Lexicographic Reinforcement Learning Benchmark on MuJoCo environment

Symbolic and Evolutionary Artificial Intelligence a.a. 2022/2023

Michele Lisi

September 2023

Contents

0.1 Introduction

Reinforcement learning, a subfield of artificial intelligence, has emerged as a powerful paradigm for training intelligent agents to make decisions in dynamic and complex environments.
This report presents a comprehensive exploration of various aspects of reinforcement learning, drawing insights from a series of experiments conducted in simulated environments.

The goal of this project is to create a benchmark to see the advantages of Lexicographic Multi-Objective Reinforcement Learning (LMORL) over classic Reinforcement Learning (RL).
The main difference in these two approaches is that, while RL is inherently a single objective optimization problem, LMORL has the possibility to optimize multiple objectives hierarchically ordered based on their importance.

0.2 Reinforcement Learning Environments

The environment chosen for the benchmark belongs to Gymnasium, which is an open source Python library for developing and comparing reinforcement learning algorithms that provides a standard API to communicate between learning algorithms and environments.

Formerly named Gym, this project was created by OpenAI, and now holds

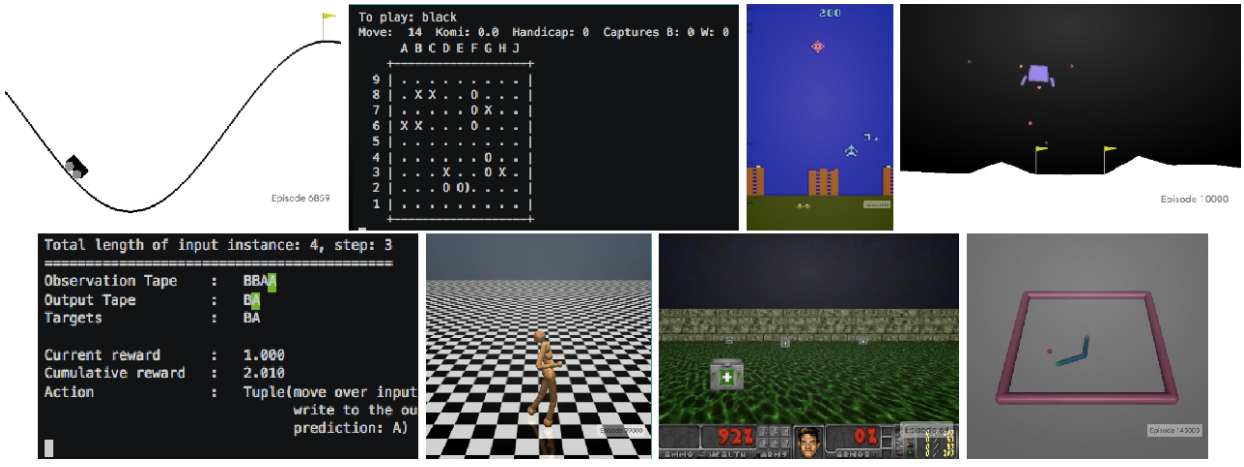

Figure 1: Images of several Gymnasium environments

several different environments that vary from a simple inverse pendulum, to complex scenarios, such as MuJoCo environments or Gymnasium - Robotics.

The former, MuJoCo, stands for Multi-Joint dynamics with Contact. It is a physics engine for facilitating research and development in robotics, biomechanics, graphics and animation, and other areas where fast and accurate simulation is needed.
The state spaces for MuJoCo environments in Gymnasium consist of two parts that are flattened and concatenated together: a position of a body part or joint and its corresponding velocity.

Among Gymnasium environments, this set of environments can be considered as more difficult ones to solve by a policy.

0.2.1 Ant-v4

Ant is based on the framework introduced by Schulman, Moritz, et al. in their paper[3]. Within this environment, there is a 3D robotic ant consisting of a central torso, which is a freely rotatable body.
Attached to this torso are four legs, and each of them comprises two body parts connected by a hinge. In total, there are nine body parts and eight hinges in the ant robot.

The objective of this environment is to coordinate the movement of the four

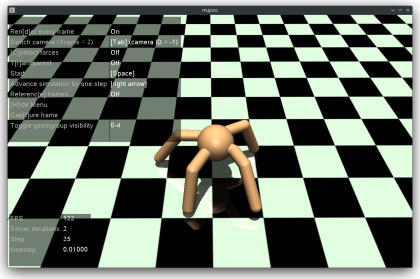

Figure 2: Ant v4 environment

legs to move the robot forward, which is in the rightward direction. This coordination is achieved by applying torques to the eight hinges, which connect the two body parts of each leg to the torso.

The state space (or observation space) consist of informations on the position and velocity of every body part belonging to the Ant, totaling 29 entries to describe its state.
While normally excluded, it is possible to add 84 entries to the state space,

extending it with the contact forces, represented by the x, y and z components of torques and forces applied to the center of mass of every limb belonging to the Ant.

The action space is made of 8 possible actions, representing every possible movement between two limbs of the agent and bounded to an interval of -1.0 and 1.0:

Num	Action	Control Min	Control Max	Joint
0	Torque applied on the rotor between the torso and back right hip	-1	1	hinge
1	Torque applied on the rotor between the back right two links	-1	1	hinge
2	Torque applied on the rotor between the torso and front left hip	-1	1	hinge
3	Torque applied on the rotor between the front left two links	-1	1	hinge
4	Torque applied on the rotor between the torso and front right hip	-1	1	hinge
5	Torque applied on the rotor between the front right two links	-1	1	hinge
6	Torque applied on the rotor between the torso and back left hip	-1	1	hinge
7	Torque applied on the rotor between the back left two links	-1	1	hinge

Table 1: List of actions that can be performed by the Ant

The possible rewards calculated are the following:

- **Healthy Reward:** a fixed reward (1.0) given every timestamp that the ant is considered healthy (which is defined below).
- **Forward Reward:** reward calculated as the difference between the x position of the Ant before and after the action, divided by `dt`, which is the time between frames. The reward is positive if the movement is in the positive direction of the x axis.
- **Control Cost:** a negative reward to penalize actions too large. It is calculated as the sum of the squared actions taken, multiplied by a weight factor

- **Distance from Origin:** represents the distance from the spawn point and is calculated similarly to the Forward Reward, but using the xy position instead of the x position

The standard reward for this environment is the sum of the Forward Reward, the Healthy Reward and the Control Cost, but it won't be the only one used.

