Learning to Identify Critical States for Reinforcement Learning from Videos Supplementary Materials

Haozhe Liu[1†], Mingchen Zhuge[1†], Bing Li[1✉], Yuhui Wang[1], Francesco Faccio[1,2]
Bernard Ghanem[1], Jürgen Schmidhuber[1,2,3]
[1]AI Initiative, King Abdullah University of Science and Technology
[2]The Swiss AI Lab IDSIA/USI/SUPSI, [3]NNAISENSE
{haozhe.liu, mingchen.zhuge, bing.li, yuhui.wang,
francesco.faccio, bernard.ghanem, juergen.schmidhuber}@kaust.edu.sa

Abstract

This appendix provides the implementation details of our Deep State Identifier. In Section 1, we provide the pseudo-code for the Deep State Identifier, its network architecture, and the hyperparameters used during training. Then, Section 2 discusses the datasets we collected and our experimental protocol. Finally, Section 3 provides additional experimental results related to the ablation study and the comparison with EDGE [7] on MuJoCo.

1. Implementation Details

This section details our implementation of the proposed method. We implement our method and conduct our experiments using PyTorch [12]. All experiments were conducted on a cluster node equipped with 4 Nvidia Tesla A100 80GB GPUs.

The proposed method—while quite effective—is conceptually simple. The training pipeline can be written in 25 lines of pseudo-code:

```
import torch as T
def cs_detector_train(input_states, labels):
    mask = cs_detector(input_states)
    loss_reg = lambda_r*T.linalg.norm(mask,ord=1)
    masked_states = mask * input_states
    output = return_predictor(masked_states)
    loss_sub = lambda_s*criterion(output,labels)
    reverse_mask = torch.ones_like(mask) - mask
    reverse_states = reverse_mask * input_states
    output_r = return_predictor(reverse_states)
    confused_label = torch.ones_like(output_r)
    *0.5 #binary classification case
    loss_vic = lambda_v * criterion(output_r,
    confused_label)
    loss_total = loss_reg + loss_sub + loss_vic
    loss_total.backward()
    optimizer_cs.step()
def return_predictor_train(input_states, labels):
    output = return_predictor(input_states)
    loss_d = criterion(output,labels)
    loss_d.backward()
    optimizer_return.step()
def main_train(input_states, labels):
    optimizer_cs.zero_grad()
    cs_detector_train(input_states, labels)
    optimizer_return.zero_grad()
    return_predictor_train(input_states, labels)
```

We use two potential network architectures in our work, 3DCNN [17], and CNN-LSTM [6, 8, 10], to implement our Deep State Identifier. Tables 1 and 2 show the specification of the corresponding architectures. We use 3DCNN architecture in Table 3 and employ LSTM structure in the other empirical studies.

Table 1. **The specification of the 3DCNN-based Neural Network adopted in this paper.** In-Norm refers to the Instance Normalization, 3D Conv. is the 3D convolutional Layer, and F.C. refers to the fully connected layer. In the last layer, the [return predictor/critical state detector] has a different architecture specified in the last column.

3DCNN	Channel	Filter	Stride	In-Norm	Activation
3D Conv.	$12 \rightarrow 32$	(1,3,3)	(1,2,2)	False	Relu
3D Conv.	$32 \rightarrow 64$	(1,3,3)	(1,1,1)	True	Relu
3D Conv.	$64 \rightarrow 128$	(1,3,3)	(1,2,2)	False	Relu
3D Conv.	$128 \rightarrow 128$	(1,3,3)	(1,1,1)	True	Relu
3D Conv.	$128 \rightarrow 256$	(3,2,2)	(1,1,1)	False	Relu
Avg Pooling	-	-	-	-	-
F.C.	$256 \rightarrow 512$	-	-	-	-
F.C.	$512 \rightarrow$ [2/12]	-	-	-	[-/sigmoid]

To train the critical state detector and return predictor, we use the Adam optimizer [9] with $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The learning rate is set as 1×10^{-4} and the weight decay is 1×10^{-4}. The input length of 3DCNN is 12 frames and is a partial observation (7×7 pixels) of the environment [5, 4].

