STRESZCZENIE

Słowa kluczowe:

Dziedziny nauki i techniki zgodne z wymogami OECD:

ABSTRACT

Keywords:

OECD field of science and technology:

CONTENTS

LIST OF ABBREVIATIONS AND NOMENCLATURE

1. INTRODUCTION

Reinforcement learning, a subfield of artificial intelligence (AI), formalise rather the most obvious and common learning strategy among animals. It is learning how to achieve predefined goals through interaction with an environment [54]. Progress has been made in developing capable agents for numerous domains using deep neural networks in conjunction with model-free reinforcement learning [23][37][49], where raw observations directly map to agent's actions. However, current state-of-the-art approaches are very sample inefficient, which means, they sometimes require tens or even hundreds of millions of interactions with the environment [36]. Moreover, they lack the behavioural flexibility of human intelligence, hence the resulting policies poorly generalize to novel tasks in the same environment.

The other branch of reinforcement learning algorithms, called model-based reinforcement learning, aims to address these shortcomings by endowing agents with a model of its environment. There are many ways of using the model: it can be used for data augmentation for model-free methods [13], some methods use the model to simulate experience for model-free methods to learn from [18], other methods focus on simulation-based search using the model [51] and there are even methods that integrate model-free and model-based approaches into the one architecture [57]. The model allows the agent to simulate an outcome of an action taken in a given state. The main upside is, that it allows the agent to plan by thinking ahead, seeing what would happen for a range of possible choices, and explicitly deciding between possible options without the risk of the adverse consequences of trial-and-error in the real environment - including making poor, irreversible decision. Even if the model needs to be synthesized from past real experience, it can be done more sample-efficiently than model-free methods, which means without so high demand on data, because it does not require propagating rewards through Bellman backups and can exploit additional unsupervised learning signals like future observations. Furthermore, the same model can be used by the agent to complete other tasks in the same environment [57] and planning carries the promise of increasing performance just by increasing the computational budget for searching for actions [52]. It gives AI hint of human intelligence flexibility and versatility.

Model-free methods are more popular and have been more extensively developed and tested than model-based methods. While model-free methods forego the potential gains in sample efficiency from using a model, they tend to be easier to implement and tune. The main downside of model-based reinforcement learning is that a ground-truth model of the environment is usually not available to an agent. If the agent wants to use a model in this case, it has to learn the model from experience, which creates several challenges. Because the model is often only an approximation of real environment, one of them is bias in the model that can be exploited by the agent [18], resulting in an agent which performs well with respect to the model, but behaves sub-optimally in the real environment. Different challenge also comes from fundamental downside of function approximation. The performance of agents employing common planning methods usually suffer

from seemingly minor model errors [55]. Those errors compound during planning, causing more and more inaccurate predictions the further horizon of a plan [56].

There are many real-world problems that could benefit from application of general planning AI system. Company called DeepMind, driven by their experience from creating winning Go search algorithm AlphaZero [51], published AlphaFold [12], a system that predicts protein structure. The 3D models of proteins that AlphaFold generates are far more accurate than any that have come before making significant progress on one of the core challenges in biology.
Real-world applications of AI algorithms like this are often limited by the problem of sample inefficiency. In a setting with e.g. a physical robot the AI agent can not afford much trial-and-error behaviour, that could cause damage to the robot. It also can not afford running for hundreds of millions of time steps, for each task separately, in order to build a sufficiently large training dataset. Those machines work in the real world, not in an accelerated and parallelised computer simulation, and often need a human assistance. Progress in sample-efficient model-based algorithms is required in order to bring reinforcement learning into the real world applications.

The aim of this work is to derive from previous work on model learning in complex high-dimensional decision making problems [7][35][5][21] and apply them to planning in complex tasks. Those methods proved to train accurate models, at least in short horizon, and should open a path for application of planning algorithms like AlphaZero [51] to i.e. Atari 2600 games, a platform used for evaluation of general competency in artificial intelligence [36]. The goal is to improve data efficiency without loss in performance compared to model-free methods. This work focuses on three benchmarks: an arcade game with dense rewards Boxing, a challenging environment with sparse rewards Freeway and a complex puzzle game Sokoban.

2. THEORETICAL BACKGROUND

2.1. Partially Observable Markov Decision Processes

Markov Decision Processes (MDPs) are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations (states) and through those future rewards. Therefore, MDPs involve delayed rewards and the need to trade-off these with an immediate reward. Partially observable Markov Decision Processes (POMDPs), which describe a more general class of problems, have one major difference, the full state is unknown. It is perceived through observations that provide only partial information about the state. A good example are Atari games used as benchmarks in this work, described in the section 4.3.1. Individual frames often does not provide full information about the game's state which is held in the game's RAM.

In this work following definition of MDP is used: it consists of a set of hidden states S, a set of observations O and a set of actions A. The dynamics of the MDP, from any state $s \in S$ and for any action $a \in A$, are determined by transition function, $P^a_{ss'} = p(S_{t+1} = s'|S_t = s, A_t = a)$, specifying the distribution over the next state $s' \in S$. A reward function, $R^a_{ss'} = p(R_{t+1}|S_t = s, A_t = a, S_{t+1} = s')$, specifies the distribution over rewards for a given state transition. Finally, as mentioned earlier, POMDP is perceived through partial observations specified via probability distribution $P_s = p(O_t|S_t = s)$. Because a state is perceived through partial observations, the state, which contains full information, is sometimes called latent or hidden. A fixed initial state s_0 is assumed. In episodic MDPs, which this work considers, a sequence of states, actions and rewards form an episode which terminates with probability 1 in one of distinguished terminal states, $s_T \in S$, after finite number of transitions T. A return $G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$ is the total reward accumulated in that episode from time t until reaching the terminal state at time T. $0 \leqslant \gamma \leqslant 1$ is a discount factor that trade-offs short-term rewards with long-term rewards. A policy, $\pi(s, a) = p(A_T = a|S_t = s)$, maps a state s to a probability distribution over actions. A state value function, $V_\pi(s) = \mathbb{E}_\pi[G_t|s_t = s]$, is the expected return from state s when following policy π where the expectation is over the distributions of the dynamics and the policy. An action value function, $Q_\pi(s, a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a]$, is the expected return after selecting action a in state s, often called state-action pair, and then following policy π where, again, the expectation is over the distributions of the dynamics and the policy. Optimal state value and action value functions are unique value functions that maximise the value of every state or state-action pair, $V^*(s) = \max_\pi V_\pi(s), \forall_{s \in S}$ and $Q^*(s, a) = \max_\pi Q_\pi(s, a), \forall_{s \in S, a \in A}$. The two are related to each other by this equality: $V^*(s) = \max_a Q^*(s, a)$. An optimal policy $\pi^*(s, a)$ is a policy that maximises the optimal action value function for every state in the MDP, $\pi^*(s, a) = \operatorname{argmax}_a Q^*(s, a)$.

Reinforcement learning assume underlying MDP or POMDP, but the dynamics, the reward function and the observations distribution are hidden from it. Consequently, these can not be used directly for planning, but one could learn them through experience. This concept is called planning and learning in literature [54] and it is fundamental for model-based reinforcement learning.

2.2. Reinforcement Learning

Reinforcement learning (RL) is learning what to do, how to map situations to actions, so as to maximise an expected return. [54] This mapping is called a policy π. RL consists of an agent that, in order to learn a good policy, acts in an environment, sometimes referred to as a world. The environment provides a response to each agent's action a that is interpreted and fed back to the agent. The response consists of: reward r that is used as a reinforcing signal and state, or observation, s that is used to condition the agent's next decision. Fig. 21 explains it in the diagram. Each state, action, reward and next state sequence is called a step, or a transition. Multiple steps in order form a trajectory. An episode is a trajectory that starts in an initial state s_0 and finishes in a terminal state s_T. After the terminal state, the environment is reset in order to start the next episode from scratch. Very often, RL agents need dozens and dozens of episodes to gather enough experience to learn the near optimal policy. In many cases the policy is an approximation of some kind to the optimal policy and hence it will never be exactly optimal.

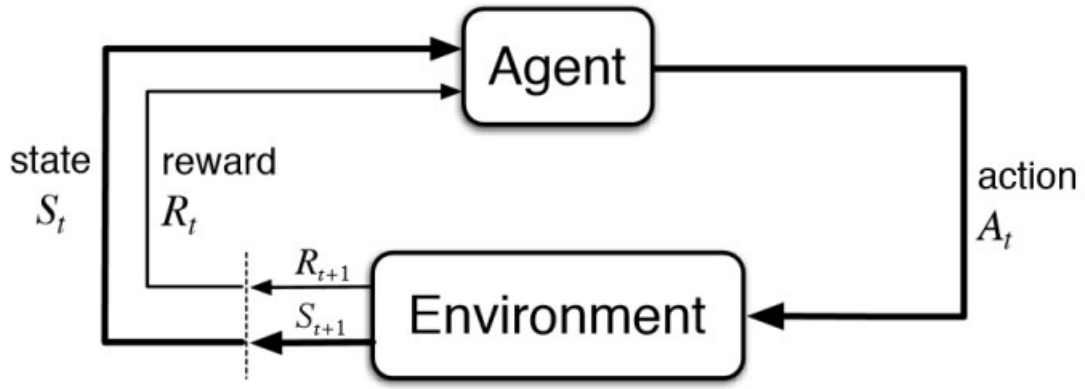


Fig. 21. Reinforcement Learning [54]

Each environment can be assigned one of two characteristics from 8 categories:

- Fully observable vs. partially observable: If an agent have access to the complete state of the environment at each point in time, then we say that the environment is fully observable. An environment might be partially observable because of noisy and inaccurate observations or because parts of the state are simply not accessible for the agent e.g. an automated taxi cannot see what other drivers are thinking.
- Single agent vs. multi-agent: The distinction between single-agent and multiagent environments may seem simple enough. For example, an agent solving a crossword puzzle by itself is clearly in a single-agent environment, whereas an agent playing chess is in a two-agent environment.
- Deterministic vs. stochastic: If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is deterministic. Otherwise, it is stochastic. If the environment is partially observable, however, then it could appear to be stochastic. Most real situations are so complex that it is impossible

to keep track of all the unobserved aspects. For practical purposes, they must be treated as stochastic. It is often said that an environment is uncertain if it is not fully observable or not deterministic, because an agent can not be sure the outcomes of its actions.

- Episodic vs. continuing: In an episodic environment, the agent's experience is divided into episodes which finish at the terminal state. Crucially, the next episode does not depend on the actions taken in the previous episode. In continuing environments, on the other hand, the agent's experience may go on and on forever and there is no a distinct terminal state.
- Static vs. dynamic: If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent. Otherwise, it is static.
- Discrete vs. continuous: The discrete/continuous distinction applies to the internal state of the environment, to the way time is handled, and to the observations and actions of the agent. For example, the chess environment has a finite number of distinct states. Chess also has a discrete set of actions. Taxi driving is a continuous-state and continuous-time problem: the speed and location of the taxi and of the other vehicles sweep through a range of continuous values and do so smoothly over time. Taxi-driving actions are also continuous: steering angles, etc.
- Known vs. unknown: Strictly speaking, this distinction refers not to the environment itself but to the agent's (or designer's) state of knowledge about the "laws of physics" of the environment. In a known environment, the outcomes, or outcome probabilities if the environment is stochastic, for all actions are given. It means that the agent have access to the environment's MDP. Obviously, if the environment is unknown, the agent will have to learn how it works in order to make good decisions.
- Dense vs. sparse rewards: If an agent observe many rewards throughout an episode, the environment is said to have dense rewards. If rewards appear rarely or only at the end of an episode, then the environment is said to have sparse rewards.

The term dynamic programming refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP). Classical dynamic programming algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense, but they are still important theoretically. Dynamic programming provides an essential foundation for the understanding of the methods presented in this thesis. In fact, all of these reinforcement learning methods can be viewed as attempts to achieve much the same effect as dynamic programming, only with less computation and without assuming a perfect model of the environment.

First question to answer is: how to compute the state value function $V(s)$ for an arbitrary policy π? The process of doing so is called policy evaluation. It is easy to note that for all states s: $V_\pi(s) = \mathbb{E}_\pi[G_t|s_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|s_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(s_{t+1})|s_t = s]$, where the expectations are over the environment dynamics and are subscripted by π to indicate that they are conditional on π being followed. The existence and uniqueness of V_π are guaranteed as long

as either $\gamma < 1$ or eventual termination is guaranteed from all states under the policy π. If the environment's dynamics are completely known, then the equation is a system of $|S|$, number of states, simultaneous linear equations in $|S|$ unknowns. In principle, its solution is a straightforward, if tedious, computation. For more practical purposes, iterative solution methods are available [54], but not described here.

Major reason for computing the value function for a policy is to help find better policies. With determined the value function V_π for an arbitrary deterministic policy π, the question now is: if for some state s changing the policy to deterministically choose an action $a \neq \pi(s)$, different then from the current policy, yields higher expected return? The value function tells how good it is to follow the current policy from s, but would it be better or worse to change to the new policy? One way to answer this question is to consider selecting a in s and thereafter following the existing policy π. The value of this way of behaving is $Q_\pi(s,a) = E[R_{t+1} + \gamma V_\pi(s_{t+1})|s_t = s, a_t = a]$, where expectation is over the environment dynamics. The key criterion is whether this is greater than or less than $V_\pi(s)$. If it is greater — that is, if it is better to select a once in s and thereafter follow π than it would be to follow π all the time — then one would expect it to be better still to select a every time s is encountered, and that the new policy would in fact be a better one overall. It is a natural extension to consider changes at all states and to all possible actions, selecting at each state the action that appears best according to $Q_\pi(s,a)$ and this way create a new, improved policy: $\pi' = \underset{a}{\text{argmax}} Q_\pi(s,a)$. The process of making a new policy that improves on an original policy, by making it greedy with respect to the value function of the original policy, is called policy improvement. If the new policy is as good as, but not better then, the old policy, then it is the optimal policy, which can be proved without much trouble [54].

Policy iteration consists of two simultaneous, interacting processes, one making the value function consistent with the current policy, policy evaluation, and the other making the policy greedy with respect to the current value function, policy improvement. In policy iteration, these two processes alternate, each completing before the other begins, but this is not really necessary. In value iteration, for example, only a single iteration of policy evaluation is performed in between each policy improvement. In asynchronous dynamic programming methods, the evaluation and improvement processes are interleaved at an even finer grain. In some cases a single state is updated in one process before returning to the other. As long as both processes continue to update all states, the ultimate result is typically the same: convergence to the optimal value function and an optimal policy.

The term generalized policy iteration is used to refer to the general idea of letting policy-evaluation and policy-improvement processes interact, independent of the granularity and other details of the two processes. Almost all reinforcement learning methods are well described as general policy iteration. That is, all have identifiable policies and value functions, with the policy always being improved with respect to the value function and the value function always being driven toward the value function for the improvement policy, as suggested by the fig.22. If both the evaluation pro-

cess and the improvement process stabilize, that is, no longer produce changes, then the value function and policy must be optimal [54].

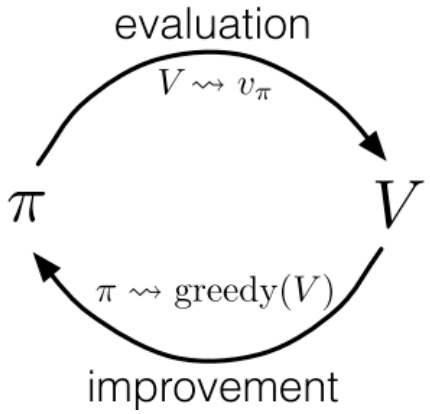


Fig. 22. General Policy Iteration [54]

Reinforcement learning faces two dilemmas. The first one is: How do you distribute credit for success among the many decisions that may have been involved in producing it? It is called credit-assignment problem: which past actions to reinforce for a positive outcome.

The second one is exploration-exploitation trade-off. To find a good policy, such that maximise an expected return, an agent needs to explore its options: if it would behave greedily all the time, it simply could not know if other actions lead to better returns. On the other hand, the agent also needs to exploit its knowledge about an environment in order to do well and progress in the environment to otherwise unavailable states. In other words, for policy evaluation to work, continual exploration needs to be assured. The most common approach to assuring that all state–action pairs are encountered is to consider only policies that are stochastic with a nonzero probability of selecting all actions in each state.

So called exploration policies are made to exploit an agent's knowledge about an environment, but at the same time, these constantly explore the environment's state-space. These policies are told to be soft, meaning that $\pi(a|s) > 0$ for every action in every state. One such family of policies is called ϵ-greedy, meaning that most of the time they choose an action that has maximal estimated action value, but with probability ϵ they instead select an action at random. That is, all non-greedy actions are given the minimal probability of selection $\frac{\epsilon}{|A(s)|}$, where $|A(s)|$ is number of legal actions in state s, and the remaining bulk of the probability, $1 - \epsilon + \frac{\epsilon}{|A(s)|}$, is given to the greedy action. The bigger the ϵ, the higher the exploration rate and vice versa.

There are two general approaches to reinforcement learning: on-policy methods and off-policy methods. On-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy different from that used to generate the data. In the former case, an agent is extra sensitive to imperfect updates, in cases when its value function or its policy are somehow approximated. Because the agent learns directly from its behaviour, if the bad update makes the agent behave worse then before, the agent effectively falls back in its learning progress. Also, the on-policy methods learn action values not

for the optimal policy, but for a near-optimal policy that still explores, which is required as noted before.

An example of on-policy algorithm would be Monte-Carlo control [54]. Because it is reinforcement learning setting an agent do not have access to underlying MDP or POMDP and needs to learn from interactions with an environment, from its experience. An obvious way to an action value function learning, or policy evaluation, is simply to play with the environment using the current policy and average the returns observed after visits to each state-action pair. It is called Monte-Carlo prediction. As more returns are observed, the average should converge to the expected value. Policy improvement is done by making the policy $\epsilon-$greedy with respect to the current value function. The two proceed according to the idea of generalized policy iteration.

In off-policy case, the behavioural exploratory policy is used for experience collection, which is then used to steadily improve the target policy towards the optimum. These methods, although favourable, are often more complex. Because the data is due to a different policy, off-policy methods are often of greater variance and are slower to converge. On the other hand, off-policy methods are more powerful and general. They include on-policy methods as the special case in which the target and behavior policies are the same. Off-policy methods also have a variety of additional uses in applications. For example, they can often be applied to learn from data generated by a conventional non-learning controller, or from a human expert. An example of off-policy algorithm would be Q-Learning [54], which is not described here.

There are two major kinds of reinforcement learning methods, model-free and model-based. Model-based methods rely on planning as their primary component, while model-free methods primarily rely on learning. The word planning is used in several different ways in different fields. In reinforcement learning the term is used to refer to any computational process that takes a model as input and produces or improves a policy for interacting with the modeled environment. Although there are real differences between these two kinds of methods, there are also great similarities. In model-free reinforcement learning, the agent samples episodes of real experience and updates its value function from real experience, this is called learning. In model-based reinforcement learning the agent samples episodes of simulated experience and updates its value function from simulated experience, now this is planning. This symmetry between learning and planning has an important consequence: algorithms for learning can also become algorithms for planning, simply by substituting simulated experience in place of real experience.

Model-free reinforcement learning, although can be used to learn effective policies for complex tasks, typically requires very large amounts of data. In fact, substantially more than a human would need to learn the same games. How can people learn so quickly? Part of the answer may be that people can learn how the game works and predict which actions will lead to desirable outcomes. Similar mechanism is used by model-based reinforcement learning.

The model of an environment can mean anything that an agent can use to predict how the environment will respond to its actions. Given a state and an action, a model produces a prediction of the resultant next state and next reward. If the model is stochastic, then there are

several possible next states and next rewards, each with some probability of occurring. Some models produce a description of all possibilities and their probabilities. These are called distribution models. Other models produce just one of the possibilities, sampled according to the probabilities. These are called sample models. The kind of model assumed in dynamic programming, estimates of the MDP's dynamics $P^a_{ss'}$ and $R^a_{ss'}$, is a distribution model.

Models can be used to mimic or simulate experience. Given a starting state and action, a sample model produces a possible transition, and a distribution model generates all possible transitions weighted by their probabilities of occurring. Given a starting state and a policy, a sample model could produce an entire episode, and a distribution model could generate all possible episodes and their probabilities. In either case, it is said that the model is used to simulate the environment or rollout the trajectory and produce simulated experience. Because these models simulate the environment dynamics in order to generate transitions, they are also called dynamics models or transition models.

