Final Project

Francesco Spena 1911090

1 Environment

The Acrobot-v1 environment in Gymnasium is a classic dynamic control problem where the goal is to swing up a double inverted pendulum to reach a specific height. Here's a detailed description of the environment:

1. System:
   - The system consists of two rigid links connected by a rotational joint.
   - The first link is fixed to a static pivot point at the top.
   - The second link is attached to the end of the first link and can rotate freely.
2. Goal: The objective is to swing the free end of the second link above a specified horizontal threshold, starting from an initial position where both links hang downward.
3. Controls:
   - The action applied is a torque on the second joint (the joint between the two links).
   - The available actions are discrete: apply positive torque, negative torque, or no torque.
4. Observations: The observable state is a 6-dimensional vector that includes the sine and cosine of the angles between the links, as well as the angular velocities of the joints.
5. Dynamics:
   - The environment is subject to gravity and the physical laws of pendulums.
   - The agent must exploit the pendulum effect and kinetic energy to achieve the goal.
6. Reward: A reward of -1 is given for each step, encouraging the agent to achieve the goal in the shortest time possible.

2 Paper

The paper is **dream to control: learning behaviors by latent imagination**.
The PDF file is in the repo folder.

Idea of the paper: We design two neural networks to approximate the environment's transition model and reward model, enabling the simulation of sequences of (state, action) pairs. Using these simulated trajectories, we estimate the value function and update both the value model and the policy model following an actor-critic framework. The policy model generates an action, which is then executed in the environment. The resulting new data is collected and added to the original dataset (replay buffer). This process is repeated for each episode.

3 Implementation

3.1 Transition Model

The transition model is a neural network capable of predicting the next latent state s_{t+1} given the current state s_t and the current action a_t. The network can be formulated as:

$$s_{t+1} = f_\theta(s_t, a_t) \tag{1}$$

where θ represents the network parameters. To train the model, it is necessary to collect a dataset of experiences from the environment, where each experience must include:

1. Current state s_t
2. Executed action a_t
3. Next state s_{t+1}

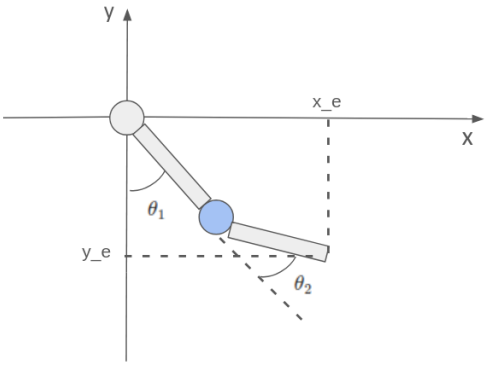
In this case, a MLP (Multi-Layer Perceptron) was chosen, consisting of two hidden layers with 128 units each and ReLU activations (to avoid the vanishing gradient problem). The state dimension corresponds to the state representation in the Acrobot environment, and the action dimension corresponds to the torque applied to the second joint.

3.1.1 Dimension of the state space

The acrobot environment represent a system with a double pendulum with the following parameters:

1. $\cos\theta_1$: Cosine of the angle of the first joint relative to the vertical.
2. $\sin\theta_1$: Sine of the angle of the first joint relative to the vertical.
3. $\cos\theta_2$: Cosine of the angle of the second joint relative to the first joint.
4. $\sin\theta_2$: Sine of the angle of the second joint relative to the first joint.
5. $\dot{\theta}_1$: Angular velocity of the first joint.
6. $\dot{\theta}_2$: Angular velocity of the second joint.

Acrobot is a double pendulum, where $\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2$ represent the angle of the first joint relative to the vertical, the angle of the second joint relative to the first joint, and the angular velocities of the joints, respectively. Using the angles directly can lead to ambiguity and discontinuity due to their periodic nature. For this reason, a representation based on sine and cosine is preferred.



