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 work we investigate the claims of Mahankali et al. (2024a) in the paper Random Latent Exploration for Deep Reinforcement Learning[1] which introduces a novel exploration technique for reinforcement learning. The main claim of Mahankali et al. (2024a) is that RLE outperforms PPO, NoisyNet, and RND in discrete (FourRoom, Atari) and continuous (IsaacGym) control tasks. We replicated their experiments in the FourRoom environment and confirm that RLE outperforms the other algorithms. However, our experiments show that the performance of RLE depends on the choice of the latent vector distribution, which contradicts the findings of Mahankali et al. (2024a). Furthermore, we replicated their experiments for the two Atari games Alien and StarGunner. While RLE outperformed all baselines in Alien, it performed worse than PPO but better than NoisyNet and RND in StarGunner, contrasting with the original results of Mahankali et al. (2024a), where RLE dominated both games. Mahankali et al. (2024a) report the performance of RLE in the IsaacGym enviornment CartPole. Since IsaacGym is deprecated we tested RLE in the CartPole environment of IsaacLab where it outperformed the two baselines. Overall, we can confirm that RLE is a suitable alternative to existing exploration-focued reinforcement learning algorithms. Lastly, we implemented two RLE variants with adaptive latent vector distribution and tested it in the Alien game where both variants outperformed standard RLE.

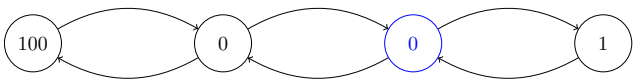
This report is prepared as part of the course project in *Advanced Topics in Machine Learning* at *Università della Svizzera italiana* in the autumn semester of 2024.

1 Introduction

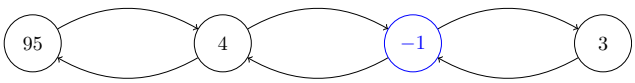
The paper we investigate deals with the problem of motivating an agent in a high-dimensional state space to explore the environment more exhaustively during training and thereby find non-obvious trajectories that can lead to higher long-term rewards for both discrete and continuous action spaces. The 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 also 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.

2 Scope of reproducibility

The main quantifiable claims made by Mahankali et al. (2024a) that we will be testing consist of the following:

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

Our findings regarding these claims will be presented in section 4 Check link before submission. Please note that there are more claims made by the authors, however, due to time constraints we are not able to test all of them, the remaining ones are shown in appendix B check link before submission.

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

In the equation above, 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[2]. Both $\boldsymbol{\phi}$ and $\mathbf{z}$ are d-dimensional vectors. Note that although the value-equation 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 the value-equation 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 14 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)

[2] 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.

3.1 Hyperparameters

Wherever possible we used the same hyperparameters as the authors, except for z-resampling frequency in the ATARI games. For the parameters that were missing a description in the paper we adopted values that seemed reasonable to us, which you can find in the appendix C.

3.2 Experimental setup and code

The code for our implementations can be found in GitHub[3]. 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[4]. This repository contains code for the ATARI and ISAACGYM[5] environments with the 4 algorithms (RLE, PPO, RND, NoisyNet). The authors used the PPO, NoisyNet, and RND implementations from Huang et al. (2022).

3.2.1 Atari

The authors of the paper under investigation chose the ATARI environment, introduced by Mnih et al. (2013), to represent a discrete action-space deep-RL benchmark. This environment comprises 57 different games, among which we performed experiments on two games, namely ALIEN and STARGUNNER. To understand the code given by the authors, we made use mainly of the official environment documentation as well as other papers that we found along the way. Here, the agent perceives the 4 most recent 84×84 grayscale frames, one frame of which can be found in figure 2a. During training, every trajectory (episode) always commences from the same initial state and can be terminated in one of the two ways: A game event occurs, or the maximum episode length is reached. The former refers to e.g. the loss of a "life" or a task being completed. As the authors provided the code for this environment, we did not modify it significantly. The main claims that we tested here were the superior performance of RLE and deeper exploration.

The game score S_G that we use to compare the different algorithms we run in this environment are normalized in the manner introduced by Badia et al. (2020) by taking into account the score an average human achieves S_H as well as the score an agent achieves by always choosing a random action S_R, which gives us the human normalized score $S_N = \frac{S_G - S_R}{S_H - S_R}$. According to Badia et al. (2020), in the case of ALIEN, we have $S_H = 7127.7$ and $S_R = 227.8$ while for STARGUNNER we have $S_H = 10250$ and $S_R = 664$. With the normalization we can get a better comparison of how much the agent behaves like a random agent ($S_N = 0$) or like a human player ($S_N = 1$), or even better $S_N > 1$. Moreover, the normalization allows for comparisons between games, adjusted for the general difficulty of a game.

Each game was run with all four algorithms (RLE, PPO, NoisyNet, RND) and the most important comparative measure to evaluate the agent's performance is the game score, which was computed from the cumulative extrinsic rewards (obtained from infos["r"] of the environment) over the episodes, treated as undiscounted sums to ensure a uniform weight across time steps for the purpose of investigating the algorithm's exploratory abilities. To this end, we also record the time evolution of the Shannon entropy of each algorithm, where a higher entropy indicates that the agent's selection of actions out of all available actions is broader and less concentrated, hinting at more thorough exploration capabilities.

3.2.2 FourRoom

Figure 2b shows the FOURROOM environment where we have $50 \times 50 + 4$ states[6] 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, we can add a goal state 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

[3] https://github.com/jonupp/adv_topics_ml_repl_chal Change github
[4] https://github.com/Improbable-AI/random-latent-exploration
[5] Please note that the ISAACGYM environment is deprecated and we are using ISAACLAB instead.
[6] 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.

and RLE algorithms from the ATARI environment, while we adapted the RND code from the ISAAC 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 several 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 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 reward function really guides the generation of diverse trajectories in a reward-free environment. To assess this, the intrinsic reward 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 and 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].

3.2.3 IsaacLab

In the case for this implementation, the code provided was develop in IsaacGym, which is depricated. [7] Therefore, we decided to reimplement it in IsaacLab which is a reinforcement learning framework built on top of IsaacSim. However, the are changes in terms of the code architecture and features of the software. As an example, the physics engine is an improved version with respect to IsaacGym, with different asset extensions.

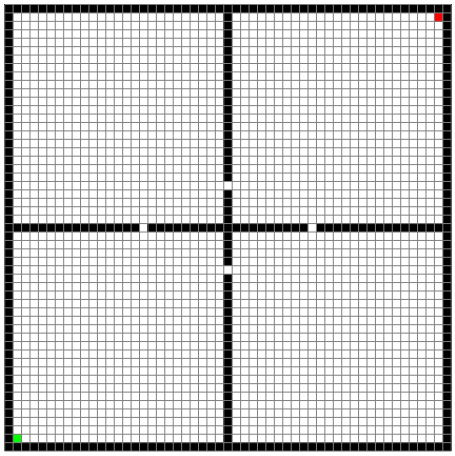
Therefore, for implementing the paper we based in an example provided by SKRL. Using their based implementation for PPO as an example, we could identify the main structure to develop our own version.

I will proceed to describe each component of the implementation. The *environment* file (*cartpole_rle_env.py*) extends the basic Cartpole environment class with the RLE necessary components as for example, the latent vector generation and the intrinsic and extrinsic rewards. The configuration file ($ppo_cfg_rle.yaml$) contains the hyperparameters for the training process. The *models.py* file contains the policy and value networks implementation for the rle case. Whereas, the *rle_network.py* file contains the feature network implementation for the latent vector generation. Besides these files, we needed to implement the trainer file (*train_rle.py*) which handles the arguments - acting as entry point, initialize IsaacSim, setup up the trackings, configures the environment and instantiates the runner (*runner_rle.py*) which instantiates the agent, coordinates the interactions between the agent and the environment and executes the training. Finally, the PPO-RLE algorithm (*ppo_rle_sk.py*) which is the custom PPO that handles the dual rewards for the two heads of the value network, the generalized advantage estimation and the learning scheduling among others.

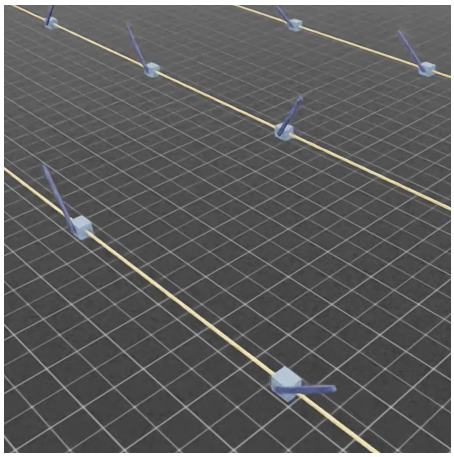
For the development of each of the parts we did not relay in the authors code unless to check the default parameters when they were not stated in the paper and as a double check of the logic of the optimization since the complexity of the environment interaction with Isaaclab is higher and was mostly managed by modifying deeply the skrl base ppo code. Since this implementation are highly strucuted we needed to implement work arounds, as for example, this algorithm has implemented the actor - critic structure handling but not a feature network, therefore we handle this by generating the latent vector in the environment, computing the intrinsic rewards and passing it through the runner to the modified ppo

[7] The poor documentation for migrating from IsaacGym to IsaacLab provided by NVIDIA `https://isaac-sim.github.io/IsaacLab/main/source/migration/migrating_from_isaacgymenvs.html`

(a) 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.



