A AVERAGE IMPROVEMENTS OF VPERLS, GIRIL AND OTHER BASELINES.

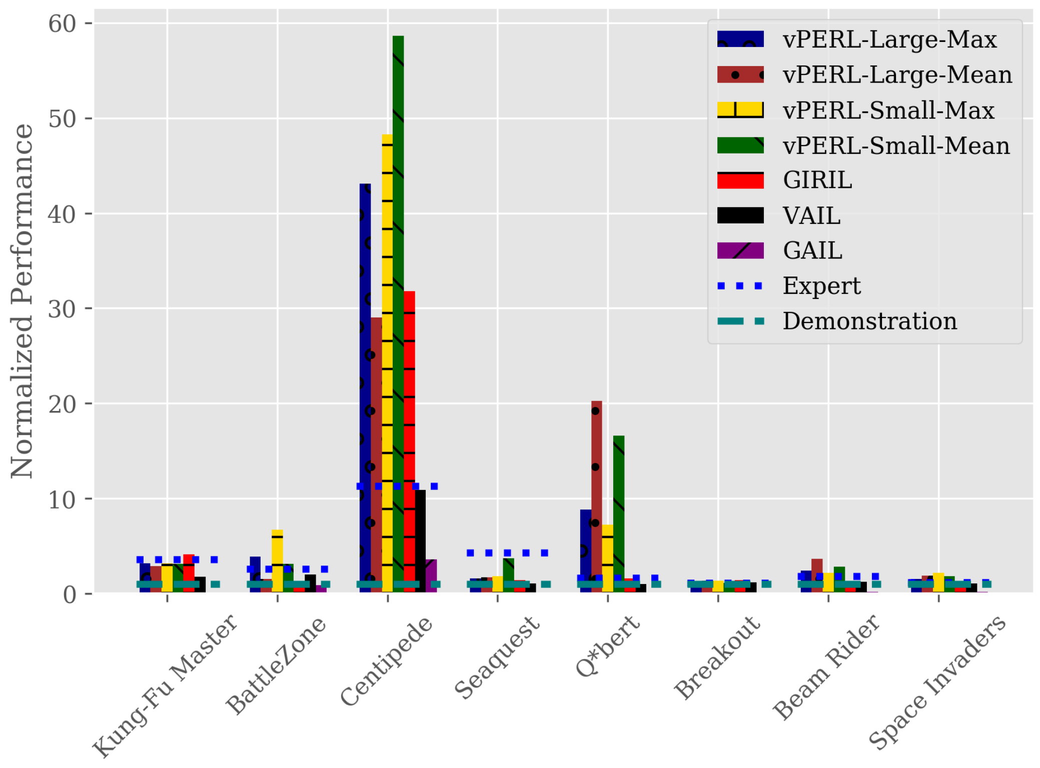


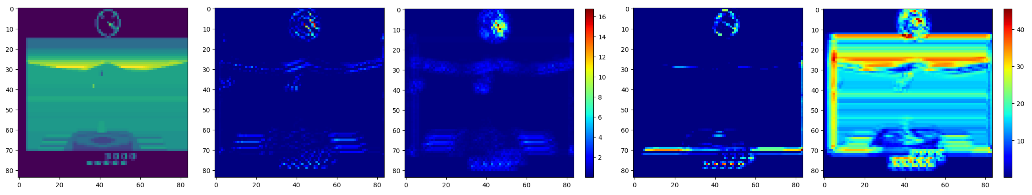
Figure 6: Performance improvement of vPERL and baselines on eight Atari games. The results are averages over 5 random seeds and reported by normalizing the one-life demonstration performance to 1.

We have provided four variants of vPERL for learning efficient reward function for imitation learning on limited demonstration data. The effectiveness of vPERL has been empirically demonstrated with diverse experiments. Figure 6 illustrates the normalized performance comparison between the four variants of vPERL and state-of-the-art imitation learning methods. The four variants of vPERL are achieved by using two network architectures (Large and Small) and two reward types (R_{Max} and R_{Mean}). For example, 'vPERL-Large-Max' means that the variant of vPERL is build on the 'Large' network architecture, and calculate the reward function as R_{Max}.

