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

[What is RL and model-free RL...]* Reinforcement learning, a subfield of artificial intelligence (AI), formalise rather the most obvious and common among animals learning strategy. It is learning how to achieve predefined goals through interaction with an environment [42]. Progress has been made in developing capable agents for numerous domains using deep neural networks in conjunction with model-free reinforcement learning [19][29][37], where raw observations directly map to agent's actions. However, current state-of-the-art approaches are very sample inefficient, they sometimes require tens or even hundreds of millions of interactions with the environment [28], and lack the behavioural flexibility of human intelligence, hence the resulting policies poorly generalize to novel tasks in the same environment.

[Model-based RL with its benefits...]* The other branch of reinforcement learning algorithms, called model-based reinforcement learning, aims to address these shortcomings by endowing agents with a model of the world. There are many ways of using the model: one can use the model for data augmentation for model-free methods [11], some methods use the model as the imagined environment to learn model-free policy in it [14], other methods focus on simulation-based search using the model [39] and there are even methods that integrate model-free and model-based approaches [45]. The model allows the agent to simulate an outcome of an action taken in a given state. The main upside of this 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. Agents can then distill the results from planning ahead into a policy. Even if the model needs to be synthesized from past real experience first it can exploit additional unsupervised learning signals, like rewards function modeling or future observations prediction [22], thus it results in a substantial improvement in sample efficiency over model-free methods. Furthermore, the same model can be used by the agent to complete other tasks in the same environment [45]. It gives AI hint of human intelligence flexibility and versatility.

[Planning vs. learning differences/similarities...]* Learning is different from planning. In the former the agent samples episodes of real experience through interaction with an environment and updates its policy or states' value estimates based on them. In the latter the agent also updates its policy or states' value estimates, but this time based on simulated experience gained through evaluation of a model. It is worth noting the symmetry which yields one important implication: algorithms for reinforcement learning can also become algorithms for planning, simply by substituting simulated experience in place of real experience. Moreover, planning carries the promise of increasing performance just by increasing the computational budget for searching for good actions [40]. Model-free methods to improve their performance need more interactions with a real environment and hence scale in amount of data not amount of computation, which very often is much cheaper than collecting more data.

[Really briefly about what is problem of learning world dynamics of POMDP...]* 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. One of them is bias in the model that can be exploited by the agent [14], resulting in an agent which performs well with respect to the learned model, but behaves sub-optimally in the real environment. Different challenge also comes from fundamental downside of function approximation, it result in models that are inherently imperfect. The performance of agents employing common planning methods usually suffer from seemingly minor model errors [43]. Those errors compound during planning, causing more and more inaccurate predictions the further horizon of a plan. [44]

[Long-term ambitious goal...]* 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 [39], published AlphaFold [10], 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, and do this over 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 accelerated computer simulation, and often need a human assistance. To apply sample efficient model-based systems, that can generalize their knowledge, accurate learned models and robust planning algorithms are needed.

[Explain your topic...]* The aim of this work is to derive from previous work on model-learning in complex high-dimensional decision making problems [5][27][4][17] and apply them to plan in Atari 2600 games, a platform for evaluating general competency in artificial intelligence [28]. Those methods proved to train accurate models, at least in short horizon, and should open a path for application of simulation-based search planning algorithms like TD-Search [38] or AlphaZero [39] to complex planning problems in environments with discrete state-action spaces and without access to a ground-truth model. This should allow for improvement in data efficiency and generalization *[We don't test generalization, maybe it should be omitted then?]* without loss in performance. This work focuses on three benchmarks: an arcade game with dense rewards Boxing, a challenging environment with sparse rewards Freeway and a complex puzzle environment MsPacman.

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 of an environment is unknown. The environment is perceived through observations that provide only partial information about the state. A good example are Atari games. Individual frames often doesn't provide full information about the game's state which is held in the game's RAM.

In this work following definition of POMDP 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 $P_s = p(O_t | S_t = s)$. A fixed initial state s_0 is assumed. In episodic POMDPs, which this work considers, an environment 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 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 environment 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 and then following policy π where, again, the expectation is over the distributions of the environment and the policy. An optimal value function is the unique value function that maximises the value of every state, $V^*(s) = \max_\pi V_\pi(s), \forall_{s \in S}$ and $Q^*(s, a) = \max_\pi Q_\pi(s, a), \forall_{s \in S, a \in A}$. An optimal policy $\pi^*(s, a)$ is a policy that maximises the action value function for every state in the POMDP, $\pi^*(s, a) = \operatorname{argmax}_a Q^*(s, a)$.

POMDPs are a mathematically idealized form of the reinforcement learning problem for which precise theoretical statements can be made. In reinforcement learning the dynamics, the observations distribution and the reward function are hidden behind an environment. Consequently, we can not directly use them for planning, but we can learn them through interaction with the environment.

2.2. Reinforcement Learning

Reinforcement learning (RL) is learning what to do, how to map situations to actions, so as to maximize a numerical reward signal. [42] This mapping is called a policy π. RL consists of an agent that, in order to learn a good policy, acts in an environment. The environment provides a response to each agent's action a that is interpreted and fed back to the agent. Reward r is used as a reinforcing signal and state s is used to condition agent's decisions. Fig. 21 explains it in the diagram. Each action-response-interpretation sequence is called a step or a transition. Multiple steps form an episode. The episode finishes in a terminal state s_T and 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.

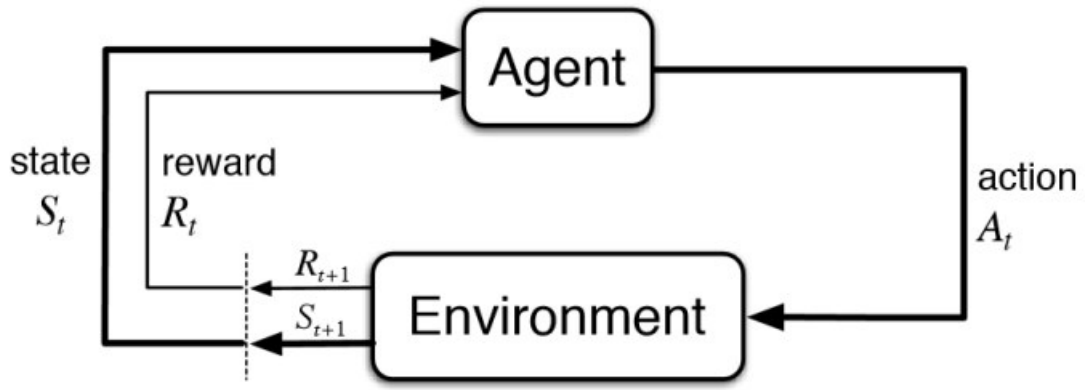

Fig. 21. Reinforcement Learning [42]

[TODO: Describe here General Policy Iteration, Monte-Carlo Control, TD-Learning, ... what else? Rather than guessing what needs to be described, continue work and see what needs more attention.]

2.3. 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 [31]. 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 we would like to write a program to detect cats in photographs. We know that cats are furry and have whiskers, so we 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 we take 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. 22 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.

