Performance comparison of model-based and model-free deep reinforcement learning methods in video game environments

Bachelor's Project Thesis

Benjamin Hucko, s4715187, b.hucko@student.rug.nl
Supervisors: J.D. Cardenas-Cartagena, M.Sc.

Abstract: Deep reinforcement learning methods require millions of samples to converge. Gathering this many samples is expensive. Model-based algorithms claim to converge with less samples. In this theses I test how online model-based algorithms work on environments with pixel-based states. Furthermore, I test whether model-based algorithms can work on an environment where the entire environment is not in frame. I compare the SimPLe algorithm (Kaiser, Babaeizadeh, Milos, Osinski, Campbell, Czechowski, Erhan, Finn, Kozakowski, Levine, Mohiuddin, Sepassi, Tucker, and Michalewski, 2020) and the Dreamer algorithm (Hafner, Lillicrap, Ba, and Norouzi, 2020a). Model-based algorithms converge to lower performance than when the agent is trained directly with the environment, i.e. with a model-free method. The algorithms struggle to learn the environment where the entire environment is not in frame. SimPLe and Dreamer can be more sampling efficient when the environment is trained partially offline and the frames used for the initial offline training are ignored.

1 Introduction

In this thesis I explore whether model-based reinforcement can outperform model-free approaches in video games. I compare performance by looking at convergence speed and sample efficiency, i.e. how much time and how many interactions with the environment were needed for convergence.

1.1 Motivation

A pilot first needs to complete several flights in a simulation before they are allowed to fly a real airplane. There are three reasons for that. Firstly, practice flights are expensive. When the airplane is in air, it consumes gas and wears out its engines. Secondly, there is a risk of the airplane crashing. This risk is especially high when the airplane is in the hands of an untrained pilot. Thirdly, each flight requires a lot of preparation time. The pilot needs to travel to an airport and the plane needs to be prepared. Therefore, the pilot trains in a simulation, even if the practice is of lower quality.

The same trade-off happens in Reinforcement learning. Model-free reinforcement learning algorithms provide higher quality data. Model-based reinforcement learning algorithms provide data at a cheaper price and lower risk. The benefit of model-based algorithms is clear for the field of robotics (Polydoros and Nalpantidis, 2017). It is intuitive that training a robot on a simulation is much cheaper than training them in the real world. Model-based approaches might also benefit agents trained to interact in virtual environments. More specifically, agents trained to play video games. This is because video games cost significant computational power to run. The interactions take longer than they would if the agent interacted with the transition model. Furthermore, we are also limited in how many interactions we can run in parallel. Interaction with the environment takes more computational resources from GPU than an interaction with the transition model (Kaiser et al., 2020) (Hafner, Lillicrap, Norouzi, and Ba, 2020b).

Most model-free deep reinforcement learning algorithms require millions of interactions to converge. When we use a model-based reinforcement learning algorithm, we can gather these interactions from the transition model. Each interaction will take less time and more can be computed in parallel. This suggests that model-based reinforcement learning methods can improve the convergence speed of the agent. This benefit is even more important for environments that have expensive interactions, such as certain video games.

1.2 State of art

Atari environments have been a staple when it comes to Reinforcement learning benchmarks. Deep reinforcement claims to overcome human performance in the Atari environments (Mnih, Kavukcuoglu, Silver, Graves, Antonoglou, Wierstra, and Riedmiller, 2013). This feat was achieved with Deep Q-Networks (DQN). Deep reinforcement learning algorithms have improved since then. Recent algorithms show even greater performance (Fan, 2021). Rainbow is an algorithm developed by optimizing DQN and achieves state of the art performance on a wide range of Atari games (Hessel, Modayil, Van Hasselt, Schaul, Ostrovski, Dabney, Horgan, Piot, Azar, and Silver, 2018). Policy gradient algorithms have also shown great performance. Many state of art algorithms used as benchmarks follow the actor-critic architecture (Fan, 2021). Proximal policy optimization (PPO) follows said architecture, and achieves comparable performance to Rainbow on many atari games (Kaiser et al., 2020).

Model-based methods work together with the model-free methods. Simulated Policy Learning (SimPLe) and Dreamer provide a transition model, i.e. a model that simulates how the environment changes with the actions of the agent. The transition model can be used by agent trained using any of the model-free algorithms. The main benefit of using SimPLe is sample efficiency (Kaiser et al., 2020). Dreamer needs more samples to converge (Hafner et al., 2020b). Dreamer shows final performance comparable to model-free methods (Hafner et al., 2020b).

Overall model-based algorithms do not achieve the same final performance as the model free algorithms in the Atari environments. The trend is that the converged performance is lower and the sample efficiency is higher (Fan, 2021). This is supported by the results of Kaiser et al. (2020) and Hafner et al. (2020b). They used the Atari environments as benchmark. However, their results are usually only cited and not replicated (Fan, 2021).

1.3 Research questions

The main hypothesis that I focus on in this thesis is the following question. Are model-based algorithms are more sampling efficient than model-free algorithms? In certain model-based approaches transition models are trained offline, i.e. with randomly sampled data and before the training of the agent. If the transition model of model based algorithms have any part of training happening offline, I count the data used for offline training when I evaluate the sampling efficiency of the algorithm.

Furthermore I want to observe whether model-based algorithms are able to converge in dynamic environments, i.e. environments where the entire environment does not fit the frame. By convergence I consider any significant improvement to the performance of the agent. Even if the performance does not reach the performance of model-free algorithms, the fact that a single transition model is able to train many model-free agents for cheap makes it worthwhile to consider using them to boost the initial training of the agent. This leads to another hypothesis. Are model-based reinforcement learning methods able to converge in dynamic environments?

I also compare the Dreamer algorithm and the SimPLe algorithm. These two algorithms use different architecture for transition model. I compare and analyze which of the two algorithm performs better. I compare them based on sample efficiency and converged performance. This leads to another hypothesis. Does the Dreamer algorithm outperform the SimPLe algorithms, when they have similar architecture and the same preprocessing?

Finally, I compare different modifications that can be done to the model-based algorithms. I compare the performance of model-based algorithms when the agent also learns from the actual environment and when the agent only learns from the simulated environment. Learning from the real environment on top of learning from the transition model is called hybrid learning. Hybrid learning has shown to boost the performance of model-based algorithms, e.g. Dyna (Sutton and Barto, 2014). I analyse what is the impact of predicting episode termination by the transition model. In certain environments, e.g. Breakout, the episode length heavily correlates with the episode returns. In such environments predicting the episode termination should improve the convergence of the algorithm (Hafner et al., 2020b). Therefore my final two hypothesis are related to these modifications. Does using the interactions from the environment to train the agent improve the performnace of model-based algorithms? Does predicting episode termination in environments where the episode termination correlates with episode returns improve the performance of SimPLe?

