Reinforcement Learning Project
2023/2024

Lorenzo Di Luccio 1797569
Renato Giamba 1816155

«Learning model-based planning from scratch»

Sapienza Università di Roma

Introduction (1)

- **Model-based planning:**
  - proposal of a **sequence of actions**
  - evaluation of such actions with a **model of the environment**
  - final refinement to optimize expected rewards.
- **Main advantages of model-based vs model-free methods:**
  - **generalization** for never encountered states
  - better linking between present actions and future rewards
  - **resolution** of states with the same values or Q values.

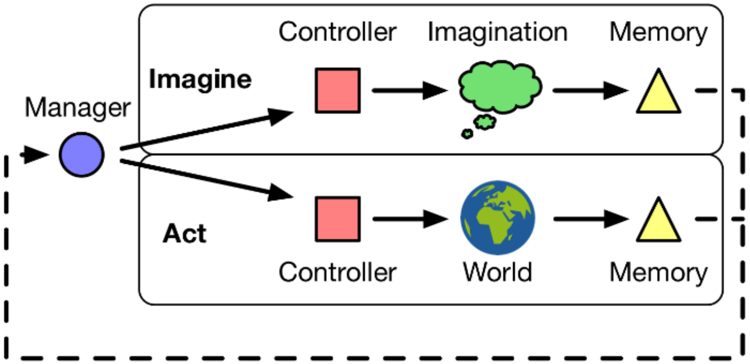
Introduction (2)

- **Planning is challenging**: a model can **evaluate** a plan and **execute** it, but it doesn’t know how to **construct** one.
- ***Imagination-based Planner (IBP)*****:** model-based agent which can perform the three stages of planning.
- **IBP is flexible:** it can deal with discrete or continuous environments.

SAPIENZA
UNIVERSITÀ DI ROMA

IBP (1) – Architecture

- The whole IBP can be seen as a **recurrent policy**, that acts on an environment (**world**) and is able to plan using *four* components:
  - **manager**
  - **controller**
  - **imaginator**
  - **memory**



IBP (2) – Manager

- The **manager** is a discrete policy $\pi_M: \mathcal{S} \times \mathcal{H} \rightarrow \mathcal{U}$ that maps a state $s \in \mathcal{S}$ and the history $h \in \mathcal{H}$ into a route $u \in \mathcal{U} = \{act, img1, img2\}$.
- It is trained with the **REINFORCE + baseline** algorithm.

```python
class Manager(torch.nn.Module):
    def __init__(
            self,
            state_dim: int,
            history_dim: int,
            hidden_dim: int,
            num_routes: int
    ) -> None:
        super(Manager, self).__init__()

        self.actor = torch.nn.Sequential(
            torch.nn.Linear(state_dim + history_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, num_routes),
            torch.nn.Softmax(dim=-1)
        )
        self.value_fn_predictor = torch.nn.Sequential(
            torch.nn.Linear(state_dim + history_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, 1)
        )
```

IBP (3) – Controller

- The **controller** is a policy $\pi_C: \mathcal{S} \times \mathcal{H} \rightarrow \mathcal{A}$ that maps a state $s \in \mathcal{S}$ and the history $h \in \mathcal{H}$ into an action $a \in \mathcal{A}$.
- It is trained with the **REINFORCE + baseline** algorithm.

```python
class Controller_DAction(torch.nn.Module):
    def __init__(
            self,
            state_dim: int,
            history_dim: int,
            num_actions: int,
            hidden_dim: int
    ) -> None:
        super(Controller_DAction, self).__init__()

        self.actor = torch.nn.Sequential(
            torch.nn.Linear(state_dim + history_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, num_actions)
        )
        self.value_fn_predictor = torch.nn.Sequential(
            torch.nn.Linear(state_dim + history_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, 1)
        )
```

