From Video Game to Real Robot: The Transfer between Action Spaces

Janne Karttunen*, Anssi Kanervisto*, Ville Hautamäki and Ville Kyrki

***Abstract*— Training agents with reinforcement learning based techniques requires thousands of steps, which translates to long training periods when applied to robots. By training the policy in a simulated environment we avoid such limitation. Typically, the action spaces in a simulation and real robot are kept as similar as possible, but if we want to use a generic simulation environment, this strategy will not work. Video games, such as Doom (1993), offer a crude but multi-purpose environments that can used for learning various tasks. However, original Doom has four discrete actions for movement and the robot in our case has two continuous actions. In this work, we study the transfer between these two different action spaces. We begin with experiments in a simulated environment, after which we validate the results with experiments on a real robot. Results show that fine-tuning initially learned network parameters leads to unreliable results, but by keeping most of the neural network frozen we obtain above 90% success rate in simulation and real robot experiments.**

I. INTRODUCTION

When it comes to training robots to solve a given task with reinforcement learning, one feasible way to do this is by training the policy in a simulation and then using transfer learning [1] to move it to a real-world robot; A *simulation-to-real* or *virtual-to-real* transfer. This way we are not hindered by expensive and slow robotics experiments. However, simulations rarely model real-world perfectly, and thus data obtained from simulation may not be directly applicable to real-world robot, a problem termed *reality gap*. To address this, one can try to create as realistic simulation as possible or use methods like domain randomization [2] for training more general policies.

Video games can act as one such simulation: They can be ran fast, are readily available and are shown to be useful in AI research [3] [4]. Software packages like ViZDoom [5] are specifically designed for reinforcement learning. When used to train policies for real-world robots, the before-mentioned domain randomization can be used to narrow the reality gap between visual appearance of the two worlds, but what about the action space? Video game requires discrete button presses, but a robot may have multiple motors which have continuous control. Same applies to other types of transfer scenarios, where the structure of a robot differs between environments [6].

In this work we study how effective four simple transfer learning methods are when we wish to transfer a deep learning reinforcement (DRL) agent from a video game (Doom) to a real robot, where action space is different, but the task is the same. These methods consist of training the agent in a video game, moving some or all of its neural network's parameters to agent on a real-world robot, and updating a selected set of the parameters with further training. We select two different DRL algorithms, *deep Q-learning* (DQN) [3] and *proximal policy optimization* (PPO) [7], and train a policy for a simple task to reach large red object in Doom. We then apply the different methods to transfer this policy to another action space, first inside the same video game and finally on a real-world robot.

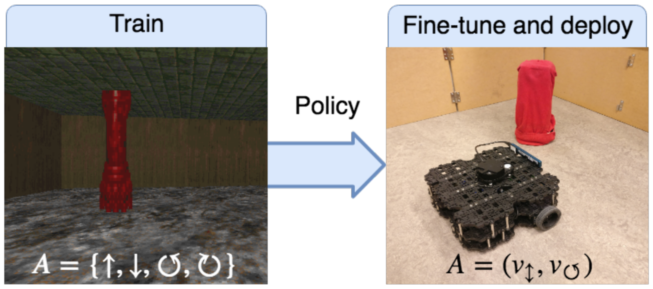

Train
Policy
Fine-tune and deploy
$A = \{\uparrow, \downarrow, \circlearrowleft, \circlearrowright\}$
$A = (v_{\updownarrow}, v_{\circlearrowleft})$

Fig. 1. A policy by was trained in a video game (Doom) in an action space consisting of four discrete actions, and then transferred to a robot with a different action space with a little amount of training on the robot.

Our experimental results show that loading pre-trained parameters and continuing training from there leads to unstable results and catastrophic forgetting, also discussed in [8]. However, freezing most of these parameters by not updating them during training yields reliable speed-ups when moving policies to different action spaces. We confirm its applicability to robotics tasks with a successful transfer from a simulation to a real-world robot with little training.

