Learning Task-Specific Representations of Environment Models in Deep Reinforcement Learning

Yota Mizutani and **Yoshimasa Tsuruoka**
The University of Tokyo
{mizutani, tsuruoka}@logos.t.u-tokyo.ac.jp

Abstract

Model-based deep reinforcement learning has recently been attracting increasing attention due to its sample efficiency. In this paper, we introduce a new model-based reinforcement learning architecture based on Imagination-Augmented Agents (I2A). With our architecture, agent learns a policy jointly with small but sufficient task-specific representations of the states, which enable the lookahead of future states with a relatively low computational cost. More specifically, our architecture allows an agent to learn a representation suitable for a given task by combining a vector representation corresponding to the current state with the encoded features corresponding to predicted states. The agent also learns how to predict the representation of the next state and learn a policy with predicted states jointly and efficiently by using training batches consisting of data arranged in chronological order. Experimental results on the game of Sokoban demonstrate that our architecture outperforms a state-of-the-art model-based reinforcement learning architecture in terms of the time of training and execution.

Introduction

Since the advent of Deep Q-Networks (Mnih et al. 2013), research on applying deep reinforcement learning to video games has attracted significant attention. In most of the previous work, agents do not have any prior knowledge about the game, and receive only the raw pixels as the observation. If an agent could learn the values of all possible states and actions, it would be able to obtain the optimal strategy. However, in complex environments like video games, learning the values of all possible states and actions is nearly impossible. Due to incomplete learning, it is difficult for model-free reinforcement learning methods to build an agent that can select actions based on a long-term plan with respect to future conditions.

On the other hand, model-based reinforcement learning allows an agent to learn a policy using a state transition model of an environment. Agents can perform imaginary state transitions and find the optimal action before actually taking an action, and therefore can make a long-term strategy with anticipation of future states. In addition, they can learn useful policies with a relatively small number of interactions with environments by using trained environment models (Clavera et al. 2018).

In complex environments such as video games and tasks in the real world, since the model of an environment is often unknown, agents need to learn it through the interaction with the environment (Sutton 1990). However, it is often difficult to obtain a perfect environment model, and the learned models can be inaccurate. Some existing methods, such as Imagination-Augmented Agents (I2A) (Racanière et al. 2017), address this issue by using neural networks to distill useful information from the predicted trajectories.

There are several issues to be addressed in model-based deep reinforcement learning. Firstly, since the observations from environments often contain high-dimensional data such as raw game screens, methods using an environment model with the raw observations as they are must handle high-dimensional input and output of the environment model. As a result, the computational cost at the time of learning and testing becomes large, limiting the applicability of the methods.

The second issue is the scheduling of training. Agents must learn multiple interdependent models including an environment model, a policy using the learned model, and in some cases, representations of observations. For example, with a poor policy, agents cannot experience latter phases of the game, and the environment model cannot learn them. Therefore, fine adjustment of scheduling these kinds of training is needed for methods that require a pre-trained environment model or a pre-trained network for converting observations to low-dimensional representations.

In this paper, we propose a new architecture for model-based reinforcement learning. Our architecture allows agents to learn low-dimensional representations of observations suitable for solving the given task by combining the vector representation with the encoded features of predicted states and using the combined vector to learn a policy. It also learns state transitions in the representation space and how to deal with the predicted states. All of these are trained jointly and efficiently by calculating the loss functions of the reinforcement learning and the prediction at the same time with training batches of Importance Weighted Actor-Learner Architecture (IMPALA) (Espeholt et al. 2018) which consist of data arranged in chronological order. Our method makes

it possible to establish a long-term strategy with a relatively small computational cost even in environments with high-dimensional input without complex scheduling of several kinds of training procedures. Experimental results on the game of Sokoban demonstrate that our architecture outperforms a state-of-the-art model-based reinforcement learning architecture.

