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

Computer science defines Artificial Intelligence (AI) as the intelligent agents in form of any device that perceives its environment and takes actions that maximize its chance of successfully achieving its goals. Progress has been made in developing capable agents for numerous domains using deep neural networks in conjunction with model-free reinforcement learning [10][15][19], where raw observations directly map to agent's actions. However, current state-of-the-art approaches usually are sample inefficient, they require thousands of millions of interactions with the environment, and lack the behavioural flexibility of human intelligence, hence the resulting policies poorly generalize to novel task in the same environment.

The other branching of reinforcement learning algorithms is called model-based reinforcement learning, which gives the agent access to (or learns) a model of the environment. There are many orthogonal ways of using the model: one can use the model for data augmentation for model-free methods[5], some methods use the model as the imagined environment to learn model-free policy in it[8], other methods focus on simulation-based search using the model[21] and there are even methods that integrate model-free and model-based approaches[23]. The model allows the agent to simulate an outcome of an action taken in a given state. The main upside to having the model 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 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 learned first it can exploit additional unsupervised learning signals, thus it results in a substantial improvement in sample efficiency over methods that do not have a model. Furthermore, the same model can be used by the agent to complete other tasks in the same environment.

Planning is different from learning. In the former the agent samples episodes of simulated experience and updates its policy based on them. In the latter the agent also updates its policy, but this time based on real experience gained through interaction with the environment. 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.

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 the agent. If an agent wants to use a model in this case, it has to learn the model purely from experience, which creates several challenges. The biggest challenge is that bias in the model can be exploited by the agent, resulting in an agent which performs well with respect to the learned model, but behaves sub-optimally in the real environment. Model-learning in complex domains is fundamentally hard and requires function approximation, so resulting models are inher-

ently imperfect. The performance of agents employing standard planning methods usually suffer from model errors. Those errors compound during planning, causing more and more inaccurate predictions the further plans' horizon.

There are many real-world problems that could benefit from application of general AI system. Company called DeepMind, driven by their experience from creating winning Go search algorithm AlphaZero[21], published AlphaFold[4], 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 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 thousands 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 models are needed.

This work explores progress in the model-learning domain and examine a possibility of using learned models in simulation-based search algorithms. Recently, there have been major steps taken in the domain of model-learning algorithms[2][14][1][9]. More accurate models, at least in short horizon, open the path for application of proofed simulation-based search planning algorithms like TD-Search[20] or AlphaZero[21] to complex planning problems in environments with discrete state-action spaces and without access to a ground-truth model. This should allow for improvements in data efficiency, generalization and agents performance for these problems. This work focuses on two benchmarks: a complex puzzle environment, Sokoban and a multi-task environment, MiniPacman.

2. THEORETICAL BACKGROUND

2.1. Markov Decision Processes

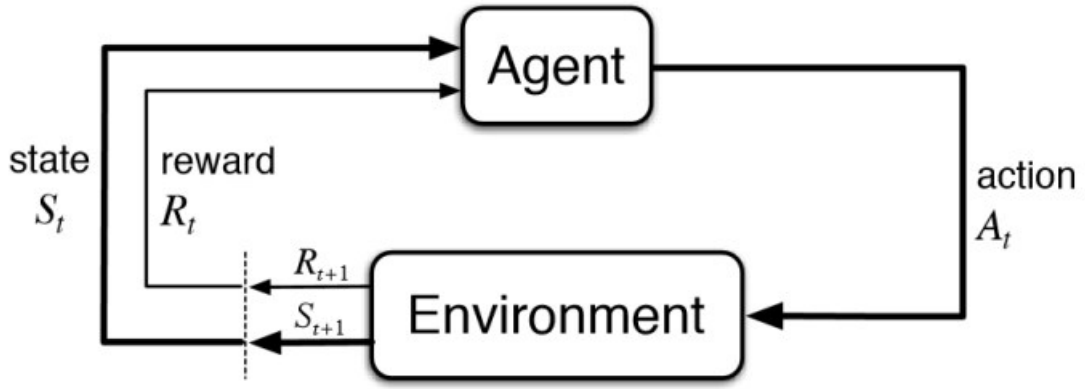
Markov Decision Processes (MDPs) are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations (states) and through those future rewards. Thus MDPs involve delayed reward and the need to tradeoff immediate and delayed reward.