(b) 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.



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

Figure 2: Examples of the environments in use.

algorithm to handle the dual rewards and the soft update of the feature network during policy updates and environment resets. In addition, the maximum number of steps is around 299 for episode in this implementation of IsaacLab using the skrl base libraries, however, the authors have a maximum step of 500. Since increase the step would require higher amount of computations plus library restrictions we opted to keep the 300 steps which means that our results are a scaled down version of the authors but with relative the same results.

3.2.4 Queue-Based Von Mises-Fisher

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 provide 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 goal is to gradually adjust $\boldsymbol{\mu}$ during training so that it converges to a point in the space where $P_{\mathbf{z}}$ generates $\mathbf{z}$ samples that consistently produce 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. κ should be large if the model has converged to a $\boldsymbol{\mu}$ that reliably gives $\mathbf{z}$ values with high returns. 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 Neural Adaptive Von Mises-Fisher

We explore further on the idea of using a distribution that allows for a natural manipulation of the exploration-exploitation trade off. Since steps of an episode have a temporal dependency, therefore, we thought as a good approach to process the trajectories with an LSTM, see figure 15. When a new episode ends, the hidden state is reset since between episodes the agent should not keep any information, however, the actual knowledge encoded in the weights of the LSTM are preserved.

After the LSTM pre-process the states features generated by the agent network, the mu network receives these processed features to predict a good direction of exploration. However, it could have different degrees of confidence on the given direction and features, reason why a kappa network will predict the kappa value that represents the confidence on the predicted mu for the given states in the trajectory. Once kappa and mu are obtained, we can sample from the VMF distribution to get the suggested direction of the next action to be taken. Keeping the sampling allows for a degree of exploration that cannot be represented by a non-stochastic output.

The obtained z, in conjunction with the processed features, are fed into the reward network that acts as the dot product in the paper's RLE network. Therefore, the reward network will operate with the respective features and the z vector to produce an intrinsic reward to guide exploration.

For the construction of the losses we refer to appendix H. check link

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.

All experiments in the Atari environment were run in Google Colab. Here we experimented with the available CPU, T4 GPU and TPU v2-8 and used Weights & Biases (W&B) for a real-time visualization of the game scores and other measures. In the end we chose the GPU for all runs. We linked a Google Drive to Colab in order to save model checkpoints. Overall the W&B workspace now contains 56 runs with a total compute time of ca. 241.2 hours, of which we are using 8 runs for the results of this project. Of that amount, ca. 76.7 compute hours were spent on StarGunner, the remaining ca. 164.5 compute hours were spent on Alien.

With respect to IsaacLab, the experiments were run on an Asus TUF Gaming A17 with a RTX 3060 Laptop GPU. The GPU characteristics are below the suggested minimum requirements for VRAM memory of NVIDIA, however, for developing the experiments for the CartPole environment was enough. The used CUDA version was 11.8 and the project packages were managed through conda besides the separate installation of Omniverse for IsaacSim and the posterior IsaacLab installation (Windows distribution).

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

In this section we present the results of our experiments.

4.1 Results reproducing original paper

4.1.1 FourRoom

The results from the experiment where every algorithm is run 20 times in the reward-free environment can be seen in figure 3. RLE shows the best performance, followed by RND, NoisyNet and PPO. Furthermore, RLE has the narrowest confidence interval. Although Mahankali et al. (2024a) did not perform an equivalent experiment, they concluded from the state visitation heatmap that RLE shows the best explorative behavior in this setting. This aligns with the result from our experiment.

Figure 4 shows the result of the experiment where every algorithm is run 20 times in the environment with a goal. Mahankali et al. (2024a) train five seeds in the environment with a goal for each algorithm, and find that the average score for RLE and NoisyNet is 0.6, while the average score for RND and PPO is 0. In our experiment RLE achieved the highest game score by a large margin while NoisyNet and RND performed similarly poorly with an average game score of < 0.2 at the end of the training. PPO achieved an average game score of 0 at the end of the training. Thus, the result that RLE is among the best performing algorithms in this setting is confirmed by our experiment.

Figure 5 depicts the result of the experiment where RLE variants with different latent vector distributions are compared in the reward-free environment by running each RLE variant 20 times. The RLE variants where the latent vector distribution follows the standard normal or Von Mises distribution perform similarly well and clearly outperform the RLE variants with standard uniform and exponential distribution. Moreover, the RLE variants with standard uniform and exponential distribution show similar performance to RND. These results indicate that the choice of the latent vector distribution is of some importance. Moreover, there are two pairs of distributions that show similar behavior although the distributions in the pair are not that similar. For instance, we chose parameters for the Von Mises distribution to make its peak small, so that it is quite different from a standard normal distribution and shows some similarity to a uniform distribution.

In figure 6 the results of the experiment where RLE variants with different latent vector distributions are compared in the environment with rewards. Again, each RLE variant was run 20 times. The RLE variant with the Von Mises distribution

performs best, followed by the standard normal variant, the exponential variant, and lastly the standard uniform variant. The difference in average game score between the RLE variants is large: the worst variant achieves an average game score slighlty above 0.2 while the best RLE variant achieves an average game score of appr. 0.8. The ranking of the RLE variants is the same as in the experiment in the reward-free environment. Again, the results indicate that the choice of the latent vector distribution impacts the performance of RLE.

We generated state visitation heatmaps for each algorithm and RLE variant for both the reward-free environment and the environment with a goal. From the 20 runs for each algorithm and RLE variant we chose the run with the highest entropy in the case of the reward-free environment and the run with the highest game score in the case of the environment with a goal. In figure 7 the state visitation heatmaps for the different algorithms in the reward-free environment are displayed. RLE and NoisyNet explore all four rooms while PPO and RND have a significant number of unexplored states. Figure 8 shows the state visitation heatmaps for the different algorithms in the environment with a goal. RLE, NoisyNet, and RND are frequently visiting states on an appr. shortest path from the start to the goal while PPO has only explored a single room. Next, the state visitation for the RLE variants are considered. In figure 9 we see that the standard normal, Von Mises, and exponential RLE variants explore all the four rooms well. Only the standard uniform RLE variant does not explore all states. In figure 10 we see that all RLE variants often visit states on one of the appr. shortest path between start and goal. However, the standard normal and Von Mises variants also explore almost the entire state space while the standard uniform and exponential RLE variants have large unexplored chunks.

Lastly, figure 11 shows that the intrinsic reward function guides the generation of diverse trajectories. The trajectories converge to areas where the intrinsic reward is large.

All the 95% confidence intervals are calculated using the bootstrapping method.

4.1.2 Atari

After running each game for a total of 40 million time steps, we have obtained the results shown in table 1.

Table 1: Results of the Atari experiments.

Algorithm	Game Score	Human-Norm.
RLE	1626	0.203
PPO	1384	0.168
NoisyNet	648	0.061
RND	1048	0.119

(a) The Alien game.

Algorithm	Game Score	Human-Norm.
RLE	42481	4.362
PPO	54564	5.623
NoisyNet	27530	2.803
RND	21565	2.180

(b) The StarGunner game.

For the Atari game, as can be seen in table 1a, the algorithm perform in a similar manner to the authors' result. Only NoisyNet performs much worse in our run, obtaining a score of 648, while the authors report a score of approximately 1113. However, regarding claim (1) we can confirm that RLE obtains the highest score of them all, which is evident from the plot of human-normalized scores in figure 12a. Additionally, we can compare the depth of exploration done by these algorithms in figure 12b by looking at the entropy distribution. Here, it is evident that RLE explores the largest part of the state space in comparison to the other algorithms. Using entropy as a measure of diverse behavior, we can moreover confirm as well claim (2) for this game.

In the case of the StarGunner game, as becomes apparent in table 1b, the RLE algorithm does not yield the best game score but PPO does. This result differs strongly from the findings of the authors. In more detail, we find that RLE obtains a game score of around 42481 while the authors report 64011, and in addition their implementation of NoisyNet performs much better at 42645 while our agent only got 27530 in the end. Thus, in this case we have to object claim (1), which is shown in more detail in figure 13a. On the contrary, we can confirm claim (2) again as shown in figure 13b since RLE does exhibit the most diverse behavior out of all four algorithms. Perhaps this hints at instabilities in the algorithmic behavior of the agent which are depending on e.g. the seed with which the run was initialized.

