2018 17th IEEE International Conference on Machine Learning and Applications

Deep Reinforcement Learning Monitor for Snapshot Recording

Giang Dao, Indrajeet Mishra and Minwoo Lee
Department of Computer Science
University of North Carolina at Charlotte
{gdao, imishra, minwoo.lee}@uncc.edu

***Abstract*— Deep reinforcement learning (DRL) has been leading to state-of-the-art performance to learn control policies for a wide range of applications. However, it does not provide an explanation of how a policy is learned and how the learned policy performs on a given task. In this paper, we answer to the inquiry: what scenes does a machine learning agent need to memorize for efficient learning and additional explanation regarding performance? Proposing a monitoring model to record the most important moments from experience–called *snapshot images*–we examine them for analysis. Sparse Bayesian Reinforcement Learning (SBRL) [1] is known to remember sparse input samples during training and to construct bases for value function approximation. Also, SBRL has successfully maintained the snapshot memory for sparse input sampling. We apply our method to a visual maze problem and Atari games to observe the recorded snapshot images. Analyzing the images, we evaluate the efficacy of the proposed monitoring model and the quality of collected snapshots.**

I. INTRODUCTION

With the success of deep learning, there are a lot of applications built based on deep learning to solve complex tasks from unprocessed, high dimensional input data. By combining advances in deep learning with sensory input processing [2] and reinforcement learning, Mnih et al. [3] have suggested Deep Q-Network (DQN) algorithm, determining actions given state inputs as raw images. DQN algorithm is capable of playing Atari games by taking raw images data through the screen at human-level of control. Many other reinforcement learning algorithms have adopted deep neural networks for Deep Reinforcement Learning (DRL) such as DQN, Deep Deterministic Policy Gradient (DDPG) [4], and Asynchronous Advantage Actor Critic (A3C) [5]. Adopting deep neural networks, DRL has shown its successful applications to many different, complex decision making and control problems. There are a wide range of real world applications including robotics [6], [7], allocating cloud computing resource [8], advertisement technology [9], and finance [10].

However, the deep neural networks, the black box model, are not capable of interpreting what they have learned [11]. There is lack of understanding of what a deep neural network model has learned. Thus, when an erroneous event happens, finding causes and fixing them can be a tedious process that requires time and effort. For example, it is very hard for neural networks to explain why self-driving car crashes (i.e. Tesla and Uber) happen or the reason for the wrong decision in neural networks due to lack of interpretation.

Several previous publications [12], [13], [14], [15] have tried to understand how neural networks' computational process works through visualization in supervised learning problems. These works, interpreting neural networks, analyze neural networks after the procedures converge. Therefore, it is not enough to fully understand what deep learning has learned during its learning phase. Furthermore, all of these works targeted supervised learning problems where the target is known and stays unchanged, which cannot be directly applicable to reinforcement learning tasks.

To the best of our knowledge, only Zahvay et al. [16] and Greydanus et al. [17] attempted to visualize to help us understand deep reinforcement learning policies. The method by Zahvay, et al., however, is not able to directly handle raw image state and need handcrafted features. Greydanus, et al. display a part of the input images play important role in the policy development by observing how the policy is affected by the changes in the image pixels values. Although the method in [17] helps us point out the most important part of input images and its computation in neural networks, it is not enough for us to visualize the policy learned by the agent. Moreover, the visualization is only when the policy has converged. That is not enough to show what and how the agent has learned through the training process.

SBRL [1] has first introduced solutions for multiple feature-engineered state environments such as cart pole, mountain car, and simulated octopus arm control. Most noticeably, the SBRL is capable of remembering important experiences during training a reinforcement learning agent. The experiences are stored to perform the analysis and interpretation of the learning process and the learned behavior. The stored experiences also prevent forgetting important samples and; more importantly, increase the stability of learning. However, SBRL requires a technical feature-engineering, so its practical application might be limited.

In this work, we present a novel method, DRL-Monitor, which automatically captures the most important moments with visual representation during training to explain why an agent makes a decision given a situation. DRL-Monitor combines DRL that trains an agent and SBRL that extracts and stores important memories. SBRL's stored experiences can be used to fill in the gap of interpreting and understanding deep reinforcement learning. Therefore, DRL with convolutional layer solves complex reinforcement learning problems with raw image inputs while DRL-Monitor complement DRL with additional interpretation with recorded images of the

978-1-5386-6805-4/18/$31.00 ©2018 IEEE
DOI 10.1109/ICMLA.2018.00095

training experiences. The stored experiences can be easily visualized to understand how agent develops its learning ability through its learning process. We apply this method to a navigation task and Atari games to examine what an agent has learned to understand the learning process and the outcomes.

Our main contributions in this work are 1) presenting a novel method of memorizing important moments during the DRL training, which can be 2) providing Bayesian inferences for further analysis, and 3) can be applied with any state-of-the-art DRL algorithms.

In Section II, we outlines the backgrounds for the proposed work. The DRL-Monitor framework is introduced in Section III, and results of applying our framework in several experiments are presented in Section IV. We review and discuss related works in Section V and draw conclusions in Section VI.

II. Background

A. Reinforcement Learning

A reinforcement learning agent learns from interaction with an environment. An environment can be described by a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, a reward function $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, and a discount factor $\gamma \in [0, 1]$. An environment starts with an initial state s_0.

A deterministic policy π maps states to actions, $\pi : \mathcal{S} \rightarrow \mathcal{A}$. At timestep t, an agent takes an action a_t based on the current policy π and state s_t, $a_t = \pi(s_t)$, and it gets the reward $r_t = r(s_t, a_t)$ and observes a transition to the next state s_{t+1}. The return is computed with the discounted future rewards $R_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$ where T is the timestep that the environment terminates.