Background

Model-based reinforcement learning

The envitonment model for a reinforcement learning problem is formulated as the Markov decision processes (MDP), whose transitions are expressed by the state transition function $T(s_{t+1}|s_t, a_t)$ and the reward function $r(s_t, a_t, s_{t+1})$, where s_t is the observation at time t, and a_t is the action which the agent takes at time t. In this work, we consider determistic environments and discrete action spaces. Agents of model-based reinforcement learning learn these transitions and use them to make policies for maximizing cumulative rewards $\sum_{t=0}^{\infty} \gamma^t r_t$, where γ is the discount factor for rewards, and r_t means the reward which the agent receives at time t.

Imagination-Augmented Agents (I2A)

Our approach to using predicted states for learning a policy is largely inspired by Imagination-Augmented Agents (I2A) (Racanière et al. 2017). I2A is an architecture for model-based reinforcement learning, which can interpret predicted states from an imperfect environment model. An agent of I2A learns an environment model that directly predicts the next observation image from the current observation image and the current action, and uses it for determining the policy. Since an image given as an observation generally has large dimensions, prediction is often complex. Therefore, the accuracy of the learned model can be low, but by extracting the features of predicted images through a convolutional neural network and encoding them in the reverse order of the time series using Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber 1997), the agent can learn how to handle predicted states considering the inaccuracy of the model. All outputs encoded by multiple imaginary rollouts are combined with hidden layers output by an ordinary model-free agent and used to calculate the value function and the policy function. In addition, they use rollout policy $\hat{\pi}$ to decide actions for rollouts. By outputting the policy close to the final policy function π, $\hat{\pi}$ can predict the same action as the agent will actually take in the future, and predicted states by rollouts tend to be useful. Therefore, agents distill $\hat{\pi}$ from the actual value of π during training.

It has been reported that I2A agents recorded higher scores than model-free Asynchronos Advantage Actor-Critic (A3C) (Mnih et al. 2016) agents in the game of Sokoban and MiniPacman. Moreover, in MiniPacman, I2A achieved high scores in multiple kinds of tasks without changing the trained environment model.

The environment model of I2A has the disadvantage that the computational cost is large because it inputs observation images and actions and outputs images directly corresponding to the next observation. As the dimension of images increases, the computational cost also increases, and therefore, it can be a serious problem in applying it to complex tasks.

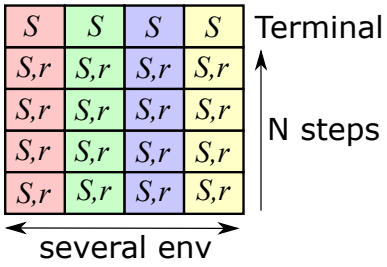

Figure 1: Training batch of IMPALA. S means an observation, and r means a reward. Data of actions and behaviour policies are also necessary for training.

Importance Weighted Actor-Learner Architecture (IMPALA)

Importance Weighted Actor-Learner Architecture (IMPALA) (Espeholt et al. 2018) is an architecture for distributed reinforcement learning, which enables efficient training with many actors. In IMPALA, a large number of actors are run in parallel, the observations for determining actions and generated training data are queued respectively, and batches are created from queues and sent to the Learner in turn. Unlike ordinary A3C and Batched A2C (Clemente, Castejón, and Chandra 2017), since there is a time gap between the time agents decide an action and the time training data are learned by the Learner, there is a possibility that learning with training data obtained by other actors is performed during the gap. This causes deviations between the policy at the time of action and the policy at the time of learning. In order to cope with the deviation, IMPALA uses importance sampling and delays the calculation of advantages until training time. Normally, in A3C, the values estimated by the value function are acquired at the time of deciding actions, and advantages are calculated using them. In IMPALA, if the advantages are calculated when a training batch is made, the values of the value function will change by the time of training. Therefore, the observations of the next states are given to training data as indicated by Figure 1 and the advantages are calculated by inputting these observations to the value function network when the training is actually done. Since an agent should calculate advantages in reverse order for n-step observations, the training batches of IMPALA consist of data in chronological order.

