Abstract—This supplementary material provides the following contents to support our manuscript: (1) a concrete example explaining how the DQN model works (for Section 3 of the manuscript); (2) the results of DQNViz for another Atari game, i.e., the Pong; (3) the result of plugging another statistical metric into the Epoch view (for Section 6.2 of the manuscript); (4) more explanations on how we aggregate the saliency maps from different convolutional filters (for Section 6.4); (5) the results of using t-SNE algorithm to replace PCA in the Segment view (to support our discussions in Section 8 of the manuscript); (6) the details to help readers reproduce our results in Section 7.2 of the manuscript; (7) some discussions regarding the disk usage of DQNViz.

1 A Concrete DQN Training Example

The following is an example showing how the parameters of Deep Q-Networks (DQN) are updated (the Learn stage in Figure 3 (left) of the manuscript). A DQN can be considered as a function, which maps an input state of $44 \times 44 \times 4$ (four consecutive game screens) to a vector of four elements (four predicted rewards for four actions).

In Figure 1, the DQN with its current parameter, $\theta$, takes the state $s$, which is four consecutive screens $\{s_{n-3}, s_{n-2}, s_{n-1}, s_{n}\}$, as input and outputs four scalar values, which are $\{1.91, 1.72, 1.95, 1.83\}$. The four values indicate that taking action 0 (noop), 1 (fire), 2 (right), and 3 (left) at the current state will lead to the final reward of $1.91, 1.72, 1.95$ and 1.83 respectively. Among them, action 2 (right) is the predicted action, as performing it will lead to the maximum reward, i.e. 1.95 (the predicted $q$ value for the current step).

![Fig. 1. One example of DQN training and network update.](image)

Without loss of generality, we assume that after taking the moving-right (right) action, the ball hits one brick on the lowest layer (i.e., the agent receives a 1-point reward, $r=1$), and the discount factor (explained in Section 3 of the manuscript) is 0.99, i.e. $\gamma = 0.99$.

The DQN with another set of parameters, $\theta^-$, takes the state $s^-$, which is the successive state of $s$, as input and outputs another four scalar values, which are $\{1.35, 1.39, 1.32, 1.43\}$. Among the four values, the maximum one is 1.43. Plugging this number into the Bellman equation with $r=1$ and $\gamma = 0.99$, we finally get the target $q$ value ($q_t$), which is 2.4175.

The loss of the network is the mean square difference between $q$ and $q_t$, which is 0.2186. Targeting on minimizing this loss value, the network will perform a back-propagation to tune its parameters, $\theta$. In every $C$ steps (e.g. $C=1000$), $\theta$ will be copied to $\theta^-$. Updating $\theta^-$ much less frequently is to stabilize the target $q (q_t)$ during training [3,4].

In practice, reward clipping and loss clipping are usually applied to stabilize the training. For complete details, we recommend readers to check the code in [1] or the original DQN papers [3,4].

2 Results for Another Game: Pong

With the limited space, we only demonstrated the visualization results for the Breakout game in our manuscript, as we do not want to divert the focus of our work. Our design of DQNViz can be easily adapted to several other Atari 2600 games (not all of them as we have discussed in the manuscript). We take this supplementary material as a chance to briefly demonstrate some results during our experiments with another game, i.e. the Pong game.

In Pong, the agent is trained to play ping pong with the computer. For both players (the agent and the computer), one will get 1 point if the other fails to catch the ball with the paddle. The winner of the game is the one who gets 21 points first; and the game terminates when one player wins the game. In Figure 2, the agent wins the game, i.e., the computer controls the left and right paddles, and the agent gets 16 and 3 points respectively. In terms of the training process, this game is different from the Breakout game in the following aspects:

- The agent has only one life (instead of five).
- There are six possible actions in this game (instead of four), and they are: no-operation (noop), firing the ball (fire), moving up (up), moving down (down), moving up and firing the ball (Ufire), moving down and firing the ball (Dfire).
- The possible rewards for the DQN training of this game are -1 (the agent fails to catch the ball, i.e., when the computer gets 1 point), 0, and 1 (the agent makes the computer fail to catch the ball). In Figure 2, the agent receives -13 points in the training.
- The statistics drawn based on the statistics shown in the View and Statistics view, especially the stacked area chart for reward distributions (i.e., after epoch 10, more than 90% of the rewards are 1 instead of -1).

Figure 3 shows the result of visualizing the Pong game in DQNViz. From the visualization, we have the following observations:

- The game is much easier than the Breakout game for the agent. The agent starts playing like a master after 10 epochs. This conclusion is drawn based on the statistics shown in the View and Statistics view, especially the stacked area chart for reward distributions (i.e., after epoch 10, more than 90% of the rewards are 1 instead of -1).
- We have similar observations about the action distribution over the training, i.e., the action distribution does not tend to be stable, even in later training stages. Also, the agent can achieve high rewards with different action distributions (different movement strategies).
3 Presenting Other Statistics in the Epoch View

We explained the Epoch view of DQNviz in Section 6.2 of our manuscript. By default, the action and reward distributions are shown in this view, as these two statistics are of the most interest. However, other statistics could also be plugged into this view. Here, we show the results of plugging the step distribution of the agent over five lives in one epoch, in Figure 5. The five sectors of the pie chart show the number of steps taken when the agent has 5, 4, 3, 2, and 1 life in the epoch (i.e., over 25,000 steps). The stacked bar chart on the right shows the same distribution, but in individual game episodes. This result is from epoch 120 of the Breakout game.