In artificial intelligence, there are two distinct approaches to planning according to the definition presented. State-space planning, which is viewed primarily as a search through the state-space for an optimal policy or an optimal path to a goal. Actions cause transitions from state to state, and value functions are computed over states.

In what is called plan-space planning, planning is instead a search through the space of plans. Operators transform one plan into another, and value functions, if any, are defined over the space of plans. Plan-space planning includes evolutionary methods.

Within a planning agent, there are at least two roles for real experience: it can be used to improve the transition model, to make it more accurately match the real environment, and it can be used to directly improve the value function and policy using the kinds of model-free reinforcement learning. The former is called model learning, and the latter is called direct reinforcement learning. The possible relationships between experience, model, values, and policy are summarized in the fig.23. It can be seen, that experience can improve value functions and policies either directly or indirectly via the transition model. This way, real experience is better utilized by using it not only once for direct reinforcement learning, but also reusing it in the future through the model. This result in better sample-efficiency over model-free methods which utilize only the direct learning path.

Planning is often conducted in small, incremental steps. This enables planning to be interrupted at any time, which appears to be a key requirement for efficiently intermixing planning with acting and with learning of the model. Planning in very small steps may be the most efficient approach even in pure planning problems if the problem is too large to be solved exactly or the agent can not wait for exact solution and needs to act based on approximated one.

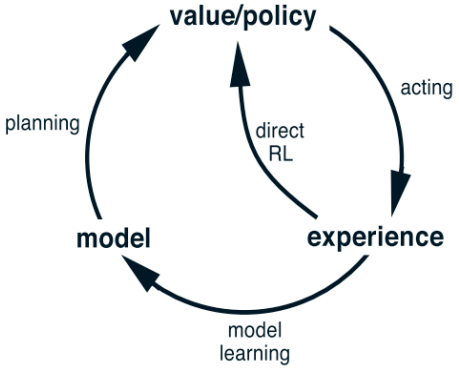


Fig. 23. Model-based Reinforcement Learning. [54] Each arrow shows a relationship of influence and presumed improvement.

2.3. Simulation-Based Search

Simulation-based search is something called planning at decision time [54] and can be used in model-based reinforcement learning setting. It requires a sample model of the MDP, which can sample state transitions and rewards from $P^a_{ss'}$ and $R^a_{ss'}$ respectively. However, it is not necessary to know these probability distributions explicitly. The next state and reward could be generated by a black box simulator. The efficiency and effectiveness of simulation-based search depends in large part on the performance and accuracy of the model. The model learning problem is described later. Now, access to a perfect sample model is assumed.

Simulation-based search algorithms sample experience in sequential episodes. Each simulation begins in a root state s_0. At each step u of simulation, an action a_u is selected according to a simulation policy, and a new state s_{u+1} and reward r_{u+1} is generated by the sample model. This process repeats, without backtracking, until a terminal state is reached. The values of states or actions are then updated from the simulated experience.

Simulation-based search is usually applied online, at every time-step t, by initiating a new search that starts from the current state $s_0 = s_t$. The distribution of simulations then represents a probability distribution over future experience from time-step t onwards. Simulation-based search exploits temporality, in other words focuses on the current situation, by learning from this specific distribution of future experience, rather than learning from the distribution of all possible experience. Furthermore, as the agent's policy improves, the future experience distribution will become more refined. It can focus its value function on what is likely to happen next, given the improved policy, rather than learning about all possible eventualities.

Monte-Carlo simulation is a very simple simulation-based search algorithm for evaluating candidate actions from a root state s_0. The search proceeds by simulating complete episodes from s_0 until termination, using a fixed simulation policy. The action values $Q(s_0, a)$ are estimated by the mean outcome of all simulations with candidate action a.

Monte-Carlo tree search (MCTS) is perhaps the best-known example of a simulation-based search algorithm. It makes use of Monte-Carlo simulation to evaluate the nodes of a search

tree. There is one node, $n(s)$, corresponding to every state s encountered during simulation. Each node contains a total count of visits of the state, $N(s)$, a value $Q(s,a)$ and a visit count $N(s,a)$ for every possible action in a state s. Simulations start from the root state s_0, and are divided into two stages. When state s_u is contained in the search tree, a tree policy selects the action with the highest value $Q(s,a)$. Otherwise, a random default policy is used to roll out simulations to completion. After each simulation, $s_0, a_0, r_1, s_1, a_1, ..., r_T$, each node $n(s_u)$ in the search tree is updated incrementally to maintain the count and mean return from that node,

$$N(s_u) \leftarrow N(s_u) + 1$$

$$N(s_u, a_u) \leftarrow N(s_u, a_u) + 1$$

$$Q(s_u, a_u) \leftarrow Q(s_u, a_u) + \frac{G_u - Q(s_u, a_u)}{N(s_u, a_u)}$$

2.4. Deep Learning

Machine learning gives AI systems the ability to acquire their own knowledge, by extracting patterns from raw data. It stands in opposition to classical computer programs which execute explicit instructions hand-coded by a programmer. One example of machine learning algorithm is logistic regression. It can determine whether to recommend cesarean delivery or not [39]. Logistic regression is an example of a linear model. Such a model takes a vector as input and then calculates dot product of it with its weights.
Another widely used machine learning algorithm called naive Bayes can distinguish between legitimate and spam e-mail.

The performance of these machine learning algorithms depends heavily on the representation of the problem they are given. For example, when logistic regression is used to recommend cesarean delivery, the AI system does not examine the patient's MRI scan directly. It would not be able to make useful predictions as individual pixels in an MRI scan have negligible correlation with any complications that might occur during delivery. It, instead, gets several pieces of relevant information, such as the presence or absence of a uterine scar, from the doctor. Each piece of information included in the representation of the data is known as a feature. Logistic regression learns the relation between those features and various outcomes, such as a recommendation of cesarean delivery. The algorithm does not influence the way that the features are defined in any way.

Sometimes it can be hard to hand-craft a good problem's representation. For example, suppose that someone would like to write a program to detect cats in photographs. People know that cats are furry and have whiskers, so they might like to use the presence of a fur and whiskers as features. Unfortunately, it is difficult to describe exactly what a fur or a whisker looks like in terms of pixel values. This gets even more complicated when taking into account e.g. shadows falling on the cat or an object in the foreground obscuring part of the animal. One solution to this problem is to use machine learning to discover not only the mapping from representation to output

but also the representation itself. This approach is known as representation learning. Learned representations often result in much better performance of machine learning algorithms than can be obtained with hand-designed representations. They also allow AI systems to rapidly adapt to new tasks with minimal human intervention.

Deep learning is a particular kind of machine learning that achieves great power and flexibility by learning to represent the world as a nested hierarchy of concepts, with each concept defined in relation to simpler concepts. Deep learning solves representation learning problem by introducing representations that are expressed in terms of other, simpler representations. Fig. 24 shows how a deep learning system can represent the concept of an image of a person by combining simpler concepts, such as corners and contours, which are in turn defined in terms of edges.

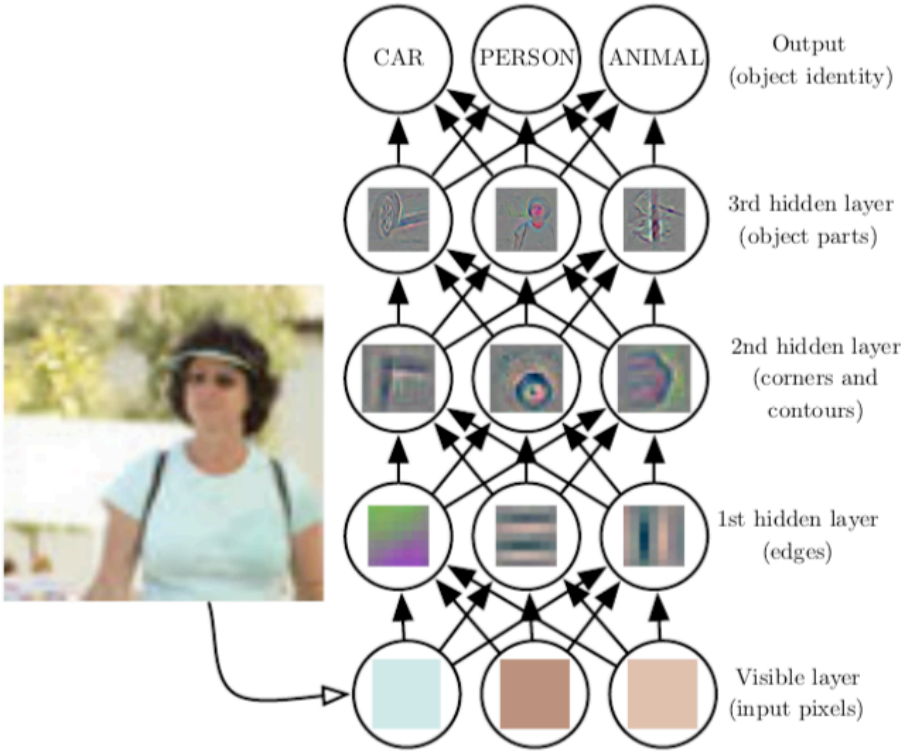


Fig. 24. Deep Learning [14]

The fundamental example of a deep learning model is a fully-connected (FC) neural network (NN), sometimes called a multilayer perceptron (MLP). A MLP is just a mathematical function mapping some set of input values to output values. The function is formed by composing many simpler functions, called neurons or perceptrons, into layers. Each layer provides a new representation of its input, which might be a vector, to all the neurons in the next layer by multiplying the input element-wise with its parameters and applying some kind of non-linear activation function to the sum of these products e.g. sigmoid function. Other common activation function, used especially at the output layer of the model, is softmax function which normalizes the output vector. Without activation functions, the multilayer NN would just simplify to a linear mapping without ability to model non-linear relationships of inputs and outputs of the NN. Fig. 25 shows example

MLP and dependencies between perceptrons. The input is presented at the input layer. Then a hidden layer (or series of them) extracts increasingly abstract features from the image. These layers are called “hidden” because their values are not given in the data. Instead, the model must determine which concepts are useful for explaining the relationships in the observed data. Finally, this description of the input in terms of the features can be used to produce the output at the output layer.

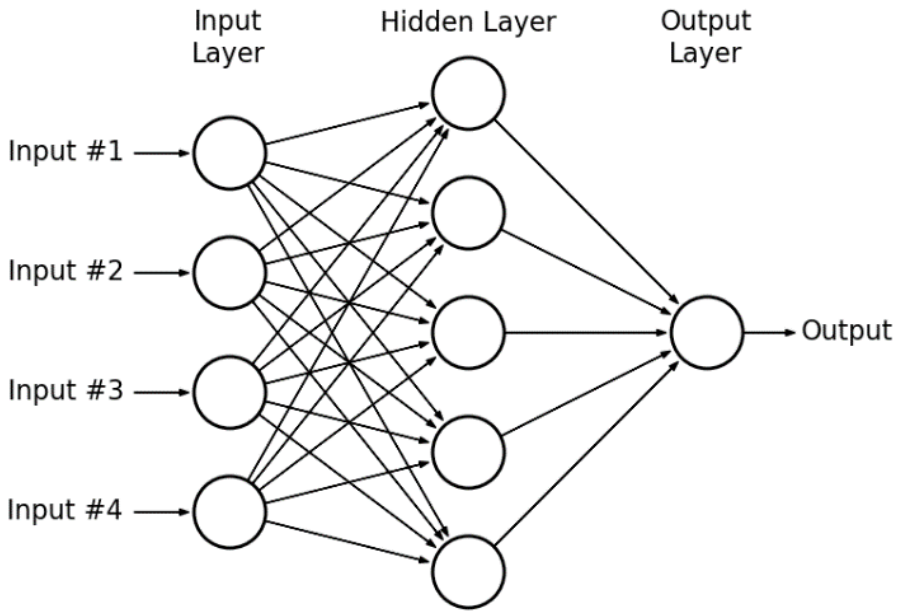


Fig. 25. Multilayer perceptron

Other type of very common neural network is a recurrent neural network (RNN). Humans do not start their thinking from scratch every second. As they read this thesis for example, they understand each word based on their understanding of previous words. Their thoughts have persistence. MLPs can not do this, that is why RNNs address this issue. These are networks with loops in them, allowing information to persist. In the fig. 26, a chunk of neural network, A, looks at some input x_t and outputs a value h_t. A loop allows information to be passed from one step of the network to the next. It turns out that RNNs are not all that different than a normal feedforward neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor.

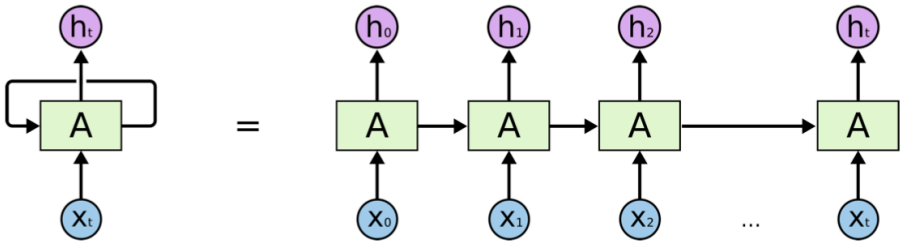


Fig. 26. A recurrent neural network [46]

Yet another very useful NN architecture is a convolutional neural network (CNN) which makes use of the fact that inputs to a NN can often have similar patterns in different parts of the input. For example in images, the object can appear in different positions in the image, but it is still the same object. CNNs exploit that fact by grouping parameters info filters in each layer and slide, or more precisely convolve, each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. The 2D outputs of all filters are stacked together to form a 3D activation map. It is then processed by the next, similar, layer with its own set of filters. At the end of CNN, the final 3D volume is often flattened into a vector and then further processed by a fully-connected NN. The example CNN is shown in fig. 27.

A CNN is able to successfully capture the spatial dependencies in an image through the application of relevant filters. The architecture performs a better fitting to the data with spatial hierarchy due to the reduction in the number of parameters involved and reusability of weights.

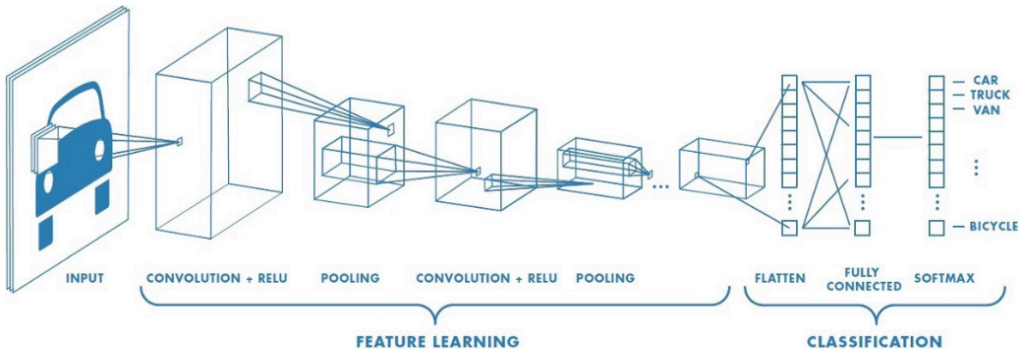


Fig. 27. A convolutional neural network [45]. ReLU [41] and Softmax are activation functions. Pooling means that each filter's output is divided into, for instance, 2 x 2 regions and only maximum value from each region is passed to the next layer.

A neural network is trained, in the simplest case, to output a correct target value for each input from a dataset. This is done by backpropagating some measure of error, how much different is the NN's output value from the target value, through the NN. This produces gradients. Gradients are vectors that point in the direction of steepest ascent of the NN's error. These are used in the opposite direction to iteratively change parameters and lower the error on the dataset. The process is repeated iteratively, updates are small steps, because gradients are first order derivatives and the directions these point in are accurate only locally. This algorithm is called gradient decent (GD) and it is at the heart of the deep learning [14].

Most common measures of error, called losses, are:

- Mean Squared Error (MSE): $\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$
- Cross-entropy: $-\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{M} y_{i,c} log(\hat{y}_{i,c})$

where N is number of training examples in the dataset, M is a number of possible classes of the target, y is ground truth target from the dataset and $\hat{y}$ is predicted response by the model. The MSE is most commonly used in a regression setting, where y is a single scalar for each data point. The cross-entropy is mostly used for a classification task, where the ground truth target and

the predicted response are vectors interpreted as probabilities of the input being each class, and hence the output must be normalized.

2.5. Generative models

Generative modeling is a broad area of machine learning which deals with models of distributions $p(X)$, defined over datapoints X in some potentially high-dimensional space. For instance, images are a popular kind of data for which one might create generative models. Each datapoint, which is an image, has thousands or millions of dimensions, in form of pixels, and the generative model's job is to somehow capture the dependencies between pixels. For example, that nearby pixels have similar color, and are organized into objects. Then, one often cares about producing more examples, that are like those already in a database, using the trained generative model.
When training a generative model, the more complicated the dependencies between the dimensions, the more difficult the models are to train. For instance, the problem of generating images of handwritten digits from 0 to 9. If the left half of the character contains the left half of a 5, then the right half cannot contain the left half of a 0, or the character will very clearly not look like any real digit. Intuitively, it helps if the model first decides which character to generate before it assigns a value to any specific pixel. This kind of decision is formally called a latent variable or a code. That is, before the model draws anything, it first randomly samples a digit value z from the set $[0, ..., 9]$, and then makes sure all the strokes match that character. It is called latent, because it is not observed, or included in the dataset, and it must be inferred in order to discover it.
To make this notion precise mathematically, the aim is to maximise the probability of each datapoint X in the training set under the entire generative process according to: $p(X) = \int p(X|z)p(z)dz$, where $p(X|z)$ is a likelihood distribution, which generates images from latent variables, and $p(z)$ is a prior distribution, which decides on latent variables. Calculating $p(X)$ for most interesting, non-linear, models is computationally intractable. One way to bypass this is to use variational inference.

From now on, POMDP notation will be used: datapoints are now observations o and latent variables are now states s. Variational inference can be used to train latent variable models by maximizing a lower bound of the marginal probability density $p(o) = \int p(o|s)p(s)ds$. The term $p(o|s)$ is the likelihood of the data given the latent variables s and, in the case of neural networks, takes the form of a parameterised model sometimes called a decoder. Instead of dealing with the prior $p(s)$ directly, variational autoencoders (VAEs) infer $p(s)$ using the posterior $p(s|o)$ [29]. As the true form of the posterior distribution $p(s|o)$ is unknown, variational inference turns the problem of inferring latent variables into an optimisation problem by approximating the true posterior distribution with a variational distribution $q(s|o)$, called an encoder, which takes the form of a simpler distribution such as a fully factorized Gaussian, meaning the latent state vector dimensions are independent from each other, parameterised by a neural network and then minimising the Kullback-Leibler (KL) divergence between $q(s|o)$ and $p(s)$, which can be interpreted as distance between two distributions. As the KL divergence is nonnegative and minimised when q is the same

as p, the training objective for VAEs is known as the variational or evidence lower bound (ELBO) on the data log-likelihood:

$$lnp(o) \geqslant \mathbb{E}_{q(s|o)}\left[lnp(o|s)\right] - D_{KL}[q(s|o)||p(s)]$$

where $p(s)$ is very often a standard normal distribution to simplify calculations and because no prior knowledge about latent variables is assumed. The left term of ELBO is called reconstruction loss, as it makes an observation o more probable given its latent variable s. The right term of ELBO is called KL penalty, or regularisation loss, or complexity or information bottleneck, as it pulls the approximate posterior q closer to the uninformed prior p and ensures this way that it explains the observation in the simplest possible way, preventing overfitting and aiding generalisation. VAEs parameters are trained using stochastic gradient descent, as common in the deep learning setting, to maximise the variational bound and this way indirectly maximise the probability of the dataset under the generative model.