IBP (4) – Imaginator

- The **imaginator** is a model of the environment $I: \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S} \times \mathcal{R}$ that maps a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}$ into a predicted next state $s' \in \mathcal{S}$ and a reward $r \in \mathcal{S}$.
- The next state part is trained with a **smooth mean absolute error loss** (continue) / **cross-entropy loss** (discrete).
- The reward part is trained with a **smooth mean absolute error loss**.

```python
class Imaginator_CState(torch.nn.Module):
    def __init__(
            self,
            state_dim: int,
            state_min: List[float],
            state_max: List[float],
            action_dim: int,
            hidden_dim: int
    ) -> None:
        super(Imaginator_CState, self).__init__()

        self.state_min = torch.tensor(state_min, dtype=torch.float32)
        self.state_max = torch.tensor(state_max, dtype=torch.float32)
        self.next_state_predictor = torch.nn.Sequential(
            torch.nn.Linear(state_dim + action_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, state_dim)
        )
        self.reward_predictor = torch.nn.Sequential(
            torch.nn.Linear(state_dim + action_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, 1)
        )
```

SAPIENZA
UNIVERSITÀ DI ROMA

IBP (5) – Memory

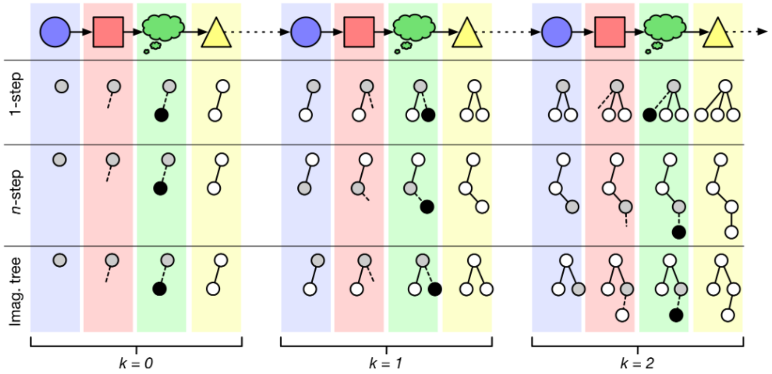
- The **memory** is a function $\mu: \mathcal{D} \times \mathcal{H} \rightarrow \mathcal{H}$ that maps the data $d \in \mathcal{D}$ from an iteration and an old history $h \in \mathcal{H}$ into a newly aggregated history $h' \in \mathcal{H}$.
- It is trained jointly with the controller using **backpropagation through time (BTT)**.

```
class Memory(torch.nn.Module):
    def __init__(
            self,
            route_dim: int,
            state_dim: int,
            action_dim: int,
            history_dim: int
    ) -> None:
        super(Memory, self).__init__()

        data_dim = route_dim + state_dim + state_dim + action_dim + state_dim + 1
        self.rec = torch.nn.LSTMCell(data_dim, history_dim)
        self.history_embeddings = torch.zeros((1, history_dim))
        self.cell_state = torch.zeros((1, history_dim))
```

IBP (6) – Imagination strategies

- IBP must have an **imagination strategy** to **construct** its plans.
- The three easiest imagination strategies are:
  - **1-step imagination**
  - **n-step imagination**
  - **imagination tree**
- **Imagination strategy used**: **simpler** version of the **imagination tree**.