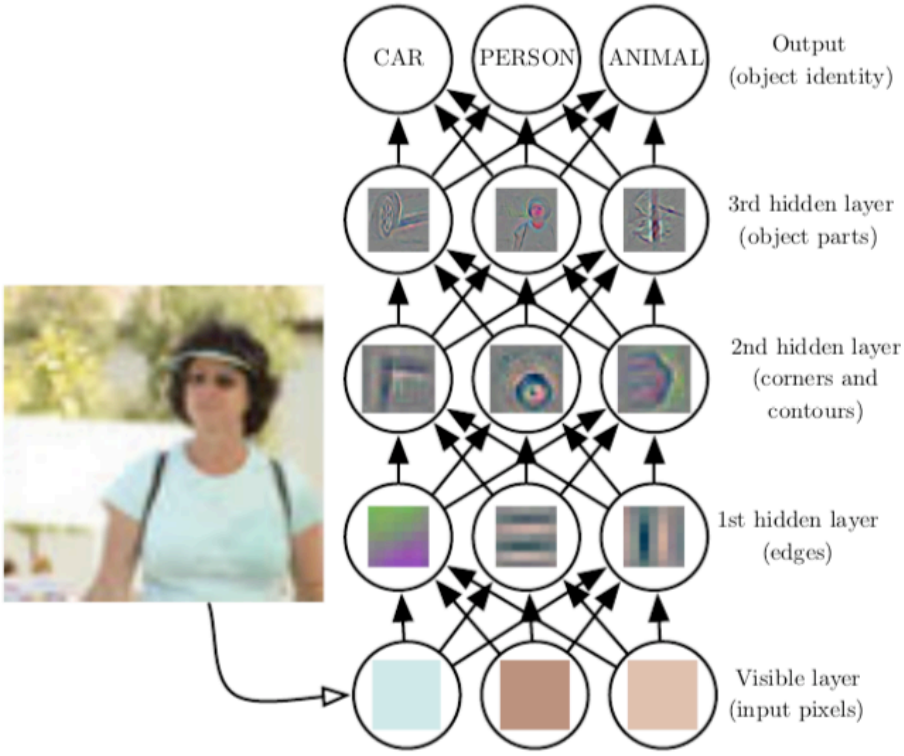

Fig. 22. Deep Learning [12]

The fundamental example of a deep learning model is a feedforward deep network or multilayer perceptron (MLP). A multilayer perceptron is just a mathematical function mapping some set of input values to output values. The function is formed by composing many simpler functions, called perceptrons, each providing a new representation of its input to next functions. Fig. 23 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.

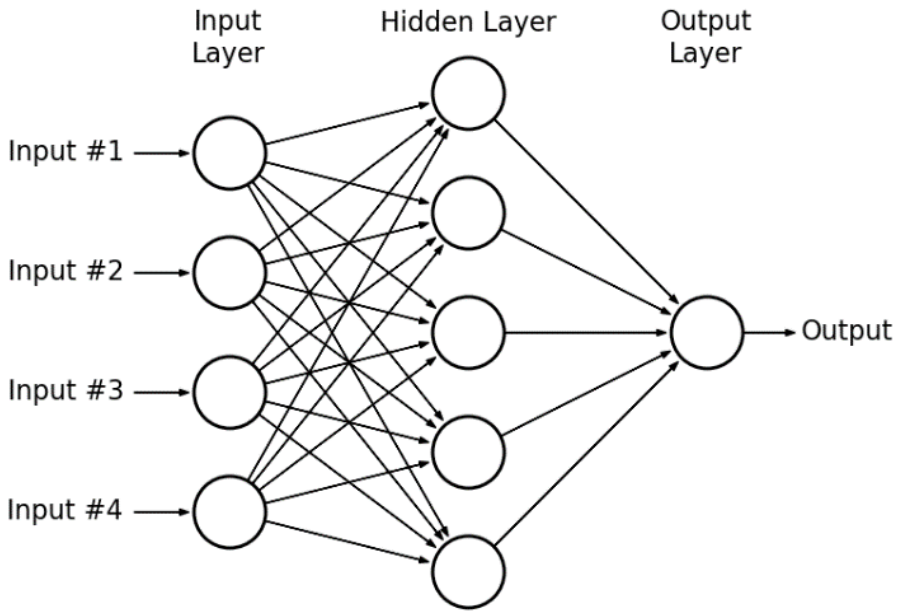

Fig. 23. Multilayer perceptron

[TODO: One word about RNNs, Conv. and other used layers here.] [TODO: Also write here about backprop.]

2.4. Variational Inference

[TODO: Describe VAEs in structured POMDPs: what are zero step, one step and open loop predicitions, what is latent state/code, etc.]

3. RELATED WORK

3.1. Model learning

3.1.1. World Models

In World Models[14] paper, the authors explore the idea of using large and highly expressive neural networks, that can learn rich spatial and temporal representation of data, and applying them 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 accomplish their goal, they decompose the problem of an agent training into two stages: they first train a generative neural network to learn a model of the agent's world in an unsupervised manner. Thereafter, by using a compressed spatial and temporal representation of the environment extracted from the world model as inputs to the agent, they train a linear model to learn 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 scarifying capacity and expressiveness via the larger world model.

Their solution consists of three components: Vision (V) for encoding the spatial information, Memory (M) for encoding the temporal information and Controller (C) which represents the agent's policy. Fig. 31 depicts a flow diagram of the agent's model.

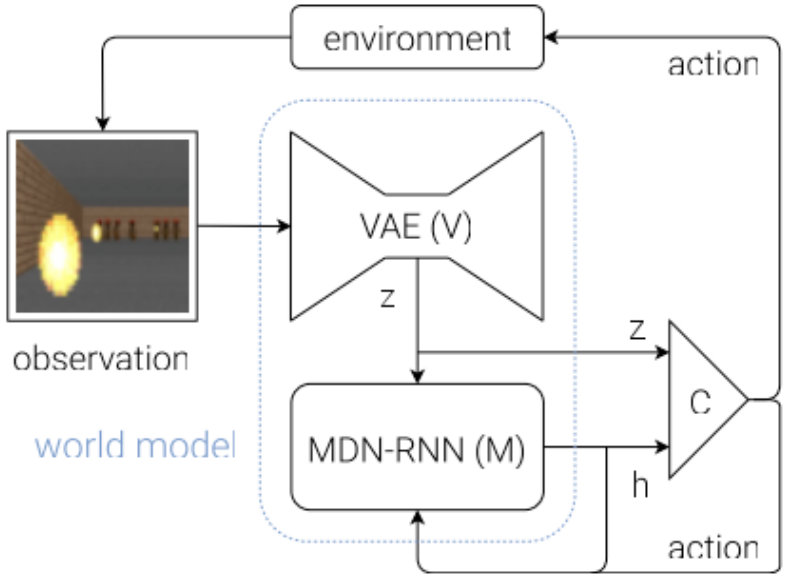

Fig. 31. Flow diagram of the agent's model[14]. The raw observation is first processed by V at each time step t to produce z_t. The input into C is this latent vector z_t concatenated with M's hidden state h_t at each time step. C will then output an action vector a_t and will affect the environment. M will then take the current z_t and action a_t as an input to update its own hidden state to produce h_{t+1} to be used at time $t+1$.

The environment provides the agent with high dimensional visual observation at each time step. The essential task of the Vision model is to encode this high dimensional observation into a low dimensional latent state. To do this, Vision is implemented as Variational Autoencoder[24]. It is trained in an unsupervised manner on randomly generated experience from the environment.

Since many complex environment are partially observable, the visual observation at each time step, and hence the latent state, doesn't include full information about the current situation in the environment. To acquire full knowledge, the agent needs to encode what happens over time. This is the role of the Memory model. It is implemented as Recurrent Neural Network[21] (RNN) and trained on the same data as Vision to predict the future latent state that Vision is expected to produce. Because many environments are stochastic in nature, the RNN is trained to output a probability density of the next latent state approximated as a mixture of Gaussian distribution - in literature, this approach is known as Mixture Density Network combined with a RNN (MDN-RNN)[13]. More specifically, the RNN will model $P(z_{t+1}|a_t, z_t, h_t)$, where z_{t+1} is the output next latent state, a_t is the action taken at time t, z_t is the latent state of the current time step t and h_t is the hidden state of the RNN that encodes past information made available to the agent from the beginning of the episode until the time step t.