A Markov Decision Process consists of a set of states S 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 probabilities, $P^a_{ss'} = Pr(s_{t+1} = s' | s_t = s, a_t = a)$, specifying the distribution over the next state $s' \in S$. Finally, a reward function, $R^a_{ss'} = \mathbb{E}[r_{t+1} | s_t = s, a_t = a, s_{t+1} = s']$, specifies the expected reward for a given state transition. Episodic MDPs terminate with probability 1 in a distinguished terminal state, $s_T \in S$, after finite number of transitions. Continuous MDPs does not include the terminal state and can run endlessly. The return $R_t = \sum_{k=t}^{T} r_k$ is the total reward accumulated in that episode from time t until reaching the terminal state at time T. In continuous MDPs this is the sum of an infinite sequence of rewards from time t. A policy, $\pi(s, a) = Pr(a_t = a | s_t = s)$, maps a state s to a probability distribution over actions. The value function, $V_\pi(s) = \mathbb{E}_\pi[R_t | s_t = s]$, is the expected return from state s when following policy π. The action value function, $Q_\pi(s, a) = \mathbb{E}_\pi[R_t | s_t = s, a_t = a]$, is the expected return after selecting action a in state s and then following policy π. The optimal value function is the unique value function that maximises the value of every state, $V^*(s) = max V_\pi(s), \forall_{s \in S}$ and $Q^*(s, a) = max Q_\pi(s, a), \forall_{s \in S, a \in A}$. An optimal policy $\pi^*(s, a)$ is a policy that maximises the action value function from every state in the MDP, $\pi^*(s, a) = \underset{a}{\text{argmax}} Q^*(s, a)$.

MDPs are a mathematically idealized form of the reinforcement learning problem for which precise theoretical statements can be made. In reinforcement learning the dynamics and the reward function are hidden behind an environment. Consequently, we can not directly use them for planning, but we can learn 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.[22] 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.



Rys. 21. Reinforcement Learning[22]

[Should I 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

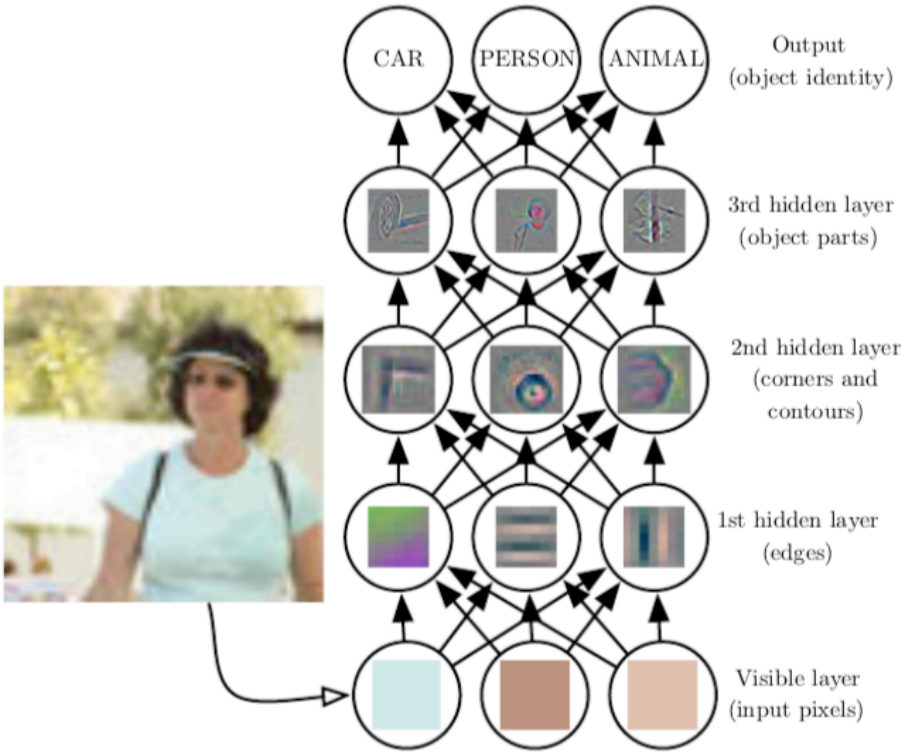
Machine learning gives AI systems the ability to acquire their own knowledge, by extracting patterns from raw data. In stands in opposition to classical computer programs which execute explicit instructions hand-coded by a programmer. One example of machine learning algorithm is logistic regression. It can determine whether to recommend cesarean delivery or not[16]. 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.