As for the termination condition, the environment resets if any of the state space values are no longer finite, or if the Ant is considered not healthy, meaning that its torso's z coordinate is no longer in the healthy range ([0.2, 1.0]).

0.3 DQN Agents

As stated earlier, one of the available environments that the OpenAI Gym toolkit offers is the ATARI selection: a collection of video game environments designed to facilitate reinforcement learning research by providing a diverse set of video games as testing grounds for training and evaluating reinforcement learning agents.

A standard DQN can easily solve this kind of environments given the fact that the state space is often continuous, while the action state is, for most of the environment, discrete: the action state represents different actions which the agent can perform or not.

This, though, poses as a limit for the application of the standard DQN: every MuJoCo environment, for example, has a continuous action space, posing the action as a range of values it can have.
This could be the case of controlling the angle of a servo motor or the speed of a brushless motor, remaining in the robotics field.

Given the fact that the chosen environment for this project belongs to the MuJoCo collection, the implemented network(s) will have to produce an output coherent with the action space.
The 'Ant-v4' environment has an action space made up by eight actions which can vary in value from -1.0 to 1.0, representing the angle of the two segments that represent the Ant's four limbs.

0.3.1 Continuous DQN

The solution is then to implement a Continuous DQN agen to use in this environment, sampling the action space.
While the vanilla DQN's output is the probability of one action to be chosen, the Contnuous DQN's output whill be the probability of one sample to be chose, for every action. Given N sample used to discretize the interval [-1.0, 1.0], the output will be a matrix of dimensions $8 \times N$.
It is worth noting that the sample have to be centered in 0.0 ad must contain both -1.0 and 1.0, otherwise the action space would be unbalanced.

The code implementation of the network as well as the structure of the agent's classes and methods is based on the implementation of lmorl[1], a GitHub repository which proposes different agents (both standard and lexicographic) to solve (outdated) Gym environments.

Although being similar in structure, this implementation will be substantially different in the functions, for different choices in the definition of the agent.

The network

The network is a simple DNN with an input shape of 27, Ant's state space excluding contact forces, and an output shape of N $\times$ 8 as previously stated.
A total of three linear layers are used where all but the last one use Relu as activation function.

So far, four hyperparameters can be identified: the number of samples (N), the dimension of the intermediate layers, the use or not of the second layer, and wether or not the layers make use of bias.

The agent

The base agent is implemented with a Double DQN, thus having a behavioural network, updated during the interaction between the agent and the environment, and a target network, updated at the end of the episode.

An Experience Replay Buffer with random sampling has been implemented, as well as the ϵ-greedy exploration: the agent will start with an epsilon value of 0.9, thus prioritizing exploration by performing, with a probability of ϵ, random actions.
After several episodes, the epsilon will start to decay by being multiplied to a decay factor of 0.99, decreasing the probability of chosing a random action over the network's output.

A minor difference worth noting is the interaction loop: instead of using a single loop counting the total interactions between the agent and the environment, like the lmorl implementation, the training will be divided into episodes which represent the totality of interaction between the spawn of the agent and the termination condition met.

0.3.2 Lexicographic Agent

The evolution of a standard agent into a lexicographic agent involves transitioning from a single-objective framework to a multi-objective one, where the agent learns to optimize its actions based on a hierarchy of multiple rewards, each assigned a specific order of importance.

Initially, the approach considered the use of BANs to facilitate non-Archimedean

computation, aiming to approximate the first reward as a primary objective while treating the subsequent rewards as ‘infinitely‘ less significant.
However, as we will explore in later sections, some problems led to the abandonment of this solution.

In the absence of BANs, the alternative method to create a lexicographic agent involves an adjustment to the agent's neural network architecture. Specifically, the network's output size is tripled, assuming there are three rewards in the hierarchy.
This expansion allows the agent to calculate Q-values for each of the individual rewards. Subsequently, the agent updates its weight parameters based on its performance concerning these rewards.

0.4 Training Environment

The training environment has been split on two machines and with implementations made both in Python and Julia programming language.

The first implementation of the Continuous DQN has been written in Python. Given its high simplicity and versatility, the code has been initially implemented in this language, which has made debugging and troubleshooting fairly easy.

In a second moment, also thanks to the similarities in syntax between Python and Julia, the code has been ported to the latter language, with the purpose of extending the model to a lexicographic one using a library built to execute BAN's calculus in Julia.

During this phase of the project, the implementation was, function-wise, complete, but, due to a still unknown fault, the code is not able to train the agent and make it walk.
All the simulation executed showed the Ant, still in place, with the network's output being the same every interaction, ignoring the changes in the state.

After several hours spent into troubleshooting with no result, the choice was made to continue the work in Python, with a slight change on the implementation, being unable to use the library built for Julia, but keeping the same scope of the project.

At this point, instead of realizing a DQN able to learn from rewards in the form of BANs, the goal was to realize a DQN which keeps the lexicographic aspect, using hierarchically ordered rewards, but which optimizes exclusively one of them based on their loss value.

While this agent was being implemented, the trainings with the standard Continuous DQN were started.

0.4.1 Initial training environment

The first batch of training has been performed on an MSI GL75 9SE laptop, with a RTX2060 GPU and an i7 9750H CPU.
Altough having a relatively good GPU for network training, it has been found that using it was slowing the training speed, so everything was moved to the CPU.

This can be explained by a combination of factors. The size of the network is surely one of the reasons: while the GPUs are optimized for the huge number of matrix computation used by most type of networks, the networks used are simple DNNs, with a number of weights far smaller than one used for object

detection, for example.
For this reason, the time difference between CPU and GPU training is less evident.

Furthermore, another obstacle is the physics computation, which is made exclusively on the CPU.
Given that the state and the action's data are essential both to the network and the physics, there will be a frequent exchange of data between the CPU cache and GPU cache, being a considerable bottleneck.