1.4 Contribution

Firstly I modify the model-based algorithms. These modifications aim to improve the performance of SimPLe and Dreamer. First modification is that I make the agent learn from the real environment on top of learning from the simulated environment. I apply these modification to both approaches. Next I modify the SimPLe approach to predict episode termination. These modification were not in the original algorithms and therefore are unique contributions. I use the results to answer whether model-based algorithms can outperform model-free algorithms.

Secondly I compare the performance of the SimPLe and the Dreamer algorithms. Hafner et al. (2020b) compares the two algorithms. However, the original implementations have many differences that are not strictly related to the algorithms. Dreamer and SimPLe have different preprocessing (Fan, 2021). Dreamer and SimPLe have different agents (Hafner et al., 2020a) (Kaiser et al., 2020). Dreamer predicts episode terminations (Hafner et al., 2020b), SimPLe does not predict episode termination (Kaiser et al., 2020). I modify the implementations to not have these differences. I compare the results achieved both by the modified SimPLe and Dreamer.

Thirdly I observe how SimPLe and Dreamer behave in more complex environments. I use the Mario environment. The Mario environment has a frame that follows the agent. I.e. when the agent moves to the right, the frame shifts to the right. This shows a new part of the environment. A better performing agent sees parts of the environment that a worse performing agent does not see. This means that the transition model must be trained online, i.e. trained during the training of the agent. I observe how the SimPLe and Dreamer algorithms perform in the Mario environment. I use the results to answer whether model-based algorithms can be scaled to dynamic environments.

2 Theoretical background

In reinforcement learning problems, we have an agent and an environment. The environment is treated as Markov's decision process (Sutton and Barto, 2014). That means that the environment consists of a state space $\mathcal{S}$ and action space $\mathcal{A}$. The environment provides the agent with information it can perceive, i.e. the state $s \in \mathcal{S}$. The agent performs an action in the environment, i.e. $a \in \mathcal{A}$. The environment also provides feedback to the actions taken by the agent in form of a reward. The cumulative rewards agent receives before reaching a terminal state is a return G_t. The agent keeps track of these returns using a value function $V(s) \doteq \mathbb{E}\left[G_t \mid S_t = s\right]$ or an action-value function $Q(s,a) \doteq \mathbb{E}\left[G_t \mid S_t = s, A_t = a\right]$. The agents goal is to maximize the reward it receives. For the agent to perform well, it needs to learn a policy $\pi(a \mid s)$. The policy determines what actions the agent takes in which situations (Sutton and Barto, 2014). The agent can use different algorithms to learn this policy. These algorithms primarily differ in their convergence speed and computational cost. The algorithms can be separated into model-free algorithms and model-based algorithms (Sutton and Barto, 2014). Model-free algorithms train the agent with the samples from the environment, model-based algorithms train the agent by sampling a transition model.

2.1 Model-free

Model-free algorithms learn the policy with the information provided by the environment. The agent chooses actions and observes which actions give better rewards. Then the agent will take actions that give it better rewards. In the most simple case this is done using tabular methods. In tabular methods, the agent keeps track of what happened when it took an action at a state. Than, when it visits the same state, it will make a decision based on its prior experiences in that state.

Tabular methods suffer when the state space is large. To make algorithms more viable on bigger state spaces, tabular methods evolved into function approximations (Sutton and Barto, 2014). Function approximation algorithms learn a function that helps agent make its decisions. The learned function can be the value function $V_\theta(s)$, the action-value function $Q_\theta(s,a)$, or the policy $\pi_\theta(a \mid s)$. The agent can learn more then a single function. Algorithms where the agent learns a value function or an action-value function alongside the policy are called Actor-Critic algorithms (Sutton and Barto, 2014). The actor-critic architecture splits the agent into the actor and the critic. The critic is used to train the actor. The actor interacts with the environment.

The traditional training loop of model-free reinforcement learning algorithms is as follows (Plaat, Kosters, and Preuss, 2020): The environment is initialized and wrapped in necessary wrappers. The agent is initialized with the action-space of the environment. The agent interacts with the environment and stores trajectories in $\mathcal{D}$. These trajectories are used to update the agent.

Algorithm 2.1 Model-free training loop

```
initialize environment $env$
initialize agent
while not converged do
    $\mathcal{D} \leftarrow$ data from $env$
    Use $\mathcal{D}$ to update agent
```

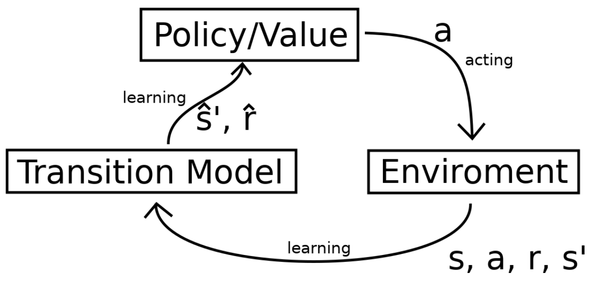

Figure 2.1: Model-based online training loop

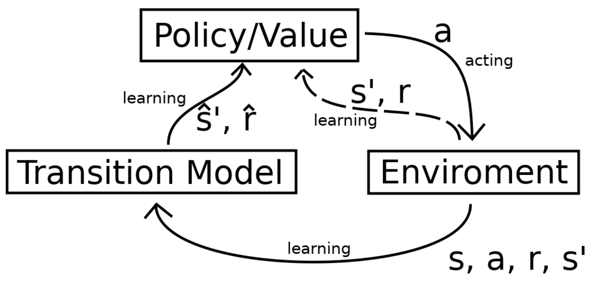

Figure 2.2: Hybrid online training loop

2.2 Model-based

Model-based algorithms learn the policy with the information provided by a transition model of the environment. The transition model of the environment can be either known or learned (Plaat et al., 2020). Known transition model can be used directly. This is the case for environments with known rules, e.g. chess, go, etc. Learned transition model needs to be trained to approximate the actual environment.
The agent can either use the transition model for planning, or use it for imagination (Plaat et al., 2020). In planning, the agent peeks into the future before choosing an action. The agent simulates several possible trajectories, and chooses an action that leads to the result it desires. The advantage of this approach is that the agent can better function in environments with sparse rewards. In imagination, the agent trains a policy function from the model of the environment. The agent interacts with the model of the environment the same way it would interact with the actual environment. The advantage of this approach is that the agent can replace sampling the environment with sampling the transition model. Sampling the transition model is cheaper (Plaat et al., 2020).

Transition models are trained online or offline, i.e. during the training of the agent or before the training of the agent (Kaiser et al., 2020) (Hafner et al., 2020a). Online learning follows the schematic shown in figure 2.1.

