DQNViz: A Visual Analytics Approach to Understand Deep Q-Networks

Junpeng Wang, Liang Gou, Han-Wei Shen, *Member, IEEE*, and Hao Yang

Fig. 1. *DQNViz*: (a) the *Statistics* view presents the overall training statistics with line charts and stacked area charts; (b) the *Epoch* view shows epoch-level statistics with pie charts and stacked bar charts; (c) the *Trajectory* view reveals the movement and reward patterns of the DQN agent in different episodes; (d) the *Segment* view reveals what the agent really sees from a selected segment.

Abstract— Deep Q-Network (DQN), as one type of deep reinforcement learning model, targets to train an intelligent agent that acquires optimal actions while interacting with an environment. The model is well known for its ability to surpass professional human players across many Atari 2600 games. Despite the superhuman performance, in-depth understanding of the model and interpreting the sophisticated behaviors of the DQN agent remain to be challenging tasks, due to the long-time model training process and the large number of experiences dynamically generated by the agent. In this work, we propose *DQNViz*, a visual analytics system to expose details of the blind training process in four levels, and enable users to dive into the large experience space of the agent for comprehensive analysis. As an initial attempt in visualizing DQN models, our work focuses more on Atari games with a simple action space, most notably the Breakout game. From our visual analytics of the agent's experiences, we extract useful action/reward patterns that help to interpret the model and control the training. Through multiple case studies conducted together with deep learning experts, we demonstrate that *DQNViz* can effectively help domain experts to understand, diagnose, and potentially improve DQN models.

Index Terms—Deep Q-Network (DQN), reinforcement learning, model interpretation, visual analytics.

1 Introduction

Recently, a reinforcement learning (RL) agent trained by Google DeepMind [32, 33] was able to play different Atari 2600 games [8] and achieved superhuman level performance. More surprisingly, the superhuman level performance was achieved by taking only the game screens and game rewards as input, which makes a big step towards artificial general intelligence [14]. The model that empowers the RL agent with such capabilities is named Deep Q-Network (DQN [33]), which is a deep convolutional neural network. Taking the Breakout game as an example (Figure 2, left), the goal of the agent is to get the maximum reward by firing the ball to hit the bricks, and catching the ball with the paddle to avoid life loss. This is a typical RL problem (Figure 2, right), which trains an *agent* to interact with an *environment* (the game) and strives to achieve the maximum reward by following certain strategies. Through iterative trainings, the agent becomes more intelligent to interact with the environment to achieve high rewards.

- *Junpeng Wang and Han-Wei Shen are with The Ohio State University. E-mail: {wang.7665, shen.94}@osu.edu.*
- *Liang Gou and Hao Yang are with Visa Research. E-mail: {ligou, haoyang}@visa.com.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

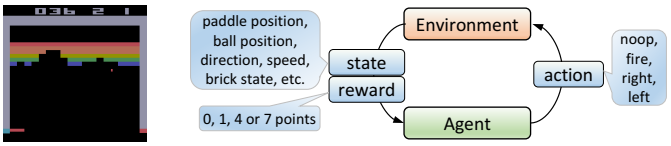

Fig. 2. The Breakout game and the reinforcement learning problem.

Despite the promising results, training DQN models usually requires in-depth know-how knowledge due to the following reasons. First, different from supervised/unsupervised learnings that learn from a predefined set of data instances, reinforcement learnings learn from the agent's experiences, which are dynamically generated over time. This requires dynamic summarizations of the experiences to achieve a good understanding of the training data. Second, interpreting the behavior of a DQN agent is also challenging. For example, when the agent moves the paddle left, what does the agent really see? Is this an intentional move or it is just a random choice? These questions are important to understand the agent, but cannot be directly answered by model statistics captured from conventional approaches. Finally, DQN models take a certain amount of random inputs during training (e.g., randomly moving the paddle in the Breakout game). The random inputs give the agent more flexibilities to explore the unknown part of the environment, but also prevent the agent from fully exploiting its intelligence. Therefore, a proper random rate is crucial to the training.

In recent years, we have witnessed the success of many visual analytics approaches to understand deep neural networks. These approaches cover supervised (e.g. CNNVis [27]) and unsupervised (e.g. GANViz [50]) deep learning models and are able to expose the models with multiple levels of details. However, to date, few visual analytics works have been reported for deep RL models. Additionally, visualization has played an increasingly important role in model diagnosis and improvement. For example, Liu et al. [26] showed how visualization can help in diagnosing training failures of deep generative models by disclosing how different neurons interact with each other. Through visualization, Bilal et al. [10] demonstrated the effects of the class hierarchy in convolutional neural networks and they successfully improved the models by considering the hierarchy. We believe similar attempts of diagnosing and improving models are also promising for deep RL models.

In this work, we propose *DQNViz*, a visual analytics system to understand, diagnose, and potentially improve DQN models. *DQNViz* helps domain experts understand the experiences of a DQN agent at four different levels through visualization. The agent's experiences are not only the inputs for next training stages, but also the outputs from previous training stages. Therefore, they decide what the agent will learn next, and also reflect what the agent has learned previously. By thoroughly studying those experiences with domain experts using *DQNViz*, we have identified several typical action and reward patterns, which are very useful in understanding the behavior of the agent, evaluating the quality of the model, and improving the performance of the training. For the challenge of understanding the agent's mind when performing an action, we dive into the structure of DQN and use guided back-propagations [46] to expose the features that different convolutional filters extracted. To sum up, the contributions of this work include:

- We present **a visual analytics system**, *DQNViz*, which helps to understand DQN models by revealing the models' details in four levels: *overall training* level, *epoch* level, *episode* level, and *segment* level.
- We propose **a visual design** for event sequence data generated from DQN models. The design can effectively reveal the movement patterns of an agent, and synchronize multiple types of event sequences.
- Through comprehensive studies with domain experts using *DQNViz*, we propose **an improvement in controlling random actions** in DQN models, which is directly resulted from our visual analytics.

2 Related Work

DQN Model and Model Challenges. Reinforcement learning (RL) aims to generate an autonomous agent interacting with an environment to learn optimal actions through trial-and-error. Researchers have developed three main approaches to address RL problems: value function based approaches [5,52], policy based approaches [23], and actor-critic approaches [24]. Different approaches have their respective merits and frailties, which have been thoroughly discussed in [4, 5]. Our work focuses on DQN [32,33], a value function based approach, to present an initial effort in probing RL problems with a visual analytics approach. DQN learns a Q-value function [5,52] for a given state-action pair with deep neural networks to handle the large number of input states (e.g., playing Atari games). We explain it with details in Section 3.

Recently, there are several important extensions of DQN models. Wang et al. [51] proposed dueling networks to learn a value function for states and an advantage function associated with the states, and combined them to estimate the value function for an action. Double DQN [17] tackles the over-estimation problem by using double estimators. Another important extension is the prioritized experience replay [41] that samples important experiences more frequently. Other extensions to reduce variability and instability are also proposed [3,19,20].

Despite these advances, there are several challenges to understand and improve DQNs. The first one is to understand the long-time training process. A DQN infers the optimal policy by enormous trial-and-error interactions with the environment, and it usually takes days/weeks to train the model [12]. It is not easy to effectively track and assess the training progress, and have intervention at the early stages. Secondly, it is not clear what strategies an agent learns and how the agent picks them up. The agent generates a huge space of actions and states with strong temporal correlations during the learning. It is helpful yet challenging to extract the dominant action-state patterns and understand how the patterns impact the training. Thirdly, it is a non-trivial task to incorporate domain experts' feedback into the training. For example, if experts observe some good/bad strategies an agent learned, they do not have a tool to directly apply such findings to the training. Finally, similar to other deep learning models, it needs numerous experiments to understand and tune the hyper-parameters of DQNs. One example is to understand the trade-off between explorations and exploitations [13], which is controlled by the exploration rate [35]. In this work, we try to shed some light on these challenges through a visual analytics attempt.

Deep Neural Networks (DNN) Visualization. The visualization community has witnessed a wide variety of visual analytics works for DNN in recent years [11,28]. According to the taxonomy in machine learning [39], those works can generally be categorized into works for *supervised* and *unsupervised* DNN. Example visualization works for supervised DNN include: CNNVis [27], RNNVis [31], LSTMVis [47], ActiVis [22], Blocks [10], DeepEyes [37], etc. For unsupervised DNN, DGMTracker [26] and GANViz [50] are typical examples.

One part missing from the above taxonomy [39] is the *reinforcement learning* [5], and few visual analytics works have been reported for this part. Specifically, for DQN, the visualization is limited to using t-SNE [30] to lay out activations from the last hidden layer of the model, as presented in the original DQN paper [33]. Although the result is effective in providing a structured overview of the large amount of input states, it is not interactive and only limited information is presented. The effectiveness of this preliminary visualization has also demonstrated the strong need of a comprehensive visual analytics solution.

