previous section we discussed methods for finding good policies by planning without using derivatives. In this section classic optimal control algorithms for planning using derivatives are discussed. In many scenarios, such as in Atari games, we don't have access to derivative, so planning without them is very useful. In some cases, however, we do have access to derivative information. Take, for example, the case of robotics in which the physics of movement is part of a robot's design. Notice, we can write the goal of a RL agent as a constrained equation,

$$\min_{a_1,\dots,a_T} \sum_{t=1}^{T} c(s_t, a_t) \quad \text{s.t.} \quad x_t = f(s_{t-1}, a_{t-1}),$$

where $c(s_t, a_t)$ is the cost function. This corresponds to the unconstrained problem,

$$\min_{a_1,\dots,a_T} \sum_{t=1}^{T} c(s_1, a_1) + c(f(s_1, a_1), a_2) + \cdots + c(f(f(\dots), \dots), a_T).$$

We could optimise this equation bu finding the derivatives, $\frac{df}{ds_t}$, $\frac{df}{da_t}$, $\frac{dc}{ds_t}$, and $\frac{dc}{da_t}$. In practices, though, it turns out to be better to apply a 2nd order method. This holds because a first order 'shooting method' relies on observing changes in the state due to small changes in an action. The problem is that a small change in action at the beginning of a trajectory may lead to large changes in where the agent lands up. Methods for optimising over states (collocation methods) exist but shall not be discussed further. We now explore the theory for some 2nd order methods.

Linear Quadratic Regulator

Linear Quadratic Regulator (LQR) solves this problem subject to a special, linear form of the function,

$$f(s_t, a_t) = F_t \begin{bmatrix} s_t \\ a_t \end{bmatrix} + f_t,$$

and a quadratic cost,

$$c(s_t, a_t) = \frac{1}{2} \begin{bmatrix} s_t \\ a_t \end{bmatrix}^T C_t \begin{bmatrix} s_t \\ a_t \end{bmatrix} + \begin{bmatrix} s_t \\ a_t \end{bmatrix}^T c_t.$$

We can start by solving for a_T as the base case,

$$Q(s_T, a_T) = \frac{1}{2} \begin{bmatrix} s_T \\ a_T \end{bmatrix}^T C_T \begin{bmatrix} s_T \\ a_T \end{bmatrix} + \begin{bmatrix} s_T \\ a_T \end{bmatrix}^T c_T + \text{constant}.$$

Optimising for the action, we find and solve for

$$\nabla_{a_T} Q(s_t, a_t) = 0.$$

After solving for the linear equations, we find the form of a_T and can substitute this back into $Q(s_T, a_T)$ which gives a solution fully specified in terms of s_T. We can repeat a similar elimination procedure for u_{T-1}. After lots of linear algebra, we arrive at the set of LQR recursion algorithms to find the optimal actions

Algorithm 10 LQR Backward Recursion

for $t = T$ until $t = 1$ **do**
  $Q_t = C_t + F_t^T V_{t+1} F_t$.
  $q_t = c_t + F_t^T V_{t+1} f_t + F_t^T v_{t+1}$.
  $Q(s_T, a_T) = \frac{1}{2} \begin{bmatrix} s_T \\ a_T \end{bmatrix}^T Q_t \begin{bmatrix} s_T \\ a_T \end{bmatrix} + \begin{bmatrix} s_T \\ a_T \end{bmatrix}^T q_T + \text{constant}$.
  $a_t \leftarrow \arg\min_{a_t} Q(s_t, a_t) = K_t s_t + k_t$.
  $K_t = -Q_{a_t, a_t}^{-1} Q_{a_t, s_t}$
  $k_t = -Q_{a_t, a_t}^{-1} q_{a_t}$
  $V_t = Q_{s_t, s_t} + Q_{s_t, a_t} K_t + K_t^T Q_{a_t, s_t} + K_t^T Q_{a_t, a_t} K_t$
  $v_t = q_{s_t, a_t} k_t + K_t^T Q_{a_t} + K_t^T Q_{a_t, a_t} k_t$
  $V(s_t) = \frac{1}{2} s_t^T V_t s_t + s_t^T v_t +$ constant
end for

Algorithm 11 LQR Forward Recursion

for $t = 1$ until $t = T$ **do**
  $a_t = K_t s_t + k_t$
  $x_{t+1} = f(s_t, a_t)$
end for

It turns out that if the the dynamics are stochastic, and these are distributed as a Gaussian with constant covariance, the solution is exactly the same! This is remarkable as this is not very computationally expensive. In fact, it is linear. What about the non-linear case? We can simply apply Taylor approximation and use our LQR algorithm on the linear terms in the Taylor expansion. This is known as DDP or iterative LQR. Remarkably, this turns out to be very similar to Newton's method for trajectories! This is discussed in a practical guide for implementation [20].

5.3 Learning without a Policy

Given some model we can apply the planning tools previously discussed to obtain a highly performant policy even in cases of large state spaces. Let us consider what happens when we are not given an explicit model of the environment. Without a model of the environment we have multiple approaches available to us. We could, of course, apply model-free techniques to directly learn a value function or policy. Once again though the problem of sample efficiency arises. What if we instead we *learn* a model of the environment and then apply the planning tools we previously developed? This approach turns out to be incredibly powerful and is now discussed.
In this scenario we wish to learn a model $f(s_t, a_t)$ for the given MDP. Let's consider the naive and obvious approach we could take:

Algorithm 12 Naive Model-Based Algorithm

Apply base policy $\pi_0(a_t, s_t)$ to collect a dataset $D = \{(s, a, s')_i\}$
Apply supervised learning to find $f(s, a)$ by minimising some loss. e.g. $\sum_i ||f(s_i, a_i) - s'_i||^2$.
Plan for actions using $f(s, a)$.

This algorithm is simple and turns out to function under certain conditions. Classical robotics often use this approach for system identification (e.g. see [21]). Performance, however, turns out to be poor when applying deep learning approaches. The problem comes in with how we collect data according to some policy and train our model on that data. Trying to extrapolate from the collected data can lead to cases where our agent can 'fall off a cliff' - that is, find itself applying knowledge it has gained from data which is not appropriate for the situation it is in. An obvious solution to this would be to try and collect more data such that it is representative of the entire state-space.

Algorithm 13 Slightly Better Model-Based Algorithm

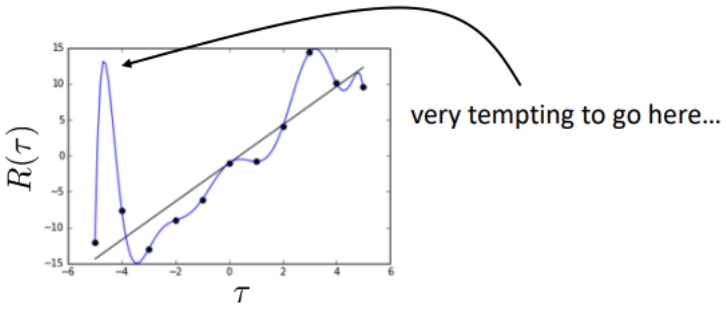
Apply base policy $\pi_0(a_t, s_t)$ to collect a dataset $D = \{(s, a, s')_i\}$
for K steps **do**
  Apply supervised learning to find $f(s, a)$ by minimising some loss. e.g. $\sum_i ||f(s_i, a_i) - s'_i||^2$.
  Plan for action a' using $f(s, a)$.
  Apply action a' and add resulting data $\{(s, a, s')_j\}$ to D.
end for

This approach seems to work with deep learning approaches, but can be sensitive to mistakes an agent makes. Consider the situation where an agent makes a sub-optimal decision and narrowly leaves the path the model has appropriate knowledge for. These mistakes can quickly gather momentum and the agent can find itself once more 'falling off the cliff'. For sufficiently large state-spaces we can never really gather enough data to be representative while still being efficient in the use of this data. How then can we address this issue? Quite simply, we can replan right after we add new data to our dataset. This works surprisingly well! So well, in fact, that replanning can compensate for an imperfect planning algorithm and imperfect agent decisions. This is useful when we want to plan with short horizons.

Algorithm 14 Replanning Model-Based Algorithm/Model Predictive Control (MPC)

Apply base policy $\pi_0(a_t, s_t)$ to collect a dataset $D = \{(s, a, s')_i\}$
for N steps **do**
  Apply supervised learning to find $f(s, a)$ by minimising some loss. e.g. $\sum_i ||f(s_i, a_i) - s'_i||^2$.
  for K steps **do**
    Plan for action a' using $f(s, a)$.
    Apply action a' and add resulting data $\{(s, a, s')_j\}$ to D.
  end for
end for

Though this replanning algorithm works, there is a large performance gap with model-free methods. The intuition for why this happens is that the model is imperfect and doesn't capture what it doesn't know very well. This imperfect model may have false local optima, leading to an agent getting stuck planning for actions it believes will yield high reward.



Uncertainty Estimation

Perhaps we could use uncertainty estimation to detect where the model may be wrong and then correct for these potential errors without having to collect much more data. This uncertainty is useful in many respects. It may

be that a certain action yields high expected reward but there is a possibility (encoded in the uncertainty) that the agent takes too far a step and 'falls off the cliff'. A pessimistic agent may want to avoid this. Or perhaps this uncertainty can guide exploration as in UCB or Thompson sampling for an optimistic agent. If the agent only chooses actions for which it believes it will obtain the highest expected reward, the agent can avoid falsely exploiting the model.

How can we create models that are aware of their own uncertainty? Perhaps we could use the entropy of the outputs of our model. That is, the entropy of the policy distribution (discrete or continuous). A higher entropy would correspond to more uncertainty, right? Wrong. We cannot confuse the two types of uncertainty associated with this statistical process. Consider a model that overfits. This model would have very low entropy while being a bad model. This difference in ideas of uncertainty can be more broadly captured as

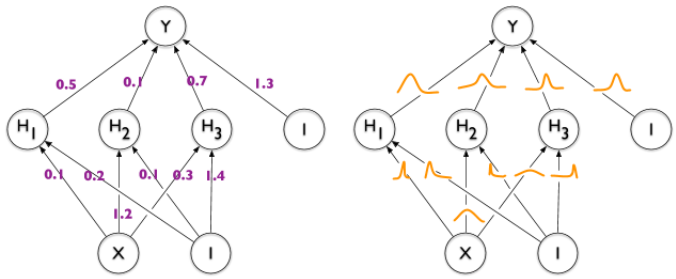
1. Aleatoric (statistical) uncertainty, and
2. Epistemic (model) uncertainty.