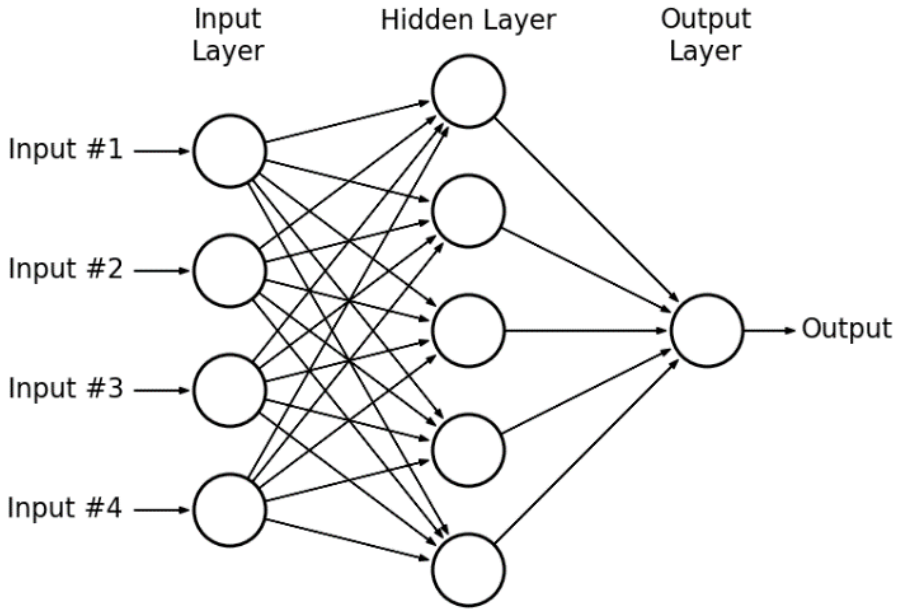
Deep learning is a particular kind of machine learning that achieves great power and flexibility by learning to represent the world as a nested hierarchy of concepts, with each concept defined in relation to simpler concepts. Deep learning solves representation learning problem by introducing representations 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.



Rys. 22. Deep Learning[6]

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.



Rys. 23. Multilayer perceptron

[Should I mention Deep Reinforcement Learning somehow in here? Or maybe in Reinforcement Learning section?] [Also, should I describe RNNs here if we use them later? Currently, there is citing in "Related Work" chapter to paper about LSTM, is it enough?]

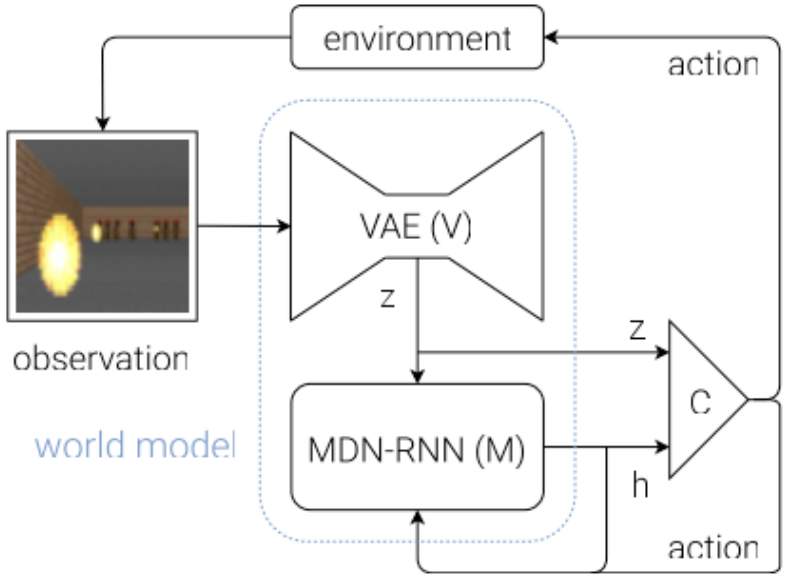
3. RELATED WORK

3.1. Model learning

3.1.1. World Models

In World Models[8] 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.



