5 Experiments

5.1 Experiment Setup

5.1.1 Basics

... can't think of a better title. ... For all of our experiments, we use unrolled steps $K = 5$, history length $L = 4$, bootstrap steps $N = 10$, a discount of 0.997, and a support Z from the interval $[-30, 30]$ ($|Z| = 61$). We use a single computer with Intel Xeon CPUs (72 $\times$ 2.3 GHz), Nvidia Tesla V100 GPUs (8 $\times$ 32 GB), and 500 Gigabytes of system memory. A running MooZi system roughly uses 50% to 75% of the CPUs, 40% to 60% of the GPUs, and 25% of the memory.

5.1.2 Neural Network Configurations

We use the residual-blocks-based variant of the network for all of our experiments.

Residual Block

We follow the residual block definition by He et al. [23]. One residual block is defined as follows:

- input x
- save a copy of x to x'
- apply a 2-D padded convolution on x, with kernel size 3 by 3, same channels
- apply batch normalization on x
- apply relu activation on x
- apply a 2-D padded convolution on x, with kernel size 3 by 3, same channels
- apply batch normalization on x
- add x' to x
- apply relu activation on x

The Representation Function

The representation function h is defined as follows

- input stacked history ψ of shape (H, W, C_h)
- apply a 2-D padded convolution on ψ, with kernel size 1 by 1, 32 channels
- apply 6 residual blocks with 32 channels on ψ
- output the hidden state $\mathbf{x}$ of shape $(H, W, 32)$

The Prediction Function

The prediction function f is parametrized as follows

- input hidden state $\mathbf{x}$ of shape $(H, W, 32)$
- apply 1 residual block with 32 channels on $\mathbf{x}$
- flatten $\mathbf{x}$, now shape $(H * W * 32)$
- apply 1 dense layer with output size of 128 on flattened $\mathbf{x}$ to obtain the value head $\mathbf{x}_v$, now shape (128)
- apply batch normalization and relu activation on $\mathbf{x}_v$
- apply 1 dense layer with output size of $|Z|$ on $\mathbf{x}_v$, now shape (1)
- output the value head $\mathbf{x}_v$ as the value prediction v
- apply 1 dense layer with output size of 128 on flattened $\mathbf{x}$ to obtain the policy head $\mathbf{x}_p$, now shape (128)
- apply batch normalization and relu activation on $\mathbf{x}_p$
- apply 1 dense layer with output size equals to the actin space size on $\mathbf{x}_p$, now shape $(|\mathcal{A}^a|)$
- output the policy head as the policy prediction $\mathbf{p}$

The Dynamics Function

The dynamics function g is parametrized as follows:

- input hidden state $\mathbf{x}$ of shape $(H, W, 32)$, action a as an integer
- encode a as action planes of shape $(H, W, |\mathcal{A}^a|)$ (same procedure as 4.3.4)
- stack $\mathbf{x}$ on top of the encoded action, now shape $(H, W, 32 + |\mathcal{A}^a|)$
- apply a 2-D padded convolution on ψ, with kernel size 1 by 1, 32 channels, now shape $(H, W, 32)$
- apply 1 residual block with 32 channels on $\mathbf{x}$ to obtain the hidden state head $\mathbf{x}_s$
- apply 1 residual block with 32 channels on the hidden state head $\mathbf{x}_s$
- output the hidden state head $\mathbf{x}_s$ as the next hidden state $\mathbf{x}'$
- apply 1 dense layer with output size of 128 on $\mathbf{x}$ to obtain the reward head $\mathbf{x}_r$, now shape of (128)
- apply batch normalization and relu activation on $\mathbf{x}_r$
- apply 1 dense layer with output size of $|Z|$ on $\mathbf{x}_r$, now shape of $(|Z|)$
- output reward head $\mathbf{x}_r$ as the reward prediction $\hat{r}$

The Projection Function

The projection function ϱ is simply one residual block.

Network Training

In the loss function (4.3.8), we use $c^v = 0.25$, $c^s = 2.0$, and $c^{L_2} = 1.0 \times 10^{-4}$. We use a batch size of 1024, a learning rate of 1.0×10^{-2}, and a global norm clipping of 5.0. We perform gradient updates with samples four times of the number of step samples generated each training step. For example, if we have 30 training workers, each with 16 environments, and each performs 100 environment steps per training step, then the total number of step samples generated is $30 * 16 * 100 = 48000$. This means we update the gradient using $4 * 48000 = 192000$ sampled training targets from the replay buffer. That is $\frac{192000}{1024} \approx 188$ gradient updates from mini-batches of size 1024 per training step. We use a target network similar to DQN (2.8.2) to smooth gradient updates, and we overwrites the target network every other 500 gradient updates.