4.1.3 IsaacLab

The results from our IsaacLab replication show similar results to the authors'. The RLE algorithm shows the most stable performance in the CartPole environment, see figure 17 which replicates figure 28 and 22 of the paper. The difference on the maximum score comes from the maximum length of the episodes which are kept in 299 steps in out replication for computational reasons, besides that is not directly manipulable in the environment wrapper as on previous IsaacGym. Notice that we also use the same hyperparameters as the authors' implementation for PPO, under the default set of parameters of skrl the results can differ from the authors.

4.2 Results beyond original paper

4.2.1 Atari

One interesting observation that we made during our experiments is the fact that in Alien, the agent seems to obtain lower rewards on average with RLE, see figure 12c, but still has the highest game score overall. We conjecture that this means the agent with RLE behaves more cautiously trying to stay alive for longer and collect lower/safer rewards but over a longer time horizon meaning the overall accumulated reward (game score) still turns out to be superior compared to the other algorithms. This would be confirmed by RLE having the largest trajectory (episode) length, see figure 12d. However, this is probably not the entire explanation, since in StarGunner, see figure 13c and 13d, RLE also gets very low reward on average but here this cannot be compensated by having the largest trajectory length. Moreover, given that we also normalized the game scores, the result suggests that the agent is better at playing StarGunner and worse at playing Alien compared to an average human.

Moreover, as mentioned in the hyperparameter overview, we increased RLE's z-sampling frequency from every 1280 time steps (authors) to every 500, but we did not find materially different results. Another observation we made is that for some reason in the RLE implementation of Alien, the results different drastically between using a GPU and a CPU/TPU for the computations, but the underlying cause of this is beyond our understanding.

4.2.2 Adaptive Von Mises-Fisher

The observed results from the adaptive VMF are promising (see figure 16a), with the following comparable scores:

Table 2: Results of the Atari experiments with Adaptive VMF

Algorithm	Game Score	Human-Norm.
QVMF	2184	0.285
NAVMF	2109	0.273
RLE	1626	0.203
PPO	1384	0.168

(a) Alien-V5

In addition, trajectories were the longest for NA-VMF and equally long with respect to RLE for QVMF, see figure 16b. This implies that in the Q-VMF case the agent was able to kill more aliens relative to the other algorithms. Therefore, both of our adaptive VMF implementations outperforms the other algorithms for Alien-V5, pointing us to further investigate this approach for other Atari games and continuous action spaces.

Furthermore, we keep the same resampling frequency as in RLE, however, for adaptive VMF the loss terms, as explained, can naturally provide the agent with the feedback needed to resample z's in the appropriate moment, which can be not fixed a priori since many decisions change on the state context.

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

As we chose a RL paper to reproduce we did not have to deal with the hassle of downloading huge data sets for training. Moreover, for us the concept of the paper, especially the new idea introduced by Mahankali et al. (2024a), was fairly straight-forward and easy to understand. Moreover, the code provided by the authors for the Atari environment was comparatively ready to run, we did not have to make a lot of modifications here.

5.2 What was difficult

The largest challenges were the lack of computational resources, time limitations. Moreover, the paper is missing a proper description of the RLE architecture, as well as there not being a good IsaacLab documentation. For more details, see appendix I.

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

From our experiments it became evident that RLE is a suitable alternative to RND and NoisyNet for RL settings, especially in the case where deep exploration is necessary. However, due to limited computational resources and time constraints we could not test all of the authors' claims but what we found is roughly in line with their results. For future efforts it may be rewarding to test all the other Atari games and Isaac evnironments to further solidify the findings. Furthermore, the adaptive VMF techniques showed promising results in the Alien game and it may be a rewarding challenge to run it on other games as well and test it in the continuous action-space domain.

Member contributions

We divided the workload among the group members as follows:

- Jonatan Bella: VMF algorithms development and experimentation, IsaacLab implementation (rewriting of each component of the project code, environment modifications for RLE, and final experimentation)
- Alessia Berarducci: Atari implementation, IQM / bootstrapped confidence interval (not used in the end due to compute time), Small Reward Compensation Hypothesis.
- 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 of Atari interface, plots, assistance in conceptual questions.

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

References

Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, and Charles Blundell. Agent57: Outperforming the atari human benchmark. 2020. doi: 10.48550/arXiv.2003.13350. URL https://arxiv.org/abs/2003.13350.

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.

Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022. URL http://jmlr.org/papers/v23/21-1342.html.

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.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 2013. doi: 10.48550/arXiv.1312.5602. URL https://arxiv.org/abs/1312.5602.

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.

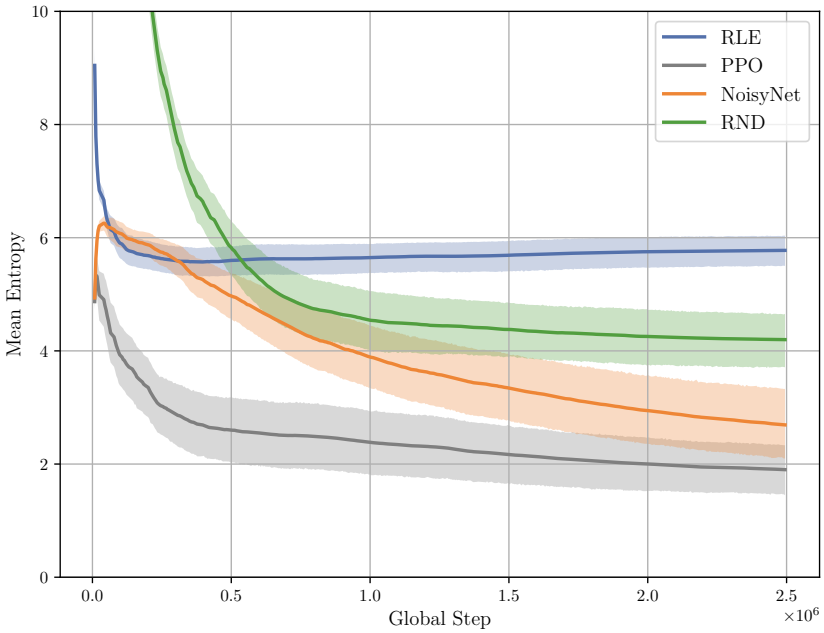


Figure 3: Mean entropy of state visitation counts over the training steps for different algorithms. The shaded areas represent the 95% confidence intervals. By the end of training, RLE achieves the highest entropy and the narrowest confidence interval.

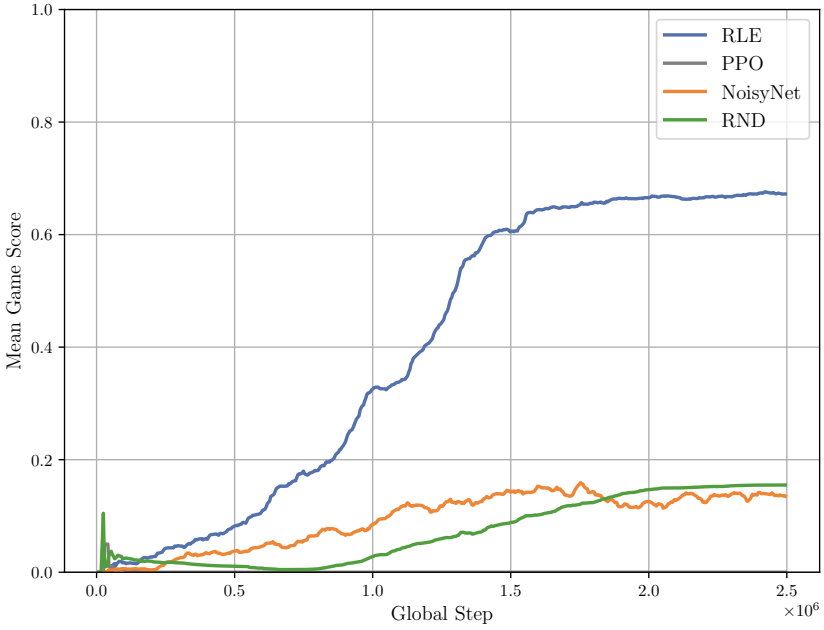


Figure 4: Mean game score of different algorithms. RLE clearly outperforms the other algorithms.