The Python implementation of the training loop uses `torch` as its choice for tensor and network handler, `numpy` for data and the `SummaryWriter` from TensorFlow for metrics writing.

The training loop simply iterates for the chosen number of episodes and for every interaction made until the termination condition is met which, in this case, is if the Ant is in a 'healthy' position or not.
This 'healty position' is a range along the z axis in which the torso of the Ant needs to be to be considered healthy.

Once the the termination condition is met, the episode is considered ended and the following metrics are saved:

- Cumulative network reward obtained during the episode
- Cumulative informations on other rewards:
  - Reward Forward
  - Healthy Reward
  - Control Cost
  - Distance from the Origin
- Average cumulative network reward respect to 100 last episodes
- Average cumulative informations on other rewards

The model is saved every few epochs and, in addition, a separate copy of the model is saved if its cumulative reward is higher than the last max reward.

After the training is complete, the best performing model is loaded used for testing. In this phase, the function that handles the computation of the action from the Q values, does not use the ϵ-greedy feature, returning only actions obtained from the network.
In addition to this, other termination condition are added, such as if the network reward reaches a max threshold, or if the total interactions reach it.

These limitations were implemented due to some unwanted outcomes that can put the test process in an endless loop.
An interesting result that led to this implementation is the behaviour of one of the network: during the test phase, this specific agent started performing the same two slightly different actions, resulting in the Ant 'vibrating' and slowly but surely moving in one direction.

This behaviour, altough far from the practical purpose of training a walking agent, is relevant because minimizes the cost of the actions while maximizing both its distance from the origin and healthy reward.

Another termination condition worth noting is due to some networks stopping their walk to stay still, due to the network stuck performing the same action.
This situation, altough mostly present in the first phases, has been quite an inconvenience, so it has been dealt with the episode ending if the network performs the same action for two consecutive interactions.

During the test, the same metrics of the train phase are saved, with the addition of a signal if the Ant has stopped and if not moving, as described before.

0.4.2 Data Analysis

As mentioned before, the SummaryWriter from TensorFlow is being used to save metrics about training and testing phases.
Consequently, initially the data was read and compared using TensorBoard, a web interface to which the data is uploaded and directly plotted, only separated by a single run.

While being a useful tool, TensorBoard was limited for the scope of comparison of this project, lacking the correct grouping based on network feature or reward type, rather than the singe run all toghether.
For this reason, during the training of the Continuous DQNs, a simple, yet powerful offline alternative of TensorBoard has been created and named `tf_reader`[2].

This program, mostly build using `tkinter` for the GUI, `matplotlib` for the plot generation, and `tbparse` for reading the binary files generated by the SummaryWriter, automatically groups the various training following a specific folder structure, and differs model feature and reward types with the use of tags.

So, after choosing the trainings with the desired model features and reward types, the preferred metrics can be plotted.
Beside this, it is possible to save individual plots as well an image containing all the plots created, and a list, ordered by the desired test metric performances' max values, of all the plotted trainings.

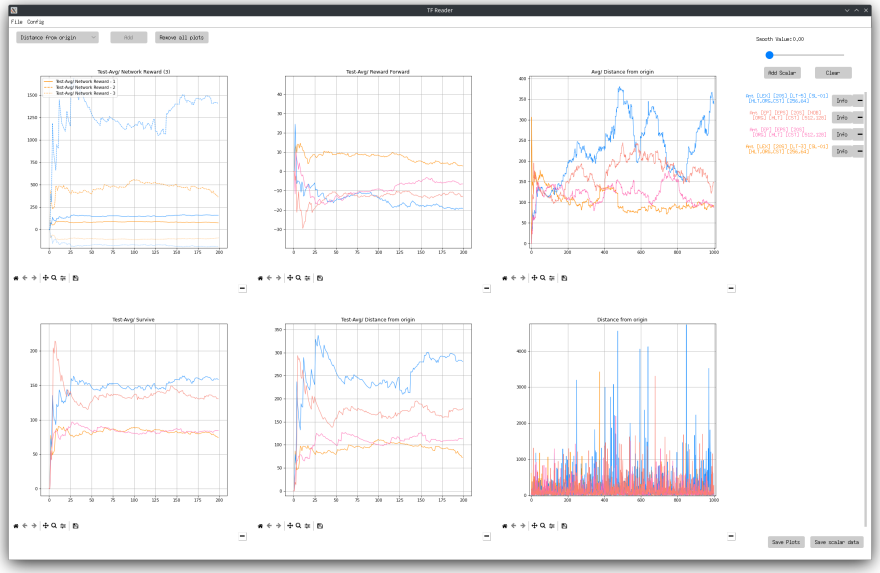

Figure 3: tf_reader main window

This tool has been crucial to visualize and compare every sort of data and metrics of training and test phases.

0.4.3 Lexicographic agent implementation

During the implementation of the Lexicographic DQN, an hardware fault has rendered the MSI laptop unusable, hence, a change in the machine has been made, which now mounts an RTX 4060 and an i7 13700H.
Besides the change in machine, the rest of the environment remains unchanged. Every consideration made for the Continuous DQN is used during the training and testing of the Lexicographic agent.

The metrics used are the same, with the difference that the cumulative network reward and its average have now three entries, along with their equivalent test metrics.
Another metric is now kept track of: the average loss per episode. This metric, with three entries, is the average of average loss during episode in respect to the lexicographic rewards.

This metric is used to keep track of which loss, on average, is being minimized.

This can be inferred because, during the update of the behavioural network, only the loss value actually backpropagated is saved, while the other are set to `np.NaN`.
At the end of the episode, the average is computed only on the valid entries, making so that if the loss for that specific reward has been used to update the network, the average will be different from zero or, at least, won't be a constant number

Regarding the actual implementation of the Lexicographic DQN, due to how the code was structured, most of the function and classes have remained the same, except for the network's `update` and `act` function.

For the first one, after the loss respective to the first reward has been calculated, it's checked if its absolute value is below a certain threshold. If so, the loss is calculated in respect to the second reward; the same is done for the third reward.

For the latter function, instead of choosing the samples for every action with the argmax, all the Q values are calculated, but only the sample whose Q value is within a certain threshold is chosen.