IBP (7) – Training algorithm

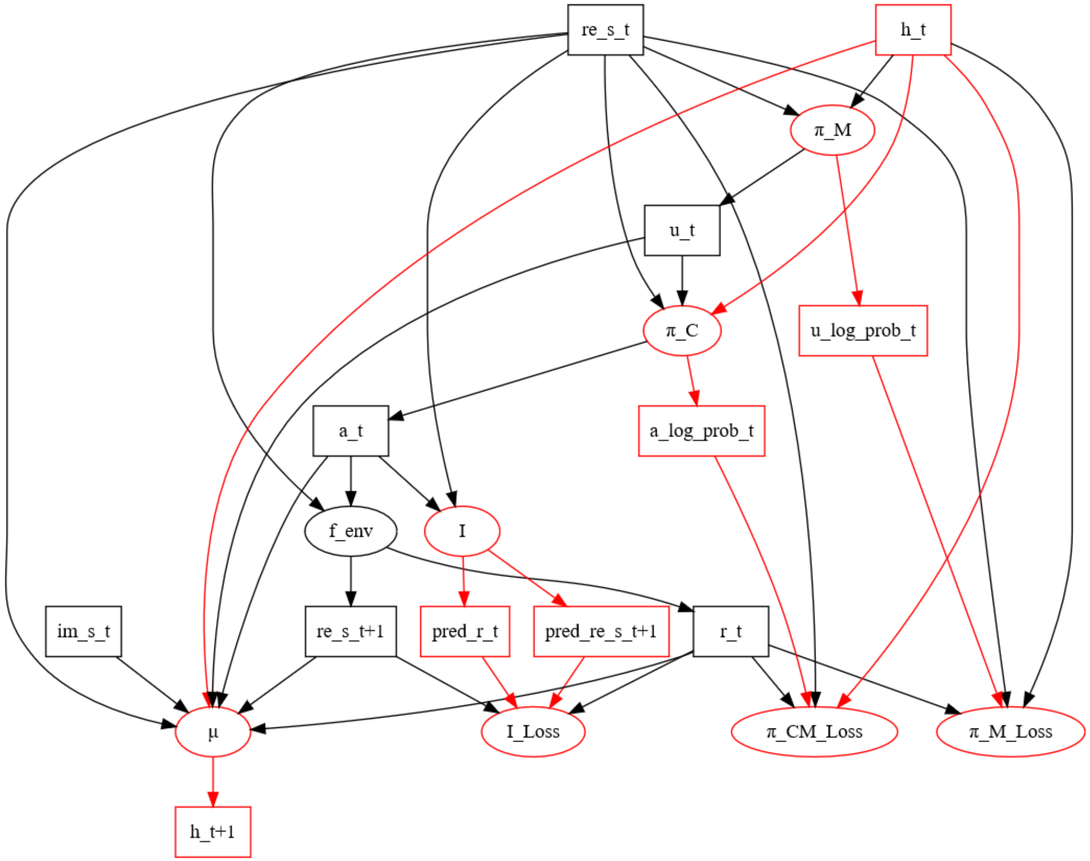
Algorithm 1: IBP agent with our simple imagination strategy. x is the current scene and x^* is the target. h is the current context

function $a^M(x, x^*)$
  $h \leftarrow ()$ ▷ Initial empty history
  $n_{real} \leftarrow 0$
  $n_{imagined} \leftarrow 0$
  $x_{real} \leftarrow x$
  $x_{imagined} \leftarrow x$
  while $n_{real} < n_{max-real-steps}$ **do** ▷ $n_{max-real-steps}$ is a hyper-parameter
    $r \leftarrow \pi^M(x_{real}, x^*, h_n)$
    if $r == 0$ or $n_{imagined} \geq n_{max-imagined-steps}$ **then** ▷ Execute action
      $c \leftarrow \pi^C(x_{real}, x^*, h_n)$
      $x_{real} \leftarrow World(x_{real}, c)$ ▷ Execute control
      $n_{real} += 1$ ▷ Increment number of executed actions
      $n_{imagined} = 0$
      $x_{imagined} \leftarrow x_{real}$ ▷ Reset imagined state
    else if $r == 1$ **then**
      $c \leftarrow \pi^C(x_{real}, x^*, h_n)$
      $x_{imagined} \leftarrow I(x_{real}, c)$ ▷ Imagine control from real state
      $n_{imagined} += 1$ ▷ Increment number of executed actions
    else if $r == 2$ **then**
      $c \leftarrow \pi^C(x_{imagined}, x^*, h_n)$
      $x_{imagined} \leftarrow I(x_{imagined}, c)$ ▷ Imagine control from previous imagined state
      $n_{imagined} += 1$ ▷ Increment number of executed actions
    end if
    $h \leftarrow \mu(h, c, r, x_{real}, x_{imagined}, n_{real}, n_{imagined})$ ▷ Update the history
  end while
end function