In hybrid learning, the data used to train the transition model is used to update agent as well. This means that the agent is trained both using a model-free and a model-based method. This method provides for faster convergence in model-based algorithms (Plaat et al., 2020). Schematic of hybrid learning can be seen in figure 2.2.

The online training loop for model-based algorithms is as follows: The environment is initialized

Algorithm 2.2 Model-based training loop

```
initialize environment $env$
initialize model of environment $env'$
initialize agent
while not converged do
    $\mathcal{D} \leftarrow$ data from $env$
    Use $\mathcal{D}$ to update $env'$
    $\mathcal{D}' \leftarrow$ data from $env'$
    if hybrid learning then
        Use $\mathcal{D}$ to update agent
    Use $\mathcal{D}'$ to update agent
```

and wrapped in necessary wrappers. The agent is initialized with the action-space of the environment. The model of the environment env' is the transition model of the model-based algorithm. The agent interacts with the environment and stores trajectories in $\mathcal{D}$. These trajectories are used to update transition model env'. The agent interacts with the transition model env' to gather additional trajectories. The trajectories from the simulated environment are stored in $\mathcal{D}'$. The agent is updated using trajectories from the transition model env' stored in $\mathcal{D}'$. If hybrid learning is turned on, agent is also updated using trajectories from the environment stored in $\mathcal{D}$.

2.3 PPO

PPO follows an actor-critic architecture. The critic approximates the value function. The actor approximates the policy. The critic calculates the episode returns and uses that to update the value function estimate using mean squared error. The critic uses the bootstrapped value function to calculate the estimated advantage (Schulman, Moritz, Levine, Jordan, and Abbeel, 2018), i.e. difference between the action-value and the bootstrapped value function for state s. The actor uses the estimated advantages to update the policy with the following formula (Schulman, Wolski, Dhariwal, Radford,

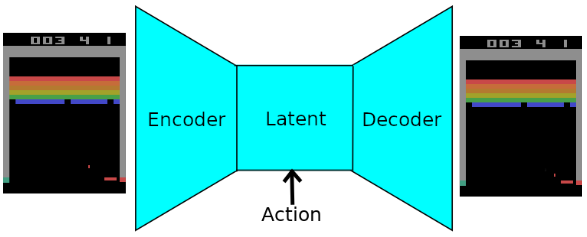

Figure 2.3: The autoencoder transition model used in the SimPLe algorithm. This is a simplified drawing of the autoencoder presented by Kaiser et al. (2020).

and Klimov, 2017).

$$\mathrm{L} = \min\left(\frac{\pi_\theta\,(a,s)}{\pi_{\theta_{old}}\,(a,s)}\mathrm{A}\,(s,a)\,,\mathrm{g}\,(\epsilon,\mathrm{A}\,(s,a))\right) \tag{2.1}$$

$$g(\epsilon,A)\begin{cases}(1+\epsilon)A & A\geq 0\\(1-\epsilon)A & A<0\end{cases} \tag{2.2}$$

$$\theta \leftarrow \theta + \alpha\nabla_\theta\frac{1}{|D|\,\tau}\sum_\tau\sum_{t=0}^{T}\mathrm{L}(s,a,\theta,\theta_{old}) \tag{2.3}$$

The policy $\pi_\theta\,(a,s)$ represents probability of taking action a in state s when following policy π_θ. The loss is the advantage multiplied by the ratio of current probability of taking action a in state s and old probability of taking action a in state s. The loss is clipped using a clipping function $g\,(\epsilon,\mathrm{A}(s,a))$. The resulting loss is the minimum of the original loss and the clipped loss. The parameters θ are updating using a gradient ascent on the loss function.

2.4 SimPLe

Simulated policy learning (SimPLe) algorithm uses an autoencoder as its transition model, which can be seen in figure 2.3. The autoencoder consists of an encoder and a decoder. The encoder consists of convolutional layers. The encoder encodes a stack of frames into latent space, i.e. lower dimensional space. Actions taken by the agent are injected into the latent. The decoder consists of transposed convolutional layer. The decoder uses the latent of the agent to create per pixel logits. These logits are used to predict the next frame. Furthermore, the pixel logits and the latent are combined to predict the reward. The frames are predicted autoregressively. Each predicted frame is put on top of the input stack stack. This is used to generate trajectories.

$$\mathrm{L}_o = \frac{\sum_{h,w}}{|\mathcal{H}|\,|\mathcal{W}|}\sum_{p=0}^{255} s'_{h,w,p}\cdot\log\left(\hat{s'}_{h,w,p}\right) \tag{2.4}$$

$$\mathrm{L}_r = \sum_r r\cdot\log\left(\hat{r}\right) \tag{2.5}$$

$$\mathrm{L} = -\left(\alpha\mathrm{L}_o + \beta\mathrm{L}_r\right) \tag{2.6}$$

The observation loss is average per pixel cross-entropy loss. The activation of the pixel p consists of integers in the range $[0,255]$. All the possible rewards are integers. The reward loss is calculated using using the cross-entrophy loss. The losses are combined using convex combination with weights α and β. The loss is multiplied with -1 to achieve proper convergence.

The main drawback of SimPLe is that the amount of information it can remember is limited by the size of the input stack. Information beyond the size of the input stack is forgotten. Therefore, SimPLe cannot learn transitions when action affects the environment in amount of frames beyond the size of the stack. To predict stochastic environments, the autoencoder needs to be modified. Kaiser et al. (2020) proposes a stochastic SimPLe for stochastic environments. In stochastic SimPLe there is a variational autoencoder (Babaeizadeh, Finn, Erhan, Campbell, and Levine, 2018) predicting mean and varience of stochastic latent. The lantent is then sampled and discretized into bits (Kaiser and Bengio, 2018). The discrete latent is injected into the latent of the deterministic autoencoder. This simulates stochastic transitions in the environment. During inference, the variational autoencoder is replaced by a LSSM that predicts the discrete latent autoregressively (Kaiser et al., 2020).
Main benefits of SimPLe is improved sampling efficiency. This is because the agent can converge using samples from the transition model. The samples from the transition model are cheaper. The transition model is able to compute multiple trajectories in parallel. This allows multiple agents to interact with the simulated environment. This leads to faster convergence (Kaiser et al., 2020).

2.5 Dreamer

Dreamer uses a modular architecture. The transition model consists of the transition module, termination module and the reward module. The transition module is a recurrent stochastic state machine (RSSM) (Hafner, Lillicrap, Fischer, Villegas, Ha, Lee, and Davidson, 2019). The architecture of RSSM is shown in figure 2.4. RSSM predicts the state autoregressively using the action of the agent as input. The RSSM

Encoder module	$o' = \mathrm{f}_\theta(o_t)$
Representational module	$p_\theta(s_t \mid s_{t-1}, a_{t-1}, o'_t)$
Transition module	$p_\theta(s_t \mid s_{t-1}, a_{t-1})$
Reward module	$p_\theta(r_t \mid s_t)$
Observation module	$p_\theta(o_t \mid s_t)$
Discount module	$p_\theta(\gamma_t \mid s_t)$

Table 2.1: Separation of the dreamer modules (inspired by Hafner et al. (2020a) and Hafner et al. (2020b). Reward and discount modules predict the output of the transition model. Representational module, encoder module and observational module are used to train the transition model. The transition module allows for recurrent rollouts.