B Untested Claims

The authors made the following claims that, due to time constraints of this project, we were not able to test:

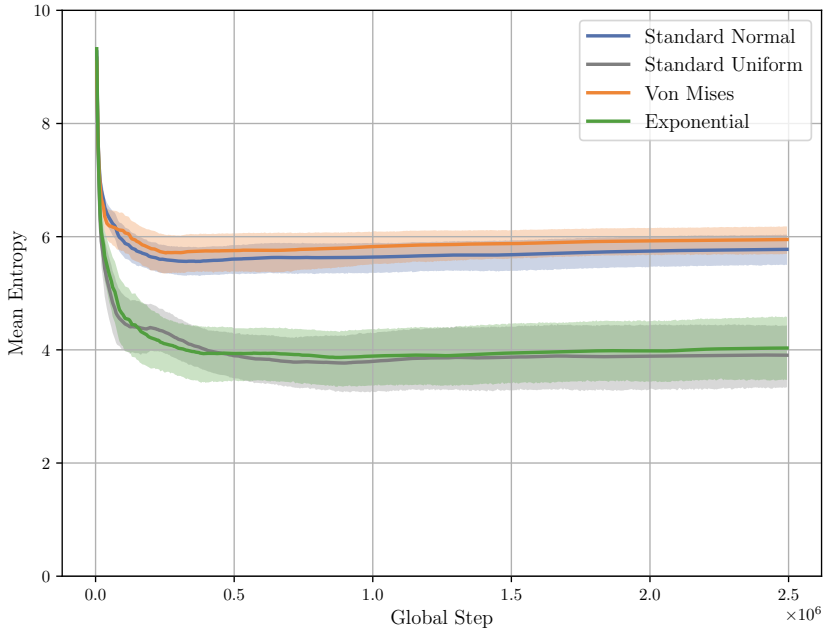


Figure 5: Mean entropy of state visitation counts over the training steps for RLE variants with different latent vector distributions. The shaded areas represent the 95% confidence intervals. The standard uniform and exponential variants explore less than the other two variants.

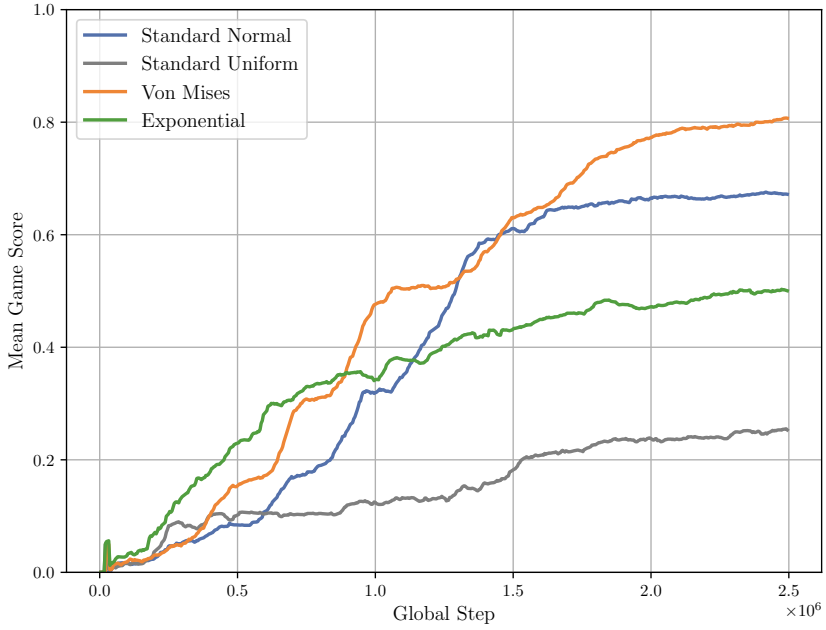


Figure 6: Mean game score of RLE variants with different latent vector distributions. The figure shows substantial differences in the mean game score across the different RLE variants.

1. For the more challenging discrete-action continuous-states[8] 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.[9]

2. 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.

[8]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.

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

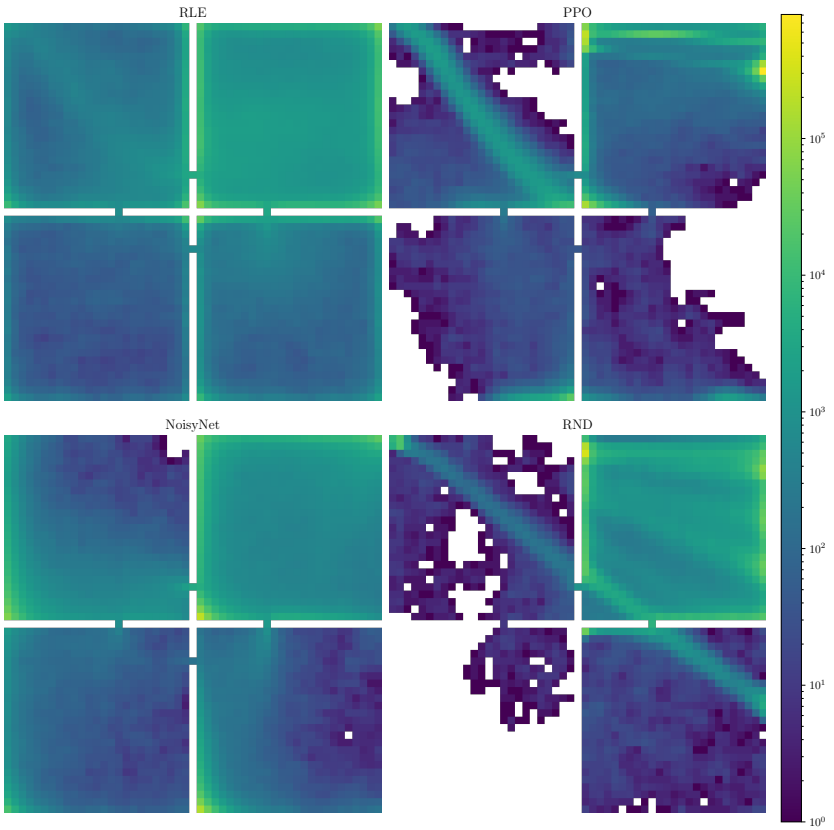


Figure 7: For each algorithm, the run with the highest entropy over the state visitation counts was selected to generate the heatmap. The heatmap illustrates the state visitation counts of all the algorithms after training for 2.5M timesteps in the reward-free environment. The start location is in the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

Moreover, they performed ablation studies and claimed the following:

1. RLE performance compared to PPO is invariant under a change in the dimensionality of the random latent vector.
2. RLE performance deteriorates if the policy and value networks are not conditioned on the random latent vector.
3. 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.
4. RLE performance is insensitive to the feature extractor architecture.

C Hyperparameters

In Atari, we mostly used the same values for the hyperparameters that the authors used. In table 3 the hyperparameters we modified are summarized. Note that contrary to the authors, due to time constraints we were only able to run every game once for each algorithm.

Table 3: Hyperparameters that differ from Mahankali et al. (2024a) for the Atari experiments.

Hyperparameter	Value
z-Sampling Frequency (switch_steps)	500
Norm-RLE-Features	True
Target-KL	None

(a) Modified RLE parameters.

Hyperparameter	Value
Anneal Learning Rate	False
Target KL Divergence	None
Sticky Action	True
Reward Normalization	True
Adam Epsilon	10^{-5}

(b) PPO parameters used in the code but not mentioned in the paper. The same holds for NoisyNet.

Hyperparameter	Value
Target KL Divergence	None
- Rest as in PPO -	

(c) RND parameters used in the code but not mentioned in the paper.

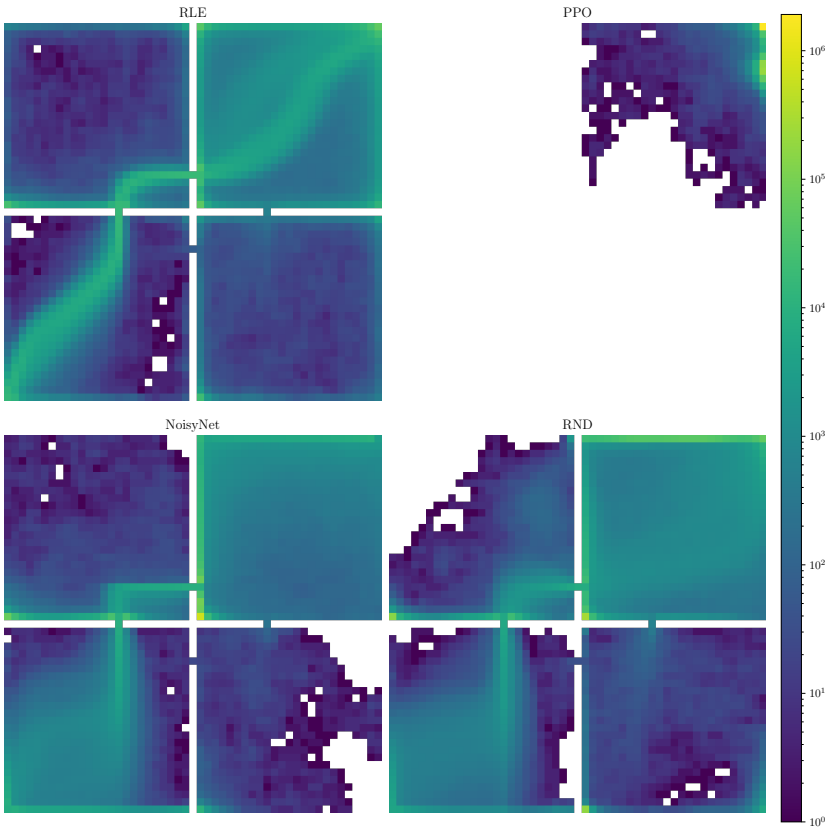


Figure 8: For each algorithm, the run with the highest game score was selected to generate the heatmap. The heatmap illustrates the state visitation counts of all the algorithms after training for 2.5M timesteps in the environment with a goal. The start location is in the top right cell and the goal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

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 values in table 4, that seemed reasonable to us although we did not spend a lot of time finetuning them.