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 state z_t and hidden state h_t at the time step t directly to the action a_t at that time step: $a_t = W[z_t h_t] + b$, where W and b are the weight matrix and bias vector of that model. 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 (V and M). 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. Shift in the agent's complexity towards the world model 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[35].

Their solution was able to solve an OpenAI Gym's CarRacing environment, which is the continuous-control, top-down racing task. It is the first known solution to achieve the score required to solve this task. In the process, the Memory model have learned to simulate the original environment. The authors show that the learned policy can function inside of the imagined environment of CarRacing, that is simulated by Memory. In the second experiment, they show that the agent is able to learn from imagined experience, produced by its Memory, and successfully transfer this policy back to the actual environment of VizDoom (see fig. 32). This result indicates that the 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. *[Should we expand on that in here? Or rather place for that is in "Planning with learned model" chapter where I describe design of my solution?]*

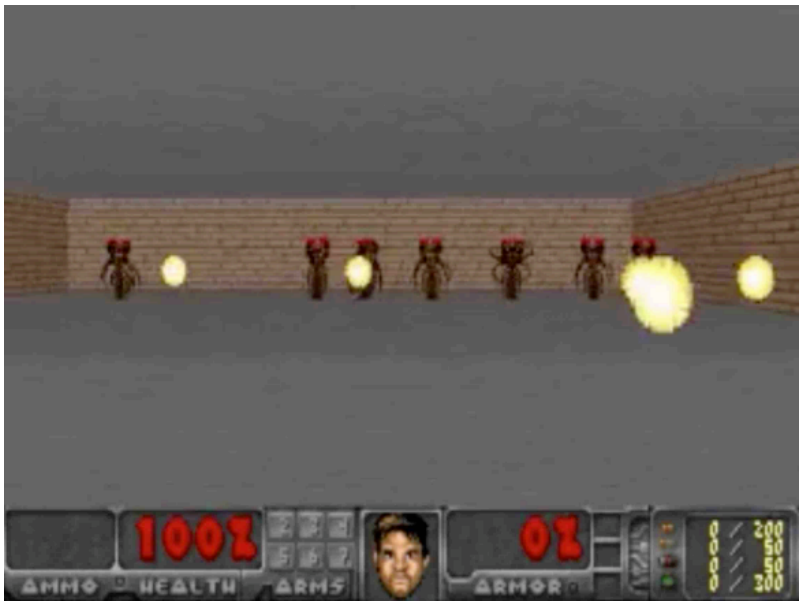

Fig. 32. VizDoom: the agent must learn to avoid fireballs shot by monsters from the other side of the room with the sole intent of killing the agent[14].

4. PLANNING WITH LEARNED MODEL

In this section 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. Such models substantially reduce the amount of computation required to make predictions, as future states can be represented much more compactly. In order to increase model accuracy and robustness, all models explicitly model uncertainty in state transitions using stochastic nodes [4]. The goal stays the same: 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 ideas towards this goal.
Before all of that, 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 [23] 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 the world represented by the Environment class. The Environment provides methods for resetting, taking steps, rendering and getting information about the world. 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.

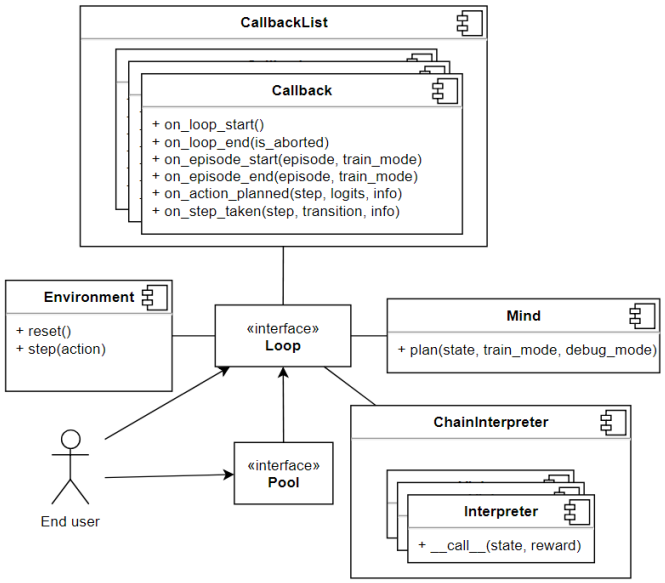

Fig. 41. HumbleRL architecture

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.

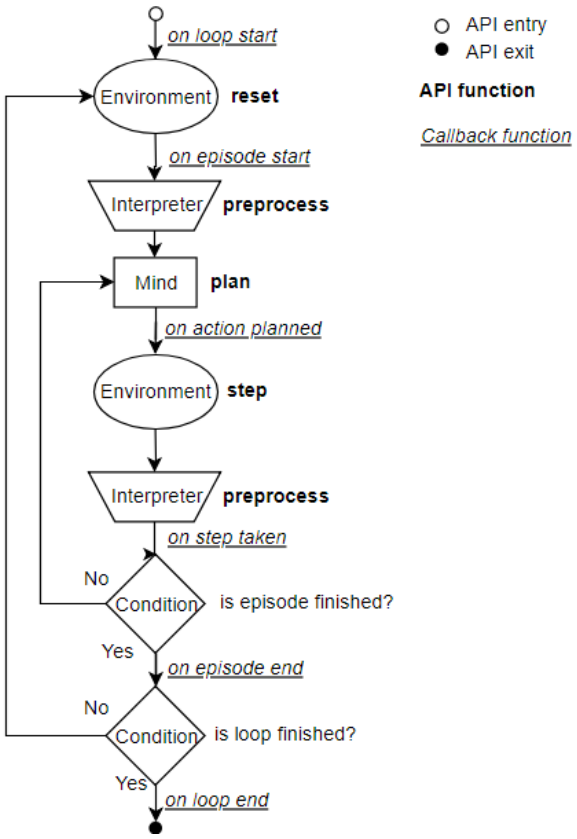

Fig. 42. HumbleRL loop function overview

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

4.2. Original World Models

This section describes the original World Models [14] algorithm which uses a learned model to create abstract task representation which, in turn, is used to solve the task by a lightweight controller. This architecture will be coupled with the AlphaZero which uses the learned model to simulate experience in the next section.

4.2.1. World model

A simple model inspired by human cognitive system is used. In this model, an agent has a visual sensory module that compresses observations into a small representative code. It also has a memory module that makes predictions about future codes based on historical information. Finally, the agent has a decision-making component, called the controller module, that decides what actions to take based only on the representations created by its vision and memory modules. This architecture allows for training of a large neural network to tackle RL tasks by dividing the agent into a large world model and a small controller model. First a large neural network learns to model the agent's world in an unsupervised manner, and only then the smaller controller focuses

on the credit assignment problem on a smaller search space of controller's parameters, while not sacrificing capacity and expressiveness via the larger world model.

An environment provides the agent with a high dimensional input observation at each time step. It is a 2D image frame that is part of a video sequence. The vision module role, as already mentioned, is to learn an abstract representation of each observed input frame. It uses a simple Variational Autoencoder [24] and, as described in details in theoretical background chapter, is trained to encode each frame into low dimensional latent vector by minimizing the difference between a given frame and the reconstructed version of the frame produced by the decoder.
The Memory module purpose is to compress the information what happens over time in its hidden state and enable simulation of the environment. To do this, it is trained to model environment's dynamics, predicting a future latent state from history of previous latent states, as a mixture of Gaussians. It models latent states with probability distribution to model uncertainty in the environment, but also create more robust environment's representation [4]. Uncertainty can originate not only from fundamental stochastic nature of the environment, but also partial observability.
The model 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 [13].