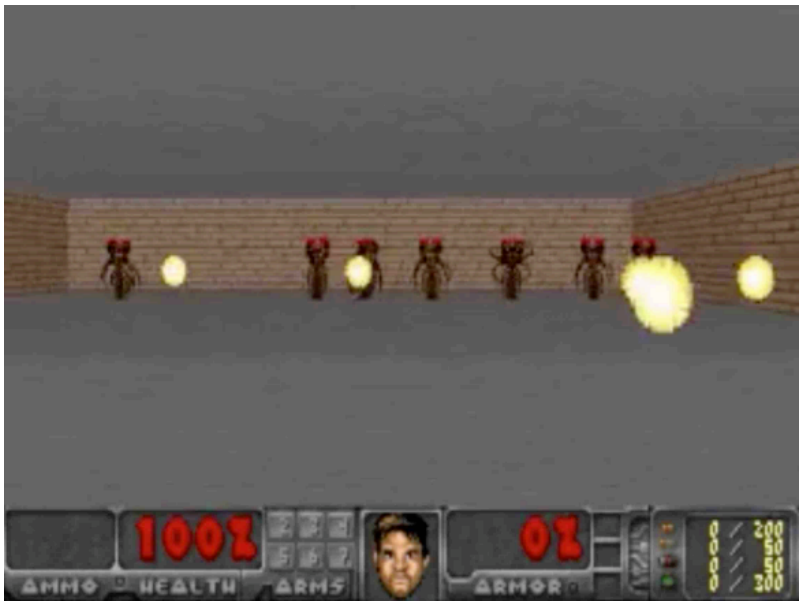
Rys. 31. Flow diagram of the agent's model[8]. The raw observation is first processed by V at each time step t to produce z_t. The input into C is this latent vector z_t concatenated with M's hidden state h_t at each time step. C will then output an action vector a_t and will affect the environment. 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[13]. 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[11] (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)[7]. 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[17].

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?]*



Rys. 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[8].

4. PLANNING WITH LEARNED MODEL

[Make diagrams of your concept and describe them. It won't be long.]

This work divides into two parts: learning the environment model and application of planning algorithms. In the first part, current advancements in model-learning domain are explored. The goal is to find and train *[Note to the advisor: find and train or just train and we assume that we found an appropriate method? Answer: describe idea not history e.g. This is problem... Two models were tested... That work, that doesn't...]* the sufficient environment model. Sufficient means, it provides accurate predictions about future states to some cut-off point in time. The further away the cut-off point, the better model-learning method. In the other part, the goal is to find and implement *[The same as above]* a planning algorithm which will use learned model. As described in the introduction, main problems are: the model bias and the compounding error *[Note to the advisor: is the "compunding error" the correct term for that?]*. The point is to find planning algorithms and refinements to them that will result in a robust method that can successfully plan using the imperfect model. Before that, the code architecture and the framework that was created to accelerate this research are described.

4.1. HumbleRL

[Framework + implementation architecture, make diagrams and describe them]

4.2. Model-learning

In this part, a model of an environment that can be used locally for planning is trained. Locally means that the planning algorithm operates on trimmed simulated episodes to the point where the model is still accurate. There are different methods examined to choose the best one.

4.2.1. World Models

World Models' agent, as shown in the paper[8], is able to learn from simulated experience. It is an example of successful planning using learned model. This work utilize the world model part of the agent in order to learn model of the two mentioned benchmarks. Also, Vision and Memory encode environment observations into low level representation. The latent state of the world models encodes abstract information about the environment and allow the planning controller to stay small.

[Write how data was generated, how each component was trained (in separation) etc.]

4.3. Planning

[This section will be written in its final shape later when we'll have learned model.]

The main contribution of this part is application of search algorithms that use the imperfect learned model to solve the complex planning problems. Specifically, the model latent state is used for planning by algorithms such as TD-Search in the complex planning problems like Sokoban.

TD-Search is Q-value function approximation with linear model learned using bootstrapping on simulated experience. Thanks to bootstrapping full episodes doesn’t need to be used.

5. EXPERIMENTS

5.1. Benchmarks

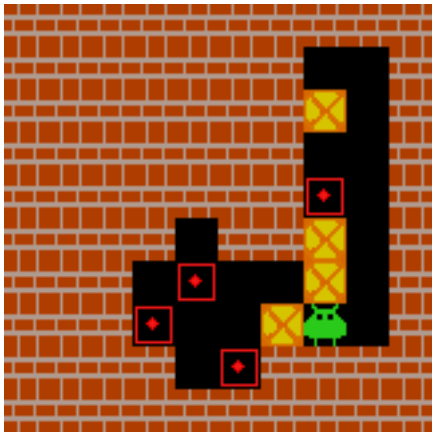
5.1.1. Sokoban

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 ruleset, Sokoban is an incredibly complex game for which no general solver exists. It can be shown that Sokoban is NP-Hard and PSPACE-complete[3] *[Revise what does it mean NP-Hard and PSPACE-complete, hehe]*. 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[18] 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. 51 shows an example of Sokoban level with 4 boxes and fig. 52 explains the meaning of the visual icons.

[Note to the advisor: Do we need to go into deeper details about e.g. how rewards are obtained etc.? Or rather point to the repository for more information?]