4 Aggregating Saliency Maps from Different Convolutional Filters

For deep neural networks that take images as inputs, many approaches have been proposed to generate sensitivity and saliency maps for individual input images, which reflect how strong different pixels of an input image affect the output of the neural networks [2, 5–7]. Among them, we adopted the guided back-propagation algorithm [7] in our
work. However, other algorithms (if preferred by domain experts) can also be plugged into the Segment view for result comparisons.

Figure 6 shows a semantic example on how we aggregate the saliency maps from different filters. Each saliency map is a gray-scale image sharing the same resolution with the network input. However, not all pixels in the saliency map are equally important to us, instead, we care more about the prominent pixels. Therefore, we use a threshold to binarize a saliency map, i.e., pixels with a value greater than the threshold are set to 1, other pixels are set to 0. In the simplified example shown in Figure 6, the resolution of each saliency map is 3×3 and the threshold is the 3rd maximum value in the map. For the saliency map from filter i, the 3rd maximum value is 3 and only one pixel (the top-left pixel) has value greater than 3. As a result, only that pixel is set to 1 (activated) in the binary saliency map. The saliency maps from the other two filters are processed in the same way. As we can see, the number of activated pixels could be less or equal to 3. After this step, the three binary saliency maps are joined together through the pixel-wise union operation.

We processed saliency maps of the selected filters (selected via brushing or lasso selections in the Segment view) in the same way in our paper. The size of each map is (84×84×4) and the threshold is the 200th maximum pixel value of the map. The number of pixels passing this threshold can be 0–200. The height of bars and the size of circles in the Segment view (Figure 7a and 7b of the manuscript) reflect this.

5 REPLACING PCA WITH t-SNE IN THE Segment VIEW

We used the PCA algorithm in the Segment view to project the high-dimensional saliency maps (resulted from the guided back-propagation of different convolutional filters) to 2D for the purpose of visualization and interaction. We chose the PCA algorithm as it is simple, sufficient for our purpose, and involves few parameters to tune with. Moreover, the projection result of PCA is also stable (i.e., different runs of the algorithm will give the same projection result). However, the PCA algorithm also has many limitations and is not necessarily the best choice. Here, we demonstrate the result from another popular dimensionality reduction algorithm, t-SNE, for comparison. The projection results shown in Figure 7 all use the same dataset, which is the same with the dataset used in Figure 7b of our manuscript.

We use the t-SNE implementation from the scikit-learn Python package (i.e., sklearn.manifold.TSNE). All parameters use the default values except perplexity and n_iter (the number of iterations). Those two parameters have big impacts on the performance of the algorithm, and we demonstrate the projection results when using different values of those two parameters in Figure 7. Although the four sets of results are not exactly the same, we can still find the same inner-to-outer layout of the convolutional filters from layer 1, 2, and 3, which is consistent with our observations when using the PCA algorithm.

6 REPRODUCING OUR RESULTS IN SECTION 7.2

To reproduce the result we demonstrated in Section 7.2 of the manuscript, one can download the trained parameter for epoch 200 (i.e., snapshots/breakout_200.pkl) from the GitHub repository in [1], and use the script, play.sh, to test the agent. The readers may need to add some customized code to dump necessary data. The $\varepsilon$ value can be changed in src/main.py (i.e., the parameter exploration_rate_test).

One can follow the details we explained in Section 7.2 to implement the pattern detection (PD) algorithm. We also provide the pseudo-code for this algorithm in Algorithm 1 of this document. This pseudo-code is for the Experiment 2 in our manuscript. For Experiment 3, one can simply change the check_len (line 1) to 100, the max_pattern (line 3) to 50, and the pattern_repeat (line 4) to 2.

7 DISK STORAGE

Among the eight types of collected data, the size of the screen data can easily become very large. To manage this, we save the full screen (84×84 pixels) of the first testing step. For the following steps, only pixels of the new screen that have different values with the corresponding pixels in the first screen are saved. Due to the temporal coherence, screens in consecutive steps are very similar. As a result, we can shrink the size of the screen data significantly. The total size of the data (the eight types of data and network parameters) collected over the 200 epochs of Breakout is around 4 GB (in binary).

REFERENCES

Algorithm 1 Random actions on-demand. Random actions are introduced when returning True. Otherwise, predicted actions are used.

1: check_len = 20 // check the latest 20 steps
2: min_pattern = 2 // minimum repeating pattern length
3: max_pattern = 7 // maximum repeating pattern length
4: pattern_repeat = 3 // the same pattern repeats for 3 times
5: if rewards[-check_len:].any() then
6:     return False // got reward in last 20 steps, no need of random
7: end if
8: if all(actions[-1] == actions[-check_len:]) then
9:     return True // repeated the same action 20 times
10: end if
11: for i = min_pattern; i < max_pattern; i++ do
12:     pattern_len = i
13:     action_len = pattern_len * pattern_repeat
14:     pattern = str(actions[-pattern_len:]) // convert to a pattern string
15:     segment = str(actions[-action_len:]) // convert to an action string
16:     // mine and count the repeating of a pattern with regExp
17:     if len(regExp(segment, pattern)) == pattern_repeat then
18:         return True // find a repeating pattern, need random
19:     end if
20: end for
21: return False // use predicted actions