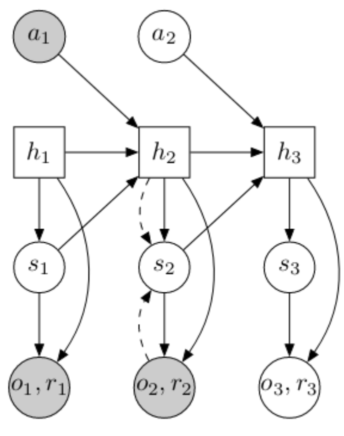

Figure 2.4: Reccurent stochastic space machine schematic (RSSM) (Hafner et al., 2019). The deterministic state, i.e. carry of the GRU, is denoted with h_t. The stochastic state of the SSM is denoted with s_t. Dashed line represents sampling from a distribution predicted by the input. The version of RSSM used in the transition module does not use observations o_t.

improves on exisiting recurrent architectures by adding stochastic transitions. RSSM consists of a gated recurrent unit (GRU) and stochastic state machine (SSM). GRU propagates the deterministic representation of the state. The input to the GRU is the stochastic state and the previous deterministic state. SSM propagates the stochastic representation of the state. SSM predicts the mean and variance of a normal distribution. The stochastic state is sampled from that normal distribution. The input to the SSM is the previous stochastic state and the current deterministic state. The state used by the agent, the termination module, and the reward module consists of the stochastic state and the deterministic state.

The state that the agent uses is different than the state from the real environment. To make this difference more explicit, the state from the environment is referred to as observation in the context of dreamer. Because of this difference, the agent unable to act in the real environment. To allow the agent to act in the real environment, Hafner et al. (2020a) uses a representation model. Representation model consists of representation module, encoder module, reward module and termination module. Encoder module encodes the observation from the environment. The representation module is very similar to the transition module. The only difference is that in the representation module, the encoded observation is combined with the deterministic state to predict the stochastic state. This results in more accurate prediction of the stochastic state.

The representation model and the transition model are trained using reconstruction. The idea behind this is conceptually similar to an autoencoder, i.e. store enough information in the latent space to reconstruct the image. The representation model tries to reconstruct the image. To reconstruct the image, the observation module is added to the representation model. The transition model shares the weights for the GRU with the representation model. The stochastic state predicted by the transition model is trained using kl loss to match the stochastic state of the representation model. The reconstruction loss Hafner et al. (2020a) uses is following:

$$\mathrm{L}_{rec} = -\frac{\sum_t}{|\mathcal{T}|}\left(L_o^t + L_r^t + L_\gamma^t + L_T^t\right) \tag{2.7}$$

$$\mathrm{L}_o = \frac{\sum_{h,w}}{|\mathcal{H}|\,|\mathcal{W}|}\sum_{p=0}^{255} s'_{h,w,p} \cdot \log\left(\hat{s'}_{h,w,p}\right) \tag{2.8}$$

$$\mathrm{L}_r = \sum_r r \cdot \log(\hat{r}) \tag{2.9}$$

$$\mathrm{L}_\gamma = \sum_\gamma \gamma \cdot \log(\hat{\gamma}) \tag{2.10}$$

$$\mathrm{L}_T = D(p_\theta(s_t \mid s_{t-1}, a_{t-1}, o_t) \parallel p_\theta(s_t \mid s_{t-1}, a_{t-1})) \tag{2.11}$$

The observation and reward loss are the same as for the SimPLe algorithm. Termination loss is denoted by L_γ. The termination loss is calculated using the cross-entropy loss. The transition loss L_T minimizes the divergence between representation module and transition module. The reconstruction loss is a summation of all four losses. The reconstruction loss is multiplied with -1 to achieve proper convergence.

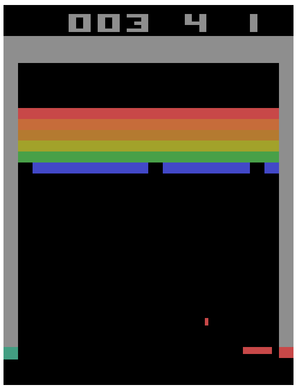

Figure 3.1: Frame from the breakout environment (Towers et al., 2023)

$\mathcal{S}$	$\{s \in \mathbb{Z}^{\mathcal{H}\times\mathcal{W}\times\mathcal{C}}, s_{h,w,c} \in [0,255]\}$
$\mathcal{A}$	$\{NOOP,\ FIRE,\ LEFT,\ RIGHT\}$
$\mathcal{R}$	$\{0,1\}$

Table 3.1: The state space, action space and possible rewards in the Breakout environment. In the original environment $\mathcal{H} = \{1, \cdots, 210\}$, $\mathcal{H} = \{1, \cdots, 210\}$, $\mathcal{C} = \{1, 2, 3\}$. This is adjusted by the wrappers. The NOOP action means no action and FIRE action starts the game.

3 Methods

3.1 Environment

3.1.1 Breakout

I use the gymnasium API (Towers et al., 2023) to interface the Breakout and the Mario environments. I take the breakout environment directly from the Atari learning environment database. I use the fifth version of the environment. In the breakout environment the agent controls a platform. The agent uses this platform to bounce a ball in a pong-like fashion. The agent receives a reward for destroying bricks with the ball. The agent looses a life when the ball is not bounced up by the platform and falls out of frame. The episode terminates when the agent looses all 3 lives.

$\mathcal{S}$	$\{s \in \mathbb{Z}^{\mathcal{H}\times\mathcal{W}\times\mathcal{C}}, s_{h,w,c} \in [0,255]\}$
$\mathcal{A}$	$\{RIGHT,\ RIGHT + JUMP\}$
$\mathcal{R}$	$\{r \in \mathbb{Z}, r \in [-15, 15]\}$

Table 3.2: The state space, action space and possible rewards in the Mario environment. The state space and the action space is filtered by wrappers. The filtered state and action space is $\mathcal{H} = \{1, \cdots, 84\}$, $\mathcal{H} = \{1, \cdots, 84\}$, $\mathcal{C} = \{1\}$. The original action space is too long to include.

