ATML Report

Jonatan Bella *jonatan.bella@usi.ch*
Alessia Berarducci *alessia.berarducci@usi.ch*
Jonas Knupp *jonas.knupp@usi.ch*
Tobias Erbacher *tobias.erbacher@usi.ch*

Abstract

In this project we analyze the claims of Mahankali et al. (2024a) in the paper Random Latent Exploration for Deep Reinforcement Learning[1] which introduces a new technique to explore the state space and thus yield better overall agent scores. Moreover, we intend to reproduce the results obtained by the authors, expand on the findings and verify that the results are not cherry-picked. <write the main findings of our project> This report is prepared as part of the course project in *Advanced Topics in Machine Learning* at *Università della Svizzera italiana* in the autumn semester of 2024. @ Update once we have all the findings

1 Introduction

The paper we investigate deals with the problem of motivating an agent in a high-dimensional state space to explore the environment more exhaustively during training and thereby find non-obvious trajectories that can lead to higher long-term rewards for both discrete and continuous action spaces. The paper compares the new *Random Latent Exploration* (RLE) technique to standard *Proximal Policy Optimization* (PPO, see Schulman et al. (2017)), *NoisyNet* (see Fortunato et al. (2018)) and *Random Network Distillation* (RND, see Burda et al. (2019)).

Exploration is one of the major challenges of Reinforcement Learning (RL) and its techniques can generally be divided into two categories: Noise-based exploration and bonus-based exploration. There are advantages and disadvantages for both of them but we always have to keep in mind that always choosing the highest short-term (local) reward does not necessarily also yield the highest long-term (global) reward. E.g., picture the environment shown in figure 1a. Here, the agent starting in the blue state will always choose the small reward of 1, i.e. going right, then move back to the initial state, then move right, and so on, dithering between these two states, instead of accepting to collect a reward of 0 by going left in order to be able to collect the much higher reward of 100 in the following step.

Mahankali et al. (2024a) use RND to represent bonus-based exploration, NoisyNet to represent noise-based exploration and standard PPO as the baseline benchmark.

(a) Pure rewards. (b) Rewards with randomization (example).

Figure 1: An exemplary environment consisting of four state where transitions can occur between neighboring states. The blue node is the initial state and the numbers are the rewards.

[1] Please note that there exists an older version of this paper by Mahankali et al. (2024b) marked in the references with "(OLD VERSION)" which was provided by the course instructors. Only where there are significant differences we will refer to the new version.

2 Scope of reproducibility

The main quantifiable claims made by Mahankali et al. (2024a) consist of the following: @Tobias Move untested claims to appendix (make sure that numbering its sensible)

1. In the abstract, Mahankali et al. (2024a) claim "[...] RLE exhibits higher overall scores across all of the tasks than other approaches, including action-noise and randomized value exploration", which they later specify for both "[...] discrete and continuous control tasks." They refer to the tasks being the Atari, IssacGym and FourRoom environments. The benchmark approaches are standard PPO, NoisyNet and RND.

2. Moreover, "[...] introducting randomness to rewards influences the [... agent] to produce diverse behaviors" (Mahankali et al., 2024a) when this randomness is used to condition the policy and value networks, which manifests itself in an incentive to explore random (and thus in aggregate lager) parts of the state space.

3. For the discrete-action discrete-states FourRoom environment absent of a goal reward, the authors claim that PPO's state visitation centers around the initial room (top right) concluding that action-noise methods cannot conduct deep exploration whereas RLE, NoisyNet and RND well reach across the four rooms and thus perform deep exploration.

4. For the more challenging discrete-action continuous-states[2] Atari environment, the authors claim that, in the majority of games, RLE achieves a higher IQM of the human-normalized score with a probability of 67%, 72% and 71%, compared to PPO, NoisyNet and RND, respectively.[3]

5. In the continuous-action continuous-states IsaacGym environment, the authors claim that, with a probability of around 83% for PPO, ca. 66% for NoisyNet and roughly 65% for RND, RLE improves performance in all of their selected IsaacGym tasks.

Our findings regarding these claims will be presented in section 4 Check link before submission. Moreover, Mahankali et al. (2024a) performed ablation studies and claim:

6. RLE performance improves over PPO irrespective of the distribution used to sample the 8-dim random latent vector.

7. RLE performance compared to PPO is invariant under a change in the dimensionality of the random latent vector.

8. RLE performance deteriorates if the policy and value networks are not conditioned on the random latent vector.

9. RLE performance improves slightly if a learning feature network is used in comparison to when the feature network has constant weights when evaluated on the Atari games.

10. RLE performance is insensitive to the feature extractor architecture.

Unfortunately, due to time constraints we are not able to test all of these claims, instead we focus on claims 1, 2, 3, and 6 in this study.

[2]While the input to the action sampler is a discrete $84 \times 84 \times 4$ tensor representing the pixels it perceives, in practice Atari games are considered to be continuous-state environments due to the high-dimensional state space and visual complexity of each frame.

[3]The authors mention that for the Montezuma's Revenge game RLE underperforms likely due to RLE not factoring in bonus-based exploration.

3 Methodology

Random Latent Exploration — The new idea that the authors Mahankali et al. (2024a) present is to augment the reward function by adding a randomized term which incentivizes the agent to explore a larger portion of the state space. In particular, we will call this term $F(s, z)$ or intrinsic reward function, where $s \in \mathcal{S}$ is any state of the state space $\mathcal{S}$, and $\mathbf{z} \in \mathbb{R}^d$ is a d-dimensional vector sampled from a given distribution $P_{\mathbf{z}}$. The authors Mahankali et al. (2024a) claim that RLE's efficiency is not significantly affected by the choice of $P_{\mathbf{z}}$. If at time step $t \in \{0, 1, 2, ..., T\}$ the agent in state s_t takes action $a_t \sim \pi(.|s_t)$ from policy π, then it will obtain the task reward $r(s_t, a_t)$. In RLE, we train a so-called latent-conditioned policy network $\pi(.|s, \mathbf{z})$ and latent-conditioned value network $V^\pi(s, \mathbf{z})$ to estimate and then maximize the expected sum of rewards from a given state, aware of the random term z:

$$V^\pi(s, \mathbf{z}) \approx \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \left(r(s_t, a_t) + F(s_{t+1}, \mathbf{z}) \right) \right] \tag{1}$$

In equation 1, γ describes the discount factor and $F(s_{t+1}, \mathbf{z}) = \boldsymbol{\phi}(s) \cdot \mathbf{z}$ where $\boldsymbol{\phi}(s) : \mathcal{S} \to \mathbb{R}^d$ is a feature extraction network. The feature network is updated as a linear combination of the old feature network's weights and the weights of the value network[4]. Both $\boldsymbol{\phi}$ and $\mathbf{z}$ are d-dimensional vectors. Note that although equation 1 is presented by Mahankali et al. (2024a), in the implementation they introduce an intrinsic and extrinsic reward coefficient. In the experiments they choose intrinsic reward coefficient $\ll$ extrinsic reward coefficient. Thus, they are not directly optimizing equation 1 in the experiments. Furthermore, on page 3 the authors state that $\boldsymbol{z}$ "is resampled at the start of each trajectory". However, in the pseudocode on page 15 the authors note that $\boldsymbol{z}$ is resampled at the end of each trajectory or after a certain number of steps has passed. Since the authors implemented the latter, more general, case we adopt the latter approach for our experiments. The authors split the critic's head into two: one head predicts the intrinsic value function, the other head predicts the extrinsic value function. Figure 12 depicts how the action logits, intrinsic value, and extrinsic value for a given observation and $\boldsymbol{z}$ is calculated. Note the residual connection from the first element-wise addition to the second element-wise addition. There is no mention of this residual connection in the paper. However, it is present in the provided code for the Atari environment. To continue the example from the introduction, in figure 1b the agent would prioritize to go left as this promises the greater reward. It is not hard to imagine that for higher dimensional environments, every time we start a new trajectory and correspondingly sample a new z, the agent will explore a different region of the state space and thus we can discover non-obvious paths to maximize the rewards. For the detailed pseudocode of this algorithm, see Appendix A (check link before submmission).