Figure 43 depicts the world model, the Vision and Memory modules interconnection, in graphical form. More specifically, the world model components are:

- Deterministic hidden state model: $h_t = f(h_{t-1}, z_t, a_t)$
- Stochastic latent state model: $z_{t+1} \sim p(z_{t+1}|h_t) = \sum_c \pi_c(h_t) p(z_{t+1}|h_t, c)$
- Observation model (decoder): $o_t \sim p(o_t|z_t)$
- Approximate state posterior (encoder): $z_t \sim q(z_t|o_t)$

where o, z and a are high-dimensional observations, latent states and actions respectively. $f(h_{t-1}, z_t, a_t)$, the hidden state model, is implemented as a recurrent neural network and h_t is its hidden state. The latent 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. 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 past observations and actions, where $q(s_t|h_t, o_t)$ is a diagonal Gaussian with mean and variance parameterised by a convolutional neural network followed by a feed-forward neural network.

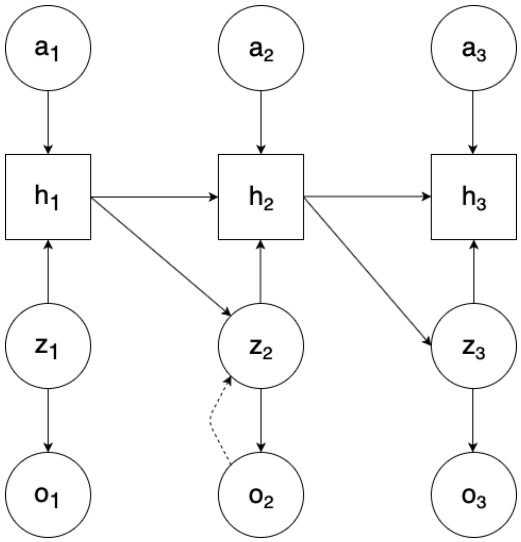

Fig. 43. World Models probabilistic graphical model: solid arrows describe the predictive model, dotted arrow describes the inference model, stochastic nodes are circles and squares depict deterministic nodes.

4.2.2. 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 world 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 state 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, the fact that is described below, it doesn't impair exploration which is done on parameters level.

4.2.3. 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 en-

vironment. This dataset is used to train the Vision module. Next, it is used to process each frame into its latent state to prepare a dataset for the Memory module training, which works entirely in the latent space.

The Controller module is trained using evolutional strategy, described below in the implementation details section, which rollouts the environment in each epoch to evaluate population.

4.2.4. 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 doesn't 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 and default 4 action repeat from OpenAI Gym [3] is used as common in reinforcement learning [30] to reduce the planning horizon and provide a clearer learning signal to the model.

4.2.5. 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 [21] 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 one layer feed-forward neural networks with linear activations. The Vision module neural networks are convolutional and deconvolutional neural networks shown in figure 44 where latent state size, μ and σ vectors dimensionality, equals 16.
For Vision training Keras [7] framework is used to adjust the parameters by the stochastic gradient descent on a standard Evidence Lower Bound loss. For Memory training PyTorch [34] 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 [25] 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 [25] too with a learning rate of 10^{-3} and $\epsilon = 10^{-8}$ on batches of 128 sequence chunks of length 1000.

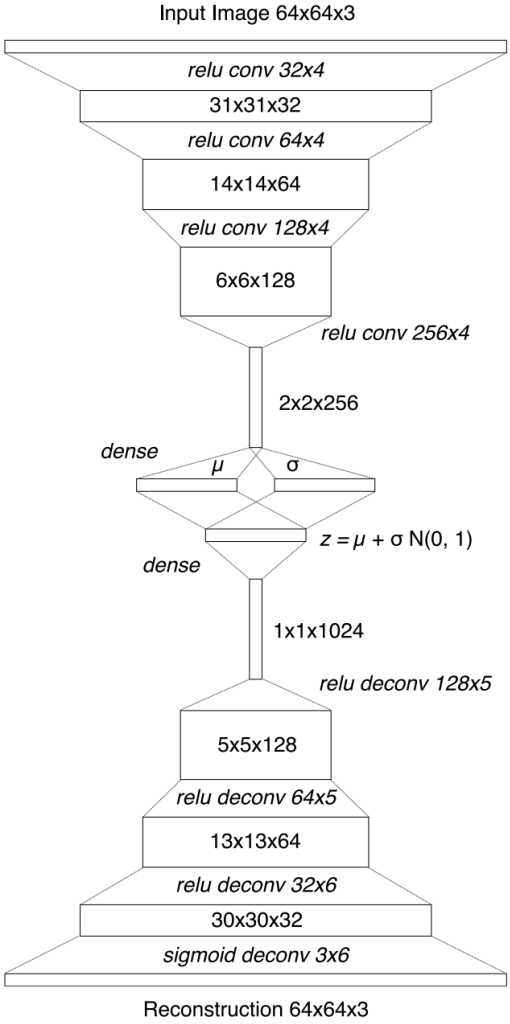

Fig. 44. World Models VAE neural network architecture [14]

The Controller module gets a feature vector as its input, consisting of latent state 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) [18] 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.

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

4.3. World Models and AlphaZero

World Models' agent [14] 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 search algorithm. This is different application of the model than in the original paper, where only future latent states

and done flag are predicted, and therefore the world model needs to be extend with a reward predictor and the controller is replaced by AlphaZero.

4.3.1. World model

The world model part, Vision and Memory modules, architecture stays the same with a slight change. 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 with a linear model instead. In fact, it uses two linear models to output the next latent state and reward.

- Deterministic hidden state model: $h_t = f(h_{t-1}, s_t, a_t)$
- Deterministic latent state model: $z_{t+1} = f(h_t)$
- Deterministic reward model: $r_{t+1} = f(h_t)$
- Observation model (decoder): $o_t \sim p(o_t|z_t)$
- Approximate state posterior (encoder): $z_t \sim q(z_t|o_t)$

where f in the latent state model and the reward model are the linear models.

4.3.2. Controller

AlphaZero [39] is very similar to the MCTS algorithm, explained in the previous chapter. 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 $U(s, a) \propto P(s, a)/(1 + N(s, a))$ [40], 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 45 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, π, 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.

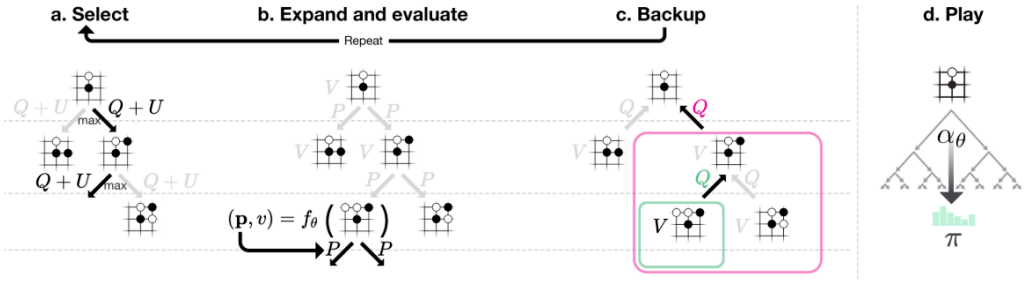

Fig. 45. Monte-Carlo tree search in AlphaZero [40]

4.3.3. 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 the LSTM hidden size is 256. Input to value and policy networks is the same concatenated features vector of latent and hidden states, $[z_t \, h_t]$.