Worker	c_1	c_2	Temperature	Dirichlet Noise	Simulations
Training	2.25	19652	1.0	0.2	25
Reanalyze	1.75	19652	-	0.2	50
Testing	1.75	19652	0.25	0.1	40

Table 5.1: Planner configurations. Reanalyze workers do not sample actions to act so the temperature parameter does not affect their behavior. c_1 is the exploration constant in the PUCT formula form 2.7. A greater c_1 favors the less visited actions more. c_2 is also an exploration constant in the PUCT formula. We use the same value as [50] because we find it sufficient to tune c_1 to balance exploration. The *temperature* controls how action is selected from the distribution of action visit counts from the root nodes. A temperature of 0 means select most visited actions at the root nodes. A temperature of ∞ means select actions at random. The *dirichlet noise* controls the exploitation noise added to the actions at the root nodes. A greater dirichlet noise adds more prior probabilities to less explored actions. The *simulations* is the number of simulations the planner performs at each timestep 2.3. The training workers favor exploration and generate data quickly with less simulations. The testing workers favor exploitation and spend more time on simulations to get better average return. The reanalyze workers do not need to interact with the environment so they spend more time performing simulations.

5.1.3 Planners Configurations

Table 5.1 shows the configurations of planners from different type of workers.

5.1.4 Driver Configuration

?? We use 30 training workers, each with 16 copies of the environment. For every training step, each training worker performs 100 environment steps. Freeway environment has much longer episodes, so each training worker uses 2 environments and performs 2500 environment steps per training step. We use 1 testing worker with 1 copy of the environment. For every 10 training steps, the testing worker collects 10 trajectories and logs the average return of the trajectories. We do not use the reanalyze except for the experiments in 5.4.1. All workers sync with the latest neural network parameters per 10 training steps.

5.2 MinAtar Games vs PPO

We run MooZi using four environments in MinAtar [61]. All environments produce frames of resolution 10×10, with four to seven environment channels C_e. In **Breakout**, the player moves a paddle left or right on the bottom of the screen to bounce a ball to hit the bricks. Reward is $+1$ for each brick broken and 0 in all other situations. The game ends when the paddle fails to catch the ball. In **Space Invaders**, the player controls a cannon that can move left, move right, or fire the cannon. A cluster of enemies move across the screen and fire at the player. Reward is

+1 for each enemy hit by the player and 0 in all other situations. The game ends when the player is hit by an enemy's bullet. In **Freeway**, the player starts at the bottom of the screen, moves up or down once every three frames to travel across a road with traffic. Cars are spawn randomly and travel horizontally across the screen, and when they hit the player the player is moved back to its starting position. Reward is +1 for each time the player successfully travel across the road, and 0 in all other situations. The game ends after 2500 timesteps. In **Asterix**, the player moves in four directions, and enemies and treasures spawn randomly on the edges. Reward is +1 if the player obtains a treasure and 0 in all other situations. The game ends when the player bump into an enemy. Enemies have different speed indicated by the color of their tail. MuZero is reported to have much better performance in more deterministic environments [43]. To compare MooZi in more deterministic environments, we compare the results reported by Gymnax, in which algorithms are evaluated in environments with no sticky action. We discuss this difference in section 5.3. Gymnax benchmarked the performance of PPO with three set of hyper-parameters in each of these four games [46, 52]. We compare our results with the best performing PPO in each of these games. The average return uses the planner setting of a test worker (5.1.3). Figure 5.1 shows the result comparisons. *... Do I need to show what the environments look like? Those will be static images so the readers will be clueless still ...*

