EVaDE : Event-Based Variational Thompson Sampling for Model-Based Reinforcement Learning

Siddharth Aravindan SIDDHARTH.ARAVINDAN@GMAIL.COM
Dixant Mittal DIXANT@COMP.NUS.EDU.SG
Wee Sun Lee LEEWS@COMP.NUS.EDU.SG
Department of Computer Science, National University of Singapore

Editors: Vu Nguyen and Hsuan-Tien Lin

Abstract

Posterior Sampling for Reinforcement Learning (PSRL) is a well-known algorithm that augments model-based reinforcement learning (MBRL) algorithms with Thompson sampling. PSRL maintains posterior distributions of the environment transition dynamics and the reward function, which are intractable for tasks with high-dimensional state and action spaces. Recent works show that dropout, used in conjunction with neural networks, induces variational distributions that can approximate these posteriors. In this paper, we propose Event-based Variational Distributions for Exploration (EVaDE), which are variational distributions that are useful for MBRL, especially when the underlying domain is object-based. We leverage the general domain knowledge of object-based domains to design three types of event-based convolutional layers to direct exploration. These layers rely on Gaussian dropouts and are inserted between the layers of the deep neural network model to help facilitate variational Thompson sampling. We empirically show the effectiveness of EVaDE-equipped Simulated Policy Learning (EVaDE-SimPLe) on the 100K Atari game suite.

Keywords: Exploration; Thompson Sampling; Model-Based Reinforcement Learning

1. Introduction

Model-Based Reinforcement Learning (MBRL) has recently gained popularity for tasks that allow for a very limited number of interactions with the environment [45]. These algorithms use a model of the environment, that is learnt in addition to the policy, to improve sample efficiency in several ways; these include generating artificial training examples [45, 35], assisting with planning [23, 9, 41, 10] and guiding policy search [22, 8]. Additionally, it is easier to incorporate inductive biases derived from the domain knowledge of the task for learning the model, as the biases can be directly built into the transition and reward functions.

In this paper, we demonstrate how domain knowledge can be utilised for designing exploration strategies in MBRL. While model-free agents explore the space of policies and value functions, MBRL agents explore the space of transition dynamics and reward functions.

One method for exploring the space of transition dynamics and reward functions is Posterior Sampling for Reinforcement Learning (PSRL) [34, 26], which uses the Thompson sampling [36] method of sampling the posterior of the model to explore other plausible models. Maintaining the posterior is generally intractable and in practice, variational distributions are often used as an approximation to the posterior [3, 40, 44].

(a) (b)

Figure 1: Rewards in Breakout, a popular Atari game. (a) shows an interaction between the ball and a brick which gives the agent a positive reward. (b) shows a state, where the paddle is unable to prevent the ball from going out of bounds. The lack of this interaction between the agent and the ball in this situation results in a penalty for the agent.

Traditionally, variational distributions are designed with two considerations in mind: inference and/or sampling should be efficient with the variational distribution, and the variational distribution should approximate the true posterior as accurately as possible. However, as the variational distribution may not fully represent the posterior, different approximations may be suitable for different purposes. In this paper, we propose to design the variational distribution to generate trajectories through parts of the state space that may potentially give high returns, for the purpose of exploration.

In MBRL, trajectories are generated in the state space by running policies that are optimised against the learned model. One way to generate useful exploratory trajectories is by perturbing the reward function in the model, so that a different part of the state space appears to contain high rewards, leading the policy to direct trajectories towards those states. Another method is to perturb the reward function, so that the parts of the state space traversed by the current policy appear sub-optimal, causing the policy to seek new trajectories.

We focus on problems where the underlying domain is object-based, meaning that the reward functions heavily depend on the locations of individual objects and their interactions, which we refer to as events. An example of such an object-based task is the popular Atari game Breakout (as shown in Figure 1). In this game, the agent receives rewards when the ball hits a brick and avoids losing a life by keeping the ball within bounds using the paddle, both of which are interactions between two objects. The rewards in this game are determined by the interactions between the ball and the bricks or the paddle.

For such domains, we introduce Event-based Variational Distributions for Exploration (EVaDE), a set of variational distributions that can help generate useful exploratory trajectories for deep convolutional neural network models. EVaDE comprises of three Gaussian dropout-based convolutional layers [33]: the noisy event interaction layer, the noisy event weighting layer, and the noisy event translation layer. The noisy event interaction layer is designed to provide perturbations to the reward function in states where multiple objects appear at the same location, randomly perturbing the value of interactions between objects. The noisy event weighting layer perturbs the output of a convolutional layer at a single location, assuming that the output of the convolutional filters captures events; this would upweight and downweight the reward associated with these events randomly. The noisy event translation layer perturbs trajectories that go through "narrow passages"; small translations can randomly affect the returns from such trajectories, causing the policy to explore different trajectories.

These EVaDE layers can be used as standard convolutional layers and inserted between the layers of the environment network models. When included in deep convolutional networks, the noisy event interaction layers, the noisy event weighting layers, and the noisy event translation layers generate perturbations on possible object interactions, the importance of different events, and the positional importance of objects/events, respectively, through the dropout mechanism. This mechanism induces variational distributions over the model parameters [33, 12].

An interesting aspect of designing for exploration is that the variational distributions can be useful, even if they do not approximate the posterior well, as long as they assist in perturbing the policy out of local optima. After perturbing the policy, incorrect parts of the model will either be corrected by data or left unchanged if they are irrelevant to optimal behaviour.

Finally, we approximate PSRL by incorporating EVaDE layers into the reward models of Simulated Policy Learning (SimPLe) [45]. We conduct experiments to compare EVaDE-equipped SimPLe (EVaDE-SimPLe) with various popular baselines on the 100K Atari test suite. In the conducted experiments, all agents operate in the low data regime, where the number of interactions with the real environment is limited to 100K. EVaDE-SimPLe agents achieve a mean human-normalised score (HNS) of 0.682 in these games, which is 79% higher than the mean score of 0.381 achieved by a recent low data regime method, CURL [21], and 30% higher than the mean score of 0.525 achieved by vanilla SimPLe agents.

2. Background and Related Work

Posterior sampling approaches like Thompson Sampling [36] have been one of the more popular methods used to balance the exploration exploitation trade-off. Exact implementations of these algorithms have been shown to achieve near optimal regret bounds [2, 18]. These approaches, however, work by maintaining a posterior distribution over all possible environment models and/or action-value functions. This is generally intractable in practice. Approaches that work by maintaining an approximated posterior distribution [29, 4], or approaches that use bootstrap re-sampling to procure samples, [28, 25] have achieved success in recent times.

Variational inference procures samples from distributions that can be represented efficiently while also being easy to sample. These variational distributions are updated with observed data to approximate the true posterior as accurately as possible. Computationally cost effective methods such as dropouts have been known to induce variational distributions over the model parameters [33, 12]. Consequently, variational inference approaches that approximate the posterior distributions required by Thompson sampling have gained popularity [3, 40, 37, 42].

Model-based reinforcement learning improves sample complexity at the computational cost of maintaining and performing posterior updates to the learnt environment models. Neural networks have been successful in modelling relatively complex and diverse tasks such as Atari games [24, 14]. Over the past few years, variational inference has been used to represent environment models, with the intention to capture environment stochasticity [15, 5, 13].

SimPLe [45] is one of the first algorithms to use MBRL to train agents to play video games from images. It is also perhaps the closest to EVaDE, as it not only employs an iterative algorithm to train its agent, but also uses an additional convolutional network assisted by an autoregressive LSTM based RNN to approximate the posterior of the hidden variables in the stochastic model. Thus, similar to existing methods [15, 5, 13], these variational distributions are used for the purpose of handling environment stochasticity rather than improving exploration. To the contrary, EVaDE-SimPLe is an approximation to PSRL, that uses a Gaussian dropout induced variational distribution over deterministic reward functions solely for the purpose of exploration. Unlike SimPLe, which uses the stochastic model to generate trajectories to train its agent, EVaDE-SimPLe agents optimize for a deterministic reward model sampled from the variational distribution and a learnt transition model. Moreover, with EVaDE, these variational distributions are carefully designed so as to explore different object interactions, importance of events and positional importance of objects/events, that we hypothesize are beneficial for learning good policies in object-based tasks.

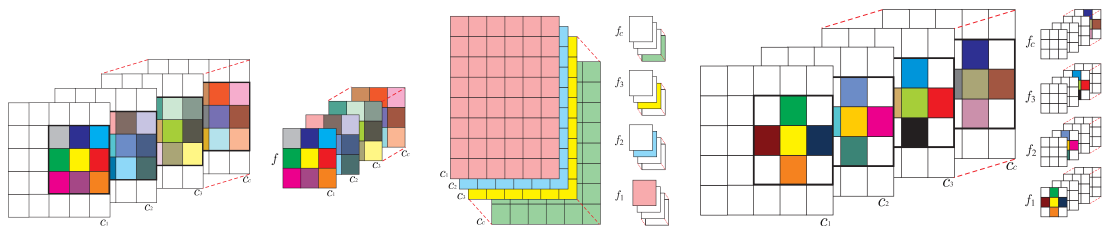
The current state of the art scores in the Atari 100K benchmark is achieved by EfficientZero [43], which was developed concurrently with our work. Its success is a consequence of combining several improvements proposed previously in addition to integrating tree search with learning to improve the policy executed by the agent. We believe that the benefits of using the variational designs induced by the EVaDE layers proposed in this paper are complementary to such search based methods, as these layers could be used in their reward models to guide the policy search by generating useful exploratory trajectories, especially in object-based domains.

3. Event Based Variational Distributions

Event-based Variational Distributions for Exploration (EVaDE) consist of a set of variational distribution designs, each induced by a noisy convolutional layer. These convolutional layers can be inserted after any intermediate hidden layer in deep convolutional neural networks to help us construct approximate posteriors over the model parameters to produce samples from relevant parts of the model space. EVaDE convolutional layers use Gaussian multiplicative dropout to draw samples from the variational approximation of the posterior. Posterior sampling is done by multiplying each parameter, θ^i_{env}, of these EVaDE layers by a perturbation drawn from a Gaussian distribution, $\mathcal{N}(1, (\sigma^i_{env})^2)$. These perturbations are sampled by leveraging the reparameterization trick [20, 31, 30, 11] using a noise sample from the standard Normal distribution, $\mathcal{N}(0, 1)$, as shown in Equation 1. The variance corresponding to each parameter, $(\sigma^i_{env})^2$, is trained jointly with the model parameters θ_{env}.

$$\tilde{\theta}^i_{env} \leftarrow \theta^i_{env}(1 + \sigma^i_{env}\epsilon^i); \;\; \epsilon^i \sim \mathcal{N}(0, 1) \tag{1}$$

When the number of agent-environment interactions is limited, the exploration strategy employed by the agent is critical. In object-based domains, rewards and penalties are often sparse and occur when objects interact. Hence, the agent needs to experience most of the events in order to learn a good environment model. Generating trajectories that contain



(a) (b) (c)

Figure 2: (a) This image shows one noisy event interaction filter acting on an input with c channels. Here f is an $m \times m$ noisy convolutional filter, which acts upon input patches at the same location across different channels, noisily altering the value of events captured at those locations. (b) This image shows how the filters of the noisy event weighting layer weight the input channels. The filters f_1, f_2, f_3 and f_c randomly upweight and downweight the events captured by the channels c_1, c_2, c_3 and c_c respectively. The white entries of the filter are entries that are set to zero, while the rest are trainable noisy model parameters. (c) The noisy event translation filter. The filters f_1, f_2, f_3 and f_c noisily translate events/objects captured by the channels c_1, c_2, c_3 and c_c respectively. The white entries of the filter are entries that are set to zero, while the rest are trainable noisy model parameters. Gaussian multiplicative dropout is applied to all the non-zero parameters of all EVaDE filters.

events is hence a reasonable exploration strategy. Additionally, a very common issue with training using a very few number of interactions is that the agent may often get stuck in a local optimum, prioritising an event, which is relatively important, but may not lead to an optimal solution. Generating potentially high return alternate trajectories that do not include that event is another reasonable exploration strategy.

