UNIVERSITÀ DI PISA

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

TESI TRIENNALE

Reinforcement Learning in Julia, using a multi-objective environment

Relatori:
Prof. Marco Cococcioni
Prof.ssa Beatrice Lazzerini

Presentata da:
Federico Lusiani

Anno Accademico 2019/2020

Contents

Abstract

In this work I apply Reinforcement Learning algorithms in Julia to train an agent in a multi-objective environment. The algorithms used are three implementations of the Deep Q-Learning algorithm (abbreviated as *DQN*, where *N* stands for *Neural*), from the *ReinforcementLearning* julia library. The environment is the *Lunar Lander* environment from OpenAI *gym* python library. The algorithms are used both on the standard version of the environment, and on a modified one. The modified version has a different reward function, to approximate a multi-objective environment. For each of the three DQN algorithms, I illustrate a training run on both the standard and the modified versions of the environment. From these runs, I verified that the third DQN implementation (Prioritized DQN) was superior to the other two implementations. I also verified that all three algorithm find the multi-objective environment much more difficult to solve, failing to converge after 2000 episodes, but finding nonetheless a good solution.

Chapter 1

Introduction

Reinforcement learning (RL) is an area of machine learning concerned with how software agents ought to take actions in an environment in order to maximize the notion of cumulative reward. Reinforcement learning is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning. Reinforcement learning differs from supervised learning in not needing labelled input/output pairs be presented, and in not needing sub-optimal actions to be explicitly corrected. Instead the focus is on finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge). [2]

As stated before, the objective of RL is to find an (approximately) optimal *policy* for a software agent. We define a *policy* as a mapping between *states* and *actions*. The *policy* is optimal if it maximises the *cumulative reward* experienced by the agent as it moves through the states of the environment. A commonly used RL technique is *Q-Learning*. The objective of the training is to approximate a *Q-function*, which maps (state, action) tuples to a G value (the *expected cumulative reward*). The Q-function is then exploited by the agent policy to choose the best action to perform, given the current state. In *Deep-Q Learning*, the Q-function is approximated by training a Neural Network model with gradient methods. For this reason, this technique is often abbreviated as DQN. In DQN, the training of the neural network model tends to be characterized by noisy gradients, making the convergence to a solution difficult. This is because the gradients computed are tied to the reward experienced by the agent during the training. Many of the features introduced by DQN algorithms, such as the *experience replay buffer* or the *target network* aim to reduce this effect. Still, the problematic must be taken into account when when designing the reward function of the environment, as ”noisy” reward functions can make the convergence slower or outright impossible. Such reward functions can happen when the reward tries to capture different objectives using very different weights values. Using weights that differ from each other by one or more orders of magnitude may be necessary in order to make sure that the agent prioritizes some objectives over others (*multi-objective optimization*).

In this work, I implement one such multi-objective environment. I modify the Lunar Lander environment, which has only one main objective (landing without crashing), by adding a correlated but different one (landing with the left leg first). In the reward function, the weight for this secondary objective is one order of magnitude smaller than the one used for the main objective.

Chapter 2

Reinforcement Learning Problems

Basic reinforcement is modeled as a *Markov decision process (MDP)*:

- a set of environment and agent states, S;
- a set of actions, A, of the agent;
- $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability of transition (at time t) from state s to state s' under action a.
- $R_a(s, s')$ is the immediate reward after transition from s to s' with action a.

A reinforcement learning agent interacts with its environment in discrete time steps. At each time t, the agent receives the current state s_t and reward r_t. It then chooses an action a_t from the set of available actions, which is subsequently sent to the environment. The environment moves to a new state s_{t+1} and the reward r_{t+1} associated with the *transition* (s_t, a_t, s_{t+1}) is determined. The goal of a reinforcement learning agent is to learn a *policy*: $\pi : A \times S \to [0, 1]$, $\pi(a, s) = \Pr(a_t = a \mid s_t = s)$ which maximizes the expected cumulative reward. When the environment is deterministic, it can be modeled using a transition function $P(s_t, a_t) = s_{t+1}$ and a reward function $R(s_t, a_t) = r_{t+1}$. When the agent policy is deterministic, it can be modeled using as a policy function $\pi(s_t) = a_t$.

From now on, we will assume that both the environment and the agent are deterministic.