There is one useful dependence between a decoder variance and scaling a KL divergence loss. Lowering the decoder variance is equivalent to lowering the KL divergence scale. It can be seen by writing the ELBO for the Gaussian decoder in the standard form $\mathbb{E}_{q(s|o)}\left[lnp(o|s)\right] - D_{KL}[q(s|o)||p(s)]$. For the Gaussian decoder the reconstruction loss is $lnp(o|s) = -0.5(o - f(s))/\sigma^2 - lnZ$, where Z is a normalizing constant, f is a model which parametrize the conditional decoder and σ^2 is the decoder variance. Multiplying the ELBO by σ^2 removes it from the log-probability term and puts it in front of the KL term as in beta-VAE [24], $\mathbb{E}_{q(s|o)}\left[lnp'(o|s)\right] - \sigma^2 D_{KL}[q(s|o)||p(s)]$. The objectives have different values because of the Gaussian normalizer Z but they share the same gradient since the normalizer is a constant.

Besides scaling a KL divergence term, other technique called free nats [6] is often used for Variational Autoencoders. ELBO with free nats looks like this: $\mathbb{E}_{q(s|o)}\left[lnp(o|s)\right] - max\big(\lambda, D_{KL}[q(s|o)||p(s)]\big)$, where λ is free nats threshold. The model is allowed to use given amount of nats, up to the threshold, without KL penalty in the variational objective. It helps the model focus on smaller details which do not contribute much to improving the reconstruction loss. Intuitively, to this threshold of KL penalty the reconstruction loss is favoured.

Because the POMDP experience forms trajectories, where future states depend on previous states and actions, something called structured variational inference is used for model learning. The aim of this method is to train a model that accurately simulate the original environment from which the data in form of transitions was gathered. The simplest data collection approach is to use an agent that randomly explores the environment.

Since the random agent may not initially visit all parts of the environment, new experience needs to be iteratively collect to refine the model. It is called data-aggregation iterative method and it is often done by planning with the partially trained model.

The joint probability of states s, observations o, and rewards r conditioned on the actions a from the initial timestep $t = 1$ to the end of a trajectory of length T is:

$$p(o_{1:T}, r_{1:T}, s_{1:T}|a_{1:T}) = p(o_1|s_1)p(s_1)\prod_{t=2}^{T} p(o_t|s_t)p(r_t|s_{t-1}, a_{t-1})p(s_t|s_{t-1}, a_{t-1})$$

where, besides the prior for the initial state $p(s_t)$, a transition model prior $p(s_t|s_{t-1}, a_{t-1})$ is introduced.

Since the model is non-linear, it is not possible to directly compute the state posteriors that are needed for parameter learning. Instead, an encoder $q(s_{1:T}|o_{1:T}, a_{1:T}) = \prod_{t=1}^{T} q(s_t|s_{t-1}, a_{t-1}, o_t)$ is used to infer approximate state posteriors from past observations and actions, where $q(s_t|s_{t-1}, a_{t-1}, o_t)$ could be a diagonal Gaussian with mean and variance parameterised by a neural network. Here, the filtering posterior [34] is used that conditions on past observations since, at the end, the model is designed to simulate future states and rewards based on past observations and actions. One may also use the full smoothing posterior during training [16], where the posterior for a single latent variable s_t depends on all future observations, rewards and actions too, but it could not be used in RL setting where at each timestep the future is unknown.

Using the encoder, a variational bound on the data log-likelihood can be constructed. For simplicity, here only losses for predicting the observations are written, but the reward losses follow by analogy:

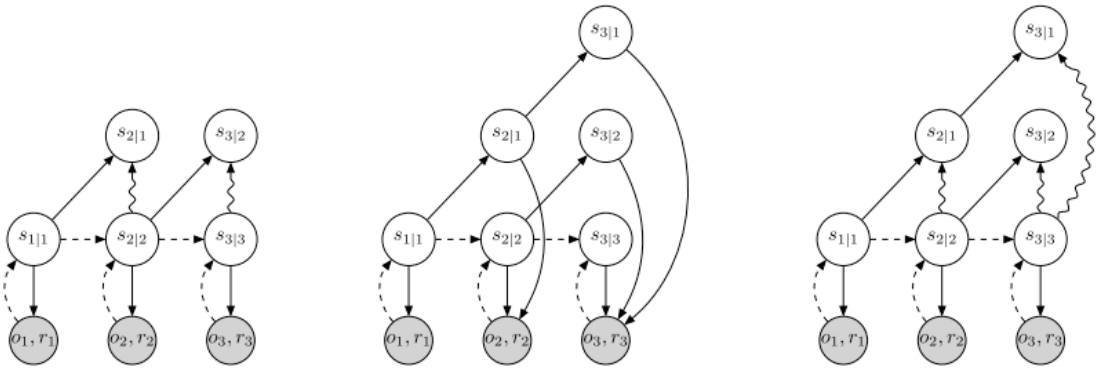
$$lnp(o_{1:T}|a_{1:T}) \geqslant \sum_{t=1}^{T} \left(\mathbb{E}_{q(s_t|o_{\leqslant t}, a_{<t})} \left[lnp(o_t|s_t) \right] - \mathbb{E}_{q(s_{t-1}|o_{\leqslant t-1}, a_{<t-1})} \left[D_{KL}[q(s_t|o_{\leqslant t}, a_{<t})||p(s_t|s_{t-1}, a_{t-1})] \right] \right)$$

Here, the left term of ELBO is still a reconstruction loss, but now the right term of ELBO trains the approximate posterior and the transition model to be consistent with each other.

Typical variational bound for learning and inference in latent sequence models, described above and shown in fig. 28a, contains reconstruction terms for the observations and KL-divergence regularizers for the approximate posteriors. A limitation of this objective is that the transition model $p(s_t|s_{t-1}, a_{t-1})$ is only trained via the KL-divergence for one-step predictions: the gradient flows through the transition model, $p(s_t|s_{t-1}, a_{t-1})$, directly into the approximate posterior, $q(s_t|o_{\leqslant t}, a_{<t})$, but never traverses a chain of multiple predicted transitions. If one could train a model to make perfect one-step predictions, it would also make perfect multi-step predictions, so this would not be a problem. However, when using a model with limited capacity and restricted distributional family, training the model only on one-step predictions until convergence does in general not coincide with the model that is best at multi-step predictions. For successful planning, accurate multi-step predictions are needed.

The authors of [21] generalize the standard variational bound to latent overshooting, as show in fig. 28c, which trains all multi-step predictions in latent space. Using only terms in latent space results in a fast regularizer that can improve long-term predictions by encouraging consistency

between one-step and multi-step predictions. It is compatible with any latent sequence model.



(a) Standard variational bound (b) Observation overshooting (c) Latent overshooting

Fig. 28. Unrolling schemes [21]. The labels $s_{i|j}$ are short for the state at time i conditioned on observations up to time j. Arrows pointing at shaded circles indicate log-likelihood loss terms. Wavy arrows indicate KL-divergence loss terms. (a) The standard variational objectives decodes the posterior at every step to compute the reconstruction loss. It also places a KL on the prior and posterior at every step, which trains the transition function for one-step predictions. (b) Observation overshooting decodes all multi-step predictions to apply additional reconstruction losses. This is typically too expensive in image domains. (c) Latent overshooting predicts all multi-step priors. These state beliefs are trained towards their corresponding posteriors in latent space to encourage accurate multi-step predictions.

Structured variational inference rely heavily on approximation techniques. Because the final trained model is an approximation of the environment, it inducts bias that can be exploited by the agent, resulting in an agent which performs well with respect to the model, but behaves sub-optimally in the real environment [18]. The approximated model can also introduce minor errors at each timestep which compound over time. The performance of planning agents can significantly suffers from these model errors [55]. Moreover, more and more inaccurate predictions over time can effectively preclude use of long, tens or hundreds of time steps, planning horizons [30].

2.6. World models

A key challenge in model-based reinforcement learning is learning accurate, computationally efficient models of complex domains and using them to solve RL problems. Recent solution [5] to this problem are computationally efficient state-space environment models that make predictions at a higher level of abstraction, both spatially and temporally, than at the level of raw pixel observations. Such models substantially reduce the amount of computation required to make predictions, as future states can be represented much more compactly. Moreover, in order to increase model accuracy, they exploit the benefits of explicitly modeling uncertainty in state transitions.

In the fig. 29, there are different environment models described. The models, p, are learned in an unsupervised way from observations, o, conditioned on actions, a. In particular, the focus is on how fast and accurately these models can predict, at time step t, some future statistics $x_{t+1:t+\tau}$, e.g. an environment's rewards, over a horizon τ by simulating a trajectory given an arbitrary sequence of actions $a_{t:t+\tau-1}$, that can later be used for decision making.

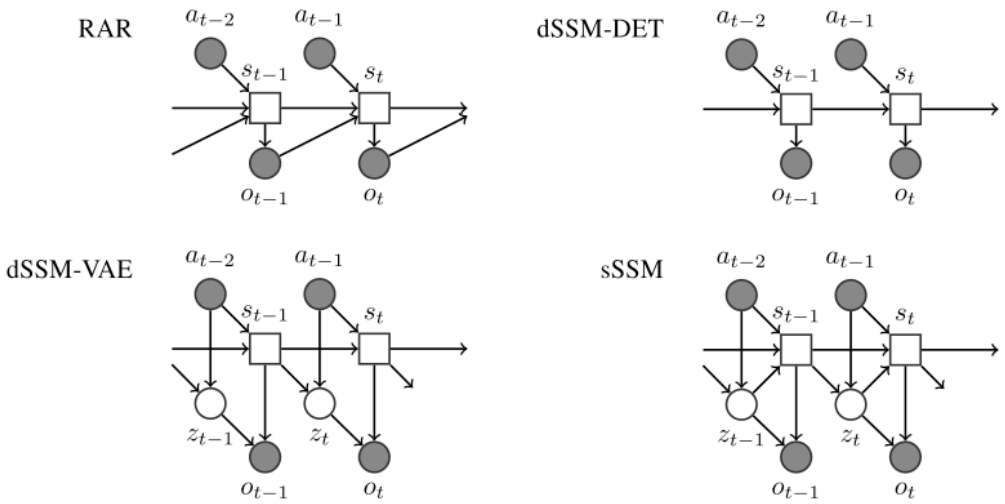


Fig. 29. The graphical models representing the architectures of different environment models. Boxes are deterministic nodes, circles are random variables and filled circles represent variables observed during training [5].

A straight-forward choice is the family of temporally auto-regressive models (AR) over the observations, $o_{t+1:t+\tau}$. If these use a recurrent mapping that recursively updates sufficient statistics $s_t = f(s_{t-1}, a_{t-1}, o_{t-1})$, therefore reusing the previously computed statistics s_{t-1}, and use them to predict future observations, $p(o_{t+1:t+\tau}|o_{\leqslant t}, a_{<t+\tau}) = \prod_{r=t+1}^{t+\tau} p(o_r|f(s_{r-1}, a_{r-1}, o_{r-1}))$, then these models are called recurrent auto-regressive models (RAR). If f is parameterised as a neural network, RARs are equivalent to recurrent neural networks. Although faster then auto-regressive models that don't reuse statistics from previous time steps, these are still expect to be slow when simulating trajectories, as they still need to explicitly render observations $o_{t+1:t+\tau}$ in order to make any predictions, $x_{t+1:t+\tau}$.

State-space models (SSMs) circumvent this by positing that there is a compact state representation s_t that captures all essential aspects of the environment on an abstract level. It is assumed that s_{t+1} can be predicted from the previous state s_t and action a_t alone, without the help of previous pixels $o_{\leqslant t}$, $p(s_{t+1}|s_{\leqslant t}, a_{<t}, o_{\leqslant t}) = p(s_{t+1}|s_t, a_t)$. Furthermore, it is assumed that s_t is sufficient to predict o_t, i.e. $p(o_t|s_{\leqslant t}, a_{<t}) = p(o_t|s_t)$. This modelling choice implies that the latent states are, by construction, sufficient to generate any future predictions $x_{t+1:t+\tau}$. Hence, the model never has to directly sample costly high-dimensional pixel observations.

There are two flavors of SSMs: deterministic SSMs (dSSMs) and stochastic SSMs (sSSMs). For dSSMs, the latent transition $s_{t+1} = f(s_t, a_t)$ is a deterministic function of the past, whereas for sSSMs, they consider transition distributions $p(s_{t+1}|s_t, a_t)$ that explicitly model uncertainty over the next state. The sSSMs are parameterised by introducing for every t a latent variable z_t whose distribution depends on s_{t-1} and a_{t-1}, and by making the state a deterministic function of the past state, action, and latent variable: $z_{t+1} \sim p(z_{t+1}|s_t, a_t)$, $s_{t+1} = f(s_t, a_t, z_{t+1})$.

The observation model, also called decoder, computes the conditional distribution $p(o_t|\cdot)$. It either takes as input the state s_t (deterministic decoder), or the state s_t and latent z_t (stochastic

decoder). sSSMs always use the stochastic decoder. dSSMs can use either the deterministic decoder (dSSM-DET), or the stochastic decoder (dSSM-VAE). The latter can capture joint uncertainty over pixels in a given observation o_t, but not across time steps. The former is a fully deterministic model, incapable of modeling joint uncertainties (in time or in space).

The paper [5] provides the first comparison of deterministic and stochastic, pixel-space and state-space models w.r.t. speed and accuracy, applied to challenging environments from the Arcade Learning Environment [2]. Specifically, the prediction of dSSM-DET exhibits "sprite splitting", or layering of multiple possible future observations, at corridors of MS-PACMAN [58], whereas multiple samples from the sSSM show that the model has a reasonable and consistent representation of uncertainty in this situation. Moreover, SSMs, which avoid computing pixel renderings at each rollout step, exhibit a speedup of more then 5 times over the standard AR model. The computational speed-up of state-space models, while maintaining high accuracy, makes their application in RL feasible.

3. STATE OF THE ART

Quite surprisingly, there is not much work on model-based planning and learning from high-dimensional observations, like images, in complex environments, like video games. This chapter review related work that helps arrive at the final solution of this thesis problem of planning in imagination.

Planning is a natural and powerful approach to decision making problems with known dynamics. For instance, Monte-Carlo Tree Search methods [33] have been used for complex search problems, such as the game of Go [52]. Moreover, planning carries the promise of increasing performance just by increasing the computational budget for searching for actions. The first paper below describes the current state of the art search method and the other one explores the idea of Monte-Carlo planning with a learned model.

3.1. World Models

In World Models [18] paper, a simple model inspired by human cognitive system is used. The authors explore the idea of using large and highly expressive neural networks to reinforcement learning. The RL algorithm is often bottlenecked by the credit assignment problem, which makes it hard for traditional RL algorithms to learn millions of weights of a large model.
To avoid this problem, the authors decompose the agent training into two stages: a world model and Controller. The world model, composed of Vision and Memory modules, learns rich spatial and temporal representation of an agent's environment. Thereafter, by using the compressed representation extracted from the world model as inputs to an agent, they train a linear model to perform a task in the environment. The small linear model lets the training algorithm focus on the credit assignment problem on a small search space, while not sacrificing capacity and expressiveness via the larger world model.

In conclusion, their solution consists of three components: Vision for encoding the spatial information, Memory for encoding the temporal information and Controller which represents the agent's policy. Fig. 31 depicts a flow diagram of the agent's model.

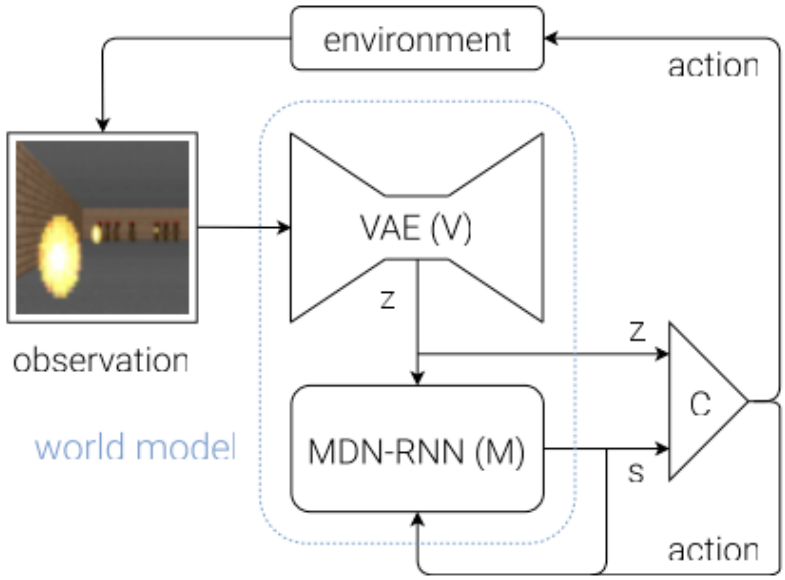


Fig. 31. Flow diagram of the agent's model [18]. The raw observation is first processed by the Vision (V) at each time step t to produce a latent variable z_t. The input into the Controller (C) is the latent variable z_t concatenated with the Memory (M) hidden state s_t at each time step. The Controller will then output an action vector a_t and will affect the environment and produce the next observation. The Memory will then take the current z_t and action a_t as an input to update its own hidden state to produce s_{t+1} to be used at time step $t + 1$.

This architecture, although quite changed, still reassembles sSSM from the section 2.6. It learns the Memory module in the latent space created by the Vision module. It also explicitly models transitions uncertainty, which will prove to be the key for successful Controller training. What is different though, it predicts the next latent variable conditioned on the next hidden state, $z_{t+1} \sim p(z_{t+1}|s_{t+1})$. The Memory module calculates its next hidden state based on the present hidden state, action and latent variable, $s_{t+1} = f(s_t, a_t, z_t)$. This way, the Memory module is less a sample transition model that predicts one future from all possible futures and more a memory which deterministically encodes facts about the past in the hidden state, which should fully describe the present latent state of the environment, and then use it to sample a latent variable.

The Controller model represents the agent's policy. It is responsible for determining course of actions to take in order to solve a given task. Controller is a simple linear model that maps the concatenated latent variable z_t and hidden state s_t at the time step t directly to the action a_t at that time step.

The authors deliberately made Controller as simple as possible, and trained it separately from Vision and Memory, so that most of the agent's complexity resides in the world model (Vision and Memory). The latter can take the advantage of current advances in deep learning that provide tools to train large models efficiently when well-behaved and differentiable loss function can be defined. This shift in the agent's complexity towards the world model, as already mentioned, allows the Controller model to stay small and focus its training on tackling the credit assignment problem in challenging RL tasks. It is trained using evolution strategy, which is rather an unconventional

choice that only currently have been considered as a viable alternative to popular RL techniques [47].

Their solution is able to solve an OpenAI Gym's CarRacing environment [4], which is the continuous-control racing task, with top-down view on the car and the track. It is the first known solution to achieve the score required to solve this task. Nonetheless, because the Controller uses real experience for training counted in millions of episodes, they did not improve sample-efficiency compared to other model-free solutions [27]. But, what is really interesting, in the process of training the Memory learns to simulate the original environment. The authors show that the learned Controller can function inside of the imagined environment of CarRacing, that is, simulated by the Memory. In the second experiment, they show that the agent can not only play in imagination, but also it is able to learn solely from imagined experience, produced by its Memory, and successfully transfer this policy back to the actual environment of VizDoom (see fig. 32). In the first trials, the Controller learned to exploit imperfect simulations of the Model, which only approximates the true environment dynamics. To mitigate this behaviour, they adjust the temperature parameter [17] of Memory to control the amount of randomness in the simulation, hence controlling the trade-off between realism and exploitability. The Controller learns from simulated experience, which means that only tens of thousands of episodes from the environment are needed for the training of Vision and Memory. Assuming that each episode consists of hundreds of frames, millions of frames in total are required for training. It makes World Models very sample efficient compared to other state-of-the-art model-free methods that require even two orders of magnitude more data [37].