5.3 Sticky Actions in MinAtar

MinAtar environments have a default sticky action probabilities of 10%. This means one out of ten timesteps, the environment uses the last taken action to step the environment instead of using the agent's output action. For example, at time t, the agent outputs action a_t = MoveLeft and moves to the left. At time $t+1$, the agent outputs action a_{t+1} = MoveRight. However, ~~this time~~, the environment applies the sticky action and overrides the action a_{t+1} = MoveLeft, and the agent moves to the left even further. The presence of a non-zero sticky action probability adds stochasticity to environments and changes the set of optimal polices. For example, in Space Invaders, if sticky action probability is 0, then the agent can move away from enemy bullets one frame before the agent is about to get hit. However, with a sticky action probability of 10%, moving away one frame in advance means there's a 10% chance that the agent will die, and moving away two frames in advance means there's a $(10\% * 10\%) = 1\%$ chance that the agent will die two frames later. We observe that a MooZi agent trained in Space Invaders with a sticky actions probability moves away from an enemy bullet right after the bullet is visible on the screen. Young and Tian shipped the MinAtar environments with four algorithms including two variants of DQN and two variants of Actor-Critic (AC) method. MinAtar's paper only reports Breakout results with sticky action probability of 10% and Gymnax only

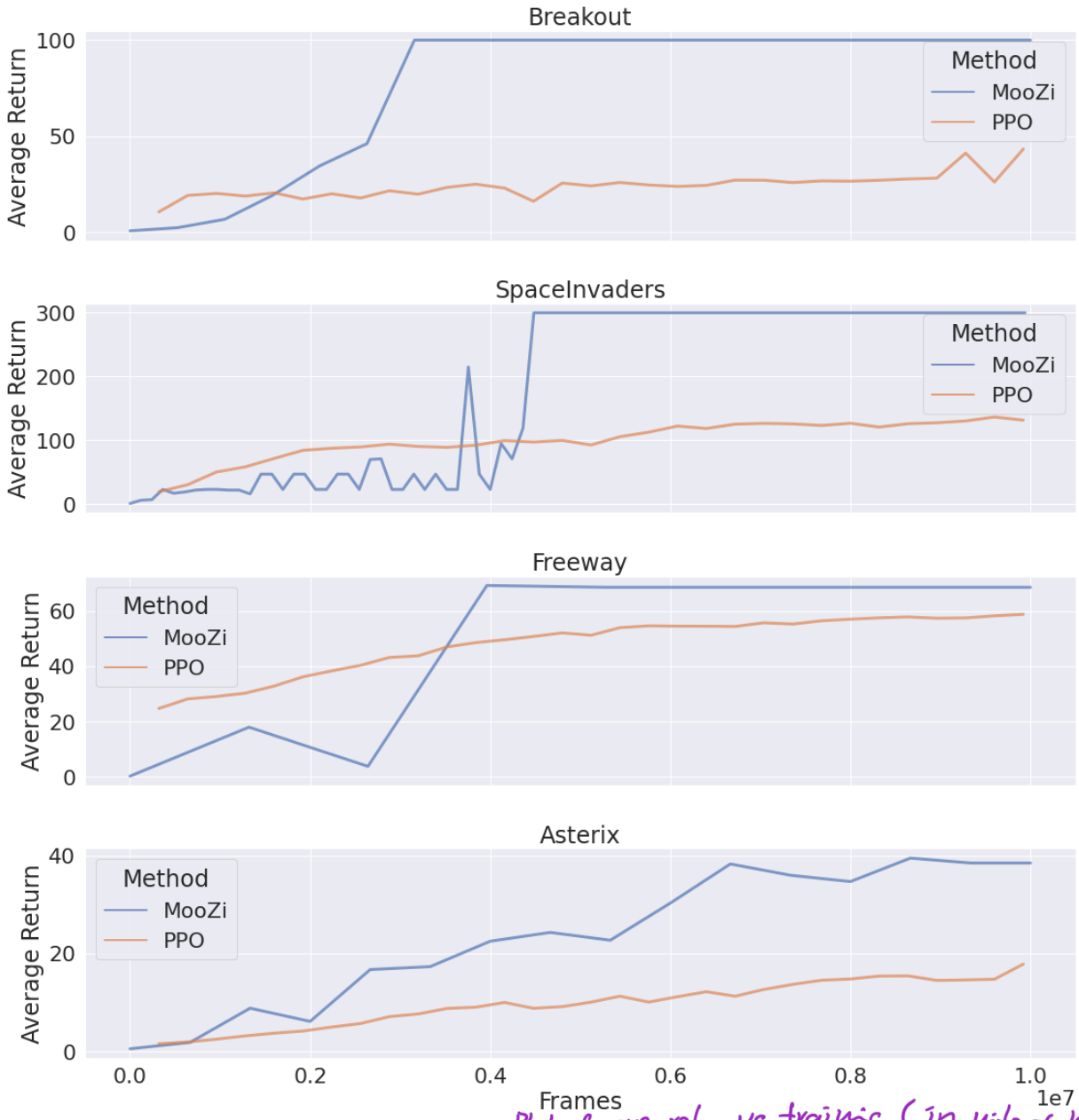

Figure 5.1: MinAtar games. MooZi vs PPO. In *Breakout*, MooZi obtained a near-optimal strategy and the paddle almost never fails to catch the ball. There is no default step limits in the environments so we cap the return at 100. Similarly in *Space Invaders*, MooZi obtained a near-optimal strategy and we cap the return at 300. In *Freeway*, MooZi also obtained a near-optimal strategy. Since episode in this environment is much longer than other environments, MooZi cycles through less training steps and save less checkpoints. Hence MooZi has much less data points in the line plot for this environment. In *Asterix*, both MooZi and PPO do not obtain an optimal strategy, but MooZi have twice as much average return as PPO.

reports Breakout results without sticky action probability. We compare with MinAtar and Gymnax in their respective testing environments using the same MooZi agent configuration. Figure 5.2 shows the comparison.

5.4 Search Budget and Testing Strength

We use the trained MooZi model in Space Invaders environment to evaluate its strength using different number of simulations. We only use model checkpoints from the first 3 million environment frames because after that the agent behaviors optimally even just using the prior to act. For each of these checkpoints, we run a testing worker to collect 30 episodes and we calculate the average return of these episodes. The testing worker uses the same planner configuration as the testing worker in 5.1.3 except for the number of simulations. Figure 5.3 shows the result.

Gymnax's results were based on environments variants without sticky actions, while the original MinAtar environments have a default sticky action probability of 10%.