II. RELATED WORK

Our work is closely related to experiments conducted by Rusu *et al.* in [8], where authors presented a neural network architecture which was able to quickly learn to play new Atari game, once it was originally trained in another game. While methods defined in their work are similar to ours, we focus on same task under different action spaces, rather than transferring skill between different tasks. Gupta *et al.* [6] presented a method to learn a feature extraction method using multiple skills, by finding which skill in one domain is a closest match to skill in another domain and demonstrated the effectiveness of the method by transferring learned skills between morphologically two different robots with different number of joints.

Janne Karttunen, Anssi Kanervisto and Ville Hautamäki are with School of Computing, University of Eastern Finland, Joensuu, Finland. {jannkar, anssk, villeh}@uef.fi
Ville Kyrki is with Department of Electrical Engineering and Automation, Aalto University, Espoo, Finland. ville.kyrki@aalto.fi
* Equal contribution.

A. Reinforcement learning in robotics with simulations

This work was motivated by the popularity of using reinforcement learning in robotics, despite RL being known to require large number of training samples and thus making it difficult to apply to robotics [9]. Part of this RL plus robotics work focuses on training policies in simulations and then transferring them to real robot, with or without further training on the robot [1]. Such work focuses on e.g. learning models that predict real-world dynamics [10] or the use of high-fidelity simulations that are tuned to match the real-world [11] [9].

Domain randomization is one such technique for learning real-world policies in simulations. This entails randomizing the simulation in different ways, so that the learned policy has to generalize over different system dynamics. This includes randomizing the visual appearance of the simulation [2], randomizing the dynamics such as friction [12] and/or including limitations of the robots such as delays between decided actions and actuated actions [13].

B. Video games as learning environments

Video games can be, and have been, used as benchmark environments for different learning techniques (e.g. Atari games [14], Doom [5], Starcraft [15] and Toribash [16]). They provide wide range of complex tasks and environments, which were originally designed for human players. While they lack the fidelity of accurate physics simulations, they emulate the real-world to an extent human players are comfortable with.

Even though video games do not model our world perfectly, their engines have been used to simulate real-world scenarios. AirSim [17] and Holodeck [1] use Unreal Engine to create a high-fidelity simulation for autonomous vehicles and/or for multi-agent scenarios. Unity engine has also been used to train a control policy for complex hand control, despite the fact Unity engine was not designed for such purposes originally [12].

III. ACTION SPACE TRANSFER IN REINFORCEMENT LEARNING

Our goal is to transfer a learned control policy from simulated/virtual environment into a real robot, where the action space differs from the simulation's action space, but rest of the variables stay roughly the same (visual appearance, dynamics, task). If after little or no training the agent is able to complete the task in the new action space, we consider the transfer successful. We will refer to this transfer as *sim-to-real* transfer.

We train a policy with reinforcement learning in a simulated environment (”*source*”), and then use that learned knowledge to speed up the training in real robot or another simulation (”*target*”). Since our two environments are similar to each other, the neural network has useful prior knowledge that can be used to speed up the training in target environment [8]. However, as the action space between these two environments differ, we can not directly re-use the same neural network to produce actions for the target environment.

With the modularity of the neural networks, we can fix this by replacing the final layer of the neural network to reflect the new action space. If randomly initialized, this final layer requires some training in the target domain to produce useful actions. At this point we have multiple choices as to how this training should be done. We opted for four similar and simple methods for our study (see Figure 2), roughly similar to baseline methods used in [8].

1) Fine-tuning: Target model uses source model's parameters as initial values, and begins training from there, *fine-tuning* the previously learned parameters [1]. This is known to be prone to catastrophic forgetting [8], where neural network ”forgets” the good parameters learned in the previous environment, and thus may not perform as well as expected.

2) Replace: We can avoid catastrophic forgetting by not updating some of the neural network parameters at all (”freezing” most of the network). Since our two environments are visually similar, feature extraction learned in the early layers of the neural network can be applicable to both environments, and thus we do not want to update them. In this work, we freeze all layers except the output layer for actions or Q-values.

3) Replace with pre-trained value function: The reason we can not directly move source policy to target environment is due to mismatch between action spaces. However, many reinforcement learning agents learn the state-value function which is a single scalar value per state, and thus transferable between action spaces. Actor-critic methods utilize value function for variance reduction during training [18], and in value-based methods like Q-learning the state-action values can be decomposed into values and advantages [19]. Since value function is dependent on the policy, we opt for loading the parameters for value-estimation function and updating them during training, instead of freezing those parameters.