Proposed Architecture

Our architecture converts each observation of the current state to a low-dimensional vector representation using a convolutional neural network, and predicts next states using a LSTM network. After that the predicted states are encoded with another LSTM network, and concatenated with the representation of the current state to calculate the policy and

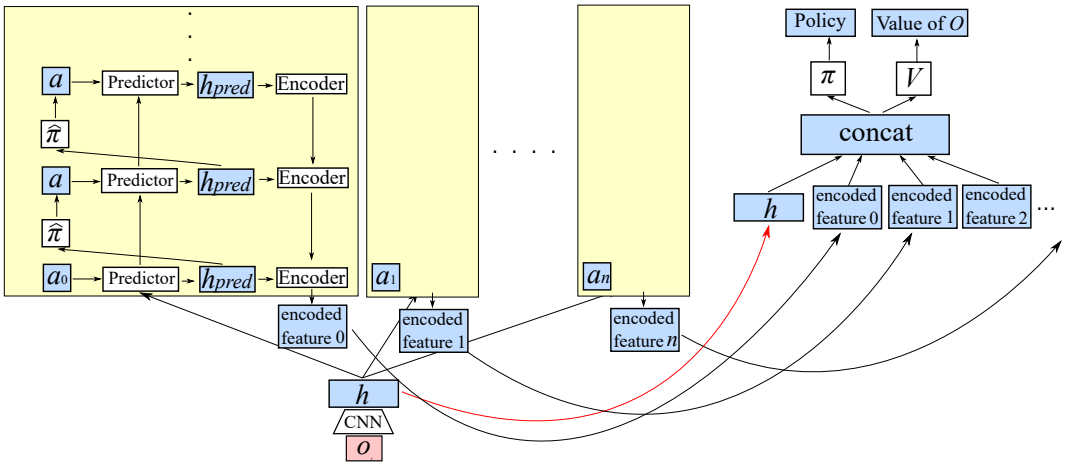

Figure 2: Overview of proposed architecture. An observation is converted to a low-dimensional vector using a CNN. The agent predicts future states for each possible action and encodes them. All encoded features and vector representation of the current state are combined to calculate the policy and value functions. White blocks in this figure have parameters to be trained.

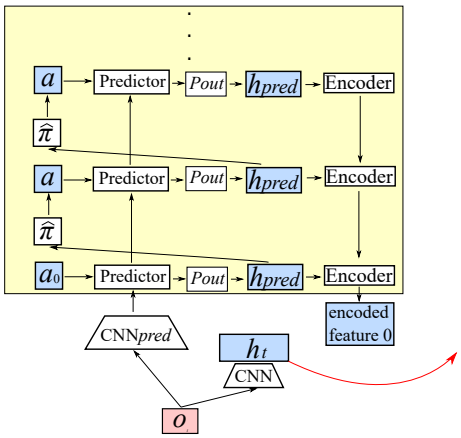

Figure 3: Predictor of the full model. A vector representation h of the current state is not used for prediction, but it is combined with encoded features from the Predictor to calculate the policy and value functions. Note that h is also used as a teacher for training the Predictor.

value functions. This allows the agent to jointly learn representations that are useful for solving the task and how to make a policy with predicted states.

An overview of the architecture is shown in Figure 2 and Algorithm 1. After converting an observation o to a feature vector h, the following operation is performed for each possible action a_i. First, the agent inputs the vector representation of a_i to a LSTM network, whose hidden state is h, in order to predict the representation of the next state. The LSTM network used for next state prediction is called the Predictor. The predicted next state h_{pred} is input to the rollout policy $\hat{\pi}$, which is a fully connected neural network, and the next action is predicted. The predicted action a_{pred} is input to the Predictor, and further predictions of next states are performed. After repeating this for N steps, the predicted states are input to another LSTM network in the reverse order of the time series to obtain an encoded feature. After obtaining the features for all a_i, we combine them with h, and input them to the feed forward neural network to obtain the values of the policy function and the value function.

