Build your own reinforcement learning environment using JAX

IndabaX Tunisia

Agenda

1. Motivation
2. RL Framework: MDP
3. Build 2048 in Jumanji
    a. Problem formulation
    b. Implementing the environment

Machine Learning Paradigm

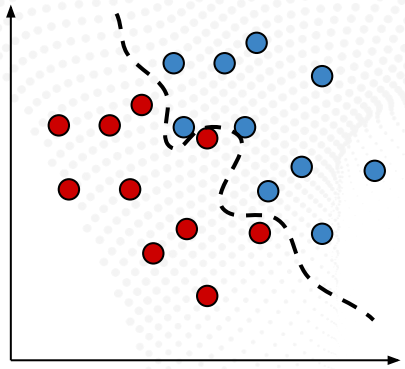
Supervised Learning

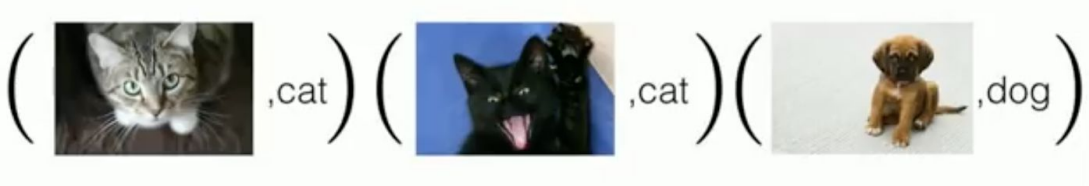
Data: (X,Y)

X is i.i.d sampled data,

Y is label

Goal: Learn a function to map X-> Y





- Makes machine learn explicitly
- Data with clearly defined output is given
- Direct feedback is given
- Predicts outcome.
- Resolves classification, regression, object detection, image captioning, etc.

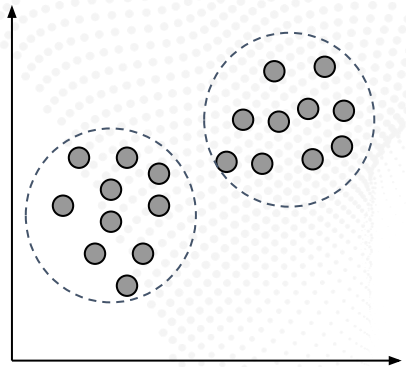
Machine Learning Paradigm

Unsupervised Learning

Data: X

Just data, no labels!

Goal: Learn some underlying hidden structure of the data



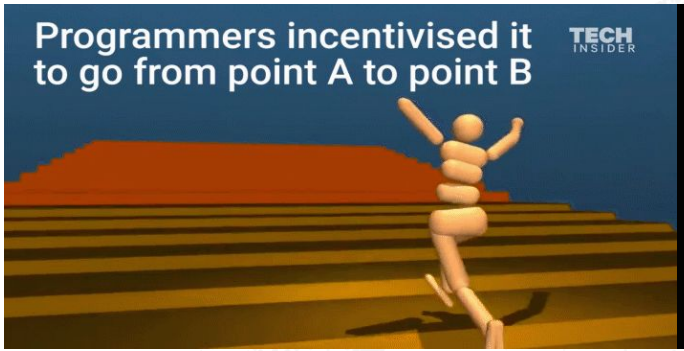
- Machine understands the data (identifies patterns/ structures)
- No supervision
- Evaluation is qualitative or indirect
- Does not predict.
- Resolves Clustering, dimensionality reduction, feature learning, etc.

Machine Learning Paradigm

Reinforcement Learning

A hallmark of reinforcement learning is that an agent interacts with its environment in a feedback loop of agent action and environment reaction as the agent learns the best actions over time.

Goal: Learn How to take actions in order to maximize reward.



- No supervision, only a reward signal
- Feedback can be delayed, not instantaneous
- Time matters
- Earlier decisions affect later interactions

The RL Framework

Goal: Win the game

State: The position of all the pieces on the board

Action: Where to put the next piece down

Reward: 1 if you win at the end of the game and 0 otherwise

Markov Decision Process (MDP)

Classical formulation of sequential decision-making systems, where actions influence not just the immediate reward but also subsequent states and through those future rewards.