† Equal Contribution.
✉ Corresponding Author.
Accepted to ICCV23.

Table 2. **The specification of the CNN-LSTM Neural Network in this paper.** In the last layer, the critical state detector outputs a vector with the same length as the input (i.e., 256→1). The return predictor estimates a scalar for the whole episode (i.e., 256 × length →2)

CNN-LSTM	Channel	Filter	Stride	In-Norm	Activation
2D Conv.	$3 \rightarrow 32$	3	2	False	Relu
2D Conv.	$32 \rightarrow 64$	3	1	True	Relu
2D Conv.	$64 \rightarrow 128$	3	2	False	Relu
2D Conv.	$128 \rightarrow 128$	3	1	True	Relu
2D Conv.	$128 \rightarrow 256$	2	1	False	Relu
Avg Pooling	-	-	-	-	-
	Input	Hidden	Bi-Direct.	Activation	
LSTM	256	128	True	-	
F.C.	[length×256] → [2/length]			[-/sigmoid]	

The remaining hyper-parameters λ_s, λ_r, and λ_v are set to 1, 5×10^{-3} and 2 respectively.

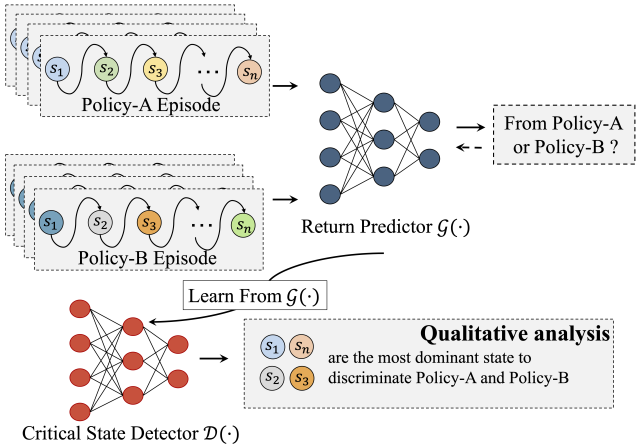

Figure 1. **Illustration of the Deep State Identifier for policy comparison.** We modify the return predictor as a binary classifier. Its training data comprises pairs $\{\mathbf{s_i}, \mathbf{c_i}\}$, where $\mathbf{s_i}$ represents a trajectory and $c_i \in \mathbb{R}$ is a class label indicating whether it belongs to policy-*A* or policy-*B*. By exploiting the return predictor, the critical state detector can directly localize the states that primarily explain the difference between policy-*A* and policy-*B*.

Fig. 1 shows how we can adapt the return predictor to find the critical frame that explains the difference in behavior between the two policies. We can train the return predictor to identify which of the two policies generates a specific trajectory.

2. Experimental details

Critical States Discovery. We use a GridWorld environment (MiniGrid-KeyCorridorS6R3-v0) to collect a dataset (Grid-World-S) to test the accuracy of the critical state detector. Data is collected by acting in the environment using an optimal policy based on a depth-first search algorithm (DFS). Additional data is collected from a random-exploring policy. Since, in this environment, one can find critical states by visual inspection (they correspond to the states immediately before or after the action of opening doors or picking up keys), we can directly test the accuracy of the proposed method. We use the F1 score as a metric.

Policy Comparison by Critical States. Here, we collect a dataset, *Grid-World-M*, for our experiments on policy comparison. The labels in *Grid-World-M* are the policies that collected the corresponding episode. We use two policies to collect data: Policy-A is the optimal policy used to collect *Grid-World-S*, while Policy-B is an exploratory policy.

