SomehowImLearning

```
   _____                     _                    _____ _           _
_
  / ____|                   | |                  |_   _( )         | |
(_)
 | (___   ___  _ __ ___   ___| |__   _____      __  | | |/ _ __ ___   | |
___  __ _ _ __ _ __  _ _ __   __ _
  \___ \ / _ \| '_ ` _ \ / _ \ '_ \ / _ \ \ /\ / /  | |  | '_ ` _ \  | |    /
_ \/ _` | '__| '_ \| | '_ \ / _` |
  ____) | (_) | | | | | |  __/ | | | (_) \ V  V /  _| |_ | | | | | | | |___|
__/ (_| | |  | | | | | | | | (_| |
 |_____/ \___/|_| |_| |_|\___|_| |_|\___/ \_/\_/  |_____| |_| |_| |_|
|______\___|\__,_|_|  |_| |_|_|_| |_|\__, |

__/ |

|___/

@AUTHORS : Pierre LAGUE & François MULLER
@ESTABLISHMENT : University of Lille, France
```

Welcome,

This repository aims to demonstrate the efficiency of known reinforcement learning algorithms on hard-coded environments.

It is an initiative of the authors following the class "Reinforcement Learning" led by Pr. Philippe Preux at the University of Lille.

The key points to remember from this repository :

- explicitely coded algorithms and environments
- easy use
- insights on the performance of the algorithms on the environments compared to state-of-the-art models
- we're just two chill guys coding stuff and somehow it's working

Explanation of the structure

> TODO : once we have all of our programs (algo, models, results) we use the `tree` command and describe each of the folders or files

Overview of Markovian Decision Problems and Reinforcement Learning Algorithms

Markovian Decision Problems (MDPs)

A **Markovian Decision Problem (MDP)** is a mathematical framework used to model decision-making in situations where outcomes are partly random and partly under the control of an agent. An MDP is defined by:

- **States (S):** A set of all possible states the environment can be in.
- **Actions (A):** A set of actions the agent can take.
- **Transition Function (P):** The probability $ P(s' | s, a) $ of transitioning from state $ s $ to $ s' $ after taking action $ a $.
- **Reward Function (R):** A function $ R(s, a, s') $ that provides the reward for transitioning from $ s $ to $ s' $ via $ a $.
- **Discount Factor (γ):** A value in $[0, 1]$ that prioritizes immediate rewards over future rewards.

The goal in an MDP is to find a **policy** $ \pi(a|s) $—a mapping from states to actions—that maximizes the expected cumulative reward, often referred to as the **return**.

Algorithms for Solving MDPs

1. Discretized Q-Learning

Discretized Q-Learning divides the continuous state-action space into discrete bins, allowing the agent to use traditional tabular methods.

- **Key Idea:** Approximate the environment using discrete representations of the states and actions.

This method is failry easy to implement when the discretization is already determined. However, a more challenging implementation would the dynamic discretization where given the reward of a discretized space, we would either discretize it even more or generalize it. (e.g. if a bin outputs a lot of reward, we decide to divide this bin by 2 in order to ger a more granular estimation of the Q function in this bin. Otherwise we generalize it with the neighbouring bins.)

2. Tabular Q-Learning

Tabular Q-Learning is a model-free reinforcement learning algorithm that updates a table of Q-values for state-action pairs:

- **Q-Update Rule:**
  $
  Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma \max_a Q(s', a) - Q(s, a) \right)
  $
  where $ \alpha $ is the learning rate.

SImilarly, this method is failry simple to implement and can output some interesting results. The issue is that for continuous state spaces, it won't work. The Q-table would need to have as many dimensions as the number of dimensions of the state space which is not possible (and not logical).

3. Neural Fitted Q-Iteration (NFQ)

NFQ extends Q-learning to continuous state-action spaces using neural networks:

- **Key Idea:** Use a neural network to approximate the Q-function $Q(s, a)$.
- **Training:** Batch updates with tuples (s, a, r, s') collected over multiple episodes.