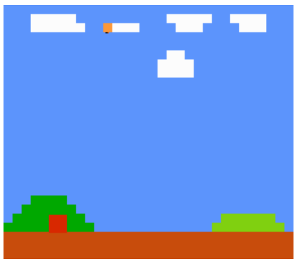

Figure 3.2: Frame from the Mario environment (Kauten, 2018)

3.1.2 Mario

I take the Mario environment from Kauten (2018). The environment provides rendering styles. I use the rectangle rendering style. This style limits the complexity of the environment by rendering the image as mono rectangles. The agent controls the Mario character within the environment. The agent looses lives when it hits a hostile mob or falls into a pit. The episode terminates when the agent looses all lives. The agent is rewarded for moving right. The agent is punished, i.e. receives negative reward, when it looses a life. The calculation of the reward follows the following rules.

- v: reward for moving right. $v = x_1 - x_0$ where x_0 is the x-coordinate before taking action a and x_1 is the x-coordinate after taking action a.
- c: punishment for slow progress $c = c_0 - c_1$ where c_0 is the game clock before taking action a and c_1 is the game clock after taking action a term c is always constant
- d: a penalty for dying upon death $d = -15$

$r = c + v + d$. The reward is clipped into the range $(-15, 15)$ (Kauten, 2018). The reward is rounded to nearest integer.

3.1.3 Wrappers

I use the wrappers provided by gymnasium API provides to modify the data from the environments. I scale the observations to 84×84. I change the observations from rgb to grayscale. Both these steps decrease the dimensions of the state space. I set up a frame skip of 4 frames. This means that each action is repeated for 4 frames and the reward is the sum of the rewards during those 4 frames. In the case of model-free algorithms and SimPLe, I add a wrapper that stacks the last 4 frames. This allows the transition model and the agent to use the information from most recent frames to make a more informed decision. E.g. the agent which direction an object is moving based on where it was the in last frame and where it is in the current frame.

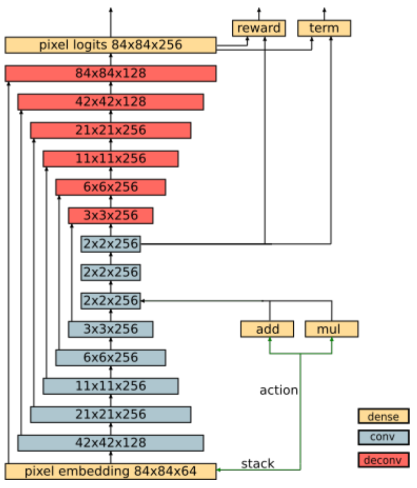

Figure 3.3: **Model following the SimPLe architecture. The model implementation is closely related to the implementation proposed by Kaiser et al. (2020)**

I reshape the frame stack to have the 4 frames in the channel dimension. Frame stacking is not needed for Dreamer, because Dreamer is based on a recurrent architecture. Therefore the information from past frame is already in the transition model. This makes the frame stack superfluous (Hafner et al., 2020a).

3.2 Model architecture

3.2.1 SimPLe transition model

The architecture of the SimPLe transition model is in figure 3.3. Each convolution and transposed convolution is preceded with and layer normalization. The pixel embedding is a dense layer which has the same weights for each pixel. The pixel embedding weights the inputs from different frames. The pixel embeddings allows the model to allocates different weight to the data from different frames in the frame stack.
The convolution, transposed convolution, and dense layers are followed by ReLu activation function. The exception to this are the add and mul dense layers. The add layer does not have any activation funciton and the mul layer has a softmax activation funciton. I add the skip connections to the activation of the layer they are connected to. After I add the skip connections, I normalize the layer.
I inject the action into the latent of the autoencoder. I encode the action with one hot encoding. I match the dimensions of the encoded action to the latent dimension using the add and mul layers. The action is first multiplied with the latent and then added to the latent.
When predicting trajectories using the transition model, I turn off the dropout.

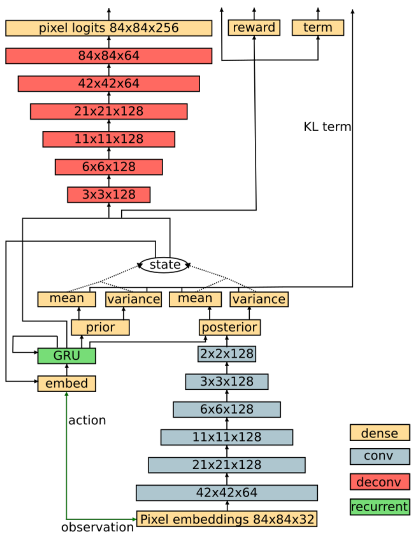

Figure 3.4: **Model following the Dreamer architecture (Hafner et al., 2020a). The architecture uses an encoder and decoder inspired from Kaiser et al. (2020)**

3.2.2 Dreamer transition model

The architecture of the Dreamer transition model and representation model is in figure 3.4. Each convolution and transposed convolution is preceded with layer normalization. The layer normalization for the first convolution layer is only possible because the pixel is embedded. Otherwise, for single feature pixels the layer normalization results in activation of 0. I do not use dropout, because the latent space of Dreamer is stochastic. Therefore adding additional stochasticity by using dropout layers is not necessary.
The convolution, transposed convolution, and dense layers are followed by ReLu activation function. The exception are the output layers, and the layers that predicts the mean and standard deviation. The layer predicting the mean has no activation function. The layer that predicts the standard deviation has a softplus activation function with minimum lower bound of δ_{min}. The stochastic state is sampled using reparameterization trick (Kingma and Welling, 2013). I.e. using $s_{stochastic} = \mu_\theta + \sigma_\theta \cdot \mathcal{N}(\mathbf{0}, \mathbf{I})$.
The dreamer transition model can be separated into different modules. This separation into different modules achieves two benefits. First benefit is that certain modules can be turned off when not needed. E.g. the observation module is only used during training. Second benefit is that we can make modules without a recurrent unit to work in parallel. This leads to a more efficient training. I first encode the observations with the encoder module in parallel. Then I predict the states sequentially using the representation module. Finally I predict the discount, termination and the reconstructed

observation in parallel. This limits the drawbacks of the recurrent architecture.
The transition module shares most layers and weights with the representation module. Therefore these two modules should not be thought of as separate. The transition module is a part of the representation module. The representation module wraps the environment. The agent interacts with the representation module when gathering samples from the real environment. Then when agent is learning from the transition model, it interacts with the transition module.
Initial stochastic and deterministic state is a vector with zeros. The stochastic and the deterministic state is reset when the environment terminates. This means that data after termination does not affect the data before termination. Empirically this results in more stable learning. When the agent interacts with the transition model, predicted termination plays similar role. When the transition model predicts termination, the reported state is not affected. However it is not fed autoregressively into the transition model. Instead, I feed the zero state into the model.