The equations describing the coordinates of the manipulator's end-effector are given by:

$$\begin{cases} x_e = l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2) \\ y_e = -l_1 \cos(\theta_1) - l_2 \cos(\theta_1 + \theta_2) \end{cases} \tag{2}$$

and the velocities of these coordinates are given by:

$$\begin{cases} \dot{x}_e = l_1 \dot{\theta}_1 \cos(\theta_1) + l_2(\dot{\theta}_1 + \dot{\theta}_2)\cos(\theta_1 + \theta_2) \\ \dot{y}_e = l_1 \dot{\theta}_1 \sin(\theta_1) + l_2(\dot{\theta}_1 + \dot{\theta}_2)\sin(\theta_1 + \theta_2) \end{cases} \tag{3}$$

The dynamic are derived from the Lagrange equations and take the form:

$$M(\theta) \begin{pmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \end{pmatrix} + C(\theta, \dot{\theta}) \begin{pmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \end{pmatrix} + G(\theta) = \begin{pmatrix} 0 \\ \tau_2 \end{pmatrix} \tag{4}$$

The equations of motion update the angles and the angular velocities. In fact, we can obtain:

$$\begin{cases} \dot{\theta}_1' = \dot{\theta}_1 + \ddot{\theta}_1 \triangle t \\ \dot{\theta}_2' = \dot{\theta}_2 + \ddot{\theta}_2 \triangle t \end{cases} \tag{5}$$

$$\begin{cases} \theta_1' = \theta_1 + \dot{\theta}_1' \triangle t \\ \theta_2' = \theta_2 + \dot{\theta}_2' \triangle t \end{cases} \tag{6}$$

From this, it is possible to compute:

$$\begin{cases} \cos(\theta_1') = \cos(\theta_1 + \dot{\theta}_1' \triangle t) \\ \sin(\theta_1') = \sin(\theta_1 + \dot{\theta}_1' \triangle t) \\ \cos(\theta_2') = \cos(\theta_2 + \dot{\theta}_2' \triangle t) \\ \sin(\theta_2') = \sin(\theta_2 + \dot{\theta}_2' \triangle t) \end{cases} \tag{7}$$

In this way, we can represent the state without ambiguity.

3.1.2 Dimension of the action space

The control system of this environment is discrete, where the action represents a torque applied to the second joint:

1. Apply positive torque (+1)
2. Apply negative torque (-1)
3. Apply no torque (0)

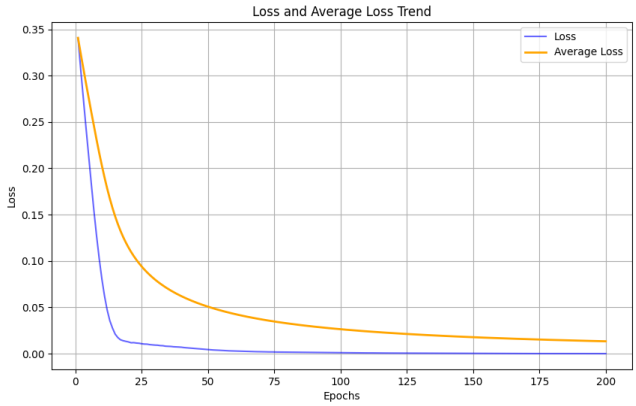
Two hidden layers with 128 units represent a good compromise between generalization and computational efficiency. Acrobot is a non-linear system with complex dynamics, this network design effectively captures these dynamics without requiring an explicit model. So the dimension of the **state** is equal to **6** and for the **action** is equal to **3**.

3.1.3 Training

The training of this model has this hyperparameters:

batch size	epochs	lr	num. ep.
32	200	1e-4	10

With the file **train-transition.py** on the repository we obtain an excellent convergence of the model:



3.2 Reward Model

The reward model is a neural network capable of predict the reward r_t given the state and the action at time t. The network can be formulate:

$$r_t = f_\theta(s_t, a_t) \tag{8}$$

with θ the parameters. In general, the reward function evaluates how desirable a given action is in a particular state. The purpose of this network is to learn such a function so that, together with the transition model, it can create models capable of simulating the environment without interacting directly with it. It is an MLP with two hidden layers, each with 32 units and a ReLU activation function. It produces a scalar output corresponding to the predicted reward associated with a given state. Since the considered environment is relatively simple, the network can be something relatively uncomplicated.