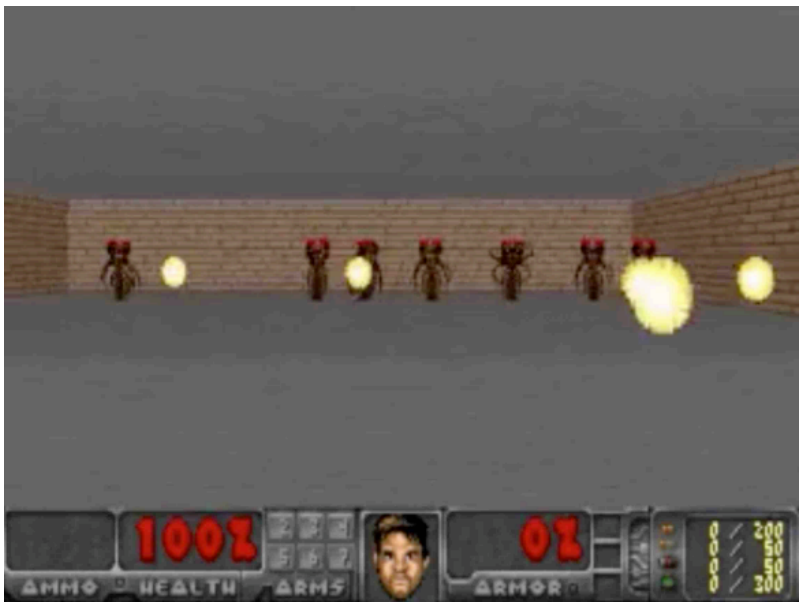


Fig. 32. VizDoom: the agent must learn to avoid fireballs shot by monsters from the other side of the room with the sole intent of killing the agent [18].

The authors results indicate that their world model is able to model complex environments from visual observations and it can be used for planning. Therefore, it may prove useful for the topic of this thesis.

3.2. Learning Latent Dynamics for Planning from Pixels

In [21], the authors propose the Deep Planning Network (PlaNet), a purely model-based agent that learns the environment dynamics from images and chooses actions through fast online planning in latent space. To achieve high performance, the dynamics model must accurately predict the rewards ahead for multiple time steps. This approach uses a latent dynamics model for its fast querying capabilities, described earlier in the section 2.6. Moreover, they propose a multi-step variational inference objective named latent overshooting already described in the section 2.5. The authors found that several dynamics models benefit from latent overshooting, although their final agent, which use the RSSM model described below, does not require it.

Despite its generality, the purely stochastic transitions make it difficult for the transition model to reliably remember information for multiple time steps. In theory, this model could learn to set the variance to zero for some state components, but the optimization procedure may not find this solution. This motivates including a deterministic sequence of activation vectors $\{s_t\}_{t=1}^{T}$ that allow the model to access not just the last latent variable, z_{t-1}, but all previous states deterministically. The authors use such a model, shown in fig. 33, that they name recurrent state-space model (RSSM). Intuitively, one can understand this model as splitting the state into a stochastic part z_t and a deterministic part s_t, which depend on the stochastic and deterministic parts at the previous time step.

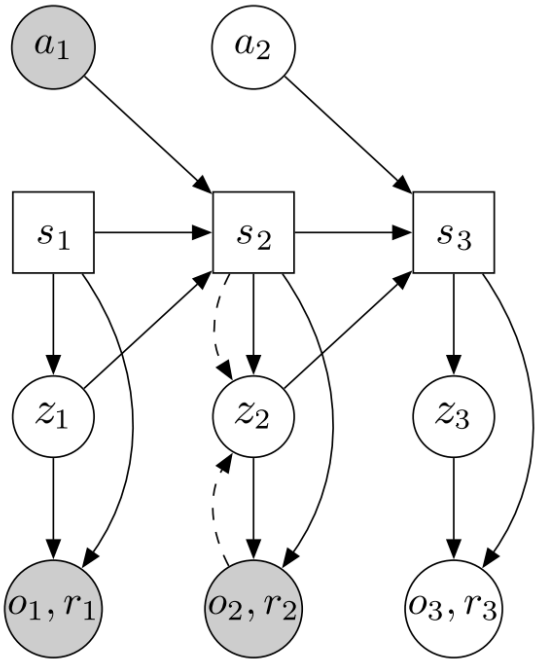


Fig. 33. Latent dynamics model design [21]. In this example, the model observes the first two time steps and predicts the third. Circles represent stochastic variables and squares deterministic variables. Solid lines denote the generative process and dashed lines the inference model. This model splits the latent state into stochastic and deterministic parts, allowing the model to robustly learn to predict multiple futures.

Similar to sSSM from the section 2.6, it learns a transition model in latent space for efficiency. This latent dynamics model is designed with both deterministic and stochastic compo-

nents [5]. Original experiments indicate having both components to be crucial for high planning performance. Like in World Models, it condition a latent variable only on a deterministic state. It calculates the next deterministic state based on the present deterministic state, action and latent variable, $s_{t+1} = f(s_t, a_t, z_t)$ and samples the next latent variable based on the next state, $z_{t+1} \sim p(z_{t+1}|s_{t+1})$. It can be viewed that stochastic part of the state is determined by deterministic part of the state at any time step. This is reversed relationship relative to sSSM and it helps preserve a deterministic part of the state for multiple time steps and this way further improve the simulation accuracy.

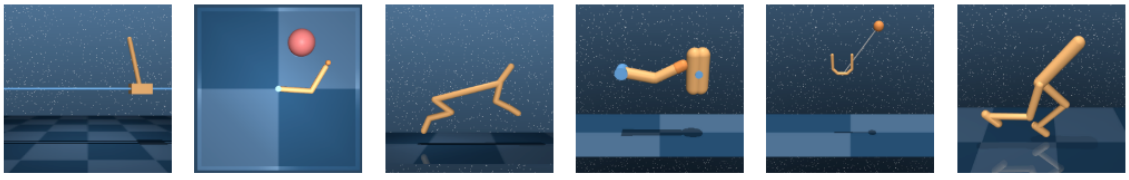
RSSM also learns an encoder $q(z_t|o_{\leqslant t}, a_{<t})$, observation model $p(o_t|s_t, z_t)$, also called a decoder, and reward model $p(r_t|s_t, z_t)$. The encoder is used to infer an approximate belief over the current latent variable from the history. The observation model provides a rich training signal but is not used for planning.

Given these components, the authors implement the policy as a planning algorithm that searches for the best sequence of future actions. In contrast to model-free and hybrid reinforcement learning algorithms, the authors do not use a policy or value network.

The cross entropy method [3] (CEM) is used to search for the best action sequence under the model. The authors decided on this algorithm because of its robustness and because it solved all considered tasks when given the true dynamics for planning. CEM is a population-based optimization algorithm that infers a distribution over action sequences that maximize the sum of rewards for a short future horizon. It uses the model to evaluate sampled candidate sequences in the optimization process. Because the reward is modeled as a function of the latent state, it can operate purely in latent space without generating images, which allows for fast evaluation of large batches of action sequences. After optimization finished, the first action from the resulting sequence is issued to the environment and the process starts from scratch in the next state.

Since the agent may not initially visit all parts of the environment, it iteratively collect new experience and further refine the dynamics model.

Using only pixel observations, PlaNet's agent solves continuous control tasks from DeepMind control suite (see fig. 34) with contact dynamics, partial observability, and sparse rewards, which exceed the difficulty of tasks that were previously solved by planning with learned models. PlaNet uses substantially fewer episodes and reaches final performance close to and sometimes higher than strong model-free algorithms.



(a) Cartpole (b) Reacher (c) Cheetah (d) Finger (e) Cup (f) Walker

Fig. 34. DeepMind Control Suite: image-based control domains used in PlaNet's experiments [21].

Within 100 episodes, PlaNet outperforms the policy-gradient method A3C [37] trained from proprioceptive states for 100,000 episodes, on all tasks. After 500 episodes, it achieves performance similar to D4PG [1], trained from images for 100,000 episodes, except for the finger task. PlaNet surpasses the final performance of D4PG with a relative improvement of 26% on the cheetah running task.

Moreover, the authors trained a single agent on all six tasks. The agent is not told which task it is facing, it needs to infer this from the image observations. The agent solves all tasks while learning slower compared to individually trained agents. This indicates that the model can learn to predict multiple domains, regardless of the conceptually different visuals.

PlaNet is a working example of a model-based agent that learns a latent dynamics model from high-dimensional image observations and chooses actions by fast planning in latent space. The authors show that their agent succeeds at several continuous control tasks from image observations, reaching performance that is comparable to the best model-free algorithms while using 200 times fewer episodes and similar or less computation time. The results show that learning latent dynamics models for planning in image domains is a promising approach. This thesis will adopt PlaNet to its objectives.

3.3. Model-Based Reinforcement Learning for Atari

In [30], the authors explore how video prediction models can enable RL agents to solve Atari games with orders of magnitude fewer interactions than model-free methods. They describe Simulated Policy Learning (SimPLe), a complete model-based deep RL algorithm based on video prediction models, and present a comparison of several model architectures, including a novel architecture that yields the best results in Atari games [2].

SimPLe, apart from an Atari 2600 emulator environment env, use a neural network simulated environment env' which they call a world model. The authors find out that crucial decisions in the design of world models are: inclusion of stochasticity, clipping reconstruction loss to a constant and, because simulator env' is supposed to consume its own predictions from previous steps which will be imperfect due to compounding errors, to mitigate a problem of the model drifting out of the area of its applicability they randomly replace in training some frames of the input by the model prediction from the previous step.

The environment env' shares the action space and reward space with env and produces visual observations in the same format, as it will be trained to mimic env. The authors principal aim is to train a policy π using a simulated environment env' so that π achieves good performance in the original environment env. Using short rollouts for a policy training is crucial to mitigate the compounding errors problem described in the section 2.5. To ensure exploration of further episode states, SimPLe starts training rollouts from randomly selected states taken from the real data buffer.

In this training process the authors aim to use as few interactions with env as possible. The initial data to train env' comes from random rollouts of env. As this is unlikely to capture all aspects of the environment, they use the data-aggregation iterative method.

Experiments evaluate SimPLe on a range of Atari games and show that it achieves competitive results compared to model-free baselines with only 100K interactions between the agent and the environment, which corresponds to about two hours of real-time play. SimPLe is significantly more sample-efficient than a highly tuned version of the state-of-the-art Rainbow algorithm [23] on almost all games. In particular, in low data regime of 100k samples, on more than half of the games, SimPLe achieves a score which Rainbow requires at least twice as many samples. In the best case of Freeway, it is more than 10x more sample-efficient.

SimPLe is a similar approach to model-based RL like in World Models. The authors train the dynamics model and use it to generate new experience, the same as in World Models, but in observations space, opposite to World Models.

Although this thesis share the goal of sample efficient RL via model-based planning and learning in complex environments, like Atari games, with SimPLe, it tries to accomplish it in fundamentally different way. The thesis solution focuses on training accurate transitions model in the latent state-space to enable fast simulation and search using this model. Pixel-perfect reconstructions are not a concern.

3.4. Value Prediction Network

Paper [42] proposes a novel deep reinforcement learning architecture, called Value Prediction Network (VPN), which integrates model-based planning and model-free learning of reward and value functions into a single neural network. In contrast to typical model-based RL methods, VPN learns the dynamics model of an abstract state space sufficient for predicting future rewards and values, rather than future observations.

In order to train VPN, the authors propose a combination of temporal-difference search [50] and n-step Q-learning [37]. In brief, VPN learns to predict values via Q-learning and rewards via supervised learning. At the same time, VPN perform lookahead planning to choose actions during play and compute target Q-values during training.

VPN has the ability to simulate the future and plan based on the simulated future abstract-states. Although many existing planning methods (e.g. MCTS) can be applied to the VPN, the authors implement a simple planning method which rollout trajectories using the VPN up to a certain depth and aggregates all intermediate value estimates as described in the paper [42]. The planning procedure estimates the current state Q-values which an agent uses to decide on the next action.

Experimental results show that VPN has several advantages over both model-free and model-based baselines in a stochastic navigation task where careful planning is required but building an accurate observation-prediction model is difficult. Furthermore, VPN outperforms Deep Q-Network [38], strong model-free baseline method, on several Atari games even with short-

lookahead planning, demonstrating its potential as a new way of learning a good state representation.

VPN uses an abstract model of rewards to augment model-free learning with good results on a number of Atari games. However, this method does not actually aim to model or predict future environment's states and achieves clear but relatively modest gains in sample-efficiency. On the other hand, it is an example of simple tree search algorithm which can successfully use the learned model for planning, something that this thesis could base on.

4. TECHNICAL DETAILS

The code architecture and the framework that was created to accelerate this research are described.

4.1. HumbleRL framework

Reinforcement Learning scientists tend to write the entire code from scratch by themselves, instead of using existing RL frameworks. This is justified by the fact, that the commonly available frameworks are not flexible enough for intended experiments or require a specific backend like TensorFlow, which might be disfavored. HumbleRL [28] was created with this problem in mind. Its simple API allows to perform a variety of RL experiments without any restrictions on the algorithms used. Since the backend is not tied to any specific technology, it is possible to mix different neural network frameworks or not use them at all. HumbleRL provides the boilerplate code of RL loop in fig.21 and determines the common interfaces between an agent and an environment, the rest is designed by the user.

4.1.1. Architecture

Framework architecture is depicted in fig. 41. An agent is represented by the Mind class. The Mind encapsulates action planning logic and provides it via the plan method. In order to learn, the agent acts in an environment represented by the Environment class. The Environment provides methods for resetting, taking steps, rendering and getting information about the environment. The MDP class provides the same methods like the environment, and in principle the Environment can be created from the MDP, but additionally it allows direct access to the sample transition model, see the section 2.2.
The framework includes a factory function that creates and returns e.g. wrapped OpenAI Gym environment. The agent is not usually presented with raw environment observations. Instead, it looks at states preprocessed by the Interpreter. Different interpreters can be joined together with the ChainInterpreter class. It acts as a preprocessing pipeline, with each subsequent interpreter using the output of a previous one as an input.

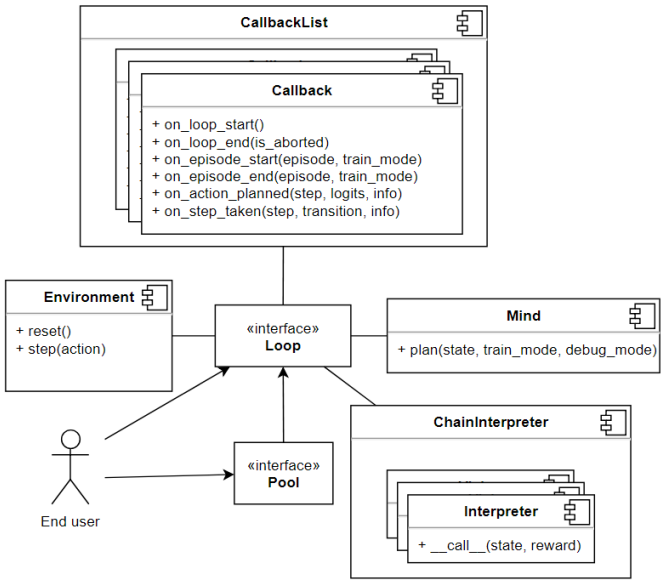


Fig. 41. HumbleRL architecture

Framework user does not need to call all of those methods directly, those are utilized by the loop function. This function gets an action from the Mind, executes it in the Environment and then next observation is preprocessed with the Interpreter in preparation for the next step. To extend basic loop functionality, user can define callbacks that implement the Callback interface. Callbacks can react to events:

- at the beginning and ending of the loop,
- at the beginning and ending of each episode,
- after action is planned by the Mind,
- after step is taken in the Environment.

Callbacks are accumulated in the CallbackList. The entire loop function logic is shown in fig. 42. Parallel version of loop function is available as the pool function. It uses predefined number of workers to execute a pool of Minds in their own Environments in parallel.

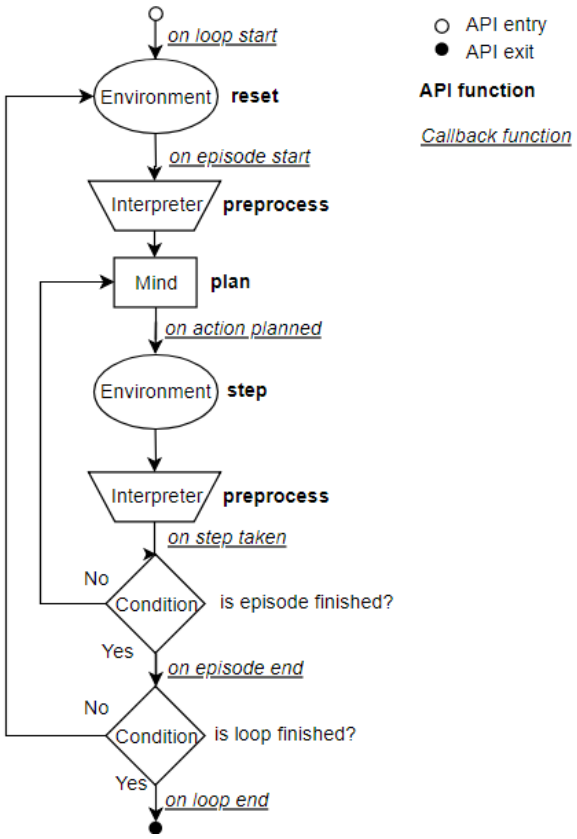


Fig. 42. HumbleRL loop function overview

World Models and AlphaZero implementations use this framework and hence can be joined together into one architecture for experiments.

4.2. Hardware

This work used DGX Station from NVIDIA to run experiments. It is the workstation with four NVIDIA Tesla V100 Tensor Core GPUs, integrated with a fully-connected four-way NVLink architecture, delivering 500 teraFLOPS of power, enabling faster experimentation. The system run TensorHive [44] for managing and monitoring computing resources.

This work did not use distributed training or evaluation. Each experiment was run on one GPU, but multiple experiments were run in parallel on multiple GPUs.

This work has been partially supported by Statutory Funds of Electronics, Telecommunications and Informatics Faculty, Gdansk University of Technology, which provided access to DGX Station deep learning server.

4.3. Benchmarks

4.3.1. Arcade Learning Environment

The Arcade Learning Environment (ALE) has became a platform for evaluating artificial intelligence agents. Originally proposed by Bellemare et. al. [2], the ALE makes available dozens

of Atari 2600 games for an agent training and evaluation. The agent is expected to do well in as many games as possible without game-specific information, generally perceiving the environment through a video stream which makes the problem partially observable as already mentioned in the section 2.1. Atari 2600 games are excellent environments for evaluating AI agents for three main reasons: they are varied enough to provide multiple different tasks, requiring general competence, they are interesting and challenging for humans and they are free of experimenter's bias, having been developed by an independent party.

In the context of the ALE, a discrete action is a number in range from 0 to 17 inclusive which encodes the composition of a joystick direction and an optional button press. The agent observes a reward signal, which is typically the change in the player's score (the difference in score between the previous time step and the current time step), and an observation $o_t \in O$ of the environment. This observation can take form of a single 210 × 160 image and/or the current 1024-bit RAM state. Because a single image typically does not satisfy the Markov property the ALE is formalised as POMDP. Observations and the environment state are distinguished, with the RAM data being the real state of the emulator. A frame (as a unit of time) corresponds to 1/60th of a second, the time interval between two consecutive images rendered to the television screen. The ALE is deterministic, which means that given a particular emulator state s and a action a there is a unique next state s', that is, $P^a_{ss'} = p(s'|s, a) = 1$.

Agents interact with the ALE in an episodic fashion. An episode begins by resetting the environment to its initial configuration, s_0, and ends at a given endpoint depending on a game. The primary measure of an agent's performance is the score achieved during an episode, namely the undiscounted sum of rewards for that episode. While this performance measure is quite natural, it is important to realize that score is not necessarily an indicator of AI progress. In some games, agents can exploit the game's mechanics to maximize sum of rewards, but not complete the game's goal in human's understanding [10].

Common preprocessing techniques include frame skipping [40] which restricts the agent decision points by repeating a selected action for 4 consecutive frames. It is used in reinforcement learning [38] to reduce the planning horizon and provide a clearer learning signal to the model, but it also speeds up execution.

This work uses ALE through OpenAI Gym API [4], specifically two Atari games are used as benchmarks: Boxing and Freeway.

Boxing is a video game based on the sport of boxing. Boxing shows a top-down view of two boxers, one white and one black. When close enough, a boxer can hit his opponent with a punch. This causes his opponent to reel back slightly and the boxer scores a point, a reward of 1. In the other situation, when the boxer gets hit, he gets a negative reward of -1. There are no knockdowns or rounds. A match is completed either when one player lands 100 punches (a 'knockout') or two minutes have elapsed. In the case of a decision, the player with the most landed punches is the winner. Ties are possible. While the gameplay is simple, there are subtleties, such as getting an opponent on the 'ropes' and 'juggling' him back and forth between alternate punches.