With these exploration strategies in mind, we introduce three EVaDE layers, namely the noisy event interaction layer, the noisy event weighting layer and the noisy event translation layer. All the three layers are constructed with the hypothesis that the channels of the outputs of intermediate layers of deep convolutional neural networks either capture object positions, or events (interaction of multiple objects detected by multi-layer composition of the network).

3.1. Noisy Event Interaction Layer

The noisy event interaction layer is designed with the motivation of increasing the variety of events experienced by the agent. This layer consists of noisy convolutional filters, each having a dimension of $m \times m \times c$, where c is the number of input channels to the layer. Every filter parameter is multiplied by a Gaussian perturbation as shown in Equation 1. The filter dimension, m, is a hyperparameter that can be set so as to capture objects within a small $m \times m$ patch of an input channel. Assuming that the input channels capture the positions of different objects, a filter that combines the c input channels locally captures the local object interaction within the $m \times m$ patch. By perturbing the filter, different combinations of interactions can be captured; if the filter is used as part of the reward function, it will correspondingly reward different interactions depending on the perturbation. The policy optimized for different perturbed reward functions is likely to generate trajectories that contain different events. Note that convolutional filters are equivariant, so the same filter will detect the event anywhere in the image and can result in trajectories that include the event at different positions in the image.

We describe the filter in more detail. Every output pixel of the filter, $y^k_{i,j}$, representing $(i,j)^{th}$ pixel of the k^{th} output channel, can be computed as shown in Equation 2. Here x is the input to the layer with c input channels, $P_{x^l_{i,j}}$ is the $m \times m$ patch (represented as a matrix) centred around $x^l_{i,j}$, the $(i,j)^{th}$ pixel of the l^{th} input channel, $\tilde{\theta}^l_k$ is the l^{th} channel of the k^{th} noisy convolutional filter, $\odot$ the Hadamard product operator, and $\mathbb{1}_m$ is an m dimensional column vector whose every entry is 1.

$$y^k_{i,j} = \sum_{l=0}^{c} \mathbb{1}_m^T \left(\tilde{\theta}^l_k \odot P_{x^l_{i,j}} \right) \mathbb{1}_m \tag{2}$$

Figure 2a shows how this filter is applied over the input channels.

3.2. Noisy Event Weighting Layer

Overemphasis on certain events is possibly one of the main causes due to which agents converge to sub-optimal policies in object based tasks. Hence, it would be useful to easily be able to increase as well as decrease the importance of an event. For this layer, we assume that each input channel is already detecting an event and design a variational distribution over model parameters that directly up-weights or down-weights the events captured by different input channels.

This layer can be implemented with the help of c 1×1 noisy convolutional filters (each having a dimension of $1 \times 1 \times c$ as shown in Figure 2b), where c is the number of input channels. We denote the l^{th} element of the k^{th} filter in the layer as θ^l_k. To implement per channel noisy weighting, we set every θ^k_k as a trainable model parameter, which has a Gaussian dropout variance parameter associated with it to facilitate noisy weighting as shown in Equation 1. All other weights, i.e., θ^l_k when $l \neq k$ are set to 0. Thus each noisy event weighting layer has c trainable model parameters and c trainable Gaussian dropout parameters. A pictorial representation of how this layer acts on its input is presented in Figure 2b.

Every output $y^k_{i,j}$, corresponding to the $(i,j)^{th}$ pixel of the k^{th} output channel, can be computed using Equation 3, where $\tilde{\theta}^k_k$ is the noisy scaling factor for the k^{th} input channel.

$$y^k_{i,j} = \tilde{\theta}^k_k x^k_{i,j} \tag{3}$$

We expect that inducing such a variational distribution that up-weights or downweights events randomly helps the agents learn from different events that are randomly emphasised by different model samples drawn from the distribution. This may eventually help them in escaping local optima caused by overemphasis of certain events.

3.3. Noisy Event Translation Layer

In object based domains, an agent often has to perform a specific sequence of actions to successfully gain some rewards and may be penalized heavily for deviation from the sequence. We refer to the specific sequence of actions as a "narrow passage". A small translation of the positions of the environment or other objects will often cause the agent to be unsuccessful. When random translations of obstacles, events or boundaries are performed within the reward function, the optimized policy may select a different trajectory, possibly allowing it to escape from a locally optimal trajectory. We thus design the noisy event translation layer to induce a variational distribution over such model posteriors that can sample a variety of translations of relevant objects.

The noisy soft-translation on an input with c channels, is performed with the help of c convolutional filters, each having a dimension of $m \times m \times c$. These filters compute a noisy weighted sum of the corresponding input pixel and the pixels near it to effect a *noisy* translation of the channel. Similar to the noisy event weighting layer, each filter of the noisy event translation layer acts on one input channel. To achieve this, every parameter except the parameters of the k^{th} channel of the k^{th} filter, θ_k^k (which has a dimension of $m \times m$), and their corresponding dropout variances, is set to 0, for all k. Moreover in the channel θ_k^k, only the middle column and row contain trainable parameters. Figure 2c shows a detailed pictorial representation of this structure of the filters. A random translation of up to n pixels of the input can be achieved by using a $(2n+1) \times (2n+1)$ noisy event translation layer.

Equation 4 shows how $y_{i,j}^k$, the $(i,j)^{th}$ output pixel of the k^{th} channel, is computed. Here, $P_{x_{i,j}^k}$ is a $m \times m$ patch centred at $(i,j)^{th}$ pixel of the k^{th} input channel, $\tilde{\theta}_k^k$ is the k^{th} channel of the k^{th} noisy convolutional filter, $\odot$ the Hadamard product operator, and $\mathbb{1}_m$ is an m dimensional column vector where all the entries are 1.

$$y_{i,j}^k = \mathbb{1}_m^T \left(\tilde{\theta}_k^k \odot P_{x_{i,j}^k}\right) \mathbb{1}_m \tag{4}$$

3.4. Representational Capabilities of EVaDE networks

Ideally, adding EVaDE layers for exploration should not hinder the network to be unable to represent the true model, even if they don't accurately approximate the posterior. Theorem 1 below states that this is indeed the case.

Theorem 1 *Let* $\mathfrak{n}$ *be any neural network. For any convolutional layer* l*, let* $m_i(l) \times n_i(l) \times c_i(l)$ *and* $m_o(l) \times n_o(l) \times c_o(l)$ *denote the dimensions of its input and output respectively. Then, any function that can be represented by* $\mathfrak{n}$ *can also be represented by any network* $\tilde{\mathfrak{n}} \in \tilde{\mathscr{N}}$*, where* $\tilde{\mathscr{N}}$ *is the set of all neural networks that can be constructed by adding any combination of EVaDE layers to* $\mathfrak{n}$*, provided that, for every EVaDE layer* $\tilde{l}$ *added,* $\tilde{l}$ *uses a stride of* 1*,* $m_i(\tilde{l}) \leq m_o(\tilde{l}), n_i(\tilde{l}) \leq n_o(\tilde{l})$ *and* $c_i(\tilde{l}) \leq c_o(\tilde{l})$*.*

Proof The proof follows from the fact that every EVaDE layer $\tilde{l}_i$ that is added is capable of representing the identity function. A detailed proof is presented in the supplementary material.

If the added EVaDE layers induce distributions that poorly approximate the posterior, performance can indeed be poorer. But with enough data, the correct model should still be learnable since it is representable, as long as the optimization does not get trapped in a poor local optimum.

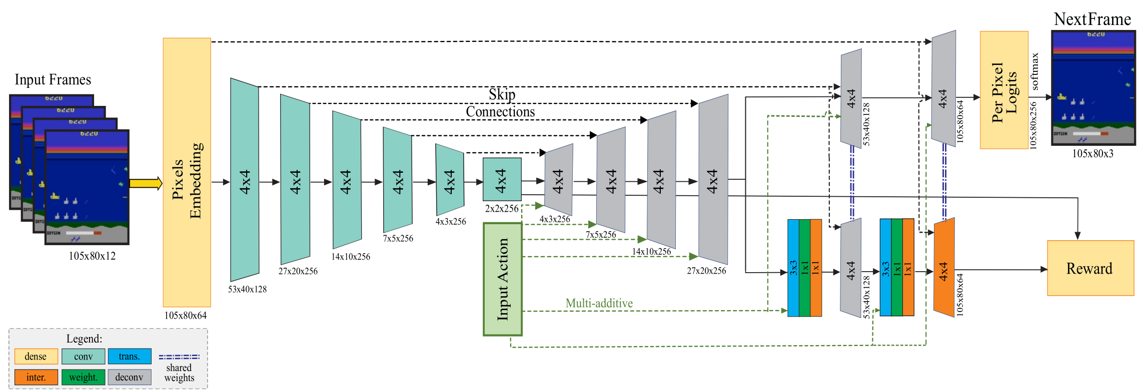


Figure 3: The network architecture of the environment model used to train EVaDE-SimPLe.

3.5. Approximate PSRL with EVaDE equipped Simulated Policy Learning

Simulated Policy Learning (SimPLe) [45] is an iterative model based reinforcement learning algorithm, wherein the environment model learnt is used to generate artificial episodes to train the agent policy. In every iteration, the SimPLe agent first interacts with the real environment using its current policy. After being trained on the set of all collected interactions, the models of the transition and reward functions are then used as a substitute to the real environment to train the policy of the agent to be followed by it in its next interactions with the real environment. PSRL [34, 26], which augments MBRL with Thompson sampling, has a very similar iterative structure as that of SimPLe. However, instead of maintaining a single environment model, PSRL maintains a posterior distribution over all possible environment models given the interactions experienced by the agent with the real environment. The agent then optimizes a policy for an environment model sampled from this posterior distribution. This policy is used in its real environment interactions of the next iteration. EVaDE equipped SimPLE approximates PSRL, by maintaining an approximate posterior distribution of the reward function with the help of the variational distributions induced by the three EVaDE layers.