An action value function, Q value function, estimates the expected return: $Q(s_t, a_t) = \mathbb{E}[R_t | s_t, a_t]$. The agent's goal is to maximize the expected returns using the following Bellman equation:

$$Q^*(s_t, a_t) = \mathbb{E}_{s_{t+1} \sim \epsilon}[r(s_t, a_t) + \gamma Q^*(s_{t+1}, a_{t+1})]$$

where Q^* denotes the optimal Q value function.

A greedy action selection rule is defined as the agent takes action based on the policy: $\pi(s) = \arg\max_{a \in \mathcal{A}} Q(s, a)$. As an alternative near-greedy method, ϵ-greedy explores the space by following a policy which takes a random action (uniformly sampled from $\mathcal{A}$) with small probability ϵ, independently of the Q value estimates, and takes a greedy action with a probability of $1 - \epsilon$.

B. Deep Q-Network

Deep Q-Network (DQN) [3] is one of the DRL algorithms, which learns from an interaction between an agent and an environment. DQN uses convolutional neural network for processing images. Therefore, DQN can understand what happens when given raw image input as state to solve a reinforcement learning task.

At each time step, from the interaction, a state transition tuple (s_i, a_i, r_i, s_{i+1}) is stored into an experience replay buffer. Randomly sampling training data from the replay buffer helps to provide an identical and independent distributions to the deep neural networks. Thus, it helps avoid possible sampling bias to improve learning performance of DQN.

To improve the stability of learning, when approximating Q^* values, DQN uses two deep neural networks, online network $Q^{online}(s_t, a_t)$ and target network $Q^{target}(s_t, a_t)$. The online network is trained using mini-batch gradient descent where the target

$$y_t = r_t + \gamma \max_{a \in \mathcal{A}} Q^{target}(s_{t+1}, a).$$

The loss can be set up as $\mathcal{L} = (y_t - Q^{online}(s_t, a_t))^2$. The target neural network is identically structured as the online network. The weights of Q^{target} are periodically set or slowly updated using Polyak-average method [18] with the weights of Q^{online}. Thus, the Q^{target} changes the estimation of Q values slower than the Q^{online}. Mnih, et al. [3] empirically showed the improved stability and performance of this structure in Atari games.

Double Deep Q-Network (Double DQN) [19] extends the DQN. Double DQN computes the target value y_t by using the both online and target networks as follows:

$$y_t = r_t + \gamma Q^{target}(s_{t+1}, \arg\max_{a \in \mathcal{A}} Q^{online}(s_{t+1}, a))$$

The authors observed further improved performance with Double DQN in their experiments.

C. Sparse Bayesian Reinforcement Learning

SBRL [1] is built based on the Sparse Bayesian Learning algorithm, also known as Relevance Vector Machines [20]. The kernel-based learning transforms $i-$th input data $\mathbf{x}_i$ into features, $\phi_i(\mathbf{x}) = k(\mathbf{x}, \mathbf{x}_i)$, using kernel function k. For reinforcement learning problems, the inputs are represented as the combination of state and action pair, $\mathbf{x} = [\mathbf{s}, \mathbf{a}]$. Since the action and state space are orthogonal, Lee [1] defines the kernel function as a product between two independent kernels for state and action: $k(\mathbf{x}, \mathbf{x}_i) = k_s(\mathbf{s}, \mathbf{s}_i) \times k_a(\mathbf{a}, \mathbf{a}_i)$. $\mathbf{\Phi}$ is defined as a matrix composed of feature vectors transformed by the kernel function: $\mathbf{\Phi} = [\phi(\mathbf{x}_1), \phi(\mathbf{x}_2), ..., \phi(\mathbf{x}_N)]$ where $\phi(\mathbf{x}_n) = [\phi_1(\mathbf{x}_n), \phi_2(\mathbf{x}_n), ..., \phi_N(\mathbf{x}_n)]^\top$.

Sparse Bayesian reinforcement learning assumes that the target $\mathbf{Q}$ is a weighted sum of the feature vectors $\hat{\mathbf{Q}} = \mathbf{\Phi}\mathbf{w}$ with some noise $\boldsymbol{\epsilon}$ such that:

$$\mathbf{Q} = \hat{\mathbf{Q}} + \boldsymbol{\epsilon} = \mathbf{\Phi}\mathbf{w} + \boldsymbol{\epsilon}$$

where $\boldsymbol{\epsilon}$ is zero-mean Gaussian noise with variance σ^2. Tipping et al. [21] suggested the initial variance: $\sigma^2 = 0.1 \times var(\mathbf{Q})$.