This idea is well captured as *"the model is certain about the data, but we are not certain about the model."*

Could we estimate model uncertainty? Theoretically this is easily done. The usual approach to reinforcement learning is to estimate $\arg\max_\theta \log p(\theta|D)$ - the parameters θ that maximises the log-likelihood. What if we instead estimate the entire distribution $p(\theta|D)$? The entropy of this distribution would give us the uncertainty in our model. We could then predict according to

$$\int p(s_{t+1}|s_t, a_t, \theta)p(\theta|D)d\theta.$$

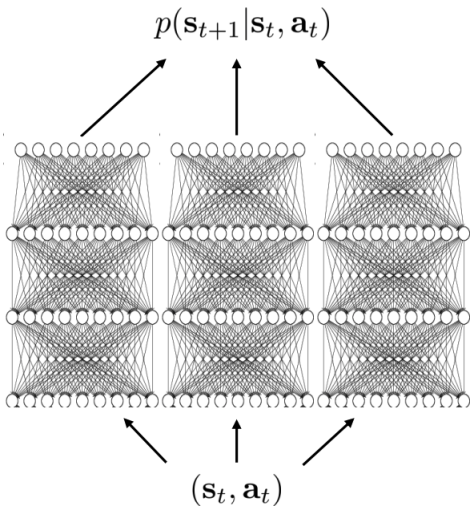
Following this approach leads to the idea of a Bayesian neural network, in which each of the weights between neurons of the network is a distribution. Often these weights are assumed to be independent such that the distribution over the weights is the product of the marginals, $p(\theta|D) = \prod_i p(\theta_i|D)$ where the marginals are normally distributed, $p(\theta_i) = N(\mu_i, \sigma_i)$. This is, of course, a bold and unjustified assumption, but we stick to it for the sake of computational tractability - as is common in machine learning.



Though the Bayesian networks approach is an active area of research, we shall not discuss it further. Instead we focus on a third approach to encapsualting the uncertainty of the model - bootstrap ensembles. The idea here is to train multiple networks on *independent* data, prediciting the same policy. These individual policies can be used to reconstruct the full distribution, $p(\theta|D) \simeq \frac{1}{N}\sum_i \delta(\theta_i)$. That is, we form a mixture model using the different models. Then,

$$\int p(s_{t+1}|s_t, a_t, \theta)p(\theta|D)d\theta \simeq \frac{1}{N}\sum_i p(s_{t+1}|s_t, a_t, \theta_i).$$

We can generate independent-like datasets by sampling with replacement from the dataset of our model, D.



5.4 Learning with a Policy

In section 5.3 we discussed methods of planning with a learned model. These purely model-based methods did not require any policy to learn a good model of the environment. In this section we discuss another approach to model-based reinforcement learning in which we use the learned model to try and recover a policy.

Recall that in the stochastic open-loop case the agent is not aware that it can change its decisions in the future, and so it plans for an optimal set of actions to achieve maximum reward. Clearly this is not optimal. At each time step an agent gains additional information about the state it is in. This changes the expected reward! Taking advantage of this ability to plan for being able to change its mind is thus an important caveat. Notice how our our previous strategies for model-based reinforcement learning was doing just this - planning on the belief that the agent cannot change its mind. This policy the agent learns can be used to make decisions in the environment. This policy can be global or local. Local policies can be combined to form a global (mixed) policy.

How can we use models to train policies? Perhaps we could apply backpropagation to maximise our objective,

$$\nabla J(\theta) = \sum_{t=1}^{T} \frac{dr_t}{ds_t} \prod_{t'=2}^{t} \frac{ds_{t'}}{da_{t'-1}} \frac{da_{t'-1}}{ds_{t'-1}}.$$

This approach, though effective in many scenarios, is susceptible to vanishing or exploding gradients, especially in long time horizon problems due to the chain-rule-like product of Jacobian terms. A great insight reveals itself when we consider that policy gradients do not actually involve explicit gradient terms. The intuition for this is that the policy gradients given by

$$\nabla J(\theta) \simeq \frac{1}{N} \sum_{i=1}^{N} (\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)) \hat{Q}^{\pi}_{i,t}$$

is given in terms of an expectation over time, whereas the backpropagation formula contains Jacobian terms. Unlike in other areas of machine learning, we cannot choose the form of our dynamics model such that it gives 'nice' gradients because the dynamics model needs to accurately represent to dynamics of the environment to get a good policy.

Algorithm 15 Backpropagation Model-Based Algorithm

```
Apply base policy π_0(a_t, s_t) to collect a dataset D = {(s, a, s')_i}
for N steps do
  Apply supervised learning to find f(s, a) by minimising some loss. e.g. Σ_i ||f(s_i, a_i) − s'_i||^2.
  for K steps do
    Apply backpropagation through f(s, a) to optimise policy π_θ(a_t|s_t).
    Run π_θ(a_t|s_t) and add (s, a, s') to D.
  end for
end for
```

What if we use derivative-free (model-free) RL algorithms to generate samples within our model-based approach? This approach leads to a class of algorithms that can be thought of as *model-free acceleration* in

that is performs model-free RL and boosts performance by using a model. As we would hope, the approaches of using backpropagation and policy gradients turn out to be equivalent in the case of stochastic systems when we apply reparamerisation of the backpropagation formula. There are, of course, trade-offs to the different approaches in practice. These are as follows:

1. **Policy gradient:** High variance - can be reduced with samples. That is, more stability with more samples.
2. **Back-propagation (pathwise) gradient:** Poor numerical conditioning for long-horizon problems and non-unity eigenvalues. Small changes at earlier times can cause large deviations in future times.

Dyna

This *model-free acceleration* idea leads to a highly effective class of algorithms initially presented by Richard Sutton [22]. The Dyna algorithm is essentially an online Q-learning algorithm that performs model-free reinforcement learning (as in section 4.3) while using a model to improve sample efficiency. The use of the model is a bit subtle. We use it to compute the expected value within the inner Q-update of the following algorithm. We can now give the pseudocode for the classic Dyna algorithm.

Algorithm 16 (Classic) Dyna

Pick action a by exploration (given state s).
Observe resultant state s' and reward r - (s, a, s', r)
Perform Q-update: $Q(s,a) \leftarrow Q(s,a) + \alpha \, \mathrm{E}_{s',r}[r + \max_{a'} Q(s',a') - Q(s,a)]$
Update models $\hat{p}(s'|s,a)$ and $\hat{r}(s,a)$ using new experience (s,a,s')
for K steps **do**
  Sample (s,a) from buffer
  Perform Q-updates: $Q(s,a) \leftarrow Q(s,a) + \alpha \, \mathrm{E}_{s',r}[r + \max_{a'} Q(s',a') - Q(s,a)]$
end for

The structure of this algorithm can be generalised in the sense that we can swap out the model-free method for updating the value method or policy and we can learn different models. Figure 11 shows this general structure which forms the basis for many modern, highly performant algorithms.

Algorithm 17 "Dyna-style" MBRL Structure

Collect data (s, a, s', r).
Learn model $\hat{p}(s'|s,a)$ and (optional) $\hat{r}(s,a)$.
for K steps **do**
  Sample $s \sim B$ from buffer.
  Choose action a in some manner. e.g. From B, π, or random.
  Simulate $s' \sim \hat{p}(s'|s,a)$ and (optional) $r = \hat{r}(s,a)$
  Train (or store) on artificial transition (s,a,s',r) using model-free methods.
  Optionally take N more model-based steps.
end for

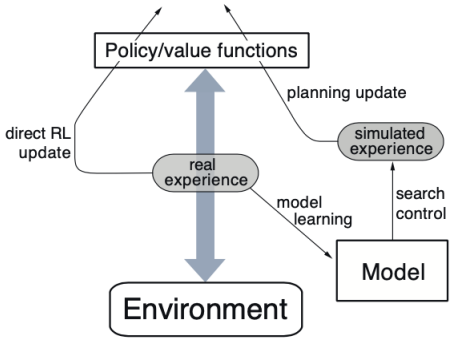


Figure 11: Diagram showing the general architecture for the Dyna algorithm.

There are many modern algorithms which apply this general "Dyna-style" structure. For example, Model-Based Acceleration (MBA [23]), Model-Based Value Expansion (MVE [24]), and Model-Based Policy Optimisation (MBPO [25]). Interestingly, in some cases it may actually be harder to train a model than to train a value

function. In this case model-free methods may be better applied. At this stage researchers don't know how to seperate these cases for sure.

6 Representation & Advanced Predictive Methods

6.1 Latent Representation in MBRL

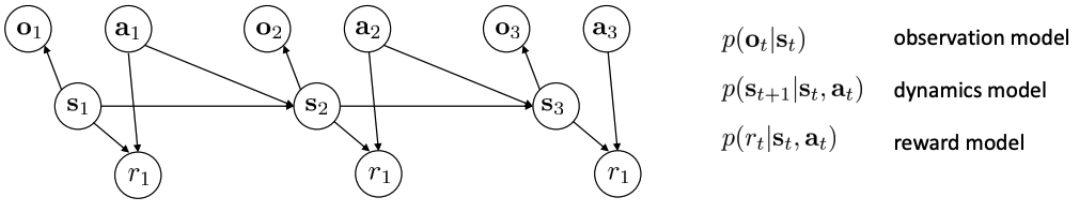
In section 5.3 we discussed methods of model-based reinforcement learning which work in practice. Several problems arise when trying to apply these methods to problems with complex observations. These problems include high dimensionality of observations, redundancy of input data, and partial observability of the state space. Given some POMDP, we can learn a observation model $p(o_t|s_t)$ that is high dimensional but, crucially, is not dynamic. In addition we can learn about the dynamics $p(s_{t+1}|s_t, a_t)$, as before. In addition, since rewards are a function of states and we now have observations, we need a reward model, $p(r_t|s_t, a_t)$. These separate models make the training more difficult than before for several reasons. Previously we had a fully observed model and could apply maximum likelihood theory,

$$\max_{\phi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \log p_\phi(s_{t+1,i}|s_{t,i}, a_{t,i}).$$

The difference with a latent space model is that we don't have s_t or s_{t+1} and so we can only find this result in expectation,

$$\max_{\phi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \mathrm{E}\left[\log p_\phi(s_{t+1,i}|s_{t,i}, a_{t,i})\right],$$

where this expectation is calculated according to $(s_t, s_{t+1}) \sim p(s_t, s_{t+1}|o_{1:T}, a_{1:T})$.



Typically calculated the posterior distribution, $p(s_t, s_{t+1}|o_{1:T}, a_{1:T})$ is intractable and so we instead calculate some approximate posterior, $q_\psi(s_t|o_{1:t}, a_{1:t})$. There are many choices for q_ψ. These include:

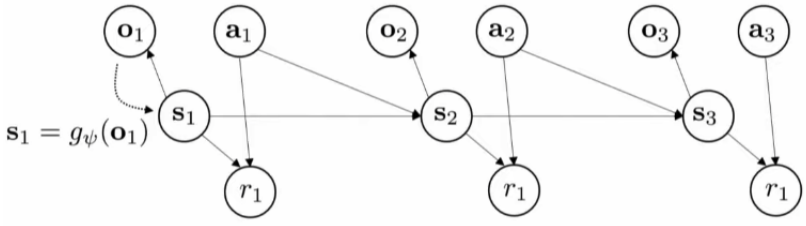
1. The full smoothing posterior, $q_\psi(s_t, s_{t+1}|o_{1:t}, a_{1:t})$. This can be complicated to calculate but will be the most accurate approximation.
2. A single-step encoder, $q_\psi(s_t|o_t)$. These are simpler to calculate, but comes at the cost of accuracy.