3.2.1 Training

The training of this model has this hyperparameters:

batch size	epochs	lr	hidden
32	50	1e-3	[32,32]

Also in this case, with the file **train-reward.py** we obtain an excellent convergence of the model:

In both cases, the loss quickly converges to 0. This is because both the transition and the reward function, in a simple environment like this, are very easy to approximate. If the state, for example, were represented by images (as described in the paper), it would be necessary to construct convolutional networks, which would require more effort to achieve the same results.

3.3 Value Model

The value model estimates the value associated with a given state. It is a network composed of 3 hidden layers with sizes [128, 64, 32] and a ReLU activation function. The output is a scalar value representing the expected value of the state. It is used during the training phase to evaluate the quality of the actions taken relative to the estimated value of the state.

3.4 Discount Model

The discount model predicts the discount factor that represents the probability of continuing the episode given the current state. It has a single hidden layer with 64 units and a ReLU activation function. The

output is a single probability obtained through a Sigmoid function. It is used to weigh future returns during the calculation of value targets, especially in environments with early termination.

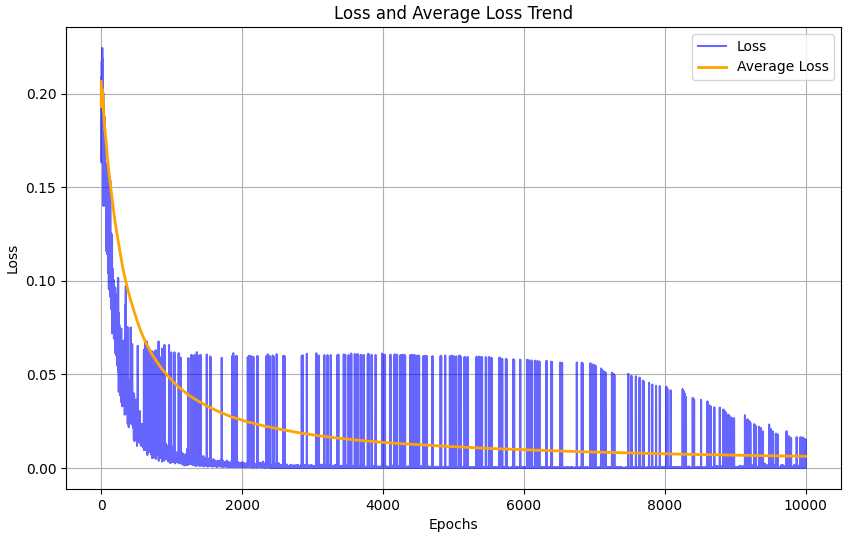


Figure 1: Discount Loss

3.5 Policy Model

The policy model represents a policy that maps a state to the probabilities of actions to take. It is a network composed of a hidden layer with 128 units and an ELU activation function. The output is a probability distribution over the possible actions. It is used to select actions during interaction with the environment and to update the policy using the Actor-Critic method.

3.6 Dreamer

To train the Dreamer we can follow the pseudocode specified in the paper and modify it to adapt for our case.