Efficient Attack using Critical States. Here we use adversarial attacks on Atari-Pong to validate whether the detected states are critical. Following the same protocol as Edge [7], we use a trained policy downloaded from `https://github.com/greydanus/baby-a3c` to collect the training data. We call the corresponding dataset *Atari-Pong-S*. In particular, we collect 21500 episodes for training and 2000 for testing, and we fix the length of each episode as 200. We augment the input by randomly increasing or decreasing the length within 50 frames, and the padding value is set as 0. To validate the generalization of the proposed method for unseen policies, we then collect another dataset, denoted *Atari-Pong-M*. We train policies with different seeds using the same implementation as Edge [7] from `https://github.com/greydanus/baby-a3c`. In particular, we use ten different policies to collect training data. In cross-policy (seeds), we use the trained policy on different random seeds to test the performance. In cross-policy (steps), we use the policy trained with 80M and 40M steps for training and testing our method, respectively. In cross-policy (Arch.), we change the architecture to make the setting more challenging. In particular, we train our method using a policy with 32 channels but test it by attacking a policy trained using 64 channels. The result in each case is collected by attacking the agent for 1500 episodes using three random seeds.

Policy Improvement. We test the potential of our method to improve policy performance in the Atari-Seaquest environment. We first train the policies based on DQN following the implementation of Tianshou [18]. Then we use the trained policies to collect a dataset called *Atari-Seaquest-S*, consisting of 8000 trajectories for training and 2000 trajectories for testing. The average length of the trajectories is 2652.5, and the average return is 2968.6. We cut the trajectory into subsequences with 200 states for training. To stabilize the training, we equip an orthogonal regularization for our method. Considering the output of the predictor is a matrix, $\mathcal{M} \in \mathbb{R}^{b \times l}$ where n refers to the batch size and l is the length of $\mathbf{m}$, we drive the model to minimize the accumulation of $\mathcal{M}\mathcal{M}^T$. As the critical states of each trajectory are generally with different time steps, this regularization can benefit our approach. We train our Deep State