Training q_ψ can be done by variational inference. For simplicity, let us consider the case when $q(s_t|o_t)$ is a deterministic single-step encoder such that $q_\psi(s_t|o_t) = \delta(s_t = g_\psi(o_t)) \implies s_t = g_\psi(o_t)$. Our likelihood-like calculation then becomes

$$\max_{\phi,\psi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \log p_\phi(g_\phi(o_{t+1,i})|g_\psi(o_{t,i}), a_{t,i}) + \log p_\phi(o_{t,i}|g_\psi(o_{t,i})).$$

This is entirely differentiable and thus can be trained using the backpropagation algorithm. We can jointly optimise for ϕ, ψ. Putting all the models together we get

$$\max_{\phi,\psi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \log \underbrace{p_\phi(g_\phi(o_{t+1,i})|g_\psi(o_{t,i}), a_{t,i})}_{\text{Latent Space Dynamics}} + \log \underbrace{p_\phi(o_{t,i}|g_\psi(o_{t,i}))}_{\text{Reconstruction}} + \log \underbrace{p_\phi(r_{t,i}|g_\psi(o_{t,i}))}_{\text{Reward Model}}.$$



Algorithm 18 Latent Representation Model-Based Algorithm

Apply base policy $\pi_0(a_t, s_t)$ to collect a dataset $D = \{(s, a, s')_i\}$
for N steps **do**
  Learn $p_\phi(s_{t+1}|s_t, a_t)$, $p_\phi(r_t|s_t)$, $p(o_t|s_t)$, $g_\psi(o_t)$ (encoder).
  for K steps **do**
    Plan for action a' using models.
    Apply action a' and add observation transition (o, a, o') to D.
  end for
end for

6.2 Contrastive Predictive Coding (CPC) & InfoNCE

A useful technique in modern machine learning is to extract useful representations from high-dimensional data. Contrastive Predictive Coding is a method proposed by Oord et. al [26] in which prediction of future states in latent space using autoregressive models allows for learning.

The key concept behind CPC is to learn representations that encode some shared underlying information between different high-dimensional signals. In so doing, noise and low-level information is discarded.

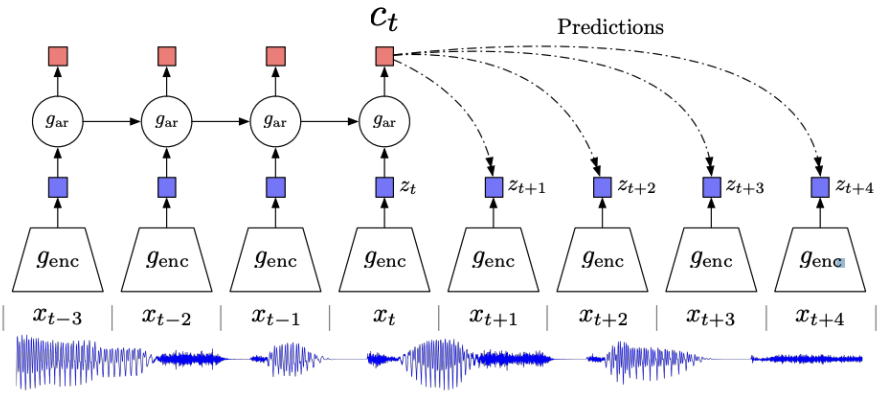


Figure 12: Overview of Contrastive Predictive Coding, the proposed representation learning approach. Although this figure shows audio as input, we use the same setup for images, text and reinforcement learning [26].

To perform such predictions, global structure needs to be inferred about the data - nullifying techniques which rely on local smoothness of data. Conventional prediction techniques rely on powerful conditional generative models which reconstruct the data to form predictions. This is computationlly intensive, and intractable in many cases. CPC instead encodes the target x (future state) and context c (present state) into vector representations in a way that maximally preserves the mutual information of the original signals x and c. Mutual information measures the amount of information an observation of one random variable gives you about another random variable.

$$MI(X;Y) = D_{KL}(p(x,y)||p(x)p(y)) = E_{p(x,y)}\left[\log \frac{p(x,y)}{p(x)p(y)}\right]$$

Mutual information can be challenging to compute. InfoNCE [26] is a loss which, when optimised for, gives a tractable lower bound on the mutual information.

$$MI(s_{t+k}; c_t) \geq E_s\left[\log \frac{f(z_{t+k}, c_t)}{f(z_{t+k}, c_t) + \sum_{s_j \epsilon S} f(z_j, c_t)}\right]$$

This method is shown to work well in the original paper. It performs well in reinforcement learning contexts as shown in the GridWorld example of figure 13 below.

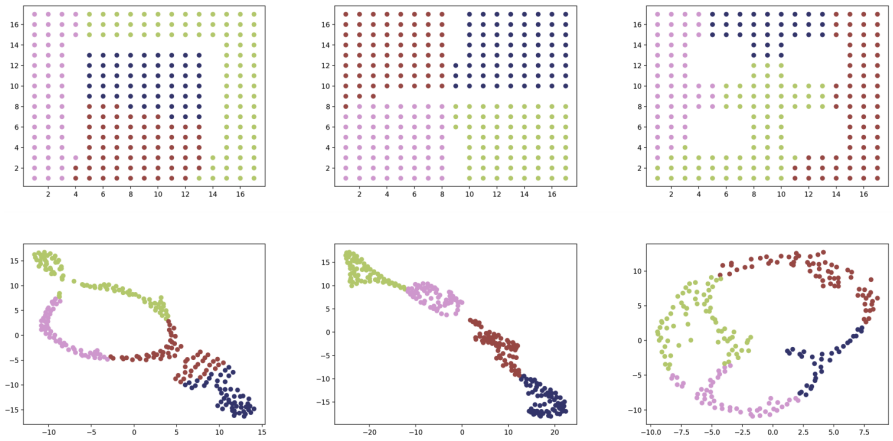


Figure 13: CPC representation of GridWorld environment: UMaze (left), Four-Rooms (middle), and Block-Maze (right). By clustering embeddings (bottom, all embeddings are visualized by T-SNE), we are able to recover clusters corresponding to natural structures of the mazes (top) [27].

6.3 Predictive Coding for Sparse Rewards

One of the challenges for modern reinforcement learning problems is that of sparse rewards. For effective and efficient learning to take place, an agent needs consistent feedback on its performance. In reinforcement learning, this feedback takes the form of a reward signal. Predicitive coding has been sugested as one method of boosting deep reinforcement learning with sparse rewards. In *Predictive Coding for Boosting Deep Reinforcement Learning with Sparse Rewards* [27], representation learning is applied to provide the agent with meaningful rewards without the need for specific domain knowledge. Predictive coding is used in an unsupervised learning context to extract features from the environment.

The first step is to train an encoder that extracts predictive features from states. Initial exploration is done to collect trajectories, which are then sampled from and used to train a CPC encoder. A fixed number of states are encoded into some latent embedding, which is then passed through a gated recurrent unit (GRU) to produce 'context'. This context is then used to predict the embedding of remaining states using some score function.

The embeddings are used in the reinforcement learning context by shaping rewards. This is done in two different ways.

1. **Clustering:** Random states are sampled from the environment and clustered according to their corresponding embeddings. An agent is rewarded for entering a cluster containing a reward.
2. **Negative Distance Optimisation:** For large state space environments, optimisation is done directly on the distance between the current state and goal state in embedding space. This is done by adding a penalty to the reward function corresponding to this distance. For each time step t, we add $-||z_t - z_g||^2$, where z_t is the embedding of the current state s_t and z_g that of the goal state.

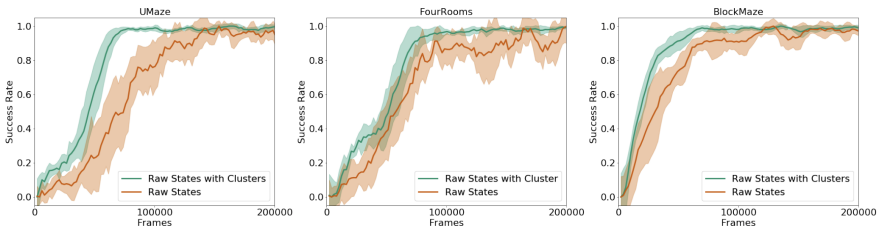


Figure 14: Success Rate in GridWorld Environments. A successful run indicates that the agent reaches the goal from a random stating potion in the maze within 100 steps [27].

Setup	U-Maze	Four-Rooms	Block-Maze
Without Clustering	61%	71%	98%
With Clustering	98%	98%	100%

Table 1: Success Rate in GridWorld Environments [27].

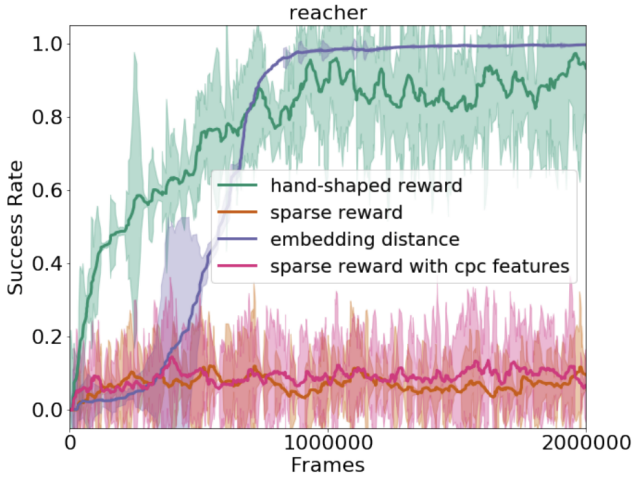


Figure 15: Illustration of learning curves for different setups. Learning on the original sparse reward problem with embeddings as features (pink) did not lead to significant learning improvement, while using embeddings to provide rewards (purple) achieved the best learning [27].

7 State-of-the-Art Models

In this section we explore various state-of-the-art approaches to learning, especially methods which make use of prediction, inference and world models. This section ties together much of the theory developed in earlier sections and tries to present a coherent picture of how the algorithms work with some level of detail as to the implementation. Looking at the original papers is, of course, encouraged.