Algorithm 1 policy and value function

o is an observation from an environment
$h^0 \leftarrow \mathrm{CNN}(s)$
for $a^1 = 1, 2, \cdots, n$ **do**
  $h^1_{pred} \leftarrow \mathrm{Predictor}(h^0, a^1)$
  for $t = 2, 3, \cdots, M$ **do** ▷ predict for M steps
    $a^t_{pred} \leftarrow \mathrm{Sample}(\hat{\pi}(h^{t-1}_{pred}))$
    $h^t_{pred} \leftarrow \mathrm{Predictor}(h^{t-1}_{pred}, a_{pred})$
  end for
  $encoded_{a^1} \leftarrow \mathrm{Encoder}(h^M_{pred}, h^{M-1}_{pred}, \cdots, h^1_{pred})$
end for
$f \leftarrow \mathrm{concat}(h^0, encoded_0, encoded_1, \cdots, encoded_n)$
$\pi \leftarrow \mathrm{softmax}(W_\pi f + b_\pi)$
$V \leftarrow W_V f + b_V$

In the early stage of the training, a next state predicted by the Predictor tends to be inaccurate, but since hidden state h of the current state is combined and used for policy making, a reasonable policy can be learned based on h. Therefore, h tends to become a useful representation for learning a policy as in model-free methods. As learning progresses, h becomes a better representation and the accuracy of the Predictor also improves. After that, it is possible to learn better policies based on predicted states of the Predictor than solely based on h.

The Predictor and the vector representations of actions are trained by their experience, which means the actions actually taken by the agent and the states actually observed by the agent during training. We used the Huber Loss be-

Algorithm 2 policy and value function (full model)

o is an observation from an environment
$h^0 \leftarrow \mathrm{CNN}(s)$
$x^0 \leftarrow \mathrm{CNN}_{pred}(s)$
for $a^1 = 1, 2, \cdots, n$ **do**
  $x^1 \leftarrow \mathrm{Predictor}(x^0, a^1)$
  $h^1_{pred} \leftarrow W_{P_{out}} x^1 + b_{P_{out}}$
  for $t = 2, 3, \cdots, M$ **do** ▷ predict for M steps
    $a^t_{pred} \leftarrow \mathrm{Sample}(\hat{\pi}(h^{t-1}_{pred}))$
    $x^t \leftarrow \mathrm{Predictor}(x^{t-1}, a_{pred})$
    $h^t_{pred} \leftarrow W_{P_{out}} x^t + b_{P_{out}}$
  end for
  $encoded_{a^1} \leftarrow \mathrm{Encoder}(h^M_{pred}, h^{M-1}_{pred}, \cdots, h^1_{pred})$
end for
$f \leftarrow \mathrm{concat}(h, encoded_0, encoded_1, \cdots, encoded_n)$
$\pi \leftarrow \mathrm{softmax}(W_\pi f + b_\pi)$
$V \leftarrow W_V f + b_V$

tween a predicted representation and the corresponding representation of the actual state as the prediction error, following the Universal Planning Networks (UPN) (Srinivas et al. 2018). In this paper, IMPALA (Espeholt et al. 2018) was used for training, and since the training batches of IMPALA already contain states in chronological order as shown in Figure 1, additional information is unnecessary for the training of state prediction. In addition, since the hidden state h, which is the target value for the Predictor, is necessary to calculate the value function and the policy function, training the policy and the Predictor at the same time eliminates redundant calculations. When calculating the error of state prediction, h and the parameters of the CNN are treated as constant, and the gradients for the CNN parameters are not calculated. This is because h is expected to be a useful representation for solving the task. If the CNN is changed by prediction error, h may become a representation that is easy to predict. Conversely, when calculating the error of reinforcement learning by the value function and the policy function, the parameters of the Predictor and the vector representations of actions are treated as constants.

Regarding the rollout policy $\hat{\pi}$, like I2A, training was performed by giving the following distillation error:

$$L_{dist}(\pi, \hat{\pi}) = \sum_{a=1}^{n} \pi(a|h) \log \hat{\pi}(a|h) \,,$$

so that π and $\hat{\pi}$ obtained from h in the training data become closer. Also in this case h was treated as a constant.