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 at the beginning of each episode to enhance exploration and after warm-up a greedy policy is used. During testing always greedy policy is used, which 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 separately from the world model from games played by the agent, called self-play. In each position s during self-play,

an MCTS search is executed, guided by the policy network. The MCTS search at the end outputs probabilities, π, of playing each move. 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 [42]. 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 [40]. 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 [7] 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 [41] is used with a learning rate of 10^{-2} and a momentum of $0,9$.

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

4.4. PlaNet

PlaNet (Deep Planning Network) [17] 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.

4.4.1. World model

This architecture uses recurrent state space model (RSSM) which is similar to what World Models does. This latent dynamics model is designed with both deterministic and stochastic components [4]. Original experiments indicate having both components to be crucial for high planning performance. It also uses a generalized variational bound that include multi-step predictions. Using only terms in latent space results in a fast regularizer that can improve long-term predictions [17].

The same as in World Models, the model is provided with image observations. It even uses the same Variational Autoencoder with the same neural network architecture to encode the

observations into latent space. The difference lies in the dynamics model shown in figure 46 with following components:

- Deterministic hidden state model: $h_t = f(h_{t-1}, s_{t-1}, a_{t-1})$
- Stochastic latent state model: $s_t \sim p(s_t|h_t)$
- Observation model (decoder): $o_t \sim p(o_t|h_t, s_t)$
- Reward model: $r_t \sim p(r_t|h_t, s_t)$
- Approximate state posterior (encoder): $s_t \sim q(s_t|o_{\leqslant t}, a_{<t}) = \prod_{i=1}^{t} q(s_t|h_t, o_t)$

where o, s and a are high-dimensional observations, latent states and actions respectively. $f(h_{t-1}, s_{t-1}, a_{t-1})$, the deterministic state model, is implemented as a recurrent neural network and h_t is its hidden state. The latent 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 the model is non-linear, directly computing the state posteriors is intractable. Instead, an encoder q is used to infer approximate state posteriors from past observations and actions, where $q(s_t|h_t, o_t)$ is a diagonal Gaussian with mean and variance parameterised by a convolutional neural network followed by a feed-forward neural network.

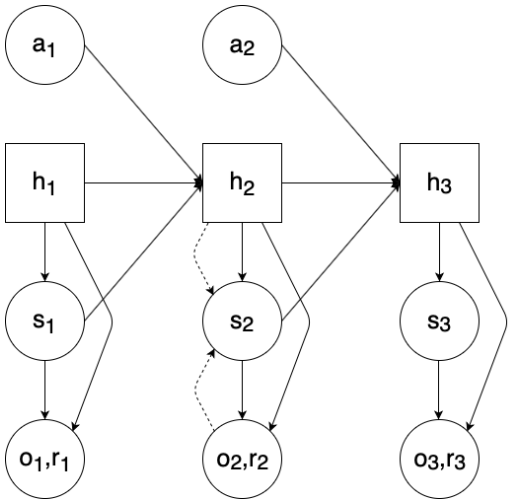

Fig. 46. PlaNet probabilistic graphical model: solid arrows describe the predictive model, dotted arrow describes the inference model, stochastic nodes are circles and squares depict deterministic nodes.

There are two deviations from the World Models architectures:

- Hidden states are “shifted” to the right, so the previous state and action are used to predict the current hidden state h and it is then used to predict the current latent state s.
- VAE’s decoder and encoder (named Vision in World Models) 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.

Intuitively, this model can be understood as splitting the state into a stochastic part and a deterministic part, which depend on the stochastic and deterministic parts at the previous time

step through the RNN. Importantly, all information about the observations must pass through the sampling step of the encoder to avoid a deterministic shortcut from inputs to reconstructions.

4.4.2. Planner

The agent plans using the cross entropy method (CEM) [2] 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. First, a time-dependent 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. 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 of the belief for the current time step μ_t 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 figure 47 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. 47. Latent planning with CEM [17]

4.4.3. Data collection

Since the agent may not initially visit all parts of the environment, new experience needs to be iteratively collect and then the dynamics model gets refined. It is done so by planning with the partially trained model. 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.

4.4.4. Preprocessing

Each image gets preprocessed by reducing the bit depth to 5 bits as in [26]. It is then resized to 64 x 64 pixels for all environments. Actions are one-hot encoded and each action gets repeated 4 times as common in reinforcement learning [30] to reduce the planning horizon and provide a clearer learning signal to the model.

4.4.5. Implementation details

This time official code from repository [16] was adjusted and used for experiments. The code architecture follows principles from other PlaNet author's paper [15]. The RSSM uses a GRU [6] 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 [33]. 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.44). 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 [25] 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 from the paper [17] is used with overshooting KL divergence terms additionally scaled by a factor of $1/50$. *[QUESTION: Should we rewrite here full loss? It's quite enormous and complicated. Reader can check it in the PlaNet's paper.]*

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.

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

5. EXPERIMENTS

In this sections the two 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.

5.1. Benchmarks

5.1.1. Arcade Learning Environment

[TODO: Include paragraph about partial observability of ALE.]

The Arcade Learning Environment (ALE) has became a platform for evaluating artificial intelligence agents. Originally proposed by Bellemare et. al. [1], 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 world through a video stream. 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. [8]

Preprocessing include frame skipping [32] which restricts the agent's decision points by repeating a selected action for 4 consecutive frames. Frame skipping results in a simpler reinforcement learning problem and speeds up execution. *[TODO: Describe other preprocessing techniques used here.]*

This work uses ALE through OpenAI Gym API [3], specifically two 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. 51. 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.

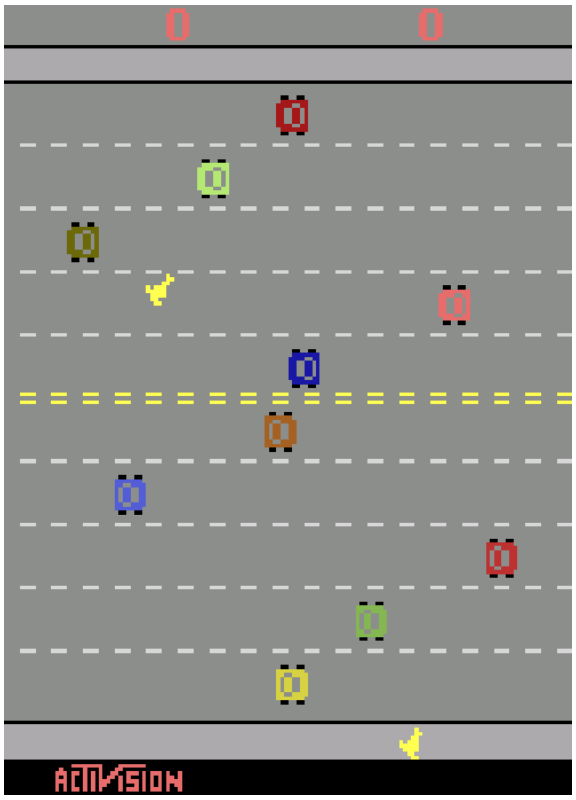

Fig. 52. Example of Freeway level

[TODO: Add more games if needed.]

5.1.2. Sokoban

[TODO: Include paragraph about fully observability of Sokoban state.]

Sokoban is a classic planning problem. 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, Sokoban is an incredibly complex game for which no general solver exists. It can be shown that Sokoban is NP-Hard and PSPACE-complete [9]. 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[36] 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, agents were trained directly on RGB sprite graphics. Fig. 53 shows an example of Sokoban level with 4 boxes.