Fig. 43. Example of Boxing level

In Freeway an agent controls a chicken who can be made to run across a ten lane highway filled with traffic in an effort to ”get to the other side.” Every time a chicken gets across a reward of 1 is earned by the agent. If hit by a car, then a chicken is forced back slightly. The goal is to score as much points as possible in the two minutes. The chicken is only allowed to move up or down. The major challenge in this environment are sparse rewards. The agent scores only when successfully crosses the highway, which is not a trivial task.

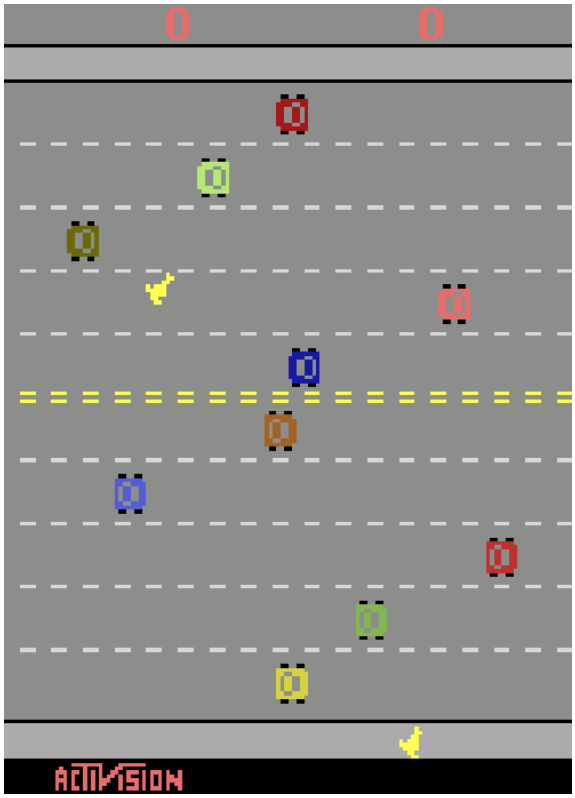


Fig. 44. Example of Freeway level

4.3.2. *Sokoban*

Sokoban is a classic planning problem which is not strictly part of ALE, but it can be added to OpenAI Gym API [48]. It is a challenging one-player puzzle game in which the goal is to navigate a grid world maze and push boxes onto target tiles. A Sokoban puzzle is considered solved when all boxes are positioned on top of target locations. The player can move in all 4 cardinal directions and only push boxes into an empty space (as opposed to pulling). For this reason many moves are irreversible and mistakes can render the puzzle unsolvable. A human player is thus forced to plan moves ahead of time. Artificial agents should similarly benefit from a learned model and simulation.

Despite its simple rule set and fully observability - each game frame shows all blocks, targets, walls and agent positions and states - Sokoban is an incredibly complex game for which no general solver exists. It can be shown that Sokoban is NP-Hard and PSPACE-complete [11]. Sokoban has an enormous state space that makes it inassailable to exhaustive search methods. An efficient automated solver for Sokoban must have strong heuristics, just as humans utilize their strong intuition, so that it is not overwhelmed by the number of possible game states.

The implementation of Sokoban used for those experiments procedurally generates a new level each episode. This means an agent cannot memorize specific puzzles. Together with the planning aspect, this makes for a very challenging environment. While the underlying game logic operates in a 10 × 10 grid world, with 7 possible elements in each grid 45, agents were trained directly on RGB sprite graphics. Fig. 46 shows an example of Sokoban level with 4 boxes. Finishing the game by pushing all blocks on the targets gives a reward of 10 in the last step. Also pushing a box on or off a target gives a reward of 1 or -1 respectively. In addition, a reward of -0.1 is given for every step which penalizes solutions with many steps.

Type	State	Graphic
Wall	Static	
Floor	Empty	
Box Target	Empty	
Box	Off Target	
Box	On Target	
Player	Off Target	
Player	On Target	

Fig. 45. Table with Sokoban possible elements in each grid, further referred to as blocks.

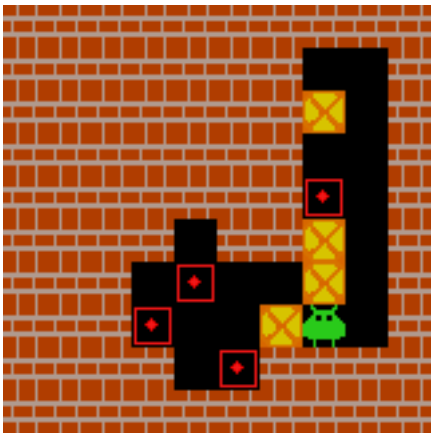


Fig. 46. Example of Sokoban level

5. SOLUTION DESCRIPTION

In this chapter three architectures are described. All of them involve a similar model learning approach, but differ substantially in technical details and planning algorithms. All architectures use computationally efficient latent space environment models that make predictions at a higher level of spatial abstraction, than at the level of raw pixel observations. The goal is to train a sufficiently accurate latent space environment's dynamics model, or such that accurately predicts future latent states and rewards to predefined cut-off point in time, and use it to plan and solve the environment. Those architectures, and experiments carried out on them, present evolution of approaches towards this goal.

5.1. Original World Models (OWM)

This section describes in details the architecture of solution based on World Models [18] called "Original World Models" (OWM), broadly described in the section 3.1. This architecture will be coupled with the AlphaZero which uses the learned model to simulate experience in the next section.

5.1.1. World model

An environment provides the agent with a high dimensional input observation, o, at each time step. It is a 2D image, a game frame, that is part of a video sequence. The Vision module role, as already mentioned in the section 3.1, is to learn an abstract representation of each observed input frame. Vision is implemented as a simple Variational Autoencoder described in details in chapter 2. It is trained to encode each frame into low dimensional latent vector, z, by minimizing the difference between a given frame and the reconstructed version of the frame produced by the decoder.

The role of the Memory module is to acquire full knowledge of an environment state by encoding what happens over time. Memory is trained to predict a probability density of the next latent variable. In consequence, Memory enables simulation of the environment. Memory is implemented as a recurrent neural network with the Mixture Density Network (MDN) on top of a RNN's hidden state. In literature this architecture is called MDN-RNN [15]. It will be further referred to as the stochastic Memory module.

Figure 51 depicts the world model, the Vision and Memory modules interconnections, in graphical form. More specifically, the world model components are:

- Deterministic state model: $h_t = f(h_{t-1}, z_{t-1}, a_{t-1})$
- Stochastic state model: $z_t \sim p(z_t|h_t) = \sum_c \pi_c(h_t) p(z_t|h_t, c)$
- Observation model (decoder): $o_t \sim p(o_t|z_t)$

where o, z and a are high-dimensional observations, latent variables and actions respectively. $f(h_{t-1}, z_{t-1}, a_{t-1})$, the deterministic state model, is implemented as a recurrent neural network and h_t is its hidden state. The stochastic state model is a mixture of Gaussians with mean and

variance parameterised by a feed-forward neural network. c is a mixture's component and $\pi(h)$ is a normalized vector of mixing coefficients as a function of the RNN's hidden state. The observation model is Bernoulli distribution parameterised by a deconvolutional neural network described below in implementation details.

Since the model is non-linear, directly computing the state posteriors is intractable. Instead, an encoder q is used to infer approximate state posteriors from observations, where $q(z_t|o_t)$ is a diagonal Gaussian with mean and variance parameterised by a convolutional neural network followed by a feed-forward neural network. Vision is trained using variational inference described in the section 2.5.

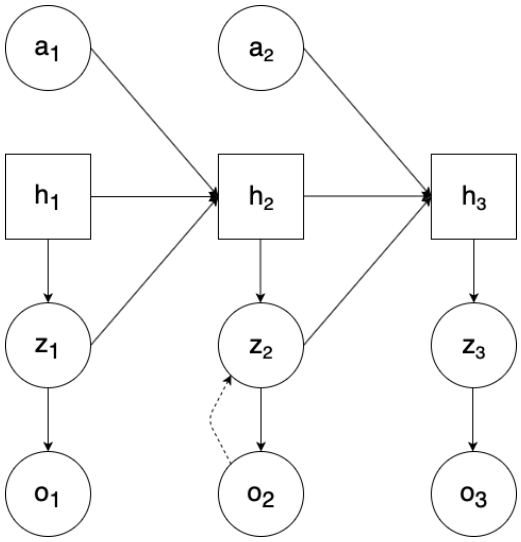


Fig. 51. World Models graphical model of Memory: solid arrows describe the predictive model, dotted arrow describes the inference model, stochastic nodes are circles and squares depict deterministic nodes.

5.1.2. Controller

The Controller module is responsible for determining the course of actions to take in order to maximize the expected cumulative reward of the agent during an episode in the environment. Because it bases its decisions on abstract environment representation, learnt by the Vision and Memory modules, it can be deliberately made as simple and small as possible and trained separately from Vision and Memory modules, so that most of the agent's complexity resides in the world model. The Controller module is a simple single layer linear model that maps a features vector constructed from latent variable of the Vision module z and hidden state of the Memory module h directly to action a at each time step:

$$a_t = W_c[z_t\, h_t] + b_c$$

where W_c and b_c are Controller's weights matrix and biases vector that maps the concatenated features vector $[z_t\, h_t]$ to the output action vector a_t. The action vector is $|A|$-dimensional, where $|A|$ is number of legal actions in an environment. It encodes predicted score of each action

in the state and is used to decide which action to choose in the environment. Maximum action is taken, which means that greedy policy is used. It is true for training and testing phases. Because evolutional strategy algorithm is used for training, it does not impair exploration which is done on parameters level.

5.1.3. Data collection

To train Vision and Memory modules first collection of 10,000 random rollouts of the environment are gathered to create a dataset. An agent is acting randomly to explore the environment multiple times and records the random actions taken and the resulting observations from the environment. This dataset is used to train the Vision module.

Next, trained Vision is used to preprocess the dataset for the Memory module training, which works entirely in the latent space. Because the encoder approximate state posteriors from observations with a diagonal Gaussian, a precomputed set of means and variances for each of the frames are stored. During training the latent variables are sampled to construct a training batch. This prevents overfitting of Memory to specific sampled latent variables.

The Controller module is trained using evolutional strategy which rollouts the environment in each epoch to evaluate population.

5.1.4. Preprocessing

Each frame, before it is used for any training, is central cropped if a frame from an environment includes some kind of border which does not inform an agent in anyway. This operation depends on a specific environment. It is then resized to 64 x 64 pixels for all environments. All three colour channels are preserved. Actions are one-hot encoded. Frame skipping described in the section 4.3.1 is used for Atari games only.

5.1.5. Implementation details

HumbleRL is used to implement the original World Models architecture from the paper. This allows for easy adjustments for experiments purposes and to couple the world model with AlphaZero implementation in HumbleRL. An agent exploring an environment (Mind) and a callback are used to gather transitions and save them to an external storage. The framework allows to focus strictly on collecting trajectories and not worry about agent-environment interactions.
Collected transitions are used to train the Vision and Memory components. The popular LSTM architecture [25] implements the Memory module RNN with 256 hidden units. The MDN is composed of 5 16-dimensional Gaussians with mean and log standard deviation parameterised by linear models. The Vision module neural networks are convolutional and deconvolutional neural networks shown in fig. 52 where latent variable size, μ and σ vectors dimensionality, equals 16.
For Vision training Keras [9] framework is used to adjust the parameters by the stochastic gradient descent on a standard Evidence Lower Bound loss. For Memory training PyTorch [43] framework is used, since it is easier to work with recurrent neural networks than in Keras, to, again, adjust

the parameters by the stochastic gradient descent on a negative log-probability loss of the mixture density network. HumbleRL is not constricted to work with any particular deep learning library, so it is not a problem to mix the solutions, as long as trained models are wrapped in HumbleRL's interfaces. The Vision module is trained using the Adam optimizer [31] with a learning rate of 10^{-3} and $\epsilon = 10^{-7}$ on batches of 256 images. The Memory module is trained using the Adam optimizer [31] too with a learning rate of 10^{-3} and $\epsilon = 10^{-8}$ on batches of 128 sequence chunks of length 1000.

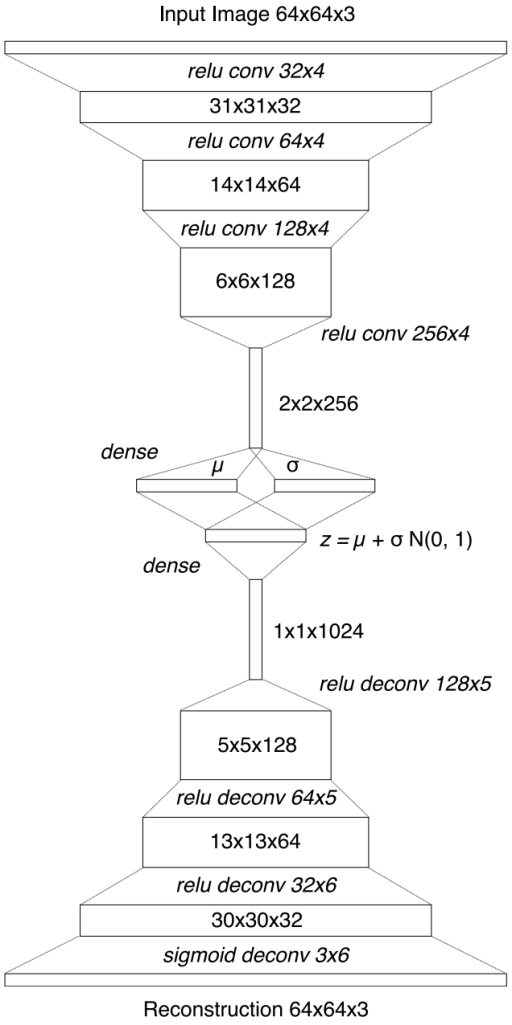


Fig. 52. World Models VAE neural network architecture [18]

The Controller module gets a feature vector as its input, consisting of latent variable z and the hidden state of the MDN-RNN. In all environments, this hidden state is the output vector h of the LSTM. Covariance-Matrix Adaptation Evolution Strategy (CMA-ES) [22] is used for the Controller module training. It evolves the weights W_c and biases b_c of the module. A population size of 64 is used and each agent plays in the environment 5 episodes. The fitness value for the agent is the average cumulative reward of the 5 episodes.

Hyper-parameters presented are used as defaults in experiments described in the next chapter.

5.2. World Models and AlphaZero (W+A)

World Models' agent [18] successfully plans using a learned model where the model is used to generate simulated experience on which the policy is trained. This section describes attempt to adjust and utilize the world model part of the agent in the AlphaZero [51] search algorithm for environments with dense rewards. The architecture is called "World Models and AlphaZero" (W+A). This is different application of the model than in the original paper, where only future latent variables and done flag are predicted, which is sufficient in their benchmark with sparse rewards, and therefore the world model needs to be extended with a reward predictor and the controller is replaced by AlphaZero.

5.2.1. World model

For the world model part Vision stays the same with a major change in Memory. Because benchmarks include only deterministic environments and AlphaZero in its original form can only work with a deterministic dynamics model, this architecture use the Memory module without the MDN and hence it lacks the stochastic state model.

- Transition model: $h_t, s_t = f(h_{t-1}, s_{t-1}, a_{t-1})$
- Reward model: $r_t = f(h_t)$
- Observation model (decoder): $o_t \sim p(o_t|s_t)$

It uses RNN, where h is its hidden state, with a linear model on top to output the next latent state, s, and a linear model to output a reward. r. It will be further referred to as the deterministic Memory module. It is presented in fig. 53.

As in original World Models, an encoder $q(s_t|o_t)$ is used to infer approximate state posteriors from observations and Vision is trained using variational inference described in the section 2.5.

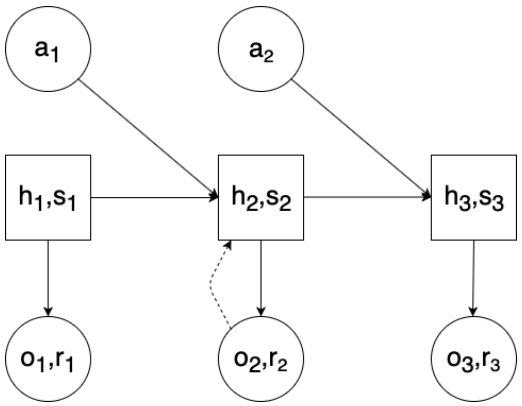


Fig. 53. World Models graphical model of deterministic Memory: solid arrows describe the predictive model, dotted arrow describes the inference model, stochastic nodes are circles and squares depict deterministic nodes.

5.2.2. Controller

AlphaZero [51] is very similar to the MCTS algorithm, explained in the chapter 2. The selection and evaluation phases are modified though. AlphaZero uses policy and value networks

to guide its search. Each edge in the search tree, (s, a, s') where s is a state, a is an action and s' is the next state, stores a prior probability of choosing it $P(s, a)$, a visit count $N(s, a)$, an action-value $Q(s, a)$ and a reward $R(s, a, s')$ of transition represented by this edge. Each simulation starts from the root state, s_t at depth or time step $t = 0$, and iteratively selects moves that maximise an upper confidence bound $Q(s, a) + U(s, a)$, where $U(s, a) \propto P(s, a)/(1 + N(s, a))$ [52], until a leaf node is encountered. This leaf position is expanded by the world model to generate both the next state and reward, $s_{t'}$ and $r_{t'}$ at depth or time step t', and evaluated by the networks to generate both prior actions probabilities and its value, $P(s_{t'}, \cdot)$ and $V(s_{t'})$. Each edge traversed in the simulation is updated to increment its visit count $N(s_t, a_t)$, and to update its action-value to the mean evaluation over these simulations:

$$Q(s_t, a_t) = 1/N(s_t, a_t) \sum_{t'} \Big[\sum_{i=t+1}^{t'} r_i + V(s_{t'}) \Big]$$

Figure 54 depicts this process. After multiple simulations, which could be stopped after the absolute number of simulations or after timeout, the result is a vector of search probabilities recommending moves to play, $\boldsymbol{\pi}$, proportional to the visit count for each move, $\pi_a \propto N(s, a)$. How these are used to choose an action to take by the agent is discussed in the next section.

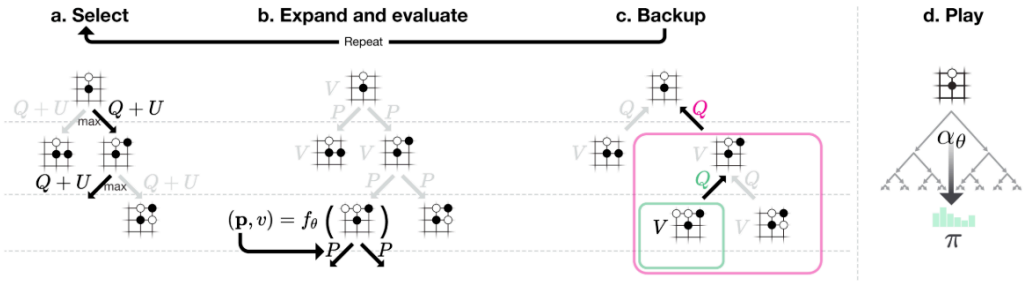


Fig. 54. Monte-Carlo tree search in AlphaZero [52]

5.2.3. *Implementation details, data collection and preprocessing*

The world model implementation details, preprocessing and data collection for Vision and Memory modules are the same as in the original World Models. The world model latent state size is 16 and RNN is LSTM [25] with the hidden size 256. An input to value and policy networks, is the concatenated features vector of latent state and hidden state, $[s_t\, h_t]$.