A (finite) MDP consists in a tuple (S, A, P, R, ρ, γ) where:

- S is a (finite) set of states
- A is a (finite) set of possible actions
- P: S x A x S -> R is the transition probability distribution
- R: S x A -> R is the reward function with $r_t = R(s_t, a_t)$
- ρ: S -> R is the distribution of the initial state s_0
- γ: the discount factor

Markov Property: The probability distribution of the future states conditioned on the present and past values depends only upon the present state.

Markov Decision Process (MDP)

Markov Property: The probability distribution of the future states conditioned on the present and past values depends only upon the present state.

Markov Property for MDPs:

$$P(\text{Next State} \mid \text{Last State, Last Action, Two states ago, ...}) =$$
$$P(S_{t+1}|S_t, A_t, S_{t-1}, A_{t-1}, \dots, S_0) =$$
$$P(S_{t+1}|S_t, A_t)$$

Andrey "GOATee" Markov (1856-1922)

States or Observations?

Observations/States are the information our agent gets from the environment.

- **State:** is a complete description of the state of the world.
- **Observation:** is a partial description of the state. In a partially observed environment.

→The chess game is **a fully observable** environment.

The agent receive **a state** from the environment since it has access to the whole check board information.

Observation = State

States or Observations?

Observations/States are the information our agent gets from the environment.

- **State:** is a complete description of the state of the world.
- **Observation:** is a partial description of the state. In a partially observed environment.

→The card game is **a partial observable** environment.

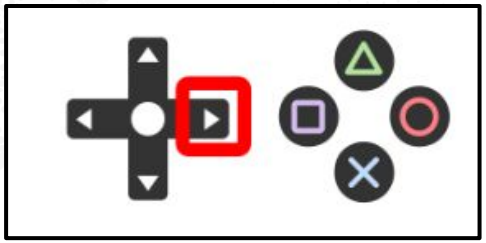
The agent can only view its own cards and does not have visibility of the cards held by the opponent, so it receive **an observation**.

Observation ⊂ State

Actions

The Action space is the set of all possible actions in an environment.

- **Discrete space:** the number of possible actions is finite.
- **Continuous space:** the number of possible actions is infinite.
- **Hybrid actions:** mix of both.



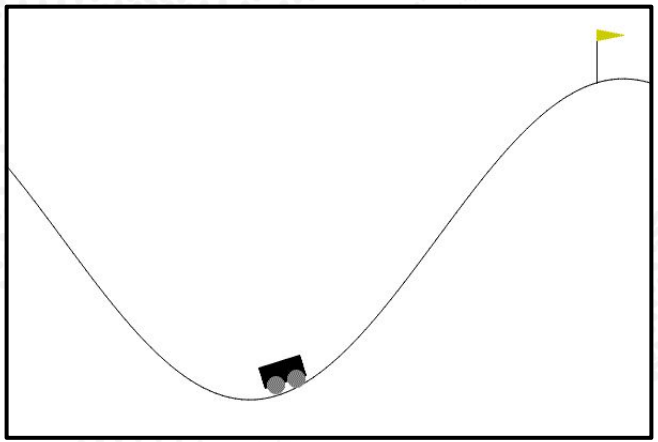
Discrete actions

The set of all valid actions in a given environment is called the **action space.**

Actions

The Action space is the set of all possible actions in an environment.

- **Discrete space:** the number of possible actions is finite.
- **Continuous space:** the number of possible actions is infinite.
- **Hybrid actions:** mix of both.



The action is in range [-1,1] representing the directional force applied on the car.