Rys. 51. Example of Sokoban level (image size 160 × 160 pixels)

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

Rys. 52. Sokoban level elements

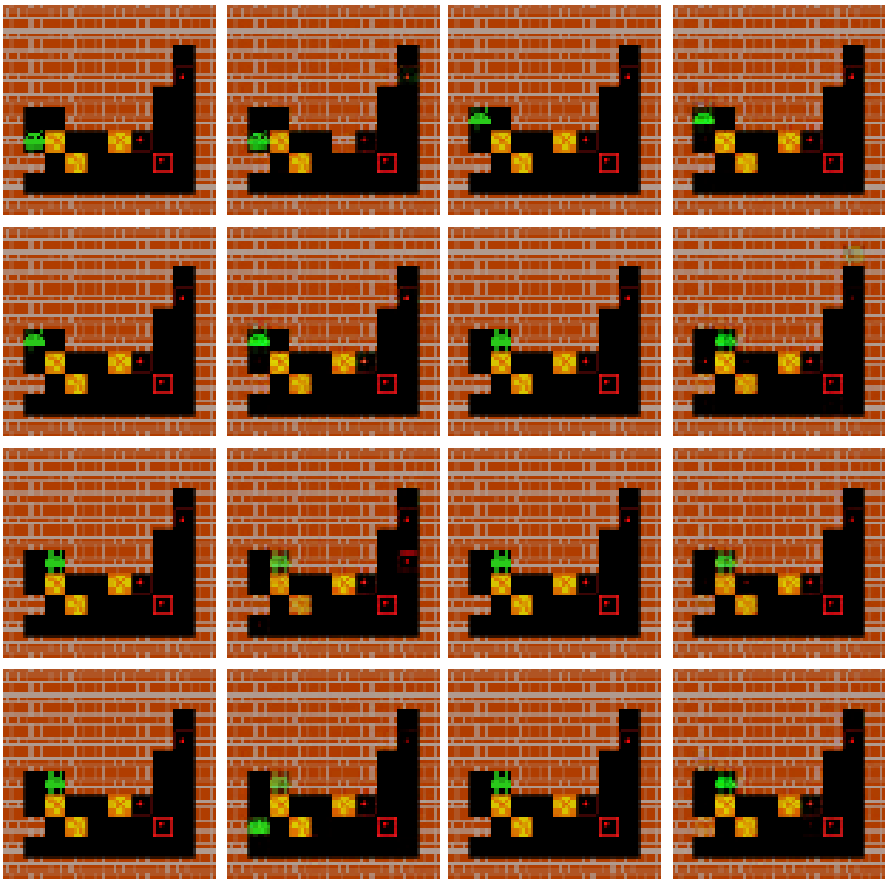
5.2. Model learning

First three experiments focus on training the world model in Sokoban environment. Because Sokoban is the deterministic environment, each experiment tested also the memory module without Mixture Density Network on top of a RNN. Instead a linear model was used to output the next latent state. *[Note to the advisor: Should it be in the previous chapter (Planning with learned model a.k.a. project plan)?]*

5.2.1. Train the world model in the Sokoban environment

In this experiment, the original world model 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.

The vision model successfully learned to encode high dimensional observations into low dimensional latent states. Fig. 53 shows original observations (first and third columns) side by side with reconstructed observations from their encodings (second and fourth columns).



Rys. 53. 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, using VAE encoder and decoder respectively.

[Make a better diagram.]

The stochastic and deterministic memory models were not able to learn Sokoban's dynamics. Fig. 54 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 aren't modeled, the agent passes through boxes. The deterministic model don't do better. The controller model failed to learn how to solve any level. 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.

Rys. 54. 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 VAE decoder.

[Make a better diagram.] [Note to the advisor: Do we need to go into great detail about how it was generated? E.g. do we need to describe how the RNN was initialized or just point to the code?]

5.2.2. Train the world model 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't 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 doesn't seem to encode useful enough information for the two to do well on their tasks. One way to improve the hidden state representation is explored in the next experiment.

5.2.3. Train the world model in Sokoban with auxiliary tasks

Auxiliary tasks[12] 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. 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. Each episode 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[23] 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. We explore those two insights in the next experiment with larger model and iterative training procedure.

6. CONCLUSION

REFERENCES

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

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

[3] 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.

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

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

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

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

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

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

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

[11] 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.

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

[13] 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.

[14] 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.

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

[16] 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.

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

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

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

[20] 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.

[21] 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.

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

[23] 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