Let M be the number of scalar in vector $\mathbf{w}$, and initialize $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, ..., \alpha_M)^\top$, a set of hyper-parameters controlling the strength of the prior over the corresponding weights, to be infinity except for one starting as:

$$\alpha_i = \frac{||\boldsymbol{\phi}_i||^2}{||\boldsymbol{\phi}_i^\top \mathbf{Q}||^2 / ||\boldsymbol{\phi}_i||^2 - \sigma^2}.$$

Given $\boldsymbol{\alpha}$, the posterior parameter distribution $p(\mathbf{w}|\mathbf{Q}, \boldsymbol{\alpha}, \sigma^2)$ is the Gaussian distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ with $\mathbf{A} = \boldsymbol{\alpha}\mathbf{I}$:

$$\boldsymbol{\Sigma} = (\mathbf{A} + \sigma^{-2}\boldsymbol{\Phi}^\top\boldsymbol{\Phi})^{-1} \quad \text{and} \quad \boldsymbol{\mu} = \sigma^{-2}\boldsymbol{\Sigma}\boldsymbol{\Phi}^\top\mathbf{Q}.$$

The marginal likelihood over the weight parameter represents zero-mean Gaussian:

$$p(\mathbf{Q}|\mathbf{w}, \boldsymbol{\alpha}, \sigma^2) \sim \mathcal{N}(\mathbf{m}, \mathbf{C})$$

where the mean and the variance are

$$\mathbf{m} = \boldsymbol{\Phi}\boldsymbol{\mu} \quad \text{and} \quad \mathbf{C} = \sigma^2\mathbf{I} + \boldsymbol{\Phi}\mathbf{A}^{-1}\boldsymbol{\Phi}^\top.$$

The variance of the marginal likelihood $\mathbf{C}$ can be decomposed as:

$$\mathbf{C} = \mathbf{C}_{-i} + \alpha_i^{-1}\boldsymbol{\phi}_i\boldsymbol{\phi}_i^\top$$

where $\mathbf{C}_{-i}$ is $\mathbf{C}$ with the contribution of basis vector i removed. The *sparsity factor*, $s_i = \boldsymbol{\phi}_i^\top\mathbf{C}_{-i}^{-1}\boldsymbol{\phi}_i$, measures the extent of overlaps of $\boldsymbol{\phi}_i$ with the existing other bases. The *quality factor*, $q_i = \boldsymbol{\phi}_i^\top\mathbf{C}_{-i}^{-1}\mathbf{Q}$, measures the alignment error of $\boldsymbol{\phi}_i$ when the $i-$th output is excluded.

3 cases need to be considered:

- If $q_i^2 > s_i$ and $\alpha_i < \infty$, re-estimate α_i.
- If $q_i^2 > s_i$ and $\alpha_i = \infty$, add ϕ_i to the model and re-estimate α_i.
- If $q_i^2 \leq s_i$ and $\alpha_i < \infty$, delete ϕ_i from the model and set $\alpha_i = \infty$.

$\boldsymbol{\alpha}_i$ (when $q_i^2 > s_i$) and the noise level σ will be updated as following:

$$\alpha_i = \frac{s_i^2}{q_i^2 - s_i} \quad \text{and} \quad \sigma^2 = \frac{||\mathbf{Q} - \hat{\mathbf{Q}}||^2}{N - M + \sum_m \alpha_m\boldsymbol{\Sigma}_{mm}}.$$

The algorithm re-computes $\boldsymbol{\Sigma}$ and $\boldsymbol{\mu}$ and the followed process until a convergent condition is met. The relevant state and action pairs can be traced back by the $\boldsymbol{\phi}$ left in the model. The predict Q-value for new data $\mathbf{x}^{new}$:

$$\hat{\mathbf{Q}}^{new} = k(\mathbf{x}^{base}, \mathbf{x}^{new})\mathbf{w}^{base}$$

where $\mathbf{x}^{base}$ are significant samples, and $\mathbf{w}^{base}$ are the weights along with the samples.

III. Deep Reinforcement Learning Monitor

Fig. 1 shows the overall structure of Deep Reinforcement Learning-Monitor (DRL-Monitor) which consists of two parts. The first part is the deep reinforcement learning for learning and controlling agents. The second part is the novel *monitor* for recording the most significant moments. In the monitor, SBRL chooses which moments to be recorded, and we call them as *snapshots*. After that, *snapshot storage* permanently remembers those moments in the format of raw image, action, and weight distribution.

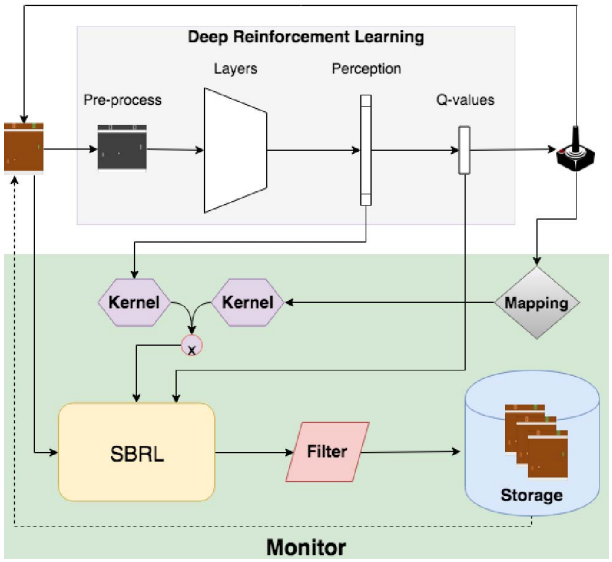


Fig. 1: The diagram for the DRL-Monitor framework. Here, we illustrate the Double DQN for the deep reinforcement learning module. The monitor can possibly be connected to many different DRL algorithms.

A. Deep Reinforcement Learning

This module is a main workhorse solving a reinforcement learning task. During both training and testing, the DRL module produces Q values or the learned policy. A raw image state from an environment is preprocessed through multiple layers (i.e. convolutional layers) to map into a feature vector, and we call the output features of these layers as *perception*. Perception is mapped through a fully connected layer to generate Q values or policies to take an action. The action taken affects the changes in the perception to continue exploration of the agent for training. The variation of the training loop follows the selected algorithm's implementation.