A description of the other three algorithms can be found in Appendix B (PPO), Appendix C (NoisyNet) and Appendix D (RND). Moreover, since the architecture of the neural networks we are using, e.g. for the feature extractor, value function approximation and policy, differ for each environment and every algorithm, we will explain them in section 3.4 in detail. (check all four links before submission)

3.1 Hyperparameters

For the FourRoom experiments the hyperparameters stated in Mahankali et al. (2024a) were used. However, Mahankali et al. (2024a) did not provide values for all available hyperparameters. For the missing hyperparameters we used the following values, that seemed reasonable to us although we did not spend a lot of time finetuning them:

@Alessia: We need your input here for Atari: Are all of the hyperparameters you used the ones from the RLE paper, or did you change/add/remove some of them, if yes which ones, what are their values and did

[4]In the pseudocode algorithm in line 19 on page 25, in the mathematical formulation, the authors state that the feature network is updated as a linear combination of the old feature network's weights and the policy network's weights. We contacted the authors, and they confirmed that it should read $\phi \leftarrow \tau \cdot V^\phi + (1 - \tau) \cdot \phi$ (i.e., the value network should be used, not the policy network). However, they state it would be interesting to explore which network should be used to update the feature network.

Table 1: Hyperparameters not stated by Mahankali et al. (2024a) for the FourRoom experiments.

Hyperparameter	**Value**
Extrinsic Reward Coefficient	1
Observation Normalization Iterations	1
Discount Rate	0.99
Intrinsic Discount Rate	0.99
Feature Network Update Rate τ	0.005
Learning Rate	0.001

you finetune them (how, what was the tuning range, how many different values did you try out)? Certainly, the z resampling frequency was changed.

@Jonathan: We need your input here for IsaacLab: Are all of the hyperparameters you used the ones from the RLE paper, or did you change/add/remove some of them, if yes which ones, what are their values and did you finetune them (how, what was the tuning range, how many different values did you try out)?

3.2 Experimental setup and code

The code for our implementations can be found in GitHub[5]. There are three types of environments in use on which we run our RL algorithms which are depicted in figure 2.

The authors of the paper at hand published code in a GitHub repository[6]. This repository contains code for the Atari and IsaacGym[7] environments with the 4 algorithms (RLE, PPO, RND, NoisyNet).

@Alessia: The information from the following two sections need to be integrated in the corresponding subsubsections

In general, after coding the scripts, we uploaded the files to Google Colab, of which we used/tested the available TPU/GPU for the training (for more detail see section 3.5 check link before submission), and linked a Google Drive to save the models, as well as a Weights & Biases workspace for a live overview of the results and various parameters while the models were still training. Regarding documentation we were guided mostly by the paper at hand and the official environment documentation, as well as other papers that we found along the way.

3.2.1 Atari

@Tobias extend description of experiments etc.

@Who it may concern: trajectory length longer in RLE, compensating lower rewards?

Atari represents "discrete action space deep RL benchmark". The human normalized scores are computed according to formula: Agent57: Outperforming the Atari Human Benchmark and the human scores are taking from

Moreover we have an example of the Atari environments in figure 2b, namely the Alien-v5 game. In this environment, the agent perceives the 4 most recent 84×84 grayscale frames.

Do we need to explain the environments as well? Alessia explains Atari stuff Atari was run 5 times per game.

[5]https://github.com/jonupp/adv_topics_ml_repl_chal Change github
[6]https://github.com/Improbable-AI/random-latent-exploration
[7]Please note that the IsaacGym environment is deprecated and we are using IsaacLab instead, however we will refer to it as IsaacGym for reasons of continuity.

3.2.2 FourRoom

Figure 2a shows the FourRoom environment where we have $50 \times 50 + 4$ states[8] which are divided into four rooms of size 25×25 and 4 holes in the walls located at positions 10 horizontally and 5 vertically, starting the count at 1 from the inner corners. Exploring the state space, beginning from the initial state marked in red, becomes a deep exploration task due to the holes being only 1 "pixel" wide. However, if we want to incentivize the agent to find a reward, then we can put it at the goal position marked in green. The FourRoom environment was implemented such that we can pass a flag to the environment's constructor to choose whether the environment is reward-free or has the goal in the bottom-left corner. At every step, the agent can move one step vertically or horizontally, but if it would run into a wall it chooses to remain in place. Mahankali et al. (2024a) refer to Sutton et al. (1999) when introducing the FourRoom environment. Sutton et al. (1999) use action stochasticity. We asked Mr. Mahankali whether they also used action stochasticity, and they stated that they did not use action stochasticity in their experiments.

The authors did not provide any code for the FourRoom environment, so this environment was implemented by us using the gymnasium library introduced by Towers et al. (2024) together with our adaptation of the code for PPO, NoisyNet and RLE algorithms from the Atari environment, while we adapted the RLE code from the IsaacGym environment. A wrapper to record the state visitation counts per environment was also implemented. Before we adapted RLE from the existing implementation for the Atari environment, we implemented it based on the description in Mahankali et al. (2024a). However, the paper lacked a lot of details present in the Atari code (e.g., split critic's head, residual connection) so that we decided to proceed with the adapted code to have consistent results over the different environments.

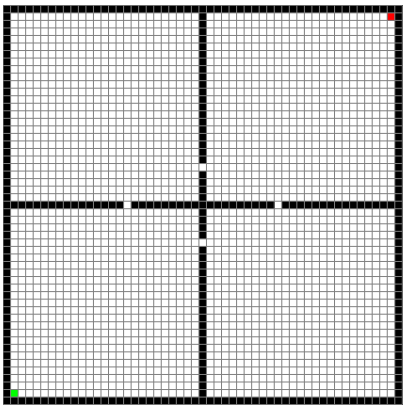
First we investigate how RLE performs in a reward-free setting in comparison to PPO, NoisyNet, and RND. Mahankali et al. (2024a) claims that RLE shows good explorative behavior based on a single heatmap of state visitation counts. To assess whether this result is expected or exceptional we trained each algorithm 20 times. Since it is not reasonable to compare 20 state visitation count heatmaps we used the Shannon entropy, introduced by Shannon (1948), of the state visitation counts as a proxy for how well an agent explored the environment. With increasing entropy the distribution of the state visitation counts becomes more uniform. Thus, the higher the entropy the better the explorative behavior.

We also ran RLE, PPO, NoisyNet, and RND in the FourRoom environment with a goal to see how RLE compares to the other algorithms in an environment with a reward. Similar to the reward-free setting, we run each algorithm 20 times on different seeds. However, instead of assessing the performance of the algorithms using the Shannon entropy, we use the average game score. The game score is the average undiscounted return over the last 128 episodes. The average game score for a certain algorithm is the average of the game scores in the different runs for this algorithm.

Furthermore, it was explored if the intrinsic value function really guides the generation of diverse trajectories in a reward-free environment. To assess this, the intrinsic value function for four latent vectors z and four trajectories generated using the same latent vector z were saved at the end of each experiment.

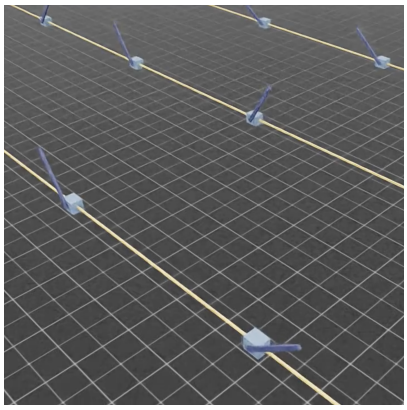
We also tested to what degree RLE is indifferent to the distribution of the latent vector. To test this, we ran RLE with different latent vector distributions. The following distributions were considered: standard normal distribution, standard uniform distribution, Von-Mises distribution with $\mu = 0$ and $\kappa = 0.3$, and the exponential distribution with $\lambda = 0.3$. We ran 20 experiments for each latent vector distribution and recorded the state visitation entropy. The same hyperparameters were used as before. Note that all latent vectors were normalized using the L2-norm, so that all latent vectors are on a unit sphere around the origin. This ensures that all intrinsic rewards are in the interval [-1,1]. Mahankali et al. (2024a) performed a similar ablation study for the Atari environment. However, they did not normalize the latent vectors for all distributions. They only applied normalization for the normal distribution which they then denoted as "Sphere". They then compare the "Sphere" to the unnormalized uniform distribution and the unnormalized normal distribution. We normalized all distributions because the extrinsic reward coefficient and the intrinsic reward coefficient hyperparameters already control the weighting between the intrinsic and extrinsic rewards, so we wanted the intrinsic rewards to be in the interval [-1,1].

[8] In Mahankali et al. (2024a), the authors claim that there are 50×50 states. We contacted the authors, and they confirmed that it should read $50 \times 50 + 4$ states.



(a) The FourRoom environment is a grid world, limited by walls, with the initial state is on the top right and the goal state where the agent receives a reward (optional) is on the bottom left.