Table 4: 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

For the IsaacLab experiments, we assume that the not specified hyperparameters under the RLE experiments section of the paper, were the same as in default PPO implementation which comprise the following values: There are two special mentions among these parameters. The number of parallel environments used is 4096, as it is the default value in the code in the authors repository for isaacgym. In addition, we choose to use a rollout of 8 as the authors suggested for "AllegroHand" and "ShadowHand" examples for CartPole, which provided higher results than 16. The timestep determination was done by observing a comparable figure with the authors and when the algorithm show a stability around a return.

D Random Latent Exploration

The architecture of our RLE implementation can be found in figure 14:

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

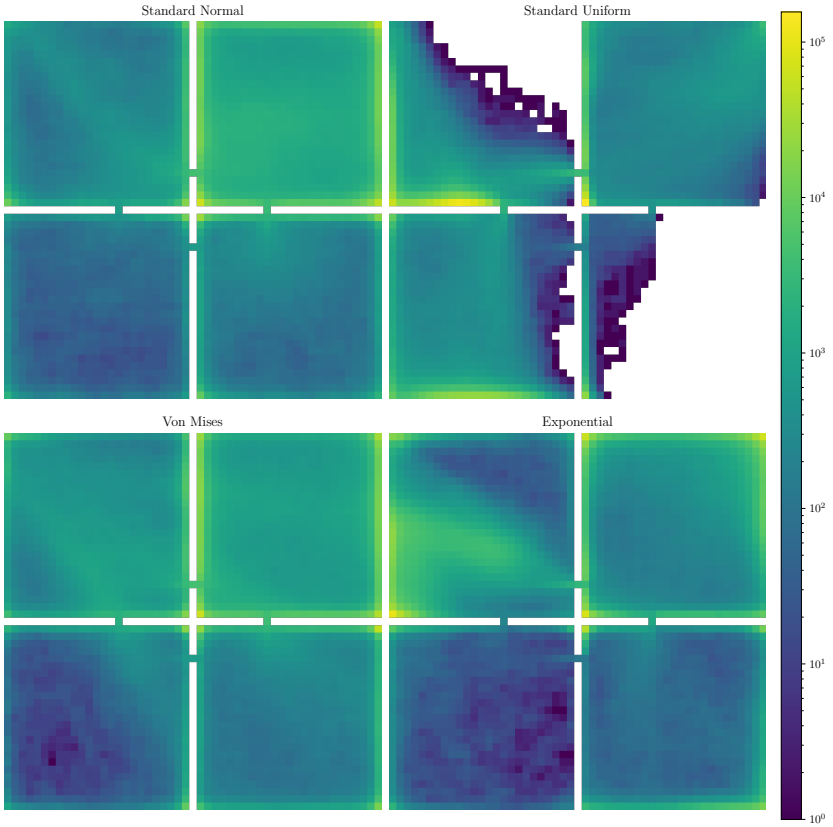


Figure 9: For each RLE variant with different latent vector distribution, the run with the highest entropy over the state visitation counts was selected to generate the heatmap. The heatmap illustrates the state visitation counts of all the RLE variants with different latent vector distribution after 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.

Table 5: Hyperparameters not stated by for IsaacLab experiments.

Hyperparameter	**Value**
Number of Parallel Environments	4096
Rollouts	8
Learning Epochs	4
Mini batches	2
Discount Rate	0.99
Generalized Advantage Estimation λ	0.95
Gradient Norm Bound	1.0
Used Clipped Value Loss	False
Value Loss Scale	2.0
Timesteps	5000

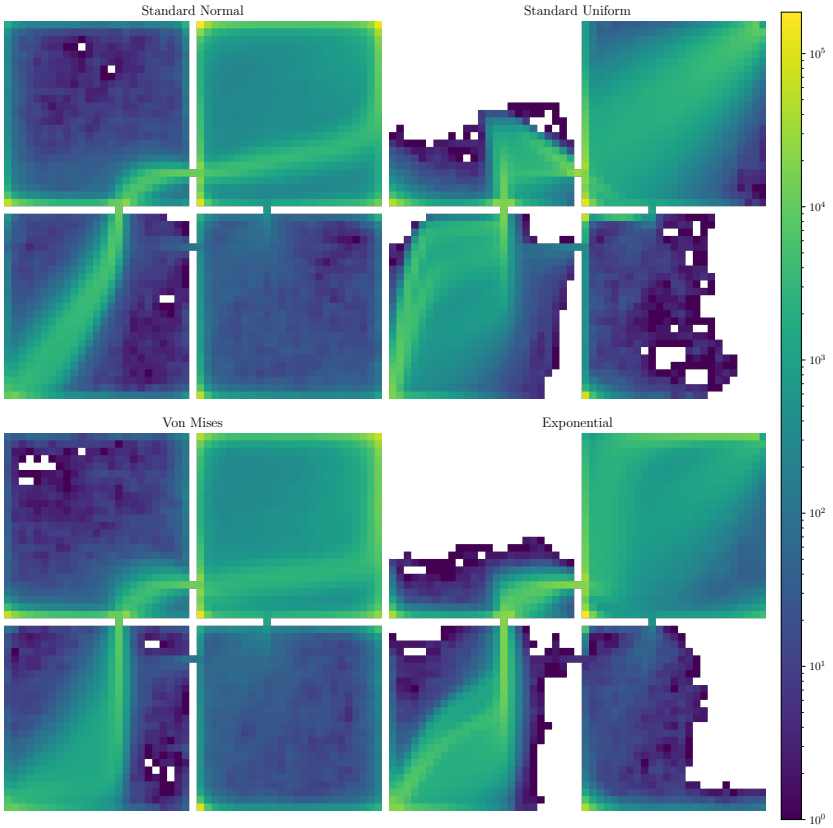


Figure 10: For each RLE variant with different latent vector distribution, the run with the highest game score was selected to generate the heatmap. The heatmap illustrates the state visitation counts of all the RLE variants with different latent vector distribution after training for 2.5M timesteps 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.

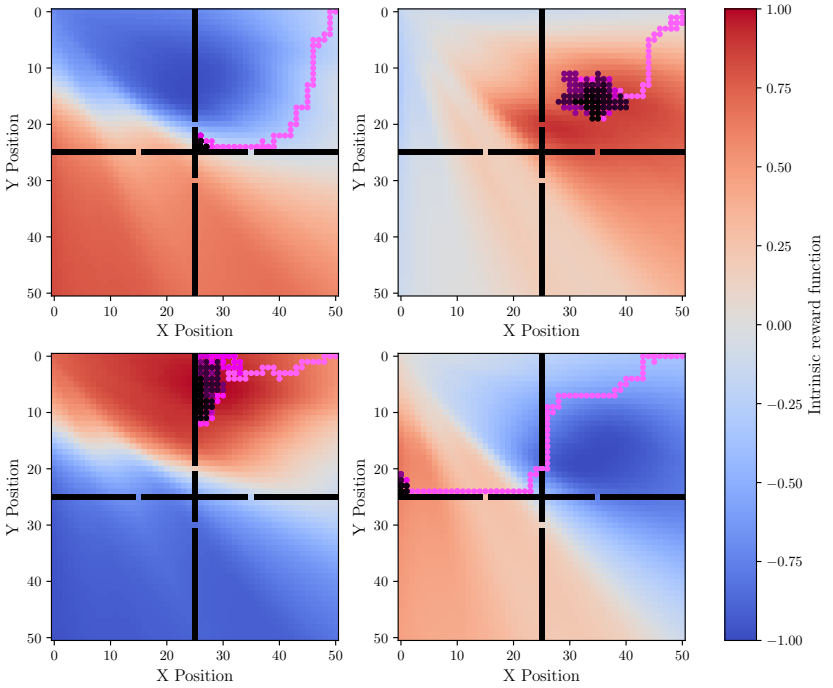
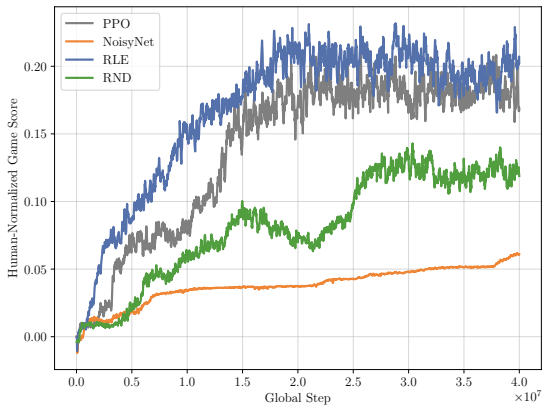
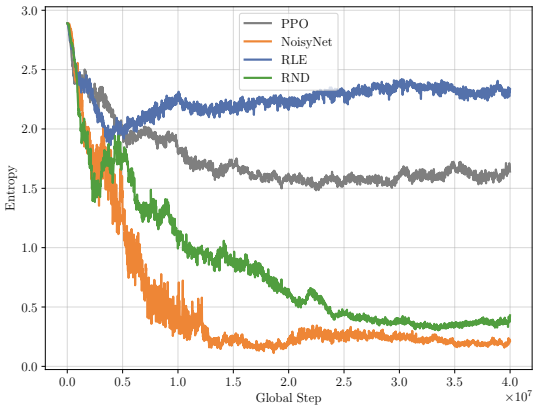