We confirm that DRL-Monitor is successfully applied to the Double DQN [19] model in Section IV. Using DRL-Monitor to monitor DQN [3], DDPG [4], and A3C [5] can be done in the same manner. Moreover, we observe that DRL-Monitor has potential applications to a large number of algorithms with perception layers. We reserve this for our future research. In this paper, we only applied DRL-Monitor in Double DQN to validate our work.

B. Monitor

From the DRL module, the monitor extracts final layer's perception by taking the outputs from the layer before the fully connected layers for Q estimation. The final layer's perception is a good feature vector representation of its complex input state (image).

For the cases that actions are in a discrete space, we mapped independent action inputs into a real value vector so that we can measure distances between actions precisely. For example, left: [-1, 0], right: [1, 0], up: [0, 1], and down: [0, -1].

In order to record significant snapshots, the radial basis kernel function computes the similarity of samples' state-action pair using the transformed perceptions and the mapped actions. The monitor trains the SBRL module with the kernel features to predict Q-values of the selected DRL. For the training step, it augments input data with the snapshots stored in the snapshot storage.

The kernel to train SBRL can be calculated as:

$$k(\mathbf{x}, \mathbf{x}_i) = k_s(\mathbf{p}, \mathbf{p}_i) \times k_a(\mathbf{a}, \mathbf{a}_i)$$

where $\mathbf{p}$ is the perception, $\mathbf{a}$ is mapped action, and i represents the index of input sample.

The snapshots, which are retained from SBRL training, are passed through a filter. In this filter, it examines whether the SBRL training was successful. When SBRL training is evaluated as success, the collected snapshots become candidates for permanent recording in the snapshot storage. One of the possible heuristic rules can be defined as:

$$\sqrt{\sum (\hat{\mathbf{Q}}_i - \mathbf{Q}_i)^2} < \tau \times \sqrt{\sum (\bar{\mathbf{Q}} - \mathbf{Q}_i)^2}$$

where $\mathbf{Q}$ is a target value, $\hat{\mathbf{Q}}$ is a predicted value, $\bar{\mathbf{Q}}$ is the average of the target $\mathbf{Q}$, and $\tau \in [0, 1]$ is a control parameter. We want SBRL prediction to be better than a naive prediction of average value by a certain ratio of τ.

The remaining snapshots are moved to the storage. For additional analysis and explanation of learning, the weight distributions (means and variance) are stored for the corresponding snapshots. This information provide the measurements of how strong a snapshot affects the environment or policy development.

If a new snapshot does not exist in the storage, the monitor uses the newly achieved weights, and add the snapshot to storage. If a snapshot was already in the storage, we updates the weight using the following convex update rule:

$$\mathbf{w}_i^{ss} = (1 - c) \times \mathbf{w}_i^{ss} + c \times \mathbf{w}_i^{new}$$

where $\mathbf{w}_i^{new}$ is the weight of the filtered candidate snapshot and $\mathbf{w}_i^{ss}$ is the weight of the same snapshot found in the storage. $c \in [0, 1]$ is a convex update parameter.

For the sparsity of the storage, we measures how different stored snapshots and new candidates are by computing their similarity using the kernel function. When new candidates have a similar snapshot in the storage, the similarity is measured higher than a preset threshold value. Then, instead of adding new snapshots (indexed by j), we updates the weight of the most similar snapshot (indexed by i) with the similarity ratio:

$$\mathbf{w}_i^{ss} = (1 - c) \times \mathbf{w}_i^{ss} + c \times (\sum_j k(\mathbf{x}_j, \mathbf{x}_i) \times \mathbf{w}_j^{new}).$$

IV. Experiments

In this section, we test the DRL-Monitor with three different game environments with vision inputs, Visual Maze, Atari Pong and MsPacman.

A. Deep Reinforcement Learning Agent

The DRL-Monitor can possibly be applied with any deep reinforcement learning agent. In our experiments, we choose Double Deep Q-Network [19], which has shown slightly improved performance of DQN. There are still some Atari games that Double DQN could not score better than human players. By using the DRL-Monitor, we explain how Double DQN learns, how it exploits learned knowledge and also why it fails to learn in some problems by examining the behavior of the Double DQN. The architecture of the neural network includes three convolutional layers with max pooling layers and one fully connected layer before outputting Q-values.

From our pilot tests, we found the following choice of parameters, which performs best. The replay buffer contains maximum 50,000 transition tuples. The mini-batch size is set to 64. We choose discount factor γ as 0.99. The similarity threshold is set at 0.99. The control parameter, $\tau = 0.5$ for maze and $\tau = 0.1$ for Pong and MsPacman. To simplify the experiment and analysis, the convex update parameter c is chosen to be 1.

B. Visual Maze

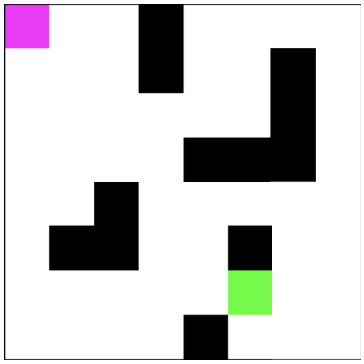


Fig. 2: Maze Environment

Visual Maze is a navigation (8×8 blocks) task with RGB image representation as state ($80 \times 80 \times 3$). An agent starts at the pink block and moves toward to the goal location in the green block. The black blocks represent obstacles. The four discrete actions are defined as left, right, up, and down. The agent receives -1 reward for the cost of each movement. If the agent moves out of the boundary or hit the obstacles, it stays at the current location, and it receives -5 as a penalty. If the agent reaches the goal, the game terminates, and it receives $+30$ reward.