Being an approximation of PSRL, an EVaDE-SimPLe agent has the same iterative training structure where it acts in the real environment using its latest policy to collect training interactions, learns a transition model and an approximate posterior over the reward model by jointly optimizing the environment model parameters, θ_{env}, and the Gaussian dropout parameters of the reward model, σ_{rew}, with the help of supervised learning. It then optimizes its policy with respect to an environment characterized by the learnt transition function and a reward model sample that is procured from the posterior with the help of Gaussian dropout as shown in Equation 1. This policy is then used by the agent to procure more training interactions in the next iteration.

4. Experiments

We conduct our experiments on the 100K Atari game test suite first introduced by [45]. This test suite consists of 26 Atari games where the number of agent-environment interactions is limited to 100K. Due to its diverse range of easy and hard exploration games [6], this test-suite has become a popular test-bed for evaluating reinforcement algorithms.

4.1. Network Architecture

In our experiments we use the network architecture of the deterministic world model introduced by [45] to train the environment models of the SimPLe agents, but do not augment it with the convolutional inference network and the autoregressive LSTM unit. Readers are referred to [45] for more details.

The architecture of the environment model used by EVaDE-SimPLe agents is shown in Figure 3. This model is very similar to the one used by SimPLe agents, except that it has a combination of a 3×3 noisy event translation layer, a noisy event weighting layer and a 1×1 noisy event interaction layer inserted before the fifth and sixth de-convolutional layers. The final de-convolutional layer acts as a noisy event interaction filter when computing the reward, while it acts as a normal de-convolutional layer while predicting the next observation. Sharing weights between layers allows us to achieve this. We insert EVaDE layers in a way that it perturbs only the reward function and not the transition dynamics.

We reuse the network architecture of [45] to train the policies in both the SimPLe and EVaDE-SimPLe agents using Proximal Policy Optimization (PPO) [32]. All the hyperparameters used for training the policy network and environment are the same as the ones used in [45].

4.2. Experimental Details

The training regimen that we use to train all the agents is the same and is structured similarly to the setup used by [45]. The agents, initialized with a random policy and collect 6400 real environment interactions before starting the first training iteration. In every subsequent iteration, every agent trains its environment model with its collection of real world interactions, refines its policy by interacting with the environment model, if it is a vanilla-SimPLe agent, or a transition model and a reward model sampled from the approximate posterior, if it is an EVaDE-SimPLe agent, and then collects more interactions with this refined policy.

PSRL regret bounds scale linearly with the length of an episode experienced by the agent in every iteration [27]. Shorter horizons, however, run the risk of the agent not experiencing anything relevant before episode termination. To balance these factors, we set the total number of iterations to 30, instead of 15. We allocate an equal number of environment interactions to each iteration, resulting in 3200 agent-environment interactions per iteration. The total number of interactions that each SimPLe and EVaDE-SimPLe agent procures (about 102K) is similar to SimPLe agents trained in [45], which allocates double the number of interactions per iteration, but trains for only 15 iterations. To disambiguate between the different SimPLe agents referred to in this paper, we refer to the SimPLe agents trained in our paper and [45] as SimPLe(30) and SimPLe respectively from here on.

We try to keep the training schedule of EVaDE-SimPLe and SimPLe(30) similar to the training schedule of the deterministic model in [45] so as to keep the comparisons fair. We train the environment model for 45K steps in the first iteration and 15K steps in all subsequent iterations. In every iteration of simulated policy training, 16 parallel PPO agents

Table 1: Comparison of the performances achieved by popular baselines and five independent training runs of EVaDE-SimPLe and SimPLe(30) agents with 100K agent-environment interactions in the 26 game Atari 100K test suite.

Game	SimPLe	SimPLe(30)	CURL	OTRainbow	Eff. Rainbow	EVaDE-SimPLe
Mean HNS	0.443	0.525	0.381	0.264	0.285	**0.682**
Median HNS	0.144	0.151	0.175	0.204	0.161	**0.267**
Vs EVaDE (W/L)	7W,19L	3W,23L	9W,17L	6W,20L	9W,17L	-
Best Performing	5	2	4	1	3	**11**

collect $z * 1000$ batches of 50 environment interactions each, where $z = 1$ in all iterations except iterations 8, 12, 23 and 27 where $z = 2$ and in iteration 30, where $z = 3$. The policy is also trained when the agent interacts with the real environment. However, the effect of these interactions (numbering 102K) on the policy is minuscule when compared to the 28.8M transitions experienced by the agent when interacting with the learnt environment model. Additional experimental details as well as the anonymized code for our agents are provided in the supplementary, which is available at https://tinyurl.com/3zb8nywx.

4.3. Results

We report the mean and median Human Normalized Scores (HNS) achieved by SimPLe(30), EVaDE-SimPLe and popular baselines SimPLe [45], CURL [21], OverTrained Rainbow [19] and Data-Efficient Rainbow [38] in Table 1. For each baseline, we report the number of games in which it is the best performing, among all compared methods, as well as the number of games in which it scores more (or less) than EVaDE-SimPLe, which are counted as its wins (or losses).

EVaDE-SimPLe agents achieve the highest score in 11 of the 26 games in the test suite, outperforming every other method on at least 17 games. Moreover, the effectiveness of the noisy layers to improve exploration can be empirically verified as EVaDE-SimPLe manages to attain higher mean scores than SimPLe(30) in 23 of the 26 games, even though both methods follow the same training routine. EVaDE-SimPLe also scores a mean HNS of 0.682, which is 79% higher than the score of 0.381 achieved by a popular baseline, CURL, and 30% more than the mean HNS of 0.525 achieved by SimPLe(30). Additionally, EVaDE-SimPLe agents also surpass the human performances [7] in 5 games, namely Asterix, Boxing, CrazyClimber, Freeway and Krull.

We also compute the Inter-Quartile Means (IQM), [1] a metric resilient to outlier games and runs, of SimPLe(30) and EVaDE-SimPLe agents. EVaDE-SimPLe agents achieve an IQM of 0.339, which is 68% higher than the IQM of 0.202 achieved by Simple(30) agents. This affirms that the improvements obtained due to the addition of the EVaDE layers are robust to outlier games and runs. In the supplementary material, we provide the scores achieved by all five independent runs of SimPLe(30) and EVaDE-SimPLe agents, which were used to compute the IQM.

Furthermore, a paired t-test on the mean HNS achieved by EVaDE-SimPLe and SimPLe(30) agents on each of the 26 games yields a single-tailed p-value of 3×10^{-3} confirming

1. IQM is well regarded in the reinforcement learning community, advocated by [1], which won the Outstanding Paper Award at NeurIPS 2021

Table 2: Scores (mean ± 1 standard error) achieved by SimPLe agents when equipped with all three EVaDE filters individually and when equipped with all filters simultaneously. All scores are averaged over five independent training runs.

Game	SimPle (30)	Inter. Layer	Weight. Layer	Trans. Layer	EVaDE-SimPLe
BankHeist	78.6±31.7	107.5±29.2	168.4±19.9	180.7 ± 16.7	**224.2 ± 35.4**
BattleZone	4544±803	6688 ± 1617	7525±2164	7750 ± 1355	**11094 ± 572**
Breakout	18.9±1.7	19.8 ± 3.6	22.4 ± 5.5	19.5 ±1.5	**24 ± 3.4**
CrazyClimber	43458±7709	59546±3164	**64191±5196**	59006±3282	60716±4082
DemonAttack	120.7±18.2	136.3 ±24.4	132 ± 14.7	133.7 ± 26	**141.8 ± 12.5**
Frostbite	260.3±2.5	254.6±5.5	254.4±3.6	263.2± 2.1	**274.2 ± 11**
JamesBond	**245.6±11.2**	202.2±65.2	182.5 ± 56.2	160.3 ± 68.4	235.6 ± 50.2
Kangaroo	576±330	**2201±993**	837.5 ± 345	1297± 321	1186.5±168
Krull	4532±883	3117±781	4818 ±440	5185 ± 991	**5335±455**
Qbert	2583±746	1953± 674	932 ±148	**3333 ± 575**	2764 ± 783
RoadRunner	2385±888	7178±1227	4853 ± 1322	6070±1834	**7799 ± 1296**
Seaquest	321.6±52	**644.4 ± 91.6**	608.5 ± 144.9	644.2 ±56.9	617.5 ± 118.1
HNS	0.52	0.56	0.65	0.69	**0.77**
IQM	0.22	0.29	0.26	0.29	**0.4**
Vs SimPLe(30) (W/L)	-	8W,4L	9W,3L	11W,1L	11W,1L

that the performance improvements over SimPLe(30) of EVaDE-SimPLe agents are statistically significant as an algorithm when applied to multiple games.

4.4. Ablation Studies

We also conduct ablation studies by equipping SimPLe(30) with just one of the three EVaDE layers to ascertain whether all of them aid in exploration. We do this by just removing the other two layers from the EVaDE environment network model (see Figure 3). Note that reward models that do not equip the noisy event interaction filter, also do not apply the Gaussian multiplicative dropout to the sixth de-convolutional layer.

We use a randomly selected suite of 12 Atari games in our ablation study. The games were chosen by arranging the 26 games of the suite in the alphabetical order, and then using the numpy [16] random function to sample 12 numbers from 0 to 25 without replacement. The corresponding games were then picked. Coincidentally, the chosen test suite contains easy exploration games such as Kangaroo, RoadRunner and Seaquest as well as BankHeist, Frostbite and Qbert, which are hard exploration games [6].

The mean scores, HNS and IQM achieved when SimPLe(30) is equipped with only one type of noisy convolutional layer and those of SimPLe(30) and EVaDE-SimPLe are presented in Table 2.

We present the learning curves of the trained EVaDE and SimPLe(30) agents in Figure 4. We omit the error bars here for clarity. Looking at the learning curves presented, it can possibly be said that an increase in scores of SimPLe(30) equipped with one of the EVaDE layers at a particular iteration would mean an increase in scores of EVaDE-SimPLe, albeit in later iterations. This pattern can clearly be seen in the games of BankHeist, Frostbite, Kangaroo, Krull and Qbert. This delay in learning could possibly be attributed to the agent draining its interaction budget by exploring areas suggested by one of the layers that are ineffective for that particular game. However, we hypothesise that since all the layers provide different types of exploration, their combination is more often helpful than wasteful.