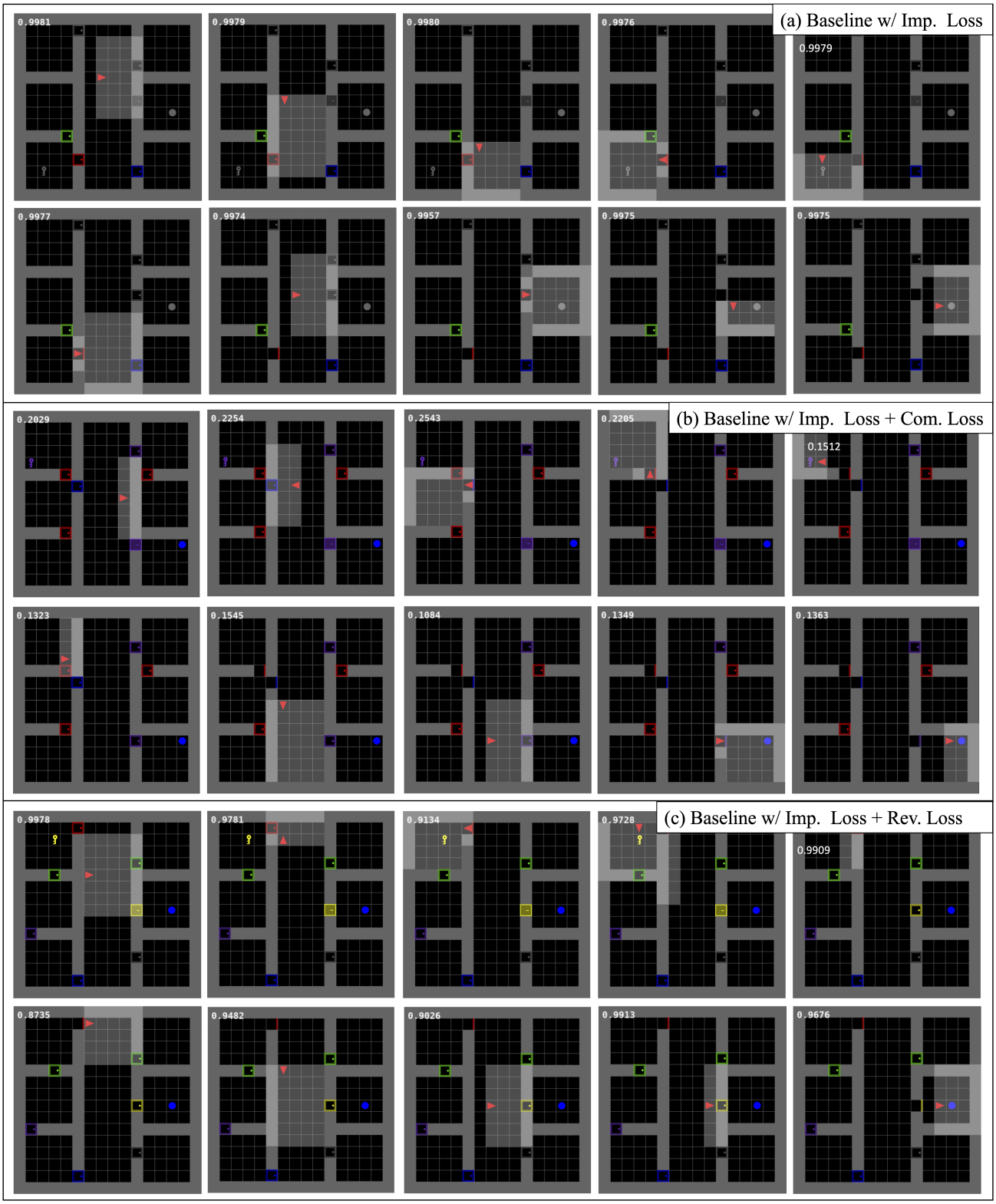

Figure 2. **Visualization of our method with different losses.** The number at the top-left corner indicates the confidence score predicted by the critical state detector, indicating whether the corresponding state is important. (a) Baseline trained with Importance Preservation loss; (b) Baseline with Importance Preservation loss and Compactness loss. (c) Baseline with Importance Preservation loss and Reverse loss. None of them can detect critical states effectively.

Table 3. **Ablation study for the Deep State Identifier.** Clean Acc. refers to the accuracy of the return predictor in the test set; Masked Acc. is the accuracy of the return predictor with the input (critical states) detected by the critical state detector; R-Masked Acc. is the accuracy of the return predictor where the masked is inverted (non-critical states are treated as critical and vice versa); L1(Mask) and Var(Mask) are the L1 norm and the average variance of the output of the critical state detector respectively.

Imp. Loss	*Com. Loss*	*Rev. Loss*	3DCNN	CNN-LSTM	Clean Acc. (%) ↑	Masked Acc.(%) ↑	R-Masked Acc.(%) ↓	L1(Mask) ↓	Var(Mask) ↑
✓	×	×	✓	×	90.07	90.07	**44.57**	63.74	2×10^{-6}
✓	✓	×	✓	×	91.45	87.71	89.28	**8.79**	0.01
✓	×	✓	✓	×	91.45	91.39	76.12	63.73	0.03
✓	✓	✓	✓	×	90.78	89.45	64.55	57.35	0.04
✓	✓	✓	×	✓	**98.66**	**98.44**	55.58	41.05	**0.12**