Algorithm 1: Dreamer

Initialize dataset $\mathcal{D}$ with S random seed episodes.
Initialize neural network parameters θ, ϕ, ψ randomly.
while *not converged* **do**
  for *update step* $c = 1..C$ **do**
    `// Dynamics learning`
    Draw B data sequences $\{(a_t, o_t, r_t)\}_{t=k}^{k+L} \sim \mathcal{D}$.
    Compute model states $s_t \sim p_\theta(s_t \mid s_{t-1}, a_{t-1}, o_t)$.
    Update θ using representation learning.
    `// Behavior learning`
    Imagine trajectories $\{(s_\tau, a_\tau)\}_{\tau=t}^{t+H}$ from each s_t.
    Predict rewards $\mathrm{E}\big(q_\theta(r_\tau \mid s_\tau)\big)$ and values $v_\psi(s_\tau)$.
    Compute value estimates $\mathrm{V}_\lambda(s_\tau)$ via Equation 6.
    Update $\phi \leftarrow \phi + \alpha \nabla_\phi \sum_{\tau=t}^{t+H} \mathrm{V}_\lambda(s_\tau)$.
    Update $\psi \leftarrow \psi - \alpha \nabla_\psi \sum_{\tau=t}^{t+H} \frac{1}{2} \big\| v_\psi(s_\tau) - \mathrm{V}_\lambda(s_\tau) \big\|^2$.
  `// Environment interaction`
  $o_1 \leftarrow$ `env.reset()`
  for *time step* $t = 1..T$ **do**
    Compute $s_t \sim p_\theta(s_t \mid s_{t-1}, a_{t-1}, o_t)$ from history.
    Compute $a_t \sim q_\phi(a_t \mid s_t)$ with the action model.
    Add exploration noise to action.
    $r_t, o_{t+1} \leftarrow$ `env.step(`a_t`)`.
  Add experience to dataset $\mathcal{D} \leftarrow \mathcal{D} \cup \{(o_t, a_t, r_t)_{t=1}^{T}\}$.

Model components

Representation	$p_\theta(s_t \mid s_{t-1}, a_{t-1}, o_t)$
Transition	$q_\theta(s_t \mid s_{t-1}, a_{t-1})$
Reward	$q_\theta(r_t \mid s_t)$
Action	$q_\phi(a_t \mid s_t)$
Value	$v_\psi(s_t)$

Hyper parameters

Seed episodes	S
Collect interval	C
Batch size	B
Sequence length	L
Imagination horizon	H
Learning rate	α

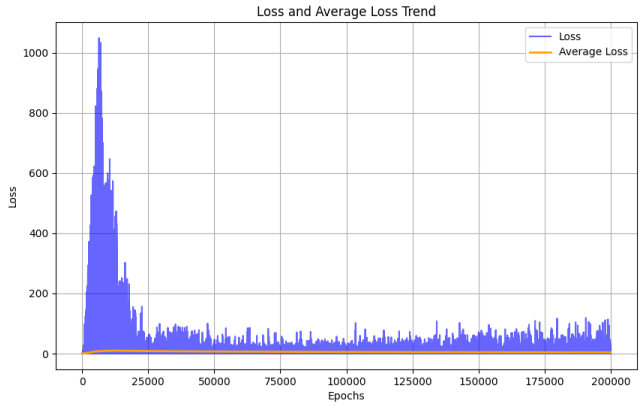
3.7 Results

We can use the script **train-actor-critic.py** to train the Policy Model with an AC schema. In the following table some hyperparameters:

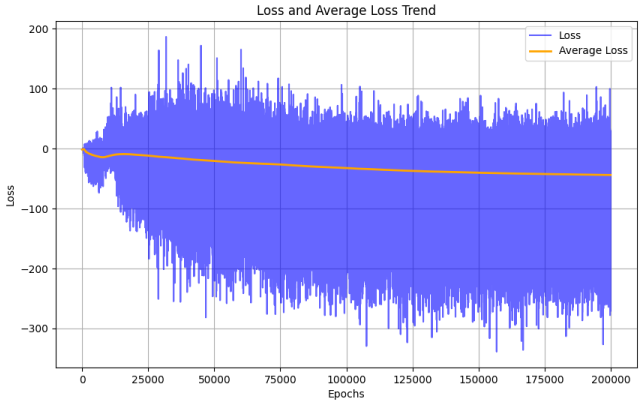
num episodes	imagination horizon	batch size	gamma	lambda	lr
200.000	30/25/15	2/8/64	0.99	0.95	1/3/5/10 (e-4)