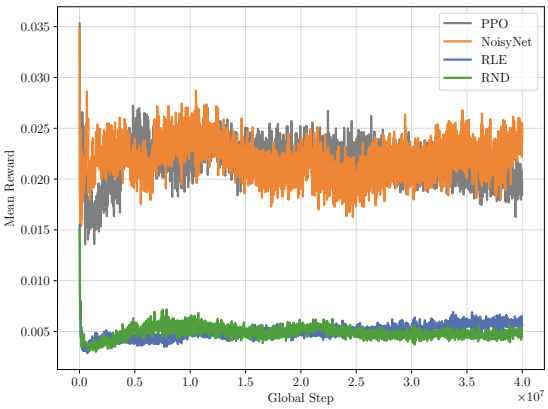
Figure 11: Trajectory and intrinsic reward function for four different $\mathbf{z}$ values. Earlier steps in the trajectory are pink and with increasing number of steps the color shifts to black. The trajectories generally tend towards regions with high intrinsic reward. The example is not cherry-picked. This behavior was observed for almost every figure we generated.



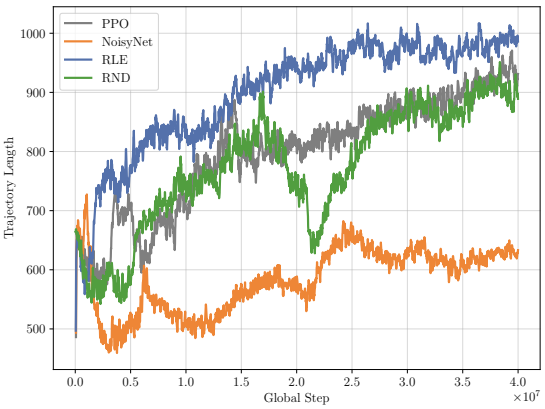
(a) Game Score.



(b) Entropy.

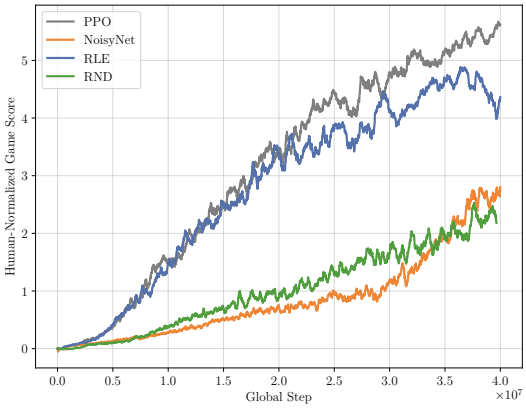


(c) Mean Reward.

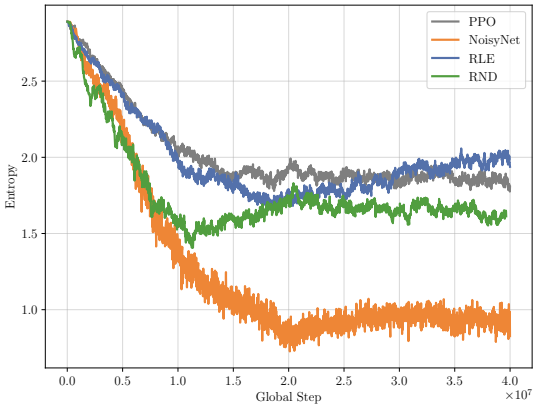


(d) Trajectory Length.

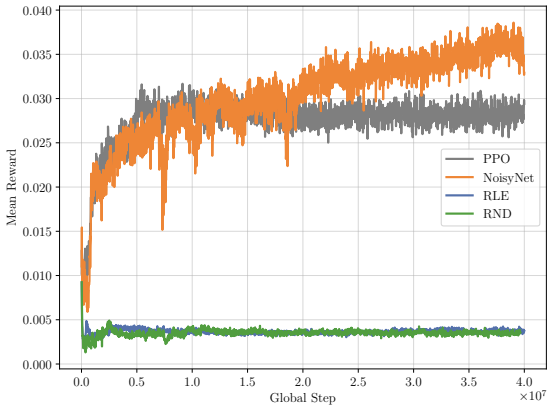
Figure 12: The ALIEN game results.



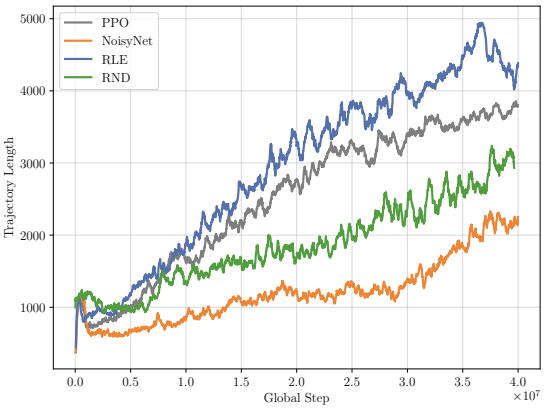
(a) Game Score.



(b) Entropy.



(c) Mean Reward.



(d) Trajectory Length.

Figure 13: The StarGunner game results.

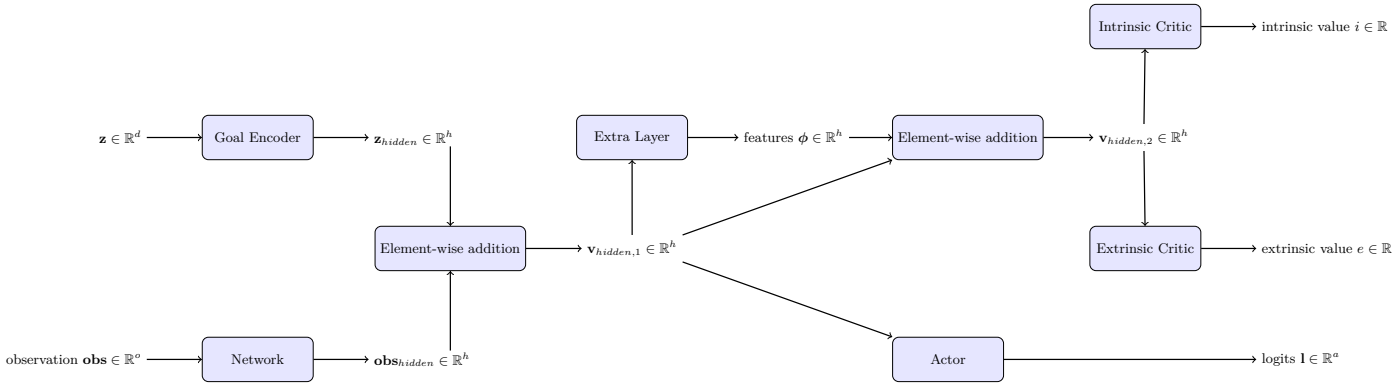


Figure 14: In RLE, the policy and critic networks are conditioned on the latent vector z. Mahankali et al. (2024a) split the critic's head in two: one head predicts the extrinsic value and the other head predicts the intrinsic value. In the vector dimension, o represents the dimension of a single observation, d represents the RLE feature dimension, and a is the cardinality of the action space.