7.1 Learning By Imagination

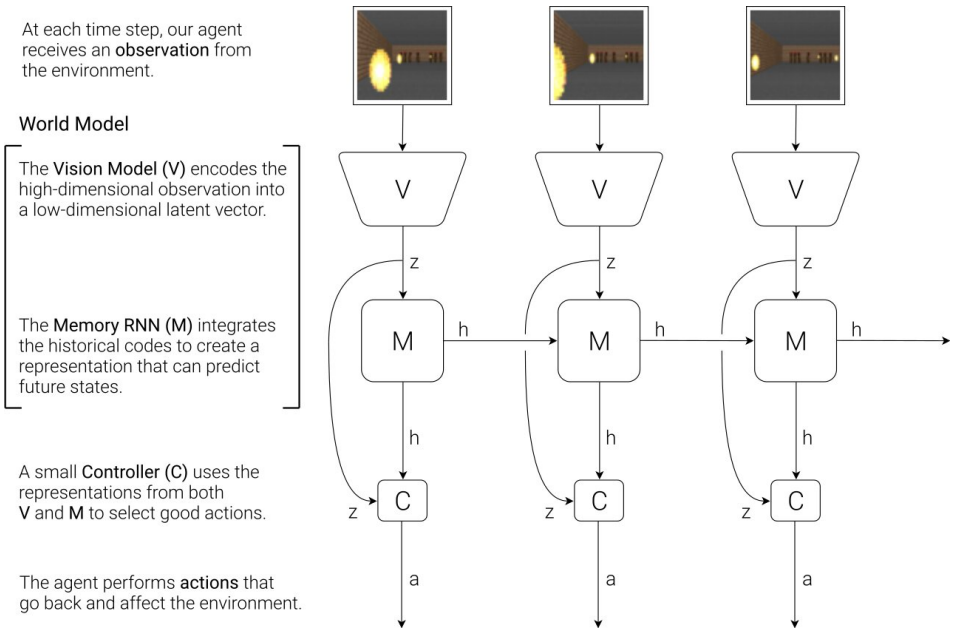
The World Models [28] paper presented at NIPS in 2018 exploits the idea of having an agent train entirely within its latent representation of the world it is in - its world model - and apply this learned knowledge to the real world. This paper distils many years and areas of research into a highly efficient framework for model based reinforcement learning. Taking a moment to understand this paper is useful because many of the advancements made since this paper released use ideas similar to the ones developed here.

Humans use world models to plan and make decisions quickly and efficiently - something model free algorithms have not been able to achieve to date. Of course, humans do not have a model of the same dimension as the world. To deal with the complexity of the world our brains form and learn abstract representations of "temporal and spatial aspects of information". As we have previously discussed, prediction plays a vital role in this process. By allowing an agent to *dream* and *imagine* about the world it is in, the agent can dramatically improve its sample efficiency and its overall performance. As discussed in [13], efficiency is said to be a crucial component of how we should measure intelligence of intelligent systems. This makes improving upon the sample inefficiencies in model free RL crucial for creating an *intelligent* system.

The key here in the world models paper to ensure that we know how limited we are based on how much information we have gathered about the environment. When building a model of our environment, we want to make sure that the information we have gathered to build a model is representative of the dynamics of the system. This model can then be trained in an unsupervised manner within the latent representations we form using the data we have gathered from the environment.

Let's take a step back. What are the pieces we need to build an agent that can learn by imagination?

1. Sample random rollouts according to a random policy.
2. Extract visual information (pixels to information) from the environment,
3. Ensure information is representative of the environment.
4. Form a latent representation, and
5. form predictions of the future states. Finally,
6. select appropriate actions.



The architecture consists of three components: the vision model (V), the memory model (M) and the controller (C). The vision and memory models form the 'world model'. This world model can accomplish steps 1-4 above. High dimensional data is encoded into a latent representation by the vision model. Historical data along with current observations allow prediction of future states. These predictions are used to select the best actions based on expected future returns.

A Variational Autoencoder (VAE) is used as the vision model. Essentially, the VAE tries to encode the important information encoded in the images as they change over time. For example, in the car racing environment the most important information is encoded in how the track changes from frame to frame. A latent representation would ideally capture only this information. The smaller the representation, the faster we can train the networks and form predictions. The VAE also learns a decoder. This allows the latent variable to be decoded.

An MDN-RNN is used for the memory model. This uses past events to output a prediction for a distribution of outcomes based on what has been and is currently being observed by the vision model. That is, given one observation the memory model outputs several predictions with an associated probability of that event occurring. This is nicely displayed is the graphic below. Interestingly, a reproduction [29] of this experiment found that training the MDN-RNN for the car racing environment does not improve performance, but removing the MDN-RNN does degrade performance. The investigators interpret this as a by considering that the MDN-RNN recurrent state contains crucial temporal information which is absent from individual frames.

Figure 16: This figure displays an example of the predictive power of RNNs, similar to the RNN used in the World Models paper. Here a single line leads to numerous predictions of what the drawing will look like given that it should be a cruise ship. This example comes from SketchRNN, a free online game for experimenting with RNNs [30].

Learning by imagination Conventionally one would form predictions to select the action which is expected to give the highest return. This action would be selected and the response of the environment would be used to update the model. This is done in the first half of the paper and state-of-the-art (at the time of release) performance is achieved. Naturally one wonders whether we could continue to train our models on the predicted latent variables. This is exactly what the authors proceed to do in the *VizDoom* environment. The memory model is amended so as to also predict whether or not the agent dies in the next frame. This is done to ensure the world model produces the same dynamics that the *VizDoom* environment would return.

Thus, by training entirely using the predictions formed by the world model, we can train our agent to be successful within the imagined world. The aim, of course, is not to simply be successful within the world model, but to be successful in the real world. Since the virtual environment has an identical interface to the real environment, the policy learned by imagination can be easily transferred and applied in the real world. This is a major insight in this paper.

The authors note that the agent can learn to exploit the imperfections in the world model. Adding a *temperature* τ parameter during the sampling of the predicted latent parameter increases the difficulty of surviving in the imagined environment and actually leads to improvements in transferred performance depending on the choice of τ. The intuition here is that it becomes harder for the agent to exploit these imperfections.

This paper is really quite remarkable in its deceptive simplicity. The individual components of the models presented can and have been improved since its publication, but the idea of training by imagination using latent representation to improve training performance was a crucial insight that is used in algorithms like SimPLE and MuZero.

A recent remarkable improvement in this area was presented in *Dream to Control: Learning Behaviours by Latent Imagination*, presented at ICLR 2020, is an incremental improvement on the success of World Models. This paper introduces an algorithm called **Dreamer** [31] which exceeds performance of existing approaches in 20 challenging visual control tasks. Dreamer is specifically geared toward deriving behaviour from long-horizon tasks with pixel (image) observations by latent imagination. This agent distinguishes itself from the approach in World Models by allowing the agent to efficiently and iteratively update by backpropagation, whereas World Models applied a non-iterative, derivative-free approach. The algorithm consists of three main components which can be run in parallel during training. These are

1. **World model:** A dataset of past experience is used to train the model.
2. **Behaviour:** The world model is used to learn behaviours purely by imagination using a value and actor network.
3. **Environment:** The agent interacts with the environment to grow the experience of the model.

The idea of training using limited experience in a simulated environment (imagination) follows from recent advancements in the understanding of how sleep and dreaming play an important role in human cognition and

skill acquisition (see [32] and [33] for example). Once again, advancements in neuroscience leads to an improved understanding of how we can increase performance in artificially intelligent agents. The key contributions in this paper are summarised by the authors as:

1. **Learning long-horizon behaviour by latent imagination:** both states and actions are predicted in long-horizon tasks, allowing for efficient learning purely of a policy by propagating analytic value gradients back through the latent dynamics.

2. **Improved emperical performance for visual control tasks:** applied to DeepMind's Control Suite environments for control tasks, Dreamer exceeds previous model-based and model-free agents in terms of data-efficiency, computation time, and final performance. Dreamer does all this using the same hyper parameters for all tasks in the suite - another mark of intelligence according to Chollet [13].

Dreamer uses an actor-critic learning procedure for behaviors that consider rewards beyond the horizon. An action model and a state model are learned in the latent space. The goals of the policy in the world model is to predict actions that solve the simulated, imagined, environment. The goal, of course, is that the learned policy will be effective in the real environment. Both the action and value models are trained using a policy iteration procedure. The value of states can be estimated in various ways. Dreamer uses an exponentially-weighted average of the estimates for a different number of steps, k, into the future. This has been found to balance bias and variance in practice. This tradeoff between bias and variance was discussed in earlier sections. See Sutton and Barto [10] for further detailed discussion on this trade-off. Formally, we can write this value estimation as

$$\mathrm{V}_{\lambda}\left(s_{\tau}\right)=(1-\lambda) \sum_{n=1}^{H-1} \lambda^{n-1} \mathrm{V}_{\mathrm{N}}^{n}\left(s_{\tau}\right)+\lambda^{H-1} \mathrm{V}_{\mathrm{N}}^{H}\left(s_{\tau}\right).$$

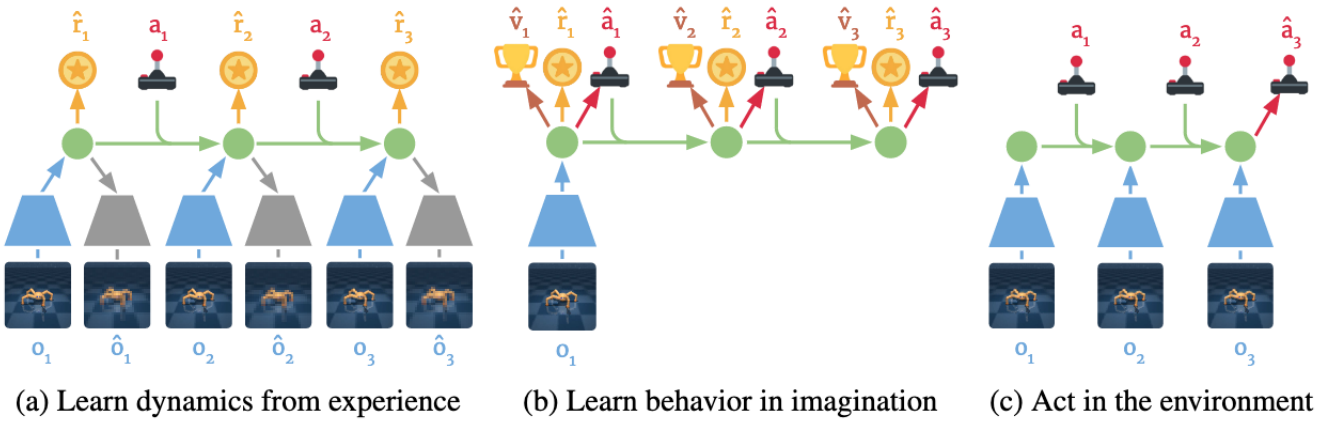


Figure 17: Components of Dreamer. (a) Using past experience, the agent learns to encode observations and actions into compact latent states. It can then and predict environment rewards. (b) In the compact latent space, Dreamer predicts state values and actions that maximise future return by propagating gradients back through imagined trajectories. (c) The agent encodes the history of the episode to compute the current model state and predict the next action to execute in the environment.

Algorithm 19 Dreamer

Initialise dataset D with S random seed episode.
Initialize neural network parameters θ, ϕ, ψ randomly.
while not converged **do**
  for update step $c = 1 \dots C$ **do**
    // Dynamics Learning
    Draw B sequences $\{(a_t, o_t, r_t)\}_{t=k}^{k+L} \sim D$.
    Compute model states $s_t \sim p_\theta(s_t|s_{t-1}, a_{t-1}, o_t)$.
    Update θ using representation learning.