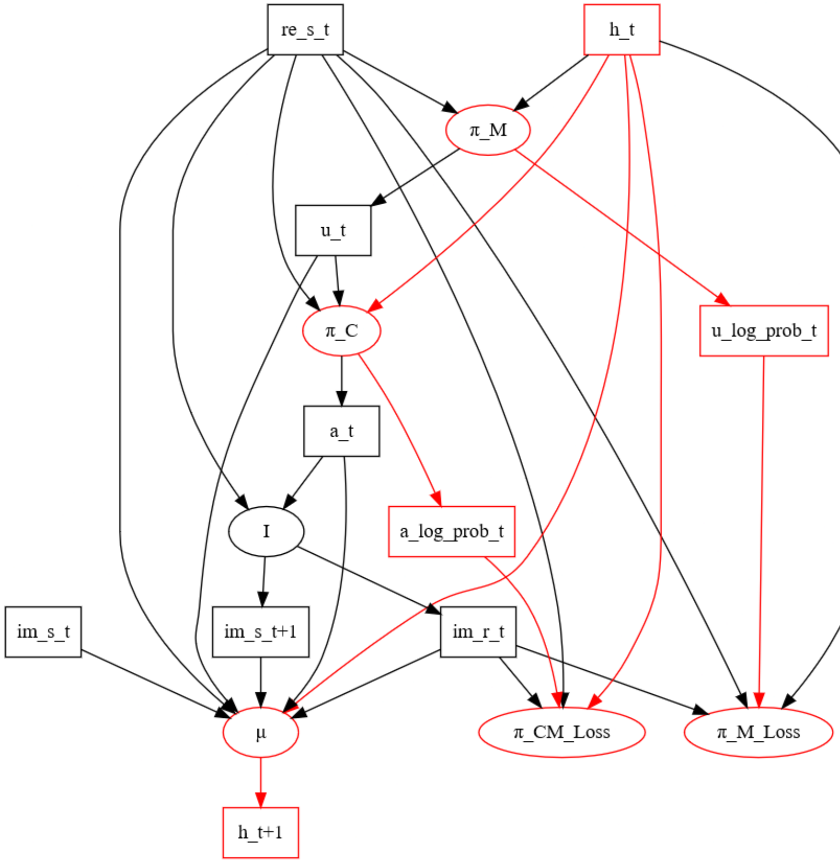
IBP (8) – Training algorithm: gradients

- The paper **doesn't clearly specify** how to build a **backward computation graph** for **gradients computation**.
- The one in the code is an implementation based on our understanding of how things **should be**.

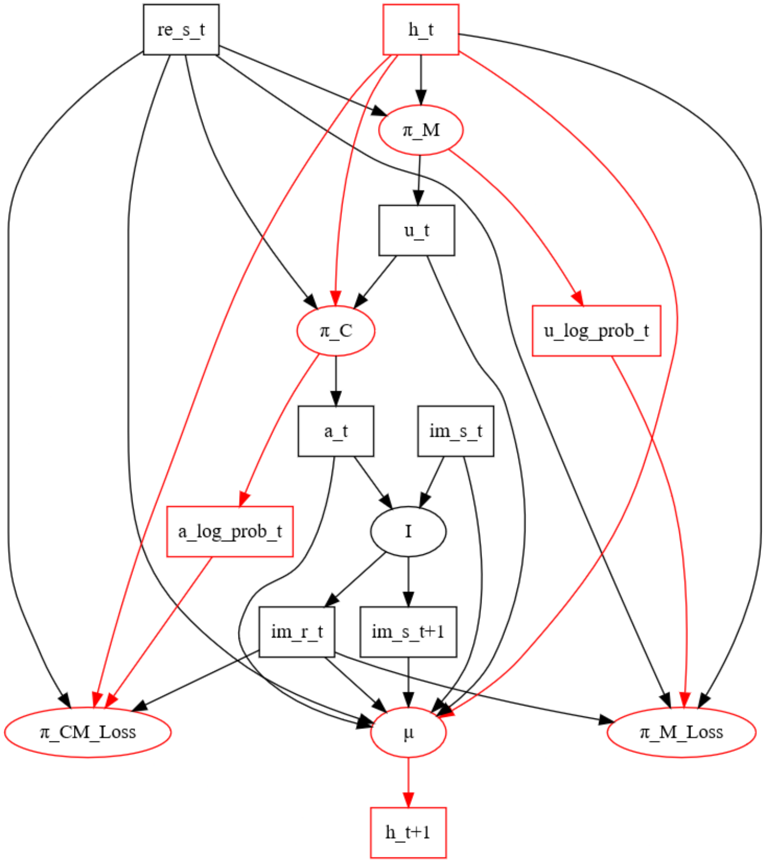
IBP (9) – Gradients for «Act»