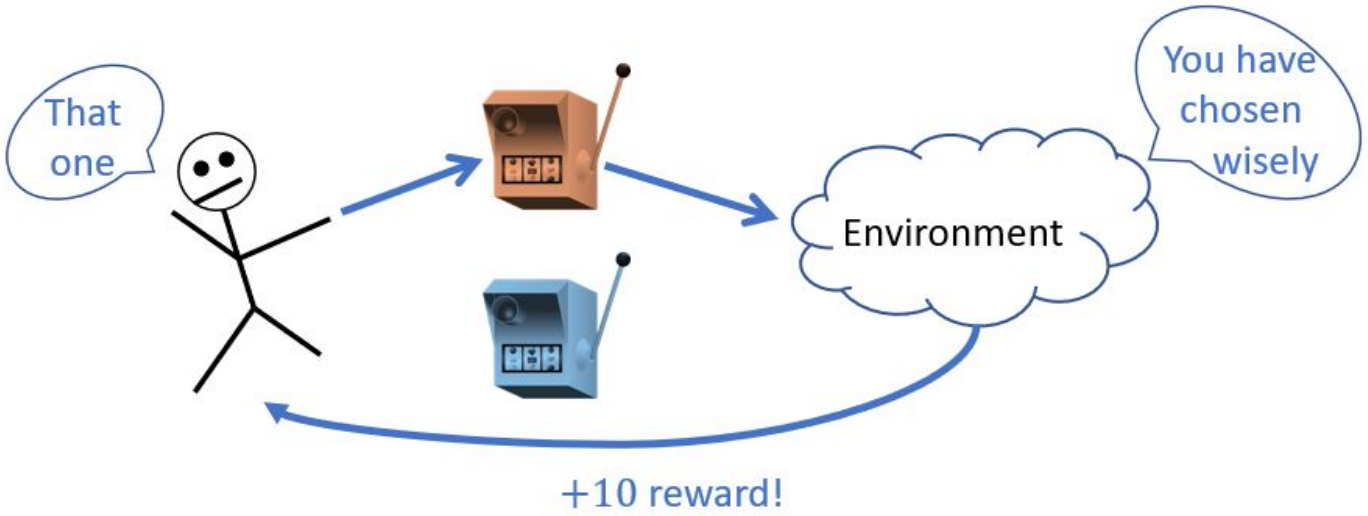
Continuous actions

The set of all valid actions in a given environment is called the **action space.**

Reward

The reward is a scalar value we obtain periodically from the environment: $r_t = R(s_t, a_t)$

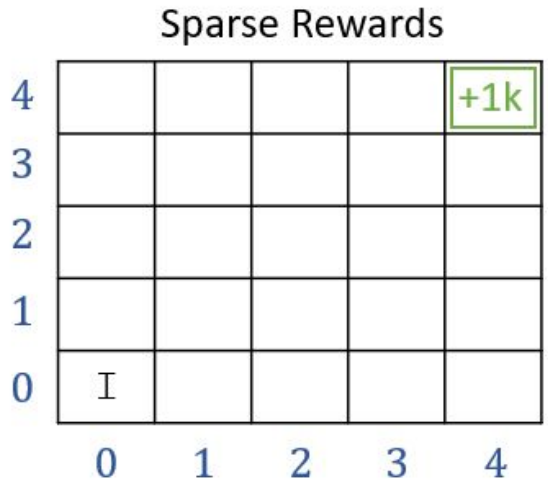
- **Frequency**: we don't define how frequently the agent receives the reward: it can be every second or once in a lifetime.

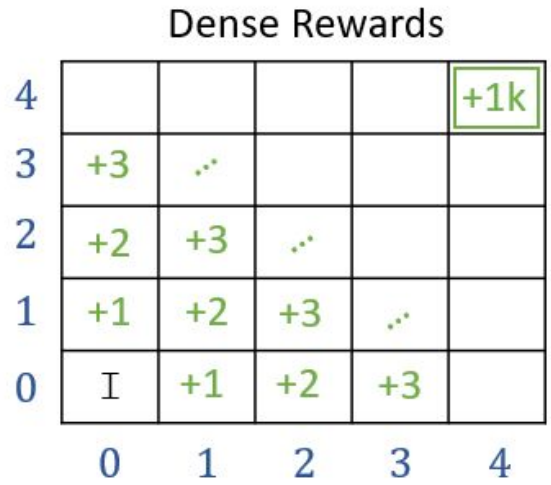


Reward

The reward is a scalar value we obtain periodically from the environment: $r_t = R(s_t, a_t)$