Event Sequence Data Visualization. To explore the action space over time for reinforcement learnings, a time-oriented sequence visualization is of our interest. A large number of event sequence data visualization works can be found from literature [42], and here we focus on two categories: *flow-based* and *matrix-based* approaches according to [16]. The flow-based approaches use a time-line metaphor to list a sequence of events and extend them along one dimension (e.g. time). Multiple sequences usually share the same extending dimension and thus can be synchronized accordingly. Example visualization works in this group include: LifeLines [38], LifeFlow [53], CloudLines [25], EventFlow [34], DecisionFlow [15], etc. The matrix-based approaches, such as MatrixFlow [36] and MatrixWave [55], can effectively aggregate events and present them with compact matrices to avoid visual clutters. The combination of both types of approaches has also been proposed recently, e.g., EgoLines [54], EventThread [16]. We also focus on event sequence data in this paper, and our objective is to visualize *multiple types* of event sequences and enable users to synchronize and

analyze them simultaneously. On the one hand, we use multiple types of statistical charts to quantitatively summarize the event sequences. On the other hand, we propose a new visual design that can qualitatively reflect the behavior patterns of a DQN agent and synchronize different types of event sequences on-demand.

3 Background on Deep Q-Networks (DQN)

DQN [32, 33], as one type of reinforcement learning, aims to train an intelligent *agent* that can interact with an *environment* to achieve a desired *goal*. Taking the Breakout game as an example (Figure 2, left), the environment is the game itself, and it responds to any action (e.g. moving left) from an agent (the trained player) by returning the state of the game (e.g. paddle position) and rewards. With the updated state and achieved reward, the agent makes a decision and takes a new action for next step. This iterative interaction between the agent and environment (Figure 2, right) continues until the environment returns a terminal state (i.e. game-over), and the process generates a sequence of states, actions, and rewards, denoted as: $s_0, a_0, r_1, s_1, a_1, r_2, ..., r_n, s_n$. The desired goal is maximizing the total reward achieved by the agent.

How to maximize the total reward? The total reward for one game episode (i.e., from game-start to game-over) is: $R{=}r_1 + r_2 + ... + r_n$. Suppose we are at time t, to achieve the maximum total reward, the agent needs to carefully choose actions onwards to maximize its future rewards: $R_t{=}r_t + r_{t+1} + ... + r_n$. To accommodate the uncertainty introduced by the stochastic environment, a discount factor, $\gamma \in [0,1]$, is usually used to penalize future rewards. Therefore, $R_t{=}r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} ... + \gamma^{n-t} r_n{=}r_t + \gamma R_{t+1}$, i.e., the maximum reward from time t onwards equals the reward achieved at t plus the maximum discounted future reward. Q-learning [52] defines the maximum future reward as a function of the current state and the action taken in the state, i.e.,

$$Q(s,a){=}r{+}\gamma \max_{a'} Q(s',a'), \text{ and } s'/a' \text{ is the state/action after } s/a. \quad (1)$$

This equation is well known as the Bellman equation [9]. The problem of maximizing the total reward is to solve this equation now, which can be conducted through traditional dynamic programming algorithms.

Why DQN is needed? One problem in solving the Bellman equation is the algorithm complexity, especially when the number of states is large. In Breakout, the states should reflect the position, direction, speed of the ball and the paddle, the remaining bricks, etc. To capture such information, RL experts use four consecutive game screens as one state, which contains both static (e.g. paddle position) and dynamic (e.g. ball trajectory) information. As a result, each state has $84{\times}84{\times}4$ dimensions (each screen is a gray scale image of resolution $84{\times}84$), and the total number of states is $256^{84\times84\times4}$. Solving the Bellman equation with input in this scale is intractable. DQN, which approximates $Q(s,a)$ through a deep neural network, emerges to be a promising solution.

How does DQN work? The core component of DQN is a Q-network that takes screen states as input and outputs the q-value (expected reward) for individual actions. One popular implementation of the Q-network is using a deep convolutional neural network (explained later in Figure 7, left), which shows strong capabilities for image inputs [44]. The DQN framework consists of 4 major stages (Figure 3, left):

- The *Predict* stage (conducted by the Q-network with its current parameters θ_i) takes the latest state (4 screens) as input and outputs the predicted reward for individual actions. The action with the maximum reward, i.e., $arg\max_a Q(s,a;\theta_i)$, is the predicted action; and the maximum reward is the predicted q(uality) value, i.e., $q{=}\max_a Q(s,a;\theta_i)$.
- The *Act* stage is handled by the environment (an Atari game emulator, we used ALE [8] in this work). It takes the predicted action as input and outputs the next game screen, the resulted reward, and whether the game terminates or not. The new screen will be pushed into the State Buffer (a circular queue storing the latest four screens) and constitute a new state with the three previous screens, which is the input for the next *Predict* stage.
- The *Observe* stage updates the Experience Replay memory (ER, a circular queue with a large number of experiences) by pushing a new tuple (of the predicted action, the reward of the action, the new screen, and the terminal value) as an experience into the ER.
- The *Learn* stage updates the Q-network by minimizing the following loss at iteration i [33], i.e., the mean square error between q and q_t:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim ER}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right] \quad (2)$$

where (s,a,r,s') are random samples from the ER and θ_i^- are the parameters of the Q-network used to generate the *target* q at iteration i, i.e., $q_t{=}r{+}\gamma\max_{a'} Q(s',a';\theta_i^-)$. To stabilize q_t during training, θ_i^- are updated much less frequently (every C steps) by copying from θ_i [32, 33] ($C{=}1000$ in Figure 3 (left) as well as our model training process).

Exploration and Exploitation Dilemma [13]. In the *Predict* stage, actions are not always from the Q-network. A certain percentage of the actions are randomly generated. The reason is that we should not only exploit the intelligence of the agent to predict actions, but also explore the unknown environment with random actions. Usually, the ratio between exploration and exploitation is dynamically updated in the training (i.e., the value of ε in Equation 3 decays over time). However, choosing a proper value for this ratio (to handle the trade-off between exploration and exploitation) is still a very challenging problem.

$$\textit{predicted action}{=}\begin{cases} \textit{random action}, & \text{with probability } \varepsilon \\ arg\max_a Q(s,a;\theta_i), & \text{with probability } 1{-}\varepsilon \end{cases} \quad (3)$$

4 Requirement Analysis and Approach Overview

4.1 Design Requirements

We maintained weekly meetings with three domain experts in deep learning for more than two months to distill their requirements of a desired visual analytics system for DQN. All the experts have 3+ years experience in deep learning and 5∼10 years experience in machine learning. Through iterative discussions and refinements, we finally identified the following three main themes of requirements:

R1: Providing in-depth *model statistics* over a training. Having an overview of the training process is a fundamental requirement of the experts. In particular, they are interested in the following questions:

- *R1.1: How does the training process evolve, in terms of common statistical summaries (e.g., the rewards per episode, the model loss)?*
- *R1.2: What are the distributions of actions and rewards, and how do the distributions evolve over time?* For example, will the action distribution become stable (i.e., a roughly fixed ratio among different actions in an epoch) in later training stages?
- *R1.3: Can the overview reflect some statistics of the agent's action/movement/reward behaviors?* For example, are there any desired patterns that happen more often than others over time?

R2: Revealing the agent's *behavior patterns* encoded in the experience data. Demonstrating the action/movement/reward patterns of the agent is a strong need from the experts [4, 5], given that few existing tools are readily applicable for this purpose.

- *R2.1: Revealing the overall action/movement/reward patterns from a large number of steps.* Facing with a large number of experiences, the experts need an effective overview to guide their pattern explorations.
- *R2.2: Efficiently detect/extract patterns of interest to understand the agent's behavior.* It is nearly impossible to scrutinize the numerous data sequences to spot all interesting patterns (merely with visualization). A mechanism of pattern detection/extraction is desirable.
- *R2.3: Being able to present other types of data on-demand to facilitate comprehensive reasoning.* The q, q_t values, random actions, etc., are important context information when analyzing the agent's behaviors. Users should be able to bring them up flexibly.

R3: Empowering *segment analysis and comparisons* by looking through the agent's eyes. This requirement enables users to dive into the architecture of DQN, to analyze how the network works.

- *R3.1: Revealing what important features are captured and how they are captured by the agent.* Specifically, the experts are curious about the functionalities of each convolutional filter in terms of extracting features and making action predictions for experience segments.
- *R3.2: Comparing convolutional filters when processing different experience segments at the same training stage.* For example, domain experts are interested in if the same filter always extracts the same feature when handling different segments in the same epoch.