The whole loss value of training data is as follows:

$$L = L_{RL} + L_{pred} + L_{dist} \,,$$

where L_{RL} is the reinforcement learning loss, including the loss of the policy and value functions, and the entropy loss, which is calculated like normal IMPALA. Its gradient is as follows:

$$\frac{\partial}{\partial \theta} L = \frac{\partial}{\partial \theta_{RL}} L_{RL} + \frac{\partial}{\partial \theta_{pred}} L_{pred} + \frac{\partial}{\partial \theta_{\hat{\pi}}} L_{dist} \,.$$

In the above equation, θ_{RL} includes the parameters of the CNN, the Encoder, π and V. θ_{pred} includes the parameters of the Predictor and the vector representation of actions.

Since h is the representation only suitable for obtaining the policy and value functions, it is conceivable that in some environments, information in h becomes insufficient to calculate h_{pred} of the next state, and the prediction of Predictors can be impossible. Therefore, a Predictor as shown in Figure 3 can be used to enable more accurate next state prediction. In this paper, we call the model using this Predictor a *full model*. In the full model, h_{pred} is obtained by not forcing h to be the hidden state of the Predictor, but applying the output layer P_{out} after the Predictor output. Moreover, since h cannot be used as the initial value of the hidden state of the Predictor, another convolutional neural network for obtaining the initial value of the Predictor from an observation o is required. The algorithm of the full model is shown in Algorithm 2.

Since the proposed method encodes predicted states using a LSTM network as in I2A, it allows the agent to learn robust handling of prediction data obtained from an inaccurate environment model. In addition, since low-dimensional vector representations extracted from images are used for prediction without directly inputting images to the environment model, the computational cost can be suppressed. There are components similar to existing methods in terms of individual devices, but the proposed architecture realizes simultaneous and efficient training of the representation that is suitable for tasks, how to handle inaccurate environment models and the state transition of the environment based on the learned representation. This is the largest difference between existing methods and our architecture.

Experiments

In order to assess the effectiveness of the proposed method, we trained agents using the proposed architecture and model-free agents in Sokoban and compared the results. Sokoban is a kind of puzzle game, whose goal is to move all the boxes to target positions by moving a character in the environment based on a grid world. The character can push a box but cannot pull it, and hence it is necessary that the agent sufficiently forecasts the future states.

In Racanière et al. (2017), the experiment was carried out using the same Sokoban game. Most of the parameters and the settings of the experiment follow theirs. Images such as those shown in Figure 4 were used as observations for agents. In this experiment, we used automatically generated 10^6 levels with four boxes in a 10×10 grid world (borders, whose width is one, are always filled with walls). If agents could not solve a level in 120 steps, they gave up and went to another level. Agents can receive a reward of -0.1 for each step, 1 when putting a box to a target position, -1 when removing a box from a target position, and 10 when solving a level.

For all methods other than I2A, we obtained a h with 512 dimension using a CNN following Racanière et al. (2017), except that the activation functions are replaced with Leaky ReLU. For the model-free method, h was directly input to fully connected networks, and policy and value functions

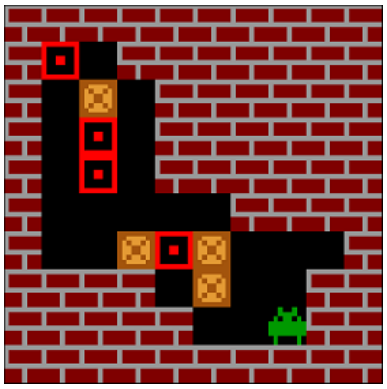

Figure 4: An example of the Sokoban game. Agents move the green character in four directions, up, down, left and right, and put all the brown boxes on the red target positions. Each observation is an image, whose size is 80×80.