    // Behaviour Learning
    Imagine trajectories $\{(s_\tau, a_\tau)\}_{\tau=t}^{t+H}$ from each s_t.
    Predict rewards $\mathrm{E}(q_\theta(r_\tau|s_\tau))$ and values $v_\psi(s_\tau)$.
    Compute value estimates $V_\lambda(s_\tau)$.
    Update $\phi \leftarrow \phi + \alpha\nabla_\phi \sum_{\tau=t}^{t+H} V_\lambda(s_\tau)$.
    Update $\psi \leftarrow \psi - \alpha\nabla_\psi \sum_{\tau=t}^{t+H} \frac{1}{2}||v_\psi(s_\tau) - V_\lambda(s_\tau)||^2$.
  end for

  // Environment Interaction
  $o_1 \leftarrow env.reset()$
  for time step $t = 1 \dots T$ **do**
    Compute $s_t \sim p_\theta(s_t|s_{t-1}, a_{t-1,a_t})$ from history.
    Compute $a_t \sim q_\phi(a_t|s_t)$ using the action model.
    Add exploration noise to the action.
    $r_t, o_{t+1} \leftarrow env.step(a_t)$.
  end for
  Add experience to dataset $D \leftarrow D \bigcup \{(o_t, a_t, r_t)_{t=1}^T\}$
end while

7.2 Free Energy of Expected Future

The active inference framework proposes agents act to maximise the evidence for a biased generative model, whereas in reinforcement learning the agent seeks to maximise the expected discounted cumulative reward. In *Reinforcement Learning Through Active Inference* [34] a new RL objective - the *free energy of the expected future* - is implemented by augmenting traditional learning techniques with concepts from active inference theory. This approach eliminates the need for training a world model and instead applies a generative model to perform active inference. This approach has two important side effects.

1. A balance between exploration and exploitation is intrinsic.
2. It allows for a sparsity and lack of reward.

Both RL and active inference highlight the importance of probabilistic models, planning efficiently, and inferring explicitly about the environment. In the context of active inference, agents seek to maximise their Bayesian model evidence for their generative model $p^\Phi(o, s, \theta)$, where θ are model parameters. This generative model can be biased towards observations that are likely and beneficial for an agent's success. Rewards are treated as prior probabilities over some observation, o. The divergence, D_{KL}, between the preferred and the expected outcome is used as a measure of the success of this model.

Recall from section 2.2 that agents can perform approximate Bayesian inference by minimising the variational free energy. Given some distribution $q(s, \theta)$, and some generative model $p^\Phi(o, s, \theta)$, the free energy is given by

$$F = D_{KL}\left[q(s, \theta)||p^\Phi(o, s, \theta)\right].$$

In this formulation, we treat the model parameters θ as random variables. We are thus treating the learning process as a process of approximate inference. In this formulation, agents maintain beliefs about about policies $\pi = o_0, \dots, o_T$. Policy selection can thus be formulated as a process of approximate inference over the distribution of policies, $q(\pi)$.

Since a policy holds information about a sequence of variables in time, the free energy formulation needs to be altered so as to incorporate future variables. This leads directly to the formulation of the *free energy of the expected future* - a quantity we wish to minimise, defined as

$$\tilde{F} = D_{KL}\left[q(o_{0:T}, s_{0:T}, \theta, \pi)||p^\Phi(o_{0:T}, s_{0:T}, \theta)\right],$$

where $q(o_{0:T}, s_{0:T}, \theta, \pi)$ is the agent's belief about future variables, and $p^{\Phi}(o_{0:T}, s_{0:T}, \theta)$ is the generative model.

$$\tilde{F} = 0 \implies D_{KL}\left[q(\pi)||-e^{-\tilde{F}_{\pi}}\right] = 0$$

where

$$\tilde{F}_{\pi} = D_{KL}\left[q(o_{0:T}, s_{0:T}, \theta|\pi)||p^{\Phi}(o_{0:T}, s_{0:T}, \theta)\right].$$

This gives the result that the free energy of the expected future is minimised when $q(\pi) = \sigma(-\tilde{F}_{\pi})$. Therefore, policies that minimise $\tilde{F}_{\pi}$ are more likely.

Naturally, the question arises as to what minimising this quantity, $\tilde{F}_{\pi}$, means. Assuming the model is only biased in its beliefs of observations, the authors propose factorising the agent's generative model as follows:

$$p^{\Phi}(0_{0:T}, s_{0:t}, \theta) = p(s_{0:t}, \theta|0_{0:T})p^{\Phi}(0_{0:T}).$$

Here, $p^{\Phi}(0_{0:T})$ is a distribution over preferred rewards - since this is the goal in RL. Using this factorisation, it can be shown that $\tilde{F}_{\pi}$ can be broken down into two terms representing distinct concept,

$$-\tilde{F}_{\pi} \simeq -\underbrace{\mathrm{E}_{q(0_{0:T}|\pi)}\left[D_{KL}\left[q(s_{0:T}, \theta|o_{0:T}, \pi)||q(s_{0:T}, \theta|\pi)\right]\right]}_{\text{Expected Information Gain}} + \underbrace{\mathrm{E}_{q(s_{0:T},\theta|\pi)}\left[D_{KL}\left[q(o_{0:T|s_{0:T},\theta,\pi})||p^{\Phi}(o_{0:t})\right]\right]}_{\text{Extrinsic Term}}.$$

The *expected information gain* is crucial as it quantifies the amount of information an agent believes it will gain by following some policy. This term promotes exploration of the state and parameter spaces, since agents hold beliefs about both the state of the environment and its own model parameters. The minimisation of the *extrinsic term* is, by definition, the minimisation of the difference between what the agent believes about future observations and what it would prefer to observe. In other words, it measures how much reward it expects to see in the future in contrast to how much reward it would like to achieve. This is exploitation. This result is key as a natural balance between exploration and exploitation is a fundamental issue in reinforcement learning in general.

This objective is evaluated in three steps:

1. Evaluate beliefs,
2. Evaluate $\tilde{F}_{\pi}$, and
3. Optimise $q(\pi)$ such that $q(\pi) = \sigma(-\tilde{F}_{\pi})$

The first step, evaluating beliefs, can be formally described similarly to reinforcement learning.

$$\begin{aligned}
q\left(\mathrm{s}_{t:T}, \mathrm{o}_{t:T}, \theta \mid \pi\right) &= q(\theta) \prod_{t=\tau}^{T} q\left(\mathrm{o}_{\tau} \mid \mathrm{s}_{\tau}, \theta, \pi\right) q\left(\mathrm{s}_{\tau} \mid \mathrm{s}_{\tau-1}, \theta, \pi\right) \\
q\left(\mathrm{o}_{\tau} \mid \mathrm{s}_{\tau}, \theta, \pi\right) &= E_{q(s_{\tau}|\theta,\pi)}\left[p\left(\mathrm{o}_{\tau} \mid \mathrm{s}_{\tau}\right)\right] \\
q\left(\mathrm{s}_{\tau} \mid \mathrm{s}_{\tau-1}, \theta, \pi\right) &= E_{q(s_{\tau-1}|\theta,\pi)}\left[p\left(\mathrm{s}_{\tau} \mid \mathrm{s}_{\tau-1}, \theta, \pi\right)\right]
\end{aligned}$$

We can then efficiently calculate the free energy of the expected future given beliefs about future variables. For a single time step, this can be computed as

$$\begin{aligned}
-\tilde{\mathcal{F}}_{\pi_{\tau}} \approx\ & E_{q(\mathbf{s}_{\tau},\theta|\pi)}\left[D_{\mathrm{KL}}\left(q\left(\mathbf{o}_{\tau} \mid \mathbf{s}_{\tau}, \theta, \pi\right) \| p^{\Phi}\left(\mathbf{o}_{\tau}\right)\right)\right] \\
& + \underbrace{\mathbf{H}\left[q\left(\mathbf{o}_{\tau} \mid \pi\right)\right] - E_{q(\mathbf{s}_{\tau}|\pi)}\left[\mathbf{H}\left[q\left(\mathbf{o}_{\tau} \mid \mathbf{s}_{\tau}, \pi\right)\right]\right]}_{\text{State information gain}} \\
& + \underbrace{\mathbf{H}\left[q\left(\mathbf{s}_{\tau} \mid \mathbf{s}_{\tau-1}, \theta, \pi\right)\right] - E_{q(\theta)}\left[\mathbf{H}\left[q\left(\mathbf{s}_{\tau} \mid \mathbf{s}_{\tau-1}, \pi, \theta\right)\right]\right]}_{\text{Parameter information gain}}
\end{aligned}$$

Finally, the policy distribution can be optimised. Here a full cross-entropy method (CEM) approach is presented as in the original paper [34].

Algorithm 20 Inference of $q(\pi)$

Initialise factorised belief over action sequences $q(\pi) \rightarrow N(0,1)$
for optimisation iteration $i = 1 \dots I$ **do**
  Sample J candidate policies from $q(\pi)$
  for candidate policy $j = 1 \dots J$ **do**
    $\pi^{(j)} \sim q(\pi)$
    $-\tilde{\mathcal{F}}_{\pi}^{j} = 0$
    for $\tau = t \dots t + H$ **do**
      $q(s_\tau|s_{\tau-1}, \theta, \pi^{(j)}) = \mathrm{E}_{q(s_{\tau-1}|\theta,\pi^{(j)})}\left[p(s_\tau|s_{\tau-1}, \theta, \pi^{(j)})\right]$
      $q(o_\tau|s_\tau, \theta, \pi^{(j)}) = \mathrm{E}_{q(s_\tau|\theta,\pi^{(j)})}\left[p(o_\tau|s_\tau)\right]$
      $-\tilde{\mathcal{F}}_{\pi}^{j} \leftarrow -\tilde{\mathcal{F}}_{\pi}^{j} + \mathrm{E}_{q(s_\tau|\theta,\pi^{(j)})}\left[D_{KL}(q(o_\tau|s_\tau, \theta, \pi^{(j)}))||p^{\Phi}(o_\tau)\right]$
        $+\boldsymbol{H}[q(s_\tau|s_{\tau-1}, \theta, \pi^{(j)})] - \mathrm{E}_{q(\theta)}\left[\boldsymbol{H}[q(s_\tau|s_{\tau-1}, \theta, \pi^{(j)})]\right]$
    end for
  end for
  $q(\pi) \rightarrow \mathrm{refit}(-\tilde{\mathcal{F}}_{\pi}^{j})$
end for
return $q(\pi)$

7.3 SimPLe

Simulated Policy Learning (SimPLe) [35] is a complete model-based RL algorithm which uses prediction models to drastically improve sample efficiency and outperform state-of-the-art model-free methods in the Atari Learning Environment (ALE) - an environment which has traditionally favoured model-free methods. One difficulty of this environment is that observations are only partially observed, and thus do not necessarily convey the true underlying state of the system. The intuition is that SimPLe *learns* how the games work, and thus can quickly learn to predict the actions that need to be taken to get desirable outcomes.