Fig. 3 illustrates the gradual snapshot acquisition as the agent learns. The bottom figure shows a total reward of each episode and above three figures show the activate snapshots, whose weights are not close to 0's, in the snapshot storage. The collected active snapshots are relevant to the perceptions and the policy at that time. There are negative snapshots at first and in the middle of training process. When the training converges, we observe that there are only positive active snapshots left because the majority of samples in the

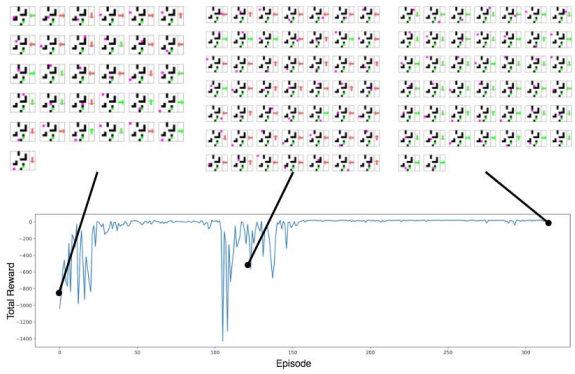


Fig. 3: DRL-Monitor gradually adds snapshots to a snapshot storage during learning. The upper images show the content of snapshot storage at each moment. Red actions (arrows) mean negative weights of the snapshots, and green actions mean positive weights.

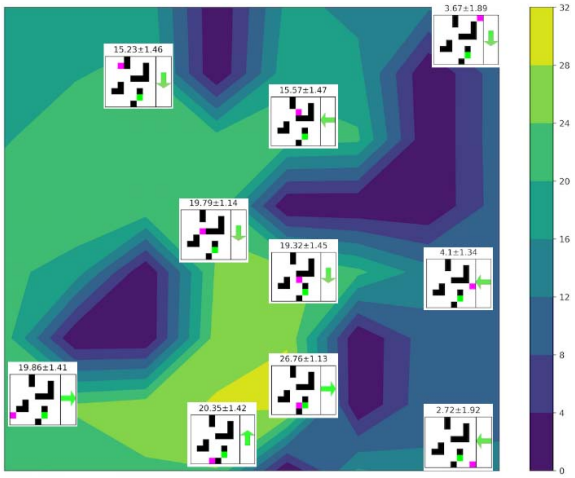
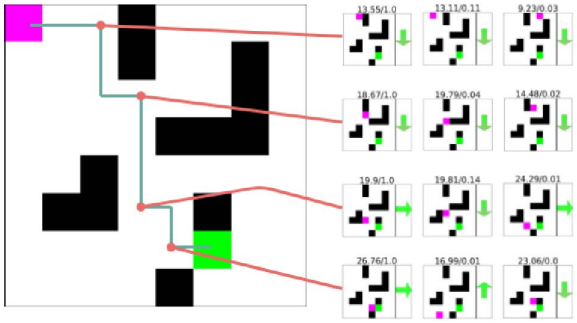


Fig. 4: Q contour plot after fully trained with snapshots. The numbers present for weight distribution.

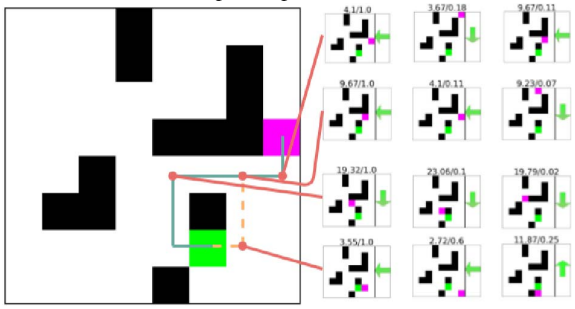
experience replay buffer contains positive ones as it exploits the learned policy more to reduce the steps to reach the goal.

Fig. 4 shows the maximum Q values in the maze and some of the active snapshot samples. When they lie on the path to the goal, and as getting closer to the goal, they end up of getting high Q values. In this path, the weights of those snapshots become high. We also observe that the weights are also affected by the exploration. Since the training always starts at the top left of the maze, there is less exploration made starting from the right side of the maze. This results in low weight values to those snapshots.

Once Double DQN achieves the knowledge from training, it exploits the learned policy as shown in Fig. 5. With two different starting positions, one with the same start (5a) and the other with a different start location (5b) from training, we present the three most relevant snapshots, which is computed by the kernel function. In Fig. 5a, the agent has developed an optimal policy that makes a right decision at each moment



(a) Optimal path selection



(b) Suboptimal path selection

Fig. 5: Exploitation of learned policy with different starting position: small images on the right are the snapshots for each circled moment. The snapshots are ordered from left to right showing its effects, and left has the highest effect. The numbers above each snapshot are weights and similarity.

(at the junction) and the snapshots has captured them. On the other hand, when we start from the start location with lack of experience, it came up with a sub-optimal path. Snapshots on the right show when and what were the wrong decisions.

The snapshots explain learned rule of what action an agent takes in a given state. The snapshots also provide information of which neural networks was able to learn. For example, there might be a dangerous zone that agent wants to take a sub-optimal solution.

C. *Pong*

Pong (Fig. 6) is one of Atari game environments that Double DQN has played very well. An agent controls the green bar. There are six possible actions: stay still, start the game (stay still if game already started), up, down, fire up (move up faster), fire down (move down faster). The agent gets $+1$ reward if the ball (white dot) passes brown bar to the left, and -1 reward if the ball passes green bar to the right. The game terminates when either the bot or the agent reaches 21 rewards (or points).