Generally, the four variants of vPERL perform consistently better than other baselines in most of the Atari games. The vPERL-Small-Mean variant achieves the highest average performance improvements against the demonstration. The vPERL-Small-Max variant performs best in Battle Zone and Space Invaders games. The vPERL-Large-Mean variant performs the best in Q*bert and Beam Rider games.

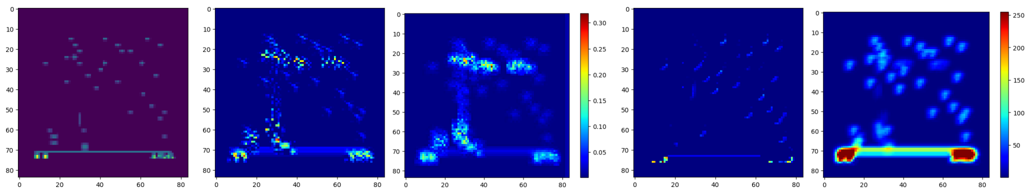
B VISUALIZATION OF REWARD AND VALUE IMAGES OF VPERL-LARGE AND VIN/BC.

In this section, we visualize the reward maps and value maps learned by vPERL-Large and VIN/BC on several Atari games. Here, both vPERL and VIN/BC are based on large-size VIN architecture. The size of reward map is 84×84. The figures show that the reward and value maps learned by vPERL are much meaningful than that by VIN/BC.



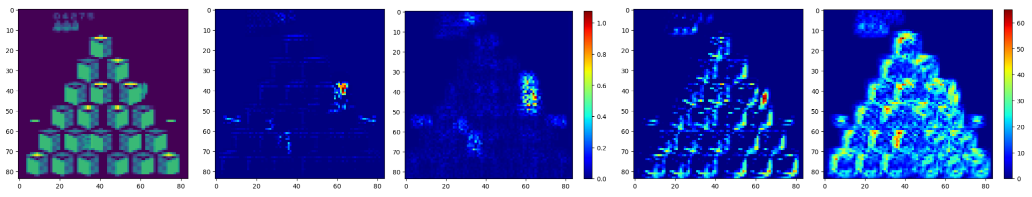
(a) State. (b) vPERL Reward and Value. (c) VIN/BC Reward and Value.

Figure 7: Visualization of state, reward map and value map on Battle Zone game. (a) The state, (b) the reward map and value map of vPERL, and (c) the reward map and value map of VIN/BC.



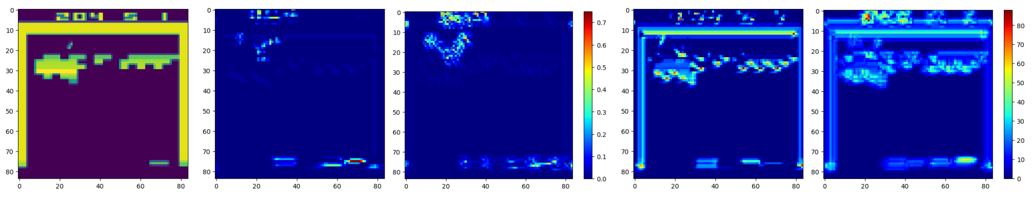
(a) State. (b) vPERL Reward and Value. (c) VIN/BC Reward and Value.

Figure 8: Visualization of state, reward map and value map on Centipede game. (a) The state, (b) the reward map and value map of vPERL, and (c) the reward map and value map of VIN/BC.



(a) State. (b) vPERL Reward and Value. (c) VIN/BC Reward and Value.

Figure 9: Visualization of state, reward map and value map on Qbert game. (a) The state, (b) the reward map and value map of vPERL, and (c) the reward map and value map of VIN/BC.



(a) State. (b) vPERL Reward and Value. (c) VIN/BC Reward and Value.

Figure 10: Visualization of state, reward map and value map on Breakout game. (a) The state, (b) the reward map and value map of vPERL, and (c) the reward map and value map of VIN/BC.

C Visualization of reward and value images of VPERL-Small and VIN/BC.

In this section, we visualize the reward maps and value maps learned by vPERL and VIN/BC on several Atari games. To enable faster training, here both vPERL and VIN/BC are based on small-size VIN architecture. The size of reward map is 18×18. The figures show that the reward and value maps learned by vPERL are much meaningful than that by VIN/BC.