Gamma controls how much weight to give to future rewards. Lambda controls the temporal horizon of the TD errors used to estimate the advantage, balancing bias and variance. If gamma is close to 1, it means the agent highly values future rewards, whereas if lambda is close to 1, it considers advantages calculated over longer horizons, reducing bias but increasing variance. So, in the follow we have a list of results by change this values:

- With a batch size = 8, imagination horizon = 25, and the learning rate = 3e-4 we have this behavior for the value and policy model:



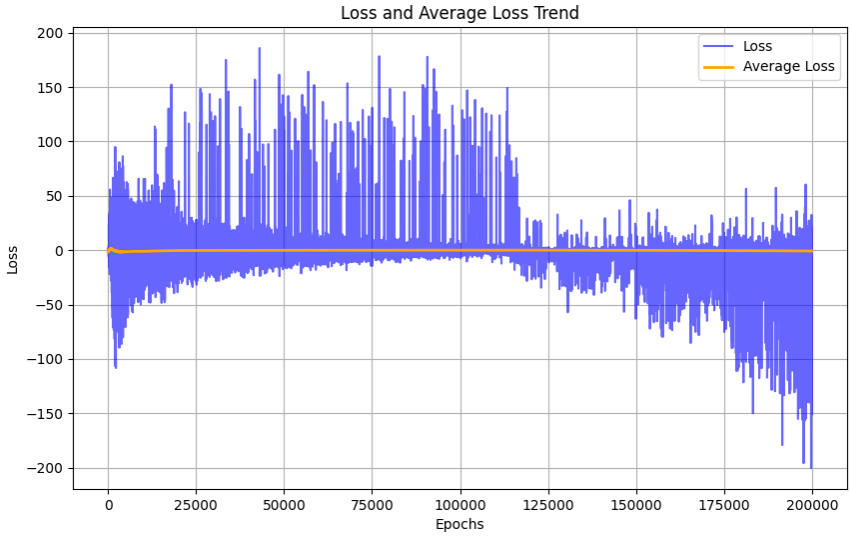
(a) Value Model



(b) Policy Model

Figure 2: Experiment 1

with a learning rate = 10e-4 we have, for the policy model, this loss:



So, if we set this big value for lr we have an instability in the training phase.

We try to change the loss function on the value mode to see if a different approach improve the performance of the model:

1. Original: Mean Square Error
2. Huber Loss defined as (Smooted L1 Loss):

$$\begin{cases} \frac{1}{2}(a-b)^2 & |a-b| \leq \delta \\ \delta \cdot |a-b| - \frac{1}{2}\delta^2 & otherwise \end{cases} \tag{9}$$

   This function is less sensitive to outliers than Mean Squared Error (MSE).

3. Regularization based loss:

$$L = L_{value} + \lambda ||\theta||_2 \tag{10}$$

   To prove to stabilize the training phase.

If we try this different loss we have this results:

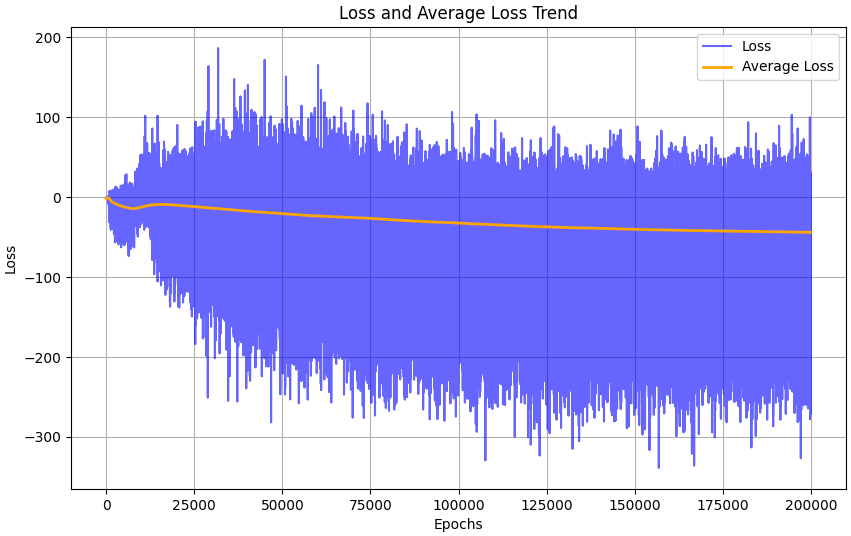


Figure 3: Original Loss

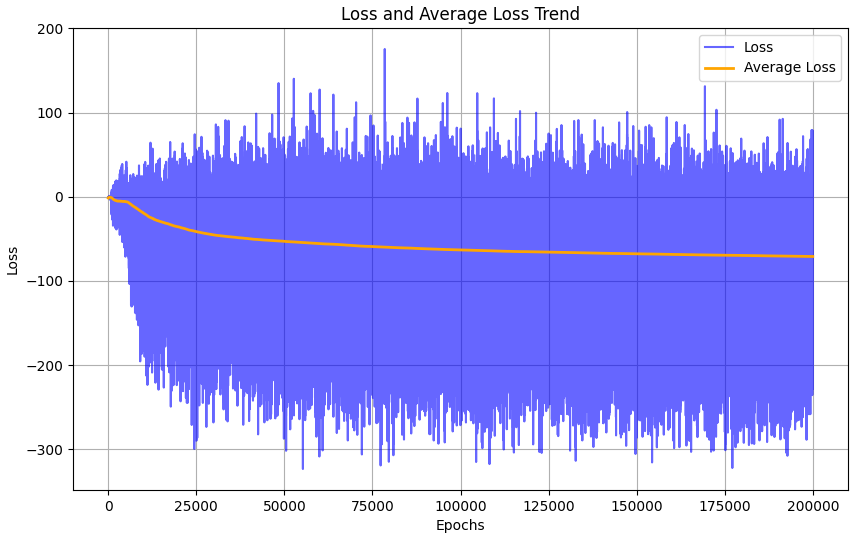


Figure 4: Huber Loss

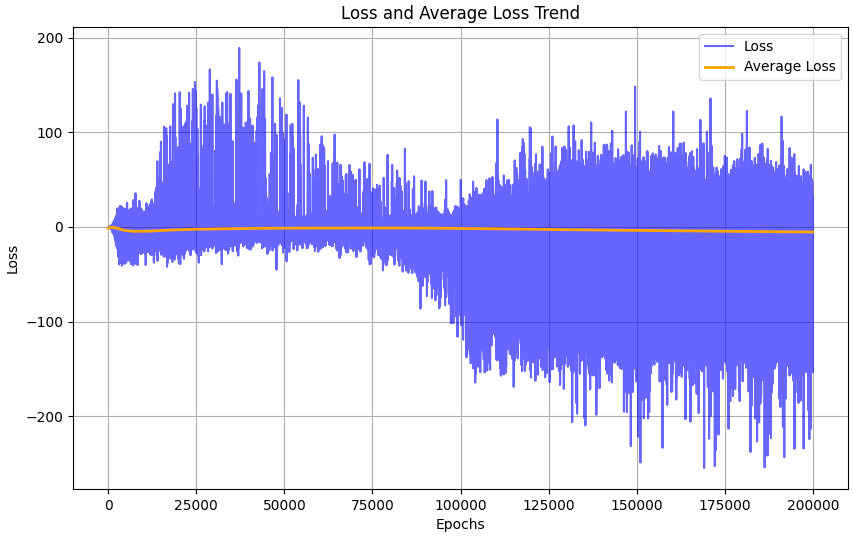


Figure 5: Reg Loss

At the following also the behavior of the mean reward, with the three different model obtain by these loss, in 1000 episodes:

Figure 6: Original Reward

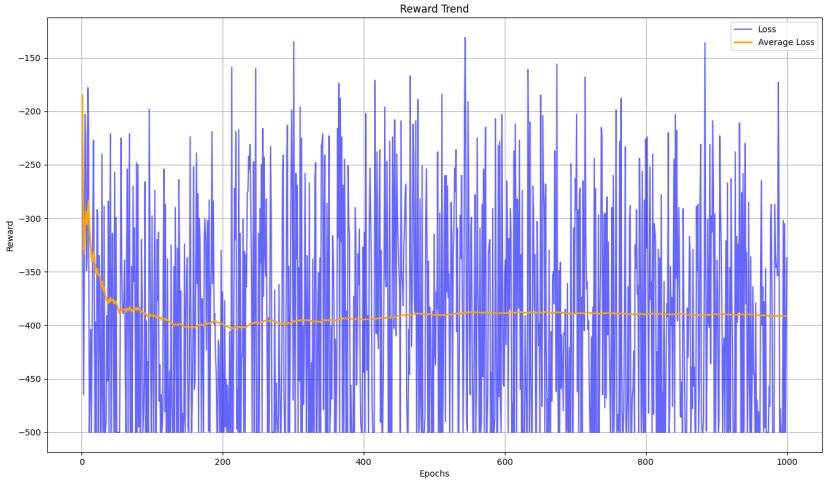


Figure 7: Huber

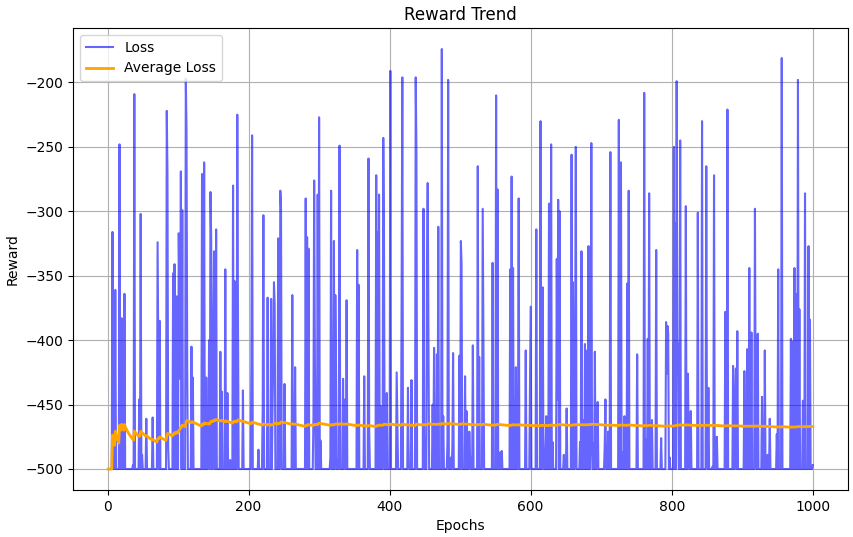


Figure 8: Reg

If we compare this algorithm with another, for example PPO we obtain this type of behavior in terms of mean reward (10.000 episodes):