IBP (10) – Gradients for «Imag1»



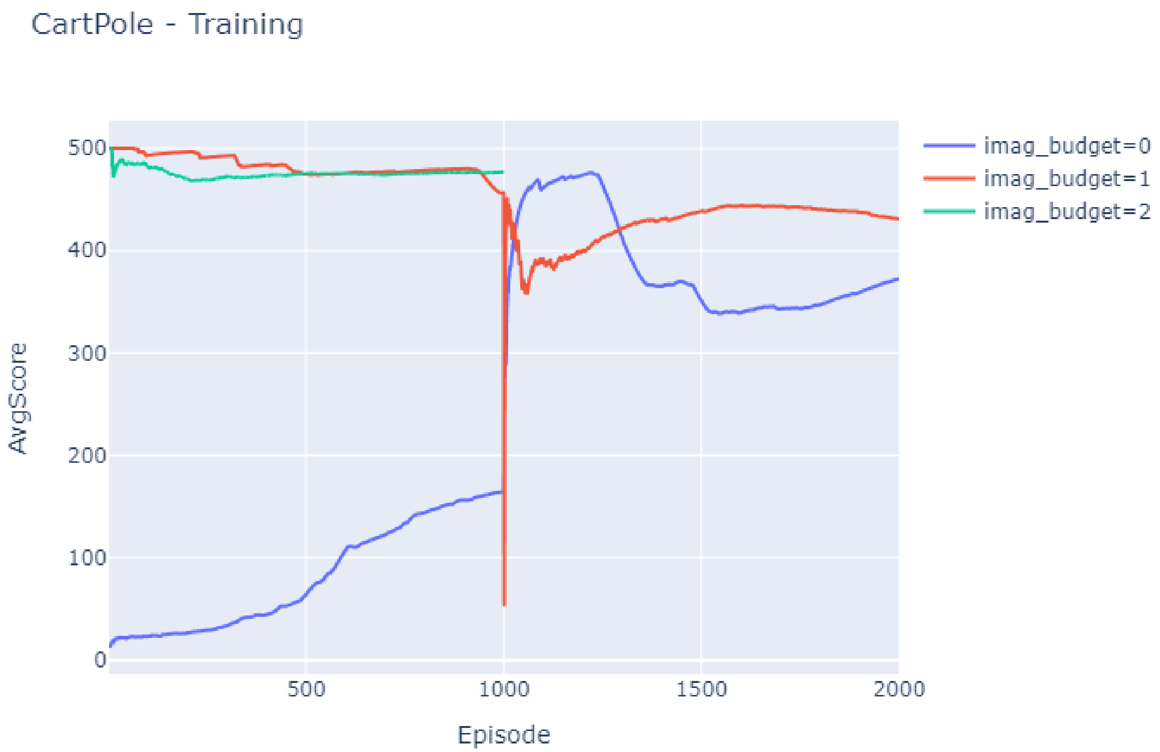
IBP (11) – Gradients for «Imag2»