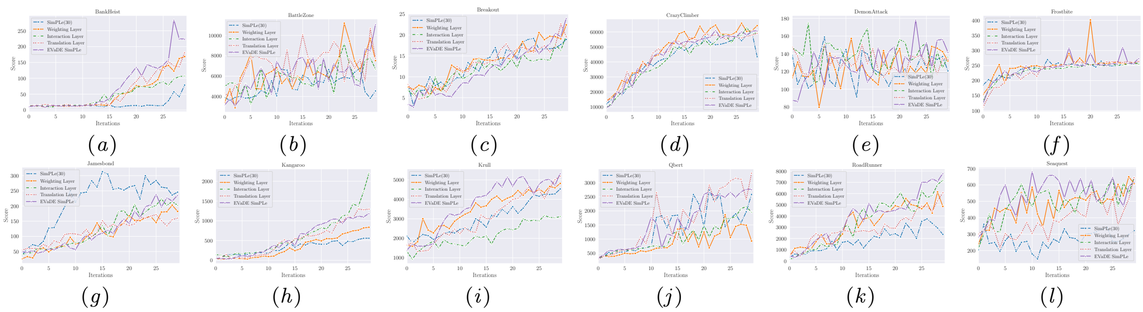


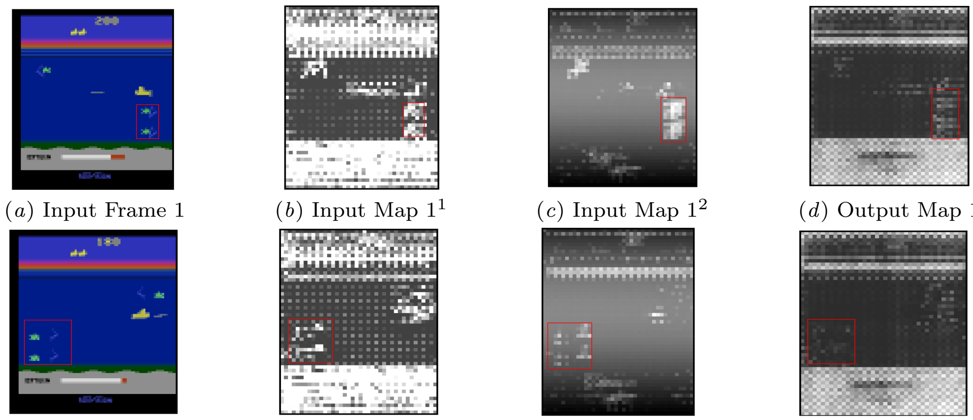
Figure 4: Learning curves of EVaDE-SimPLe agents, SimPLe(30) agents and agents which only add one of the EVaDE layers trained on the 12 game subset of the Atari 100K test suite.

This is validated by the fact that EVaDE-SimPLe achieves higher mean HNS and IQM than any other agent in this study.

We also look at the policies learnt by these agents in RoadRunner, where every EVaDE layer seems to improve the performance of the SimPLe agent even when added individually. We find that the vanilla-SimPLe RoadRunner agent learns a suboptimal policy where it frequently either collides with a moving obstacle or gets caught by the coyote. When the agent adds noisy interaction layers into its reward models, the policy learns to trick the coyote into colliding with the obstacle, an interaction beneficial to the roadrunner. On the other hand, when the reward models use only the noisy translation layers, the agent seems to learn a less *risky* policy, as it aggressively keeps away from both the obstacle as well as the coyote. The agent that adds only the noisy weighting layers seems to prioritize collecting points. This allows the coyote to get near the roadrunner, which could be undesirable. The policy learnt by the EVaDE-SimPLe agent combines the properties of the agents that add only the interaction and translation layers, as it tricks the coyote to colliding with the obstacle, while keeping a safe distance from both. The behaviour of the agent in this game provides some evidence that EVaDE layers can allow us to design different types of exploration based on our prior knowledge. We include videos of one episode run of each agent in the supplementary.

From Table 2, it can be seen that individually, each filter achieves a higher HNS than SimPLe(30), thus indicating that, on average, all the filters help in aiding exploration. Moreover, we see that with the exception of the noisy event interaction layer, the increase in HNS when the other two EVaDE layers are added individually is considerable. Additionally, all agents that add any of the noisy convolutional layers achieve higher IQM scores than baseline SimPLe agents. Furthermore, we hypothesise that all the layers provide different types of exploration since their combination is more often helpful than wasteful. This is validated by the fact that EVaDE-SimPLe achieves a higher mean HNS and IQM than any other agent in this study.

While having more parameters enlarges the class of functions representable by the model used by EVaDE agents, we emphasize that it is the design of the layers and not the higher number of parameters that is the reason for the performance gains. The agents that include only the noisy translation layers outperform SimPLe(30) on all metrics but add just 4K



(a) Input Frame 1 (b) Input Map 1^1 (c) Input Map 1^2 (d) Output Map 1
(e) Input Frame 2 (f) Input Map 2^1 (g) Input Map 2^2 (h) Output Map 2

Figure 5: This figure shows the output map that captures interactions between two input maps when passed through the noisy event interaction layer.

(a) Input Frame (b) Input Map (c) Output Map

Figure 6: This figure shows an output map (channel) that up-weights the corresponding input feature map when passed through the noisy event weighting layer.

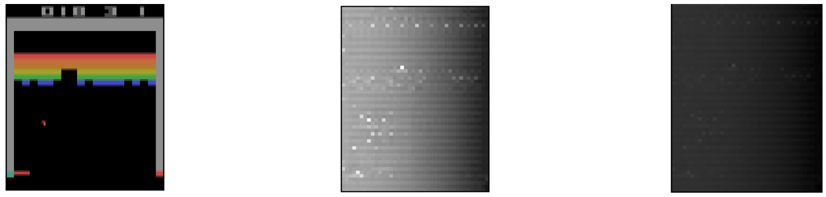
parameters to the reward model which contains about 10M parameters. Similarly, EVaDE-SimPLe adds only 0.43M parameters for its improvements.

4.5. Visualizations of the EVaDE layers

We also present some visualizations that help us understand the functionality of the EVaDE layers. All these visualizations were obtained after the completion of training, with the learned weights and variances of the final trained model.

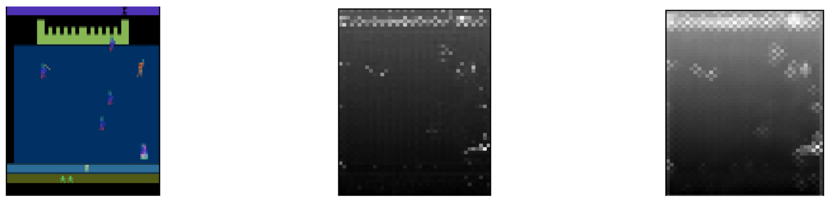
In Figure 5 we show illustrations of output map of a noisy event interaction layer detecting interactions between the right facing green-coloured enemy ships and the right facing blue-coloured divers given different input images from the game of Seaquest. We also show two input feature maps, which seem to capture the positions of these objects at the same locations. We observe that the pixels in the output feature map in Figure 5d are brighter at the locations where the two objects are close to each other, whereas in the same feature map these pixels are dimmer when the two objects are separated by some distance (Figure 5h).

In Figures 6 and 7, we show two feature maps before and after passing them through a noisy event weighting layer. The input images for these visualizations were taken from the game of Breakout. The input feature maps to the noisy event weighting layer seem



(a) Input Frame (b) Input Map (c) Output Map

Figure 7: This figure shows an output map (channel) that down-weights the corresponding input feature map when passed through the noisy event weighting layer.



(a) Input Frame (b) Input Map (c) Output Map

Figure 8: This figure shows the function of the noisy translation layer. The output map translates the input pixels to its top, bottom, left and right to different degrees

to capture the objects from the input image. The output feature map in Figure 6 is an upweighted version of its input, as the pixels seem to be brighter. On the other hand, the output feature map in Figure 7 seems to down-weight its input feature map, as the pixels seem a lot dimmer. The weighting factors for the input-output pairs shown in Figures 6 and 7 are 1.93 and 0.57 respectively. In Figure 8, we show the input and output feature maps of a game state from Krull, before and after passing it through a noisy event translation layer. The input feature map seems to capture different objects from the input image. The translation effect in output feature map can be seen clearly as every light pixel in the input seems to have lightened up the pixels to its top, bottom, left and right to different degrees.

5. Conclusion

In this paper, we introduce Event-based Variational Distributions for Exploration (EVaDE), a set of variational distributions over reward functions. EVaDE consists of three noisy convolutional layers: the noisy event interaction layer, the noisy event weighting layer, and the noisy event translation layer. These layers are designed to generate trajectories through parts of the state space that may potentially give high rewards, especially in object-based domains. We can insert these layers between the layers of the reward network models, inducing variational distributions over the model parameters. The dropout mechanism then generates perturbations on object interactions, importance of events, and positional importance of objects/events. We draw samples from these variational distributions and generate simulations to train the policy of a Simulated Policy Learning (SimPLe) agent.

We conduct experiments on the Atari 100K test suite, a test suite comprising 26 games where the agents are only allowed 100K interactions with the real environment. EVaDE-SimPLe agents outperform vanilla-SimPLe(30) on this test suite by achieving a mean HNS of 0.682, which is 30% more than the mean HNS of 0.525 achieved by vanilla-SimPLe(30) agents. EVaDE-SimPLe agents also surpass human performances in five games of this suite. Additionally, these agents also outperform SimPLe(30) on median HNS as well as IQM scores, which is a metric that is resilient to outlier runs and games. Furthermore, a paired-t test also confirms the statistical significance of the improvements in HNS on the test suite ($p = 3 \times 10^{-3}$). We also find, through an ablation study, that each noisy convolutional layer, when added individually to SimPLe results in a higher mean HNS and IQM. Additionally, the three noisy layers complement each other well, as EVaDE-SimPLe agents, which include all three EVaDE layers, achieve higher mean HNS and IQM than agents which add only one noisy convolutional layer. Finally, we also present visualizations that help us understand the functionality of the EVaDE layers.