After doing so for every action, and then for every reward, the result is a set 'preferred' samples for every reward. Then, for any action, if any action space sample is present in the 'preferred' sample of multiple rewards, it is chosen, meaning that it is *optimal* for maximizing more than one reward.
If there is no intersection between the rewards' 'preferred' samples, the first objective one's are used.

In other words, it's checked if the sample of every action generated by different rewards are the same, meaning that the sample would optimize for more than one objective.

0.5 Trainings Results

In this section, all the trainings done will be described and analyzed. All the trained agents have been thoroughly labeled, exploiting their folder name as a source of information, with the use of tags that identify the various model features and reward types.
In Figure 4 can be seen the folder structure used for the trainings: the main folder identifying the session is named after the environment (‘Env‘) and contains both model tags (‘[MT1]‘) and reward tags (‘[RT1]’).

For example, the folder ‘Ant [EP] [AWG] [EPS] [20S] [NOB] | [ORG] [CST]‘ contains the training of a Continuous DQN on the Ant environment, with a sample size of 20 (‘[20S]‘), which doesn't use bias in its layers (‘[NOB]’) and that is trained using the sum of the distance from origin and cost of control rewards (‘[ORG] [CST]‘).

Inside this folder, there are all the trainings of that specific model and rewards, identified by the timestamp of the start of the training, which contain, the model saved every iteration, a ‘.params‘ file, which is the dataclass with the network and agent's parameters rendered to text, and another folder containing the best model and the tf events file, a binary generated by TensorBoard's SummaryWriter.

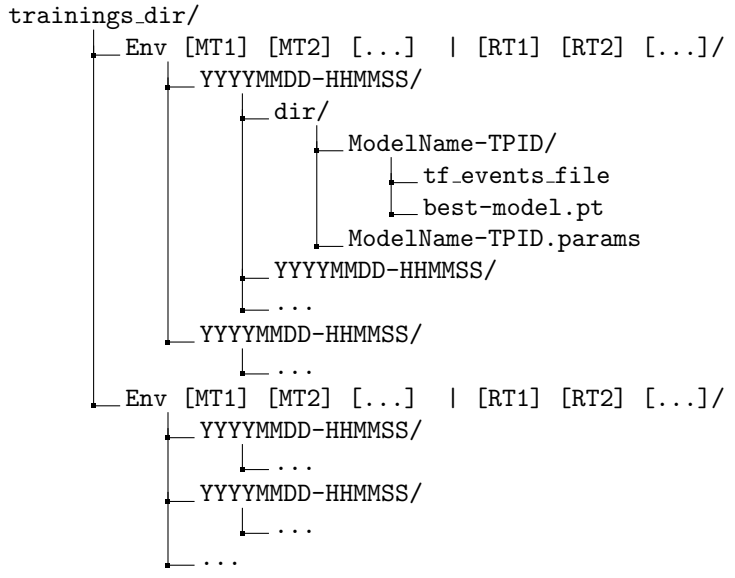

Figure 4: Folder structure used

With this standardized trainings structure, and with `tf_reader` built around that, the retrieval of the data is totally automated and user friendly.

Every agent, regardeless of its type, has been trained for 1000 epochs, which has proven more than enough for most of the networks, and tested for 200 epochs.

The first batch of trainings focuses on creating a baseline to refer to when judging the lexicographic agent's performances, but also on exploring the effects of possible rewards and model features on the performances.

Given the high amount of hyperparameters that can be varied, some of them have been fixed: the sample size will be 20 throughout all the sessions, which has proved to be an approximation good enough of the action space.
Additionally, parameters like the learning rate and the discount factor has not been changed from the standard values often used, respectively of 0.01 and 0.99.

0.5.1 Baseline trainings

The first variation introduced is for the reward given to the network; the following combinations have been used:

1. [**5FWD**] [**HLT**] [**CST**] - 5×Reward Forward + Healthy Reward + Control Cost
2. [**ORG**] [**HLT**] - Distance from the Origin + Healthy Reward
3. [**ORG**] [**HLT**] [**CST**] - Distance from the Origin + Healthy Reward + Control Cost

The reason of the FWD reward scaled by a factor of 5 is due to several preliminary trainings that showed that the Forward Reward, as it is, could not be enough for the network to generalize and learn to walk forward.
With this scaling, the hope is to have a reward big enough to solve the problem.

Every combination of rewards has been used on models with and without bias in their layers. Furthermore, for each of them, the batch size of the Experience Replay Buffer and the size of the layers has two variations: one with layer size of 256 and a batch size of 64, while the other with a layer size of 512 and a batch size of 128.

Figure 5 shows a comparison of all the networks using the first combination of rewards, with plots of network's reward as well allo other types of rewards obtained during test on average, and the average network reward during training.