4) Adapter: Instead of updating parameters of the source network, we keep them all fixed and learn a mapping from source actions to new actions. I.e. we figure out which action in source environment matches an action in the target environment. Similar method has been used successfully with policy transfer from one domain to another [20]. We implement this by adding a new layer which maps old actions to new actions.

IV. EXPERIMENTS

A. Experimental setup

Agent's task is to navigate to a red goal pillar in a square room without obstacles, using visual input as the observations (see Figure 1). Agent receives 1 reward for reaching the goal and -1 reward if episode times out after 1000 environment steps. Agent chooses an action every 10 environment steps in case of simulation experiments. On every agent step (referred as steps from now on) agent receives an RGB image from the current view-point which

[1]https://github.com/byu-pccl/holodeck

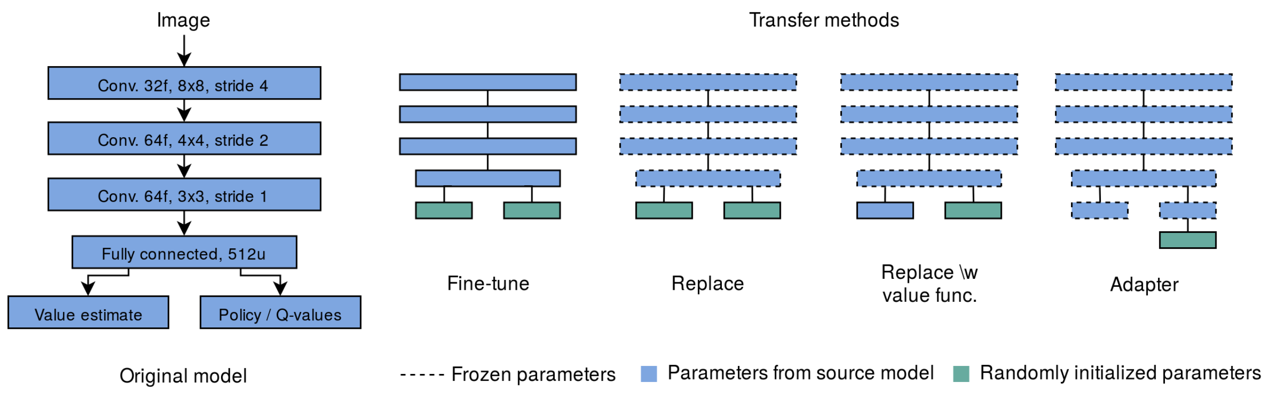


Fig. 2. Overview of the different methods of moving pre-trained models to new action space (right) and the neural network architecture used throughout the experiments (left).

is re-scaled to an 80×60 image. We only use green channel to highlight the goal object in the image.

We use two RL learning algorithms for our experiments: Deep Q-learning [3] and Proximal Policy Optimization [7]. We use double Q-learning [21] and dueling architecture [19] with the DQN. We use dueling architecture specifically to obtain the state-value function explicitly for the DQN.

We select DQN as it is known to be sample efficient, thanks to its off-policy learning and replay memory. We also include experiments with PPO for its applicability to continuous action spaces and for its closer connection to optimizing policy directly. We use existing, curated implementations of these methods to avoid implementation bugs [22], and all our experiments use neural network architecture presented in Figure 2. The experiments are run on an Ubuntu 16.04 machine using Python.

1) Source environment: For the source environment, we use ViZDoom [5] platform, which is based on the Doom (1993) video-game. ViZDoom offers lightweight 3D game environment, and outputs raw visual image frames. Custom map was created with the Slade 3 Doom map editor[2]. The map is a small room with four walls, ceiling and floor (Figure 1). The agent starts each episode from the center of the room, facing to a random direction. The agent should navigate to a tall red goal pillar, which is placed on random location in the room, yet not too close to the wall or the agent. The action space consists of four different actions; move forward, move backward, turn left and turn right.