3.3 Transition model outputs

Image is represented using one integer per pixel. The integer has values from 0 to 255. The integer represents the grayscale value of the pixels. I predict the pixel activation using softmax distribution. I use probability distribution because there is no correlation for similar colours inside the environment for Mario and Breakout. Possible rewards from the environment are represented using a categorical distribution. Each reward has an associated integer. I represent termination as a categorical distribution with 2 possible categories, i.e. True and False. I use a categorical softmax for pixel activation, reward and termination. I predict the image, reward and termination by taking the most probable option. The loss function is the cross-entropy loss function. The cross-entropy loss compares categorical distribution predicted by the model with the actual result. The resulting loss is 0 when the predicted probability of the actual category is 100%. The loss for each of the outputs is clipped using $\mathrm{L} = \max(C, \mathrm{L})$. This means that we only require the model to predict the actual category with $1 - C$ probability.

3.4 Training of the transition model

I train the transition model using a random agent for the first N_{init} updates. Then the transition model is trained with the data gathered by the model-free agent. I use all gathered data when I update the model. The data is passed in batches to limit the memory requirements on the GPU. In the case of SimPLe, the sequence in which the data was gathered is not important. Therefore the data is randomly shuffled. In the case of Dreamer, the sequence in which the data was gathered is important. Therefore the data is not shuffled and training batches consist of the data from the same environment. Each update of the model consists of N_{epoch} epochs.

3.5 Training of the PPO agent

I use combined actor critic loss. I add the entropy of the agent policy as an additional loss term with weight $w_{entrophy}$. These two modifications modify the equation 2.3.

$$\mathrm{L_c} = \left(\hat{V}(s_t) - G_t\right)^2 \tag{3.1}$$

$$\mathrm{L_e} = -\sum_a \pi_\theta(a \mid s) \cdot \log\left(\pi_\theta\left(a \mid s\right)\right) \tag{3.2}$$

$$\mathrm{L} = L_a + L_c + w_{entropy} \cdot L_e \tag{3.3}$$

Using a shared loss for the actor critic allows me to use a combined architecture for the actor critic. Combined architecture requires less updates because of the shared parameters. The entropy loss helps the agent explore by dissuading policy that is too deterministic (Shen, 2024). The data is passed in batches to limit the memory requirements on the GPU. Each update of the model contains of N_{epoch} epochs. The data is passed to the network in random order. This decreases the risk of getting stuck in a local minimum.

3.6 Experiment details

I train the algorithms for 12 hours or for a maximum of 4000 updates. Update happens every 400 frames. If the agent reaches termination in any environment within the 400 episodes, I reset the environment and set the discount factor at the terminating state to 0. I do this so the data gathered after termination does not affect value of states before the termination. I train The transition model either fully online or partially online. The algorithm is trained fully online when $N_{init} = 0$. When the agent interacts with the transition model, I update the agent each 50 frames. I perform $N_{model_updates}$ updates using the transition model for each update using the environment. Updates using the transition model are not counted in the 4000 update limit.

3.7 Materials

I use Habrok cluster to train the agent. The agent is trained on the GPU. The agent is trained on a node with 4 NVIDIA A100 cards with 40GB memory each. The program is compiled on to the

parameter	description	value
$w_{entrophy}$	Weight of the entropy coefficient in PPO	0.01
N_{init}	Number of initial updates of the transition model	0, 20, 100
N_{epoch}	Number of training epochs	4
C	Lower bound of the cross-entropy loss	0.03
δ_{min}	Minimum standard deviation of Dreamer Gaussian distributions.	0.1
optimizer	Optimizer used to update the parameter	ADAM

Table 3.3: Hyperparameter table. In these table are all the hyperparameters mentioned in the method section of these thesis.

GPU using accelerated linear algebra open source software (XLA).

To train the models I use the jax library (Bradbury, Frostig, Hawkins, Johnson, Leary, Maclaurin, Necula, Paszke, VanderPlas, Wanderman-Milne, and Zhang, 2018). The jax library achieves optimal GPU usage with its just in time compilation. Jax uses pure functions. I use tools for automatic differentiation and vectorization. To build the models I use the flax library (Heek, Levskaya, Oliver, Ritter, Rondepierre, Steiner, and van Zee, 2023). I use the implementations for commonly used layers and modules, e.g. convolution layer, GRU, etc. I use reinforcement learning functions from the jax ecosystem to implement commonly used functions, e.g. one hot encoding of actions. I use tools from the jax ecosystem to track the data and save the models. Using tools instead of manually coding everything results in more concise and readable code. I chose these tools because they are integrated with the jax library.

4 Results

I trained the three different algorithms shown in figure 4.1 for different number of frames. This is because the algorithms took different times to update. The dreamer takes roughly 2 times as long to update as SimPLe. SimPLe takes roughly 3 times as long to update than the model-free PPO. The converged performance is significantly lower in the model based version. The model-based algorithms stop improving and start to stagnate at much lower episode return. In Breakout, SimPLe stops improving at around 5 destroyed bricks per episode. Dreamer stops improving at 3 destroyed bricks per episode. The model-free PPO is able

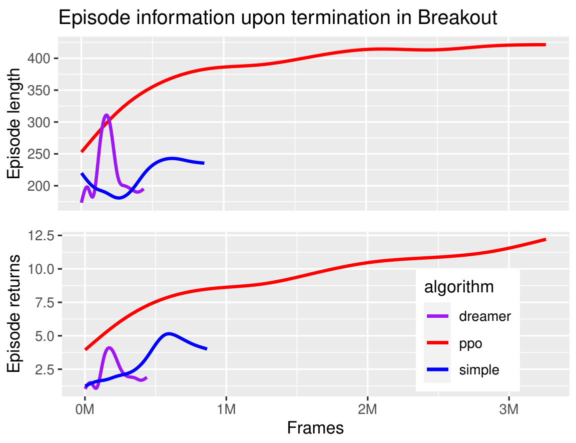

Figure 4.1: Episode length and episode returns for model-based algorithms in breakout compared to the model-free agent in Breakout. The algorithms do not reach the same number of frames in the 12 hour training time. The results are smoothed using Generalised Additive Model (GAM). The smoothing makes it seem that the models start at different performance, despite starting at the same performance.

to reach 13 destroyed bricks per episode. Both Dreamer and Simple collapse during the training, i.e. the performance starts decreasing. The episode length is highly correlated with the episode return in the Breakout environment (0.77 for model-free PPO, 0.88 for Dreamer, 0.8 for simple). This suggests that predicting episode termination is important.