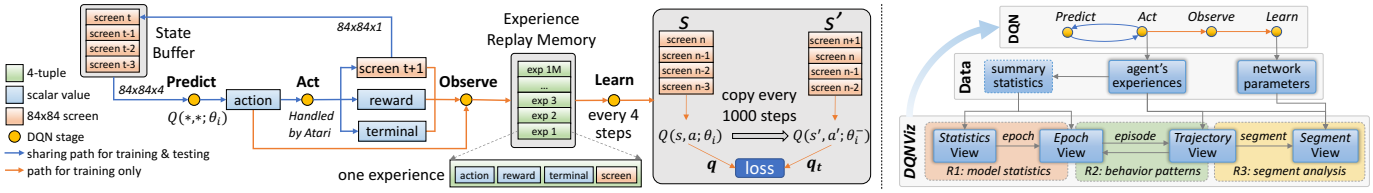

Fig. 3. Left: the four stages of DQN: *Predict*, *Act*, *Observe*, and *Learn*; right: the overview of our framework to analyze and improve DQN models.

- *R3.3: Comparing filters when processing the same experience segment at different stages.* This requirement aims to reveal if the agent treats the same experience segment differently in different epochs.

4.2 Approach Overview

Figure 3 (right) shows an overview of our approach to understand DQN with *DQNViz*. **First**, we train the DQN model and collect two types of data during the training: (1) the agent's experiences, which are heterogeneous time-varying sequences (Section 5); (2) the model losses and network parameters, which are used to assess the model quality and *read the agent's mind* at different training stages. A pre-processing on the experience data is performed to derive useful summary statistics, which include the average reward, average q values, etc. **Second**, the *DQNViz* system takes the two types of data and the derived statistics as input, and presents them to domain experts (Section 6). Aligned with the design requirements, the components of *DQNViz* are organized into three modules: (R1) model statistics, (R2) behavior patterns, and (R3) segment analysis, which are implemented through four visualization views. The four views, following a top-down exploration flow, present the collected data at four levels of details: overall training level (*Statistics* view), epoch-level (*Epoch* view), episode-level (*Trajectory* view), and segment-level (*Segment* view). **Lastly**, we demonstrate several case studies in which the knowledge learned from *DQNViz* has helped domain experts to diagnose and improve a DQN model (Section 7).

5 Training Process and Data Collection

We focus on the Breakout game to present *DQNViz* in this work, as it is one of the most frequently tested/used games in previous works, and a DQN agent can achieve superhuman performance on it.

In Breakout, the agent has five lives in each game episode. Life loss happens when the agent fails to catch the ball with the paddle. The game terminates if the agent loses all five lives. There are four possible actions: no-operation (*noop*), firing the ball (*fire*), moving left (*left*), and moving right (*right*). The agent receives rewards of 1, 4 and 7 when the ball hits bricks in the bottom two rows, middle two rows, and top two rows respectively. Otherwise, the reward is 0. On the top of the game scene, two numerical values indicate the current reward and the number of lives left (e.g., they are 36 and 2 in Figure 2, left).

We trained the DQN model from [1] for 200 epochs. Each epoch contains 250,000 training steps and 25,000 testing steps. The testing part (the blue paths in Figure 3, left) does not update the model parameters, and thus is used to assess the model quality. At each testing step, we collected the following eight types of data:

1. *action*: a value of 0, 1, 2 or 3 representing *noop*, *fire*, *right*, and *left*.
2. *reward*: a value of 0, 1, 4 or 7 for the reward from an action.
3. *screen*: an array of 84×84 values in the range of [0, 255].
4. *life*: a value in [1, 5] for the number of lives the agent has currently.
5. *terminal*: a boolean value indicating if an episode ends or not.
6. *random*: a boolean value indicating if an action is a random one.
7. q: the predicted q (a floating-point value) for the current step.
8. q_t: the target q value, i.e. q_t (see Section 3), for the current step.

At the training stage, the random rate ε starts with 1, decays to 0.1 in one million steps (i.e., 4 training epochs), and keeps to be 0.1 to the end. For testing, ε is always 0.05. During data collection, if an action is a random one, we still use the DQN to derive its q and q_t value, though the action to be executed will be the randomly generated one.

There is an inherent hierarchy in the collected experiences. A ***step*** that composed by the eight types of data is an atomic unit. A ***segment*** is a consecutive sequence of steps in an episode with a customized length. An ***episode*** includes all steps from a game-start to the corresponding game-end (five lives). A testing ***epoch*** contains 25,000 steps. In summary, the relationship for them is: $step \subseteq segment \subseteq episode \subseteq epoch$.

6 Visual Analytics System: *DQNViz*

Following the requirements (Section 4.1), we design and develop *DQNViz* with four coordinated views, which present the data collected from a DQN model at four different levels in a top-down order.

6.1 *Statistics* View: Training Process Overview

The *Statistics* view shows the overall training statistics of a DQN model with line charts and stacked area charts. Those charts present the entire DQN training process over time, along the horizontal axis.

The line charts (revealing the trend of different summary statistics over the training) are presented as small-multiples [49] (R1.1, R1.3). As shown in Figure 1a, the five line charts track five summary statistics (from left to right): *average rewards per episode*, *number of games per epoch*, *average q value*, *loss value*, and *number of bouncing patterns*. Users are able to plug other self-defined statistics into this view, such as *maximum reward per episode*, *number of digging patterns*, etc.

The two stacked area charts demonstrate the distribution of actions and rewards over time (R1.2). The evolution of action/reward distributions provides evidence to assess the model quality. For example, by looking at the distribution of reward 1, 4, and 7 in Figure 1-a2, one can infer that the model training is progressing towards the correct direction, as the high rewards of 4 and 7 take increasingly more portions over the total reward. To the rightmost, the even distribution of reward 1, 4, and 7 reflects that the agent can hit roughly the same number of bricks from different rows, indicating a good performance of the agent.

All charts in this view are coordinated. When users hover one chart, a gray dashed line will show up in the current chart, as well as other charts, and a pop-up tooltip in each individual chart will show the corresponding information, as shown in Figure 1a (the mouse is in the stacked area chart for the reward distribution). Meanwhile, the hovering event will trigger the update in the *Epoch* view (presented next).

6.2 *Epoch* View: Epoch-Level Overview

The *Epoch* view presents the summary statistics of the selected epoch with a combined visualization of a pie chart and a stacked bar chart, as shown in Figure 1b (R1.2, R2.1). The pie chart shows the action/reward distribution of all steps in the current epoch; whereas the stacked bar chart presents the action/reward distribution of each individual episode in the epoch. As shown in Figure 1-b2, there are 20 episodes in the current epoch (one stacked bar for one episode), and the stacked bars are sorted decreasingly from left to right to help users quickly identify the episode with the maximum number of steps/rewards.

The two types of charts are linked with each other via user interactions. For example, when hovering over the white sector of the pie chart (representing *noop* actions), the *noop* portion of all stacked bars will be highlighted, as the area of the sector is the sum of all white regions from the stacked bars. By default, the distributions of actions and rewards are presented in this view, as these two are of the most interest. But, users can also plug in other variables (e.g., the step distribution in different lives of the agent) by modifying the configuration file of *DQNViz*.

All views of *DQNViz* share the same color mapping. For example, the red color always represents the *fire* action; and the purple color always indicates 4-point reward. Therefore, the pie charts (with text annotation in different sectors) also serve as the legends for *DQNViz*.

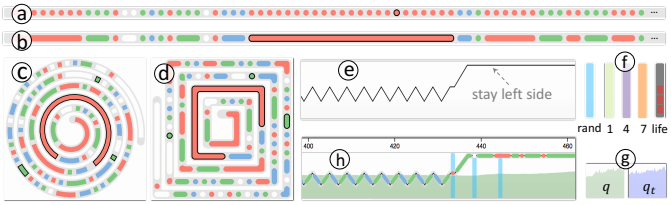

Fig. 4. Different designs for the event sequence data from DQN.

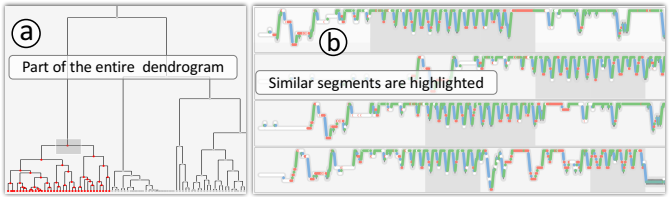

Fig. 6. Clustering segments (segment length is 100) in epoch 120.

6.3 *Trajectory* View: Episode-Level Exploration

The *Trajectory* view aims to provide an overview of all steps in one epoch, and reveal the action/reward patterns in all episodes of the epoch.

6.3.1 Visual Design for DQN Event Sequence Data

The data collected from one episode is essentially an event sequence (also called as a *trajectory* by experts [12]). We, therefore, start the design of this view with event sequence visualization solutions, and several key design iterations are briefly discussed as follows.