To ensure that policy can be transferred to a robot, we randomize visual environmental details for every episode. Textures for walls, ceiling and floor were randomly chosen from 68 different textures. Every wall in the room had the same texture. Agent's field of view (FOV) was set randomly, as well as the view height and small head up and down movement. Each frame was modified to have random amount of white or Gaussian noise, with a random gamma-correction value.

2) Simulation to simulation: Experimenting directly with real robot would be expensive, which is why we transfer the policy first between two Doom environments and experiment the most efficient way to learn the new action space. In these simulation-to-simulation experiments (”*sim-to-sim*”) the policy is transferred similarly as it would be transferred to the real-world robot.

The setup is similar as in the source environment setup, but now the domain is not randomized and the surfaces of the room have new unseen textures. Also, action space is different from previously used four actions. For PPO algorithm we have defined two continuous actions, forward/backward velocity and left/right velocity. Since DQN on default does not support continuous action space [3], we simulate continuous actions by creating 24 discrete actions, containing different velocities for forward/backward movement and left/right turning. Since we aim for high sample-efficiency, we train DQN agents only for $50,000$ steps in sim-to-sim experiments but we train PPO agents for $1,000,000$ steps as they require more training samples.

3) Robot: To study the transfer to a real robot, Turtlebot 3 Waffle with Intel Realsense camera was used. The policy update is done on remote server and the Turtlebot communicates over ROS platform [4] to send states and receive actions.

Setup for the real-world experiments is similar as the sim-to-sim setup, but the difference is now that the experiments are ran on real-world. The environment is structurally similar (Figure 1), but the dynamics and textures differ from the Doom environment. The room is built in open office and has four walls, floor, but no ceiling. We increased number of environment steps per action for the robot experiments from 10 to 15 to compensate slow moving speed of the robot.

4) Hyperparameters: Most of the DQN hyperparameter values were default values which were used in original DQN Atari experiments [3]. The training starts after 1000 steps with learning rate of $5 \cdot 10^{-4}$ and discount factor 0.99. One network update was done every 4 steps with batch size of

[2] `http://slade.mancubus.net/`

[3]Methods like deep deterministic policy gradients [23] can be seen as continuous variants of DQN, but for simplicity we opt to use DQN.

[4] `http://www.ros.org/`

128. Adam was used as the optimizer and target network updated between 5000 steps. Replay memory size was 50000 steps.

To find a suitable exploration strategy, we performed hyperparameter search for the epsilon in the sim-to-sim environment with *replace* method. Epsilon was annealed from 1.0 to 0.02 for $\{500, 2500, 5000, 7500, 10000, 12500, 25000\}$ steps and we found out that best results were obtained by annealing epsilon for 7500 steps.

The PPO implementation used in this work includes generalized advantage estimation (GAE) [24] and normalization of the returns. We used discount factor of 0.99. Optimizer was Adam with $2.5 \cdot 10^{-4}$ learning rate. The number of steps per policy update was chosen to be 128, as our experiments with 512 and 1024 values did not improve the results. Policy was updated for 4 epochs over gathered samples per policy update. Weight of value loss was 0.5, maximum value for gradient clipping 0.5 and λ parameter for GAE 0.95. Similar to DQN's epsilon value, PPO has weight of entropy loss, which encourages exploration. We ran search over different magnitudes between $[1, 10^{-6}]$ values and selected 10^{-3} for further experiments as best performing weight for entropy loss.

B. *Source models*

To perform action space transfer, we first trained agents in the Doom environment with four discrete actions. Since the source model parameters can have an effect on final performance of the transfer, we train three separate source models with both DQN and PPO, and repeat all experiments over the three source models. The agents were trained for $1,000,000$ steps, as the robust policy is more important at this point than the length of training time.

The three DQN source models reached a $90-95\%$ testing success rate and mean episode length of $18.57-27.54$ steps, while PPO reached $98-100\%$ success rate with mean of $10.45-14.01$ steps per episode. The models performed well, but DQN did not score as well as PPO. By analyzing the gameplay of the agents, we observed that certain dark textures seemed to cause issues with the agent performance. From each source model, the best success rate scoring policy was selected for the next experiments.

C. *Sim-to-sim experiments*

The policy was transferred from domain randomized Doom environment with four actions to a similar environment with different, continuous action space (24 discrete actions with DQN). The main goal of these experiments was to find out which approaches suit best for the action space transfer.