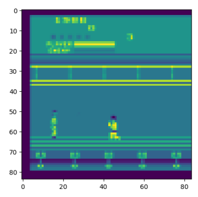
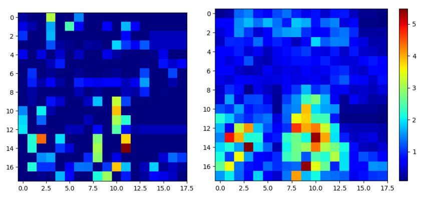
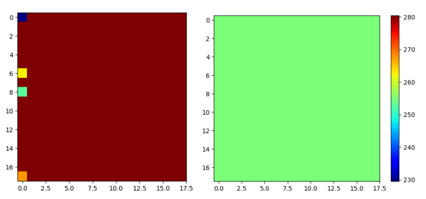

(a) State.

(b) vPERL Reward and Value.

(c) VIN/BC Reward and Value.

Figure 11: Visualization of state, reward map and value map on Kung-Fu Master game. (a) The state, (b) the reward map and value map of vPERL, and (c) the reward map and value map of VIN/BC.

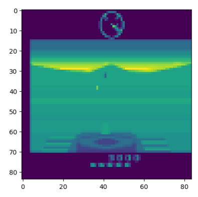
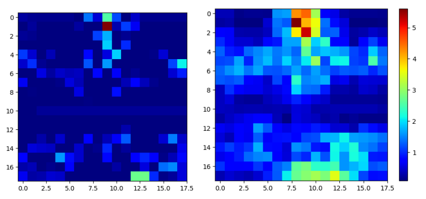
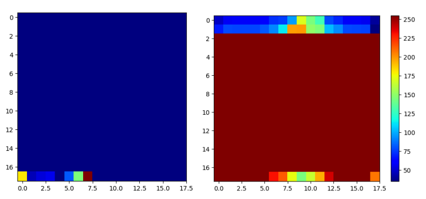

(a) State.

(b) vPERL Reward and Value.

(c) VIN/BC Reward and Value.

Figure 12: Visualization of state, reward map and value map on Battle Zone game. (a) The state, (b) the reward map and value map of vPERL, and (c) the reward map and value map of VIN/BC.

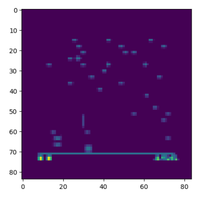
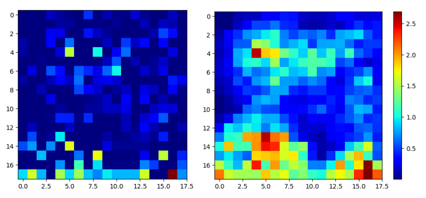
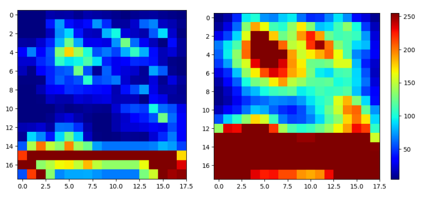

(a) State.

(b) vPERL Reward and Value.

(c) VIN/BC Reward and Value.

Figure 13: Visualization of state, reward map and value map on Centipede game. (a) The state, (b) the reward map and value map of vPERL, and (c) the reward map and value map of VIN/BC.