Our first design presents one episode with a line of circles representing different types of actions, as shown in Figure 4a. The color white, red, blue, and green represent *noop*, *fire*, *right*, and *left* respectively. The circles with the black stroke represent the actions with a reward. Many previous works (e.g. [16, 54]) have adopted this type of design, as it is straightforward and easy to understand. However, the design cannot reflect the agent's movement patterns and it is also not scalable. To overcome these limitations, we tried to merge consecutive circles representing the same actions as one line (Figure 4b), which can effectively reveal the repeat of different actions. We also explored the spiral layout to address the scalability issue (Figure 4c, 4d). The spiral layout can present one entire episode on the screen, but compactly arranging all episodes (in one epoch) with varying lengths becomes a problem.

To reveal the movement patterns of the agent (R2.2), we visualize the displacement of the paddle to the right boundary over time (Figure 4e). The design is based on our observation that the moving behavior of the agent is visually reflected by the position of the paddle. For example, the oscillation in the first half of Figure 4e indicates that the agent keeps switching between *left* and *right* to adjust the position of the paddle. Also, this design is scalable. It allows us to flexibly compress the curve horizontally, like a spring, to get an entire view of a long episode (R2.1). One limitation is that the paddle positions, though reflecting the agent's movement patterns, cannot accurately reflect the action taken by the agent. For example, in the right half of Figure 4e, the paddle stays at the leftmost position (the top side). The action that the agent is taking now can be *noop*, *fire* (*fire* does not change the paddle position), or *left* (the paddle is blocked by the left boundary and cannot go further). To address this issue, we overlay the action circles/lines onto the curve. From the visualization (Figure 4h), we found that the agent takes three types of actions (i.e., *left*, *noop*, and *fire*) in the right half of Figure 4e.

Lastly, this design can synchronize other types of data with the action data (R2.3). As shown in Figure 4h, some actions are highlighted with background bars in cyan, indicating they are random actions. Bars with color light green, purple, and orange encode the reward of 1, 4, and 7 respectively (Figure 4f, 1-c5). We also design glyphs for actions with a life loss, as shown in the rightmost of Figure 4f. This glyph is a gray bar with 0∼4 dark red rectangles inside, indicating the number of remaining lives after the life loss action. The *terminal* information is also encoded in this glyph (gray bars with 0 dark red rectangle). The q and q_t values are presented as transparent area charts with green and blue color respectively in the background (Figure 4g, 4h). When users click the action circles/lines, a video clip (Figure 5) will pop up and show the *screen* data (a sequence of screens). The two vertical yellow bars on the two sides of the video are the progress bars. We found they are very useful in reflecting the progress of static videos, e.g., when the agent is repeating the *noop* action. The color bar on the bottom shows the predicted action. It changes from frame to frame when the video is playing. To avoid visual clutters in the *Trajectory* view, users can show/hide the eight types of data on demand through a set of check-box widgets (Figure 1-c2).

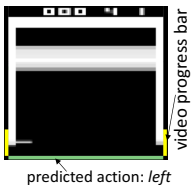

Fig. 5. A video clip.

6.3.2 Segment Clustering and Pattern Mining

It is not a trivial task to visually identify the common patterns from the large number of actions in one epoch. To address this issue, we adopted two techniques of segment clustering (R2.1) and pattern mining (R2.2). To cluster segments, we first cut the episodes in one epoch into many smaller segments and cluster them using hierarchical clustering [18]. A segment is expressed by a sequence of values, indicating the paddle positions. The segment length is set to 100 by default, but it can be adjusted on-demand from the configuration file of *DQNViz*. To better quantify the similarity between segments, we used the dynamic time warping algorithm (DTW) [40]. Specifically, for a pair of segments (i.e. two temporal sequences), the DTW algorithm can find the best temporal alignment between them and derive a more comprehensive similarity score. Applying DTW to all pairs of segments will derive a similarity matrix for all segments in one epoch, which is the input of the clustering algorithm. When clicking the "Tree" button in Figure 1-c1, a dendrogram visualization of the clustering results will show up (Figure 6a). Selecting different branches of the dendrogram will highlight different clusters of segments in the *Trajectory* view (Figure 6b).

We found some typical movement patterns of the agent while exploring the clustering results. By defining those patterns and mining them in other epochs, we can provide more insight into the agent's behaviors. The regular expression [2, 48] is used to define a pattern as it is simple and flexible. For example, an action sequence can be expressed as a string (of 0, 1, 2, and 3) and a predefined regular expression can be used to search on the string to find when and where a particular pattern happens. Table 1 presents two example movement patterns, i.e., *repeating* and *hesitating* (frequently switching between *left* and *right*).

Table 1. Formalizing action/reward patterns with regular expressions.

Pattern	Regular Exp.	Explanation
repeating	`0{30,}`	repeating *noop* (0) for at least 30 times.
hesitating	`(20*30*){5,}`	switching *left* (2) and *right* (3) for at least 5 times. There might be multiple *noop* actions between the *left* and *right*.
digging	`10+10+40+ 40+70+70+`	the two 1s, 4s, and 7s are where the ball hits the bottom, middle and top two rows of bricks, the 0s in between are the round trip of the ball between the paddle and bricks.
bouncing	`(70+){5,}`	hitting top 2 rows for at least 5 times.

Reward patterns can be defined similarly. For example, we found that the agent becomes very smart in later training stages, and it always tries to dig a tunnel through the bricks, so that the ball can bounce between the top boundary and the top two rows to achieve 7-point rewards. The *digging* and *bouncing* pattern can be defined using regular expressions, as shown in Table 1. We can also visually verify the patterns from the bars in the *Trajectory* view (Figure 1-c5). It is worth mentioning that the regular expression for each pattern can be relaxed. For example, the *digging* pattern in Table 1 can be relaxed to `10+40+40+70+`.

Tracking patterns is an effective way to provide insight into the evolution of the agent's behaviors. For example, the decreasing of *repeating* indicates the agent became more flexible in switching among actions. The increasing of *digging* reflects the agent obtained the trick of digging tunnels. The number of a pattern can be defined as a metric (R1.3) and plugged into the *Statistics* view for overview (Figure 1-a1).

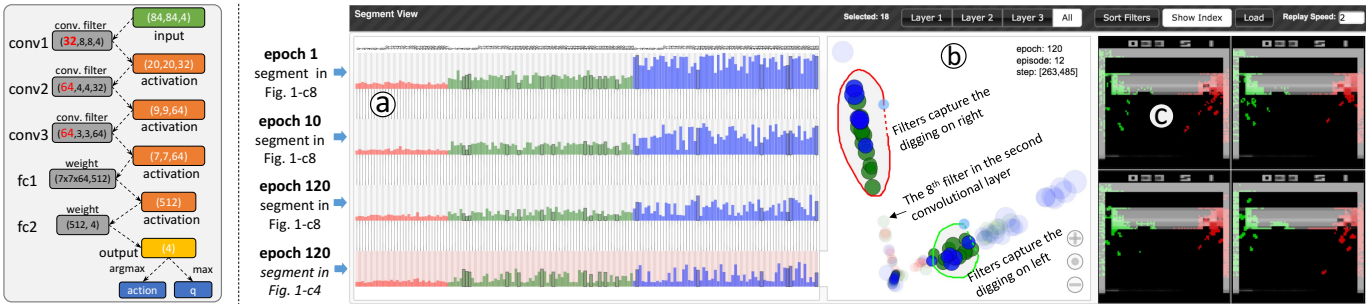

Fig. 7. Left: the DQN structure; right: *Segment* View: (a) bar charts view, (b) PCA view with lasso selections, (c) average screens with saliency maps.

6.4 *Segment* View: Segment-Level Interpretation

6.4.1 See through the agent's eyes

The *Segment* view targets to reveal what the agent really sees and how it gains such vision in the screen states from a trajectory segment. To achieve this, we dive into the DQN structure [33] (Figure 7, left) and reveal what have been extracted by the neural network (R3.1). The input of the network is a state of size 84×84×4 (in green), and the output is a vector of four values (in yellow) representing the predicted rewards for the four actions. Among the four values, the maximum one is the predicted q, and its index is the predicted action. Between the input and output are 3 convolutional and 2 fully connected layers.

The filters in the convolutional layers are the basic computational units that extract features from the input states. We focus on them to interpret what the agent sees. The four numbers in each gray rectangle of Figure 7 (left) represent *the number of filters*, *the width and height of each filter*, and *the number of channels*. There are *32*, *64*, and *64* filters in the first, second, and third convolutional layer (160 in total).