(b) The Alien-v5 environment is an example of the Atari games consisting of a maze filled with 'eggs' to destroy while being hunted by aliens. A flamethrower or occasional power-up may be used to scare the aliens.



(c) The CartPole environment is an example of the IsaacGym environment.
What does the agent actually perceive?

Figure 2: Examples of the environments in use.

3.2.3 IsaacGym

@Jonathan explain environment and experiments (NOT RESULT)

3.2.4 Adaptive Von Mises-Fisher (Simple)

@Jonas finish Mahankali et al. (2024a) suggest it might be worthwhile to explore using a latent vector distribution that changes during the training. Our hypothesis is that at the start of the training the latent vector distribution $P_{\mathbf{z}}$ should give diverse $\mathbf{z}$ since the agent should explore the environment. However, with increasing training progress, the agent should put less emphasis on exploration. We use the Von Mises-Fisher distribution with the parameters $\boldsymbol{\mu}$, the mean direction, and κ, the concentration to model $P_{\mathbf{z}}$. The idea is to slowly change $\boldsymbol{\mu}$ during the training, so that it points to a point in the space such that $P_{\mathbf{z}}$ samples $\mathbf{z}$ values that yield high returns. Furthermore, κ should be slowly increased during the training to increase the probability to sample $\mathbf{z}$ close to $\boldsymbol{\mu}$. Since PyTorch does not provide an implementation to sample from this distribution, we used the implementation described by Pinzón & Jung (2023) that is based on rejection sampling. We maintain a queue of $\mathbf{z}$ values with good episodic return and a queue with the respective returns. After every finished episode, the queues are updated if necessary. We then update $\boldsymbol{\mu}$ as a linear combination of the $\mathbf{z}$ values in the success memory. The weighting is done using the returns in the returns memory. The κ should be large if the model has converged to a $\boldsymbol{\mu}$ that reliably gives $\mathbf{z}$ values with high return. Thus, κ is updated by linearly interpolation between 0 and 30. The weight is the average pairwise cosine similarity between the $\mathbf{z}$ values in the success memory.

3.2.5 Adaptive Von Mises-Fisher (LSTM)

@Jonathan

3.3 Computational requirements

All FourRoom experiments were performed on an Apple M2 Pro with 16 GB RAM. Every experiment was conducted in the reward-free environment and in the environment with a goal. Every algorithm and every RLE variant with different latent vector distribution was run 20 times with different seeds in each of the two environments. This resulted in a total of 280 runs. It took ca. 1 minute to do a single PPO run, ca. 1 minute and 20 seconds to do a single NoisyNet run, ca. 2 minutes to do a single RLE run, and ca. 2 minutes

and 20 seconds to do a single RND run. The experiments for the different algorithms took at total time of ca. 4.4 hours. In addition, we trained RLE with three different latent vector distributions (apart from the standard normal distribution). These experiments took ca. 4 hours. Thus, the total computing time for all FourRoom experiments amounts to ca. 8.4 hours.

@Alessia: We need your input here for Atari: Did you run all of the experiments on colab (which CPU or GPU was used (name))? What are the average training times e.g. per episode or global step? How many runs did you do overall (i.e. how many compute hours did you spend on this project) for each of the algorithms? For Atari, we experimented with the CPU, T4 GPU and TPU v2-8 in Google Colab. Weights & Biases (wandb) was used for easier runtime visualization of the scores and other measures.

@Jonatan: We need your input here for Isaac: Did you run all of the experiments on your mac/other laptop or did you use colab as well (which CPU or GPU was used (name))? What are the average training times e.g. per episode or global step? How many runs did you do overall (i.e. how many compute hours did you spend on this project) for each of the algorithms?

We would like to note, however, that Mahankali et al. (2024a) had the HPC resources of the MIT Supercloud and the Lincoln Laboratory Supercomputing Center available. Within the framework of this course, we did not have the equivalent computing power to replicate all experiments in their original form and had to restrain ourselves to a mere few of them.

4 Results

all figures in appendix as first item

```
Start with a high-level overview of your results. Do your results support the main
claims of the original paper? Keep this section as factual and precise as possible,
and reserve your judgment and discussion points for the next "Discussion" section.
```

4.1 Results reproducing original paper

```
For each experiment, say (1) which claim in Section 2 it supports, and (2) if it
successfully reproduced the associated experiment in the original paper. For example,
an experiment training and evaluating a model on a dataset may support a claim that that
model outperforms some baseline. Logically group related results into subsections.
```

4.1.1 FourRoom

@Jonas

4.1.2 Atari

@Alessia

4.1.3 IsaacGym

@Jonathan

4.2 Results beyond original paper

```
Often papers don't include enough information to fully specify their experiments, so
some additional experimentation may be necessary. For example, it might be the case
that batch size was not specified, and so different batch sizes need to be evaluated to
reproduce the original results. Include the results of any additional experiments here.
Note: this won't be necessary for all reproductions.
```

4.2.1 Adaptive Von Mises-Fisher

@Jonathan

5 Discussion

Discuss claims

`Give your judgment on if your experimental results support the claims of the paper. Discuss the strengths and weaknesses of your approach - perhaps you didn't have time to run all the experiments, or perhaps you did additional experiments that further strengthened the claims in the paper.`

5.1 What was easy

@Tobias

- Atari code was ready to run. - The concept of the paper is not hard to understand (also mathematically)

`Give your judgment of what was easy to reproduce. Perhaps the author's code is clearly written and easy to run, so it was easy to verify the majority of original claims. Or, the explanation in the paper was really easy to follow and put into code.`

`Be careful not to give sweeping generalizations. Something that is easy for you might be difficult for others. Put what was easy in context and explain why it was easy (e.g. code had extensive API documentation and a lot of examples that matched experiments in papers).`

5.2 What was difficult

A major challenge we faced was evaluating RLE across three distinct environments, each presenting unique difficulties. For the FourRoom environment, no implementation was available, requiring us to develop one from scratch. We aimed to replicate the approach used by Mahankali et al. (2024a) as closely as possible to ensure comparability. The Atari environment, on the other hand, could be used without modification, but training agents demanded substantial computational resources. @Jonathan add something about Isaac here. Furthermore, in the paper, an important implementation detail, the split critic's head, is omitted. The paper itself also contains some contradicting information (Atari feature network size, condition to resample z) and minor errors (FourRoom environment description and typo in pseudocode regarding feature network update). Lastly, as RLE builds upon PPO with optimizations such as generalized advantage estimation, we had to dedicate considerable time to thoroughly understand the underlying PPO framework.

5.3 Communication with original authors

During our replication study, a number of questions, mainly regarding parameters and model/environment architecture, came up which we were unable to answer by ourselves, so we sent an email to Mr. Mahankali. and got a response a few days later. Our questions and Mr. Mahankali's answers will be provided upon request.

6 Conclusion

`Try to summarize the achievements of your project and its limits, suggesting (when appropriate) possible extensions and future works.`

Member contributions

We divided the workload among the group members as follows:

- Jonatan Bella: IsaacLab implementation, please let me know what you want to add
- Alessia Berarducci: Atari implementation, please let me know what you want to add
- Jonas Knupp: Implementation of the FourRoom environment, adaption of PPO, NoisyNet, RLE, and RND to the FourRoom environment, conducting the experiments with the FourRoom environment
- Tobias Erbacher: Report writing, project management, numpy-visualization.

However, we met regularly and assisted each other in their tasks so the split is not completely clear-cut.

References

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=H1lJJnR5Ym.

Meire Fortunato, Mohammad G. Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration. *International Conference on Machine Learning*, 2018. doi: 10.48550/arXiv.1706.10295.

Srinath V. Mahankali, Zhang-Wei Hong, Ayush Sekhari, Alexander Rakhlin, and Pulkit Agrawal. Random latent exploration for deep reinforcement learning (new version). *Forty-first International Conference on Machine Learning*, 2024a. doi: 10.48550/arXiv.2407.13755. URL https://srinathm1359.github.io/random-latent-exploration/.

Srinath V. Mahankali, Zhang-Wei Hong, Ayush Sekhari, Alexander Rakhlin, and Pulkit Agrawal. Random latent exploration for deep reinforcement learning (old version). 2024b. URL https://openreview.net/forum?id=Y9qzwNlKVU.

Carlos Pinzón and Kangsoo Jung. Fast Python sampler for the von Mises Fisher distribution. working paper or preprint, August 2023. URL https://hal.science/hal-04004568.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. 2017. doi: 10.48550/arXiv.1707.06347.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018. URL https://arxiv.org/abs/1506.02438.

C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3): 379–423, 1948. doi: 10.1002/j.1538-7305.1948.tb01338.x.

Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1):181–211, 1999. URL https://www.sciencedirect.com/science/article/pii/S0004370299000521.

Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: A standard interface for reinforcement learning environments, 2024. URL https://arxiv.org/abs/2407.17032.

A Figures

Since we could not fit all figures in eight pages, we include them here.

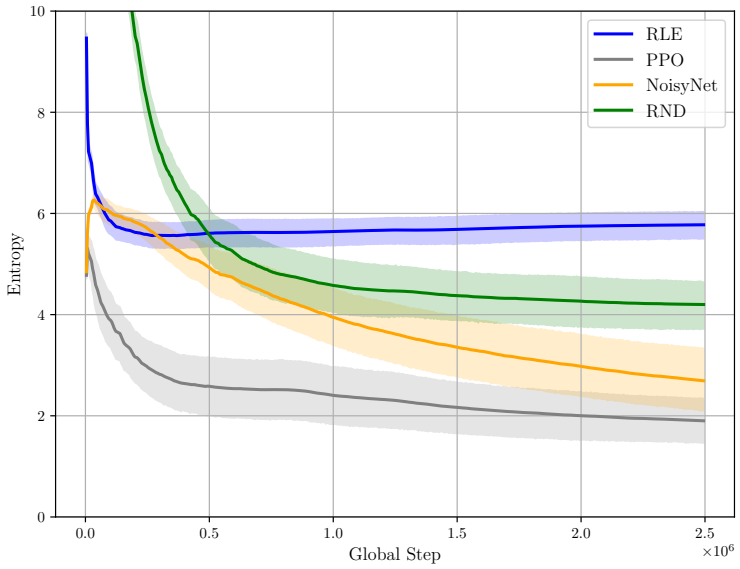


Figure 3: Mean entropy of state visitation counts over the training steps, with the shaded area representing the 95% confidence interval for different algorithms. By the end of training, RLE achieves the highest entropy and the narrowest confidence interval.

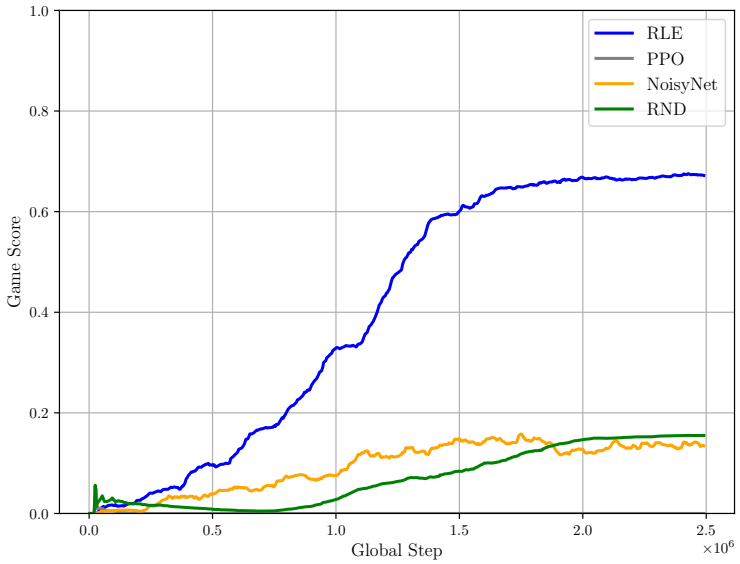


Figure 4: Mean game score of different algorithms. RLE clearly outperforms the other algorithms.

B Random Latent Exploration

The architecture of our RLE implementation can be found in figure 12:

For the RLE algorithm, the detailed pseudocode according to Mahankali et al. (2024a) is as follows:

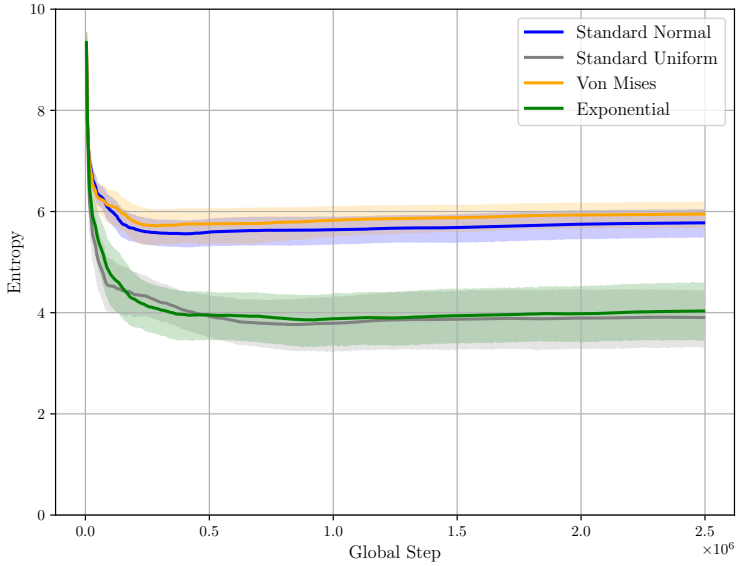


Figure 5: Mean entropy of state visitation counts over the training steps, with the shaded area representing the 95% confidence interval for RLE variants with different latent vector distributions. The standard uniform and exponential variants explore less than the other two variants.

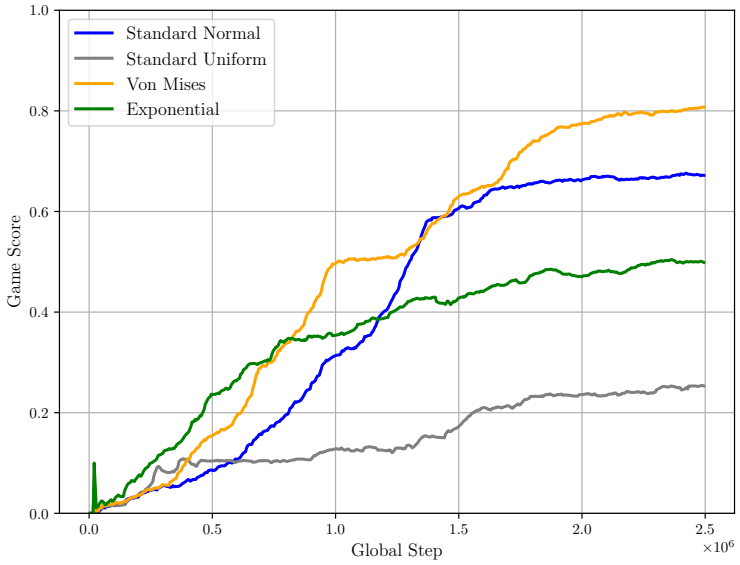


Figure 6: Mean game score of RLE variants with different latent vector distributions. The figure shows substantial differences in the mean game score across the different RLE variants.

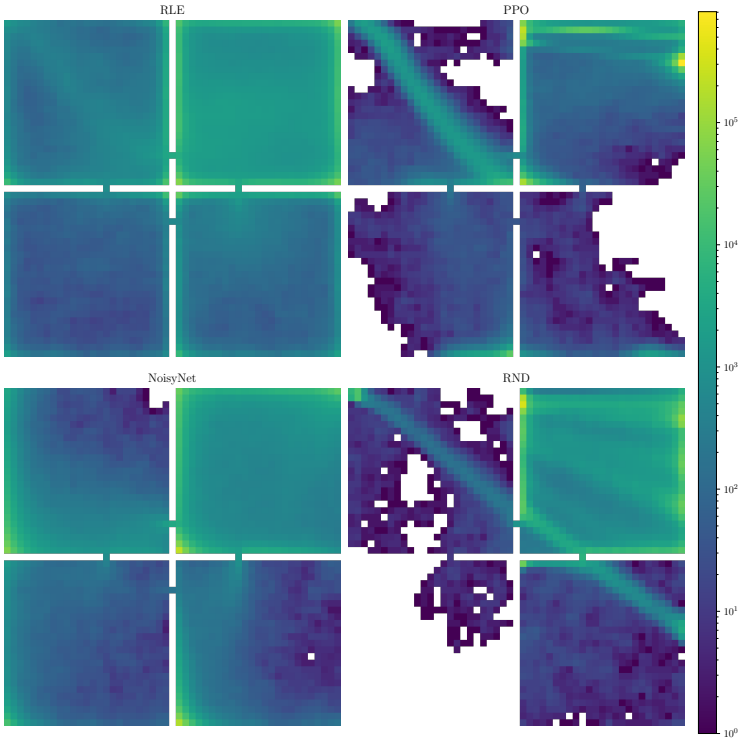


Figure 7: For each algorithm, the run with the highest entropy over the state visitation counts was selected to generate the heatmap. The heatmap illustrates the state visitation counts of all the algorithms after training for 2.5M timesteps in the reward-free environment. The start location is in the top right cell. RLE achieves the best state visitation coverage.