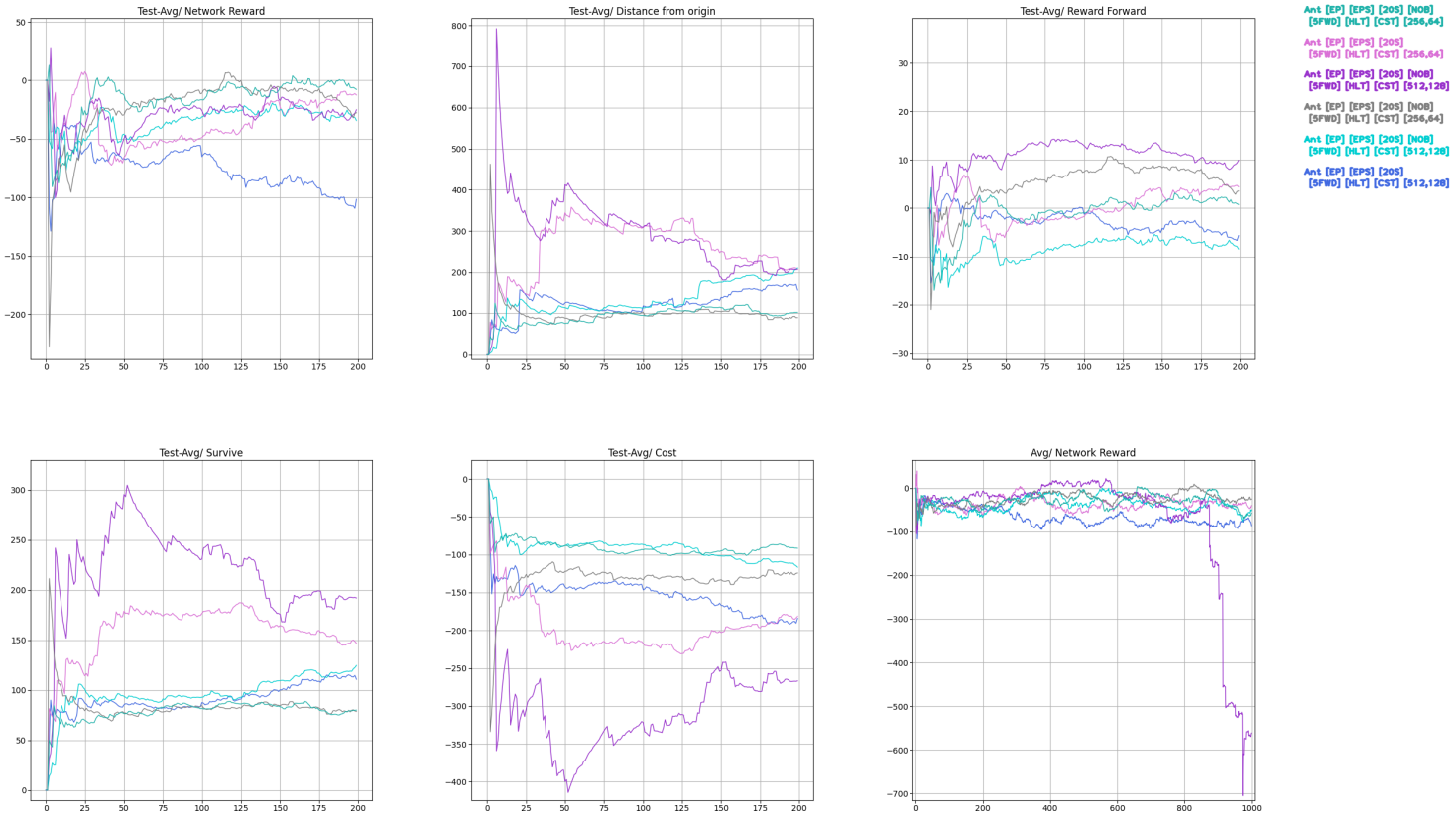

Figure 5: Comparison between networks trained with the combination of rewards [5FWD] [HLT] [CST]

On the right, all the plotted models' names are written, ordered by their performance during test.

As can be seen, starting from the average network reward during training, the network is not correctly generalizing, given the fact that all the curves, except for one, are oscillating around more or less the same value, showing that little to no progress is being made.

The third model, with no bias, a batch size of 128 and a network size of 512, stands out from the others: during the training, its network reward value drops below 700 in the last hundred of epochs.
This behaviour is due to the model wandering away from the origin in a direction different from the x axis (thus the small contribution of the FWD reward),

consequently obtaining a big ORG value (not counted in this reward combination) and a proportional cost of control.
Given the fact that only the healthy reward (fixed for every timestamp) is the consistent positive one, this explains the big difference and drop in network reward value.

On the other hand, looking at test results, this network has a pretty standard curve for the network reward. This is due to the fact that the best model, used for the tests, is saved based on an improvement of the average network reward, so the models during and after the drop in value are excluded.
While it would have obtained interesting results based on the distance from the origin, for the purpose of this set of rewards it's considered useless.

Despite the network reward values, this model has a higher than average healthy reward as well as a higher cost but an average distance from the origin. This indicates that the agent has wandered around without going in any specific direction.

Another thing worth noting are the points in the graph. These are the Inactive Terminations, or when the episode termination condition is triggered because the agent has stopped moving and stays still.
While the reason behind this behaviour is unknown, and only occurs in some epochs, it is worth considering to assess the reliability of the network.

Moving on, in Figure 6 can be seen the results of the trainings that uses the distance from the origin instead of the reward forward, along with the best network from the previous group.
While the network reward's curves can't be directly compared due to the difference in magnitude of the rewards, a performance evaluation can be done analyzing the other available information.

First of all, the average network reward during training's evolution shows that with this set of rewards, the network is correctly generalizing and learning, showing a positive trend.
Stating the obvious, these agents have obtained ORG values substantially higher than the best network from the previous reward set. Beside this, it can be seen that there is also an increase in the healthy reward, meaning that this agents survive longer on average.

Another consideration can be made by considering the last two agents listed, respectively the worst agent from this group and the best of the last. Despite having similar values for both ORG and HLT rewards, the cost of the first is higher than the second.

Figure 6: Comparison between networks trained with the combination of rewards [ORG] [HLT] [CST] and the best agent from [5FWD] [HLT] [CST]

This could mean that the agents trained with the distance from the origin are more cost-effective in respect to those that use the forward reward: both agents have been, on average, alive for the same time, but the on trained with ORG reward has managed to distance itself further from the origin while spending less.

A better yet result can be seen by plotting the results of the last group of rewards, identical to the former, but excluding the cost from the reward (Figure 7).

Even here, despite being similar in the composition, the network rewards can't be directly compared, mostly because the CST reward, being negative, has a big impact.

Figure 7: Comparison between networks trained with the combination of rewards [ORG] [HLT] and the best agent from [ORG] [HLT] [CST]

For the same reason, while at first sight it may seem that these networks have been learning better compared to the one with the control cost, this is not (entirely) the case: the average training curves naturally have a higher slope because of the absence of the negative impact of the cost.

Despite this, 3 out of 4 instances of this agent have outperformed on all fields the best agent from last group.

Furthermore, they outperform even the best [5FWD] [HLT] [CST] model for as concerns the Forward Reward, as can be seen in Figure 8.

With this overall improvement so far, the networks trained with ORG and HLT rewards result the best on every metric, except for cost, where they show an

Figure 8: Forward Reward obtained during test by all [ORG] [HLT] modes and the best [5FWD] [HLT] [CST] and [ORG] [HLT] [CST] models

evolution consistent with last group's training.