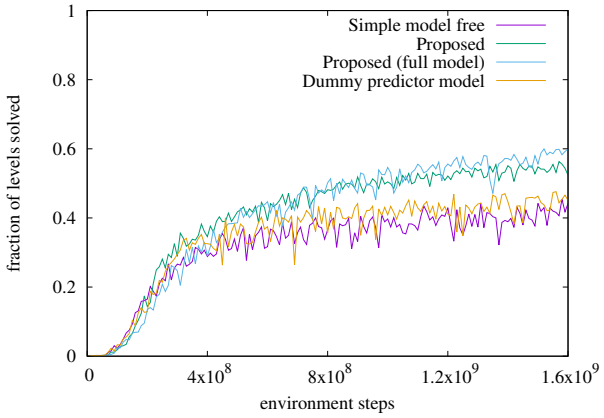

Figure 5: Scores of the game of Sokoban per trained environment steps.

Table 1: Number of parameters

Model	RL	Prediction
Simple model free	1.26×10^6	N/A
Proposed	3.37×10^6	2.10×10^6
Proposed (full model)	3.37×10^6	3.36×10^6
Dummy predictor model	3.37×10^6	N/A
I2A	6.02×10^6	1.32×10^6

were calculated. For *Proposed* and *Proposed (full model)*, each action had a 512 dimensional vector representation, the input size and hidden layer size of the Predictor and Encoder were all set to 512, and the number of prediction steps was five. IMPALA was used for the training, and the number of steps for advantage calculation was 12. RMSprop (Tieleman and Hinton 2012) was used for the optimizer.

In order to assess the contribution of prediction, we also tested *a dummy predictor model*, which was almost the same as our proposed model, but L_{pred} was set to zero and the Predictor was never trained.

The number of parameters of each model is shown in Table 1. *RL* in the table means the number of parameters trained by L_{RL}, and *Prediction* means the number of parameters used for prediction. The parameters for the rollout policies are included in the *Prediction*.

Figure 5 shows the fraction of levels that the agents were able to solve as a result of training. Each time agents were trained for a certain number of steps, we gave 10^4 levels and evaluated the solved rate. We trained agents twice for each model and averaged them. Training of the model-free method was unstable and the value of the value function diverges three of five times of training procedures, and therefore we used the result of the two training procedures which were normally completed. The result shows that our proposed methods achieve higher scores than the model-free method. Also, the score of the dummy predictor model is almost the same as the socre of the simple model-free method, suggesting that the main reason why our models outperform the model-free method is not the increase in the model size but the prediction. In this experiment, the difference between the simple proposed model and the full proposed model was small. Since the error on the next state prediction does not affect the expression of h, prediction accuracy might not be obtained by the simple model. This does not happen in this experiment.

Table 2: Training speed

Model	Training speed (Steps/s)
Simple model free	2.68×10^4
Proposed	9.22×10^3
Proposed (full model)	7.23×10^3
I2A	8.07×10^2
I2A (with env model learning)	6.44×10^2
Proposed (small batch size)	5.75×10^3

The training speed of each method is shown in Table 2. The experiment was carried out on a machine having a CPU with more than 6 cores / 12 threads and a GPU board of GeForce GTX 1080 Ti. We see that the proposed model can be trained with about three times more computational costs than the simple model-free method. For I2A, we measured the calculation time in the case of traing with a pre-trained environment model following Racanière et al. (2017) and joint learning of the environment model as our proposed method. We see that the proposed method can be trained an order of magnitude faster than I2A. Since I2A requires much GPU memory, we could not use the same batch size for I2A as other methods. Therefore, we also tested our proposed method with the same batch size as that of I2A, which is noted as *Proposed (small batch size)* in Table 2. The relationships between the scores and training times are shown in Figure 6. Note that I2A in this figure means I2A with joint learning of the environment model.

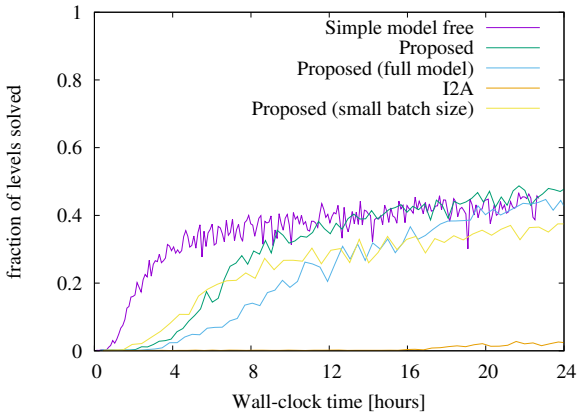

Figure 6: Scores of the game of Sokoban and training time.