In Mario, SimPLe and Dreamer end up with a suboptimal deterministic policy. This happens when the agent trains on a poorly trained transition model. This can be solved by training the transition model partially offline. When the transition model of SimPLe is trained partially offline, the policy remains stochastic. However, even with a transition model trained partially offline, the algorithm does not converge. The reward loss and the observation loss remain high. This means that the transition model is not able to properly learn the higher dimensional reward distribution and the changing observation. The agent does not outperform the model-free PPO in any of the model-based versions.
The episode length is highly correlated with the episode return in the Mario environment for model-free PPO and Dreamer (0.83 for model-free PPO, 0.71 for Dreamer). However, it is negatively correlated for partially online SimPLe (-0.096). This difference is explained by the fact that the agent can get stuck when moving right. Getting stuck increases episode length but does not provide positive rewards. I still believe the correlation is high enough to justify predicting episode

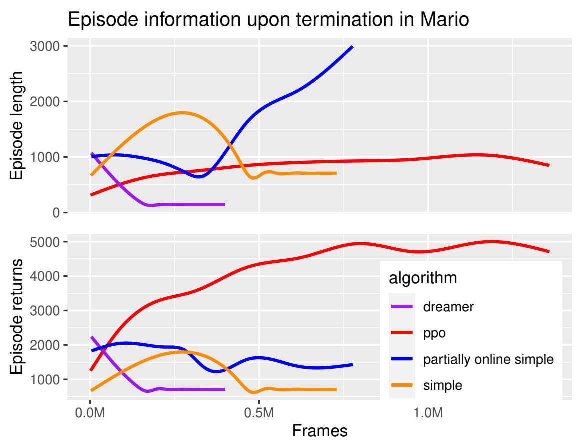

Figure 4.2: Episode length and episode returns for model-based algorithms compared to the model-free agent in Mario. The algorithms do not reach the same number of frames in the 12 hour training time

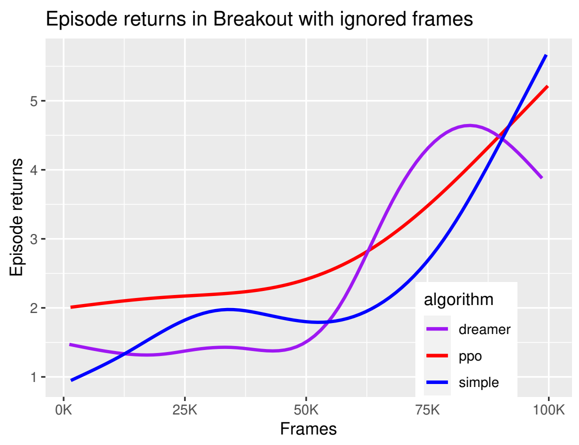

Figure 4.3: Results for partially online versions of Dreamer and Simple shifted to the moment when the learning went from offline to online.

termination.

Both Dreamer and SimPLe can be modified to partially online learning. This means that the model is first trained offline before we start training the agent. If we ignore the frames trained offline, the model-based methods have sample efficiency comparable or better than the sample efficiency of the model-free PPO for the first 100 000 frames of updating the agent. Dreamer outperforms model-free version of PPO after 50 000 frames. SimPLe outperforms model-free version of PPO after 100 000 frames. These results assume pretrained transition model for both SimPLe and Dreamer. These results show that the offline learned transition model can be used to accelerate learning. These results can not be used to support the hypothesis that model-based methods are more sample efficient than model-free

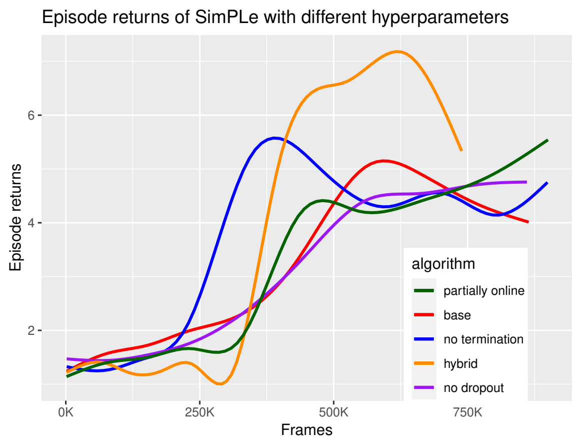

Figure 4.4: The episodic return of SimPLe with different design decisions. The base algorithm predicts episode termination and has dropout layers

algorithm	time [ms]	parallel agents
Mario *env*	20	8
Breakout *env*	1.4	8
SimPLe *env'*	0.72	16
Dreamer *env'*	0.028	100

Table 4.1: Time needed to interact with the transition models compared with the time needed to interact with the environment. The interaction is done with different number of parallel agents.

methods for online learning.

SimPLe is stable across wide range of hyperparameters Predicting episode termination seems to have no effect on the algorithm in the long term. In the short term predicting episode termination makes the algorithm converge more slowly. Removing the dropout layer does not seem to have a significant effect on the algorithm. Using hybrid learning speeds up the convergence. Moreover, hybrid learning allows the agent converge to higher episode return. Training the world model partially online results in slightly faster convergence speed and better performance.

The inference time for SimPLe is higher than

Algorithm	*env* interactions	*env'* interactions
PPO	100 000	0
SimPLe	100 000	100 000
Dreamer	100 000	625 000

Table 4.2: Number of interactions done with the transition model compared with the number of interactions with the environment. Different architectures allow for different number of agents to be trained in parallel.

inference time for Dreamer as can be seen in table 4.1. SimPLe is able to generate 16 frames in parallel in 0.72 milliseconds. Dreamer is able to generate 100 frames in parallel in 0.028 milliseconds. Both are able to generate the frames much faster than the original environment. SimPLe and Dreamer are able to predict more frames in parallel. The amount of frames that are predicted in parallel, i.e. the number of parallel agents, are limited by the GPU. These statistics confirms the assumption that sampling the transition model is computationally cheaper and faster. The model-based algorithms have different number of agents trained in parallel. This means that number of samples from the real environment mean different number of samples from the transition model as can be seen in table 4.2. This explains the differences in sample efficiency between dreamer and simple that is present in figure 4.3

5 Conclusions

The model-based methods converge to a lower episode return than the model-free agent. The performance is bounded by the quality of the transition model. The significantly worse performance and lower sample efficiency of model-based algorithms does not meet the expectations from original research (Hafner et al., 2020b) (Kaiser et al., 2020).

From my experience, the transition model is sensitive to hyperparameter changes. Small change in the architecture causes the model to converge to higher performance. This is because the transition model propagates any error in predicting the next state to future predictions. That makes predicting the state after an erroneous state also erroneous. Furthermore, any wrong prediction about the reward or termination impact the stability of PPO. This results in more unstable learning than the model-free PPO. The differences in my implementation of SimPLe and implementation of SimPLe by Kaiser et al. (2020) explain the difference in performance between my reproduction, and the performance reported by Kaiser et al. (2020).