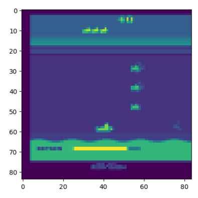
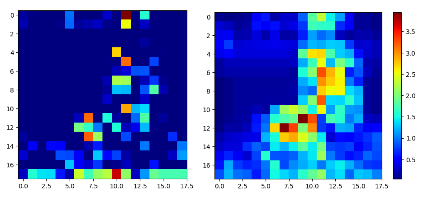
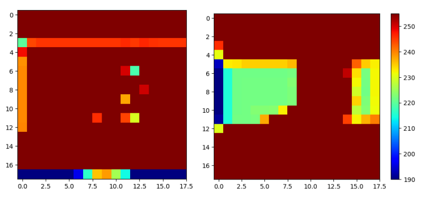
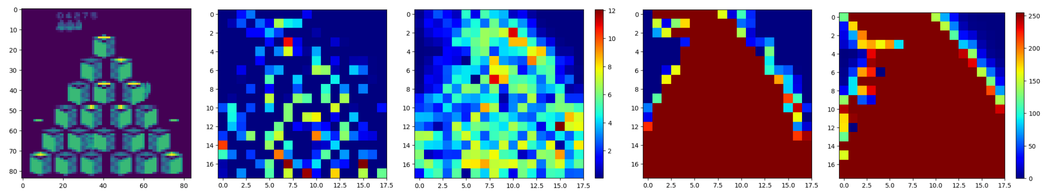
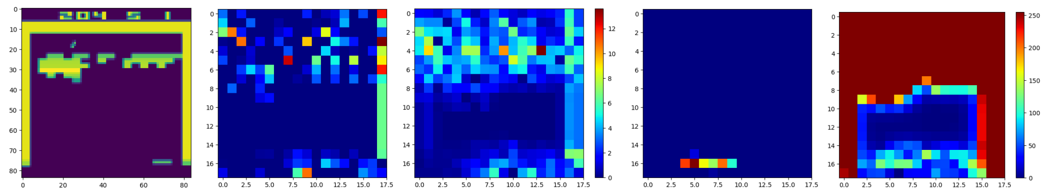

(a) State.

(b) vPERL Reward and Value.

(c) VIN/BC Reward and Value.

Figure 14: Visualization of state, reward map and value map on Seaquest game. (a) The state, (b) the reward map and value map of vPERL, and (c) the reward map and value map of VIN/BC.



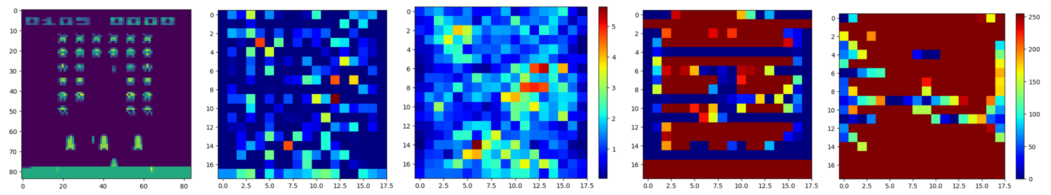
(a) State. (b) vPERL Reward and Value. (c) VIN/BC Reward and Value.

Figure 15: Visualization of state, reward map and value map on Qbert game. (a) The state, (b) the reward map and value map of vPERL, and (c) the reward map and value map of VIN/BC.



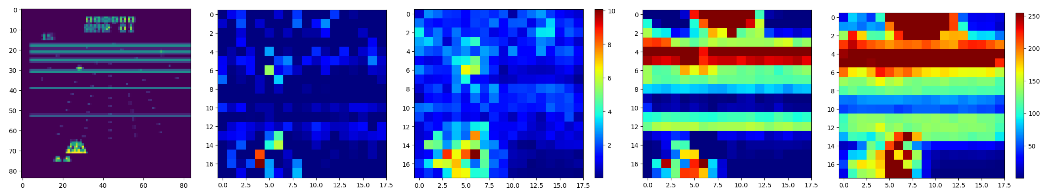
(a) State. (b) vPERL Reward and Value. (c) VIN/BC Reward and Value.

Figure 16: Visualization of state, reward map and value map on Breakout game. (a) The state, (b) the reward map and value map of vPERL, and (c) the reward map and value map of VIN/BC.



(a) State. (b) vPERL Reward and Value. (c) VIN/BC Reward and Value.

Figure 17: Visualization of state, reward map and value map on Space Invaders game. (a) The state, (b) the reward map and value map of vPERL, and (c) the reward map and value map of VIN/BC.



(a) State. (b) vPERL Reward and Value. (c) VIN/BC Reward and Value.

Figure 18: Visualization of state, reward map and value map on Beam Rider game. (a) The state, (b) the reward map and value map of vPERL, and (c) the reward map and value map of VIN/BC.

D Additional Details of Experimental Setup on Atari

D.1 Network Architecture

For a fair comparison, we used an identical policy network for all methods. We used the implementation of policy network in the code base (Kostrikov, 2018). The architecture of the policy network is outlined in Table 4.