- **Dense reward:** are given to evaluate the agent in many different states.
- **Sparse reward:** are those given for only a small handful of states / events.
- **Shaped reward / Deceptive Reward…**





Trajectory and Return

The trajectory τ is a sequence of states and actions: $\tau = (s_0, a_0, s_1, a_1, .., s_T, a_T)$

- With the very first state, s0 , is randomly sampled from the start-state distribution (denoted usually by $\rho 0$)

⇒ The goal of the agent is to maximize the **expected cumulative reward** it receives in the long run. In the simplest case, **the return** after time step t is the discounted sum of future rewards:

$$R(\tau) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \dots$$

Return: cumulative reward

Gamma: discount rate

Trajectory (read Tau)
Sequence of states and actions

$$R(\tau) = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

Trajectory and Return

The discount factor γ is the process of devaluing future reward for the purpose of preferring more immediate, and is typically accomplished

- A reward received k time steps in the future is worth only if γ^{k-1} times what it would be worth if it were received immediately.

⇒The agent cares more about the value reward that is coming sooner.

Markov Decision Process (MDP)

Classical formulation of sequential decision-making systems, where actions influence not just the immediate reward but also subsequent states and through those future rewards.

A (finite) MDP consists in a tuple (S, A, P, R, ρ, γ) where:

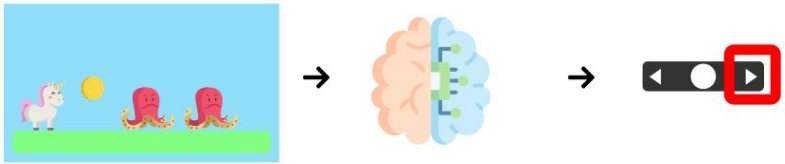
- S is a (finite) set of states
- A is a (finite) set of possible actions
- P: S x A x S -> R is the transition probability distribution
- R: S x A -> R is the reward function with $r_t = R(s_t, a_t)$
- ρ: S -> R is the distribution of the initial state s_0
- γ: the discount factor

Markov Property: The probability distribution of the future states conditioned on the present and past values depends only upon the present state.

The policy (π)

In general, the agent's policy is responsible for performing the mapping between state and action

- **Deterministic policy:** is a mapping π: S -> A.
- **Stochastic policy:** is a mapping π: S x A -> [0,1] $\pi(s|a) = P(A_t=a| S_t = s)$



Informally the agent's goal is to choose each action so as to maximize the discounted sum of future rewards, to choose each $\mathbf{a}_t$ to maximize $\mathbf{R}_{t+1} + \gamma\mathbf{R}_{t+2} + \gamma^2\mathbf{R}_{t+3} \cdots$

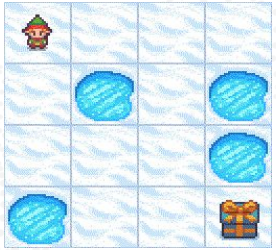
Through our journey in RL, we are **searching for a policy!**

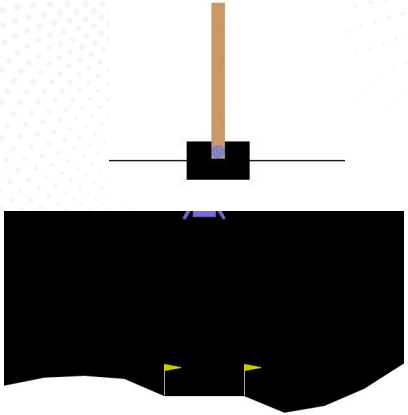
Problem Formulation

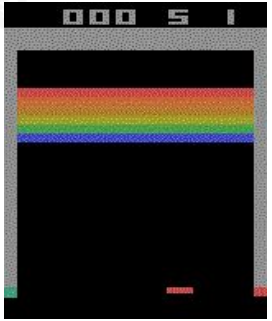
GYM or Gymnasium

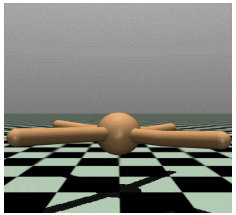
- Gym is a library that provides **implementations** of common **RL environments**, such as cartpole, pendulum, mountain-car, mujoco, atari, and more.
- Gym API provides 4 functions for each environment: **make, reset, step** and **render**

```
class ExampleEnv(gym.Env):
    def __init__(self, **kwargs):
        pass
    def reset(self):
        pass
    def step(self, action):
        pass
    def render(self, mode='human', close=False):
        pass
```