RLE: Detailed Pseudocode

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).
Randomly initialize a feature network (ϕ) with the same backbone architecture as the policy and value networks.
Initialize running mean $\boldsymbol{\mu} = \mathbf{0}$ and standard deviation $\boldsymbol{\sigma} = \mathbf{1}$ estimates of $\boldsymbol{\phi}(s)$ over the state space.
Sample an initial latent vector $\mathbf{z} \sim P_{\mathbf{z}}$ for each parallel worker.
Repeat $(^1)$
| Sample initial state s_0.
| **For** $t = 0, ..., T$ **do** $(^2)$
| | Take action $a_t \sim \boldsymbol{\pi}(.|s_t, \mathbf{z})$ and transition to s_{t+1}.
| | Compute feature $\mathbf{f}(s_{t+1}) = \frac{\boldsymbol{\phi}(s_{t+1}) - \boldsymbol{\mu}}{\boldsymbol{\sigma}}$.
| | Compute random reward $F(s_{t+1}, \mathbf{z}) = \frac{\mathbf{f}(s_{t+1})}{\|\mathbf{f}(s_{t+1})\|} \cdot \mathbf{z}$.
| | Receive reward $r_t = R(s_t, a_t) + F(s_{t+1}, \mathbf{z})$.
| | **For** $i = 0, 1, ..., N-1$ **do** $(^3)$
| | | **If** worker i terminated **or** S timesteps passed without resampling, **then** $(^4)$
| | | | Resample sample $\mathbf{z} \sim P_{\mathbf{z}}$ for worker i.
| | | $(^4)$
| | $(^3)$
| $(^2)$
| Update policy network $\boldsymbol{\pi}$ and value network $V_{\boldsymbol{\pi}}$ with the collected trajectory $(\mathbf{z}, s_0, a_0, r_0, s_1, ..., s_T)$.
| Update feature network $\boldsymbol{\phi}$ using the value network's parameters $\boldsymbol{\phi} \leftarrow \tau \cdot V^{\boldsymbol{\pi}} + (1 - \tau) \cdot \boldsymbol{\phi}$.
| Update $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ using the batch of collected experience.
$(^1)$ **until convergence**

E 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 $t \in [0, T]$:

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

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:

$$L_t^{\text{CLIP+VF+S}}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{\text{CLIP}}(\theta) - c_1 L_t^{\text{VF}}(\theta) + c_2 S[\boldsymbol{\pi}_\theta](s_t)\right] \tag{2}$$

In equation 2, 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:

$$L_t^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t\right)\right] \tag{3}$$

$$L_t^{\text{VF}}(\theta) = \left(V_\theta(s_t) - V_t^{\text{targ}}\right)^2 \tag{4}$$

In equation 3 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:

$$\text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) = \begin{cases} 1-\epsilon, & r_t(\theta) < 1-\epsilon \\ r_t(\theta), & 1-\epsilon \leq r_t(\theta) \leq 1+\epsilon \\ 1+\epsilon, & r_t(\theta) > 1+\epsilon \end{cases} \tag{5}$$

Moreover in equation 4, 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 2. In equation 5, we make use of a hyperparameter $\epsilon \in [0, 1]$ and in combination with the minimizer from equation 3 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)**

F 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:

$$\hat{Q}_i = \sum_{j=i}^{k-1} \gamma^{j-i} r_{t+j} + \gamma^{k-i} V(x_{t+k} | \zeta, \varepsilon_i) \tag{6}$$

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.

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

NoisyNet:	Detailed Pseudocode, for each actor-learner thread
1:	**Input:** Global shared parameters (ζ_π, ζ_V), global shared counter (T), and maximal time $(T_{\max})$.
2:	**Input (each thread):** Thread-specific parameters (ζ'_π, ζ'_V), set of random variables (ε), thread-specific counter (t), and roll-out size $(t_{\max})$.
3:	**Output:** Policy $\pi(\cdot\|\zeta_\pi, \varepsilon)$ and value function $V(\cdot\|\zeta_V, \varepsilon)$.
4:	$t \leftarrow 1$
5:	**Repeat (1)**
6:	| Reset cumulative gradients: $d\zeta_\pi \leftarrow 0, d\zeta_V \leftarrow 0$.
7:	| Synchronize thread-specific parameters: $\zeta'_\pi \leftarrow \zeta_\pi, \zeta'_V \leftarrow \zeta_V$.
8:	| Set counter $c \leftarrow 0$.
9:	| Get state x_t from environment.
10:	| Sample noise $\xi \sim \varepsilon$.
11:	| Create lists for rewards $r \leftarrow []$, actions $a \leftarrow []$ and states $x \leftarrow []$.
12:	| **Repeat (2)**
13:	| | Choose action $a_t \leftarrow \pi(\cdot\|x_t, \zeta'_{pi}, \zeta'_V)$.
14:	| | $a[-1] \leftarrow a_t$
15:	| | Obtain reward r_t and new state x_{t+1}.
16:	| | $r[-1] \leftarrow r_t$, $x[-1] \leftarrow x_t$, $t \leftarrow t+1$, $T \leftarrow T+1$, $c \leftarrow c+1$
17:	| **(2) until** x_t **is terminal or** $c > t_{\max}$
18:	| **If** x_t is terminal, **then (3)**
19:	| | $Q \leftarrow 0$
20:	| **else**
21:	| | $Q \leftarrow V(x_t\|\zeta'_V, \xi)$
22:	| **(3)**
23:	| **For** $i = c-1, ..., 0$ **do (4)**
24:	| | $Q \leftarrow r[i] + \gamma Q$
25:	| | $d\zeta_\pi \leftarrow d\zeta_\pi \nabla_{\zeta'_\pi} \log(\pi(a[i]\|x[i], \zeta'_\pi, \xi))[Q - V(x[i]\|\zeta'_V, \xi)]$
26:	| | $d\zeta_V \leftarrow d\zeta_V \nabla_{\zeta'_V} [Q - V(x[i]\|\zeta'_V, \xi)]^2$
27:	| **(4)**
28:	| Update asynchronously parameter $\zeta_\pi \leftarrow \zeta_\pi + \alpha_\pi d\zeta_\pi$.
29:	| Update asynchronously parameter $\zeta_V \leftarrow \zeta_V - \alpha_V d\zeta_V$.
30:	**(1) until** $T > T_{\max}$

G 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, 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:

RND: Detailed Pseudocode

Input: Number of rollouts (N), Number of optimization steps (N_{opt}), and length of initial steps for initializing observation normalization (M).
$t \leftarrow 0$
Sample state $s_0 \sim p_0(s_0)$.
For $m = 1, ..., M$ **do** **(1)**
  Sample action $a_t \sim \mathcal{U}(a_t)$.
  Sample state $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$.
  Update observation normalization parameters using s_{t+1}.
  $t \leftarrow t + 1$
(1)
For $i = 1, ..., N$ **do** **(2)**
  For $j = 1, ..., K$ **do** **(3)**
    Sample action $a_t \sim \pi(a_t|s_t)$.
    Sample state $s_{t+1}, e_t \sim p(s_{t+1}, e_t|s_t, a_t)$.
    Calculate intrinsic reward $i_t = \|\hat{f}(s_{t+1}) - f(s_{t+1})\|^2$.
    Add $s_t, s_{t+1}, a_t, e_t, i_t$ to optimization batch B_i.
    Update running estimate of reward standard deviation using i_t.
    $t \leftarrow t + 1$
  (3)
  Normalize the intrinsic rewards contained in B_i.
  Calculate returns $R_{I,i}$ and advantages $A_{I,i}$ for intrinsic reward.
  Calculate returns $R_{E,i}$ and advantages $A_{E,i}$ for extrinsic reward.
  Calculate combined advantages $A_i = A_{I,i} + A_{E,i}$.
  Update observation normalization parameters using B_i.
  For $j = 1, ..., N_{\text{opt}}$ **do** **(4)**
    Optimize θ_π w.r.t. PPO loss on batch B_i, R_i, A_i using Adam.
    Optimize $\theta_{\hat{f}}$ w.r.t. distillation loss on B_i using Adam.
  (4)
(2)

H Neural-adaptive VMF

Even more interesting was the development of a loss that can guide the training of the networks. We first have an ALIGNMENT term:

$$\mathcal{L}_{align} = -\sum_{t=1}^{T} (\boldsymbol{\mu}_{t-1} \cdot \mathbf{z}_{t-1}) \cdot r_t \tag{7}$$

Such that when sampled z from a VMF with μ as direction produced high return r that will be seen in the next step, then most we want μ to be like z, aligning μ with the highest rewarding z.

The second term of the loss is the differential entropy of VMF:

$$H(\text{vMF}) = -\log(C_d(\kappa)) - \kappa A_d(\kappa) \tag{8}$$

where:

$$C_d(\kappa) = \frac{\kappa^{d/2-1}}{(2\pi)^{d/2} I_{d/2-1}(\kappa)} \tag{9}$$

$$A_d(\kappa) = \frac{I_{d/2}(\kappa)}{I_{d/2-1}(\kappa)} \tag{10}$$

Taking the logarithm of $C_d(\kappa)$:

$$\log(C_d(\kappa)) = (d/2-1)\log(\kappa) - \frac{d}{2}\log(2\pi) - \log(I_{d/2-1}(\kappa)) \tag{11}$$

Therefore, the complete entropy expression is:

$$L(\text{vMF}) = -\left[(d/2-1)\log(\kappa) - \frac{d}{2}\log(2\pi) - \log(I_{d/2-1}(\kappa))\right] - \kappa\frac{I_{d/2}(\kappa)}{I_{d/2-1}(\kappa)} \tag{12}$$

This term pushes the model for exploration (lower kappa) with two weighting parameters, one based on progress such that the term decays while training progresses and another based on stagnation, such that if the recent returns show little variation compared to the overall returns (over a short window), indicating stagnation, the function increases the exploration boost.