The Dreamer often stops improving, and the episode returns achieved by the agent start decreasing. This should not be the case in the implementation by Hafner et al. (2020b). There are many potential reasons for this. Firstly, Hafner et al. (2020b) do not share the architecture of their model. Therefore, the architecture I use is different. Secondly, the stochastic architecture of dreamer might make the transition model more unreliable. Thirdly, I do not take the precautions against instability that Hafner et al. (2020a) take. Hafner et al. (2020a) use a random agent to gather initial samples for the transition model. When Hafner et al. (2020a) update the transition model, Hafner et al. (2020a) store the data used to update the model instead of discarding them. Then during the following update, they use all stored data instead of just the most recent data. I did not do this due to memory constraints. I use the differences between my implementation to explain the difference in performance between my reproduction and the performance reported by Hafner et al. (2020b).

According to my results sample efficiency of model-based algorithms is worse than the sample efficiency of model free algorithms. However this can be circumvented when the transition model is trained offline. In that case the agent can be trained on the cheaper samples from the transition model. Despite the convergence to low episode returns, model-based reinforcement learning with offline learned transition model can be used to boost initial learning. This might be especially useful when we need to train multiple model-free agents in an environment. In such cases the samples saved by training the model of the environment offset the frames needed to train the model of the environment. This results in better sampling efficiency even when when I include the samples from offline learning of the transition model.

Model-based reinforcement learning methods struggle to converge in a dynamic environment. Because big part of the frame changes with each action, the model needs more updates to learn the transition. I conclude this based on the image loss, which decreases much slower than when the entire environment is in the frame. Higher image loss also causes all the other losses to converge more slowly. Therefore the transition model will need many more updates and possibly different architecture to make model-based algorithms converge.

Hybrid learning allows the agent to perform better than if it was only trained with the model of the environment. Furthermore hybrid learning improves sampling efficiency of model-based algorithms, because the agent does more updates and converges faster with the same amount of samples. Predicting episode termination for Breakout and Mario seems logical, because episode length is highly correlated with episode returns. However, the results do not show significant benefit in doing so.

5.1 Future research

The agent was unable to perform in the Mario environment. This is due to the higher dimensional reward distribution and the changing observation. Both these problems need to be tackled separately. Model-based algorithms need to be observed on more environments. These environments need to be chosen specifically so only one of these problems is present in them. Therefore in these different environments, one of the two problems that were present in the Mario environment can be tackled at a time.

The same agent converged to lower performance when trained from the world model. Therefore main bottleneck for model based reinforcement learning is the quality of the transition model. MLP based model is unable to catch information from beyond the last 4 frames. Recurrent based model need sequential training and therefore converge more slowly. Using a transformer architecture for the transition model should help with the problems mentioned above (Chen, Wu, Yoon, and Ahn, 2022). Furthermore, some information about the environment can be integrated into the transition model. E.g. for breakout the transition model needs to focus mostly on the position of the ball and the agent controlled reflecting mirror. Therefore the transition model only needs to predict the location of these two stimuli.

The agent modified to be more robust to incorrect predictions of rewards. This limits the negative impact caused by errors from the transition model. Therefore the convergence performance of model-based methods would improve. This can be achieved by either using different model-free algorithms or by changing the architecture of the agents actor critic network. E.g. adding a dropout layer to the network architecture might make the agent more robust to occasional mistakes from the transition model (Baldi and Sadowski, 2013).

Learning of multiplayer video_games requires more frames to converge than learning of single player games (Silver, Huang, Maddison, Guez, Sifre, Van Den Driessche, Schrittwieser, Antonoglou, Panneershelvam, Lanctot, et al., 2016). This means that the agent needs to gather more samples from the environment to converge. This is because the agent needs to learn how to react to the actions of the other player, which creates more variety. However, the environment transitions inside the environment are not necessarily more complex. E.g. the environment transitions between a multiplayer pong game are comparable to the environment transition inside breakout. This means that it is not that more difficult to train a transition model for multiplayer games than single player games. Therefore multiplayer games should not need more samples to converge than single player games when we use model-based algorithms. I recommend exploring how model-based methods can be applied to multiplayer games.

References

Mohammad Babaeizadeh, Chelsea Finn, Dumitru Erhan, Roy H. Campbell, and Sergey Levine. Stochastic Variational Video Prediction, March 2018. URL `http://arxiv.org/abs/1710.11252`. arXiv:1710.11252 [cs].

Pierre Baldi and Peter J Sadowski. Understanding dropout. *Advances in neural information processing systems*, 26, 2013.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

Chang Chen, Yi-Fu Wu, Jaesik Yoon, and Sungjin Ahn. Transdreamer: Reinforcement learning with transformer world models. *arXiv preprint arXiv:2202.09481*, 2022.

Jiajun Fan. A review for deep reinforcement learning in atari: Benchmarks, challenges, and solutions. *arXiv preprint arXiv:2112.04145*, 2021.

Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. In *International conference on machine learning*, pages 2555–2565. PMLR, 2019.

Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to Control: Learning Behaviors by Latent Imagination, March 2020a. URL `http://arxiv.org/abs/1912.01603`. arXiv:1912.01603 [cs].

Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models. *arXiv preprint arXiv:2010.02193*, 2020b.

Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2023. URL `http://github.com/google/flax`.

Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H. Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Afroz Mohiuddin, Ryan Sepassi, George Tucker, and Henryk Michalewski. Model-Based Reinforcement Learning for Atari, February 2020. URL `http://arxiv.org/abs/1903.00374`. arXiv:1903.00374 [cs, stat].

Łukasz Kaiser and Samy Bengio. Discrete Autoencoders for Sequence Models, January 2018. URL `http://arxiv.org/abs/1801.09797`. arXiv:1801.09797 [cs, stat].

Christian Kauten. Super Mario Bros for OpenAI Gym. GitHub, 2018. URL `https://github.com/Kautenja/gym-super-mario-bros`.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Aske Plaat, Walter Kosters, and Mike Preuss. Deep Model-Based Reinforcement Learning for High-Dimensional Problems, a Survey, December 2020. URL `http://arxiv.org/abs/2008.05598`. arXiv:2008.05598 [cs].

Athanasios S Polydoros and Lazaros Nalpantidis. Survey of model-based reinforcement learning: Applications on robotics. *Journal of Intelligent & Robotic Systems*, 86(2):153–173, 2017.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, August 2017. URL `http://arxiv.org/abs/1707.06347`. arXiv:1707.06347 [cs].

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation, October 2018. URL `http://arxiv.org/abs/1506.02438`. arXiv:1506.02438 [cs].

Yuqing Shen. Proximal policy optimization with entropy regularization. In *2024 4th International Conference on Computer, Control and Robotics (ICCCR)*, pages 380–383. IEEE, 2024.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

Richard S. Sutton and Andrew Barto. *Reinforcement learning: an introduction.* Adaptive computation and machine learning. The MIT Press, Cambridge, Massachusetts, nachdruck edition, 2014. ISBN 978-0-262-19398-6.

Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. Gymnasium, March 2023. URL `https://zenodo.org/record/8127025`.