As a baseline result, we trained a model from the scratch in sim-to-sim environment. Without loading any weights from the previous models, the DQN agent was able to reach on average the 99.5% success rate with the final policy (Table I). The episode was solved efficiently in less than 20 steps on average after $30,000-40,000$ steps of training (Figure 3). PPO agents were not able to solve the task with continuous

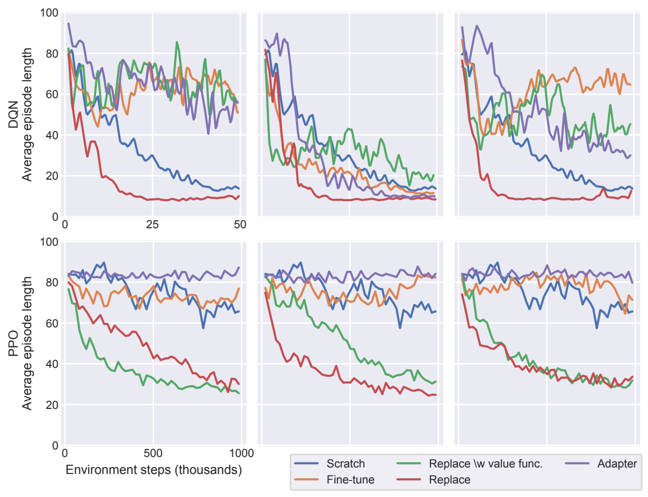


Fig. 3. Results of transferring policy to new action space with different transfer methods, source models and learning algorithms. Lower is better. Each line is an average over five repetitions. We omitted variance for visual clarity. All three source models achieved similar performance in initial training, but using their parameters for transfer learning produces very different results (upper three plots). Only by freezing most of the network we can reliably train policies in a new action space (red line, "Replace"). Rows share same baseline result (no transfer learning, "Scratch").

action space even after $1,000,000$ environment steps when trained from scratch, while they were able to solve the task in simpler action space utilizing the source models.

With the *adapter* method, the DQN reached mean of $66.3-100\%$ testing success rates across the source models when trained for $50,000$ steps and PPO $29.7-31.6\%$ success rates, trained for $1,000,000$ steps (Table I). The *replace* method approach performed more robustly with the mean success rates of $99.2-100\%$ with DQN and $95.2-98.0\%$ with PPO. With *replace* method, the DQN agent solved the task efficiently at around $20,000$ steps on average, while with the PPO, this took about $500,000$ steps (Figure 3).

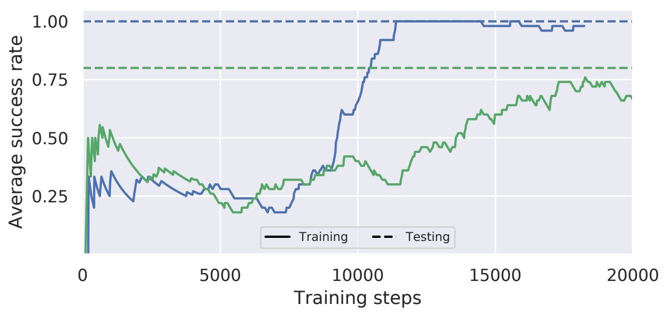
For the DQN, *fine-tune* and *replace with loaded value function* did not improve the performance. Instead, they performed worse than training from scratch (Table I) while it would be intuitive to think that using the pre-loaded state-value function improved the performance. Interestingly, source model 2 scored good success rates with all the tested methods on the final policy. With the PPO, *fine-tuning* of the model performed worse than previous replace layer method, scoring a low $31.7-52.7\%$ success rate. However, *replace with loaded value function* resulted to slightly better performance than the *replace* method with PPO.

As a summary, only the *replace* method resulted in robust transfer between action spaces for both learning algorithms, while *replace with loaded value function* performed well with PPO. For DQN, other methods than *replace* performed worse than learning a new model from scratch, except for one source model (source model 2) where all the methods performed better than learning from scratch. Based on these results, parameters from source model are only beneficial if applied correctly for transfer between action spaces.