2.0.1 Reinforcement Learning: Q-Learning

A class of solvers for Reinforcement Learning problems is *Q-Learning*. The objective of a Q-Learning solver is to find the *action-value function* $Q : A \times S \to \mathbb{R}$, which maps $(action, state)$ pairs to a value $Q(a, s)$ representing a prediction of the *total future reward* that will be experienced by an agent performing the action a in the state s. More formally, we define the G_t value (the *total discounted future reward*) as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1}$$

where γ is the *discount factor*. We can then define $Q(a_t, s_t)$ as

$$Q_t = Q(a_t, s_t) = G_t$$

This means that the $Q(a_t, s_t)$ is the total discounted future reward that the agent will experience along the trajectory $[a_t, s_t, a_{t+1}, s_{t+1}, a_{t+2}, s_{t+2}, ...]$. Often, we can only find an approximation of the Q function. In this case, $Q(a_t, s_t)$ is a value that tries to predict the total discounted future reward.

An agent policy is optimal when it maximizes the total reward experienced by the agent. This means that when the agent is in the state s_t, the optimal policy π_* chooses the action a_t that maximizes G_t. Then, we can use the Q function to define an optimal policy by appling a *greedy* strategy:

$$\pi_*(s) = arg \max_a Q(a, s)$$

Therefore, if we can compute the Q function, we can implement the optimal policy π_*, which is the answer to the problem. Moreover, it is reasonable to think that finding a good approximation of the Q function should yield a good approximation of the optimal policy π_*. Indeed, finding a good approximation of the Q function is the goal of Q-learning algorithms, since it is implicit in the formulation of a RL problem that finding an exact formulation of the Q function is unfeasible.

Characteristics of Q-learning

Q-learning is a *values*-based learning algorithm. Value based algorithms updates the value function based on an equation (particularly Bellman equation). Whereas the other type, *pol-*

icy-based estimates the value function with a greedy policy obtained from the last policy improvement. To apply the Bellman equation to Q, we start by its definition:

$$Q(a_t, s_t) = G_t = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1} = r_{t+1} + \sum_{k=1}^{\infty} \gamma^k \cdot r_{t+k+1}$$

Considering that

$$Q(a_{t+1}, s_{t+1}) = G_{t+1} = \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+1+k+1} = \sum_{k=1}^{\infty} \gamma^{k-1} \cdot r_{t+k+1}$$

we see that if we multiply $Q(a_{t+1}, s_{t+1})$ by γ, we can then substitute it in the first equation, yielding:

$$Q(a_t, s_t) = r_{t+1} + \gamma Q(a_{t+1}, s_{t+1}) \tag{2.1}$$

which is the Bellman equation applied to the Q function.

At every transition $(s_t, a_t) \rightarrow (s_{t+1}, a_{t+1})$, the equation gives us a new estimate of $Q(a_t, s_t)$, which we can use to improve our current one. Since our estimate of $Q(a', s')$ improves only when the agent performs the action a' in the state s', it is evident that improving the approximation of Q depends on the trajectories performed by the agent during training, which depend on the policy of the agent. Intuitively, a *random-policy* ($\pi(s) = rand(A)$, where A is the action space) would ensure that in the long run, all the possible action-state pairs are encountered by the agent multiple times, but this is often unfeasible in practice. However, an optimal policy π_* can be implemented only knowing the Q values for the action-state pairs that will effectively be encountered by the agent using π_*. Therefore, we don't need good estimations of all the (a, s) pairs, but only for the ones we suspect would be encountered by an optimal agent. As the training progresses, our approximation of the Q function improves, and so does the corresponding greedy policy $\pi_Q(s) = arg \max_a Q(a, s)$. Therefore, as the training goes on, it would make sense to apply this approximation of $\pi_*(s)$ to the agent to approximate an optimal agent, and improve only the Q values for "optimal" (a, s) pairs.

This behaviour (random at the start, and then progressively more greedy) can be achieved using an *epsilon-greedy* policy. At every step t, an epsilon-greedy policy acts as random-policy with a probability of ϵ and as a greedy-policy with a probability of $1 - \epsilon$. As t increases, ϵ is decreased. One such way of accomplishing this is by using a *epsilon-decay factor* ϵ_{decay} and to set $\epsilon_{t+1} = \epsilon_t \cdot \epsilon_{decay}$.

Despite this, when applying the Bellman equation (2.1) at the step t, the next action a_{t+1} is always chosen considering a greedy agent that upon entering the state s_{t+1} will take the action $a_{t+1} = arg\max_a Q(a, s_{t+1})$. For this reason, Q-learning is an *off-policy* learner. If instead we choose a_{t+1} by applying the current agent's policy to the state s_{t+1} we have a *on-policy* learner (such as *SARSA*).