Fig. 7 shows active snapshots at 4 different training iteration (300K, 800K, 1.5M, 1.9M iterations). The bottom plot represents total rewards evaluated every 100K iterations. We observe that the slightly growing number of active snapshots thanks to the sparse control parameters in kernel, a filter heuristic, and snapshot storage. In our experiment,

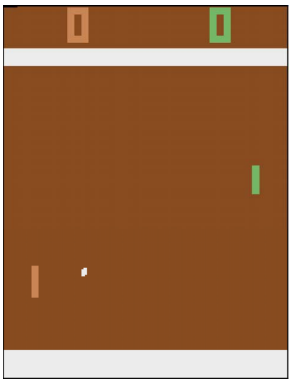


Fig. 6: OpenAI Gym Pong Environment

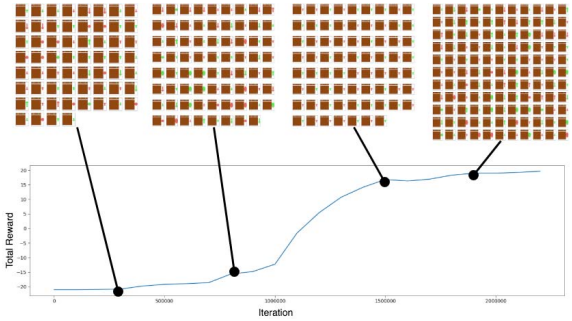
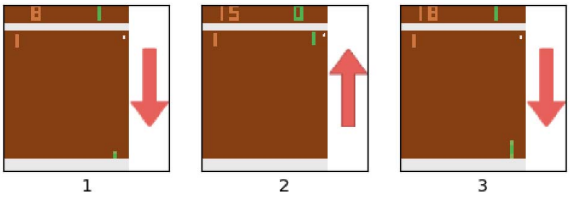


Fig. 7: DRL-Monitor on Pong with reward curve and active snapshots. Red actions mean negative weights of the snapshots, and green actions mean positive weights of the snapshots. Longer actions indicate a faster action toward a direction.

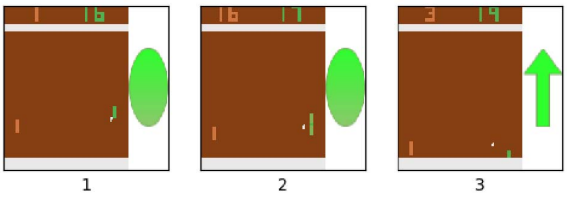
the total number of snapshots at the end of the training is 764 snapshots, and 221 of them are active.

Fig. 8 presents the three most effective snapshots that has the high absolute weight values at 300K, 1.5M, and 1.9M iteration. At 300K iteration, agent remembers key snapshots that lead directly to a negative reward signal because Double DQN has not been successfully train the agent. We can also see lack of training by looking at the reward curve in Fig. 7. As the training progresses, the agent learns how to catch the ball to avoid negative reward shown in Fig. 8b. However, the agent only knows how to correctly behave when the ball is close. When the agent starts to converge at 1.9M iteration, agent reuses more past experiences stored in snapshot storage to mark down a harmful state-action pair as well as to master how to move when the ball is far away.

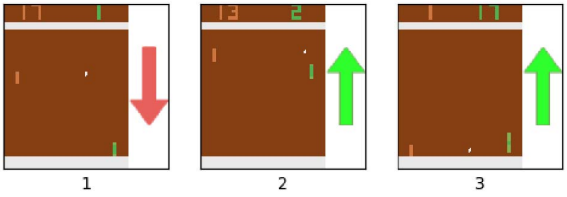
In Fig. 9, we visualize exploitation of the learned policy in a Pong game. Here, we show the Q-values at each frame of the video game screen. On the top, it shows how the agent exploits the winning policy with a game scene, active snapshots in the storage, and three most relevant snapshots. The three most relevant snapshots are selected by kernel similarity between the game scene and active snapshot image. Since the agent uses the same strategy repeatedly to defeat its opponent, the point 2 in the figure has three most



(a) Pong most effective snapshots at 300K iteration



(b) Pong most effective snapshots at 1.5M iteration



(c) Pong most effective snapshots at 1.9M iteration

Fig. 8: The three most effective snapshots each timestamps by the weights.

significant (similar) moments to send the ball through the winning trajectory shown in the dashed arrow line. There are three similar relevant snapshots at point 2 indicating a strong action selected such that the ball hit the edge of the agent to go back down and defeat its opponent.

D. MsPacman

In MsPacman (Fig. 10), an agent controls yellow Pacman. The agent will have three lives. There are 9 different actions: stay still, left, right, up, down, up right, up left, down right, and down left. If the agent eats one dot, the agent gets $+10$ reward. The agent loses one life if ghost catches it, and it is reseted to the starting position. The game terminates when the agent runs out of its lives.

We show the gradually snapshot collection as the agent learns with active snapshots at 4 different iterations (300K, 1.2M, 2.4M, and 3.6M) in Fig. 11 along with the reward curve. Comparing to Pong, there are more active snapshots are gathered at each time. We observe the total number of snapshots is 993, and the active snapshots at the end is 236.