[TODO: Go into deeper details about e.g. how rewards are obtained etc.]

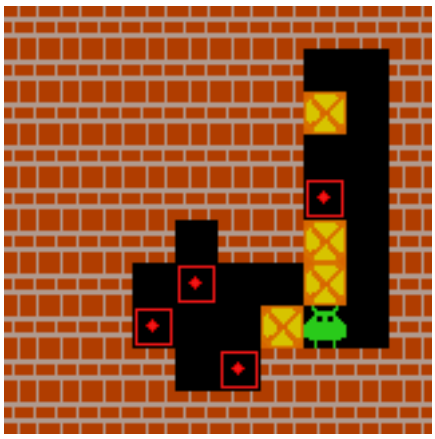

Fig. 53. Example of Sokoban level

5.2. Hardware

[TODO: Add HW specification.]

5.3. World Models for Sokoban

This section focuses on a problem of model learning in Sokoban that is used to solve the environment. The goal was to train a model to generate sharp and accurate open loop predictions of observations and obtain high score using it. In theory, how probable are sequences of ground truth observations is measured using log probability. In practice, though, it is far more useful to compare those generated sequences of future observations with ground truth sequences with an eye of the researcher. What the researcher 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 or does not model actions issued well.

To generate an open loop prediction 60 consecutive frames and actions on average are fed into the model to initialize its hidden state first. Then, the model is ready to generate an open loop future prediction with its memory module by feeding it with subsequent actions and its own predictions as a contemporary state at following time steps. Reader should notice that what gets generated are not frames, but latent state's vectors. Those latent states can then be decoded into images for inspection.

5.3.1. Train the unchanged World Models implementation in the Sokoban environment

In this experiment, the original World Models was trained in the Sokoban environment. No modification to the original method described in the related work chapter was made, beyond addition of the deterministic variant of memory module described in the previous chapter.

The vision model successfully learned to encode high dimensional observations into low dimensional latent states. Fig. 54 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.

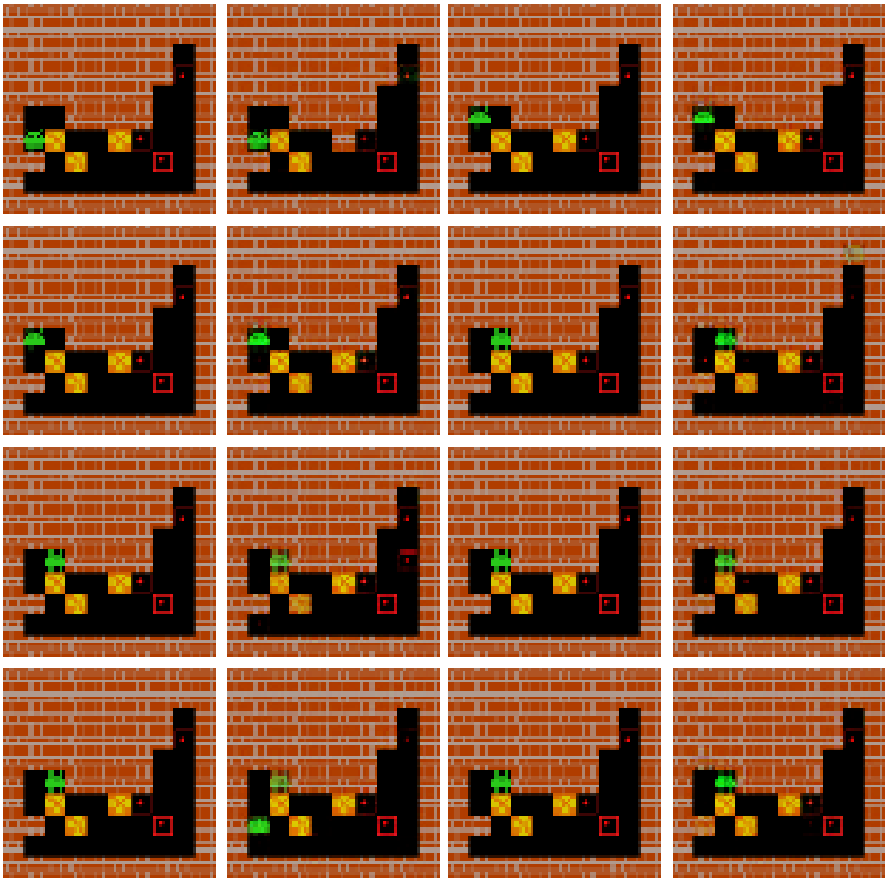

Fig. 54. Qualitative result of the vision model training in Sokoban. First and third columns include original observations. Second and fourth columns include reconstructions. Each reconstruction was obtained by first encoding the original observation and then decoding it, using VAE encoder and decoder respectively.

The stochastic and deterministic memory models were not able to learn Sokoban's dynamics. Fig. 55 shows that the stochastic model very often can not determine the agents position. The agent disappears and blocks change their types. The eighth row shows that pushing mechanics are not modeled, the agent passes through boxes. The deterministic model does not do better. The controller model failed to learn how to solve any level, it behaved comparably to a random play. We suspect that VAE is unable to generate usable abstract Sokoban representation and the

shallow memory and controller models can not grasp complex dynamics of Sokoban using this poor representation. This idea is further developed in the next experiment.

Fig. 55. Qualitative result of the memory model training in Sokoban. Each row depicts the memory model 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 model and then decoding it using the VAE decoder.

5.3.2. Train World Models in 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 block types organized in an 8 x 8 grid. In this experiment, this domain knowledge is exploited and the agent uses those 64 block types as an input vector to the memory

module, bypassing the vision model. It is worth noting, that the vision model 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 (block types in each position) to infer optimal next action and this is exactly what the controller model 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 latent features can not model useful policy.

On the other hand, this representation includes, not well represented, but perfect information about an environment state. The memory model 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. One way to improve the hidden state representation is explored in the next experiment.

5.3.3. Train World Model in Sokoban with auxiliary tasks

Auxiliary tasks[22] have proved to help create more informative representation of an environment. In this experiment, reward and value prediction tasks are added to the memory model. 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 *[TODO: Add information how you train values i.e. Monte-Carlo prediction]*. In theory, it should help form a more informative hidden state of the memory model. Consequently, it should help learn Sokoban's dynamics, but also generate representation on a higher level of abstraction that could prove useful for the controller. Moreover, a reward prediction will be needed in further work on planning with learned model.

For all that, the memory model have not been able to learn to predict the rewards and values. Also, there was no improvement in memory's and controller's 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 model 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.

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 in a different episode. 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. For instance I2A[45] uses deep neural network architecture to handle Sokoban complexity. A more flexible memory model with larger capacity could manage this complexity and need for generalization. The two insights are explored in the PlaNet experiments, which have larger model and uses iterative training procedure. However before that, World Models are tested in an easier environment, Boxing from Atari.

5.4. World Models for Atari

In two experiments below ideas from previous section were put into the test. Firstly, the original World Models is trained for the Boxing environment, which has dense rewards and data collection using a random agent cover most of the state-space. Then, World Models is coupled with AlphaZero planner and both are trained jointly.

5.4.1. Train unchanged World Models in the Boxing environment

In this experiment, the original World Models was trained in the Boxing environment. No modification to the original method described in the related work chapter was made. In this experiment discrete memory was not tested as original stochastic one did well.

The vision model successfully learned to encode high dimensional observations into low dimensional latent states. Fig. 56 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.