Algorithm 1 Picking out the maximumly activated state (`max_state`) from a segment (`screens`), and generating the corresponding saliency map (`map`) of the state for each convolutional filter in each layer.

```
1:  screens = [s_1, s_2, ..., s_n] // input: a segment of n screens
2:  states = [{s_1, s_2, s_3, s_4}, ..., {s_{n-3}, s_{n-2}, s_{n-1}, s_n}] // n-3 states
3:  for i = 0; i < layers.length; i++ do
4:      for j = 0; j < layers[i].filters.length; j++ do
5:          activations = DQN.f_prop(layers[i].filters[j], states)
6:          max_idx = argmax(activations)
7:          max_state = states[max_idx]
8:          max_activation = activations[max_idx]
9:          map = DQN.b_prop(layers[i].filters[j], max_activation)
10:         output[i][j] = blend(max_state, map) // algorithm output
11:     end for
12: end for
```

Algorithm 1 shows how we visualize what features each filter extracted from an input state. The algorithm includes three main steps. **First**, given a segment, we find the state that is maximumly activated by each of the 160 filters. Specifically, for each filter in each layer, we first apply forward propagations on all the input states of the segment (Algorithm 1, *line 5*) to get the state (`max_state` in *line 7*) that is maximumly activated by the filter. **Then**, using the activation of this state (`max_activation` in *line 8*), we perform guided back-propagations to generate a saliency map (`map` in *line 9*) for the state. The saliency map will have the same size with the input state (i.e., 84×84×4), and the pixel values in the map indicate how strong the corresponding pixels of the input state have been activated by this filter (the back-propagation computes the gradient of the maximum activation on the input state, details can be found in [46]). **Finally**, we blend the input state with its corresponding saliency map (`blend` in *line 10*). The blending image can expose which region of the input state has been seen by the current filter (like an eye of the agent). For example, Figure 9a shows the blending result of the second screen of a state with its corresponding saliency map. We can see that the filter extracts the ball from the screen.

6.4.2 Analysis components with the agent's eyes

The *Segment* view enables users to analyze the 160 filters along with the 160 states they have maximumly activated in three sub-views (Figure 7, right): a parallel bar charts view, a principal component analysis (PCA) view, and a view showing the average state of the input segment.

The bar charts view (Figure 7a) shows the size of features that individual convolutional filters extracted from the input states. Each row is a bar chart representing one segment (four rows are in the view in Figure 7a). Each bar in each row represents a filter from the DQN, and the height of the bar indicates the size of the feature that the filter extracted (i.e., the number of activated pixels in the corresponding saliency map, see Algorithm 1). The color red, green, and blue indicate the filter is from the first, second, and third layer respectively. Different rows represent the filters for different source segments selected by users, and the corresponding filters are linked together across rows for comparisons (R3.2, R3.3). Clicking the "Sort Filters" button in the header of this view will sort the bars based on their height. Users can focus on filters in layer 1, 2, 3 or all of them for analysis by interacting with the widgets in the header (currently all filters are in analysis). The row with the pink background (the 4th row) is the segment in selection and currently analyzed in the other two sub-views. Clicking on different rows will update contents of the other two sub-views.

The PCA view (Figure 7b) presents how the filters can capture similar or dissimilar features from the screen states (R3.1). It projects the 160 convolutional filters of the selected row based on their saliency map, i.e., reducing the 84×84×4 dimensional saliency maps to 2D using PCA. Each circle in this view represents one filter, and the color red, green, and blue indicate the filter is from the first, second, and third layer. The size of circles encodes the size of features extracted by the filters. Moreover, the circles in this view are coordinated with the bars of the selected row in the *bar charts view*. Clicking any bars/circles will pop up a four-frame video showing the blending result of the input state and the corresponding saliency map. Figure 9a shows the second frame of the pop-up video when clicking the 8th filter in the second layer (position indicated in Figure 7b). Semantic zoom (the user interfaces in the bottom right of this view) is also enabled to reduce visual clutters. This interaction can enlarge the view while maintaining the size of circles, which will mitigate the overlap of the circles by increasing the distances among them.

The four screens (Figure 7c) show the aggregated results of states from a selected segment. For example, the top-left screen is the result of averaging the first screen from all input states of the selected segment. We also introduce some interactions to help users easily observe which part of the screen is seen by the filters. For example, when users select different convolutional filters from the *bar charts view* (via brushing) or from the *PCA view* (via lasso selection), the union of the corresponding saliency maps will be highlighted on the aggregated screens. In Figure 7c, the two selected clusters of filters (Figure 7b, circles in the green and red lasso) capture the features that the ball is digging the left and right corner of the bricks respectively (R3.1).

7 Case Studies

We worked with the same three domain experts (E1, E2, and E3) in the design stage of *DQNViz* on several case studies. Here, we present two

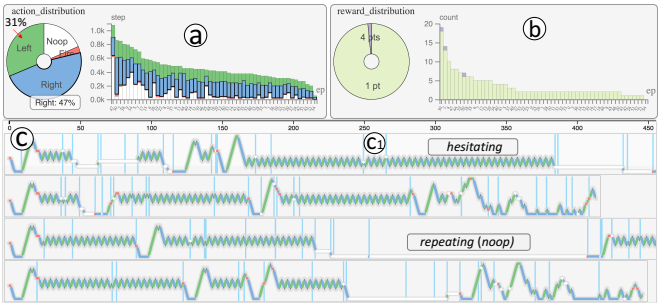

Fig. 8. The *Epoch* (a, b) and *Trajectory* (c) view of Epoch 37.

of them: one emphasizes how *DQNViz* can help the experts understand the DQN training process; the other shows how the experts use *DQNViz* to diagnose and improve the exploration rate of the DQN model.

7.1 Unveiling the model training evolution

Overall training statistics. The experts all confirmed that the training was successful based on observations from the *Statistics* view in Figure 1a. First, both `average_reward` and `average_q` value are increasing, indicating the agent expected and was able to receive higher and higher rewards. Meanwhile, the decreasing number of games (`nr_games`) shows the agent could survive longer in individual episodes (R1.1). Second, the experts observed that the agent became more intelligent and strategic to receive higher points as the training evolves (R1.3). For example, in Figure 1-a1, the agent adopted more `bouncing` patterns that can collect high points along with the training. Also, the reward distribution in Figure 1-a2 echoes the same observation as the agent obtained an increasing amount of 4 and 7 points over time (R1.2).

One interesting observation pointed by one expert is that the action distribution is very diverse even in the later training stages (R1.2). However, the agent can still achieve high rewards with diverse action distributions. This indicates that by merely looking at the action distribution, it is difficult to understand why and how the agent achieved high rewards. It also calls for the investigation of detailed action patterns, which is conducted using the *Trajectory* view later on by the experts.

The experts also spotted some abnormal epochs. For example, the reward distribution in epoch 37 did not follow the general trend in the stacked area chart. This observation led the experts to select epoch 37 and drill down to explore its details in the *Epoch* view.

Epoch statistics. From the statistics shown in the *Epoch* view, the experts suspected that the agent repetitively moved the paddle left and right in epoch 37, but those moves were mostly useless. As shown in Figure 8a, the *left* and *right* actions take 31% and 47% of the total 25,000 steps in this epoch (R1.2). Meanwhile, it can be imagined that the agent mostly hit bricks in the bottom two rows as the achieved rewards were mostly 1-point reward (Figure 8b). From the stacked bar charts in Figure 8a and 8b, the experts observed that most episodes in this epoch lasted for less than 0.6k steps (much shorter than normal episodes) and achieved less than 5 points. Therefore, the large number of *left* and *right* did not contribute much to a better performance (R2.1).

Action/Reward patterns in episodes. Drilling down to the agent's movement patterns in the *Trajectory* view, the experts confirmed that the large number of *left* and *right* actions in epoch 37 were mostly useless. As shown in Figure 8c, the agent repeated a lot of *hesitating* and *repeating* patterns, which contributed nothing to achieving rewards. By highlighting the random actions (cyan bars in Figure 8c), the experts also learned that random actions play an important role in terminating the *hesitating* and *repeating* patterns. Moreover, terminating those patterns may need multiple random actions (e.g., the two random actions in Figure 8-c1 are not sufficient to terminate the *hesitating* pattern).

The experts were also interested to know the agent's behaviors in normal epochs. Among the many epochs whose statistics follow the general trend, the experts randomly selected epoch 120 to explore. A few *hesitating* and *repeating* patterns can still be found in this epoch as shown in Figure 1-c3 and 1-c7. Moreover, many *digging* and *bouncing* patterns were found in this normal epoch. From the exploration of

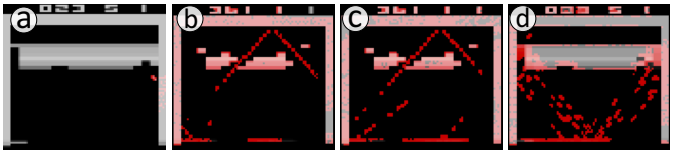

Fig. 9. (a) Blending a state with its saliency map. (b,c,d) what the agent sees from the segment in the 1st, 2nd, and 4th row of Figure 7a.

the hierarchical clustering result (Figure 6), the experts found many similar movement patterns shown in Figure 1-c4 and 1-c6. By zooming into Figure 1-c4, visualizing the reward data, and replaying the video clips, the experts found that the agent was digging a tunnel through the bricks and a *bouncing* pattern appeared right after the *digging* pattern, as shown in Figure 1-c5 (R2.2, R2.3). In the digging pattern, the agent periodically moved the paddle to catch the ball and received 1-point, 4-point, and 7-point rewards with regular step intervals. In the successive *bouncing* pattern, the agent received 7-point rewards very frequently.