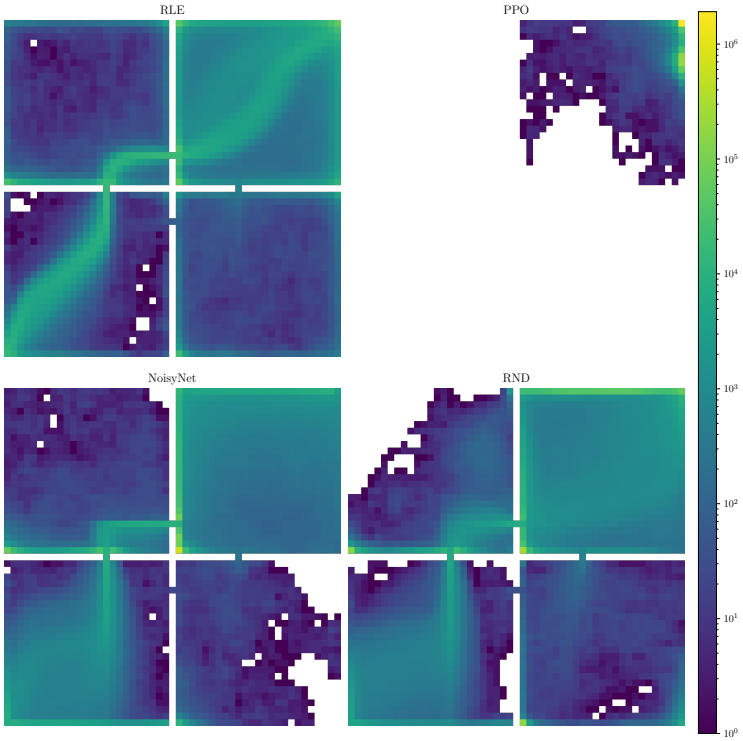


Figure 8: For each algorithm, the run with the highest game score was selected to generate the heatmap. The heatmap illustrates the state visitation counts of all the algorithms after training for 2.5M timesteps in the environment with a goal. The start location is in the top right cell and the goal location in the bottom left cell.

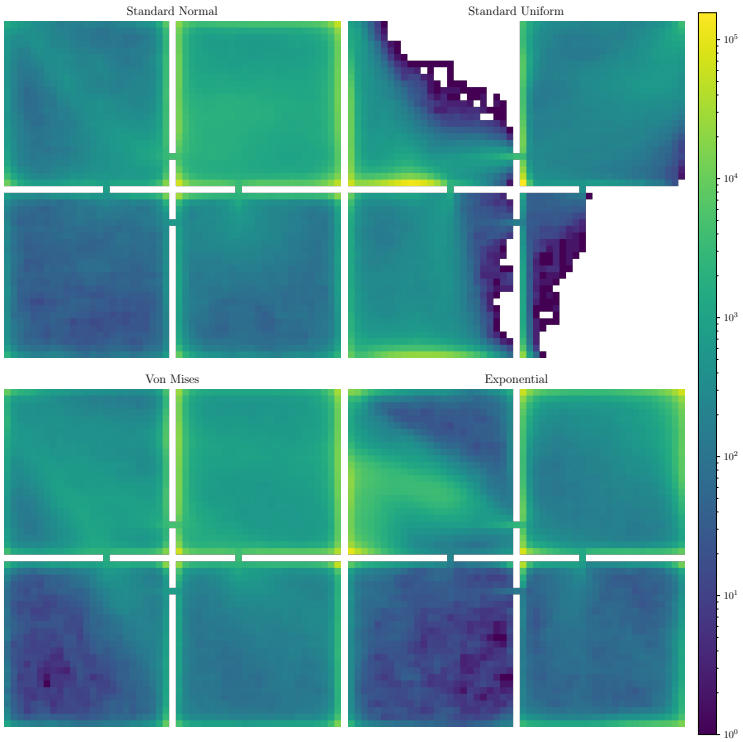


Figure 9: For each RLE variant with different latent vector distribution, the run with the highest entropy over the state visitation counts was selected to generate the heatmap. The heatmap illustrates the state visitation counts of all the RLE variants with different latent vector distribution after training for 2.5M timesteps in the reward-free environment. The RLE variants where the latent vector distribution is a standard normal and the Von Mises distribution achieve the best state visitation coverage.

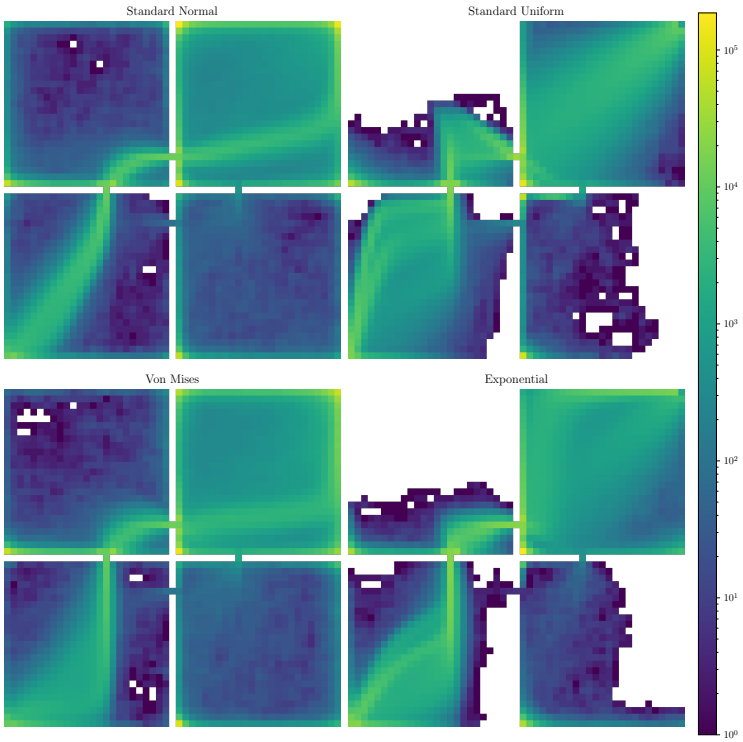


Figure 10: For each RLE variant with different latent vector distribution, the run with the highest game score was selected to generate the heatmap. The heatmap illustrates the state visitation counts of all the RLE variants with different latent vector distribution after training for 2.5M timesteps in the environment with a goal. The start location is in the top right cell and the goal location in the bottom left cell. All RLE variants reach the goal but the standard normal and Von Mises variants also explore most of the state space.

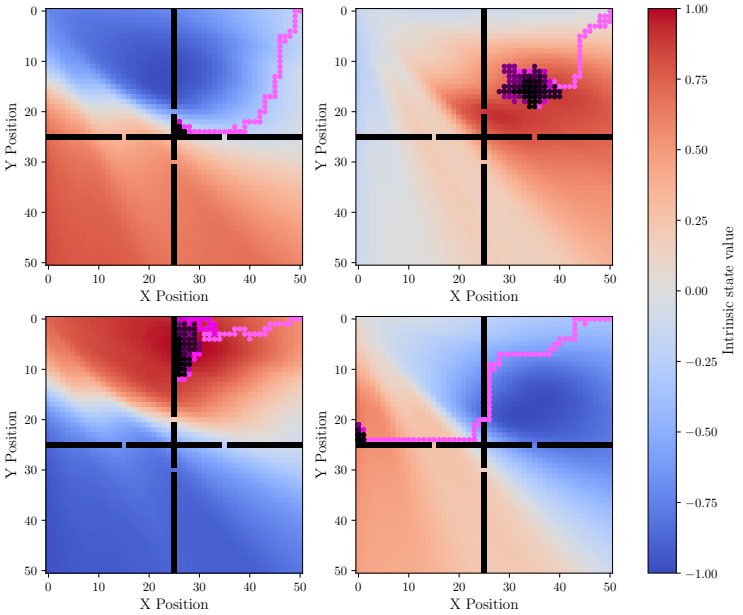


Figure 11: Trajectory and intrinsic reward function for four different **z** values. The trajectories generally tend towards regions with high intrinsic reward. The example is not cherry-picked. This behavior was observed for almost every figure we generated.