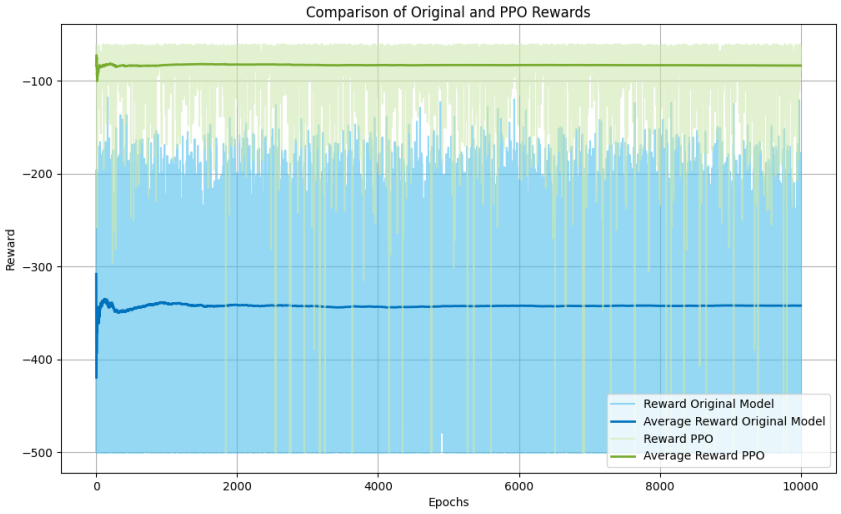


Figure 9: Reward Original vs PPO

One of the reasons why PPO performs better than the proposed algorithm, aside from the available computational capacity, could be due to the fact that the algorithm proposed in the paper uses a classic policy update based on policy gradients. This can result in large variations in the policy, making the training unstable. PPO, on the other hand, introduces a trust region constraint by limiting the magnitude of the policy update. This is achieved through a clipping term, ensuring that the update stays within a controlled range. Specifically, we have:

$$L^{PPO}(\theta) = E[min\ (r_t(\theta) \cdot A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon) \cdot A_t)] \tag{11}$$

where r_t is the relation between the new and old probability of the action. The experiences collected by the proposed algorithm are used similarly to on-policy methods, while PPO makes better use of these experiences because it allows the same batch of data to be used for multiple policy updates until the probability ratio exits the established constraint. Although entropy and gradient norm clipping terms were added, PPO performs better because it successfully balances the amount of exploration and exploitation, and performs clipping more naturally thanks to the probability constraint. The last prove that we have try to improve the performance is to add the KL divergence in the training step to update the policy. The KL divergence is used to measure the difference between two probability distributions. In our context, it can be used to regularize the distribution of predicted actions with respect to a target distribution. This should help stabilize the training phase to some extent.

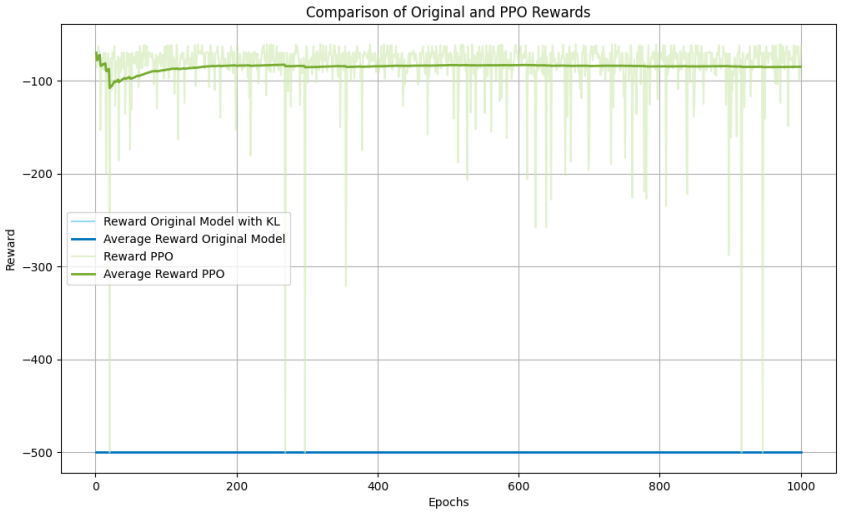


Figure 10: Reward Original KL vs PPO

It is very likely that, since this aspect was added, the policy is overly constrained to follow the target distribution, limiting its ability to explore and adapt to the actual reward signal. In this context, broad exploration is crucial.

3.8 World Models

The method proposed in our paper in based on the World Models architecture. A World Model is a generative model that learns to predict the evolution of the world's states based on past experiences. These models learn an internal representation of the world in which the agent operates. These internal models allow simulating experiences in the environment without having to interact with it directly. It is based on three main components:

- Representation Model: Transforms high-dimensional observations (such as images) into a latent space.
- Transition Model: Predicts how latent states change in response to the agent's actions, allowing it to "imagine" the future.
- Reward Model: Estimates future rewards based on latent states.

In our case, we do not need the Representation Model because we do not have a high-dimensional state; instead, it consists of 6 variables. However, we do have the Transition and Reward Models, which do not take a latent state representation as input but rather the original state.