The AlphaZero controller uses the world model for simulations. The Vision module is used as the Interpreter which encodes incoming observations into latent space. The Memory module, wrapped in the MDP interface from HumbleRL, is used in the expansion phase of AlphaZero. The Mind class, which is implemented by the AlphaZero algorithm, returns actions' scores. These are actions visit counts from the root state node, which are then used to choose an action by a policy. During training actions are sampled with probability proportional to these visit counts for 12 warm-up steps, which is 48 game frames assuming four frames skipping, at the beginning of each episode which proved sufficient to ensure stochastic starts and enhance exploration. After

the warm-up phase, a greedy policy is used. During testing always greedy policy is used. It picks an action which was visited most often. Pseudo-code written in Python of the search algorithm in the Mind is shown below:

```
def plan(self, state):
    # Get/create root node
    root = self.query_tree(state)

    # Perform simulations
    simulations = 0
    start_time = time()
    while time() < start_time + self.timeout and simulations < self.max_simulations:
        # Simulate
        simulations += 1
        leaf, path = self.simulate(root)

        # Expand and evaluate
        value = self.evaluate(leaf)

        # Backup value
        self.backup(path, value)

    # Get actions' visit counts
    actions = np.zeros(self.model.action_space.num)
    for action, edge in root.edges.items():
        actions[action] = edge.num_visits

    return actions
```

AlphaZero value and policy networks are two linear models trained by reinforcement learning from self-play games, starting from randomly initialized parameters. Each game is played by running the search, described above, at each position and then selecting a move either proportionally (for exploration) or greedily (for exploitation) according to returned move probabilities, which are normalised visit counts at the root state, π. These search probabilities usually select much stronger moves than the raw move probabilities of the policy network. The MCTS search may therefore be viewed as a powerful policy improvement operator [54]. Self-play with search – using the improved MCTS-based policy to select each move by the agent, then using the game cumulative reward, or the return, as a sample of the value at each time step – may be viewed as a powerful policy evaluation operator. The main idea of this reinforcement learning algorithm is to use these search operators repeatedly in a policy iteration framework [52]. The linear models' parameters are updated to make the move probabilities and values more closely match the improved search probabilities and self-play cumulative rewards at each step. These new parameters are used in the next iteration of self-play to make the search even stronger.

The agent's experience and score statistics used for training are gathered using callbacks during the self-play phase. Maximum of 1000 latest games are kept. The models training phase takes place after 10 self-play games and lasts for 5 epochs. The training phase is performed using the Keras [9] framework. Specifically, the parameters are adjusted by the stochastic gradient descent on a loss function that sums over mean squared errors of the value network, cross-entropy losses of the policy network and L2 weights regularization scaled by a factor of 10^{-4} in batches of 256 examples, where each example is a features vector of a state, a value sample from self-play and a MCTS action probabilities vector. The Nesterov's momentum optimizer [53] is used with a learning rate of 10^{-2} and a momentum of $0,9$.

Hyper-parameters presented are used as defaults in experiments described in the next chapter.

5.3. Discrete PlaNet (DPN)

The PlaNet paper [21] shows working example of a planning agent that searches for the best sequence of future actions using a learned model in continuous control tasks. This is close to what this work tries to accomplish, but for a different type of environments. This section describe how it was utilized in episodic discrete tasks. The architecture is further referred to as “Discrete PlaNet” (DPN).

5.3.1. World model

This architecture uses recurrent state space model (RSSM). Same as in World Models, the model is provided with image observations. It even uses the same Variational Autoencoder to encode the observations into latent space. The difference lies in the dynamics model shown in fig. 55 with following components:

- Deterministic state model: $h_t = f(h_{t-1}, z_{t-1}, a_{t-1})$
- Stochastic state model: $z_t \sim p(z_t|h_t)$
- Reward model: $r_t \sim p(r_t|h_t, z_t)$
- Observation model (decoder): $o_t \sim p(o_t|h_t, z_t)$

where o, z and a are high-dimensional observations, latent variables and actions respectively. $f(h_{t-1}, z_{t-1}, a_{t-1})$, the deterministic state model, is implemented as a recurrent neural network and h_t is its hidden state. The stochastic state model is Gaussian with mean and variance parameterised by a feed-forward neural network, the observation model is Gaussian with mean parameterised by a deconvolutional neural network and identity covariance, and the reward model is a scalar Gaussian with mean parameterised by a feed-forward neural network and unit variance. Since directly computing the state posteriors is intractable, an encoder $q(z_t|o_{\leqslant t}, a_{<t}) = \prod_{i=1}^{t} q(z_t|h_t, o_t)$ is used to infer approximate state posteriors from past observations and actions, where $q(z_t|h_t, o_t)$ is a diagonal Gaussian with mean and variance parameterised by a convolutional neural network followed by a feed-forward neural network. The model is trained using variational inference with the latent overshooting regularizer described in the section 2.5.

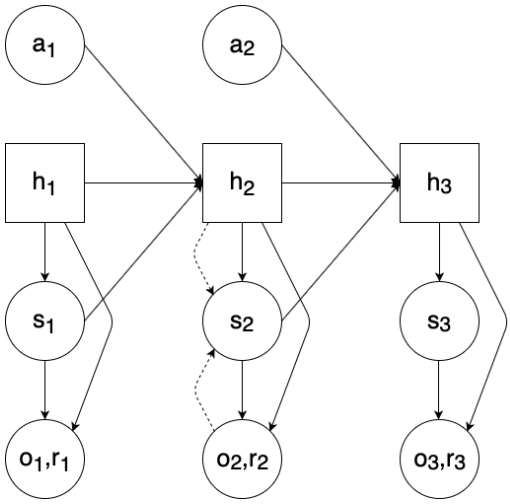


Fig. 55. PlaNet probabilistic graphical model: solid arrows describe the predictive model, dotted arrow describes the inference model, stochastic nodes are circles and squares depict deterministic nodes.

The main deviations from the World Models architecture are VAE's decoder and encoder (named Vision in World Models). They use the dynamics model's hidden state for prediction and inference. This way the likelihood and the posterior are conditioned on past observations too, as opposed to World Models.

5.3.2. Planner

The agent plans using the cross entropy method (CEM) [3] to search for the best action sequence under the model. CEM is a population-based optimization algorithm that infers a distribution over action sequences that maximize the objective. It was originally used in PlaNet for tasks with continuous action-space and here it is adjusted for discrete action-space of Atari games. First, a time-dependent multidimensional diagonal Gaussian belief over optimal action sequences gets initialized: $a_{t:t+H} \sim Normal(\mu_{t:t+H}, \sigma^2_{t:t+H} I)$, where t is the current time step of the agent and H is the length of the planning horizon. The action vector, a, can be interpreted as actions scores. Each dimension represent score for one discrete action in the environment.
Starting from zero mean and unit variance, it is used to sample candidate action sequences. To evaluate a candidate action sequence under the learned model a trajectory starting from the current state is sampled by the model using the action sequence as an input and the predicted rewards are summed along the sequence. This sum is used as the action sequence fitness score. Since it is a population-based optimizer, it is sufficient to consider a single trajectory per action sequence and thus focus the computational budget on evaluating a larger number of different sequences. Then, the belief gets re-fitted to the best sequences and next optimization iteration starts the same way. At the end, the planner returns the mean vector, μ_t, of the belief for the current time step, representing scores for actions in the current time step, which is then used by the policy to choose discrete action. Importantly, after receiving the next observation, the belief over action sequences starts from zero mean and unit variance again to avoid local optima.

Because the reward is modeled as a function of the latent state, the planner can operate purely in latent space without generating images, which allows for fast evaluation of large batches of action sequences. The algorithm is presented in the fig. 56 below.

Input : H Planning horizon distance | $q(s_t \mid o_{\leq t}, a_{<t})$ Current state belief
I Optimization iterations | $p(s_t \mid s_{t-1}, a_{t-1})$ Transition model
J Candidates per iteration | $p(r_t \mid s_t)$ Reward model
K Number of top candidates to fit

Initialize factorized belief over action sequences $q(a_{t:t+H}) \leftarrow \text{Normal}(0, \mathbb{I})$.
for *optimization iteration* $i = 1..I$ **do**
  `// Evaluate` J `action sequences from the current belief.`
  for *candidate action sequence* $j = 1..J$ **do**
    $a^{(j)}_{t:t+H} \sim q(a_{t:t+H})$
    $s^{(j)}_{t:t+H+1} \sim q(s_t \mid o_{1:t}, a_{1:t-1}) \prod_{\tau=t+1}^{t+H+1} p(s_\tau \mid s_{\tau-1}, a^{(j)}_{\tau-1})$
    $R^{(j)} = \sum_{\tau=t+1}^{t+H+1} \mathrm{E}[p(r_\tau \mid s^{(j)}_\tau)]$
  `// Re-fit belief to the` K `best action sequences.`
  $\mathcal{K} \leftarrow \text{argsort}(\{R^{(j)}\}_{j=1}^{J})_{1:K}$
  $\mu_{t:t+H} = \frac{1}{K} \sum_{k \in \mathcal{K}} a^{(k)}_{t:t+H}, \quad \sigma_{t:t+H} = \frac{1}{K-1} \sum_{k \in \mathcal{K}} |a^{(k)}_{t:t+H} - \mu_{t:t+H}|.$
  $q(a_{t:t+H}) \leftarrow \text{Normal}(\mu_{t:t+H}, \sigma^2_{t:t+H}\, \mathbb{I})$
return *first action mean* μ_t.

Fig. 56. Latent planning with CEM [21]

5.3.3. Data collection

Since the agent may not initially visit all parts of the environment, the data-aggregation iterative method described in the section 2.5 is used. Starting from a small amount of 6 seed episodes collected under random actions, the model is trained and one additional episode is added to the data set every 5000 update steps.

5.3.4. Preprocessing

Each image gets preprocessed by reducing the bit depth to 5 bits as in [32]. It is then resized to 64 x 64 pixels for all environments without any cropping. Actions are one-hot encoded for all benchmarks and frame skipping described in the section 4.3.1 is used for Atari games only.

5.3.5. Implementation details

This time official code from repository [20] was adjusted and used for experiments. The code architecture follows principles from other PlaNet author's paper [19]. The RSSM uses a GRU [8] with 200 units as deterministic path in the dynamics model and implements all other functions as two fully connected layers of size 200 with ReLU activations [41]. Distributions in latent space are 30-dimensional diagional Gaussians with predicted mean and standard deviation. The observation model and the approximate state posterior are implemented with the Variational Autoencoder, the

same as in World Models (see fig.52). The reward model is implemented with a feed-forward neural network with two hidden layers of size 100. The model is trained jointly using the Adam optimizer [31] with a learning rate of 10^{-3} and $\epsilon = 10^{-4}$, and gradient clipping norm of 1000 on batches of 50 sequence chunks of length 50. The KL divergence terms are also scaled relatively to the reconstruction terms and the model is granted free nats by clipping the divergence loss below this value. Both parameters are tuned in the experiments chapter. The latent overshooting KL divergence terms are additionally scaled by a factor of $1/50$.

Planner uses planning horizon of 12, which means that it evaluates 12 actions in the future. Starting from zero mean and unit variance, 1000 candidate action sequences are sampled and evaluated under the learned model. Then the belief gets re-fitted to the top 100 action sequences with the highest fitness scores. After 10 iterations, the planner returns the mean of the belief for the current time step μ_t which is then used by the policy to choose discrete action. When collecting episodes for the training data set the epsilon greedy policy is used with $\epsilon = 0,3$. During test phase the greedy policy is used, which chooses the action that maximises the returned belief.

Hyper-parameters presented are used as defaults in experiments described in the next chapter.

6. EXPERIMENTS

In this chapter the three architectures described in the previous chapter are subject to different experiments that aim at making them to work. The architectures are evaluated using two metrics:

- The more accurate the model at the cut-off point, which is environment dependent, the better model learning algorithm.
- The higher final score of the planning agent using this learned model, the better.

The first metric is evaluated using observations reconstructions at each timestep which are compared to ground truth recordings from the dataset. The second metric is simply final score from the environment. Before experiments descriptions benchmarks and used hardware get reviewed.

6.1. OWM for Sokoban

This section focuses on a problem of training OWM in Sokoban. The goal was to train the Memory module to generate sharp and accurate future predictions of observations and obtain high score of Controller playing in the environment.

6.1.1. Train OWM in the Sokoban environment

The Vision module successfully learned to encode high dimensional observations into low dimensional latent states. Fig. 61 shows original observations (first and third columns) side by side with reconstructed observations from their encodings (second and fourth columns). These are zero step predictions, no future is predicted only encoding to latent space and decoding to image space again is done.

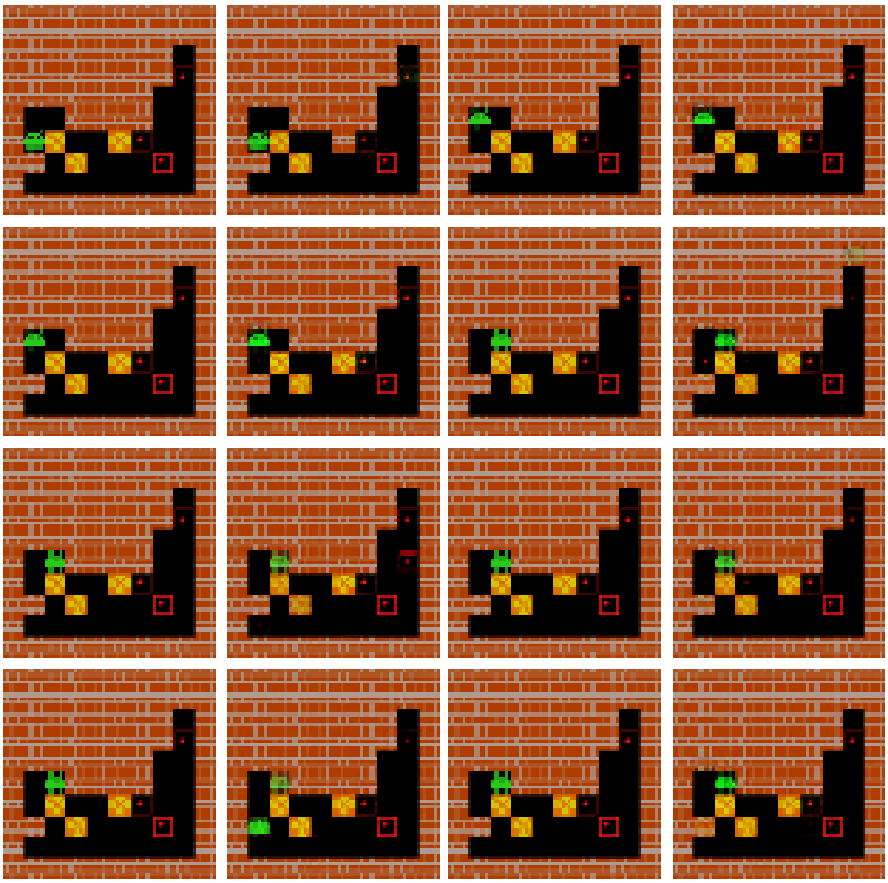


Fig. 61. Qualitative result of the Vision module training in Sokoban. First and third columns include original observations. Second and fourth columns include reconstructions. Each reconstruction was obtained by first encoding the original observation and then decoding it, using VAE encoder and decoder respectively.

It is hard to quantitatively measure the Memory module performance with e.g. log-likelihood. In practice, the single number does not tell much about quality of the predicted future states and its hard to interpret when the log-likelihood is high enough. In practice, though, it is far more useful to compare generated sequences of future observations with ground truth sequences with an eye of an expert, in this case the author of this thesis. What the expert looks for are sharp generated images which accurately resemble frames from the game. Moreover, the sequence needs to simulate subsequent actions properly, otherwise it is told that the sequence is noisy.

To generate a future prediction, 60 consecutive frames and actions from the dataset are fed into Memory to initialize its hidden state first. Then, when Memory is ready to generate future predictions, subsequent actions from the dataset and Memory own predicted latent states, as a contemporary states at following time steps, are fed into it to rollout the Memory module into the

future. What gets generated are not frames, of course, but latent states. Those latent states are then decoded into images by Vision.

The stochastic and deterministic Memory modules were not able to learn Sokoban dynamics. Fig. 62 shows that the stochastic Memory module very often can not determine an agent position. The agent disappears and blocks change into other blocks. The eighth row shows that pushing mechanics are not modeled, the agent passes through boxes. The deterministic Memory module does not do better.

The Controller module failed to learn how to solve any level, it behaved comparably to a random play. We suspect that VAE is unable to generate high-level abstract Sokoban representation and the shallow Memory and Controller modules can not grasp complex dynamics of Sokoban using this poor representation. This idea is further developed in the next experiment.

Fig. 62. Qualitative result of the Memory module training in Sokoban. Each row depicts the Memory module rollout in one episode. The first column include original observations from the evaluation dataset from which the rollouts start. The RNN's hidden state was initialized on preceding transitions in each episode. Each subsequent reconstruction was obtained by first predicting the next latent state by the Memory module and then decoding it using the VAE decoder.

6.1.2. Train OWM in the Sokoban environment on 10x10 grid world states

The latent state vector size is set to 64. This means that, in theory, this vector can accommodate full information about an observation. As noted before, Sokoban underlying game logic operates in a 10 × 10 grid world, where far edges of a level are always walls. This means that the level is described by 64 blocks organized in an 8 x 8 grid. In this experiment, this domain knowledge is exploited and the agent uses those 64 blocks as an input vector to the Memory module, bypassing the Vision module. It is worth noting, that the Vision module should learn this representation as it is the optimal encoding when the objective is to compress a pixel image into

a 64-dimensional vector and then reconstruct the original observation from it. However, despite use of the optimal encoding, the results have not been improved.

The proposed input format is optimal encoding if one wants to compress a pixel image and then reconstruct it. However, it is really poor representation of a current state of the environment if one wants to use linear combination of those features (blocks in each position) to infer optimal next action and this is exactly what the Controller module is tying to do. Modeling a value function could have more sense e.g. the value function could learn that a box on a target position yields higher value, but even it would have a hard time modeling more complex relations between entities in the environment. More useful for the Controller would be e.g. representation that includes information about distance between the box and each target position. Nevertheless, this could be not enough too. The box on the target position would get discounted for not being on some other target positions. Hence, there is need for feature saying "the box X placed on the target position Y". In the end, the linear combination of the proposed 64 raw features can not be used to learn good policy, better then a random play.

On the other hand, this representation includes, not well represented, but perfect information about an environment state. The Memory module creates its own environment representation encoded in its hidden state and then uses this representation to predict the next latent state. This Memory's hidden state is also utilized by the Controller. Still, it does not seem to encode useful enough information for the two, Memory and Controller, to do well on their tasks.

6.1.3. Train OWM in the Sokoban environment with auxiliary tasks

Auxiliary tasks [26] have proved to help create more informative representation of an environment. In this experiment, reward and value prediction tasks are added to the Memory module. In short, two additional linear models are added on top of the RNN to predict the next reward in the environment and model a value function. The reward head is trained in supervised manner to minimise MSE loss. The value head uses Monte-Carlo prediction algorithm, described in the section 2.2, to estimate target states value and it is trained in supervised manner to minimise MSE loss just like the reward head. In theory, it should help form a more informative hidden state of the Memory module. Consequently, it should help learn Sokoban's dynamics, but also generate representation on a higher level of abstraction that could prove useful for Controller. Moreover, the reward prediction will be needed in further work on planning with learned model.

For all that, the Memory module have not been able to learn to predict the rewards and values. Also, there was no improvement in Memory and Controller performance. It is suspected, that the main cause of this failure are sparse rewards in the training dataset. A random agent used to generate the dataset does not receive many positive rewards. Effectively, most of the episodes do not have any positive reward. Hence, the Memory module soon overfit on more or less constant reward and value. This yields insight that the data generation procedure does not cover state-space well. Iterative approach to gathering data, from a better and better agent, could solve this problem.

Moreover, it is not without significance that Sokoban has enormous state-space. Because each episode, or level, is randomly generated it is much different from the others - it is nearly impossible for an agent to see a similar state multiple times. Hence, Sokoban requires strong generalization capability from the Memory module. Simple RNN can lack capacity to create good representation and in turn achieve good prediction performance. A more flexible Memory module with larger capacity could manage this complexity and need for generalization.

The two insights from above are explored in the experiments with DPN, which have larger model and uses iterative training procedure.

6.2. World Models for Atari

Two more experiments below put World Models into the test. Firstly, OWN is trained in the Boxing environment, which has dense rewards and data collection using a random agent cover most of the state-space. If insights from the previous section are the only problems, then OWM should be able to achieve a good score in Boxing.
Then, the W+A architecture is tested in Boxing too.

6.2.1. Train OWM in the Boxing environment

The Vision module successfully learned to encode high dimensional observations into low dimensional latent states. Fig. 63 shows original observations (first and third columns) side by side with reconstructed observations from their encodings (second and fourth columns). These are zero step predictions, no future is predicted only encoding to latent space and decoding to image space again is done.