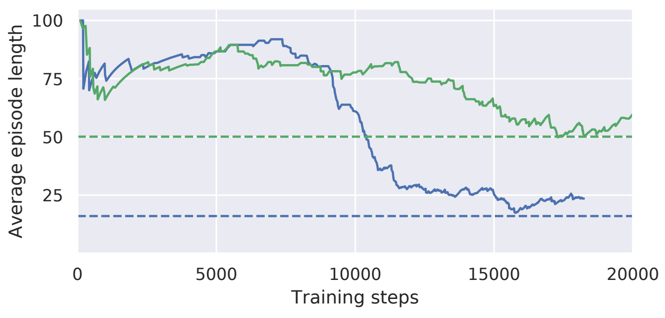
TABLE I

MEAN AND STANDARD DEVIATION OF FINAL SUCCESS RATES IN PERCENTAGES. EACH RESULT IS BASED ON AVERAGE PERFORMANCE OVER LAST 10% TRAINING EPISODES, AND AVERAGED OVER FIVE REPETITIONS. EXPERIMENTS WITH AVERAGE SUCCESS RATE ABOVE 90% ARE HIGHLIGHTED. *Replace* METHOD IS RELIABLE WAY TO TRANSFER TRAINED MODEL TO NEW ACTION SPACE, WHILE INITIALIZING WITH A LOADED VALUE FUNCTION ALSO WORKS FOR PPO. SOURCE MODEL PARAMETERS CAN DRAMATICALLY AFFECT THE FINAL PERFORMANCE (DQN, SOURCE MODEL 2).

	Learning method and source model					
	DQN			PPO		
Transfer method	1	2	3	1	2	3
Fine-tune	58.9 ± 28.7	**100 ± 0.0**	54.1 ± 26.7	46.8 ± 28.4	31.7 ± 14.6	52.7 ± 21.2
Replace	**99.9 ± 0.5**	**100 ± 0.0**	**99.2 ± 3.0**	**95.2 ± 6.7**	**98.0 ± 2.3**	**95.9 ± 5.4**
Replace \w value func.	60.1 ± 33.6	**95.7 ± 6.5**	86.8 ± 9.5	**98.0 ± 2.3**	**96.1 ± 3.5**	**98.0 ± 1.3**
Adapter	66.3 ± 31.4	**100 ± 0.0**	88.9 ± 19.0	29.7 ± 8.7	30.0 ± 7.4	31.6 ± 7.8
Scratch		**99.5 ± 1.1**			56.0 ± 25.4	



(a)



(b)

Fig. 4. Running average of success rates (a) and episode lengths (b) of the two Turtlebot experiment runs, in different colors. Averaging is done over 50 episodes or less at the beginning of the curve. Training lasted until $20,000$ steps or when agent was able to complete the task reliably (i.e. on every run). Dashed line represents the performance of the best model obtained during training. The agent learns to complete the task in one experiment in roughly $12,000$ steps.

D. Robot experiments

Finally we validate our sim-to-sim experiments on the Turtlebot platform. Based on the previous results, we chose DQN algorithm with the *replace* method for these experiments. We selected DQN model 3 as the source model for robot experiments, due to its fastest learning in the *replace* method experiment. We confirmed that the domain transfer from Doom to real robot works by evaluating the source model on the robot with four discrete actions successfully, with the mean success rate of 100% and 15.33 mean episode length, over 30 test episodes.

As with the sim-to-sim experiments, we define new action space as 24 discrete actions to roughly simulate continuous action space. We trained the agent for $20,000$ steps or until the agent performance did not increase. Turtlebot takes approximately two actions per second, which translated to $4-5$ hours of wall-clock time per one experiment, including the time to reset the episode.

We conducted two training runs with the Turtlebot. The agent of first run scored 80% mean testing success rate and 50.1 episode length with its best policy (Figure 4). The second agent performed clearly better, scoring 100% mean success rate and 16.0 episode length. Subjectively, the first agent was attracted by the goal but repeatedly chose to reverse away from the goal. The second agent rotated in place until red pillar appeared to its field-of-view and began moving towards to goal, doing small fine-adjustments to stay on correct path.