The SimPLe algorithm was developed with the solving games in the ALE - a de facto benchmarking suite for RL algorithms. SimPLe managed to advance the state-of-the-art in model-based RL methods by experimenting with stochastic video prediction techniques and discrete latent variables (representation learning). SimPLe is especially impressive in low-data regimes, where it outperforms previous state-of-the-art methods (Rainbow [36]) using only half the number of samples in many of the games - even when they are tuned for sample efficiency.

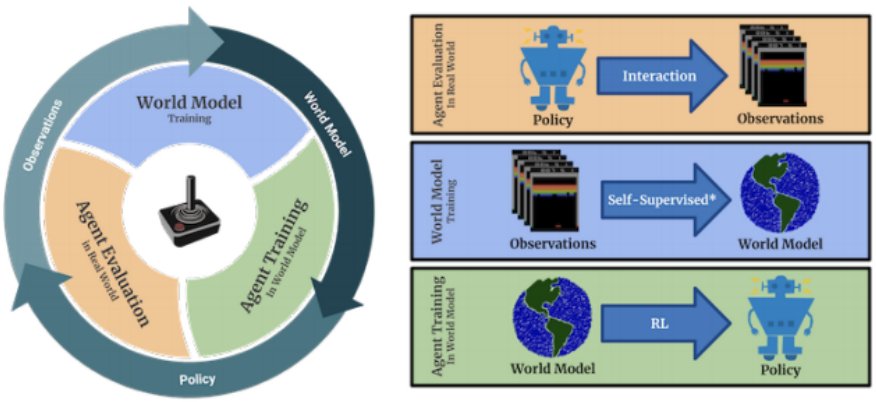


Figure 18: Main loop of SimPLe. 1) the agent starts interacting with the real environment following the latest policy (initialised to random). 2) the collected observations will be used to train (update) the current worl model. 3) the agent updates the policy by acting inside the world model. The new policy will be evaluate to measure the performance of the agent as well as collecting more data (back to 1). Note that world model training is self-supervised for the observed states and supervised for the reward [35].

In Atari games, the goal is to find a policy π such that following π maximises the expected discounted return from the beginning of the game - the value function $E_\pi[\sum_{-\infty}^{\infty} \gamma^t r_{t+1}|s_0 = s]$. In SimPLe, a neural network is used to simulate the dynamics of the environment *env* by some approximation *env*$'$. This *world model* shares both the action and reward spaces of *env* and produces predicted (expected) visual observations in the same format of the true environment. A sub-goal of SimPLe is thus to accurately mimic *env* with *env*$'$ using as

few interactions with *env* as possible. Random trajectories are sampled from *env* and used to train env' as in algorithm 21 below.

Algorithm 21 SimPLe

Initialise policy π
Initialise model parameters θ of env'
Initialise empty set D
while `not done` **do**
  D $\longleftarrow$ D $\cup$ COLLECT(env, π)
  $\theta \longleftarrow$ TRAIN_SUPERVISED(env', D)
  $\pi \longleftarrow$ TRAIN_RL(π, env')
end while

In the case of the of the SimPLe paper, proximal policy optimisation (PPO) was used to train the policy, as indicated by TRAIN_RL in algorithm 21. A discount factor of $\gamma = 0.95$ was used. As stated earlier, the algorithm randomly samples trajectories from the environment to train the world model. The imperfections in this approximated model, env', can compound over time. For this reason, the original paper suggests uniformly sampling the initial state from the ground-truth buffer D and restarting env' every N steps (typically, N=50). After N steps, the evaluation of the value function is added to the reward. This is done to ensure the PPO algorithm is not degraded by the short random rollouts (trajectories).

The explicit architecture of SimPLe as in the original paper can be summarised as follows.

- **Deterministic Model**: Input to the model consists of four consecutive frames and some action a. Stacked convolution layers process the input. Actions are vectorised and multiplied with output from convolutional layers. The next frame of the game and associated reward is predicted.
- **Loss functions**: A clipped loss $\max(Loss, C)$ (constant C) was used to improve the model. The authors suggested that clipping removes the weight given to fine-tuning of image backgrounds in Atari games, allowing for focus on important aspects of the game (e.g. ball in Pong).
- **Scheduled sampling**: The model env' is susceptible to compound errors and drift since it is trained by its own predictions. This is mitigated by random replacement of input to the model by a prediction from the previous step.
- **Stochastic Models**: Even though Atari games are deterministic in their nature, given partial observability and limited horizon of past observations they can appear stochastic to the agent. The proposed model includes a stochastic component, which means SimPLe can be used with truly stochastic environments. An additional network receives both the input frames and the target to approximate the posterior. Latent values z_t are sampled at each time step, and passed as input to the predictive model. During testing, latent values are sampled from an assumed prior distribution. A KL-divergence term is added to the loss term to ensure matching of assumed and approximate prior distributions.

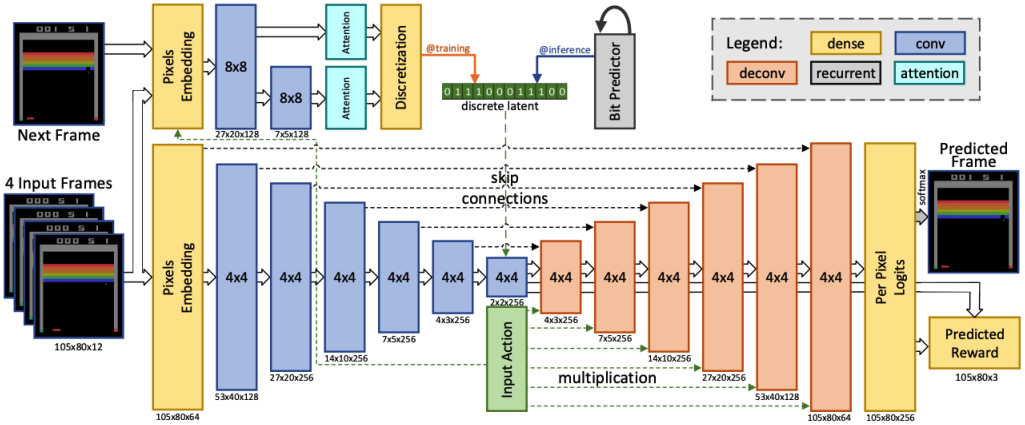


Figure 19: Architecture of the proposed model. The input to the model is four stacked frames (as well as the action selected by the agent) while the output is the next predicted frame and expected reward. Input pixels and action are embedded using fully connected layers. First, the bottom part of the network which consists of a skip-connected convolutional encoder and decoder. To condition the output on the actions of the agent, the output of each layer in the decoder is multiplied with the (learned) embedded action. Second part of the model is a convolutional inference network which approximates the posterior given the next frame. During training the sampled latent values from the approximated posterior will be discretized. A third LSTM based network is trained to approximate each bit given the previous ones. At inference time, the latent bits are predicted auto-regressively using this network. The deterministic model has the same architecture as this figure but without the inference network [35].

7.4 MuZero

One of the most important reinforcement learning results in recent times is given in *Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model* [37]. This paper is important as it provides a new state-of-the-art in the Atari ALE which is visually complex and has favoured model-free approaches in the past. It also matches superhuman performance in planning tasks such as Go, Chess and Shogi. Adding to this, MuZero does not rely on knowledge of the environment's dynamics. This work builds upon AphaZero [38] by including a learned model in the training procedure.

Once again, prediction plays a key role in the success of this agent. MuZero aims to predict the aspects of the future that will be directly relevant for planning. Observations are received as input and transformed into a hidden (latent) state. At each time-step, the previous hidden state and a possible action is received. The model uses this input to predict the correct policy, the value function, and the immediate reward that will be received. The model is trained so as to accurately predict these quantities.

At each time-step t, and for each roll-out step $k = 1...K$, a model μ_θ conditioned on past observations $o_1, \ldots, o_t$ and future actions $a_1, \ldots, a_t$ is used to make predictions of the policy p_t^k, value function v_t^k and immediate reward r_t^k. Here,

$$p_t^k \simeq \pi(a_{t+k+1}|o_1, \ldots, o_t, a_1, \ldots, a_t),$$
$$v_t^k \simeq E\left[u_{t+k+1} + \gamma u_{t+k+2} + \ldots | o_1, \ldots, o_t, a_1, \ldots, a_t\right],$$
$$p_t^k \simeq u_{t+k},$$

where π is the policy to select real actions, u is the true reward given by he environment, and γ is the discount factor.

The model itself is built up of three components - a representation function, a dynamics function, and a prediction function. The dynamics function is a recurrent process which takes in some state-action pair, (s^{k-1}, a^k), and returns a state s^k and associated reward r^k. Note, the states in this dynamics function do not represent those of the true environment. They merely serve to the purpose of accurately predicting the dynamics of the true environment - that is, mimicking the MDP. This mimicry is the job of the representation function. The prediction function takes in the state s^k and computes a policy and value function.

Given that we have a dynamics function, we can apply planning algorithms to state spaces and internal rewards. MuZero applies MTCS at each time step. An estimated policy π_t and an estimated value v_t, from which an action $a_{t+1} \sim \pi_t$ is selected. Selection is done in proportion the the visit count for each action from the root node.

The different components of the model are jointly trained so as to most accurately represent to MDP of the true environment. There are three main objectives during this training. These are minimising the error between

1. the predicted policy p_t^k and search policy π_{t+k},
2. the predicted value v_t^k and the target value z_{t+k}, and
3. the predicted reward r_t^k and the observed rewards u_{t+k}.

The final aspect of the model is an L2 regularisation term. The overall loss function for the model is thus:

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

where l^r, l^v, l^p are loss function for the reward, value and policy respectively.

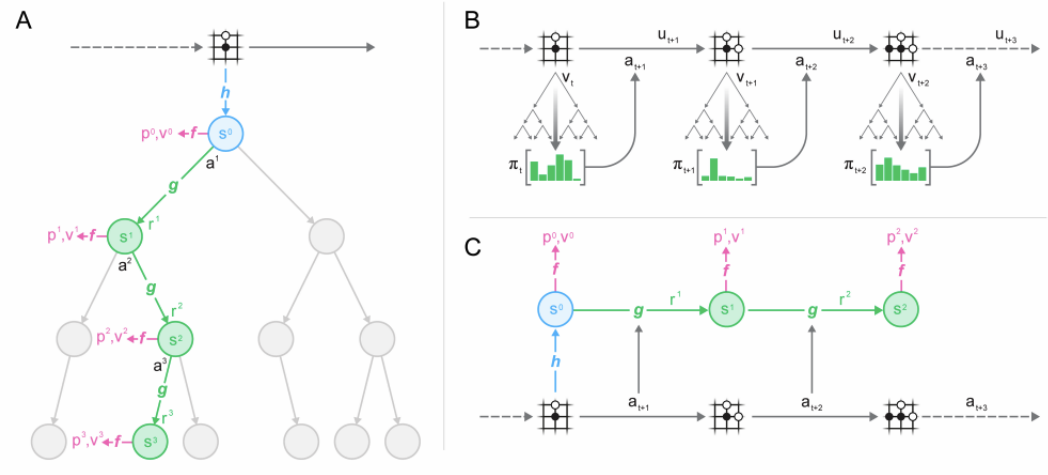


Figure 20: **Planning, acting, and training with a learned model**.
(A) How MuZero uses its model to plan. The model consists of three connected components for representation, dynamics and prediction. Given a previous hidden state s_{k-1} and a candidate action a_k, the dynamics function g produces an immediate reward r_k and a new hidden state s_k. The policy p_k and value function v_k are computed from the hidden state s_k by a prediction function f. The initial hidden state s_0 is obtained by passing the past observations (e.g. the Go board or Atari screen) into a representation function h. **(B)** How MuZero acts in the environment. A Monte-Carlo Tree Search is performed at each time step t, as described in A. An action a_{t+1} is sampled from the search policy π_t, which is proportional to the visit count for each action from the root node. The environment receives the action and generates a new observation o_{t+1} and reward u_{t+1}. At the end of the episode the trajectory data is stored into a replay buffer. **(C)** How MuZero trains its model. A trajectory is sampled from the replay buffer. For the initial step, the representation function h receives as input the past observations $o_1, \ldots, o_t$ from the selected trajectory. The model is subsequently unrolled recurrently for K steps. At each step k, the dynamics function g receives as input the hidden state s_{k-1} from the previous step and the real action a_{t+k}. The parameters of the representation, dynamics and prediction functions are jointly trained, end-to-end by back-propagation-through-time, to predict three quantities: the policy $p_k \simeq \pi_{t+k}$, value function $v_k \simeq z_{t+k}$, and reward $r_{t+k} \simeq u_{t+k}$, where z_{t+k} is a sample return: either the final reward (board games) or n-step return (Atari) [37].