It is also obvious that the q value kept increasing in *digging*, but immediately started decreasing after the tunnel was created. The experts interpreted this observation as follows. During digging, the agent can see the progress of the tunnel (from input states), and the expected reward keeps increasing as the tunnel will potentially result in *bouncing* (i.e., keep getting high rewards). However, when the *bouncing* starts, the bricks (especially in the top two rows) keep being destroyed, therefore, the expected future reward starts decreasing. The experts also found that when *bouncing* happens, the paddle mostly stays at the leftmost position, which is *"quite phenomenal"*. This deepens their understanding on the successful playing strategies of the agent and they would like to conduct further theoretical investigations on this.

Segment-level investigations. Intrigued by *digging* and *bouncing* patterns, the experts were curious about how the agent can observe states and make the action predictions. This motivated them to select some segments of interest and analyze them in the *Segment* view (R3.1). From there, they found that filters from higher convolutional layers usually have stronger activations and capture larger and more diverse features from the input states. For example, to further investigate the *digging* pattern in Figure 1-c4, the experts selected that segment (step 263∼485 of episode 12 in epoch 120) into the *Segment* view (the fourth row in Figure 7a). By exploring it, they found that: (1) the height of bars representing filters from layer 1 (red), 2 (green), and 3 (blue) shows an increasing trend (Figure 7a); (2) the circles representing filters from three layers show an inner-to-outer layout in the PCA view (Figure 7b).

By examining the *digging* segment, the experts also learned that the agent dug tunnels from both sides of the bricks and they identified what filters captured the *digging* on different sides. Specifically, the two groups of filters in the green and red lasso of Figure 7b are filters that captured the *digging* on the left and right of the scene (Figure 7c). From the aggregated saliency map of all filters in Figure 9d, it can be seen that the agent moves the paddle between the left boundary and the middle of the scene to catch the ball and dig tunnels from both sides.

Compare segments from the same epoch (R3.2). To have more understandings on the functionality of different filters, the experts also compared the above *digging* segment with other segments in this epoch. For example, one expert selected another segment shown in Figure 1-c8, where the agent moved the paddle all the way to the left then to the right. Figure 1d shows the fourth average screen and the aggregated saliency map from all filters for this segment. It is clear that the agent tracked the moving path of the ball, i.e., A-B-C-D-E-F, and it moved the paddle all the way to the left then to the right to catch the ball in B and F. By comparing the filters in these two segments, the experts found that certain filters behave similarly, e.g., filter 16 from the third convolutional layer traces the ball in both segments. However, some filters also have dramatically different functions, e.g., filter 23 in the second layer stares at the top-left tunnel in the *digging* segment; while it traces the ball in the other segment. This result provides evidence that filters in the same stage may not always have the same functions.

Track a segment over time (R3.3). The experts also wanted to understand how filters evolved over time by comparing the same segment

across epochs. As shown in the top three rows in Figure 7a, one expert tracked the segment in Figure 1-c8 in epoch 1, 10 and 120. The blended saliency maps are shown in Figure 9b, 9c, and 1d (the saliency maps from all filters on the fourth average screen). From them, the expert understood that the agent did not have a clear vision on the input states in early stages (Figure 9b), and it gradually developed its attention on important parts, e.g. the moving path of the ball (Figure 9c, 1d).

7.2 Optimizing the Exploration Rate (Random Actions)

One expert was very interested in the random actions, especially after he saw that random actions can terminate bad movement patterns in Figure 8c. Here, we describe the experiments that we worked with him to diagnose and improve the use of random actions using *DQNViz*.

Experiment 1: No random action. The expert first hypothesized that random actions are not necessary after the model is well trained. The logic behind this is that an action predicted by a well-trained agent should be better than a randomly generated one. To test this, the expert set the exploration rate (ε in Equation 3) to 0, after 200 training epochs, and used the agent to play the Breakout for 25,000 steps (a testing epoch) to see if anything will go wrong (by default ε=0.05 in testing).

The result of the 25,000 steps in the *Trajectory* view is shown in Figure 10a. The expert had two observations: (1) there is only one episode in the 25,000 steps, and the episode is very long; (2) the agent keeps repeating the *noop* action in roughly 60% of the episode.

In detail, the single and long episode can roughly be cut into three phases as labeled in Figure 10a. In phase **I** (Figure 10-a1), the agent played very well in the first $\sim$1,080 steps, and this phase ended with a life loss. In phase **II**, the agent kept repeating *noop* for $\sim$15,000 steps (i.e., *trapped* by the environment). By looking at the *screen* data (Figure 10c) at the position indicated in Figure 10-a2, the expert understood that the paddle stays around the middle of the scene and the ball is not in the scene (as there is no *fire* action). This indicates that the agent does not know that he needs to *fire* the ball at the beginning of a game, but just keeps waiting for the ball. The expert was very surprised about the agent's movement pattern in phase **III**, i.e., how did the agent get out of the trap without the help of random actions? After checking the screen data at the position indicated by Figure 10-a3, he realized that the game has crashed actually, as the numbers for reward and life disappear and the entire scene becomes lighter (Figure 10d).

Experiment 2: Random actions on demand. From Experiment 1, the expert learned that random actions are necessary. Next, he was wondering if one can control the initiation of random actions when they are really needed. The expert hypothesized that random actions are needed when the agent keeps repeating the same patterns but gets no reward (e.g. *hesitating* in Figure 8c, and *noop* in Figure 10-a2).

Experiment 2 tested this hypothesis with a pattern detection (PD) algorithm, which can be explained as follows. First, a buffer that stores the latest 20 steps is maintained. At each step of the game, if the agent received rewards in the latest 20 steps, no random action is needed. However, if the agent did not receive any reward, but kept repeating the same action/pattern in those steps (detected using regular expressions, see Section 6.3.2), a random action would be introduced. As observed before, a repeating pattern usually has a basic repeating unit which is very short, e.g., the basic unit of the *hesitating* pattern in Figure 4h is *left-left-right-right* and the pattern length is 4. Experiment 2 checks pattern length from 2$\sim$7, and a random action is introduced if a pattern has been repeated for 3 times. For example, if the latest three actions are `230` (*right-left-noop*) and this pattern can be found 3 times in the latest 9 steps, then the next action will be a random action.

Figure 10b shows the result of applying the PD algorithm to DQN. Similarly, only one episode is generated in the 25,000 steps, and the episode can be cut into three similar phases. The expert first noticed that the PD algorithm worked well in terminating the *repeating* of one action in phase **I** (before $\sim$1,800 steps). For example, in Figure 10-b1, the *noop* action has been repeated for 20 times, and the agent introduced several random actions and got out of the repeating of *noop* finally.

However, the expert also observed that the agent was trapped by the environment again in phase **II**. After zooming into this phase, the expert found that the agent kept repeating a long pattern with the length of around 50 steps (Figure 10-b2). By replaying the game, as shown in Figure 10e, he realized that the agent kept moving the paddle between point A and D to catch the ball, and the ball repeated the loop A-B-C-D-C-B-A. No random action was introduced, as the length of the repeating pattern exceeded the threshold (i.e., 7) in the PD algorithm.

The game crashed again in phase **III**. However, by exploring different segments in this phase, we could still see that the PD algorithm worked well in introducing random actions (e.g., Figure 10-b3, b4, b5).

Experiment 3: Improved random actions. With the lessons learned in Experiment 2, the expert applied the following changes to the PD algorithm: (1) changing the maximum pattern length from 7 to 50; (2) increasing the buffer size from 20 to 100; (3) introducing a random action if a pattern repeats twice. With these changes, the agent was able to play the game very well and no longer trapped by the environment. In 25,000 steps, the agent played 12 episodes and received 5,223 total rewards. The number of random actions in those steps is 501, which is much less than 1,250 (i.e., 5% of 25,000 in the original setting).

Table 2. Statistics of random actions per epoch (averaged over 10 runs).

	steps	episodes	total rewards	random actions
ε=0.05 (5%)	25,000	16.6	4198.6	1269.4
PD Algorithm	25,000	11.4	4899.2	503
ε=0.02 (2%)	25,000	9.9	3780.8	492.1

To quantitatively evaluate this improvement, the expert compared the PD algorithm with other two baselines using ε=0.05 and ε=0.02, as shown in Table 2 (results are averaged over 10 runs). As we can see, the PD algorithm worked better than the other two random-only methods. Compared to the method of ε=0.05, the PD algorithm introduced less random actions, but achieved $\sim$700 more rewards in 25,000 steps. Also, it led to fewer life losses, as the number of episodes is less than the baseline using ε=0.05. Compared to the method of ε=0.02, the PD algorithm obtained much higher total rewards in 25,000 steps, though the number of random actions was similar. This comparison further verifies that the PD algorithm can effectively control random actions.