5.4.1 Sample Efficiency with Reanalyze

5.4.2 System Throughput Benchmark

5.4.3 Search Analysis

Q-values

Multi-step Reward

Dummy Action

5.4.4 Hidden Space Analysis

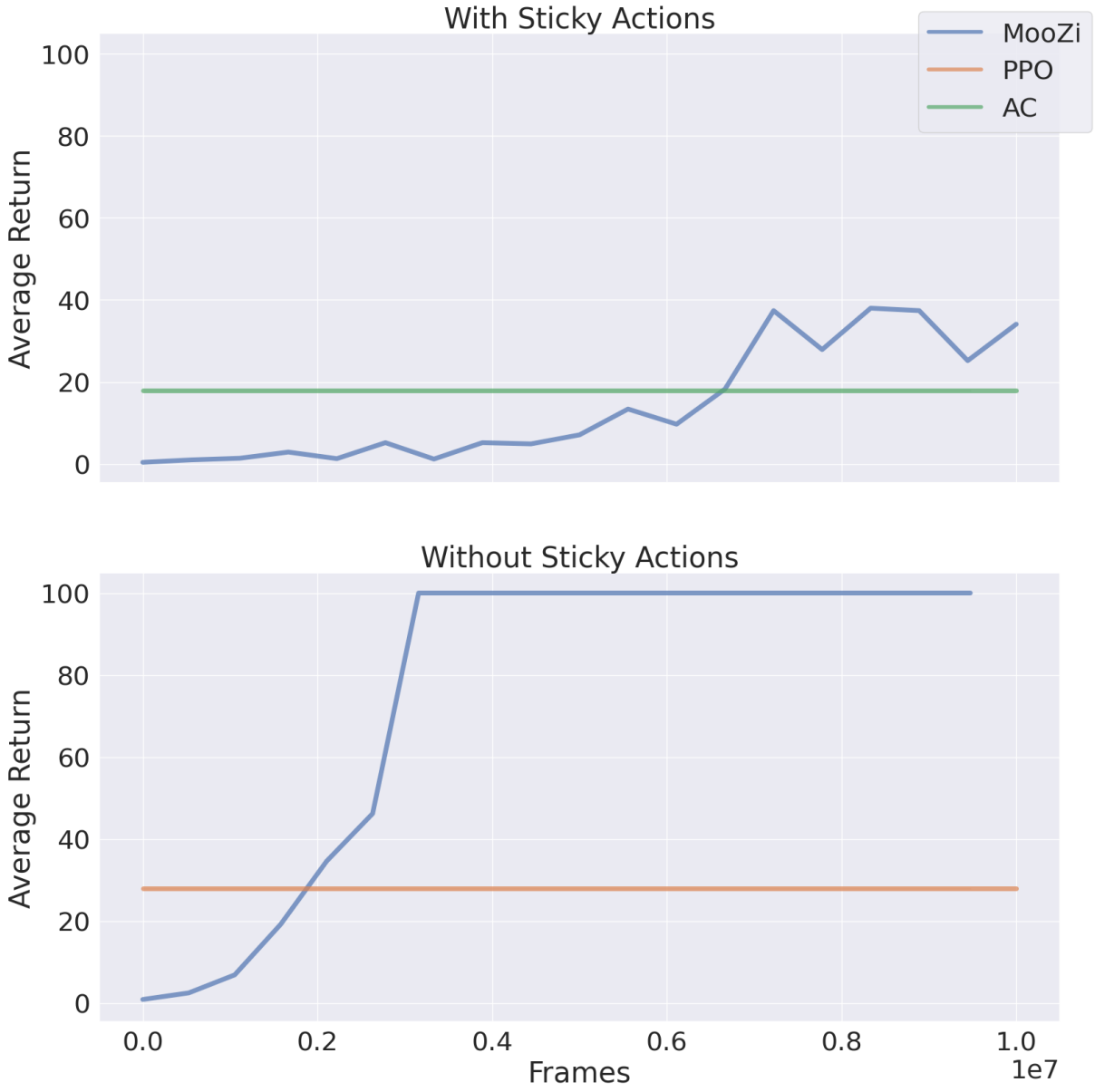

Figure 5.2: MinAtar Breakout with or without sticky actions. MooZi vs PPO vs AC. We use the final average returns of PPO and AC reported in Gymnax and MinAtar's paper respectively. In Breakout with sticky actions, MooZi learns slower and the final average return is much lower. In Breakout without sticky actions, MooZi quickly learns a near-optimal polices and never fails to return the ball. In both environments, MooZi out-performs the other algorithm.

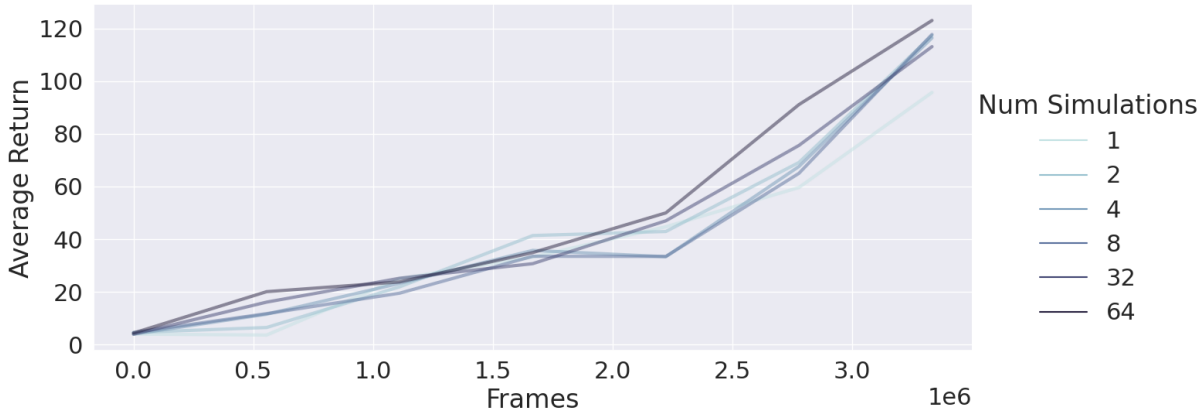

Figure 5.3: Agent strength with different number of simulations. We observe that with a greater number of simulations, the agent tends to perform better. With more training, the agent always perform better. The difference of performance due to simulation count seems to be smaller than the difference due to training. For example, with 2.5 million frames of training, acting using 64 simulations gives an average return around 70. With 3 million frames of training, acting according to the prior (1 simulation) does even better and gives an average return around 75. The prior policy, with a bit more training, quickly catches up to or even exceeds the deep search policy with less training. This finding aligns with the analysis by Hamrick et al. [19]: the planner contributes more to the algorithm by *generating better data for training the model* rather than *exploiting the model for better testing*.

Bibliography

Fix your references.

[1] Martın Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: (), p. 19.