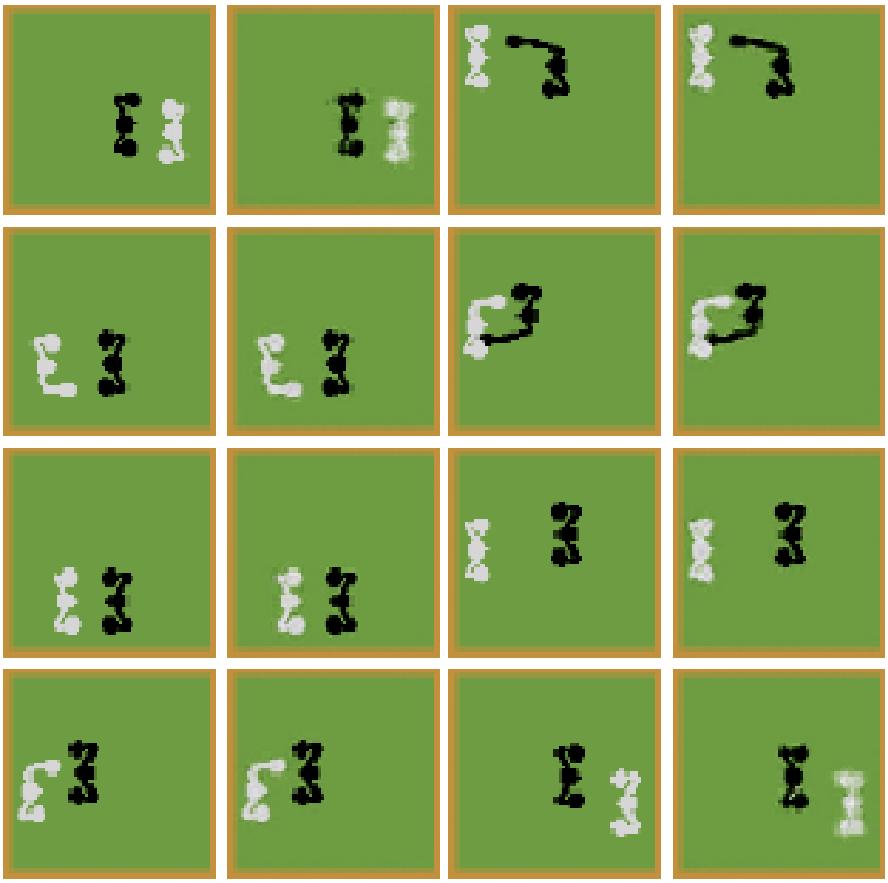

Fig. 56. Qualitative result of the vision model training in Boxing. First and third columns include original observations. Second and fourth columns include reconstructions. Each reconstruction was obtained by first encoding the original observation and then decoding it, using VAE encoder and decoder respectively.

The stochastic memory models was able to learn Boxing's dynamics. Fig. 57 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 model successfully learned how to solve the game scoring above 18 points on average across 5 runs. We suspect that World Models with latent state of size 16 was 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.

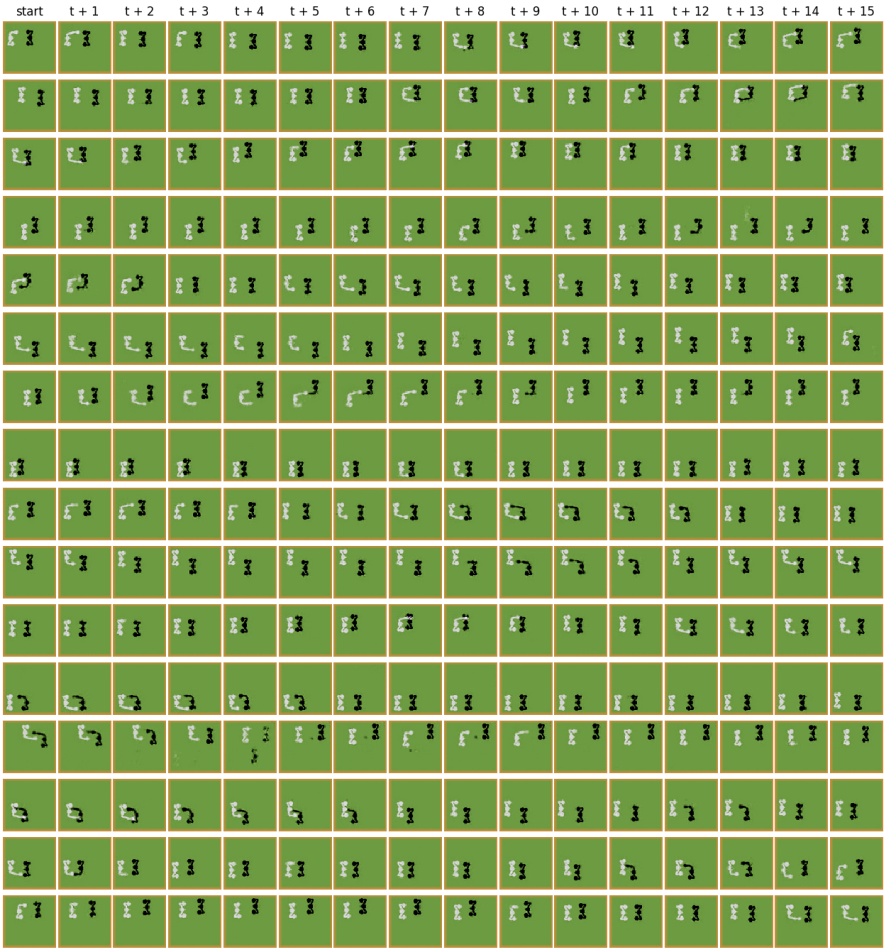

Fig. 57. Qualitative result of the memory model training in Boxing. Each row depicts the memory model rollout in one episode. The first column include original observations from the evaluation dataset from which the rollouts start. The RNN's hidden state was initialized on preceding transitions in each episode. Each subsequent reconstruction was obtained by first predicting the next latent state by the memory model and then decoding it using the VAE decoder.

5.4.2. Train World Models with AlphaZero planner in the Boxing environment

Despite getting really accurate future predictions in the previous experiment and hyper-parameter tuning of this architecture the AlphaZero planner training was unstable. It did not train to properly plan in latent space and play the game. Decision was to abandon this solution and move to PlaNet which shown that, in deed, it is possible to plan in continuous control tasks.

5.5. PlaNet for Sokoban

5.5.1. Train unchanged PlaNet in the Sokoban environment

PlaNet did not capture Sokoban dynamics too. In the figure below (Fig. 58) future predictions are blurred, multiple agents appear and other artifacts, like changing block type, are present. Similarly like in World Models case decision was to move to Atari games as easier environments to start with.

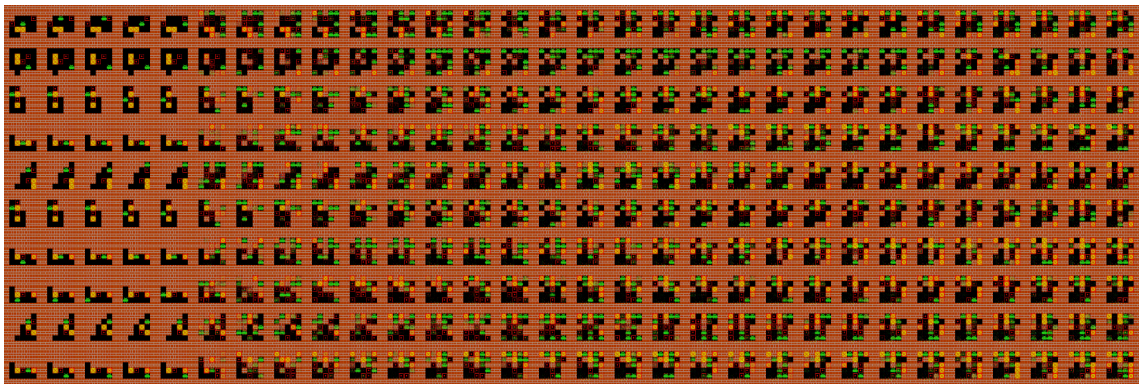

Fig. 58. Qualitative result of the model training in Sokoban. Each row depicts the model rollout in one episode. The first five columns include original consecutive observations from the evaluation dataset from which the rollouts start. The model hidden state was initialized on these transitions. Each subsequent reconstruction was obtained by first predicting the next latent state by the model and then decoding it using the decoder.