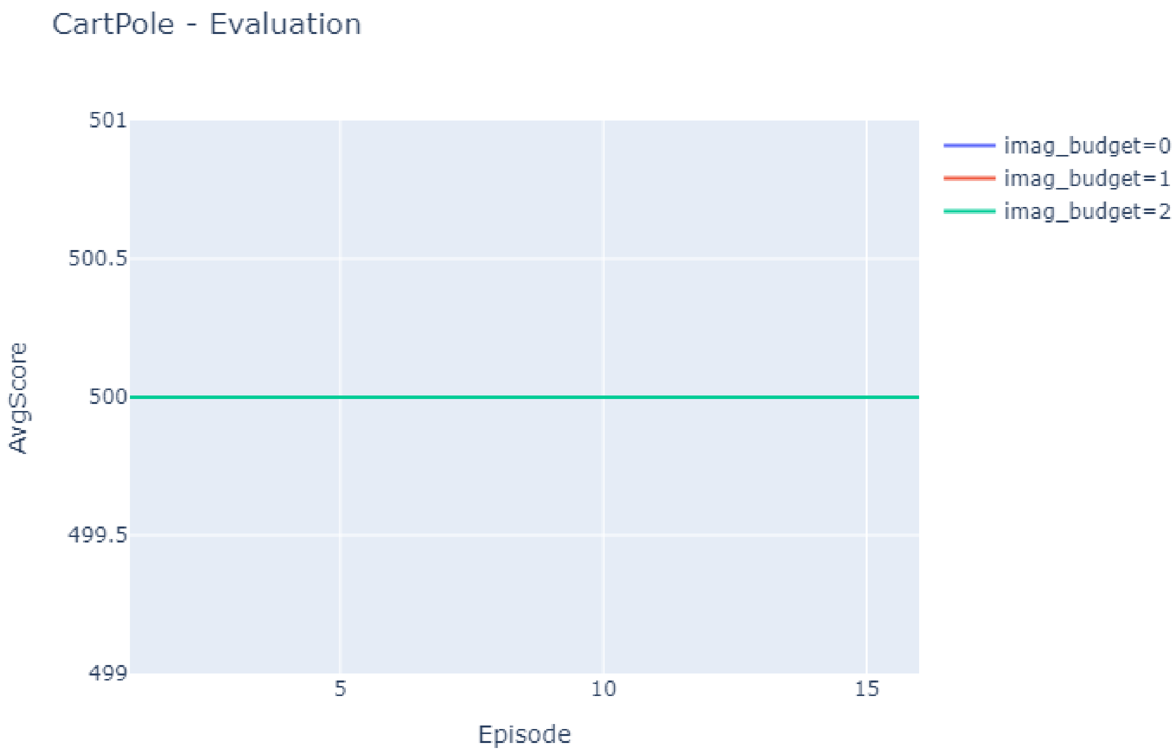
Experimental settings and hyperparameters

- **Optimizers:** 3 ADAM optimizer for manager, imaginator and controller + memory (trained jointly).
- **Learning rates:** 0.001 for each optimizer to start then decrease when performances don’t improve.
- **Gamma factor**: 0.99
- **Run:** 1,000 episodes for each run.
- **Episode:** variable number of steps. It stops if the environment terminates or truncates.
- **Imagination budgets:** 0, 1 or 2. The models with 1 or 2 are pretrained on the model with 0.