Table 4: Architectures of *discriminator* and policy network for Atari games.

discriminator	policy network
$4 \times 84 \times 84$ States and Next States	$4 \times 84 \times 84$ States
concatenate States and Next States 3×3 conv, 32 LeakyReLU 3×3 conv, 32 LeakyReLU 3×3 conv, 64 LeakyReLU 3×3 conv, 64 LeakyReLU dense $1024 \rightarrow 1$	8×8 conv, 32, stride 4, ReLU 4×4 conv, 64, stride 2, ReLU 3×3 conv, 32, stride 1, ReLU dense $32 \times 7 \times 7 \rightarrow 512$ dense $1024 \rightarrow$ # Actions
0/1	Actions

D.2 Experimental setup of continuous control tasks

Our first step was also to train a reward learning module for each continuous control task on one demonstration. To build our reward learning module for continuous tasks, we used a simple VIN and inverse VIN as the model bases of action back-tracing and transition modeling submodules, respectively. In the simple VIN model, we used 1D convolutional layer with a kernel size of 2 and stride of 1 to implement the function f_R, reward map $\overline{R}$ and Q value $\overline{Q}$. To accomplish the action back-tracing, the final value map of VIN was fully connected with a hidden layer with a size of 32. Reversely, we used 1D deconvolutional layer to implement the inverse VIN model. We kept the size of feature maps in both VIN and inverse VIN unchanged across all the layers. We set $K = 10$ for both the VIN and inverse VIN in all tasks. The dimension of latent variable z is set to the action dimension for each task. Additionally, we used a two-layer feed forward neural network with tanh activation function as policy architecture. The number of hidden unit is set to 100 for all tasks. To extend our method on continuous control tasks, we made minor modification on the training objective. In Atari games, we used the KL divergence to measure the distance between the expert policy distribution and the action distribution in Eq. (1). In continuous control tasks, we instead directly treated the latent variable z as the back-traced action and used mean squared error (MSE) to measure the distance between the back-traced action and the true action in the demonstration. We set the scaling weight α in Eq. (1) to 1.0 for all tasks. Training was conducted with the Adam optimizer (Kingma & Ba, 2015) at a learning rate of 3e-5 and a mini-batch size of 32 for 50, 000 epochs. In each training epoch, we sampled a mini-batch of data every 20 states.

To evaluate the quality of our learned reward, we used the trained reward learning module to produce rewards, and trained a policy to maximize the inferred reward function via PPO. We trained the PPO on the learned reward function for 5 million simulation steps to obtain our final policy. The PPO is trained with a learning rate of 3e-4, a clipping threshold of 0.1, a entropy coefficient of 0.0, a value function coefficient of 0.5, and a GAE parameter of 0.95 (Schulman et al., 2016).

For a fair comparison, we used the same VIN as the model base for all the baselines. The reward function of GAIL and VAIL was chosen according to the original papers (Ho & Ermon, 2016; Peng et al., 2019). The information constraint I_c in VAIL was set to 0.5 for all tasks. To enable fast training, we trained all the imitation methods with 16 parallel processes.

D.3 Additional details of GAIL and VAIL

The *discriminator* for both GAIL and VAIL takes in a state (a stack of four frames) and an action (represented as a 2d one-hot vector with a shape of ($|\mathcal{A}| \times 84 \times 84$), where $|\mathcal{A}|$ is the number of valid discrete actions in each environment) (Brown et al., 2019b). As shown in Table 4, the network architecture of GAIL's *discriminator* D is a standard 4-layer CNN that has a comparable number of parameters as the VI module in our vPERL model. The *discriminator* outputs a binary classification value, and $-\log(D(s, a))$ is the reward. VAIL was implemented according to the repository of Karnewar (2018). The *discriminator* network architecture has an additional convolutional layer (with a kernel size of 4) as the final convolutional layer to encode the latent variable in VAIL. We used the default setting of 0.2 for the information constraint (Karnewar, 2018). PPO with the same hyper-parameters was used to optimize the policy network for all the methods. For both GAIL and

VAIL, we trained the *discriminator* using the Adam optimizer with a learning rate of 0.001. The *discriminator* was updated at each policy step.