[2] *Atari 2600*. In: *Wikipedia*. July 24, 2022. URL: https://en.wikipedia.org/w/index.php?title=Atari_2600&oldid=1100194806 (visited on 07/29/2022).

[3] Marc G. Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (June 14, 2013), pp. 253–279. ISSN: 1076-9757. DOI: 10.1613/jair.3912. arXiv: 1207.4708.

[4] *Bridge Pattern*. In: *Wikipedia*. June 27, 2022. URL: https://en.wikipedia.org/w/index.php?title=Bridge_pattern&oldid=1095365747 (visited on 07/22/2022).

[5] Greg Brockman et al. "OpenAI Gym". June 5, 2016. DOI: 10.48550/arXiv.1606.01540. arXiv: 1606.01540 [cs].

[6] Albin Cassirer et al. *Reverb: A Framework For Experience Replay*. Feb. 9, 2021. arXiv: 2102.04736 [cs]. URL: http://arxiv.org/abs/2102.04736 (visited on 07/30/2022).

[7] Guillaume Chaslot et al. "Monte-Carlo Tree Search: A New Framework for Game AI". In: (2008), p. 2.

[8] Rémi Coulom. "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search". In: *Computers and Games*. Ed. by H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers. Red. by David Hutchison et al. Vol. 4630. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 72–83. ISBN: 978-3-540-75537-1 978-3-540-75538-8. DOI: 10.1007/978-3-540-75538-8_7.

[9] Ivo Danihelka et al. "POLICY IMPROVEMENT BY PLANNING WITH GUMBEL". In: (2022), p. 22.

[10] Joery A. de Vries et al. "Visualizing MuZero Models". Mar. 3, 2021. arXiv: 2102.12924 [cs, stat]. URL: http://arxiv.org/abs/2102.12924 (visited on 10/28/2021).

[11] *Dm_env: The DeepMind RL Environment API*. DeepMind, June 7, 2022. URL: https://github.com/deepmind/dm_env (visited on 06/09/2022).

Try to find the full published papers, whenever possible.

[12] Werner Duvaud and Aurèle Hainaut. *MuZero General*. July 21, 2022. URL: https://github.com/werner-duvaud/muzero-general (visited on 07/22/2022).

[13] Lasse Espeholt et al. *IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures*. June 28, 2018. DOI: 10.48550/arXiv.1802.01561. arXiv: 1802.01561 [cs].

[14] Lasse Espeholt et al. "SEED RL: Scalable and Efficient Deep-RL with Accelerated Central Inference". Feb. 11, 2020. arXiv: 1910.06591 [cs, stat]. URL: http://arxiv.org/abs/1910.06591 (visited on 09/18/2021).

[15] Roy Frostig, Matthew James Johnson, and Chris Leary. "Compiling Machine Learning Programs via High-Level Tracing". In: (2019), p. 3.

[16] *Futures and Promises*. In: *Wikipedia*. Aug. 2, 2022. URL: https://en.wikipedia.org/w/index.php?title=Futures_and_promises&oldid=1101913451 (visited on 08/11/2022).

[17] Sylvain Gelly et al. "Modification of UCT with Patterns in Monte-Carlo Go". In: (2006). URL: http://citeseerx.ist.psu.edu/viewdoc/citations?doi=10.1.1.96.7727 (visited on 06/03/2022).

[18] *Haiku: Sonnet for JAX*. In collab. with Tom Hennigan et al. Version 0.0.3. DeepMind, 2020. URL: http://github.com/deepmind/dm-haiku.

[19] Jessica B Hamrick et al. "ON THE ROLE OF PLANNING IN MODEL-BASED DEEP REINFORCEMENT LEARNING". In: (2021), p. 37.

[20] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (July 1968), pp. 100–107. ISSN: 2168-2887. DOI: 10.1109/TSSC.1968.300136.

[21] Hado Hasselt. "Double Q-learning". In: *Advances in Neural Information Processing Systems*. Vol. 23. Curran Associates, Inc., 2010. URL: https://proceedings.neurips.cc/paper/2010/hash/091d584fced301b442654dd8c23b3fc9-Abstract.html (visited on 07/24/2022).

[22] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI'16. Phoenix, Arizona: AAAI Press, Feb. 12, 2016, pp. 2094–2100.

[23] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Las Vegas, NV, USA: IEEE, June 2016, pp. 770–778. ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.90.