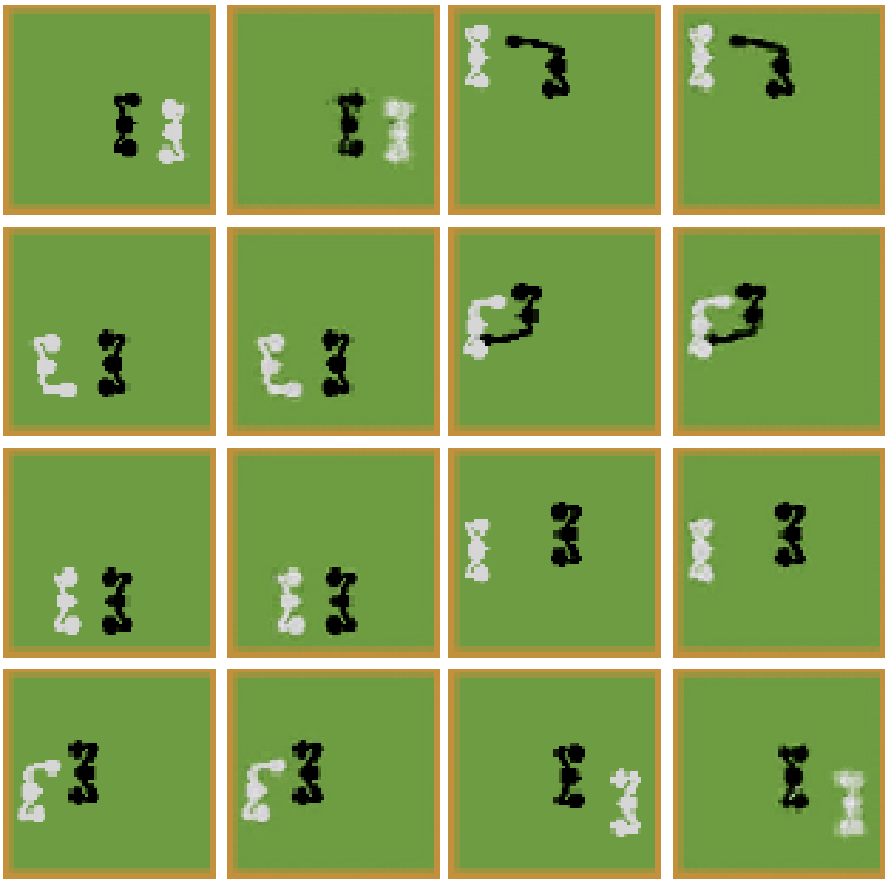


Fig. 63. Qualitative result of the Vision module training in Boxing. First and third columns include original observations. Second and fourth columns include reconstructions. Each reconstruction was obtained by first encoding the original observation and then decoding it, using VAE encoder and decoder respectively.

The same procedure as in Sokoban was used to generate Memory observations predictions.

The stochastic Memory modules was able to learn Boxing's dynamics. Fig. 64 shows that the stochastic model generates very sharp and accurate predictions that model agents movement and punches really well. The agents does not disappear like in Sokoban and actions are smooth. The Controller module successfully learned how to solve the game scoring above 18 points on average across 5 runs. OWM within its latent state of size 16 could be forced to encode two characters positions and hands states which are useful high-level features when deciding on the next action. It is worth pointing out here that similar experiment with such a small latent space did not yield improvement in Sokoban.

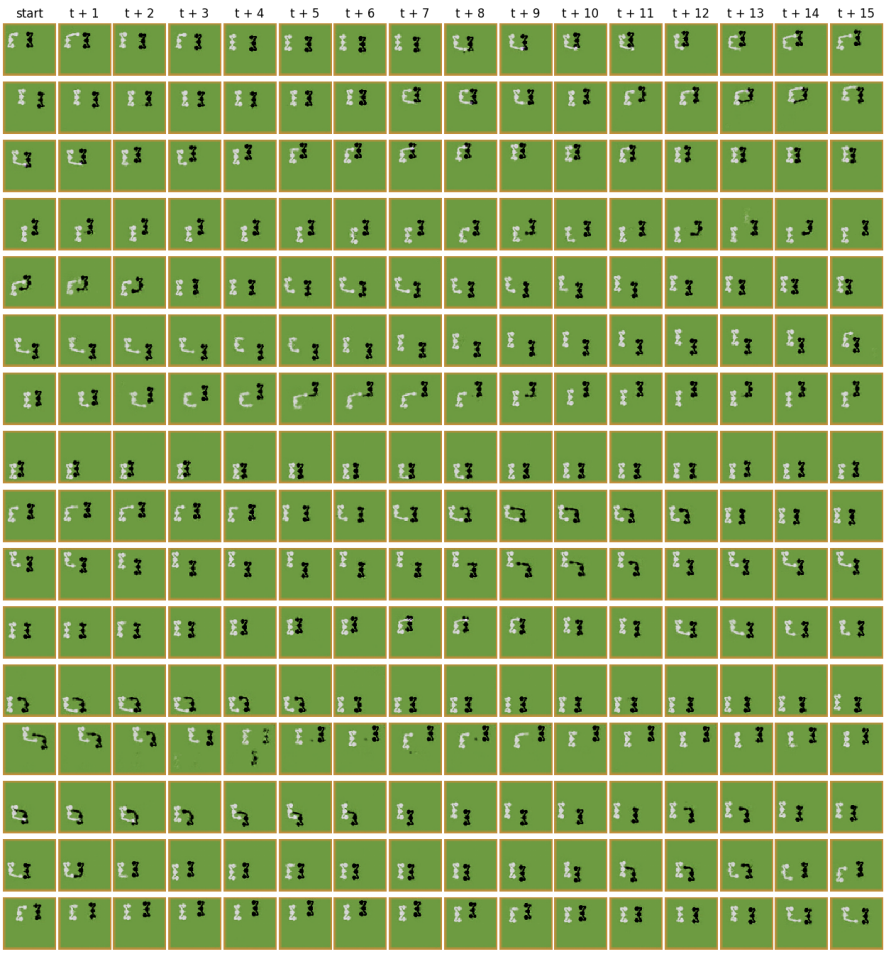


Fig. 64. Qualitative result of the stochastic Memory module training in Boxing. Each row depicts the Memory module rollout in one episode. The first column include original observations from the evaluation dataset from which the rollouts start. The RNN's hidden state was initialized on preceding transitions in each episode. Each subsequent reconstruction was obtained by first predicting the next latent state by the Memory module and then decoding it using the VAE decoder.

6.2.2. Train W+A in the Boxing environment

Despite many attempts and hyper-parameter tuning, the deterministic Memory module was not able to model the Boxing dynamics as good as the stochastic one. This only proves that stochastic nodes are key for the accurate modeling. Fig. 65 shows future predictions which, as can be seen, are imperfect and noisy. The AlphaZero planner training was unstable and it did not train to properly plan using this model. Therefore, it was not able to play the game. Because the AlphaZero planner in its current form can only work with deterministic world models, decision was to abandon this solution and move to the DPN architecture.

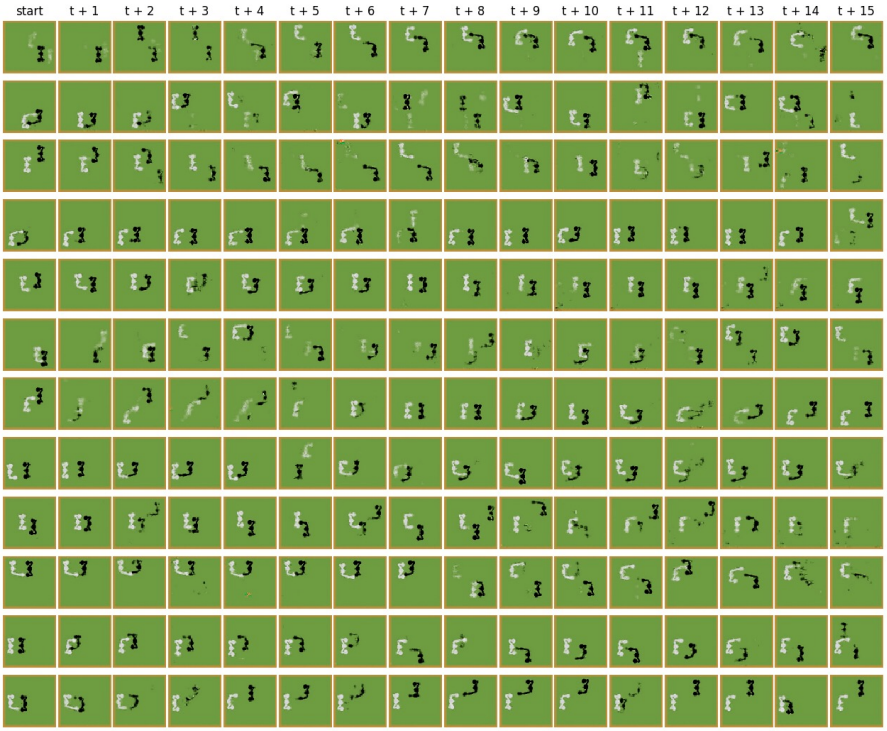


Fig. 65. Qualitative result of the deterministic Memory module training in Boxing. Each row depicts the Memory module rollout in one episode. The first column include original observations from the evaluation dataset from which the rollouts start. The RNN's hidden state was initialized on preceding transitions in each episode. Each subsequent reconstruction was obtained by first predicting the next latent state by the Memory module and then decoding it using the VAE decoder.

6.3. DPN for Sokoban

6.3.1. Train DPN in the Sokoban environment

DPN could not capture Sokoban dynamics like OWM. In the figure below (Fig. 66) future predictions are blurred, multiple agents appear and other artifacts, like changing blocks, are present. Similarly like in OWM case, the decision was to move to Atari games as easier environments to start with.

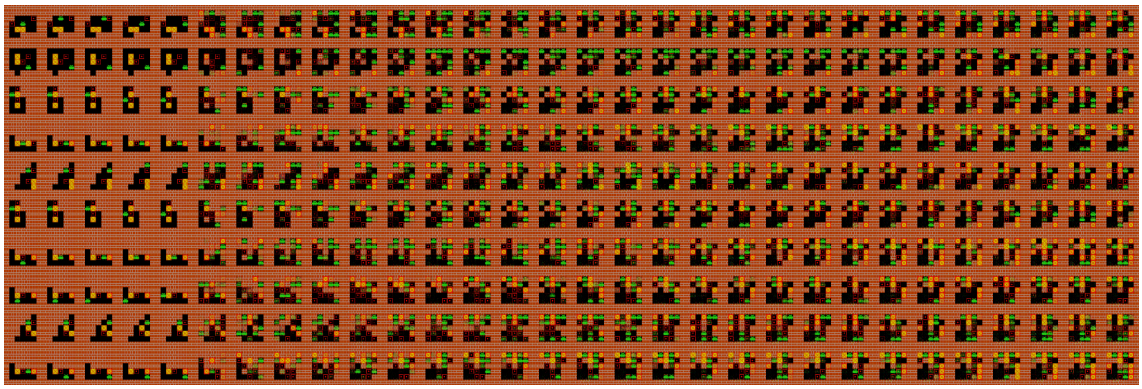


Fig. 66. Qualitative result of the model training in Sokoban. Each row depicts the model rollout in one episode. The first five columns include original consecutive observations from the evaluation dataset from which the rollouts start. The model hidden state was initialized on these transitions. Each subsequent reconstruction was obtained by first predicting the next latent state by the model and then decoding it using the decoder.

6.4. DPN for Atari

This section focuses on tuning DPN to yield high scores, comparable to model-free methods, preserving sample-efficiency of model-based methods.

6.4.1. Train DPN in the Boxing environment

DPN did not work out of the box with the default parameters from the original paper [21]. Fig. 67 shows that future predictions turn into a blurry blob, where it is not possible to distinguish one player from another.

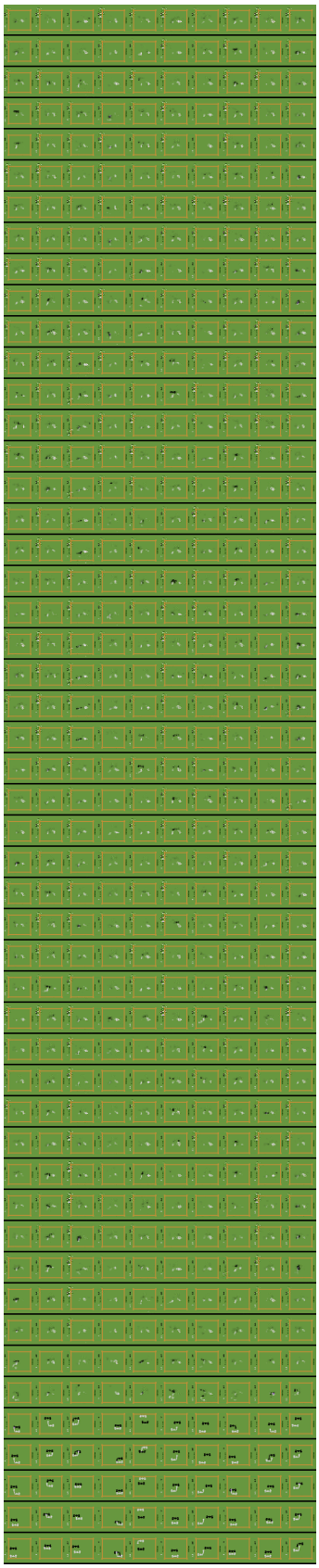


Fig. 67. Qualitative result of the model training in Boxing. Each row depicts the model rollout in one episode. The first five columns include original consecutive observations from the evaluation dataset from which the rollouts start. The model hidden state was initialized on these transitions. Each subsequent reconstruction was obtained by first predicting the next latent state by the model and then decoding it using the decoder.

By default a decoder variance of 1 is used, which means the model explains a lot of variation in the image as random noise. While this leads to more robust representations, it also leads to more blurry images. If the changes in consecutive frames are minor, then the posterior collapses because the model explains everything as observations noise. There are two possible solutions to this issue: one is to increase an action repeat and the other is to try to reduce the decoder variance. These are examined next.

6.4.2. Train DPN in the Boxing environment with the increased action repeat

The action repeat will result in a bigger difference between consecutive frames and thus more signal for the model to learn from, that can not be easily modeled as noise. In practice though, it did not help and even made the agent play worse then a random agent.

6.4.3. Train DPN in the Boxing environment with the lowered decoder variance

The predictions are more blurry with a higher variance of a decoder because the decoder models more observations that differ slightly from the same latent code. This leads to the posterior, or the encoder, explaining more similar observations with the same code. If consecutive frames are very similar, then the posterior collapses and explain them with one code. By lowering the variance of the decoder it becomes more sensitive to small changes in observations and, hence, alleviate the problem of collapsing posterior when observations does not differ much from frame to frame.

To lower a decoder variance, the KL divergence scale is lowered, as explained in the section 2.5. Other reason that lowering the divergence scale can help with collapsing posterior is that it allows the model to absorb more information from its observations by loosening the information bottleneck. On the other hand, it is recommended to keep the divergence scale as high as possible while still allowing for good performance. For instance, when the divergence scale is set to zero, it could learn to become a deterministic autoencoder which reconstruct observations well but is less likely to generalize to state in latent space that the decoder hasn't seen during training.

Random search resulted in the best divergence scale being around 0.03. It was tuned jointly with a free nats parameter which is described in the next section with detailed tuning results in the table 61.

6.4.4. Train DPN in the Boxing environment with increased free nats

The free nats technique, described in the section 2.5, is used too. In case of Boxing, it helped to model boxers moves and actions more accurately. The best free nats turned out to be 12. Fig. 68 shows final result.

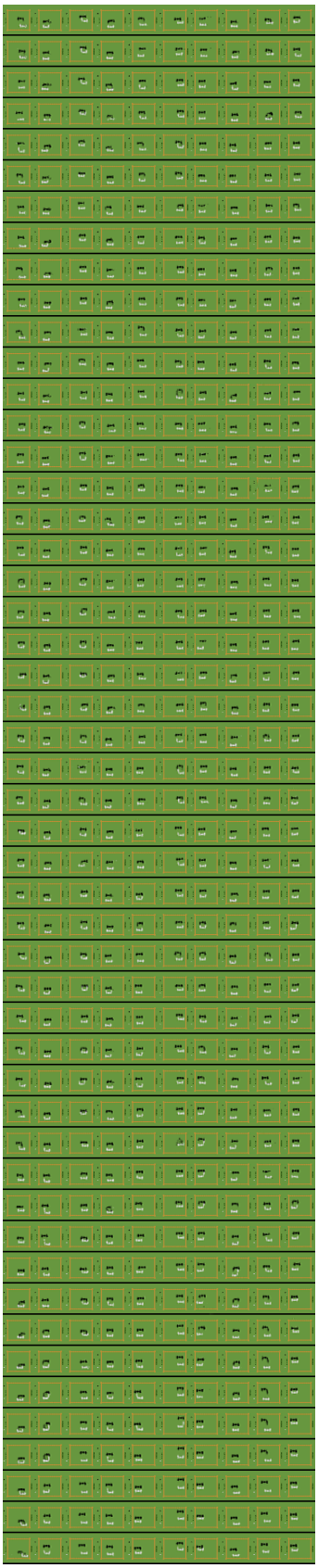


Fig. 68. Qualitative result of the model training in Boxing. Each row depicts the model rollout in one episode. The first five columns include original consecutive observations from the evaluation dataset from which the rollouts start. The model hidden state was initialized on these transitions. Each subsequent reconstruction was obtained by first predicting the next latent state by the model and then decoding it using the decoder.

Table. 61 summarize hyper-parameters tuning experiment. The specialist evaluated noisiness of the model rollouts, the same as in fig. 68, in scale from 0 to 2, where 0 is a little or no noise in the future reconstructions and 2 is a lot of noise in the future reconstructions.

Free nats	Divergence scale	Noisiness
2	1E-04	2
16	1E-04	2
13	1E-04	2
11	2E-04	2
3	2E-04	2
1	3E-04	2
8	3E-04	2
4	4E-04	2
17	1E-03	2
7	2E-03	2
10	2E-03	2
15	2E-03	2
6	2E-03	2
9	3E-03	2
13	4E-03	2
15	4E-03	2
15	5E-03	2
8	5E-03	2
5	6E-03	2
10	1E-02	1
4	2E-02	1
8	2E-02	0
17	3E-02	0
12	1E-01	0

Table 61. PlaNet for Boxing hyper-parameters tuning results.

The lower divergence scale the nosier predictions are. The higher free nats the better, more stable, movement predictions are. Best params turned out to be: the divergence scale around 3E-02 and the free nats around 12.

It achieved final score around 45 after 1 million steps of data collection in the environment. The learning curve is shown in fig. 69.

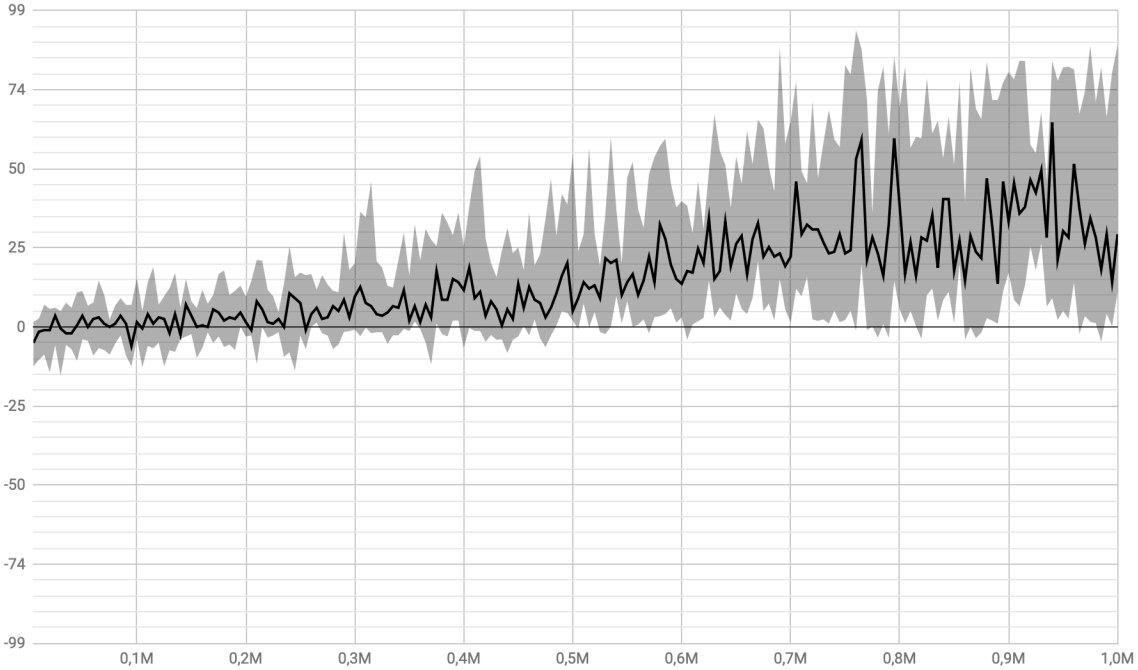


Fig. 69. Boxing learning curve after tuning and with overshooting. The line presents median and the shaded area are percentiles 5 to 95 over 5 training runs. On the Y axis is the environment score and on the X axis is number of steps in the real environment for the data collection.