Note that while in general the task and dynamics are the same, our robot environment differs from simulation environment notably: The agent was able to glide against the walls in Doom, the Doom environment had a ceiling, exact timing of the reward varied due to human input in robotics experiments and all hyperparameters were selected based on the experiments in simulations.

V. CONCLUSIONS

In this work we show how simply freezing most of the pre-trained neural network parameters can be used to effectively transfer a control policy from a video game to a robot, despite different action spaces between these two environments. We trained a policy on raw image data to solve a simple navigation task in Doom video game, and then successfully transferred it to a Turtlebot robot where it was able to complete the same task with relatively little amount of training. These methods have promise to utilize crude simulations like video games to train policies for robots with different physical properties.

The future work could extend the present work in terms of learning complicated abstract task in video game and then transferring to the vastly different action space structure in the physical robot. We also plan to study if catastrophic forgetting can be avoided by using the Bayesian methods with good priors for the network parameters.

ACKNOWLEDGMENT

We would like to thank Juhani Seppälä for the comments on the manuscript. This research was partially funded by the Academy of Finland (grant #313970) and Finnish Scientific Advisory Board for Defence (MATINE) project nr. 2500M-0106. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used for this research.

REFERENCES

[1] M. E. Taylor and P. Stone, "Transfer learning for reinforcement learning domains: A survey," *Journal of Machine Learning Research*, vol. 10, no. Jul, pp. 1633–1685, 2009.

[2] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 23–30, IEEE, 2017.

[3] 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*, vol. 518, no. 7540, pp. 529–533, 2015.

[4] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, *et al.*, "Starcraft ii: A new challenge for reinforcement learning," *arXiv preprint arXiv:1708.04782*, 2017.

[5] M. Kempka, M. Wydmuch, G. Runc, J. Toczek, and W. Jaśkowski, "Vizdoom: A doom-based ai research platform for visual reinforcement learning," in *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pp. 1–8, IEEE, 2016.

[6] A. Gupta, C. Devin, Y. Liu, P. Abbeel, and S. Levine, "Learning invariant feature spaces to transfer skills with reinforcement learning," in *ICLR*, 2017.

[7] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[8] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell, "Progressive neural networks," *arXiv preprint arXiv:1606.04671*, 2016.

[9] L. Tai, G. Paolo, and M. Liu, "Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 31–36, IEEE, 2017.

[10] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter, "Learning agile and dynamic motor skills for legged robots," *Science Robotics*, vol. 4, no. 26, p. eaau5872, 2019.

[11] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke, "Sim-to-real: Learning agile locomotion for quadruped robots," *Proceedings of Robotics: Science and System*, 2018.

[12] OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, *et al.*, "Learning dexterous in-hand manipulation," *arXiv preprint arXiv:1808.00177*, 2018.

[13] R. Antonova, S. Cruciani, C. Smith, and D. Kragic, "Reinforcement learning for pivoting task," *CoRR*, vol. abs/1703.00472, 2017.

[14] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, jun 2013.

[15] G. Synnaeve, N. Nardelli, A. Auvolat, S. Chintala, T. Lacroix, Z. Lin, F. Richoux, and N. Usunier, "Torchcraft: a library for machine learning research on real-time strategy games," *arXiv preprint arXiv:1611.00625*, 2016.

[16] A. Kanervisto and V. Hautamäki, "Torille: Learning environment for hand-to-hand combat," *arXiv preprint arXiv:1807.10110*, 2018.

[17] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and service robotics*, pp. 621–635, Springer, 2018.

[18] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *International Conference on Machine Learning*, 2016.

[19] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning," *International Conference on Machine Learning*, 2016.

[20] F. Fernández and M. Veloso, "Policy reuse for transfer learning across tasks with different state and action spaces," in *ICML Workshop on Structural Knowledge Transfer for Machine Learning*, Citeseer, 2006.

[21] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning.," in *AAAI*, pp. 2094–2100, 2016.

[22] A. Hill, A. Raffin, M. Ernestus, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines." `https://github.com/hill-a/stable-baselines`, 2018.

[23] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *International Conference on Learning Representations*, 2016.

[24] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," in *International Conference on Learning Representations*, 2016.