The third term is the REWARD loss - Huber loss:

$$\mathcal{L}_{huber} = \sum_{t=1}^{T} \begin{cases} \frac{1}{2}(r_t - \hat{r}_t)^2, & \text{if } |r_t - \hat{r}_t| < \delta \\ \delta\left(|r_t - \hat{r}_t| - \frac{1}{2}\delta\right), & \text{otherwise} \end{cases} \tag{13}$$

We use the Huber loss such that since reward can be pretty sparse and since what we finally want to teach the model is to differentiate what are good states to be in comparison to others then, we can soften the effect of this outliers such that the small errors goes under the quadratic loss whereas the outliers receive a linear loss, reducing their impact on the final loss.

We also need to teach the model how to form a confidence around a certain μ. The KAPPA loss is constructed by first estimating a good target kappa, which can be done by setting a maximum and minimum kappa bound. Visualizing the VMF distribution helps to notice that a k $\in$ [0,40] is a good range, allowing still even reaching the maximum to sample values at a certain range of distance from μ, allowing for exploration.

$$\kappa_{target} = \kappa_{min} + (\kappa_{max} - \kappa_{min}) \cdot \text{returns}_{scaled} \tag{14}$$

The scaled returns are passed through a sigmoid, which means that kappa final value will be proportional to the range between max and min bounds of kappa where that proportion is determined by the returns produced by the corresponding z. Then, this target will be the input of the Huber loss for kappa. Although, it can be use an MSE loss too.

Finally, there is an optional term, KL-DIVERGENCE term. The idea of this term is to naturally provide a consistent exploration. The authors in the paper offered to set resampling of z after certain steps, but it could be controlled naturally by restricting the distributions change in this case. However, this term is not explored give the time constrain.[10]

Is important to notice that we develop each method with PyTorch, even considering that there is no Bessel function of higher order than 1 in it. However, there is a beautiful property of Bessel functions that allow to write the higher order

[10]The respective method is implemented in the code to be used

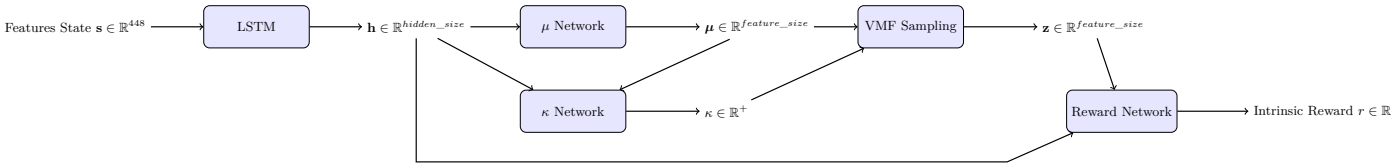
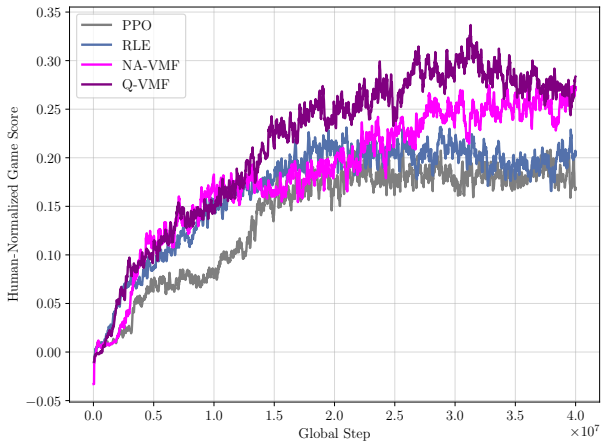
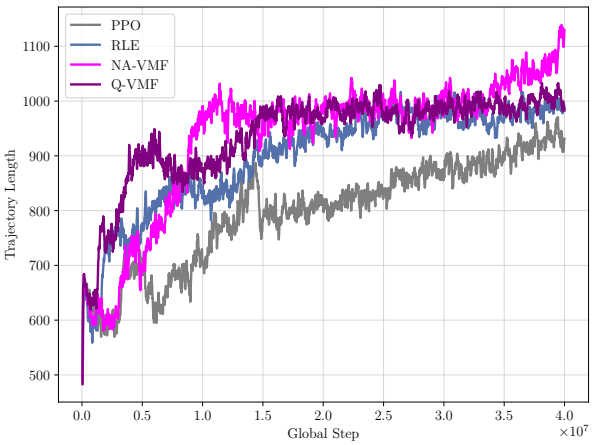


Figure 15: Neural-adaptive VMF architecture.



(a) Game Score.



(b) Trajectory Length.

as a function of the previous two.[11] Since the first two orders of the first kind Bessel functions are implemented, then we had all the ingredients to make a perfect differentiable entropy and KL-DIVERGENCE term which will be necessary for BPTT. In addition, could be interesting to test what would happen if we keep the cell (long term) memory of the lstm between episodes of the same parallel environment.

I Difficulties

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. 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.

There are not many documentation neither active blogs about IsaacLab, which it makes the migration highly difficult considering the higher complexity with respect to isaacgym. In addition, the deprication of isaacgym generates uncompatibilities with pre installed cuda libraries in providers as Lambda or Google which requires uninstalling and installing older versions which can generate other issues with other pre installed libraries or requires a high amount of time debugging NVIDIA related issues.

[11] `https://proofwiki.org/wiki/Recurrence_Formula_for_Bessel_Function_of_the_First_Kind`

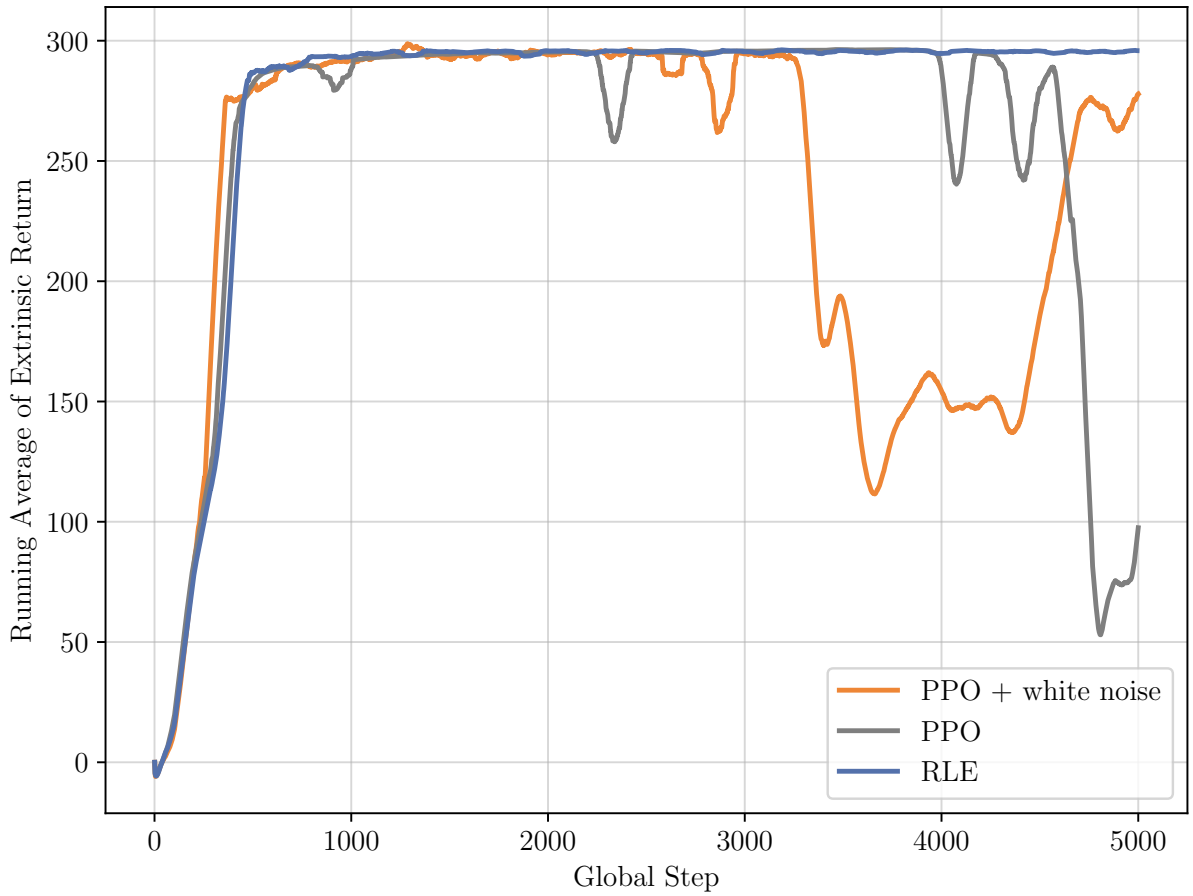


Figure 17: Cartpole Extrinsic Returns.