Appendix A. EVaDE-SimPLe as an approximation of PSRL

Algorithm 1 Approximate PSRL with EVaDE equipped Simulated Policy Learning

Initialize agent policy, environment model and reward model dropout parameters $\theta_\pi, \theta_{env}, \sigma_{rew}$ respectively;
Initialize empty real environment interaction dataset $D_{real} \leftarrow \{\}$;
for *iteration in* $1 \cdots T$ **do**
  $s \leftarrow \emptyset$
  `// Interact with real environment`
  while k_{real} *real-world interactions not collected* **do**
    if s *is terminal or* $\emptyset$ **then**
      Start new episode, initialize start state s
    end
    Sample action $a \sim \text{Policy}(s, \theta_\pi)$
    $(s', r) \leftarrow \text{Interact\_Real\_World}(s, a)$
    $D_{real} \leftarrow D_{real} \cup \{(s, a, r, s')\}$, $s \leftarrow s'$
  end
  `// Learn a variational posterior`
  $\theta_{env}, \sigma_{rew} \leftarrow \text{Supervised\_Learn}(\theta_{env}, \sigma_{rew}, D_{real})$
  `// Draw a sample from the posterior`
  for *layer* i *in environment model* **do**
    if i *is an EVaDE layer* **then**
      Sample $\epsilon^i \sim N(0, 1)$
      $\tilde{\theta}^i_{env} \leftarrow \theta^i_{env}(1 + \sigma^i_{rew}\epsilon^i)$
    end
    else
      $\tilde{\theta}^i_{env} \leftarrow \theta^i_{env}$
    end
  end
  `// Train policy with environment sample`
  $s \leftarrow \emptyset$, $D_{sim} \leftarrow \emptyset$, steps $\leftarrow 0$
  while k_{sim} *interactions not completed* **do**
    if s *is terminal or* $\emptyset$ **then**
      Start new episode, initialize start state s
    end
    Sample action $a \sim \text{Policy}(s, \theta_\pi)$
    $(s', r) \leftarrow \text{Interact\_Env\_Sample}(\tilde{\theta}_{env}, s, a)$
    $D_{sim} \leftarrow D_{sim} \cup \{(s, a, r, s')\}$, $s \leftarrow s'$, steps $\leftarrow$ steps + 1
    if *steps* mod *update_frequency = 0* **then**
      Update policy: $\theta_\pi \leftarrow \text{Reinforcement\_Learn}(\theta_\pi, D_{sim})$
    end
  end
end
return θ_π

We present the pseudocode of EVaDE-SimPLe in Algorithm 1. As mentioned in Section 3, an EVaDE-SimPLe agent has the same iterative training structure as SimPLe and PSRL. In the first step of each iteration, the agent interacts with the real environment using its latest policy to collect interactions. The agent then updates its posterior distribution over the environment model parameters by jointly optimizing the environment model parameters θ_{env} which include the parameters of the transition and reward function and the Gaussian dropout parameters of the reward network σ_{rew} with the help of supervised learning. A sample from this approximate posterior distribution is then acquired with the help of Gaussian dropout. Subsequently, the agent updates its policy by optimizing the parameters of the policy network, θ_π by interacting with this environment sample. This optimized policy is used by the agent to procure more training interactions by interacting with the

Appendix B. Proof of Theorem 1

We provide the proof for Theorem 1, which is restated below, in this section.

Theorem 1 : Let $\mathbb{n}$ be any neural network. For any convolutional layer l, let $m_i(l) \times n_i(l) \times c_i(l)$ and $m_o(l) \times n_o(l) \times c_o(l)$ denote the dimensions of its input and output respectively. Then, any function that can be represented by $\mathbb{n}$ can also be represented by any network $\tilde{\mathbb{n}} \in \tilde{\mathscr{N}}$, where $\tilde{\mathscr{N}}$ is the set of all neural networks that can be constructed by adding any combination of EVaDE layers to $\mathbb{n}$, provided that, for every EVaDE layer $\tilde{l}$ added, $\tilde{l}$ uses a stride of 1, $m_i(\tilde{l}) \leq m_o(\tilde{l}), n_i(\tilde{l}) \leq n_o(\tilde{l})$ and $c_i(\tilde{l}) \leq c_o(\tilde{l})$.

B.1. Notations

Neural Networks

Any function f represented by a k-layer neural network $\mathbb{n}$ is an ordered composition of the functions $f_1, f_2, \cdots, f_k$ computed by its layers $N_1, N_2, \cdots N_k$ respectively, i.e., $f = f_k \circ f_{k-1} \circ \cdots f_1$.

Convolutional Layers

Any $m \times n$ convolutional layer l has a total of $m \times n \times c_i(l) \times c_o(l)$ learnable parameters, where $c_i(l)$ and $c_o(l)$ are the number of channels in the input and output of the layer respectively. The parameters of any convolutional layer l can be partitioned into $c_o(l)$ filters, where each filter has $m \times n \times c_i(l)$ parameters, and is responsible for computing one output channel.

We denote the set of parameters of any convolutional layer by θ. We denote the set of parameters of the k^{th} filter by θ_k, and the parameters of the l^{th} channel of this filter by θ_k^l. We denote the $(i,j)^{th}$ parameter of the l^{th} channel of the k^{th} filter by $\theta_k^{l,i,j}$. For noisy convolutional layers, we have a learnable Gaussian dropout parameter attached to every parameter of the convolutional layer (see Equation 1). We use σ_k, σ_k^l and $\sigma_k^{l,i,j}$ to denote the dropout parameters of the k^{th} filter, the l^{th} channel of the k^{th} filter and the $(i,j)^{th}$ parameter of the l^{th} channel of the k^{th} filter respectively.

Strides

A stride is a hyperparameter of a convolutional layer, that determines the number of pixels of the input that each convolutional filter moves, to compute the next output pixel.

B.2. Implications of the constraints in Theorem 1

Theorem 1 states that every EVaDE layer $\tilde{l}$ added uses a stride of 1 and satisfies the constraints $m_i(\tilde{l}) \leq m_o(\tilde{l}), n_i(\tilde{l}) \leq n_o(\tilde{l})$ and $c_i(\tilde{l}) \leq c_o(\tilde{l})$. This means that for any inserted EVaDE layer, every output dimension is at least as large as its corresponding input dimension. This eventually implies that for every EVaDE layer, all input and output dimensions match, i.e., $m_i(\tilde{l}) = m_o(\tilde{l}), n_i(\tilde{l}) = n_o(\tilde{l})$ and $c_i(\tilde{l}) = c_o(\tilde{l})$.

To see why, let us assume that the EVaDE layers $\tilde{l_j}, \cdots \tilde{l_k}$ are inserted, in order, in between the layers N_i and N_{i+1} of a neural network $\mathbb{n}$. As N_i and N_{i+1} are two consecutive layers of $\mathbb{n}$, we must have $m_i(N_{i+1}) = m_o(N_i), n_i(N_{i+1}) = n_o(N_i)$ and $c_i(N_{i+1}) = c_o(N_i)$. This implies that the dimensions of the input to layer $\tilde{l_j}$ match the dimensions of the output of the layer $\tilde{l_k}$, i.e., $m_i(\tilde{l_j}) = m_o(\tilde{l_k}), n_i(\tilde{l_j}) = n_o(\tilde{l_k})$ and $c_i(\tilde{l_j}) = c_o(\tilde{l_k})$. However, under the constraints imposed in Theorem 1, every output dimension is greater than or equal to its corresponding input dimension for every EVaDE layer. Thus, matching the output dimensions of $\tilde{l_k}$ with the input dimensions of $\tilde{l_j}$ is only possible if the input and output dimensions match for every EVaDE layer $\tilde{l_j}, \cdots \tilde{l_k}$ that is added.

With the above implications, the constraint of using a stride of 1, forces SAME padding for every EVaDE layer, and also ensures that patches centred around every $(i, j)^{th}$ pixel of every channel in the input are used to compute the outputs. This is an important implication that will help us prove the claims that all EVaDE layers can represent the identity transformation.

B.3. Claims

We prove the three following claims by construction, i.e., showing that there is a combination of parameters using which these layers can perform the identity transformation.

Claim 1 *The noisy event interaction layer can represent the identity transformation.*

Proof Let us assume an $m \times m$ noisy event interaction layer. With the help of the observations made in the previous section, we are ensured of using patches centred around every input $x^l_{i,j}$ $\forall i, j, l$ and the constraints also ensure that the number of filters in this layer is equal to the number of input channels.

The identity transformation can be achieved with the following parameter assignments.

- The dropout parameter $\sigma_k^{l,i,j}$ corresponding to every convolutional layer parameter $\theta_k^{l,i,j}$ is set to zero.
- The layer parameter corresponding to the central entry of the k^{th} layer of the k^{th} convolutional filter, i.e., $\theta_k^{k,\lceil\frac{m}{2}\rceil,\lceil\frac{m}{2}\rceil}$ is set to 1 $\forall k$.
- All other convolutional layer parameters are set to 0.

As stated in Equation 2, the event interaction layer computes the outputs $y^k_{i,j}$ $\forall i, j, k$ using the following equation.

$$y^k_{i,j} = \sum_{l=0}^{c} \mathbb{1}_m^T \left(\tilde{\theta}^l_k \odot P_{x^l_{i,j}} \right) \mathbb{1}_m$$

Applying the above parameter assignments, we get $y_{i,j}^k = x_{i,j}^k$, as the only non-zero parameter in the k^{th} filter, that is set to 1, aligns with $x_{i,j}^k$. This is the required identity transformation.

■

Claim 2 *The noisy event weighting layer can represent the identity transformation.*

Proof The noisy event weighting layer uses c 1×1 convolutional filters, where c is the number of input channels. Consequently, θ_k^k, is just a single trainable parameter instead of a grid of trainable parameters as in the other two EVaDE layers.

The identity transformation can be achieved with the following parameter assignments.

- The dropout parameter σ_k^l corresponding to every convolutional layer parameter θ_k^l is set to zero.
- The layer parameter corresponding to the k^{th} layer of the k^{th} convolutional filter, i.e., θ_k^k is set to 1 $\forall k$.
- All other convolutional layer parameters are set to 0.

This is a valid assignment, as the only parameters set to 1 are trainable, while the other parameters are forced to be set to 0 by construction (see Section 3.2).

As stated in Equation 3, the event interaction layer computes every output $y_{i,j}^k$ using the following equation.

$$y_{i,j}^k = \tilde{\theta}_k^k x_{i,j}^k$$

Setting $\theta_k^k = 1$ and $\sigma_k^k = 0$ $\forall k$, yields $y_{i,j}^k = x_{i,j}^k$ $\forall i, j, k$, which is the identity transformation required. ■

Claim 3 *The noisy event translation layer can represent the identity transformation.*

Proof

In this case, we can use the parameter assignments as stated in the proof of Claim 1 to produce an identity transformation. This is possible, since we construct the noisy event translation layer with the same structure of an $m \times m$ convolutional layer with the number of filters equalling the number of input channels. Moreover, the only non-zero parameter (which is set to 1) in the k^{th} filter, $\theta_k^{k,\lceil \frac{m}{2} \rceil, \lceil \frac{m}{2} \rceil}$ is in the middle row and middle column of its k^{th} channel, making it a valid assignment for the noisy event translation layer (see Section 3.3).

As stated in Equation 4, the event interaction layer computes every output $y_{i,j}^k$ using the following equation.

$$y_{i,j}^k = \mathbb{1}_m^T \left(\tilde{\theta}_k^k \odot P_{x_{i,j}^k} \right) \mathbb{1}_m$$

As in the case with the noisy event interaction layer, substituting these assignments, we get $y_{i,j}^k = x_{i,j}^k$ $\forall i, j, k$, which is the identity transformation required.

