@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)?

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

Table 1: 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 $\lambda$ | 0.95 |
| Gradient Norm Bound | 1.0 |
| Used Clipped Value Loss | False |
| Value Loss Scale | 2.0 |
| Timesteps | 5000 |

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.

### 0.0.1 IsaacGym

@Jonathan explain environment and experiments (NOT RESULT) In the case for this implementation, the code provided was develop in IsaacGym, which is depricated. [1] 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.

[1] 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`

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 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.

### 0.0.2 Neural Adaptive Von Mises-Fisher

@Jonathan
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 (ARCHITECTURE FIGURE LINK HERE). 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.

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

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

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

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

$$H(\text{vMF}) = -\log(C_d(\kappa)) - \kappa A_d(\kappa) \tag{2}$$

where:

$$C_d(\kappa) = \frac{\kappa^{d/2-1}}{(2\pi)^{d/2} I_{d/2-1}(\kappa)} \tag{3}$$

$$A_d(\kappa) = \frac{I_{d/2}(\kappa)}{I_{d/2-1}(\kappa)} \tag{4}$$

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

$$\log(C_d(\kappa)) = (d/2-1)\log(\kappa) - \frac{d}{2}\log(2\pi) - \log(I_{d/2-1}(\kappa)) \tag{5}$$

Therefore, the complete entropy expression is:

$$L(\text{vMF}) = -\left[(d/2-1)\log(\kappa) - \frac{d}{2}\log(2\pi) - \log(I_{d/2-1}(\kappa))\right] - \kappa \frac{I_{d/2}(\kappa)}{I_{d/2-1}(\kappa)} \tag{6}$$

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:

$$\mathcal{L}_{huber} = \sum_{t=1}^{T} \begin{cases} \frac{1}{2}\left(r_t - \hat{r}_t\right)^2, & \text{if } |r_t - \hat{r}_t| < \delta \\ \delta\left(|r_t - \hat{r}_t| - \frac{1}{2}\delta\right), & \text{otherwise} \end{cases} \tag{7}$$

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 $\mu$. 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 $\mu$, allowing for exploration. (ADD PLOT HERE)

$$\kappa_{target} = \kappa_{min} + (\kappa_{max} - \kappa_{min}) \cdot \text{returns}_{scaled} \tag{8}$$

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. [2]

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 as a function of the previous two. [3]. 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.

## Computational Requirements

@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?

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

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

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

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).

## What was difficult?

There are not many documentation nither 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.

## Member contributions

- Jonatan Bella: VMF algorithms development and experimentation, IsaacLab implementation (rewriting of each component of the project code, environment modifications for RLE, and final experimentation)