5.6. PlaNet for Atari

In this section experiments that lead to first successful case are described. The next section will focus on tuning this method to yield high scores, comparable to model-free methods, with the smallest amount of data possible.

5.6.1. Train unchanged PlaNet in the Boxing environment

It did not start to work out of the box of course. Fig. 59 shows that future predictions turn into a blurry blob, where it is not possible to distinguish one player from another.

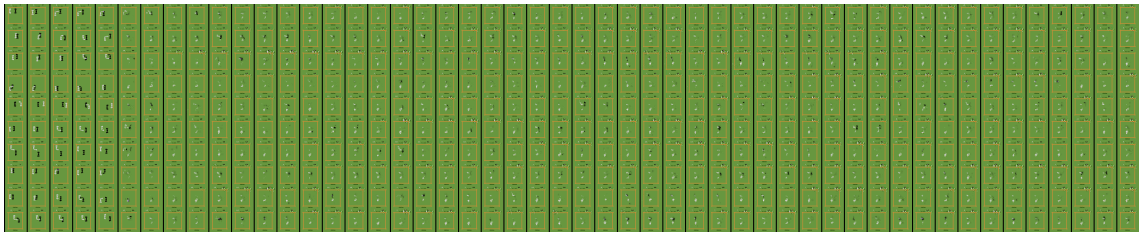

Fig. 59. Qualitative result of the model training in Boxing. Each row depicts the model rollout in one episode. The first five columns include original consecutive observations from the evaluation dataset from which the rollouts start. The model hidden state was initialized on these transitions. Each 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.

5.6.2. Train PlaNet 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 cannot be easily modeled as noise. In practice though, it did not help and even made the agent play worse then a random agent. The random agent is taking moves at random.

5.6.3. Train PlaNet in the Boxing environment with the lowered decoder variance

The predictions are more blurry with a higher variance because the decoder generate more observations that differ slightly from the same latent code. This leads to the posterior 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 it becomes more sensitive to small changes in observations. *[TODO: Proofread this explanation once more and/or ask Danijar if it is right.]*

Lowering the decoder variance is equivalent to lowering a KL divergence scale in the PlaNet loss. It can be seen by writing the ELBO for a Gaussian decoder in the standard form $E_q(z)[lnp(x|z)] - KL[q(z)||p(z)]$. The log-likelihood terms is $lnp(x|z) = -0.5(x - f(z))/\sigma^2 - lnZ$. 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 [20]. The objectives have different values because of the Gaussian normalizer Z but they share the same gradient since the normalizer is a constant. 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 example, 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. *[QUESTION: I should add diagram with random search results, but how to do this if those are evaluated with a researcher eye?] [TODO: You should better describe this random search experiment. What parameters where tuned, which turned out to be the most important, for how long and how much runs you were running etc.]*

5.6.4. *Train PlaNet in the Boxing environment with increased free nats*

Free nats technique is often used for static Variational Autoencoders. The model is allowed to use given amount of nats without KL penalty in 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 divergence (between a prior and a posterior) reconstruction loss is favoured. In case of Boxing, it helped to model boxers moves and actions more accurately. The best free nats turned out to be 12. Fig. 510 shows final result.

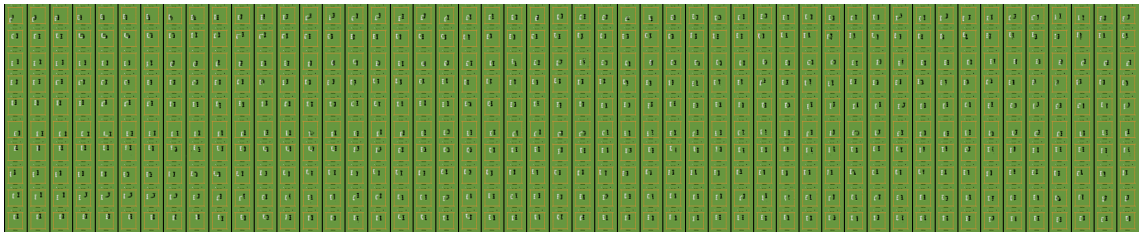

Fig. 510. Qualitative result of the model training in Boxing. Each row depicts the model rollout in one episode. The first five columns include original consecutive observations from the evaluation dataset from which the rollouts start. The model hidden state was initialized on these transitions. Each subsequent reconstruction was obtained by first predicting the next latent state by the model and then decoding it using the decoder.

It achieved final score around 30. *[TODO: Download .csv with results from five Boxing training and plot nice curve.]*

5.6.5. *Train tuned PlaNet in the Freeway environment*

The same random search procedure was applied to the Freeway environment described earlier. Fig. 511 shows final result.

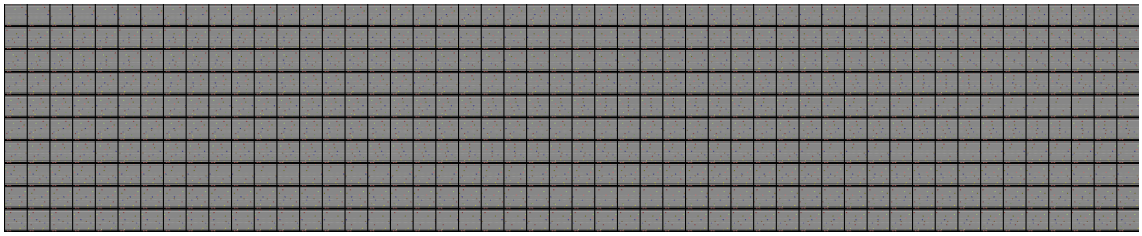

Fig. 511. Qualitative result of the model training in Freeway. Each row depicts the model rollout in one episode. The first five columns include original consecutive observations from the evaluation dataset from which the rollouts start. The model hidden state was initialized on these transitions. Each subsequent reconstruction was obtained by first predicting the next latent state by the model and then decoding it using the decoder.

Despite really good future observations prediction the agent failed to solve the task. *[TODO: In benchmark description you should write what you consider as solved task in each environment.]* Possibly planner horizon is to short to cover a plan which ends with positive reward on the other side of the road *[TODO: In nomenclature or somewhere else you should clearly describe what is "a plan".]*. This is explored in the next experiment.

5.6.6. *Train tuned PlaNet in the Freeway environment with a longer planning horizon*

[NOTE: This is the last experiment in progress, but it does not seem promising.]

6. CONCLUSION

REFERENCES

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

[2] 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.

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

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

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

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

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

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

[9] 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.

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

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

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

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

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

[15] 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.

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

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

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

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

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

[21] 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.

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

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

[24] 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.

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

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

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

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

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

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

[38] 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.

[39] David Silver et al. “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”. In: *arXiv e-prints* (2017). arXiv: 1712.01815.

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

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

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

[43] 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.

[44] 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.

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

LIST OF FIGURES

LIST OF TABLES