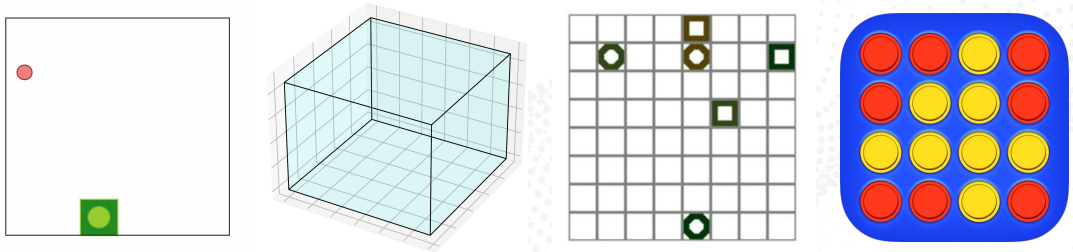


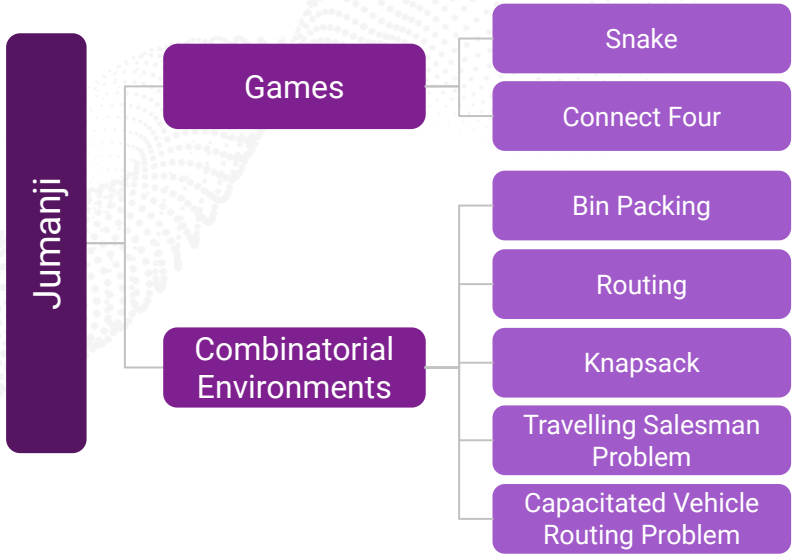




Jumanji - InstaDeep Suite of RL Environments written in JAX

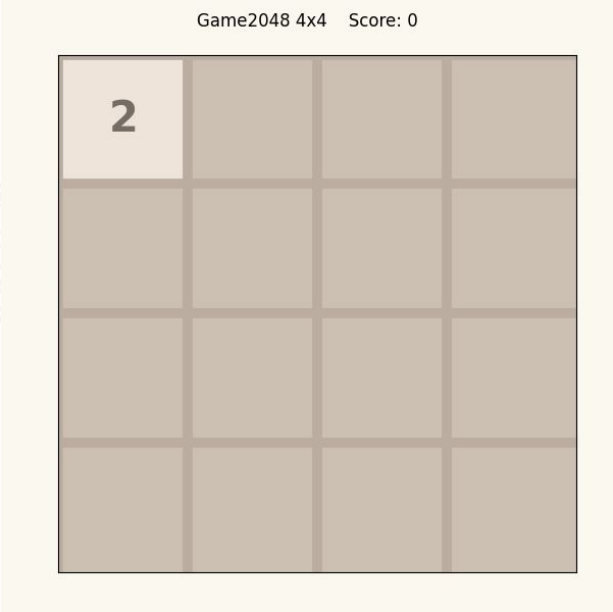
- Think of Jumanji as **InstaDeep's Gym** for JAX environments.
- Environments written in **JAX** to be able to **compile** entire environment loops (acting and learning jitted on a device).
- Support for different **types of environments**: single-agent, multi-agent, sequential and turn-by-turn.