As in Pong, we show the three most effective snapshots at 300K, 1.2M, and 2.4M iterations by comparing the absolute values of the weights (Fig. 12). At the beginning of the training, the snapshots are favor of staying still action. The agent understands that staying still is not a good action. In fact, the agent should always move in MsPacman game to be successfully survive. In the middle of the training when the agent is developing its policy, the agent learns the importance of the food which give positive immediate

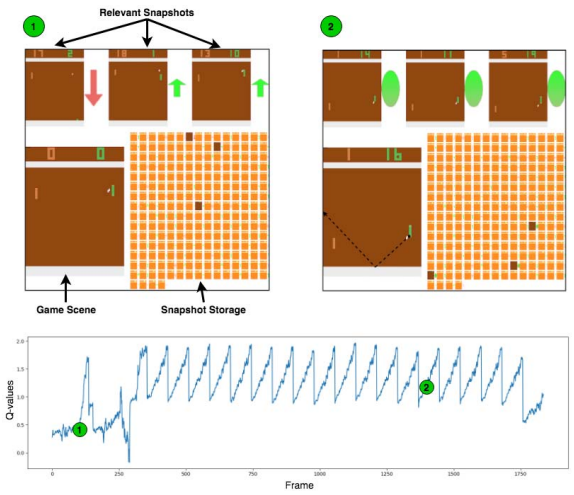


Fig. 9: Visualization of learned Q-values on Pong. The video is available at: `https://youtu.be/Si4SvglUjzk`.

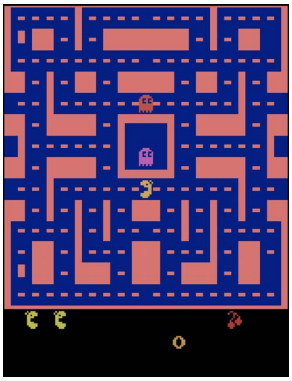
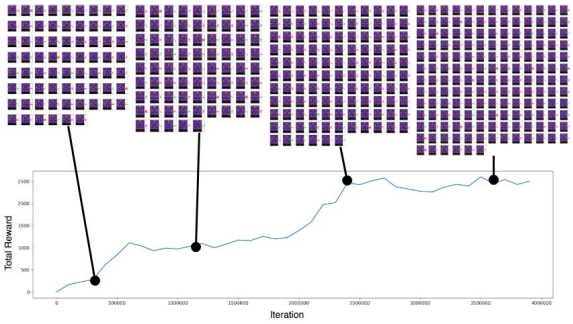
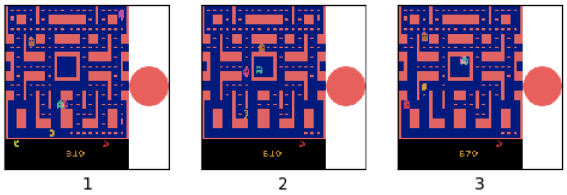


Fig. 10: OpenAI Gym MsPacman Environment

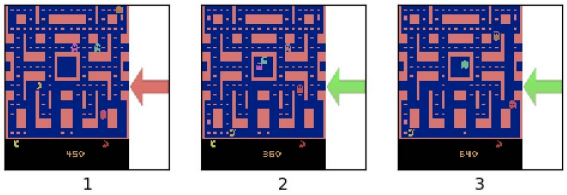


s

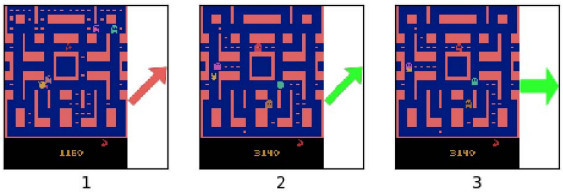
Fig. 11: DRL-Monitor on MsPacman with reward curve and active snapshots of 4 different timestamps. Red actions mean negative weights of the snapshots, and green actions mean positive weights of the snapshots



(a) MsPacman most effective snapshots at 300K iteration



(b) MsPacman most effective snapshots at 1.2M iteration



(c) MsPacman most effective snapshots at 2.4M iteration

Fig. 12: MsPacman most effective snapshots each timestamps by the weights. Red actions mean negative weights of the snapshots, and green actions mean positive weights of the snapshots.

rewards. Moreover, the agent develops better perception of the foods as well as its own location than previous iterations. When the Double DQN starts to converge, we observe that the agent avoids the ghost shown in Fig. 12c.1 by a negative weight of the snapshot.

However, Fig. 12c.2 and Fig. 12c.3 show that agent is heading towards to the ghost with positive weight snapshots. From the snapshot images, we observe that the agent is heading to the foods although the ghost is nearby. The agent overly weighted on eating foods to be safe keeping its lives. This is the key reason that Double DQN cannot reach human-level in MsPacman game. This might happen because of a sample bias or imbalance between positive and negative reward samples in the replay buffer so that the agent is not able to develop an optimal policy.

V. Discussions

Understanding and interpreting neural networks are active and important research area to handle various applications that require confidence or trust of a model. There has been a number of techniques proposed to understand what neural networks have learned. Interpretable learning of the complex model affects to selection of an efficient and effective network architectures, identification and mitigation of bias, accounting for the context of problems, and improvement of generalization and performance [22]. Especially in decision making problems, understanding what action is taken and

why the action is taken at a certain time is beneficial to interpret knowledge for transfer learning, to improve learning performance, and to fix potential errors in the critical domain as in autonomous car and medical applications.

Previous publications [12], [13], [15], [14], [22] have attempted to visualize the neural networks to understand the learned model for supervised learning problems. Not many have attempted to visualize reinforcement learning problems yet but recent work by Zahavy et al. [16] and Greydanus et al. [17]. Zahany et al. apply Semi Aggregate Markov Decision Process and t-SNE to visualize strategy of a DQN agent with an Atari game. Greydanus et al. visualize the A3C policies through a salient map for Atari game agents. By observing the changes in the policy learned by the agent in input images, the method make it possible to observe the salient pixels that is highly relevant to the learned policy.