Chapter 3

Q-Learning and Deep-Q Learning

3.0.1 A Q-Learning implementation

In this implementation of the algorithm, we consider an environment discreet both in the action space A and in the state space S. The $Q(a, s)$ function is implemented as a table $Q[a, s]$. When updating the current $Q(a, s)$ value with the value provided by the Bellman equation (2.1), we actually perform an interpolation, using the learning factor $\alpha \in (0, 1]$. The ϵ value is used to perform the epsilon-greedy policy on the agent.

Algorithm 1: Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

```
Algorithm parameters: learning factor α ∈ (0, 1], small epsilon > 0;
Initialize Q(s, a), for all s ∈ S⁺, a ∈ A(s), arbitrarily except that Q(terminal, ·) = 0;
forall E ∈ Episodes do
    Initialize S;
    forall step ∈ E do
        Choose A from S using policy derived from Q (e.g., epsilon-greedy);
        Take action A, observe R, S′;
        Q(S, A) ← Q(S, A) + α[R + γ max_a Q(S′, a) − Q(S, A)];
        S ← S′;
    end
end
```

However, many Reinforcement Learning problem use environments with a finite action space A, but a continuous state space S. A common example is the realization of a robotic agent, where the actions are a fixed set of instruction for the actuators, and the state is given by the information provided by the sensors. Even when the state space is discrete, the sheer number of possible state-action pairs might make unfeasible storing the $Q[a, s]$ table.

A solution is to use a different function approximator for the Q-function, such as a neural network model. In this case, the technique takes the name of *Deep-Q Learning*, often abbreviated as *DQN*, where the *N* refers to the use of a neural network.

3.0.2 Deep Q-Learning

- Q_θ is a neural network model with parameters θ
- The input is a state $s \in \mathbb{R}^n$.
- The output is the vector $Q_\theta(s) \in \mathbb{R}^m$.
- Given a finite action space $A = a_1, a_2, ...a_m$, we define $Q_\theta(s, a_i)$ as $Q_\theta(s)[i]$.

Since we now use a neural network instead of a table as the Q-function approximator, we must change the update rule presented in Algorithm 1. Instead of performing an interpolation between the old and the new estimate of $Q(a, s)$, we will compare them with a loss function L, and use a gradient method to update the parameters θ of Q_θ in order to diminish the loss value. The exact operations behind this update won't be further detailed in this work, and from now on, we will assume that the reader has a basic understanding of neural networking models and their training.

Experience Replay Buffer

Reinforcement learning is unstable or divergent when a nonlinear function approximator such as a neural network is used to represent Q. This instability comes from the correlations present in the sequence of observations, the fact that small updates to Q may significantly change the policy and the data distribution, and the correlations between Q and the target values.

A technique used to tackle this problem is *experience replay*, a biologically inspired mechanism that uses a random sample of prior actions instead of the most recent action to proceed. This removes correlations in the observation sequence and smooths changes in the data distribution.[4] Another advantage of the replay buffer is that, when updating the network, we can sample more than one transition from the buffer and combine them in a mini-batch. Following is a basic implementation of a DQN algorithm:

Algorithm 2: Basic DQN-learning algorithm

Algorithm parameters: learning factor $\alpha \in (0, 1]$, small $epsilon > 0$ loss function $L(x, x')$, network model Q_θ; Initialize the network parameters θ with arbitrary values;

forall $E \in Episodes$ **do**

  Initialize S;

  forall $step \in E$ **do**

    Choose A from S using policy derived from Q_θ (e.g., epsilon-greedy);

    Take action A, observe R, S';

    $\theta \leftarrow \theta - \alpha \nabla_\theta L[R + \gamma \max_a Q_\theta(S', a), Q_\theta(S, A)]$;

    $S \leftarrow S'$;

  end

end

Note that in Algorithm 2, a simple gradient descent method has been reported for the update of the network parameters θ. However, in the actual implementation of the algorithm, different gradient methods can be used, along with different optimizers (such as *ADAM*).