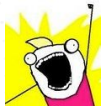


Jumanji: Industry-Driven Hardware-Accelerated RL Environments, *C. Bonnet et al.* https://github.com/instadeepai/jumanji.

Jumanji - 2048



- 2048 is a popular single-player puzzle game that involves sliding numbered tiles on a 4x4 grid.
- The game starts with one random tiles (either 2 or 4) placed on the grid.
- After each turn, a new tile is added to the grid in a randomly chosen empty cell, with a value of either 2 or 4.
- The objective of the game is to combine tiles with the same number to create larger tiles, with the goal of reaching the 2048 tile, and ultimate one of reaching above 2048 tile.



Let’s implement 2048 in Jumanji!

Formulate your problem as an MDP!

The RL Framework

Goal: Win the game

State: The position of all the pieces on the board

Action: Where to put the next piece down

Reward: 1 if you win at the end of the game and 0 otherwise

Termination conditions: This specifies the conditions under which an episode of interaction between the agent and the environment will end.

Environment Dynamic: includes the rules for how the environment changes in response to the agent's actions, as well as any uncertainties or stochasticity in the environment.

Extra information: Baselines / Env variation..

Problem Formulation (1/3)

State and Observation 👀

This section of the document describes the state and observation of the 2048 game.

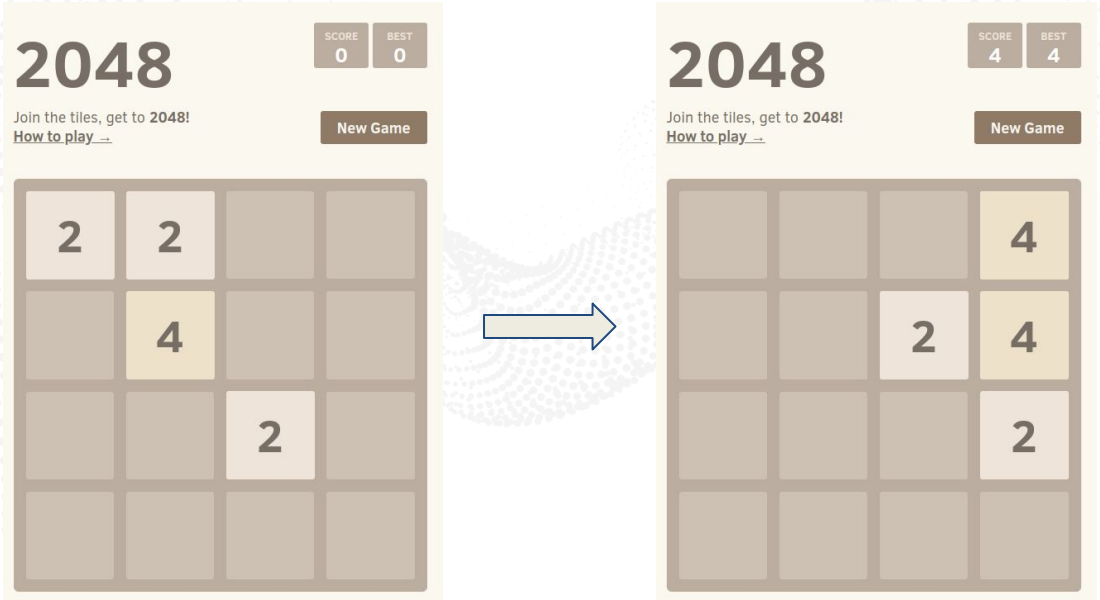
```
obs = [
  [0, 0, 0, 0],
  [0, 0, 0, 2],
  [0, 0, 0, 0],
  [0, 0, 0, 0],
]

class State(NamedTuple):
    board: Board
    step_count: Numeric
    action_mask: Array
    key: PRNGKey
    score: Numeric

class Observation(NamedTuple):
    board: Board
    action_mask: Array
```

Problem Formulation (2/3)

Reward 🏆

The reward function is a way to tell the agent how well it is doing.