■

B.3.1. Proof of Theorem 1

We have to prove that all elements from $\tilde{\mathscr{N}}$, the set of neural networks that can be constructed by adding any combination of EVaDE layers to the neural network $\mathtt{n}$, can represent the functions represented by k-layered neural network $\mathtt{n}$.

Let $\tilde{n}$ be a general element from $\tilde{\mathscr{N}}$, that adds the EVaDE layers $\tilde{l_1}, \tilde{l_2}, \cdots \tilde{l_m}$, in order, after the layers $N_{i_1}, N_{i_2}, \cdots N_{i_m}$ of the neural network $\mathtt{n}$, where $i_{j-1} \leq i_j \leq i_{j+1}\ ; \forall 2 \leq j \leq m-1$ and $i_1 \geq 0, i_m \leq k$. Adding an EVaDE layer after N_0 refers to it being added after the input layer and before the first layer of $\mathtt{n}$. Note that more than one EVaDE layer can be added after any layer N_j of $\mathtt{n}$.

Also, let $f_1, f_2, \cdots f_k$ be the functions computed by the layers $N_1, N_2, \cdots, N_k$ of $\mathtt{n}$ respectively. Thus the function represented by $\mathtt{n}$ is $f = f_k \circ f_{k-1} \circ \cdots f_1$.

Let $\tilde{f_1}, \tilde{f_2}, \cdots \tilde{f_m}$ be the functions computed by the EVaDE layers $\tilde{l_1}, \tilde{l_2}, \cdots \tilde{l_m}$ respectively. Thus the function computed by $\tilde{n}$ is $\tilde{f} = f_k \circ f_{k-1} \circ \cdots \circ \tilde{f_m} \circ f_{i_m} \cdots \circ \tilde{f_1} \circ f_{i_1} \circ \cdots f_1$. As all $\tilde{f_1}, \tilde{f_2}, \cdots \tilde{f_m}$ can learn to represent the identity transformation, $\tilde{f}$ can learn to represent f. This implies that $\tilde{n}$ can represent any function represented by $\mathtt{n}$.

Appendix C. Variational Distributions using Dropouts

Variational methods are used to approximate inference and/or sampling when using intractable posterior distributions. These methods work by using variational distributions that facilitate easy sampling and/or inference, while approximating the true posterior as closely as possible.

These methods require the user to define two distributions, the prior $p(\theta)$, and the variational distribution $q(\theta)$. Given a set of training samples $D = (X, Y)$, where X is the set of input samples and Y the set of corresponding labels, variational methods work to minimize the KL-divergence between the learnt variational distribution $q(\theta)$ and the true posterior $p(\theta|D)$. This is equivalent to maximizing the Evidence Lower Bound (ELBO) as shown below.

$$KL(q(\theta), p(\theta|D)) = \int q(\theta) \log \frac{q(\theta)}{p(\theta|D)} d\theta$$

Now,

$$\begin{aligned} p(\theta|D) =& p(\theta|X,Y) = \frac{p(\theta)p(X,Y|\theta)}{p(X,Y)} = \frac{p(\theta)p(Y|X,\theta)p(X|\theta)}{p(X,Y)} \\ =& \frac{p(\theta)p(Y|X,\theta)p(X)}{p(X,Y)}; \end{aligned}$$

Substituting the value for $p(\theta|D)$,

$$KL(q(\theta), p(\theta|D)) = \int q(\theta) \left[\log \frac{q(\theta)p(X,Y)}{p(\theta)p(Y|X,\theta)p(X)}\right] d\theta$$
$$= \int q(\theta) \log \frac{q(\theta)}{p(\theta)p(Y|X,\theta)} d\theta + \int q(\theta) \log \frac{P(X,Y)}{P(X)} d\theta$$
$$= \int q(\theta) \log \frac{q(\theta)}{p(\theta)p(Y|X,\theta)} d\theta + \log \frac{P(X,Y)}{P(X)}$$
$$= \int q(\theta) \log \frac{q(\theta)}{p(\theta)} d\theta - \int q(\theta) \log p(Y|X,\theta) d\theta + \log \frac{P(X,Y)}{P(X)}$$
$$= KL(q(\theta), p(\theta)) - \int q(\theta) \log p(Y|X,\theta) d\theta + \log \frac{P(X,Y)}{P(X)}$$

Since $P(X,Y)$ and $P(X)$ are constants with respect to θ, the set of parameters that minimize $KL(q(\theta), p(\theta|D))$ are the same as the ones that maximize the ELBO, i.e.,

$$\arg\min_{\theta} KL(q(\theta), p(\theta|D)) = \arg\max_{\theta} \int q(\theta) \log p(Y|X,\theta) d\theta - KL(q(\theta), p(\theta))$$

C.1. Dropouts as Variational Distributions

[12] introduces the usage of dropout as a mechanism to induce variational distributions, samples from which are used to approximate the ELBO. The first term of the ELBO can be re-written as,

$$\int q(\theta) \sum_{i=1}^{N} \log p(y_i|x_i,\theta) d\theta$$

where every (x_i, y_i) is a training example from D.

This integral can be approximated by averaging out the log-probabilities using several samples drawn from the variational distribution $q(\theta)$ (Equation 5).

$$\int q(\theta) \sum_{i=1}^{N} \log p(y_i|x_i,\theta) d\theta \approx \sum_{i=1}^{N} \log p(y_i|x_i,\theta_i);\ \text{where } \theta_i \sim q(\theta) \tag{5}$$

Neural networks that use different types of dropouts help us maintain variational distributions $q(\theta)$ that approximate posteriors over deep Gaussian processes [12, 20, 3]. Procuring a sample from this posterior using $q(\theta)$ is easy, as a random dropped out network corresponds to a sample from the posterior over these deep Gaussian processes.

Appendix D. Network Architectures

In this section, we detail the network architectures used for training the environment models of SimPLe [45] and EVaDE-SimPLe, and the policy network architectures used by both the methods.

D.1. Environment Network Architecture

D.1.1. SimPLe

In our experiments we use the network architecture of the deterministic world model introduced in [45] to train the environment models of the SimPLe agents, but do not augment it with the convolutional inference network and the autoregressive LSTM unit.

Given four consecutive game frames and an action as input, the network jointly models the transition and reward functions, as it predicts the next game frame and the reward using the same network. The network consists of a dense layer, which outputs a pixel embedding of the stacked input frames. This layer is followed by a stack of six 4×4 convolutional layers, each with a stride of 2. These layers are followed by six 4×4 de-convolutional layers. For $1 \leq i \leq 5$, the i^{th} de-convolutional layers, take in as input, the output of the previous layer, as well as the output of the $6 - i^{th}$ convolutional layer. The last de-convolutional layer takes in as input the output of its previous layer and the dense pixel embedding layer. An embedding of the action input is multiplied and added to the input channels of every de-convolutional layer. The outputs from the last de-convolutional layer is passed through a softmax function to predict the next frame. The outputs from the last de-convolutional layer is also combined with the output of the last convolutional layer and then passed through a fully connected layer with 128 units followed by the output layer to predict the reward.

D.1.2. EVaDE-SimPLe

The architecture of the environment model used by EVaDE agents is shown in Figure 3. This model resembles the model of SimPLe agents until the fourth de-convolutional layer. All the stand-alone EVaDE layers that we use, use a stride of 1 and SAME padding so as to keep the size of the inputs and outputs of the layer same. As the EVaDE layers are added only to the reward function, we split the network into two parts, one that predicts the next frame (the transition network) and one that predicts the reward (the reward network) respectively. We denote the last two de-convolutional layers in each part d_5^t, d_6^t and d_5^r, d_6^r respectively.

As shown in Figure 3, in the transition network, the outputs of the fourth de-convolutional layer and the first convolutional layer are passed to d_5^t. d_6^t takes in as inputs the outputs of d_5^t and the pixel embedding layer.

The reward network adds a combination of a 3×3 noisy event translation layer, a noisy event weighting layer and a 1×1 noisy event interaction layer which are inserted before both d_5^r and d_6^r. d_5^r shares weights with d_5^t, and takes in the outputs of the previous event interaction layer and the first convolutional layer as inputs. Likewise, d_6^r shares weights with d_6^t, and takes in the outputs of the previous event interaction layer and the pixel embedding layer as inputs. Moreover, we also apply Gaussian multiplicative dropout to the weights of d_6^r, to make it act as an event interaction layer. As with SimPLe agents, an embedding of the action input is multiplied and added to the input channels of every de-convolutional layer (also shown in Figure 3).

The outputs of d_6^t are passed through a softmax to predict the next frame, while the outputs of d_6^r are combined with the output of the last convolutional layer and passed through a fully connected layer with 128 units followed by the output layer to predict the reward.

D.2. Policy Network

The policy network for both SimPLe and EVaDE-SimPLe agents consists of two convolutional layers followed by a hidden layer and an output layer. The inputs to the policy network are four consecutive game frames, which are stacked and passed through two 5×5 convolutional layers, both of which use a stride of 2. These convolutional layers are followed by a fully connected layer with 128 hidden units, which is followed by the output layer, that predicts the stochastic policy, i.e., the probabilities corresponding to each valid action, and the value of the current state of the agent.

Appendix E. Experimental Details

E.1. Codebase used and Hyperparameters

The code for Simple(30) and EVaDE-SimPLe agents is provided in the supplementary.

We build our SimPLe(30) and EVaDE-SimPLe agents by utilizing the implementation of SimPLe agents from [39]. To keep the comparison fair, we use the same hyperparameters as used by [39] to train all our agents. The codebase in [39] uses an Apache 2.0 license, thus allowing for public use and extension of their codebase.

E.2. Computational Hardware Used

We train our agents on a cluster of 4 NVIDIA RTX 2080 Ti GPUs with an Intel Xeon Gold 6240 CPU. The total time taken to train 5 independent runs of all 5 algorithms on the test suite of 12 games in addition to 5 independent runs of SimPLe(30) and EVaDE-SimPLe on the rest of the 14 games in the suite was around 195 days (or about 6.5 months).

E.3. Human Normalized Score

We use the human normalized scores from [7] as defined in Equation 6 to compare our agents.

$$\mathrm{HNS}_{\mathrm{agent}} = \frac{\mathrm{Score}_{\mathrm{agent}} - \mathrm{Score}_{\mathrm{random}}}{\mathrm{Score}_{\mathrm{human}} - \mathrm{Score}_{\mathrm{random}}} \tag{6}$$

where $\mathrm{Score}_{\mathrm{agent}}, \mathrm{Score}_{\mathrm{human}}$ and $\mathrm{Score}_{\mathrm{random}}$ denote the scores achieved by agent being evaluated, a human and an agent which acts with a random policy respectively.

We also list the baseline scores achieved by humans and random agents, as listed in [7] in Table 3 for easy access.