7.3 Feedback from Domain Experts

We collected the experts' feedback via in-depth interviews after the case study sessions. Overall, all experts believed that the tool is *"extremely helpful to have actionable insight into the (model) evolution"*, and *"has great potentials to comprehend and improve deep RL models"*.

In terms of aiding their understanding of the model, all experts agreed that the overall training statistics are the basic need as they *"use them in their daily model building practice"*. The overall trend and distribution of DQN-specific metrics (e.g., the number of *bouncing* pattern) really *"provide a glimpse into the evolution details"* and *"would jump-start their diagnosing towards the hurdle of model training"*.

All experts believed the most useful component is the *Trajectory* view, where they *"could explore, observe, and extract meaningful patterns"* and *"ponder how the agent developed its playing strategies"*. All experts spent the most time on this view to explore useful patterns that could help them comprehend and potentially improve the model. *"It is also quite entertaining to look at the video play-back. This level info is absent in other tools."*, commented by E3. Another example is the three experiments on random actions. Towards the end of that study, E1 concluded that *"the visualization had significantly improved his understanding about random actions"*. He explained that the final choice of *"the (pattern) length to be 50 is not random"*. In fact, that value should be (or close to be) the upper bound for the number of steps that the ball needed for a round trip (between the paddle and bricks).

The experts expressed that the *Segment* view is *"very fun to play with"* and appreciated this view *"provides many details about how the agent parses the game screens"*. E1 and E2 both made a connection between this view and the *"attention mechanism"* [29,45] by commenting *"it is very enlightening to know layers have attentions over different parts of the screens"*. E1 even proposed a hypothesis that *"filter activation in a complicated network may correspond to different playing strategies"*.

Additionally, the experts also mentioned several desirable features and suggested some improvements. E2 was first confused with the

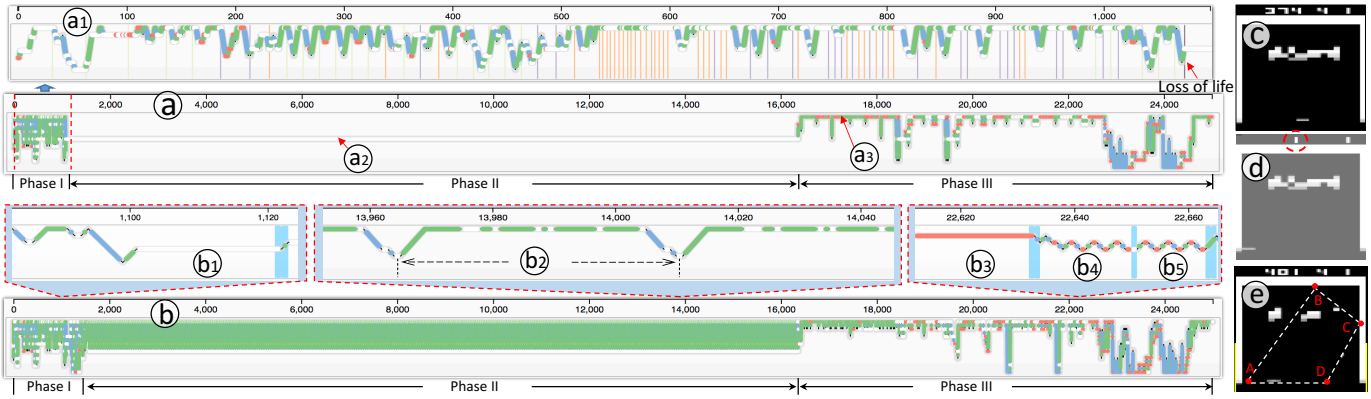

Fig. 10. (a, b) Result of Experiment 1, 2; (c, d) game screen at a2, a3; (e) ball trajectory for segment b2, the ball follows the loop A-B-C-D-C-B-A.

visual encoding of paddle positions in the *Trajectory* view, though later on she understood it with some explanations. She felt *"(there is) a disconnection between the horizontal location in the game interface and the vertical position encoding"*, and *"some labels (at the vertical axis of the Trajectory view) may be helpful"*. For the *Segment* view, two experts thought *"there is a high learning curve"*, and *"it is especially true for the layout of filters"*. *"Showing network structures and highlighting where those filters come from may help"*, suggested by E1. The three experts also shared their concerns on the generalization of *DQNViz*. E3 pondered *"how this tool can visualize games with a larger action space"* or *"even real games, like AlphaGO [43]"*. These comments are very useful in guiding our further improvements of *DQNViz*.

8 Discussion, Limitations and Future Work

Dimensionality Reduction. The *Segment* view uses a dimensionality reduction algorithm to project the high-dimensional saliency maps (resulted from individual filters) to 2D for visualization and interaction. The use of the PCA algorithm came out in one of our discussions with the domain experts and it turns out that the algorithm is simple, sufficient for our objective, and involves few parameters to tune with. However, it also suffers from many drawbacks as shown in previous works [6, 7, 21], and we do believe that other dimensionality reduction algorithms can be used here as alternative solutions. Most notably, we have tested the t-SNE algorithm [30] and demonstrated some results of it with different parameter settings in our supplementary material. Both algorithms lay out filters of higher layers in outer locations, indicating higher layer filters usually capture more diverse features.

Generalization. *DQNViz* can be applied to several other Atari games that involve simple movements, e.g., Pong and Space-Invader. However, as a preliminary prototype, it is not readily applicable to all Atari games, especially the ones with sophisticated scenes and large action spaces, like Montezuma's Revenge. This limitation motivates us to generalize *DQNViz* from several directions in the future. The *first* direction is to adapt the *Trajectory* view to games with a larger action space. Currently, this view can only capture movement patterns in 1D (i.e., moving horizontally or vertically). Enabling the view to capture 2D movement patterns is our first planned extension. In detail, we can divide the possible actions into three categories: horizontal movement actions, vertical movement actions, and other actions. Multiple *Trajectory* views can be used to visualize actions in different categories, and coordinated interactions can be used to connect those views to retrieve screen states and observe action patterns. *Second*, we believe visual analytics can help the diagnoses and improvements of DQN far more than optimizing the exploration rate. Our next attempt targets on prioritizing the agent's experiences [41] through visualization to accelerate the training. *Lastly*, we also want to explore if any components of *DQNViz* or the designs in our work can be reused to analyze other RL models. For example, the four-level exploration framework may be directly applicable to other RL models, though the details in each level will have many differences. The idea of looking from the agent's eyes through guided back-propagations could also be reused in other deep RL models.

Scalability. One potential challenge with *DQNViz* is its scalability, including large parameter settings, games with a very long training process, and so on. As to address the large parameter settings, some components of *DQNViz* should be improved to accommodate them. For example, the *Trajectory* view currently presents all actions in one epoch containing 25,000 steps. However, if the number of steps in an epoch is very large, the *Trajectory* view will have to aggregate and smooth those steps, and use semantic zoom to help users explore the sequence patterns. Similarly, the scalability problem may also occur when extending *DQNViz* to other games with a very long training process. For example, in certain games, the length of individual game episodes may be too long to be presented in the *Trajectory* view. In those cases, we can first cut a long episode into many very short segments and use the most frequently appeared action in each segment to represent the segment (i.e. binning and voting). In short, intelligent data aggregations, effective uses of the visualization space, and friendly user interactions always deserve more considerations in addressing the scalability problem.

More Future Works. We are also interested in exploiting the power of regular expressions in pattern mining, and more user-defined regular expressions may be used to explore the agent's experiences. It may also be possible to extract certain patterns using automatic data mining algorithms. Additionally, our system analyzes DQN models off-line currently (i.e., after training). Enabling domain experts to directly interact with the model training process (i.e., guiding the agent to learn specific behaviors during training) is an important and interesting direction for us to investigate in the future. Lastly, the current version of *DQNViz* targets to serve domain experts with certain knowledge on DQN models. Simplifying the complex interface and generalizing the domain-specific components of *DQNViz* to extend the tool to common users are also potential research directions that worth to be explored.

9 Conclusion

In this work, we present *DQNViz*, a visual analytics system that helps to understand, diagnose, and potentially improve DQN models. The system reveals the large experience space of a DQN agent with four levels of details: overall training level, epoch-level, episode-level, and segment-level. From our thorough studies on the agent's experiences, we have identified typical action/movement/reward patterns of the agent, and those patterns have helped in controlling the random actions of the DQN. The insightful findings we demonstrated, the improvements we were able to achieve, and the positive feedback from deep learning experts validate the effectiveness and usefulness of *DQNViz*.