This offline method allows us to get very interesting results on simple environments with only a few training iterations. It is limited by the complexity of the enviornments and its stochasticity. It can be pretty computationally intensive too, and the choice of hyperparameters is very important for a successful exeperiment.

4. Deep Q-Network (DQN)

DQN further improves upon NFQ by employing deep neural networks and advanced techniques like:

- **Experience Replay:** Store and reuse past transitions to break correlations in data.
- **Target Networks:** Stabilize training by using a fixed target Q-network for updates.
- **Key Equation:**
  $$Q_{\theta}(s, a) \leftarrow r + \gamma \max_a Q_{\theta'}(s', a)$$

This online method is the most common one in T.A.S (tool assisted speedrun) for example. It works with two networks, a main Q-network and a target Q-network that provides the "label" for the estimation of the Q function, deriving the value directly from the Bellman equation. It is not as fast as the NFQ algorithm but scales very well to large state spaces and complex environments. Similarly though, it will need a very good hyperparameter setup.

5. Direct Policy Optimization

Direct Policy Optimization methods optimize a parameterized policy $\pi_\theta(a|s)$ directly:

- **Objective Function:** Maximize the expected cumulative reward:
  $$J(\theta) = \mathbb{E}\left[\sum_t \gamma^t R(s_t, a_t) \right]$$
- **Approaches:** Include Policy Gradient Methods (e.g., REINFORCE, PPO, TRPO).
- **Advantages:** Effective in high-dimensional or continuous action spaces.
- **Challenges:** High variance in gradient estimates and reliance on stable optimization techniques.

This progression reflects the increasing complexity of both environments and algorithms, culminating in methods suitable for modern, high-dimensional problems.

How to use the repo

The root

At the root of the repository, you will find a python script `simulation_environment.py`. This script simulates an environment with a specific model, for a certain amount of time. (e.g. `python simulation_environment.py <env_name> <model_name> <simulation_time>`).

- For the car simulation :
  - it ends once the car has gotten a positive result
  - you can monitor its speed, reward, position and movements on the hill
- For the pendulum simulation :
  - it lasts a certain time (cli argument)
  - you can monitor rewards, angular position, angular velocity, actions

The `algorithms/environments` folder

In this folder you will find an implementation of each of our environments. The structure of the environment is purely indicative, but change it and the programs wont work so good after that. We tried to make it as general as possible (define your envs features and at least a `reset` and a `step` function).

> We highly encourage you to try and develop new environments (and visualisations if you want) and then open an issue to inform us.

Results for now

NFQ (offline)

Car Environment

- *Setup 1* :
  - nb_episodes : 10
  - episodes_per_iter : 200
  - evaluation_episodes : 10
  - gamma : 0.99
  - lr : 0.001
  - hidden_layers : (5, 5)
  - **Training Time : 544.59 seconds (~9min)**
  - **Iterations to first success : 17**
  - model : `nfq_car_model_10_ep.pkl`

Pendulum Environment

- *Setup 1* :

  - nb_episodes : 150
  - episodes_per_iter : 5000
  - evaluation_episodes : 10
  - gamma : 0.99
  - lr : 0.001
  - hidden_layers : (5, 5)
  - **Training Time : 140.84 seconds (~2min30)**
  - model : `nfq_pendulum_model_150_ep.pkl`

DQN (online)

Car Environment

- *Setup 1* :
  - nb_episodes : 500
  - iterations_per_ep : 200
  - target_network_update : 5
  - gamma : 0.99
  - lr : 0.001
  - hidden_layers : (5, 5)
  - epsilon_decay : 0.9995
  - **Training Time : 402.24 seconds (~7min)**
  - **Iterations to first success : 17**
  - model : `dqn_car_500_ep.pth`

Pendulum Environment

- *Setup 1* :
  - nb_episodes : 50000
  - iterations_per_ep : 5000
  - target_network_update : 5
  - gamma : 0.99
  - lr : 0.001
  - hidden_layers : (5, 5)
  - epsilon_decay : 0.9995
  - **Training Time : 305.56 seconds (~5min)**
  - model : `dqn_pendulum_50000_ep.pth`