E.4. Inter-Quartile Mean

Benchmarking the results of reinforcement learning algorithms is inherently noisy, as the results of most training runs of these algorithms depend on a variety of factors including random seeds, choice of the evaluation environment and the codebase used by these runs [17]. While the human normalized scores will average out the variability in the performances of these training runs with a large number of training runs, often these scores are skewed by outlier games or scores, i.e., games or random trials in which the algorithm achieves unusually high or low scores.

Table 3: Baseline human and random values used to calculate Human Normalized Scores

Game	Human Score	Random Score
Alien	7,127.7	227.8
Amidar	1719.5	5.8
Assault	742	222.4
Asterix	8503.3	210
BankHeist	753.1	14.2
BattleZone	37187.5	2360
Boxing	12.1	0.1
Breakout	30.5	1.7
ChopperCommand	7387.8	811
CrazyClimber	35829.4	10780.5
DemonAttack	1971	152.1
Freeway	29.6	0
Frostbite	4334.7	65.2
Gopher	2412.5	257.6
Hero	30826.4	1027
JamesBond	302.8	29
Kangaroo	3035	52
Krull	2665.5	1598
KungFuMaster	22736.3	258.5
MsPacman	6951.6	307.3
Pong	14.6	-20.7
PrivateEye	69571.3	24.9
Qbert	13455	163.9
RoadRunner	7845	11.5
Seaquest	42054.7	68.4
UpNDown	11693.2	533.4

The inter-quartile mean [1] (IQM) of a reinforcement learning algorithm that is evaluated on n tasks, with m evaluation runs per task, can be computed as the mean of the human normalised scores of those training runs that comprise the 25 - 75 percentile range of these $n \times m$ training runs. In doing so, this metric judges the algorithm on the group of games as a whole, while ignoring the outliers.

E.5. More Experimental Details

We present the scores achieved by all five independent runs of all agents trained on the 12-game subset of the Atari 100K test-suite in Table 4. Additionally, we also present the learning curves with error bars equal to a width of 1 standard error on each side are shown in Figure 9.

We present the scores of all five independent runs of EVaDE-SimPLe and SimPLe(30) agents trained on rest of the 14 games in the 100K test-suite in Table 5 and in Table 6, we present the mean scores achieved by SimPLe(30), EVaDE-SimPLe and other baselines in the Atari 100K test-suite.

shows the learning curves as shown in Figure 4 with error bars equal to a width of 1 standard error on each side.

Looking at the learning curves presented in Figure 4, it can possibly be said that an increase in scores of SimPLe(30) equipped with one of the EVaDE layers at a particular iteration would mean an increase in scores of EVaDE-SimPLe, albeit in later iterations. This pattern can clearly be seen in the games of BankHeist, Frostbite, Kangaroo, Krull and Qbert. This delay in learning could possibly be attributed to the agent wasting its interaction budget exploring areas suggested by one of the layers that is ineffective for that particular game. However, we hypothesise that since all the layers provide different types of exploration, their combination is more often helpful than wasteful. This is validated by the fact that EVaDE-SimPLe achieves higher mean HNS, IQM and SimPLe-NS than any other agent in this study.

Table 4: Scores achieved by every independent run of every SimPLe agent and when equipped with different EVaDE layers in the 12 game subset of the Atari 100K test-suite.

Game	SimPLe(30)	Inter. Layer	Weight. Layer	Trans. Layer	EVaDE-SimPLe
BankHeist	133.1	85	232.2	218.4	155.3
	9.375	12.5	195.3	128.8	205.9
	13.13	186.9	154.7	158.4	347.8
	69.38	142.8	130.6	187.5	250.9
	167.8	110.3	129.4	210.6	160.9
BattleZone	4156	1313	9250	4438	10844
	6969	8031	4250	6000	9375
	3344	9781	4938	6844	11063
	5719	4750	3906	9313	11219
	2531	9563	15281	12156	12969
Breakout	20.09	8.563	29.78	14.45	35.38
	18.25	25.03	27.56	23.64	20.91
	20.94	24.81	0.625	19.69	20.5
	12.69	14.25	26.13	21.13	15.84
	22.69	26.53	28	18.81	27.59
CrazyClimber	54569	57534	75300	69494	55194
	51244	58522	74141	59838	59934
	12959	62266	65431	61503	70719
	47391	69391	47234	53328	68747
	51128	50019	58847	50866	48984
DemonAttack	55.31	215.3	134.1	129.7	169.1
	112.7	71.41	155.5	105.9	100.9
	127.7	152.2	142.2	62.5	166.4
	159.5	102	75.31	151.1	141.1
	148.1	140.6	153	219.5	131.3
Frostbite	261.3	256.6	250	263.4	268.4
	251.9	241.6	259.4	258.1	249.4
	259.1	242.2	259.1	267.5	268.4
	262.5	268.1	261.3	259.1	315.6
	266.9	264.4	242.2	268.1	269.4
JamesBond	268.8	12.5	59.38	371.9	232.8
	240.6	282.8	332.8	117.2	101.6
	256.3	82.81	92.19	23.44	228.1
	257.8	350	126.6	262.5	203.1
	204.7	282.8	301.6	26.56	412.5
Kangaroo	987.5	3294	293.8	25	1144
	56.25	1588	362.5	1481	956.3
	112.5	37.5	175	1719	1444
	37.5	5500	1756	1681	1663
	1688	587.5	1600	1581	725
Krull	5639	6124	5150	5548	5569
	4873	3103	3290	7266	4906
	2868	2142	4460	1443	5744
	7035	2384	5386	6236	3864
	2244	1831	5806	5430	6591
Qbert	3002	3935	516.4	1193	1082
	4151	1133	1420	4190	3916
	4106	857	873.4	3494	4208
	806.3	3198	1034	3325	3983
	849.2	640.6	814.1	4462	631.3
RoadRunner	2793	8794	3034	5709	8666
	831.3	8744	4763	6763	8541
	5034	6188	7397	12622	9538

	3219	2791	8069	2581	9566
	46.88	9375	1000	2675	2684
	392.5	791.3	221.3	649.4	536.3
	419.4	286.3	288.1	604.4	813.8
Seaquest	395	692.5	849.4	854.4	861.9
	249.4	760.6	851.3	513.8	671.9
	151.9	691.3	832.5	598.8	203.8

Table 5: Scores achieved by every independent run of every SimPLe(30) and EVaDE-SimPLe agent when trained on the remaining 14 games of the Atari 100K test suite.

Game	SimPLe(30)	EVaDE-SimPLe
Alien	579.1	671.9
	494.1	605.9
	387.5	545.9
	330.3	444.1
	33.44	595.6
Amidar	90.25	112.1
	29.56	128.7
	92.25	171.5
	84.41	99.25
	75.19	149.8
Assault	2170	1255
	895.2	868.1
	901.9	1315
	768.4	868.8
	961.5	837.8
Asterix	1559	1228
	345.3	1322
	1269	1150
	1569	1727
	403.1	917.2
Boxing	47.91	26.69
	35.94	44.63
	41.56	44.66
	27.81	42.16
	17.38	39.44
ChopperCommand	859.4	734.4
	878.8	984.8
	487.5	953.1
	821.9	818.2
	872.7	875
Freeway	33.63	33.94
	20.84	32.5
	32.47	33.66
	33.44	33.31
	33.78	33.41
Gopher	659.4	752.5
	293.8	793.1
	690.6	1854
	656.9	530
	1038	423.8
Hero	3028	3056
	2908	2904
	2976	3009
	71.88	3275
	3079	3004
KungFuMaster	13703	14175
	17406	14384
	13481	21191
	11175	21684
	13006	14248
MsPacman	1194	1794
	939.1	1551
	1400	1050

	1118	1483
	1058	1688
Pong	4.313	6.375
	-1.781	16.13
	-17.78	13.03
	-7.781	10.22
	17.31	20.09
PrivateEye	0	34.09
	-38.25	100
	332	0
	4071	100
	100	100
UpNDown	566.9	1452
	1163	1182
	1016	1681
	1870	1586
	236.6	1264

Table 6: Mean scores achieved by SimPLe(30), EVaDE-SimPLe and other popular baselines in the Atari 100K test-suite.

Game	SimPLe	SimPLe(30)	Curl	OTRainbow	Eff Rainbow	EVaDE-Simple
Alien	616.9	364.888	558.2	**824.7**	739.9	572.68
Amidar	88	74.332	142.1	82.8	**188.6**	132.27
Assault	527.2	**1139.4**	600.6	351.9	431.2	1028.94
Asterix	1128.3	1029.08	734.5	628.5	470.8	**1268.84**
BankHeist	34.2	78.557	131.6	182.1	51	**224.16**
BattleZone	5184.4	4543.8	**14870**	4060.6	10124.6	11094
Boxing	9.1	34.12	1.2	2.5	0.2	**39.516**
Breakout	16.4	18.932	4.9	9.84	1.9	**24.024**
ChopperCommand	**1246.9**	784.06	1058.5	1033.33	861.8	873.1
CrazyClimber	**62583.6**	43458.2	12146.5	21327.8	16185.3	60715.6
DemonAttack	208.1	120.662	**817.6**	711.8	508	141.76
Freeway	20.3	30.832	26.7	25	27.9	**33.364**
Frostbite	254.7	260.34	**1181.3**	231.6	866.8	274.24
Gopher	771	667.74	669.3	778	349.5	**870.68**
Hero	2656.6	2412.576	6279.3	6458.8	**6857**	3049.6
Jamesbond	125.3	245.64	**471**	112.3	301.6	235.62
Kangaroo	323.1	576.35	872.5	605.4	779.3	**1186.46**
Krull	4539.9	4531.8	4229.6	3277.9	2851.5	**5334.8**
KungFuMaster	**17257.2**	13754.2	14307.8	5722.2	14346.1	17136.4
MsPacman	1480	1141.82	1465.5	941.9	1204.1	**1513.2**
Pong	12.8	-1.1438	-16.5	1.3	-19.3	**13.169**
PrivateEye	58.3	**892.95**	218.4	100	97.8	66.818
Qbert	1288.8	2582.9	1042.4	509.3	1152.9	**2764.06**
RoadRunner	5640.6	2384.836	5661	2696.7	**9600**	7799
Seaquest	**683.3**	321.64	384.5	286.92	354.1	617.54
UpNDown	**3350.3**	970.5	2955.2	2847.6	2877.4	1433