Each time cells collapse, the agent receives a reward of $+S$ with S: the sum of the values of collapsed cells by that action for example:



Problem Formulation (3/3)

Action Space: In this environment, the agent has the following discrete action space:

$$A=\{ \text{up, down, right, left}\}$$

Termination Criteria: The episode terminates when no legal moves are possible, i.e., all squares are occupied, and no two adjacent tiles share the same value.

Environment Dynamic:

1. The first uncertainty comes from where the new tile will appear on the grid after the move. The new tile can appear on a random empty block on the grid. The probability of it appearing on any empty block is the same.
2. The second uncertainty comes from what number will appear on the new tile. The number on the tile is either 2 or 4. The probability for each number to appear is as follows: P(2 appears) = 0.90, P(4 appears) = 0.10.

2048 Implementation

Jumanji Structure

```
jumanji
├ testing
└ environments
  ├ logic
  ├ routing
  └ packing
    ├ ...
    └ binpack
      ├ conftest.py
      ├ env.py
      ├ env_test.py
      ├ env_viewer.py
      ├ instance_generator.py
      ├ instance_generator_test.py
      ├ reward.py
      ├ reward_test.py
      ├ space.py
      ├ space_test.py
      ├ specs.py
      ├ specs_test.py
      ├ types.py
      └ types_test.py
```

1. types.py: define the observation, state and any needed constant.

2. env.py: the main environment class.

3. utils.py

4. env_viewer.py: rendering the environment.

The implementation needs to be written in jax and consider the env must be just-in-time (jitted) compiled!

Jax 

JAX is a Python library offering **high performance** in machine learning with **XLA** and Just In Time (**JIT**) compilation.

1. Just-in-Time (**JIT**) compilation.
2. API is **similar** to **NumPy**
3. Automatic **differentiation** and **vectorization**.
4. **Express** and **compose** transformations of numerical programs.
5. Pseudo random number generation.
6. More options for **loops** and **control flow.**

Pseudo random number generation PRNG.

Unlike the stateful pseudorandom number generators (PRNGs) that users of NumPy and SciPy may be accustomed to, JAX random functions all require an explicit PRNG state to be passed as a first argument.

```
from jax import random

key = random.PRNGKey(0)

# key contains Array([0, 0], dtype=uint32)

a = random.uniform(key)

# Array(0.41845703, dtype=float32)

a = random.uniform(key)

# Array(0.41845703, dtype=float32)

key, subkey = random.split(key)

random.uniform(subkey)

# Array(0.10536897, dtype=float32)
```

Jax Tracing

What is Jax Tracing:

- JAX Tracing is a for accelerating numerical computations on CPUs, GPUs, and TPUs.
- However, JAX Tracing introduces several powerful features to optimize and parallelize computations, making it ideal for machine learning and scientific computing tasks.

Jax Tracing Features:

- *Just-in-Time (JIT) Compilation:* JAX Tracing compiles Python code on the fly for efficient execution on GPUs and TPUs, significantly boosting performance.

- *Accelerated Linear Algebra (XLA compiler):* In order to perform matrix operations as fast as possible, the code is compiled into a set of computation kernels that can be extensively optimized based on the nature of the code.

Jax Tracing

Jax Tracing Features:

- *Automatic Differentiation (Grad):* JAX Tracing provides automatic differentiation capabilities, it is a method for finding the derivative of a mathematical function without having to work it out by hand.
- *Vectorization (VMap):* allows you to apply a function to multiple inputs in parallel, improving efficiency and reducing code complexity.
- *Parallelization (PMap):* JAX Tracing provides parallelization capabilities through PMap, allowing you to distribute computations across multiple devices or processors, enabling even faster computation.

TODO ✅

types.py: contains State dataclass and Observation namedtuple.

env.py: contains YourEnv class with the implementation of step and reset.

env_viewer.py: implement the environment's render method

env_test.py: unit tests for the environment dynamics.

Thank you!