Related Work

Model-based reinforcement learning with low-dimensional representations

Reinforcement Learning with Hidden Layer Predictor (RL-HLP) (Kameko et al. 2017) allows agents to learn a policy using future states by learning a state transition model of the hidden layer of an A3C agent. They obtain a convolutional neural network $h^t_{A3C} = \mathrm{CNN}(s_t; \theta_{cnn})$ and a feed forward neural network $\pi^t = \mathrm{Softmax}(\mathrm{FF}(h^t_{A3C}; \theta_\pi))$ from a pre-trained agent of A3C. Using these networks, they train the network that predicts the next hidden layer $h^{t+1}_{pred,a}$ from the current hidden layer h^t_{A3C} and the current action a. However, since $\mathrm{CNN}(s_t; \theta_{cnn})$ and $\mathrm{FF}(h^t_{A3C}; \theta_\pi)$, which are used for prediction, are fixed during the training of RL-HLP, there is the problem that the upper limit of the performance is determined by the pre-trained A3C agent.

World Models (Ha and Schmidhuber 2018) are another model-based architecture. Their World Model encodes images obtained as observations from an environment with a Variational Autoencoder (Kingma and Welling 2014). Therefore, in the World Model, vector representations tend to obtain features for reconstructing images, while in RL-HLP, vector representations tend to obtain information necessary for achieving the task. Since features in the World Model do not depend on the task, the method has an advantage of the ability to reuse a trained environment model even if the rewards of reinforcement learning changes. However, the information necessary for achieving the task can be different from the features made by the Variational Autoencoder, and hence, the representation of the World Model can be inefficient in terms of achieving the task. To train the vector representations and the state transitions in the World Model, they use observations gathered by random agents beforehand. Thus, it is difficult to respond to latter phases of the game. Ha and Schmidhuber (2018) have proposed a method that repeats the training of the Variational Autoencoder and the state transition model with observations generated by the trained agent.

Buesing et al. (2018) proposed and compared several environment models with compact state representations. They use decoders which convert low-dimensional states to observation images, and train environment models to be able to predict the next observations. They reported that their agents, which use their proposed environment models and I2A architecture, outperformed model-free baselines on the game Ms. Pac-Man. However, before training agents, they trained environment models with observations obtained by pre-trained policy. Therefore, their approach can have the same problem as the World Model.

Abstract representations of observation images from environments

In order to predict the next state in complex environments, there are several studies to convert high-dimensional image observations into low-dimensional intermediate representations.

In RL-HLP, an intermediate representation is obtained by using the CNN of pre-trained A3C agents. By using the Variational Autoencoder in the World Model, features as images on the game screen are used. In Universal Planning Networks (UPN) (Srinivas et al. 2018), by training to imitate experts' behavior, an intermediate representation suitable for solving the task is obtained. In Contrastive Predictive Coding (Oord, Li, and Vinyals 2018), a useful representation for next state prediction is obtained by combining prediction of several steps ahead and negative sampling. Burda et al. (2018) compare several intermediate representations such as those based on a Variational Autoencoder and inverse dynamics in the next state prediction for curiosity based learning.

The objective of reinforcement learning is to maximize cumulative rewards from environments, in other words, to achieve a task efficiently. An intermediate representation that does not depend on tasks such as the one learned with a Variational Autoencoder flexibly responds to changes in rewards, but on the other hand, it may be an inefficient representation from the viewpoint of task achievement. A UPN learns representations suitable for task achievement by imitating experts' behavior, but requires experts' trajectories.