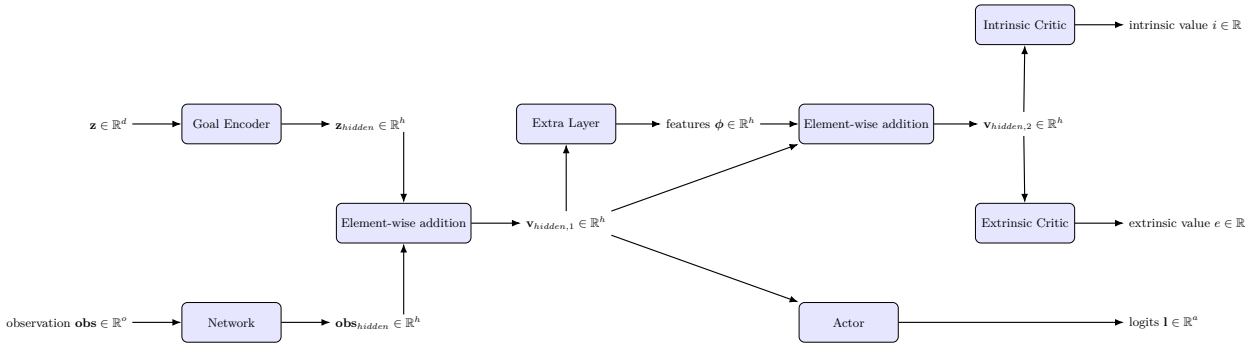


Figure 12: In RLE, the policy and critic networks are conditioned on the latent vector z. Mahankali et al. (2024a) split the critic's head in two: one head predicts the extrinsic value and the other head predicts the intrinsic value. In the vector dimension, o represents the dimension of a single observation, d represents the RLE feature dimension, and a is the cardinality of the action space.

RLE:	Detailed Pseudocode
1:	**Input:** Latent distribution ($P_{\mathbf{z}}$), number of parallel workers (N), number of steps per update (T), number of steps per sampling (S), and feature network update rate (τ).
2:	Randomly initialize a feature network (ϕ) with the same backbone architecture as the policy and value networks.
3:	Initialize running mean $\boldsymbol{\mu} = \mathbf{0}$ and standard deviation $\boldsymbol{\sigma} = \mathbf{1}$ estimates of $\boldsymbol{\phi}(s)$ over the state space.
4:	Sample an initial latent vector $\mathbf{z} \sim P_{\mathbf{z}}$ for each parallel worker.
5:	**Repeat** $(^1)$
6:	| Sample initial state s_0.
7:	| **For** $t = 0, ..., T$ **do** $(^2)$
8:	| | Take action $a_t \sim \boldsymbol{\pi}(.\|s_t, \mathbf{z})$ and transition to s_{t+1}.
9:	| | Compute feature $\mathbf{f}(s_{t+1}) = \frac{\boldsymbol{\phi}(s_{t+1}) - \boldsymbol{\mu}}{\boldsymbol{\sigma}}$.
10:	| | Compute random reward $F(s_{t+1}, \mathbf{z}) = \frac{\mathbf{f}(s_{t+1})}{\|\mathbf{f}(s_{t+1})\|} \cdot \mathbf{z}$.
11:	| | Receive reward $r_t = R(s_t, a_t) + F(s_{t+1}, \mathbf{z})$.
12:	| | **For** $i = 0, 1, ..., N-1$ **do** $(^3)$
13:	| | | **If** worker i terminated **or** S timesteps passed without resampling, **then** $(^4)$
14:	| | | | Resample sample $\mathbf{z} \sim P_{\mathbf{z}}$ for worker i.
15:	| | | $(^4)$
16:	| | $(^3)$
17:	| $(^2)$
18:	| Update policy network $\boldsymbol{\pi}$ and value network $V_{\boldsymbol{\pi}}$ with the collected trajectory $(\mathbf{z}, s_0, a_0, r_0, s_1, ..., s_T)$.
19:	| Update feature network $\boldsymbol{\phi}$ using the value network's parameters $\boldsymbol{\phi} \leftarrow \tau \cdot V^{\boldsymbol{\pi}} + (1-\tau) \cdot \boldsymbol{\phi}$.
20:	| Update $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ using the batch of collected experience.
21:	$(^1)$ **until convergence**

C Proximal Policy Optimization

The algorithm constituting the foundation of RLE is Proximal Policy Iteration (PPO), described in Schulman et al. (2017), which can itself be run to solve RL problems. It is an on-line policy gradient method where after running the policy $\boldsymbol{\pi}_{\theta_{\text{old}}}$ in parallel with N actors for $T << E$ timesteps, where E is the length of an episode, we compute the advantage estimator as proposed by Schulman et al. (2018) for each time index $t \in [0, T]$:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\,\delta_{t+1} + ... + (\gamma\lambda)^{T-t+1}\,\delta_{T-1} \tag{2}$$

where we have $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$. Here, λ is the Generalized Advantage Estimation (GAE) parameter and $V(s)$ is a learned state-value function. Afterwards, for each t the surrogate objective is computed and the parameters θ are optimized e.g. with minibatch SGD or Adam (note that we optimize by maximizing the objective). This surrogate objective takes the form:

$$L_t^{\text{CLIP+VF+S}}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{\text{CLIP}}(\theta) - c_1 L_t^{\text{VF}}(\theta) + c_2 S[\boldsymbol{\pi}_\theta](s_t) \right] \tag{3}$$

In equation 3, we have the VF coefficient c_1 and the entropy coefficient c_2, together with the (optional, to enhance exploration) entropy bonus $S[\boldsymbol{\pi}_\theta](s_t)$, which is calculated as the sum of $-p \log p$ over each of the action-probabilities p resulting from the policy distribution $\boldsymbol{\pi}_\theta$ for a state s_t, and the objectives:

$$L_t^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right] \tag{4}$$

$$L_t^{\text{VF}}(\theta) = \left(V_\theta(s_t) - V_t^{\text{targ}} \right)^2 \tag{5}$$

In equation 4 we denote the probability ratio as $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$, where $\pi_{\theta_{\text{old}}}$ is the policy before the update, and the clipping function is:

$$\text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) = \begin{cases} 1-\epsilon, & r_t(\theta) < 1-\epsilon \\ r_t(\theta), & 1-\epsilon \leq r_t(\theta) \leq 1+\epsilon \\ 1+\epsilon, & r_t(\theta) > 1+\epsilon \end{cases} \tag{6}$$

Moreover in equation 5, V_t^{targ} represents some target state-value function that is to be obtained. Note that there exist also other objective functions mentioned in Schulman et al. (2017) that can substitute $L_t^{\text{CLIP}}(\theta)$ in equation 3. In equation 6, we make use of a hyperparameter $\epsilon \in [0, 1]$ and in combination with the minimizer from equation 4 we can prevent a single policy update to accidentally ruin the policy forever by sending the algorithm to a gradient region with a worse local extremum.

For the PPO algorithm, the detailed pseudocode according to Schulman et al. (2017) is as follows:

PPO:	Detailed Pseudocode, Actor-Critic Setup
1:	**Input:** Number of iterations (I), number of actors (N), number of timesteps per iteration (T), number of epochs (K), minibatch size ($M \leq NT$), learned state-value function implicitly in the advantage estimator ($V(s)$), target value (V_t^{targ}) implicitly in the objective L.
2:	**For** $i = 1, ..., I$ **do** $(^1)$
3:	| **For** $n = 1, ..., N$ **do** $(^2)$
4:	| | Run policy $\boldsymbol{\pi}_{\theta_{\text{old}}}$ in environment for T timesteps.
5:	| | Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$.
6:	| $(^2)$
7:	| Optimize objective L w.r.t. θ, for given K and M, using e.g. minibatch SGD or Adam.
8:	| Update parameters $\theta_{\text{old}} \leftarrow \theta$.
9:	$(^1)$

D NoisyNet

The central idea in a NoisyNet architecture brought forward by Fortunato et al. (2018) is to introduce random noise into the weights learned by the network (p inputs, q outputs), i.e. if we have a neural network whose output can be parametrized by a vector of weights $\boldsymbol{\theta}$ as $\mathbf{y} = f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + \mathbf{b}$, then we can learn the set of parameter vectors $\zeta = (\boldsymbol{\mu}, \boldsymbol{\Sigma})$ to compute $\mathbf{w} = (\boldsymbol{\mu}^w + \boldsymbol{\sigma}^w \odot \boldsymbol{\varepsilon}^w)$ and $\mathbf{b} = (\boldsymbol{\mu}^b + \boldsymbol{\sigma}^b \odot \boldsymbol{\varepsilon}^b)$ with the following dimensionalities: $\boldsymbol{\mu}^w$, $\boldsymbol{\sigma}^w$, $\boldsymbol{\varepsilon}^w \in \mathbb{R}^{q \times p}$ implying $\mathbf{w} \in \mathbb{R}^{q \times p}$, and $\boldsymbol{\mu}^b$, $\boldsymbol{\sigma}^b$, $\boldsymbol{\varepsilon}^b \in \mathbb{R}^q$ resulting in $\mathbf{b} \in \mathbb{R}^q$. Note that $\mathbf{x} \in \mathbb{R}^p$, $\mathbf{y} \in \mathbb{R}^q$ and $\odot$ denotes element-wise multiplication.