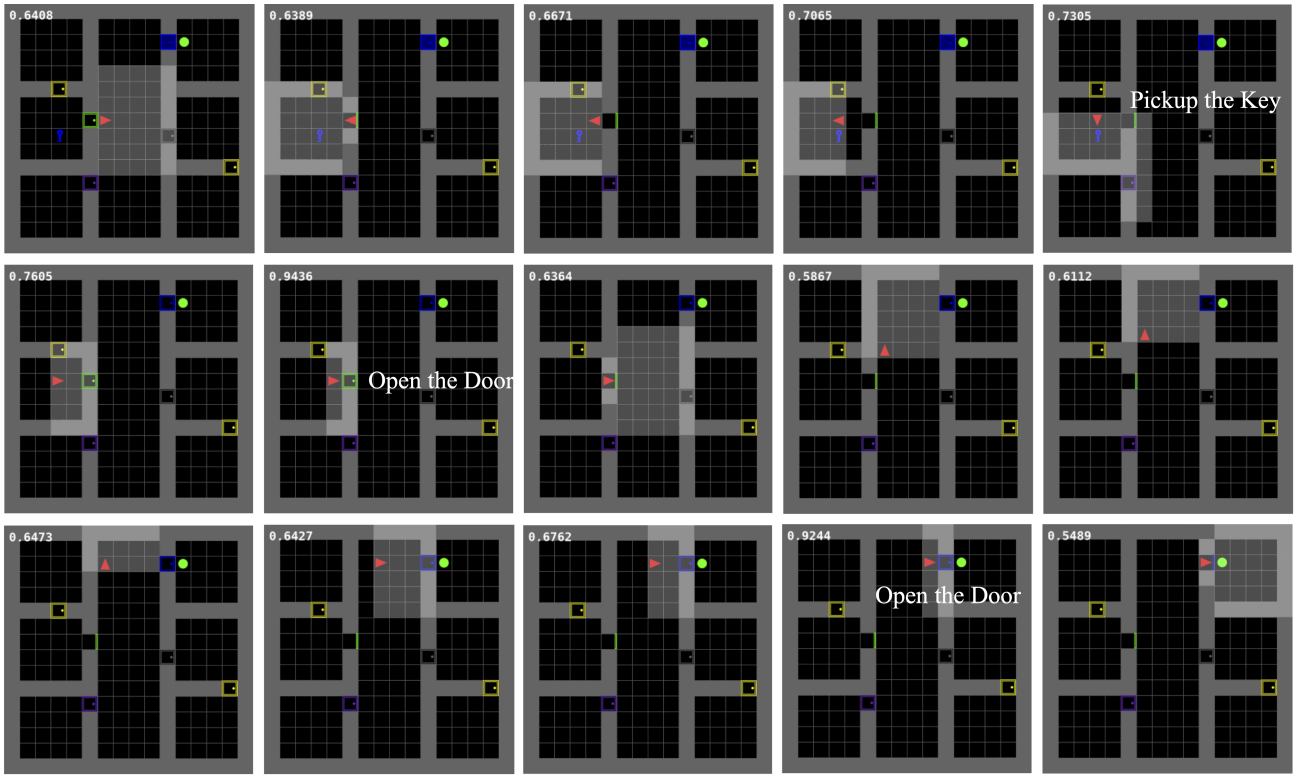

Figure 3. **Sampled observations from an episode collected by our method.** The number at the top-left corner indicates the confidence score predicted by the critical state detector, indicating whether the corresponding state is critical. Our method can localize the critical states effectively.

Identifier on this video dataset and then test its effectiveness by re-training a new adaptive multi-step DQN from scratch, where the critical state detector adaptively determines the lookahead step. We use our trained critical state detector to determine the lookahead steps for rapid credit assignment during re-training.

3. Experimental Results

To justify the effectiveness of the proposed method, we carry out some additional visualization and analysis. Table 3 shows some statistics of the output of the critical state detector and return predictor. We observe that the identified states are few (the L1 Norm is low), and the output of the return predictor does not change when it ignores non-critical states. If instead, the return predictor observes only states identified as non-critical, then the performance is much lower. These results further validate the effectiveness of the proposed method. We provide additional visualization of the performance of our method when using different losses for the critical state detector. The results are con-

Table 4. **Sensitivity Analysis of the Deep State Identifier.** We show the F1 score ↑ of our method using different hyper-parameters on GridWorld-S datasets.

$\lambda_r(\times 10^{-3})$	1	2.5	5	7.5	10	**Variance**
F1 Score	76.69	78.44	80.28	78.26	76.44	**1.39**
λ_s	0.5	0.75	1	1.25	1.5	**Variance**
F1 Score	76.68	77.50	80.28	78.18	78.83	**1.22**
λ_v	1.5	1.75	2	2.25	2.5	**Variance**
F1 Score	77.77	77.04	80.28	78.76	**83.78**	**2.39**

Table 5. **Win rate changes of the agent before/after attacks by following the protocol of EDGE [7]** We compare the methods on two MuJoCo environments: You-Should-Not-Pass game [3] (MuJoCo-Y) and Kick-And-Defend game [3] (MuJoCo-K).