Experiment 1: CartPole (1) – Training

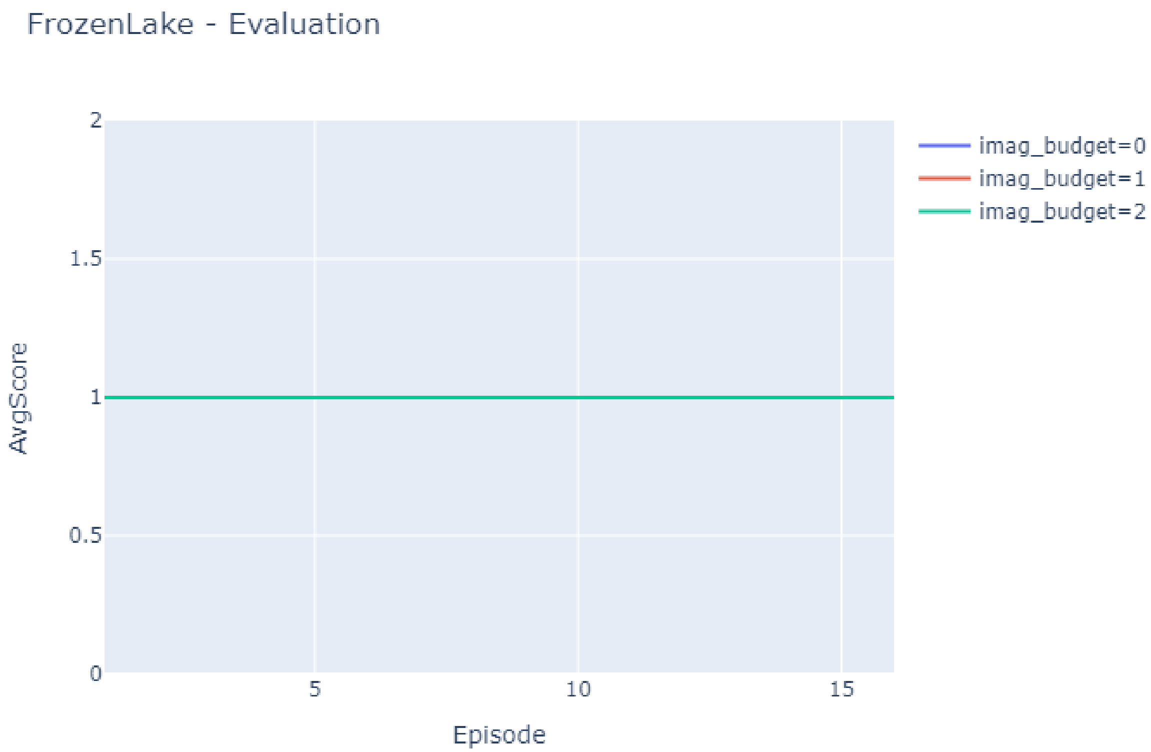


Experiment 1: CartPole (2) – Evaluation

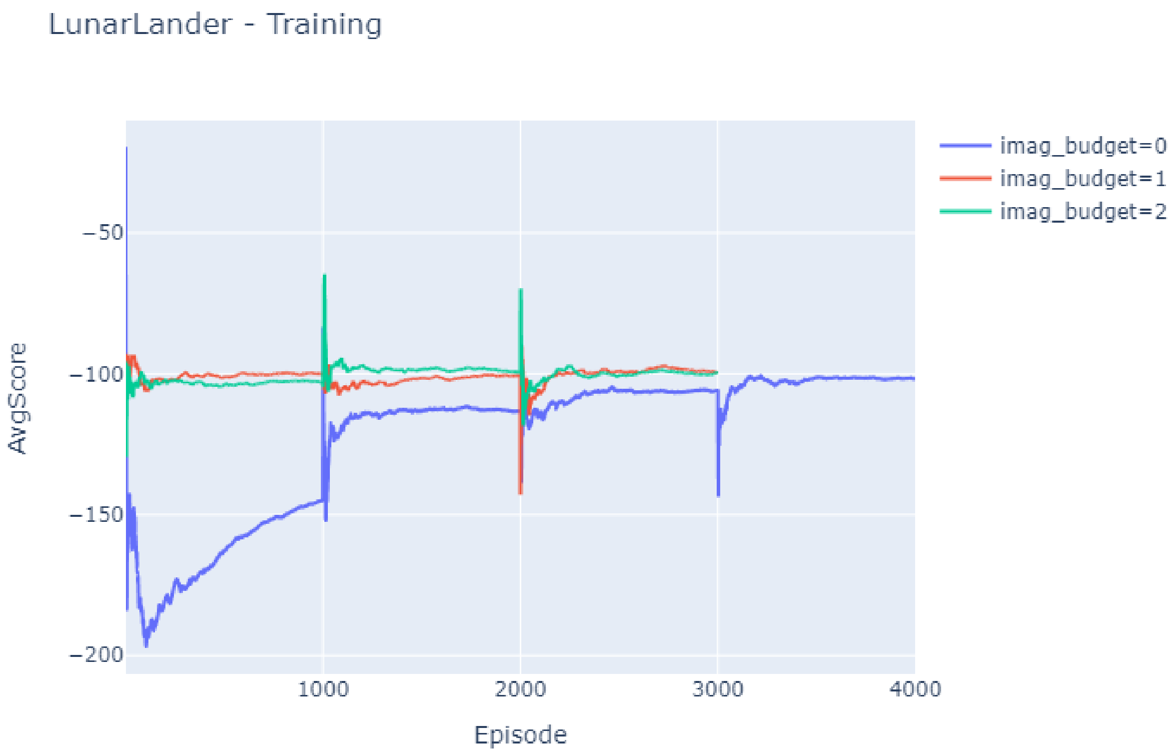


Experiment 2: FrozenLake (1) – Training

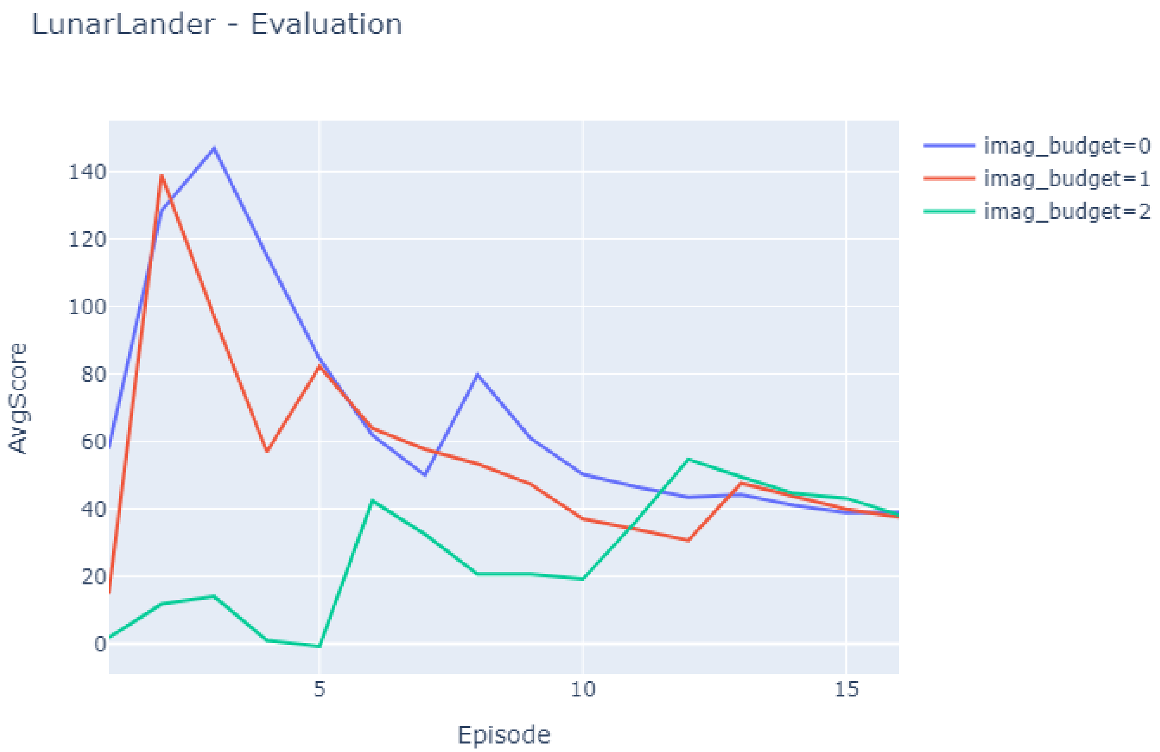
Experiment 2: FrozenLake (2) – Evaluation



Experiment 3: LunarLander (1) – Training



Experiment 3: LunarLander (2) – Evaluation



Final considerations (1)

- The paper sounds **vague** and **unclear** in some parts:
  - use of an unspecified **target state** $\boldsymbol{x}^*$ in the training algorithm
  - lack of **detailed instructions on gradients**
  - some **graphs** are **difficult to understand**
  - almost no **graphs of results** with metrics and statistics

Final considerations (2)

- The agent is generally **unstable** for various reasons:
  - use of **four different neural networks**
  - the REINFORCE algorithm generates **gradients with high variance**
  - training with other RL algorithms could be **harder** and **costly**

Final considerations (3)

- The learning algorithm seems **tuned** for something **very specific**:
  - **good (?) performances** on **environment created ad-hoc** (spaceship task)
  - **poor performances** on **simpler ones**
  - it may not **fail** on **generalization** if the components are implemented with **more powerful and complex neural networks** (**GNNs**)
  - **advanced neural networks** could **slow down** the agent

Thanks for your attention!

SAPIENZA
UNIVERSITÀ DI ROMA