We optimize the loss $\bar{L}(\zeta) = \mathbb{E}_\varepsilon[L(\boldsymbol{\theta})]$, i.e. the expectation of the loss $L(\boldsymbol{\theta})$ over the noise ε which has a mean of zero and fixed statistics, using gradient descent on the parameters ζ. The noise can be sampled either independently or in a factorized way from a Gaussian. The former method samples the noise $\varepsilon^w_{i,j} \in \boldsymbol{\varepsilon}^w$ and $\varepsilon^b_j \in \boldsymbol{\varepsilon}^b$ for every input to the network leading to a total of $pq+q$ random samples. Factorized sampling on the other hand greatly reduces computational complexity by factorizing $\varepsilon^w_{i,j} = f(\varepsilon_i)f(\varepsilon_j)$ and then $\varepsilon^b_j = f(\varepsilon_j)$ thereby lowering the number of required random samples to $p + q$, where $f : \mathbb{R} \to \mathbb{R}$. In Fortunato et al. (2018) the function $f(x) = \text{sign}(x)\sqrt{|x|}$ is used. After sampling the noise, in theory we compute the gradients $\nabla\bar{L}(\zeta) = \nabla\mathbb{E}_\varepsilon[L(\boldsymbol{\theta})] = \mathbb{E}_\varepsilon[\nabla_{\boldsymbol{\mu},\boldsymbol{\Sigma}} L(\boldsymbol{\mu}+\boldsymbol{\Sigma}\odot\boldsymbol{\varepsilon})]$, whereas in practice we approximate $\nabla\bar{L}(\zeta) \approx \nabla_{\boldsymbol{\mu},\boldsymbol{\Sigma}} L(\boldsymbol{\mu}+\boldsymbol{\Sigma}\odot\boldsymbol{\varepsilon})$ with the Monte Carlo method.

In the code of Mahankali et al. (2024a), a NoisyNet is used in conjunction with the A3C-algorithm, in particular the loss is computed from the action-value function is estimated using:

$$\hat{Q}_i = \sum_{j=i}^{k-1} \gamma^{j-i} r_{t+j} + \gamma^{k-i} V(x_{t+k}|\zeta, \varepsilon_i) \tag{7}$$

Moreover, for factorized networks, we initialize $\mu_{i,j} \sim \mathcal{U}\left[-\frac{1}{\sqrt{p}}, +\frac{1}{\sqrt{p}}\right]$ and $\sigma_{i,j} = \frac{\sigma_0}{\sqrt{p}}$ with p being the number of inputs to the corresponding linear layer, $\mathcal{U}$ is a uniform distribution over the specified interval, and σ_0 is a hyperparameter.

Are we using independent or factorized sampling in the code? (I'm guessing factorized).

For the NoisyNet + A3C algorithm, the detailed pseudocode according to Fortunato et al. (2018) is as follows:

E Random Network Distillation

In Random Network Distillation, as described by Burda et al. (2019), we compose the reward which the agent obtains as $r_t = e_t + i_t$ where e_t represents a sparse extrinsic (environment's) reward and i_t is the intrinsic reward for transitioning, i.e. the exploration bonus emanating from the transition at time step t. The latter measures the novelty of a state and should yield a higher value, the less frequently a state has been visited so far. In case of a finite state space we can use $i_t = \frac{1}{n_t}$ or $i_t = \frac{1}{\sqrt{s}}$ where $n_t(s)$ denotes the number of times state s has been visited until time step t. However, there exist alternatives, e.g. we could use state density based approaches to calculate an exploration bonus, which are described in the paper mentioned above. Moreover, they specify that in the paper the intrinsic reward is the prediction error of a randomly generated problem $i_t = \|\hat{f}(x|\theta) - f(x)\|^2$ involving a target network $f : \mathcal{O} \to \mathbb{R}^k$ (fixed and randomly initialized, maps an observation $\mathcal{O}$ to an embedding $\mathbb{R}^k$) to sample the problem as well as a predictor network $\hat{f} : \mathcal{O} \to \mathbb{R}^k$ which was trained on the agent's data by gradient descent to optimize i_t w.r.t. parameters $\theta_{\hat{f}}$.

Since the overall return can be composed as a sum of returns $R = R_E + R_I$ we can train two heads V_E and V_I to estimate the value function $V = V_E + V_I$ and in extension the advantages. In the end, a policy is trained with standard PPO using the advantage estimators. Note that the intrinsic rewards have to be normalized in order to be useful because otherwise we could not guarantee that they are on a consistent scale, which is done by dividing i_t by a running estimate of the standard deviations of the intrinsic return. Furthermore,

NoisyNet: Detailed Pseudocode, for each actor-learner thread

Input: Global shared parameters (ζ_π, ζ_V), global shared counter (T), and maximal time $(T_{\max})$.
Input (each thread): Thread-specific parameters (ζ'_π, ζ'_V), set of random variables (ε), thread-specific counter (t), and roll-out size $(t_{\max})$.
Output: Policy $\pi(\cdot|\zeta_\pi, \varepsilon)$ and value function $V(\cdot|\zeta_V, \varepsilon)$.
$t \leftarrow 1$
Repeat $(^1)$
  Reset cumulative gradients: $d\zeta_\pi \leftarrow 0, d\zeta_V \leftarrow 0$.
  Synchronize thread-specific parameters: $\zeta'_\pi \leftarrow \zeta_\pi, \zeta'_V \leftarrow \zeta_V$.
  Set counter $c \leftarrow 0$.
  Get state x_t from environment.
  Sample noise $\xi \sim \varepsilon$.
  Create lists for rewards $r \leftarrow []$, actions $a \leftarrow []$ and states $x \leftarrow []$.
  Repeat $(^2)$
    Choose action $a_t \leftarrow \pi(\cdot|x_t, \zeta'_{pi}, \zeta'_V)$.
    $a[-1] \leftarrow a_t$
    Obtain reward r_t and new state x_{t+1}.
    $r[-1] \leftarrow r_t$, $x[-1] \leftarrow x_t$, $t \leftarrow t+1$, $T \leftarrow T+1$, $c \leftarrow c+1$
  $(^2)$ **until** x_t **is terminal or** $c > t_{\max}$
  If x_t is terminal, **then** $(^3)$
    $Q \leftarrow 0$
  else
    $Q \leftarrow V\left(x_t|\zeta'_V, \xi\right)$
  $(^3)$
  For $i = c-1, ..., 0$ **do** $(^4)$
    $Q \leftarrow r\left[i\right] + \gamma Q$
    $d\zeta_\pi \leftarrow d\zeta_\pi \nabla_{\zeta'_\pi} \log\left(\pi(a[i]|x[i], \zeta'_\pi, \xi)\right)[Q - V(x[i]|\zeta'_V, \xi)]$
    $d\zeta_V \leftarrow d\zeta_V \nabla_{\zeta'_V} \left[Q - V\left(x[i]|\zeta'_V, \xi\right)\right]^2$
  $(^4)$
  Update asynchronously parameter $\zeta_\pi \leftarrow \zeta_\pi + \alpha_\pi d\zeta_\pi$.
  Update asynchronously parameter $\zeta_V \leftarrow \zeta_V - \alpha_V d\zeta_V$.
$(^1)$ **until** $T > T_{\max}$

the observation also has to be normalized as $x \leftarrow \text{clip}(\frac{x-\mu}{\sigma}, -5, 5)$. The normalization parameters can be initialized by letting a random agent briefly move in the environment before beginning the optimization.

The detailed pseudocode according to Burda et al. (2019) is as follows:

F Citations, figures, and tables

Citations When the authors or the publication are included in the sentence, the citation should not be in parenthesis, using `\citet{}` (as in "See (CITATION REMOVED) for more information."). Otherwise, the citation should be in parenthesis using `\citep{}` (as in "Deep learning shows promise to make progress towards AI (CITATION REMOVED).").