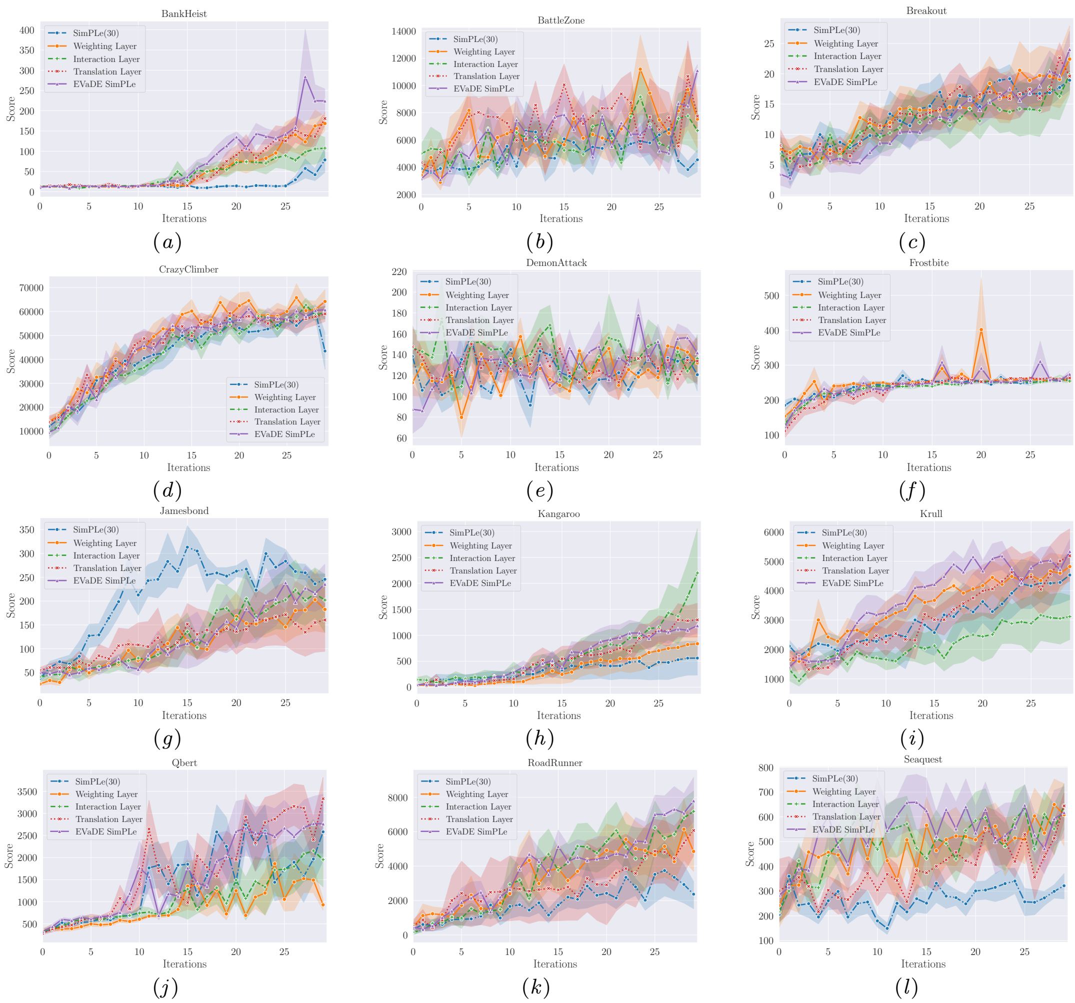


Figure 9: Learning curves of EVaDE-SimPLe agents, SimPLe(30) agents and agents which only add one of the EVaDE layers with error bars of 1 standard error.

References

[1] Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C Courville, and Marc Bellemare. Deep Reinforcement Learning at the Edge of the Statistical Precipice. *Advances in Neural Information Processing Systems*, 34, 2021.

[2] Shipra Agrawal and Randy Jia. Optimistic Posterior Sampling for Reinforcement Learning: Worst-Case Regret Bounds. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper/2017/file/3621f1454cacf995530ea53652ddf8fb-Paper.pdf.

[3] Siddharth Aravindan and Wee Sun Lee. State-Aware Variational Thompson Sampling for Deep Q-Networks. In *20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021)*, pages 124–132, 2021.

[4] Kamyar Azizzadenesheli, Emma Brunskill, and Animashree Anandkumar. Efficient Exploration through Bayesian Deep Q-Networks. In *2018 Information Theory and Applications Workshop (ITA)*, pages 1–9. IEEE, 2018.

[5] Mohammad Babaeizadeh, Chelsea Finn, Dumitru Erhan, Roy H. Campbell, and Sergey Levine. Stochastic Variational Video Prediction. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=rk49Mg-CW.

[6] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying Count-Based Exploration and Intrinsic Motivation. In *Advances in Neural Information Processing Systems*, pages 1471–1479, 2016.

[7] Marc Brittain, Joshua R. Bertram, Xuxi Yang, and Peng Wei. Prioritized Sequence Experience Replay. *CoRR*, abs/1905.12726, 2019. URL http://arxiv.org/abs/1905.12726.

[8] Yevgen Chebotar, Mrinal Kalakrishnan, Ali Yahya, Adrian Li, Stefan Schaal, and Sergey Levine. Path Integral Guided Policy Search. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 3381–3388. IEEE, 2017.

[9] Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[10] Sebastian Curi, Felix Berkenkamp, and Andreas Krause. Efficient Model-Based Reinforcement Learning through Optimistic Policy Search and Planning. *Advances in Neural Information Processing Systems*, 33, 2020.

[11] Meire Fortunato, Mohammad Azar, Bilal Piot, Jacob Menick, Matteo Hessel, Ian Osband, Alex Graves, Volodymyr Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy Networks for Exploration. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=rywHCPkAW.

[12] Yarin Gal and Zoubin Ghahramani. Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. In *international conference on machine learning*, pages 1050–1059. PMLR, 2016.

[13] Karol Gregor, George Papamakarios, Frederic Besse, Lars Buesing, and Theophane Weber. Temporal Difference Variational Auto-Encoder. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=S1x4ghC9tQ.

[14] David Ha and Jürgen Schmidhuber. Recurrent World Models Facilitate Policy Evolution. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 2455–2467, 2018.

[15] 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.

[16] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

[17] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[18] Thomas Jaksch, Ronald Ortner, and Peter Auer. Near-Optimal Regret Bounds for Reinforcement Learning. *Journal of Machine Learning Research*, 11(Apr):1563–1600, 2010.

[19] Kacper Kielak. Importance of Using Appropriate Baselines for Evaluation of Data-Efficiency in Deep Reinforcement Learning for Atari. *CoRR*, abs/2003.10181, 2020. URL https://arxiv.org/abs/2003.10181.

[20] Diederik P Kingma, Tim Salimans, and Max Welling. Variational Dropout and the Local Reparameterization Trick. In *Proceedings of the 28th International Conference on Neural Information Processing Systems-Volume 2*, pages 2575–2583, 2015.

[21] Michael Laskin, Aravind Srinivas, and Pieter Abbeel. CURL: Contrastive Unsupervised Representations for Reinforcement Learning. In *International Conference on Machine Learning*, pages 5639–5650. PMLR, 2020.

[22] Sergey Levine and Vladlen Koltun. Guided Policy Search. In *International conference on machine learning*, pages 1–9. PMLR, 2013.

[23] Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning. *CoRR*, abs/1708.02596, 2017. URL http://arxiv.org/abs/1708.02596.

[24] Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard Lewis, and Satinder Singh. Action-Conditional Video Prediction Using Deep Networks in Atari Games. In *Proceedings of the 28th International Conference on Neural Information Processing Systems-Volume 2*, pages 2863–2871, 2015.

[25] Ian Osband and Benjamin Van Roy. Bootstrapped Thompson Sampling and Deep Exploration. *arXiv preprint arXiv:1507.00300*, 2015.

[26] Ian Osband and Benjamin Van Roy. Why is Posterior Sampling Better than Optimism for Reinforcement Learning? In *International Conference on Machine Learning*, pages 2701–2710, 2017.

[27] Ian Osband, Daniel Russo, and Benjamin Van Roy. (More) Efficient Reinforcement Learning via Posterior Sampling. In *Advances in Neural Information Processing Systems*, pages 3003–3011, 2013.

[28] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep Exploration via Bootstrapped DQN. In *Advances in Neural Information Processing Systems*, pages 4026–4034, 2016.

[29] Ian Osband, Benjamin Van Roy, and Zheng Wen. Generalization and Exploration via Randomized Value Functions. In *Proceedings of the 33rd International Conference on Machine Learning-Volume 48*, pages 2377–2386. JMLR. org, 2016.

[30] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter Space Noise for Exploration. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=ByBAl2eAZ.

[31] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution Strategies as a Scalable Alternative to Reinforcement Learning. *arXiv preprint arXiv:1703.03864*, 2017.

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

[33] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

[34] Malcolm Strens. A Bayesian Framework for Reinforcement Learning. In *International Conference on Machine Learning*, volume 2000, pages 943–950, 2000.

[35] Richard S Sutton. Dyna, an Integrated Architecture for Learning, Planning, and Reacting. *ACM Sigart Bulletin*, 2(4):160–163, 1991.

[36] William R Thompson. On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples. *Biometrika*, 25(3/4):285–294, 1933.

[37] Iñigo Urteaga and Chris Wiggins. Variational Inference for the Multi-Armed Contextual Bandit. In *International Conference on Artificial Intelligence and Statistics*, pages 698–706. PMLR, 2018.

[38] Hado van Hasselt, Matteo Hessel, and John Aslanides. When to Use Parametric Models in Reinforcement Learning? In *NeurIPS*, pages 14322–14333, 2019.

[39] Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Lukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. Tensor2Tensor for Neural Machine Translation. *CoRR*, abs/1803.07416, 2018. URL http://arxiv.org/abs/1803.07416.

[40] Zhendong Wang and Mingyuan Zhou. Thompson Sampling via Local Uncertainty. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 10115–10125. PMLR, 13–18 Jul 2020. URL http://proceedings.mlr.press/v119/wang20ab.html.

[41] Grady Williams, Andrew Aldrich, and Evangelos Theodorou. Model Predictive Path Integral control Using Covariance Variable Importance Sampling. *arXiv preprint arXiv:1509.01149*, 2015.

[42] Sirui Xie, Junning Huang, Lanxin Lei, Chunxiao Liu, Zheng Ma, Wei Zhang, and Liang Lin. NADPEx: An On-Policy Temporally Consistent Exploration Method for Deep Reinforcement Learning. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=rkxciiC9tm.

[43] Weirui Ye, Shaohuai Liu, Thanard Kurutach, Pieter Abbeel, and Yang Gao. Mastering atari games with limited data. *Advances in Neural Information Processing Systems*, 34, 2021.

[44] Ruiyi Zhang, Zheng Wen, Changyou Chen, Chen Fang, Tong Yu, and Lawrence Carin. Scalable Thompson Sampling via Optimal Transport. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 87–96. PMLR, 2019.

[45] Kaiser Łukasz, Babaeizadeh Mohammad, Miłos Piotr, Osiński Błażej, Campbell Roy, H, Czechowski Konrad, Erhan Dumitru, Finn Chelsea, Kozakowski Piotr, Levine Sergey, Mohiuddin Afroz, Sepassi Ryan, Tucker George, and Michalewski Henryk. Model Based Reinforcement Learning for Atari. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=S1xCPJHtDB.