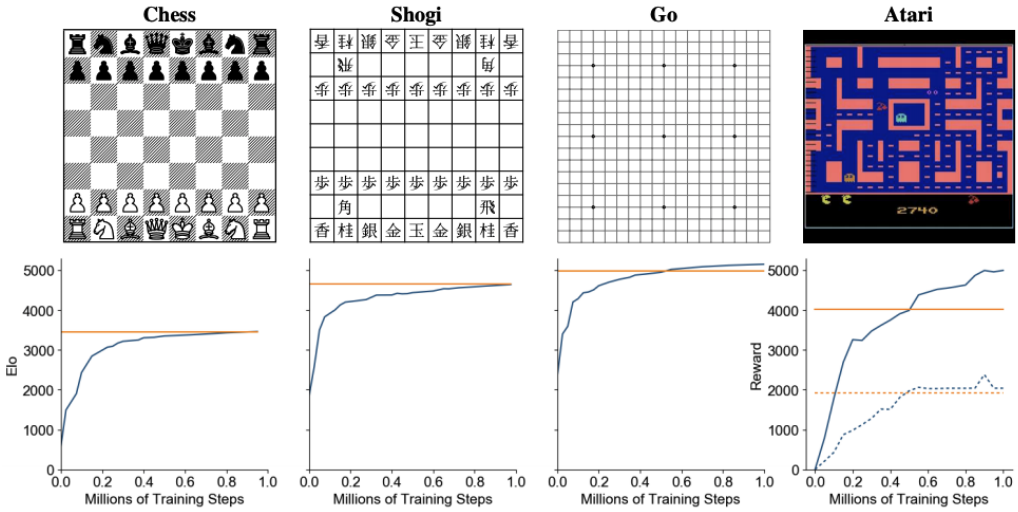


Figure 21: The following is taken directly from the original paper [37]. **Evaluation of MuZero throughout training in chess, shogi, Go and Atari**. The x-axis shows millions of training steps. For chess, shogi and Go, the y-axis shows Elo rating, established by playing games against AlphaZero using 800 simulations per move for both players. MuZero's Elo is indicated by the blue line, AlphaZero's Elo by the horizontal orange line. For Atari, mean (full line) and median (dashed line) human normalized scores across all 57 games are shown on the y-axis. The scores for R2D2 [39] (the previous state of the art in this domain, based on model-free RL) are indicated by the horizontal orange lines. Performance in Atari was evaluated using 50 simulations every fourth time-step, and then repeating the chosen action four times.

8 Simulations & Implementation

We are now ready to discuss an implementation of the MuZero agent presented in the previous section. This implementation is based on the work of Johan Gras [40]. For simplicity, this model is implemented on OpenAI's CartPole environment with some minor differences from the full MuZero agent implemented by DeepMind.

1. Fully connected layers are used instead of convolutional layers. Gras points out that the nature of the CartPole environment has no spatial correlations in the vector of observations, as the MuZero agent would experience when interacting with Atari environments.

2. Instead of scaling the hidden state to a probability $[0, 1]$ using min-max, the tanh function is used to map values on $(-1, 1)$.

3. The more complex transform for value prediction is simplified by removing linear terms.

4. No prioritised replay is used. Instead, random draws from a uniform distribution is used.

5. Loss is not scaled by the number of unrolled steps. It is assumed the number of unrolled steps is constant.

The pseudocode provided by DeepMind in the original paper [37] has a structure based on the theory discussed in the previous section. This is shown in detail in figure 22. We start off by defining the muzero agent. Here we initialise the shared storage and replay buffer. We run the training loop and output the training score to measure the performance as the algorithm is training. In the full implementation, seperate jobs would be initialised to train the agent in parallel. This is not necessary for the simple application of CartPole.

```
def muzero(config: MuZeroConfig):
  storage = SharedStorage()
  replay_buffer = ReplayBuffer(config)

for loop in range(config.nb_training_loop):
    print("Training loop", loop)
    score_train = run_selfplay(config, storage, replay_buffer, config.nb_episodes)
    train_network(config, storage, replay_buffer, config.nb_epochs)

    print("Train score:", score_train)
    print("Eval score:", run_eval(config, storage, 50))
    print(f"MuZero played {config.nb_episodes * (loop + 1)}" f"episodes and trained for
        {config.nb_epochs * (loop + 1)} epochs.\n")
```

```
    train_network(config, storage, replay_buffer)

    return storage.latest_network()
```

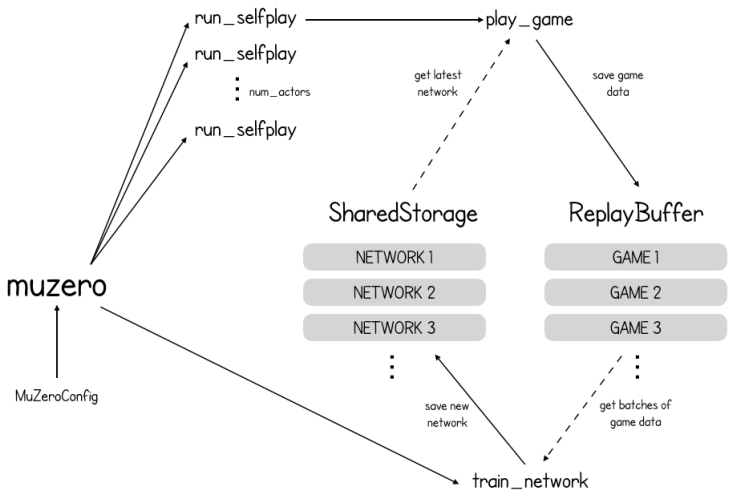


Figure 22: Figure showing the structure of the python implementation of DeepMind's MuZero agent.

Next we setup the shared storage and replay buffers. For brevity we leave the implementations out of this discussion (see [37], [40]). Next, we have to set up one of the most crucial parts of MuZero - the self-play. This feature draws on the work of AlphaZero and is a major reason MuZero can so effectively learn. In the full implementation each parallel job runs the *run_selfplay()* function that takes the latest version of the network from the memory bank and plays a game with it. The details of this game is then saved to the shared buffer.

```
def run_selfplay(config: MuZeroConfig, storage: SharedStorage,
                 replay_buffer: ReplayBuffer):
  while True:
    network = storage.latest_network()
    game = play_game(config, network)
    replay_buffer.save_game(game)
```

The next step of the implementation is building the three networks that make up the inference process MuZero goes through during training. Recall, these are the prediction, dynamics, and representation networks. The structure of this network code is given below. Here the networks are optimised together. This is to ensure that strategies that perform well inside the imagined environment translate well into real world performance.

```
class Network(object):
  def initial_inference(self, image) -> NetworkOutput:
    # representation + prediction function
    return NetworkOutput(...)

  def recurrent_inference(self, hidden_state, action) -> NetworkOutput:
    # dynamics + prediction function
    return NetworkOutput(...)

  def get_weights(self):
    # Returns the weights of this network.
    return ...

  def training_steps(self) -> int:
    # How many steps / batches the network has been trained for.
    return ...
```

Next up, we have to actually run a game! This is where the interesting planning theory we discussed comes into play. First we create a game object according to the configuration we set up. We then repeat proceed with the Monte-Carlo Tree Search procedure of expanding a node in the state-action space, exploring where necessary. MCTS is used to select the action to take, and the performance (value) of this action is stored. The

nodes themselves store valuable information about how often they have been visited and whether or not it has children. This helps when selecting a node for exploration vs exploitation.

```
def play_game(config: MuZeroConfig, network: Network) -> Game:
  game = config.new_game()
  while not game.terminal() and len(game.history) < config.max_moves:
    root = Node(0)
    current_observation = game.make_image(-1)
    expand_node(root, game.to_play(),
        game.legal_actions(),network.initial_inference(current_observation))
    add_exploration_noise(config, root)

    run_mcts(config, root, game.action_history(), network)
    action = select_action(config, len(game.history), root, network)
    game.apply(action)
    game.store_search_statistics(root)
  return game
```

The MCTS implementation is fairly standard and follows from the pseudocode discussed in section 5.1. Selecting a node follows from taking the node with the highest UCB score. Recall this can be written as

$$a^k = \arg\max_a \left[Q(s,a) + P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)} \left(c_1 + \log\left(\frac{\sum_b N(s,b) + c_2 + 1}{c_2} \right) \right) \right].$$

The value predicted by the network based on this selection is back-propagated up the tree of possible state-actions - along the search path taken. After *num_simulations* passes through the tree, we stop the search and select the action with the number of times each child node of the root has been visited.

```
def select_child(config: MuZeroConfig, node: Node,min_max_stats: MinMaxStats):
  _, action, child = max((ucb_score(config, node, child, min_max_stats), action,child) for action,
      child innode.children.items())
  return action, child
```

Finally we get to the training step. How does MuZero actually improve? As in the full implementation, we apply a momentum based optimiser and have a decaying learning rate. The agent is trained in batches and weights are updated as the training progresses.

```
def train_network(config: MuZeroConfig, storage: SharedStorage,replay_buffer: ReplayBuffer):network =
    Network()
  learning_rate = config.lr_init * config.lr_decay_rate**(tf.train.get_global_step() /
      config.lr_decay_steps)
  optimizer = tf.train.MomentumOptimizer(learning_rate, config.momentum)

  for i in range(config.training_steps):
    if i % config.checkpoint_interval == 0:
      storage.save_network(i, network)
    batch = replay_buffer.sample_batch(config.num_unroll_steps, config.td_steps)
    update_weights(optimizer, network, batch, config.weight_decay)
  storage.save_network(config.training_steps, network)
```