In reinforcement learning, the distribution of observations can change through training by, for example, progressing the stage of the game. Therefore, the data which are obtained when agents cannot performe well are sometimes not very useful. With respect to the methods that preliminarily train intermediate representations, there is a risk that the performance depends on the distribution of the data obtained at the time of preliminary training. In the World Model, some methods to address this issue have been proposed by training agents and cellecting data repeatedly, but it decreases learning efficiency and it becomes necessary to adjust the repetition cycle of training.

Conclusion and Future Work

In this paper, we have proposed a new model-based reinforcement learning architecture which enables the prediction of the next state using low-dimensional representations suitable for task achievement. The agent jointly and efficiently learns an intermediate representation of game states,

which is useful for accomplishing the task, the state transition model of the environment based on the learned representation, and the policy based on the obtained environment model. We have shown that our architecture achieved higher scores than an existing model-free method in Sokoban, and that the computational cost of our architecture is significantly smaller than that for I2A.

Our architecture can perform training with realistic calculation time even if it is combined with a complicated method by suppressing the computational cost for prediction. For future work, it is conceivable to handle stochastic state transitions by combining with MDN-RNN (Ha and Eck 2017). Also, a combination with a network that predicts rewards that were not handled in this paper may be useful for improving performance.

References

Buesing, L.; Weber, T.; Racaniere, S.; Eslami, S.; Rezende, D.; Reichert, D. P.; Viola, F.; Besse, F.; Gregor, K.; Hassabis, D.; et al. 2018. Learning and querying fast generative models for reinforcement learning. *arXiv:1802.03006*.

Burda, Y.; Edwards, H.; Pathak, D.; Storkey, A.; Darrell, T.; and Efros, A. A. 2018. Large-scale study of curiosity-driven learning. *arXiv:1808.04355*.

Clavera, I.; Nagabandi, A.; Liu, S.; Fearing, R. S.; Abbeel, P.; Levine, S.; and Finn, C. 2018. Learning to adapt in dynamic, real-world environments through meta-reinforcement learning.

Clemente, A. V.; Castejón, H. N.; and Chandra, A. 2017. Efficient parallel methods for deep reinforcement learning. *arXiv:1705.04862*.

Espeholt, L.; Soyer, H.; Munos, R.; Simonyan, K.; Mnih, V.; Ward, T.; Doron, Y.; Firoiu, V.; Harley, T.; Dunning, I.; et al. 2018. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *Proceedings of the International Conference on Machine Learning (ICML)*.

Ha, D., and Eck, D. 2017. A neural representation of sketch drawings. *arXiv:1704.03477*.

Ha, D., and Schmidhuber, J. 2018. World models. *arXiv:1803.10122*.

Hochreiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural computation* 9(8):1735–1780.

Kameko, H.; Suzuki, J.; Mizukami, N.; and Tsuruoka, Y. 2017. Deep reinforcement learning with hidden layers on future states. *Computer Games Workshop at IJCAI*.

Kingma, D. P., and Welling, M. 2014. Auto-encoding variational Bayes. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing Atari with deep reinforcement learning. *NIPS Deep Learning Workshop*.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous methods for deep reinforcement learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 1928–1937.

Oord, A. v. d.; Li, Y.; and Vinyals, O. 2018. Representation learning with contrastive predictive coding. *arXiv:1807.03748*.

Racanière, S.; Reichert, D.; Weber, T.; Vinyals, O.; Wierstra, D.; Buesing, L.; Battaglia, P.; Pascanu, R.; Li, Y.; Heess, N.; et al. 2017. Imagination-augmented agents for deep reinforcement learning. In *Advances in Neural Information Processing Systems*, 5692–5699.

Srinivas, A.; Jabri, A.; Abbeel, P.; Levine, S.; and Finn, C. 2018. Universal planning networks: Learning generalizable representations for visuomotor control. In *Proceedings of the International Conference on Machine Learning (ICML)*, 4739–4748.

Sutton, R. S. 1990. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine Learning Proceedings 1990*. Elsevier. 216–224.

Tieleman, T., and Hinton, G. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning* 4(2):26–31.