Method	MuJoCo-Y	MuJoCo-K
Rudder [1]	-32.53	-21.80
Saliency [13, 14, 15]	-29.33	-37.87
Attention RNN [2]	-33.93	-41.20
Rationale Net [11]	-30.00	-7.13
Edge [7]	-35.13	-43.47
Ours	**-45.10**	**-48.03**

sistent with our empirical studies. In particular, Fig. 2(a) shows that when using only the importance preservation loss, all the states are considered critical. When adding only the compactness loss (see Fig. 2(b)) or the reverse loss (see Fig. 2(c)), the performance is still not satisfactory. The proposed method can precisely detect the critical states only when using all three losses. Indeed, as shown in Fig. 3, our method correctly outputs high confidence when the agent observes critical states (0.73, 0.94, and 0.92) and low confidence (0.6) otherwise.

3.1. Non-Vision Environment

We also tested the performance in non-vision environments [3] and compared our method with the same methods in Table **??**. As shown in Table 5, our method achieves a win rate change of **-45.10** on the MuJoCo [16] environment You-Shall-Not-Pass game, surpassing the performance of EDGE (-35.13) by 28.38%. In the Kick-and-Defense environment, our method achieves a win rate change of **-48.03**, outperforming EDGE (-43.47) by 10.49%. The consistent improvement indicates that our method exhibits strong robustness in non-vision environments.

3.2. Sensitivity analysis

We evaluate the performance of our method using different values of hyperparameters λ_r,λ_s, and λ_v. Table 4 shows that our algorithm has moderate sensitivity to hyperparameters when their values are within a specific range. For example, given $\lambda_s \in [0.5, 1.5]$, the performance variance is only 1.22, indicating stable performance. To determine the optimal hyperparameters, we searched a range of values. The best hyperparameters found in GridWorld-S were then used in all other environments.

References

[1] Jose A Arjona-Medina, Michael Gillhofer, Michael Widrich, Thomas Unterthiner, Johannes Brandstetter, and Sepp Hochreiter. Rudder: Return decomposition for delayed rewards. *NIPS*, 32, 2019.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *ICLR*, 2015.

[3] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. *arXiv preprint arXiv:1710.03748*, 2017.

[4] Maxime Chevalier-Boisvert, Dzmitry Bahdanau, Salem Lahlou, Lucas Willems, Chitwan Saharia, Thien Huu Nguyen, and Yoshua Bengio. Babyai: A platform to study the sample efficiency of grounded language learning. *arXiv preprint arXiv:1810.08272*, 2018.

[5] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. Minimalistic gridworld environment for gymnasium. *Github*, 2018.

[6] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[7] Wenbo Guo, Xian Wu, Usmann Khan, and Xinyu Xing. Edge: Explaining deep reinforcement learning policies. *NIPS*, 34:12222–12236, 2021.

[8] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.

[9] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[11] Tao Lei, Regina Barzilay, and Tommi Jaakkola. Rationalizing neural predictions. *EMNLP-IJCNLP*, 2017.

[12] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[13] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

[14] Daniel Smilkov, Nikhil Thorat, Been Kim, Fernanda Viégas, and Martin Wattenberg. Smoothgrad: removing noise by adding noise. *arXiv preprint arXiv:1706.03825*, 2017.

[15] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *ICML*, pages 3319–3328. PMLR, 2017.

[16] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[17] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. Learning spatiotemporal features with 3d convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 4489–4497, 2015.

[18] Jiayi Weng, Huayu Chen, Dong Yan, Kaichao You, Alexis Duburcq, Minghao Zhang, Yi Su, Hang Su, and Jun Zhu. Tianshou: A highly modularized deep reinforcement learning library. *JMLR*, 23(267):1–6, 2022.