The final piece of the puzzle is how to update the weights of the network. We apply the loss function we developed in the theory section,

$$l_t(\theta) = \sum_{k=0}^{K} l^r\left(u_{t+k}, r_t^k\right) + l^v\left(z_{t+k}, v_t^k\right) + l^p\left(\pi_{t+k}, \mathbf{p}_t^k\right) + c\|\theta\|^2.$$

These losses over the entire rollout are added together to generate the loss for a given position in a specific batch. Large weights are penalised by a regularisation term to ensure no drastic changes occur.

```
def update_weights(optimizer: tf.train.Optimizer, network: Network, batch,weight_decay: float):
  loss = 0
  for image, actions, targets in batch:
    value, reward, policy_logits, hidden_state = network.initial_inference(image)
    predictions = [(1.0, value, reward, policy_logits)]

    for action in actions:
```

```
      value, reward, policy_logits, hidden_state = network.recurrent_inference(hidden_state, action)
      predictions.append((1.0 / len(actions), value, reward, policy_logits))

      hidden_state = tf.scale_gradient(hidden_state, 0.5)

    for prediction, target in zip(predictions, targets):
      gradient_scale, value, reward, policy_logits = prediction
      target_value, target_reward, target_policy = target
      l = (
          scalar_loss(value, target_value) +
          scalar_loss(reward, target_reward) +
          tf.nn.softmax_cross_entropy_with_logits(logits=policy_logits, labels=target_policy)
        )
      loss += tf.scale_gradient(l, gradient_scale)

  for weights in network.get_weights():
    loss += weight_decay * tf.nn.l2_loss(weights)
  optimizer.minimize(loss)
```

And that's MuZero! Of course, many of the details of the implementation are left out here. The important details are captured. It turns out, however, that training this algorithm on a simple game like CartPole, without parallel jobs, large computational power and detailed hyper parameter tuning, is not very effective. This following figure shows performance of a short training run of this implementation of MuZero on the CartPole environment.

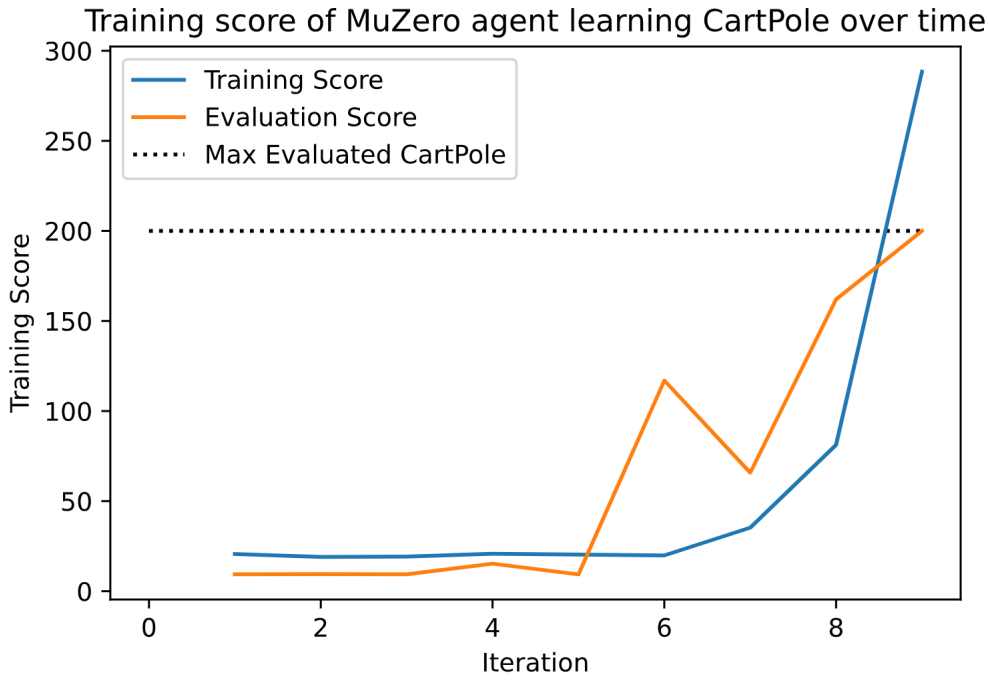


Figure 23: Graph showing the training and evaluation scores of the MuZero algorithm implemented in this section when tested on the CartPole environment.

9 Conclusions

This paper started by introducing the field of reinforcement learning and laying the groundwork for key ideas in modern neuroscience, optimal control and reinforcement learning. Some important recurring problems such as exploration and exploitation, as well as sample efficiency and trade-offs were presented. A comprehensive introduction to model-free followed. A detailed introduction to model-based methods of planning and learning with and without derivative based approaches was presented, before exploring some advanced ideas of representation and prediction in reinforcement learning. Finally, some state-of-the-art predictive and model-based methods were explored.

Possibility for Future Investigation

There is much room for exploration in this area, and new state-of-the-art methods seemingly appear everyday with the current pace of innovation and research in this area. Though little emphasis was placed on generative model approaches to inference and control, the benefits of free-energy approaches to reinforcement learning should be explored. I strongly believe that the ideas of imagination and building a model of the world around us is a crucial component for a human-like intelligence. It is also evident that these methods can greatly improve performance in both general and specific skill-based tasks.

It is clear that the fields of neuroscience and reinforcement learning have lots to contribute to each other. Implementation in artificial intelligence form a test-bed for what theories 'work' in the real world. Neuroscience contributes by adding new ideas for direction of research and ideas for practical and theoretical implementations for artificial agents. I only see this connection getting stronger in the future as ideas such as curiosity, causality and other intrinsic motivations used in unsupervised reinforcement learning becomes more popular in the academic community. The interesting ideas of free energy and inference will undoubtedly play a part in how artificially intelligent agents are designed in the future. This certainly warrants further investigation.

Acknowledgements

I would like to thank Dr Jonathan Shock for supervising this research-based coursework, and providing a platform for exploring this field.

References

[1] Karl Tuyls, Julien Pérolat, Marc Lanctot, Joel Z. Leibo, and Thore Graepel. A generalised method for empirical game theoretic analysis. In *AAMAS*, 2018.

[2] Petter N. Kolm and Gordon Ritter. Modern perspectives on reinforcement learning in finance. *Econometrics: Mathematical Methods Programming eJournal*, 2019.

[3] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

[4] Massimo Silvetti and Tom Verguts. Reinforcement learning, high-level cognition, and the human brain. 2012.

[5] Tingwu Wang, Xuchan Bao, Ignasi Clavera, Jerrick Hoang, Yeming Wen, Eric Langlois, S. Zhang, Guodong Zhang, Pieter Abbeel, and Jimmy Ba. Benchmarking model-based reinforcement learning. *ArXiv*, abs/1907.02057, 2019.

[6] Adam Linson, Andy Clark, Subramanian Ramamoorthy, and Karl J. Friston. The active inference approach to ecological perception: General information dynamics for natural and artificial embodied cognition. *Frontiers Robotics AI*, 5:21, 2018.

[7] Giovanni Pezzulo, Francesco Rigoli, and Karl Friston. Active inference, homeostatic regulation and adaptive behavioural control. *Progress in Neurobiology*, Volume 134:Pages 17–35, November 2015.

[8] Rafal Bogacz. A tutorial on the free-energy framework for modelling perception and learning. *Journal of Mathematical Psychology*, Volume 76, Part B:Pages 198–211, February 2017.

[9] Karl J. Friston. What is optimal about motor control? *Neuron*, 72:488–498, 2011.

[10] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. MIT Press, second edition edition, 2018.

[11] Maor Gaon and Ronen I. Brafman. Reinforcement learning with non-markovian rewards. In *AAAI*, 2020.

[12] Gabriel Dulac-Arnold, Daniel J. Mankowitz, and Todd Hester. Challenges of real-world reinforcement learning. *ArXiv*, abs/1904.12901, 2019.

[13] Franccois Chollet. On the measure of intelligence. *ArXiv*, abs/1911.01547, 2019.

[14] Aron Barbey. Network neuroscience theory of human intelligence. *Trends in Cognitive Sciences*, 22, 10 2017.

[15] Sergey Levine. Deep rl at berkeley: Cs285. http://rail.eecs.berkeley.edu/deeprlcourse/, 2019.

[16] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

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

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[19] Nikolaus Hansen. The cma evolution strategy: A tutorial. *ArXiv*, abs/1604.00772, 2008.

[20] Yuval Tassa, Tom Erez, and Emanuel Todorov. Synthesis and stabilization of complex behaviors through online trajectory optimization. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4906–4913, 2012.

[21] Anusha Nagabandi, Kurt Konoglie, Sergey Levine, and Vikash Kumar. Deep dynamics models for learning dexterous manipulation, 2019.

[22] Richard S. Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *Intelligencesigart Bulletin*, July 1991.

[23] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration, 2016.

[24] Vladimir Feinberg, Alvin Wan, Ion Stoica, Michael I. Jordan, Joseph E. Gonzalez, and Sergey Levine. Model-based value estimation for efficient model-free reinforcement learning, 2018.

[25] Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. When to trust your model: Model-based policy optimization, 2019.

[26] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *arXiv:1807.03748v2*, January 2019.

[27] Xingyu Lu, Stas Tiomkin, and Peter Abbeel. Predictive coding for boosting deep reinforcement learning with sparse rewards. *arXiv:1912.13414v1*, December 2019.

[28] David R Ha and Jürgen Schmidhuber. World models. *ArXiv*, abs/1803.10122, 2018.

[29] Corentin Tallec, Léonard Blier, and Diviyan Kalainathan. Reproducing world models: Is training the recurrent network really needed? https://ctallec.github.io/world-models/, 2018.

[30] David Ha, Jonas Jongejan, and Ian Johnson.

[31] Danijar Hafner, Timothy P. Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *ArXiv*, abs/1912.01603, 2020.

[32] Martin Dresler, Stefan P. Koch, Renate Wehrle, Victor I. Spoormaker, Florian Holsboer, Axel Steiger, Philipp G. Sämann, Hellmuth Obrig, and Michael Czisch. Dreamed movement elicits activation in the sensorimotor cortex. *Current Biology*, 21:1833–1837, 2011.

[33] Sarah Fiona Schoch, Maren Jasmin Cordi, Michael Schredl, and Björn Rasch. The effect of dream report collection and dream incorporation on memory consolidation during sleep. *Journal of Sleep Research*, 28, 2019.

[34] Alexander Tschantz, Beren Millidge, Anil K. Seth, and Christopher L. Buckley. Reinforcement learning through active inference. *ArXiv*, abs/2002.12636, 2020.

[35] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H. Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Ryan Sepassi, George Tucker, and Henryk Michalewski. Model-based reinforcement learning for atari. *ArXiv*, abs/1903.00374, 2020.

[36] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning, 2017.

[37] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy P. Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *ArXiv*, abs/1911.08265, 2019.

[38] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

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

[40] Johan Gras. Muzero. https://github.com/johan-gras/MuZero, 2019.