6.4.5. Train DPN in the Freeway environment

The same random search procedure was applied to the Freeway environment described earlier. Fig. 610 shows final result.

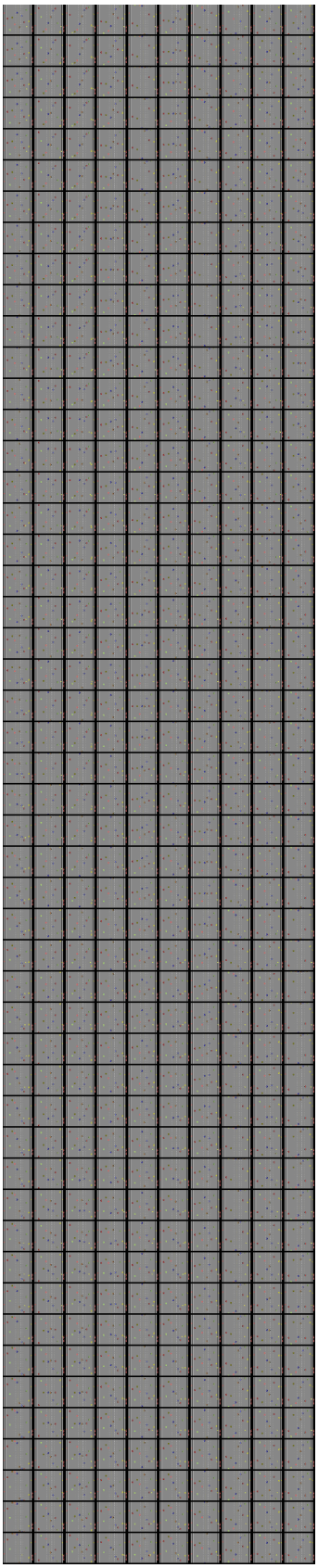


Fig. 610. Qualitative result of the model training in Freeway. Each row depicts the model rollout in one episode. The first five columns include original consecutive observations from the evaluation dataset from which the rollouts start. The model hidden state was initialized on these transitions. Each subsequent reconstruction was obtained by first predicting the next latent state by the model and then decoding it using the decoder.

Table. 62 summarize hyper-parameters tuning experiment conducted in the same way like for Boxing.

Free nats	Divergence scale	Noisiness
9	8E-02	2
6	2E-02	2
5	3E-03	2
4	6E-04	2
4	4E-04	2
4	2E-04	2
7	3E-02	1
6	5E-04	1
4	1E-04	1
2	2E-04	1
5	8E-03	0
2	2E-03	0

Table 62. PlaNet for Freeway hyper-parameters tuning results.

High free nats, above 6, and too low, below 1E-03, or too high, above 9E-03, divergence scale makes future predictions very noisy and blurry. Best parameters chosen were: the divergence scale of 8E-03 and the free nats of 3.

Despite really good future observations prediction, the agent failed to reliably play the game with high score. Results in fig. 611 show that the agent in the early phase of the training, when the model is still untrained and returns mostly random signals, does better then in the late training phase, when the model predicts next states and rewards well. One explanation of this phenomena is that a planner horizon is too short to cover a plan which ends with a positive reward on the other side of the road. The longer planning horizon might help and this is explored in the next experiment.

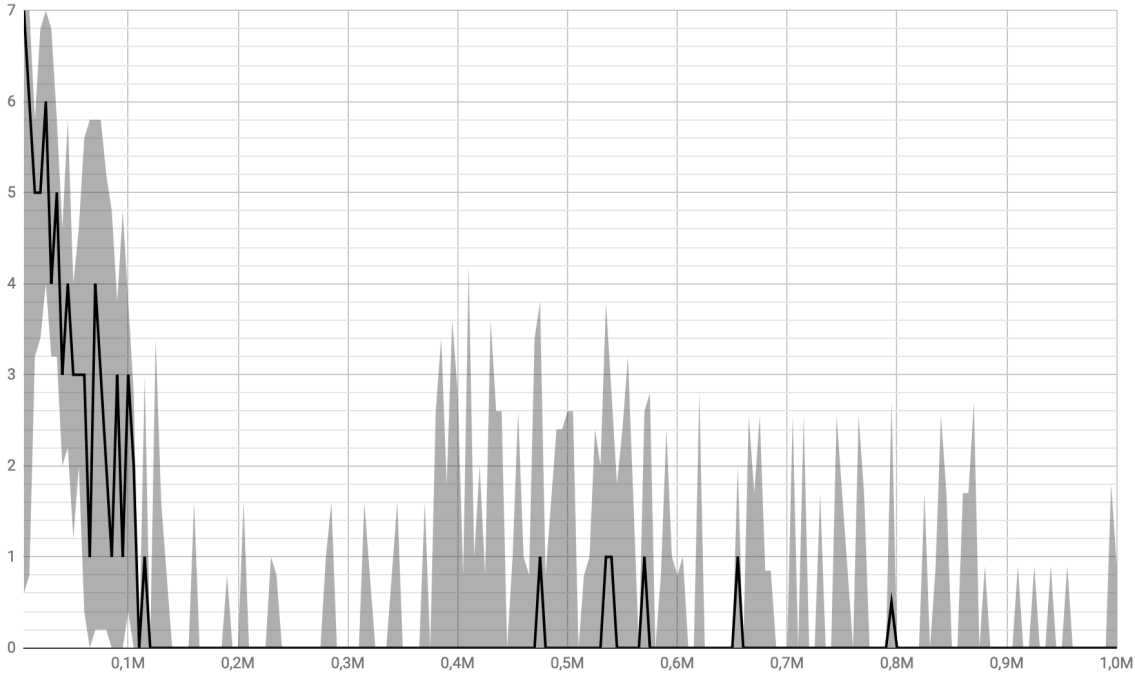


Fig. 611. Freeway learning curve after tuning and with overshooting. The line presents median and the shaded area are percentiles 5 to 95 over 5 training runs. On the Y axis is the environment score and on the X axis is number of steps in the real environment for the data collection.

6.4.6. Train DPN in the Freeway environment with a longer planning horizon

Two experiments were run: first for planning horizon of 25 and second for planning horizon of 50. Both failed to improve the agent score. After many attempts, the Freeway environment was left unsolved.

6.4.7. Train DPN in the Boxing environment without overshooting

The authors in the final version of the original PlaNet paper [21] found out, that disabling overshooting can increase performance for their RSSM architecture. It was put into the test and final results are shown in fig. 612. It helped increase performance of DPN tuned for Boxing.

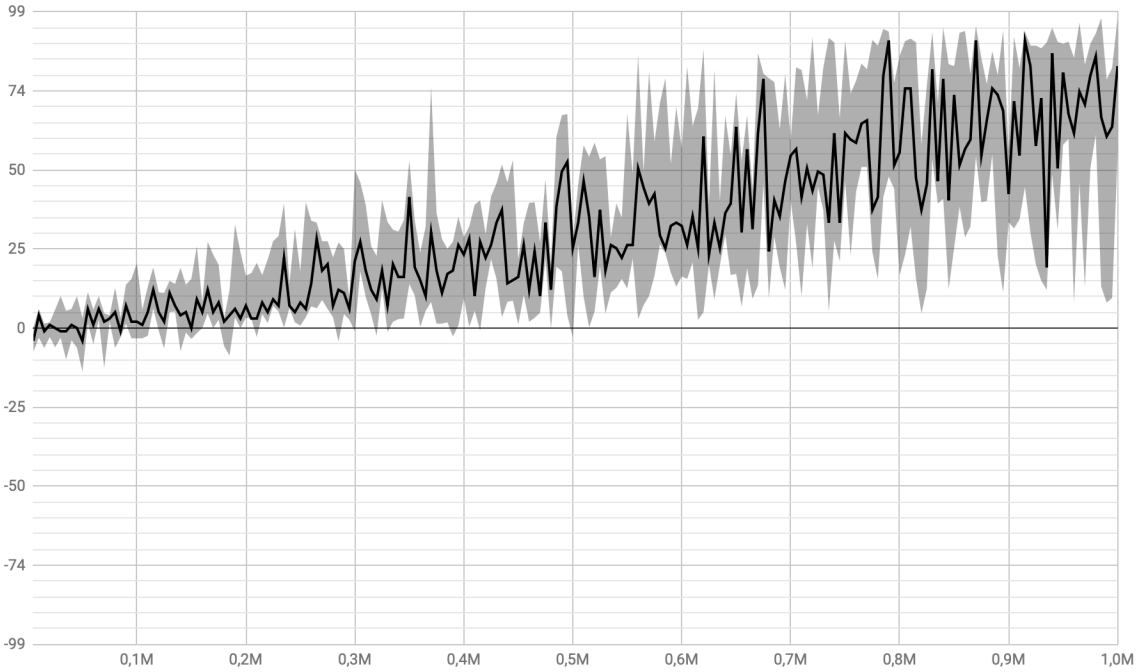


Fig. 612. Boxing learning curve after tuning and without overshooting. The line presents median and the shaded area are percentiles 5 to 95 over 5 training runs. On the Y axis is the environment score and on the X axis is number of steps in the real environment for the data collection.

This is the best and final solution which gets compared to strong model-based baseline SimPLe [30] and model-free baselines Rainbow [23] and PPO [49]. Score of randomly issuing actions in the environment is also reported. Comparison is done in low data regime of 100K, 500K and 1M interactions with the real environment. The baseline results are taken from the SimPLe paper [30].

Algo.	100K	500K	1M
Ours	6,2 (10,7)	35,2 (8,3)	78,2 (19,1)
SimPLe	9,1 (8,8)	NDA	NDA
PPO	-3,9 (6,4)	3,5 (3,5)	19,6 (20,9)
Rainbow	0,9 (1,7)	58,2 (16,5)	80,3 (5,6)
Random		0,3	

Table 63. Mean scores and standard deviations (in brackets) over five training runs.

Tuned DPN architecture without overshooting is better than PPO in every data setting and from Rainbow in low data regime of 100K interactions with the real environment. The performance is also comparable with SimPLe in all data settings and with Rainbow in 500K and 1M data settings. Therefore, it is clear that the DPN architecture is sample-efficient without sacrificing final performance of the agent.

7. CONCLUSION

REFERENCES

[1] Gabriel Barth-Maron et al. “Distributed Distributional Deterministic Policy Gradients”. In: *arXiv e-prints* (2018). arXiv: 1804.08617.

[2] Marc G. Bellemare et al. “The Arcade Learning Environment: An Evaluation Platform for General Agents”. In: *arXiv e-prints* (2012). arXiv: 1207.4708.

[3] Zdravko I. Botev et al. “Chapter 3 - The Cross-Entropy Method for Optimization”. In: *Handbook of Statistics*. Vol. 31. Handbook of Statistics. Elsevier, 2013, pp. 35 –59. DOI: https://doi.org/10.1016/B978-0-444-53859-8.00003-5.

[4] Greg Brockman et al. *OpenAI Gym*. 2016. arXiv: 1606.01540.

[5] Lars Buesing et al. “Learning and Querying Fast Generative Models for Reinforcement Learning”. In: *arXiv e-prints* (2018). arXiv: 1802.03006.

[6] Xi Chen et al. “Variational Lossy Autoencoder”. In: *arXiv e-prints* (2016). arXiv: 1611.02731.

[7] Silvia Chiappa et al. “Recurrent Environment Simulators”. In: *arXiv e-prints* (2017). arXiv: 1704.02254.

[8] Kyunghyun Cho et al. “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: *arXiv e-prints* (2014). arXiv: 1406.1078.

[9] François Chollet et al. *Keras*. https://keras.io. 2015.

[10] Jack Clark and Dario Amodei. *Faulty Reward Functions in the Wild*. https://openai.com/blog/faulty-reward-functions/. Accessed: 2019-05-25.

[11] Dorit Dor and Uri Zwick. “SOKOBAN and other motion planning problems”. In: *Computational Geometry* (1999). ISSN: 0925-7721. DOI: https://doi.org/10.1016/S0925-7721(99)00017-6. URL: http://www.sciencedirect.com/science/article/pii/S0925772199000176.

[12] Richard Evans et al. “De novo structure prediction with deep-learning based scoring”. Dec. 2018. URL: https://deepmind.com/blog/alphafold/.

[13] Vladimir Feinberg et al. “Model-Based Value Estimation for Efficient Model-Free Reinforcement Learning”. In: *arXiv e-prints* (2018). arXiv: 1803.00101.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[15] Alex Graves. “Generating Sequences With Recurrent Neural Networks”. In: *arXiv e-prints* (2013). arXiv: 1308.0850.

[16] Karol Gregor et al. “Temporal Difference Variational Auto-Encoder”. In: *arXiv e-prints* (2018). arXiv: 1806.03107.

[17] David Ha and Douglas Eck. “A Neural Representation of Sketch Drawings”. In: *arXiv e-prints* (2017). arXiv: 1704.03477.

[18] David Ha and Jürgen Schmidhuber. “World Models”. In: *arXiv e-prints* (2018). arXiv: 1803.10122.

[19] Danijar Hafner, James Davidson, and Vincent Vanhoucke. “TensorFlow Agents: Efficient Batched Reinforcement Learning in TensorFlow”. In: *arXiv e-prints* (2017). arXiv: 1709.02878. URL: http://arxiv.org/abs/1709.02878.

[20] Danijar Hafner et al. *Deep Planning Network*. https://github.com/google-research/planet. 2019.

[21] Danijar Hafner et al. “Learning Latent Dynamics for Planning from Pixels”. In: *arXiv e-prints* (2018). arXiv: 1811.04551v4.

[22] Nikolaus Hansen. “The CMA Evolution Strategy: A Tutorial”. In: *arXiv e-prints* (2016). arXiv: 1604.00772.

[23] Matteo Hessel et al. “Rainbow: Combining Improvements in Deep Reinforcement Learning”. In: *arXiv e-prints* (2017). arXiv: 1710.02298.

[24] Irina Higgins et al. “beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework”. In: *ICLR* (2017).

[25] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Comput.* (1997). ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[26] Max Jaderberg et al. “Reinforcement Learning with Unsupervised Auxiliary Tasks”. In: *arXiv e-prints* (2016). arXiv: 1611.05397.

[27] Se Won Jang, Jesik Min, and Chan Lee. “Reinforcement Car Racing with A3C”. In: (2017). URL: https://www.scribd.com/document/358019044/Reinforcement-Car-Racing-with-A3C.

[28] Piotr Januszewski, Grzegorz Beringer, and Mateusz Jablonski. *HumbleRL - Straightforward reinforcement learning Python framework*. 2019. URL: https://github.com/piojanu/humblerl.

[29] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. “Stochastic Backpropagation and Approximate Inference in Deep Generative Models”. In: *arXiv e-prints* (2014). arXiv: 1401.4082.

[30] Lukasz Kaiser et al. “Model-Based Reinforcement Learning for Atari”. In: *arXiv e-prints* (2019). arXiv: 1903.00374.

[31] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *arXiv e-prints* (2014). arXiv: 1412.6980.

[32] Diederik P. Kingma and Prafulla Dhariwal. “Glow: Generative Flow with Invertible 1x1 Convolutions”. In: *arXiv e-prints* (2018). arXiv: 1807.03039.

[33] Levente Kocsis and Csaba Szepesvári. “Bandit Based Monte-Carlo Planning”. In: *Machine Learning: ECML 2006*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Springer Berlin Heidelberg, 2006.

[34] Rahul G. Krishnan, Uri Shalit, and David Sontag. “Deep Kalman Filters”. In: *arXiv e-prints* (2015). arXiv: 1511.05121.

[35] Felix Leibfried, Nate Kushman, and Katja Hofmann. “A Deep Learning Approach for Joint Video Frame and Reward Prediction in Atari Games”. In: *arXiv e-prints* (2016). arXiv: 1611.07078.

[36] Marlos C. Machado et al. “Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents”. In: *arXiv e-prints* (2017). arXiv: 1709.06009.

[37] Volodymyr Mnih et al. “Asynchronous Methods for Deep Reinforcement Learning”. In: *arXiv e-prints* (2016). arXiv: 1602.01783.

[38] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. URL: http://dx.doi.org/10.1038/nature14236.

[39] S Mor-Yosef et al. “Ranking the risk factors for cesarean: Logistic regression analysis of a nationwide study”. In: *Obstetrics and gynecology* 75 (July 1990), pp. 944–7.

[40] Y. Naddaf and University of Alberta. Department of Computing Science. *Game-independent AI Agents for Playing Atari 2600 Console Games*. University of Alberta, 2010. URL: https://books.google.pl/books?id=c85vnQAACAAJ.

[41] Vinod Nair and Geoffrey E. Hinton. “Rectified Linear Units Improve Restricted Boltzmann Machines”. In: ICML’10 (2010), pp. 807–814. URL: http://dl.acm.org/citation.cfm?id=3104322.3104425.

[42] Junhyuk Oh, Satinder Singh, and Honglak Lee. “Value Prediction Network”. In: *arXiv e-prints* (2017). arXiv: 1707.03497.

[43] Adam Paszke et al. “Automatic Differentiation in PyTorch”. In: *NIPS Autodiff Workshop*. 2017.

[44] Paweł Rościszewski et al. *TensorHive*. https://github.com/roscisz/TensorHive. 2018.

[45] Sumit Saha. *A Comprehensive Guide to Convolutional Neural Networks*. [Online; accessed 25-June-2019]. 2018. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[46] Sumit Saha. *Understanding LSTM Networks*. [Online; accessed 25-June-2019]. 2015. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[47] Tim Salimans et al. “Evolution Strategies as a Scalable Alternative to Reinforcement Learning”. In: *arXiv e-prints* (2017). arXiv: 1703.03864.

[48] Max-Philipp B. Schrader. *gym-sokoban*. https://github.com/mpSchrader/gym-sokoban. 2018.

[49] John Schulman et al. “Proximal Policy Optimization Algorithms”. In: *arXiv e-prints* (2017). arXiv: 1707.06347.

[50] David Silver, Richard S. Sutton, and Martin Müller. “Temporal-difference search in computer Go”. In: *Machine Learning* 87.2 (2012), pp. 183–219. ISSN: 1573-0565. DOI: 10.1007/s10994-012-5280-0. URL: https://doi.org/10.1007/s10994-012-5280-0.

[51] David Silver et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science* 362 (2018).

[52] David Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* 550 (2017). URL: http://dx.doi.org/10.1038/nature24270.

[53] Ilya Sutskever et al. “On the importance of initialization and momentum in deep learning”. In: (2013). URL: http://proceedings.mlr.press/v28/sutskever13.html.

[54] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction 2nd*. The MIT Press, 2018. ISBN: 9780262039246.

[55] Erik Talvitie. “Agnostic System Identification for Monte Carlo Planning”. In: AAAI’15 (2015), pp. 2986–2992. URL: http://dl.acm.org/citation.cfm?id=2888116.2888132.

[56] Erik Talvitie. “Model Regularization for Stable Sample Rollouts”. In: UAI’14 (2014), pp. 780–789. URL: http://dl.acm.org/citation.cfm?id=3020751.3020832.

[57] Théophane Weber et al. “Imagination-Augmented Agents for Deep Reinforcement Learning”. In: *arXiv e-prints* (2017). arXiv: 1707.06203.

[58] Wikipedia contributors. *Ms. Pac-Man — Wikipedia, The Free Encyclopedia*. [Online; accessed 21-June-2019]. 2019. URL: https://en.wikipedia.org/w/index.php?title=Ms._Pac-Man&oldid=900278576.

LIST OF FIGURES

LIST OF TABLES