However, these are not sufficient enough to understand what and how deep reinforcement learning agent has learned. In real world, we (as human) recalls relevant past experiences when encounter a new problem. To mimic this psychological information processing process, our work focuses on memorizing significant moments through training the monitor to has an automated retention of snapshots with Bayesian model that accounts for what an agent is doing.

The benefits of our method are that we have a sparse set of snapshots that takes less memory and is easy to interpret. The snapshots interpretation discovers unconsidered factors or features for learning process. The DRL-Monitor is complimentary to be combine with Graydanus, Zahany, or other visualization.

VI. Conclusions

In this work, we applied a new, novel method to monitor deep reinforcement learning by memorizing important moments during training. The stored images help interpret what and how the agent builds up its knowledge and solves a reinforcement learning task. The method can be applied with many different state-of-the-art DRL algorithms to understand and evaluate their learning. By adopting a Bayesian model, the interpretation was able to bring additional insights.

We have discussed about the observed evidences explaining possible reasons for unsuccessful Double DQN solution in MsPacman. Our next naturally move is to improve the Double DQN algorithms and re-verify the efficacy of evidence-based interpretation. Understanding the significance of sparsity, we will examine different kernel functions that can improve sparsity of DRL-Monitor. We will also extend the model application to various algorithms (other than Double DQN) and environments to verify the compatibility.

ACKNOWLEDGMENT

This work was supported, in part, by funds provided by the University of North Carolina at Charlotte. The Titan Xp used for this research was donated by the NVIDIA Corporation.

References

[1] M. Lee, "Sparse Bayesian Reinforcement Learning," Ph.D. dissertation, Colorado State University, 2017.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances In Neural Information Processing Systems*, pp. 1–9, 2012.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level Control through Deep Reinforcement Learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[4] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Hess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous Control with Deep Reinforcement Learning," *Foundations and Trends® in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2016.

[5] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," in *International Conference on Machine Learning*, 2016, pp. 1928–1937.

[6] Y. F. Chen, M. Everett, M. Liu, and J. P. How, "Socially Aware Motion Planning with Deep Reinforcement Learning," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 1343–1350.

[7] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. Pieter Abbeel, and W. Zaremba, "Hindsight Experience Replay," in *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., 2017, pp. 5048–5058.

[8] N. Liu, Z. Li, J. Xu, Z. Xu, S. Lin, Q. Qiu, J. Tang, and Y. Wang, "A Hierarchical Framework of Cloud Resource Allocation and Power Management Using Deep Reinforcement Learning," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 372–382.

[9] J. Zhao, G. Qiu, Z. Guan, W. Zhao, and X. He, "Deep Reinforcement Learning for Sponsored Search Real-time Bidding." *arXiv preprint arXiv:1803.00259*, 2018.

[10] Y. Deng, F. Bao, Y. Kong, Z. Ren, and Q. Dai, "Deep Direct Reinforcement Learning for Financial Signal Representation and Trading." *IEEE transactions on neural networks and learning systems*, vol. 28, no. 3, pp. 653–664, 2017.

[11] J. Yosinski, J. Clune, A. M. Nguyen, T. J. Fuchs, and H. Lipson, "Understanding Neural Networks Through Deep Visualization." *CoRR*, vol. abs/1506.06579, 2015. [Online]. Available: http://arxiv.org/abs/1506.06579

[12] C. Olah, A. Satyanarayan, I. Johnson, S. Carter, L. Schubert, K. Ye, and A. Mordvintsev, "The Building Blocks of Interpretability." *Distill*, 2018, https://distill.pub/2018/building-blocks.

[13] H. Li, Z. Xu, G. Taylor, and T. Goldstein, "Visualizing the Loss Landscape of Neural Nets." *arXiv preprint arXiv:1712.09913*, 2017.

[14] M. Tulio Ribeiro, S. Singh, and C. Guestrin, "Why Should I Trust You?: Explaining the Predictions of Any Classifier." *arXiv preprint arXiv:1602.04938*, 2016.

[15] G. Montavon, W. Samek, and K.-R. Müller, "Methods for Interpreting and Understanding Deep Neural Networks." *Digital Signal Processing*, 2017.

[16] T. Zahavy, N. Ben-Zrihem, and S. Mannor, "Graying the Black Box: Understanding DQNs." in *International Conference on Machine Learning*, 2016, pp. 1899–1908.

[17] S. Greydanus, A. Koul, J. Dodge, and A. Fern, "Visualizing and Understanding Atari Agents." *arXiv preprint arXiv:1711.00138*, 2017.

[18] B. Polyak and A. B. Juditsky, "Acceleration of Stochastic Approximation by Averaging." *SIAM Journal on Control and Optimization*, vol. 30, pp. 838–855, 07 1992.

[19] H. Van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-Learning." in *AAAI*, vol. 16, 2016, pp. 2094–2100.

[20] M. E. Tipping, "Sparse Bayesian Learning and the Relevance Vector Machine," *Journal of Machine Learning Research*, vol. 1, pp. 211–244, 2001.

[21] M. E. Tipping, A. C. Faul *et al.*, "Fast Marginal Likelihood Maximisation for Sparse Bayesian Models." in *Proceedings of International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2003.

[22] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization." in *ICCV*, 2017, pp. 618–626.