At this point, the presence or absence of bias in the layers does not have a clear influence on the network's performance, thus, this variation will not be in the lexicographic's agent training, which will focus on other parameters.

The same can be said for the network's size and batch size, but they will be kept regardless.

0.5.2 Lexicographic Agents' Trainings

As stated before, with this iteration of trainings, the bias will no longer be part of the hyperparameters.
On the other hand, while network and batch sizes, as well as the rewards, will vary, two new criterions are introduced: the loss threshold and slack.

For the latter one, this is the parameter that dictates how close to the max Q value should the other action samples be.
In other words, for a sample to be chosen, its Q value should be greater or equal to

$$max - slack \times |max| \tag{1}$$

While in the initial implementation the max Q value was multiplied directly for the slack, using its absolute value has been found to work better.

Throughout the training the slack parameter has been set to 0.01, which is a value found with multiple tests to assure that the agent has a fair number of action to choose from.

The parameter which will vary in the trainings is the loss threshold, responsible for choosing which rewards' set of Q value to optimize for.

Two iteration of trainings will be done, one with a threshold of 0.5, while the other with a threshold of 0.3, respectively corresponding to the model tags [LT-5] and [LT-3].
While the first value allows for a more dynamic change of objective, the second value is more conservative, optimizing for the others objectives only after the loss has stabilized on a lower value.

The rewards used, being the same, are arranged both to compare the agent with its non lexicographic counterpart, and to explore with the effect of their order. The reward tags associated indeed indicate the hierarchical order in which they're given to the network:

1. **[5FWD,HLT,CST]**
2. **[HLT,5FWD,CST]**
3. **[ORG,HLT,CST]**
4. **[HLT,ORG,CST]**
5. **[CST,ORG,HLT]**

From now on, being the agent that outperforms all others, the trainings performed with bias and using both the distance from the origin and the healthy reward, namely 'Ant [EP] [EPS] [20S] | [ORG] [HLT] [512,128]', will be used to

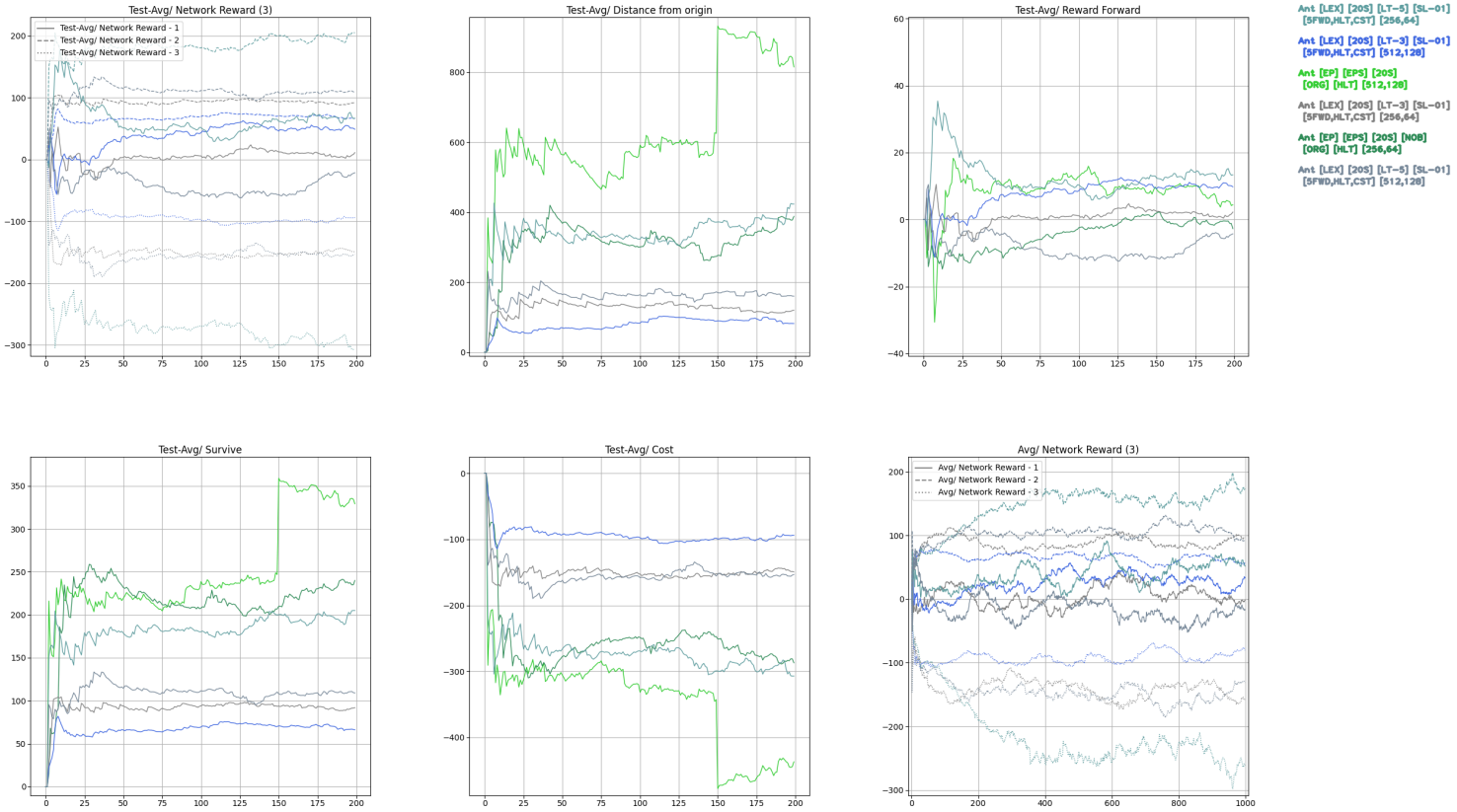

Figure 9: Rewards obtained from agents trained with [5FWD,HLT,CST] and the best standard DQN

compare the performance of the Lexicographic agents, without considering the distance from the origin.
This is because, given instantly high value, and by looking at its test values, not the average, it's clear that it's a 'lucky run': the high average value is due to a single episode during the test which totaled a distance from the origin of over 30.000.