Acknowledgments

The authors would like to thank Wei Zhang, Yan Zheng, and Dean Galland from Visa Research for their insightful comments, valuable feedback, and great helps. This work was supported in part by US Department of Energy Los Alamos National Laboratory contract 47145, UT-Battelle LLC contract 4000159447, NSF grants IIS-1250752, IIS-1065025, and US Department of Energy grants DE-SC0007444, DE-DC0012495, program manager Lucy Nowell.

REFERENCES

[1] Github simple dqn. https://github.com/tambetm/simple_dqn. Accessed: 2018-02-08.

[2] A. V. Aho and J. D. Ullman. *Foundations of Computer Science (Chapter 10: Patterns, Automata, and Regular Expressions)*. Computer Science Press, Inc., New York, NY, USA, 1992.

[3] O. Anschel, N. Baram, and N. Shimkin. Deep reinforcement learning with averaged target DQN. *CoRR*, abs/1611.01929, 2016.

[4] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. A brief survey of deep reinforcement learning. *arXiv preprint, http://arxiv.org/abs/1708.05866*, 2017.

[5] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[6] M. Aupetit. Visualizing distortions and recovering topology in continuous projection techniques. *Neurocomputing*, 70(7-9):1304–1330, 2007.

[7] M. Aupetit, N. Heulot, and J.-D. Fekete. A multidimensional brush for scatterplot data analytics. In *Visual Analytics Science and Technology (VAST), 2014 IEEE Conference on*, pp. 221–222. IEEE, 2014.

[8] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.(JAIR)*, 47:253–279, 2013.

[9] R. E. Bellman. *Dynamic Programming*. Dover Publications, Incorporated, 2003.

[10] A. Bilal, A. Jourabloo, M. Ye, X. Liu, and L. Ren. Do convolutional neural networks learn class hierarchy? *IEEE transactions on visualization and computer graphics*, 24(1):152–162, 2018.

[11] J. Choo and S. Liu. Visual analytics for explainable deep learning. *IEEE Computer Graphics and Applications*, 38(4):84–92, Jul 2018. doi: 10.1109/MCG.2018.042731661

[12] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems*, pp. 4302–4310, 2017.

[13] V. François-Lavet, R. Fonteneau, and D. Ernst. How to discount deep reinforcement learning: Towards new dynamic strategies. *NIPS Deep Reinforcement Learning Workshop, arXiv preprint arXiv:1512.02011*, 2015.

[14] B. Goertzel and C. Pennachin. *Artificial general intelligence*, vol. 2. Springer, 2007.

[15] D. Gotz and H. Stavropoulos. Decisionflow: Visual analytics for high-dimensional temporal event sequence data. *IEEE transactions on visualization and computer graphics*, 20(12):1783–1792, 2014.

[16] S. Guo, K. Xu, R. Zhao, D. Gotz, H. Zha, and N. Cao. Eventthread: Visual summarization and stage analysis of event sequence data. *IEEE transactions on visualization and computer graphics*, 24(1):56–65, 2018.

[17] H. V. Hasselt. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, eds., *Advances in Neural Information Processing Systems 23*, pp. 2613–2621. 2010.

[18] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference and prediction (second edition)*. Springer, 2009.

[19] F. S. He, Y. Liu, A. G. Schwing, and J. Peng. Learning to play in a day: Faster deep reinforcement learning by optimality tightening. *CoRR*, abs/1611.01606, 2016.

[20] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*, 2017.

[21] N. Heulot, J.-D. Fekete, and M. Aupetit. Proxilens: Interactive exploration of high-dimensional data using projections. In *VAMP: EuroVis Workshop on Visual Analytics using Multidimensional Projections*. The Eurographics Association, 2013.

[22] M. Kahng, P. Y. Andrews, A. Kalro, and D. H. P. Chau. Activis: Visual exploration of industry-scale deep neural network models. *IEEE transactions on visualization and computer graphics*, 24(1):88–97, 2018.

[23] J. Kober and J. R. Peters. Policy search for motor primitives in robotics. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, eds., *Advances in Neural Information Processing Systems 21*, pp. 849–856. 2009.

[24] V. R. Konda and J. N. Tsitsiklis. On actor-critic algorithms. *SIAM J. Control Optim.*, 42(4):1143–1166, 2003.

[25] M. Krstajic, E. Bertini, and D. Keim. Cloudlines: Compact display of event episodes in multiple time-series. *IEEE transactions on visualization and computer graphics*, 17(12):2432–2439, 2011.

[26] M. Liu, J. Shi, K. Cao, J. Zhu, and S. Liu. Analyzing the training processes of deep generative models. *IEEE transactions on visualization and computer graphics*, 24(1):77–87, 2018.

[27] M. Liu, J. Shi, Z. Li, C. Li, J. Zhu, and S. Liu. Towards better analysis of deep convolutional neural networks. *IEEE transactions on visualization and computer graphics*, 23(1):91–100, 2017.

[28] S. Liu, X. Wang, M. Liu, and J. Zhu. Towards better analysis of machine learning models: A visual analytics perspective. *Visual Informatics*, 1(1):48–56, 2017.

[29] T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. In *EMNLP*, pp. 1412–1421. The Association for Computational Linguistics, 2015.

[30] L. v. d. Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[31] Y. Ming, S. Cao, R. Zhang, Z. Li, Y. Chen, Y. Song, and H. Qu. Understanding hidden memories of recurrent neural networks. In *Visual Analytics Science and Technology (VAST), 2017 IEEE Conference on*. IEEE, 2017.

[32] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*. 2013.

[33] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[34] M. Monroe, R. Lan, H. Lee, C. Plaisant, and B. Shneiderman. Temporal event sequence simplification. *IEEE transactions on visualization and computer graphics*, 19(12):2227–2236, 2013.

[35] I. Osband, C. Blundell, A. Pritzel, and B. Van Roy. Deep exploration via bootstrapped dqn. In *Advances in neural information processing systems*, pp. 4026–4034, 2016.

[36] A. Perer and J. Sun. Matrixflow: temporal network visual analytics to track symptom evolution during disease progression. In *AMIA annual symposium proceedings*, vol. 2012, p. 716. American Medical Informatics Association, 2012.

[37] N. Pezzotti, T. Höllt, J. Van Gemert, B. P. Lelieveldt, E. Eisemann, and A. Vilanova. Deepeyes: Progressive visual analytics for designing deep neural networks. *IEEE transactions on visualization and computer graphics*, 24(1):98–108, 2018.

[38] C. Plaisant, B. Milash, A. Rose, S. Widoff, and B. Shneiderman. Lifelines: visualizing personal histories. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 221–227. ACM, 1996.

[39] J. Qiu, Q. Wu, G. Ding, Y. Xu, and S. Feng. A survey of machine learning for big data processing. *EURASIP Journal on Advances in Signal Processing*, 2016(1):67, 2016.

[40] S. Salvador and P. Chan. Toward accurate dynamic time warping in linear time and space. *Intelligent Data Analysis*, 11(5):561–580, 2007.

[41] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. In *International Conference on Learning Representations*. Puerto Rico, 2016.

[42] S. F. Silva and T. Catarci. Visualization of linear time-oriented data: a survey. In *Proceedings of the First International Conference on Web Information Systems Engineering*, vol. 1, pp. 310–319 vol.1, 2000. doi: 10.1109/WISE.2000.882407

[43] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[44] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[45] I. Sorokin, A. Seleznev, M. Pavlov, A. Fedorov, and A. Ignateva. Deep attention recurrent q-network. 2015.

[46] J. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for simplicity: The all convolutional net. In *ICLR (workshop track), 2015*.

[47] H. Strobelt, S. Gehrmann, H. Pfister, and A. M. Rush. Lstmvis: A tool for visual analysis of hidden state dynamics in recurrent neural networks. *IEEE transactions on visualization and computer graphics*, 24(1):667–676, 2018.

[48] K. Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[49] E. R. Tufte. *Beautiful Evidence*. Graphics Press, 2006.

[50] J. Wang, L. Gou, H. Yang, and H.-W. Shen. Ganviz: A visual analytics approach to understand the adversarial game. *IEEE transactions on*

visualization and computer graphics, 24(6):1905–1917, 2018. doi: 10.1109/TVCG.2018.2816223

[51] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas. Dueling network architectures for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, pp. 1995–2003, 2016.

[52] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[53] K. Wongsuphasawat, J. A. Guerra Gómez, C. Plaisant, T. D. Wang, M. Taieb-Maimon, and B. Shneiderman. Lifeflow: visualizing an overview of event sequences. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pp. 1747–1756. ACM, 2011.

[54] J. Zhao, M. Glueck, F. Chevalier, Y. Wu, and A. Khan. Egocentric analysis of dynamic networks with egolines. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pp. 5003–5014. ACM, 2016.

[55] J. Zhao, Z. Liu, M. Dontcheva, A. Hertzmann, and A. Wilson. Matrixwave: Visual comparison of event sequence data. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pp. 259–268. ACM, 2015.