Figures All artwork must be neat, clean, and legible. The figure number and caption always appear after the figure. Place one line space before the figure caption, and one line space after the figure. The figure caption is lowercase (except for the first word and proper nouns). Make sure the figure caption does not get separated from the figure. Leave sufficient space to avoid splitting the figure and figure caption.

Tables All tables must be centered, neat, clean and legible. The table number and title always appear before the table. See Table 2. Place one line space before the table title, one line space after the table title,

RND:	Detailed Pseudocode
1:	**Input:** Number of rollouts (N), Number of optimization steps (N_{opt}), and length of initial steps for initializing observation normalization (M).
2:	$t \leftarrow 0$
3:	Sample state $s_0 \sim p_0(s_0)$.
4:	**For** $m = 1, ..., M$ **do** $(^1)$
5:	| Sample action $a_t \sim \mathcal{U}(a_t)$. Are we sampling uniformly from all available actions at time t?
6:	| Sample state $s_{t+1} \sim p(s_{t+1}\|s_t, a_t)$.
7:	| Update observation normalization parameters using s_{t+1}.
8:	| $t \leftarrow t + 1$
9:	$(^1)$
10:	**For** $i = 1, ..., N$ **do** $(^2)$
11:	| **For** $j = 1, ..., K$ **do** $(^3)$
12:	| | Sample action $a_t \sim \pi(a_t\|s_t)$.
13:	| | Sample state $s_{t+1}, e_t \sim p(s_{t+1}, e_t\|s_t, a_t)$.
14:	| | Calculate intrinsic reward $i_t = \|\hat{f}(s_{t+1}) - f(s_{t+1})\|^2$.
15:	| | Add $s_t, s_{t+1}, a_t, e_t, i_t$ to optimization batch B_i.
16:	| | Update running estimate of reward standard deviation using i_t.
17:	| | $t \leftarrow t + 1$
18:	| $(^3)$
19:	| Normalize the intrinsic rewards contained in B_i.
20:	| Calculate returns $R_{I,i}$ and advantages $A_{I,i}$ for intrinsic reward.
21:	| Calculate returns $R_{E,i}$ and advantages $A_{E,i}$ for extrinsic reward.
22:	| Calculate combined advantages $A_i = A_{I,i} + A_{E,i}$.
23:	| Update observation normalization parameters using B_i.
24:	| **For** $j = 1, ..., N_{\text{opt}}$ **do** $(^4)$
25:	| | Optimize θ_π w.r.t. PPO loss on batch B_i, R_i, A_i using Adam.
26:	| | Optimize $\theta_{\hat{f}}$ w.r.t. distillation loss on B_i using Adam.
27:	| $(^4)$
28:	$(^2)$

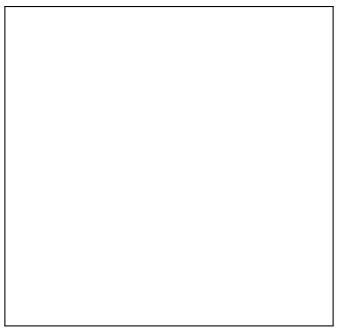


Figure 13: Sample figure caption.

and one line space after the table. The table title must be lowercase (except for the first word and proper nouns).

G Math Notation

In an attempt to encourage standardized notation, we have included the notation file from the textbook, *Deep Learning* (CITATION REMOVED) available at `https://github.com/goodfeli/dlbook_notation/`. Use of this style is not required and can be disabled by commenting out `math_commands.tex`.

Table 2: Sample table title

PART	DESCRIPTION
Dendrite	Input terminal
Axon	Output terminal
Soma	Cell body (contains cell nucleus)

Numbers and Arrays

a	A scalar (integer or real)
$\boldsymbol{a}$	A vector
$\boldsymbol{A}$	A matrix
$\mathbf{A}$	A tensor
$\boldsymbol{I}_n$	Identity matrix with n rows and n columns
$\boldsymbol{I}$	Identity matrix with dimensionality implied by context
$\boldsymbol{e}^{(i)}$	Standard basis vector $[0, \dots, 0, 1, 0, \dots, 0]$ with a 1 at position i
$\text{diag}(\boldsymbol{a})$	A square, diagonal matrix with diagonal entries given by $\boldsymbol{a}$
a	A scalar random variable
$\mathbf{a}$	A vector-valued random variable
$\mathbf{A}$	A matrix-valued random variable

Sets and Graphs

$\mathbb{A}$	A set
$\mathbb{R}$	The set of real numbers
$\{0, 1\}$	The set containing 0 and 1
$\{0, 1, \dots, n\}$	The set of all integers between 0 and n
$[a, b]$	The real interval including a and b
$(a, b]$	The real interval excluding a but including b
$\mathbb{A} \backslash \mathbb{B}$	Set subtraction, i.e., the set containing the elements of $\mathbb{A}$ that are not in $\mathbb{B}$
$\mathcal{G}$	A graph
$Pa_{\mathcal{G}}(\mathrm{x}_i)$	The parents of x_i in $\mathcal{G}$

Indexing

a_i	Element i of vector $\boldsymbol{a}$, with indexing starting at 1
a_{-i}	All elements of vector $\boldsymbol{a}$ except for element i
$A_{i,j}$	Element i, j of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{i,:}$	Row i of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{:,i}$	Column i of matrix $\boldsymbol{A}$
$A_{i,j,k}$	Element (i, j, k) of a 3-D tensor $\mathbf{A}$
$\mathbf{A}_{:,:,i}$	2-D slice of a 3-D tensor
a_i	Element i of the random vector $\mathbf{a}$

Calculus

$\frac{dy}{dx}$	Derivative of y with respect to x
$\frac{\partial y}{\partial x}$	Partial derivative of y with respect to x
$\nabla_{\boldsymbol{x}} y$	Gradient of y with respect to $\boldsymbol{x}$
$\nabla_{\boldsymbol{X}} y$	Matrix derivatives of y with respect to $\boldsymbol{X}$
$\nabla_{\mathbf{X}} y$	Tensor containing derivatives of y with respect to $\mathbf{X}$
$\frac{\partial f}{\partial \boldsymbol{x}}$	Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{m \times n}$ of $f : \mathbb{R}^n \to \mathbb{R}^m$
$\nabla^2_{\boldsymbol{x}} f(\boldsymbol{x})$ or $\boldsymbol{H}(f)(\boldsymbol{x})$	The Hessian matrix of f at input point $\boldsymbol{x}$
$\int f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral over the entire domain of $\boldsymbol{x}$
$\int_{\mathbb{S}} f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral with respect to $\boldsymbol{x}$ over the set $\mathbb{S}$

Probability and Information Theory

$P(\mathrm{a})$	A probability distribution over a discrete variable
$p(\mathrm{a})$	A probability distribution over a continuous variable, or over a variable whose type has not been specified
$\mathrm{a} \sim P$	Random variable a has distribution P
$\mathbb{E}_{\mathrm{x} \sim P}[f(x)]$ or $\mathbb{E} f(x)$	Expectation of $f(x)$ with respect to $P(\mathrm{x})$
$\mathrm{Var}(f(x))$	Variance of $f(x)$ under $P(\mathrm{x})$
$\mathrm{Cov}(f(x), g(x))$	Covariance of $f(x)$ and $g(x)$ under $P(\mathrm{x})$
$H(\mathrm{x})$	Shannon entropy of the random variable x
$D_{\mathrm{KL}}(P \Vert Q)$	Kullback-Leibler divergence of P and Q
$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Gaussian distribution over $\boldsymbol{x}$ with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$

Functions

$f : \mathbb{A} \to \mathbb{B}$	The function f with domain $\mathbb{A}$ and range $\mathbb{B}$
$f \circ g$	Composition of the functions f and g
$f(\boldsymbol{x}; \boldsymbol{\theta})$	A function of $\boldsymbol{x}$ parametrized by $\boldsymbol{\theta}$. (Sometimes we write $f(\boldsymbol{x})$ and omit the argument $\boldsymbol{\theta}$ to lighten notation)
$\log x$	Natural logarithm of x
$\sigma(x)$	Logistic sigmoid, $\dfrac{1}{1+\exp(-x)}$
$\zeta(x)$	Softplus, $\log(1+\exp(x))$
$\|\|\boldsymbol{x}\|\|_p$	L^p norm of $\boldsymbol{x}$
$\|\|\boldsymbol{x}\|\|$	L^2 norm of $\boldsymbol{x}$
x^+	Positive part of x, i.e., $\max(0, x)$
$\mathbf{1}_{\text{condition}}$	is 1 if the condition is true, 0 otherwise