Outside this single value, the run results consistent with other trainings. Instead, 'Ant [EP] [EPS] [20S] [NOB] | [ORG] [HLT] [256,64]', its unbiased and smaller variant, will be used for comparison on the ORG values.

In Figure 9 the results of the trainings can be seen.

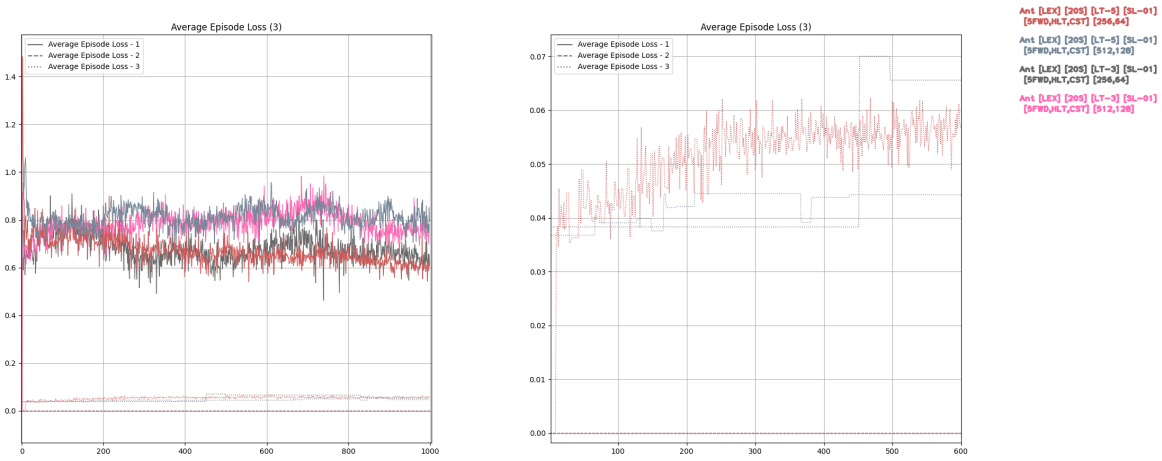

Figure 10: Average loss evolution during training

In terms of Forward Reward, the first priority for the networks, the first two have slightly better results than the standard agent, although, in the rest of the rewards the lexicographic agent has unremarkable performances.

An interesting insight can be gained by looking at the value taken by the average loss per episode.
The first plot in Figure 10 is an overview of the three losses during the training, while in the second plot is zoomed to see the two smaller curves.

As can be seen, the loss for the first objective fluctuates continuously, meaning that during the training it was being used to update the network.

The same can't be said for the second network reward: the dashed line can barely be seen. The explanation, obtainable only by monitoring the loss of the other trainings, is that the healthy reward mostly generates losses of low magnitude, being already under the loss threshold both of [LT-3] and [LT-5].
Confirmation of this can be found by looking at the dotted line representing the loss values for the third objective: although less, it's used for the update of the model.

Analyzing the lexicographic agent with the distance from the origin as first priority instead of the reward forward (Figure 11) shows similar performances with a slight improvement.

Figure 11: Rewards obtained from agents trained with [ORG,HLT,CST] and the best standard DQN

Although not being their first priority, most of the lexicographic agents have a better average reward forward than the standard agent.
On the other hand, only one lexicographic agent has managed to have ORG and HLT values higher than the standard networks' best, exception made by the average anomaly already spoken of.

The dominance of only one of the agents can be studied further by looking at the plot of the average network reward during the training.
Only this network and its [LT-3] same size counterpart do not reach a plateau in their training rewards. This could mean that a network size of 256 and a batch size of 64 are not enough to generalize this kind of rewards.

Without going into further detail through the remaining trainings, where the rewards' priority is changed, their performance has been overall unremarkable if compared to the standard agents rewards.

Despite this, it's worth talking of one agent in particular, another 'lucky run' which, unlike the one used for the benchmark, does not ends to a single episode with a high reward, but extends over the entire training.

In Figure 12 are plotted the training data relative to the lexicographic agent whose first objective is the healthy reward, followed by the forward reward and control cost.
It's the only network with different rewards' priorities from the others already seen that dominates every other agent trained, both in test and train rewards.

This could be seen during the training already. Being the training time also subject to the number of interactions that the agent performs with the environment, the more an agent is alive, the more its training time would be.

This particular agent, took about 70 hours to be trained, against the average 4 for the smaller agents and 10 for the bigger ones.

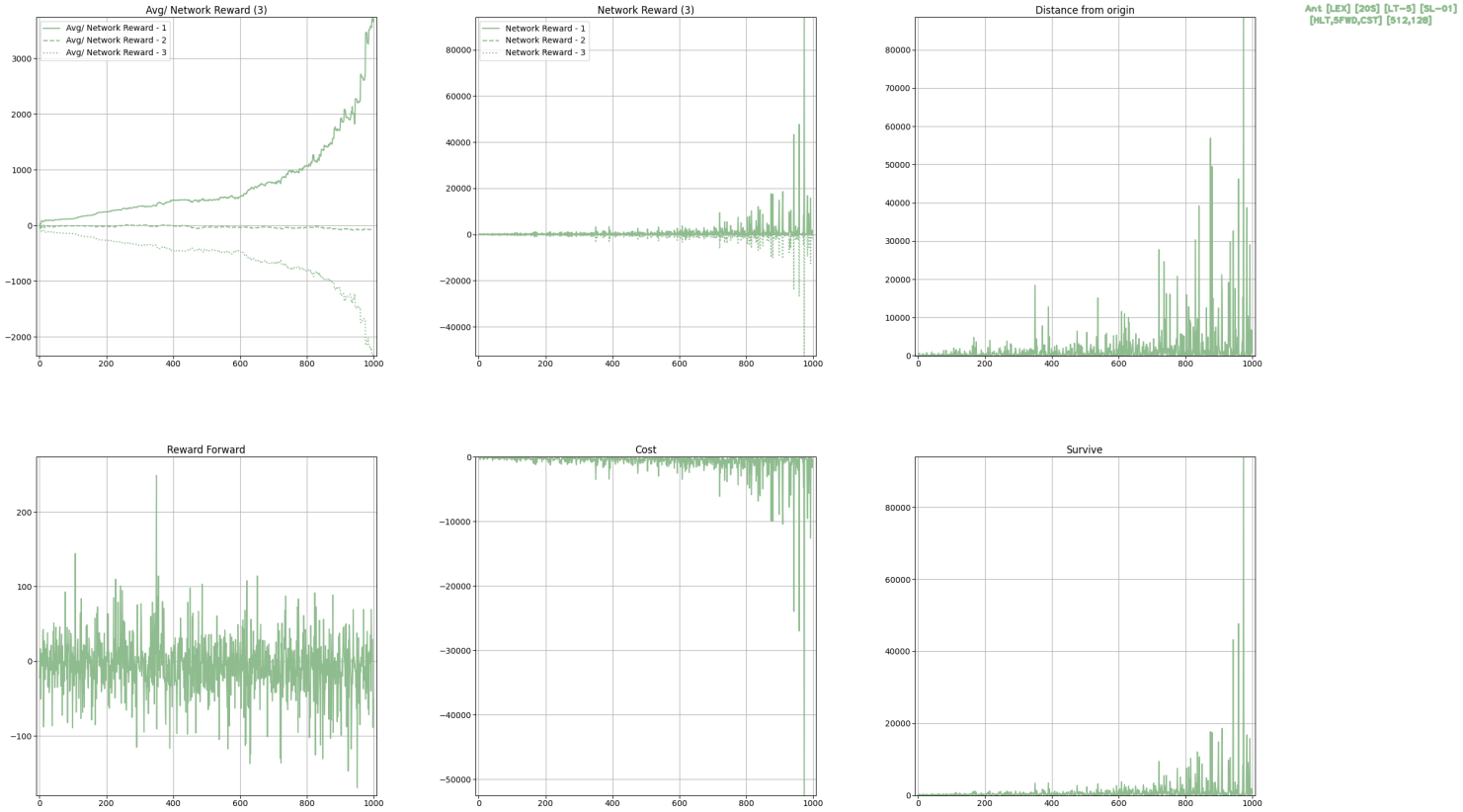

Figure 12: Training cumulative rewards and average network reward of ‘Ant [LEX] [20S] [LT-5] [SL-01] | [HLT,5FWD,CST] [512,128]‘

0.6 Conclusions

In conclusion, the extensive experimentation with reinforcement learning agents in various configurations has provided valuable insights into the complex dynamics of training in such environments.

These findings emphasize the critical role of reward combinations in shaping agent performance, although not as intuitively as it may seem.
It can be observed that the combination of rewards [ORG, HLT, CST] does not consistently outperformed [5FWD, HLT, CST] in both distance from the origin and forward reward.

The introduction of lexicographic agents brought both promise and challenges. While these agents performed on par with or slightly better than standard agents in certain scenarios, some trainings show that potentially one such agent could outperform all others.
Understanding how these priorities, as well as the loss threshold and network sizes influence training outcomes is crucial for harnessing the full potential of lexicographic agents.

Speaking of loss threshold, this revealed its significant impact on the learning process.
A higher loss threshold introduced more dynamism in optimizing different objectives, while a lower threshold promoted stability by delaying optimization of secondary objectives, although resulting more conservative and less performing.

Network size and batch size, while important, did not uniformly correlate with improved performance.
It has been found that larger networks and batch sizes did not guarantee better results, emphasizing the need for thoughtful selection of these hyperparameters based on the specific task and reward landscape.
The same consideration can be made for the presence or absence of bias, which did not exhibit a clear-cut influence on agent performances.

In summary, this study underscores the complexity of training reinforcement learning agents and highlights the importance of tailoring reward structures, priorities, and hyperparameters to the specific task at hand.
Effective experimentation and iterative refinement remain crucial for achieving optimal agent performance in dynamic and challenging environments.

0.6.1 Further Development

As stated before, correctly tuning the hyperprameters is considered crucial, mostly in lexicographic agents.

The goal would be to find the conditions, both on rewards selections and network parameters, that make anomalies such as the agent in Figure 12 comparable with the standard lexicographic agent.

In addition to this, adaptive learning rate schedules and exploration strategies could be used to lead to faster convergence and more stable training.
Adaptive methods that dynamically adjust learning rates or exploration rates during training may be worth investigating.

Another improvement could be obtained by solving the problem of the low speed of CPU trainings (if compared to GPU training of other networks).
While the considerations made in the description of the training environment still holds, making the CPU the fastest way to train, there is an alternative.

Using Isaac Gym, a framework specifically built for physics simulation envi-

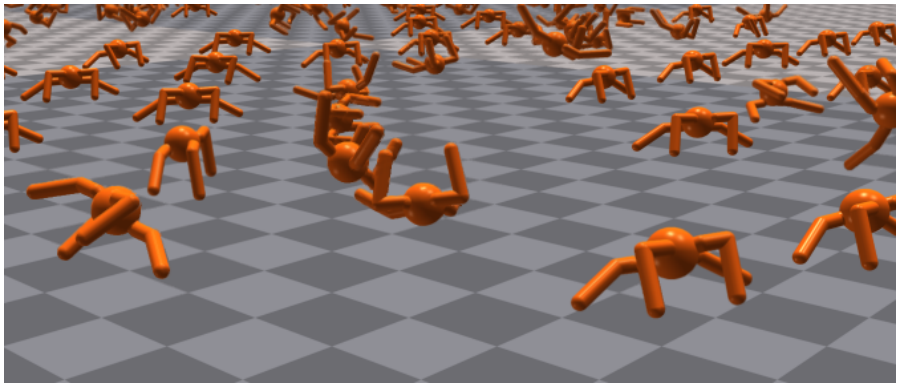

Figure 13: Isaac Gym simultaneous training

ronment and reinforcement learning research by NVIDIA, the CPU bottleneck can be overcome.
Using its API it is possible to hold both the physics calculations and the network on the GPU, enabling thousands of environments to run in parallel on a single workstation, and eliminating the downside of having to move data from GPU cache memory to the CPU.

Bibliography

[1] lrhammond. lmorl: Lexicographic Multi-Objective Reinforcement Learning, Year.

[2] mlisi1. tf_reader, 2023.

[3] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.