Target Network

Looking at the functioning of Algorithm 2, it is easy to see that every update step of the parameters θ immediately affect the following ones, since both member of the loss function depend on the network output. This can make the learning very unstable. For this reason, a *target network* is introduced. The aim is to decouple (to some extension) the network we are training from the one we are using for exploration. During training, the agent policy will use the target network $Q_{T\theta}$, instead of Q_θ. $Q_{T\theta}$ shares the same topology with Q_θ, but uses a different parameter vector $T\theta$. The parameters $T\theta$ are synced periodically to θ. In this way, the training agent policy uses a network that is close to our current estimate, but does not get updated at every single step. When applying the update rule to θ, we also use the target network to estimate the value of the next state S'. Following is an implementation of DQN using a target network:

Algorithm 3: DQN-learning algorithm (with target network)

Algorithm parameters: learning factor $\alpha \in (0, 1]$, small $epsilon > 0$ loss function $L(x, x')$, network models Q_θ and $Q_{T\theta}$ with the same topology, target network update period T_{steps}; Initialize the network parameters θ with arbitrary values; Initialize $T\theta = \theta$; Initialize $t = 0$;

forall $E \in Episodes$ **do**
    Initialize S;
    forall $step \in E$ **do**
        Choose A from S using policy derived from $Q_{T\theta}$ (e.g., epsilon-greedy);
        Take action A, observe R, S';
        $\theta \leftarrow \theta - \alpha \nabla_\theta L[R + \gamma \max_a Q_{T\theta}(S', a), Q_\theta(S, A)]$;
        $S \leftarrow S'$;
        if $t \,\%\, T_{steps} == 0$ **then** $T\theta = \theta$;
        $t{+}{+}$;
    end
end

Prioritized DQN (PDQN)

As we have stated before when introducing the Experience Replay Buffer, the actual implementation of a DQN algorithm does not use the transitions as they are performed by the training agent: instead, when agent performs a transition, this is added to the replay buffer. Conversely, when we want to update the parameters θ, we sample one transition randomly from the replay buffer (or more, if using mini-batches). The idea of Prioritized DQN is that transitions that produce higher loss values contribute more significantly to the learning of the network, and should therefore be sampled more often from the buffer. To achieve this, we assign a *priority* value (initially random) to each transition: a higher priority means a higher chance to be sampled from the buffer. When a transition is sampled and its loss value is computed, we can also update its priority. The exact formula won't be detailed: what matters, is that a higher loss leads to a higher transition priority.

Chapter 4

DQN using Julia

The code I have developed for this project is licensed under MIT-license, and can be read at my GitHub repository [3].

4.0.1 The Lunar Lander environment

The Lunar Lander is one of the Box2D environments included offered by *OpenAI Gym* python library. OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. It gives access to a standardized set of environments. It is open-source, and licensed under MIT-license. The source code is available at the GitHub repository page [1]. For this work I used the Lunar Lander environment, which is one of the gym environments that uses Box2D to simulate a 2D physical system.
The Lunar Lander environment consists in a space-ship agent that must land correctly using three propellers. From the source code:

””” Rocket trajectory optimization is a classic topic in Optimal Control. According to Pontryagin's maximum principle it's optimal to fire engine full throttle or turn it off. That's the reason this environment is OK to have discreet actions (engine on or off). The landing pad is always at coordinates (0,0). The coordinates are the first two numbers in the state vector. Reward for moving from the top of the screen to the landing pad and zero speed is about 100..140 points. If the lander moves away from the landing pad it loses reward. The episode finishes

if the lander crashes or comes to rest, receiving an additional -100 or +100 points. Each leg with ground contact is +10 points. Firing the main engine is -0.3 points each frame. Firing the side engine is -0.03 points each frame. Solved is 200 points. Landing outside the landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Please see the source code for details.
"""

As we can see, the reward function in the original Lunar Lander environment does not consider just the landing or crashing of the agent, but three other factors: the angle, the speed, and the position. However, all these are correlated to the main objective, since a good landing usually requires the spaceship to stay upright (low angle) and to descend slowly (low speed) on an even surface (such as the landing pad, which is always at coordinates [0,0]). For this reason, we can actually consider the environment to have only one objective.

The modified Lunar Lander environment

For this work, I have made a custom version of the Lunar Lander environment. In this version, the reward function has been modified, so that the agent receives or looses reward depending on which leg makes contact first with the ground. Specifically, the reward is increased by +50 when the left legs makes contact first. Conversely, the reward is decreased by -50 when the right leg makes contact first.

Moreover, the lading and crashing of the agent generate respectively a reward of +500 and -500 (instead of +100 and -100). In this way, we ensure that the agent prioritizes landing correctly over landing with the left leg first. Note that these two objectives are not actually correlated, as it is easy for the agent to accomplish the second and fail the first, and vice versa. Note also that depending on the circumstances, trying to land with the left leg first might worsen the chances of a correct landing. Therefore, we expect that a well-trained (but not optimal) agent will have to forsake this secondary objective, prioritizing the main one.

4.0.2 Training the agent

I have applied the BasicDQN, DQN and PDQN algorithms (explained in the previous chapter) to both the original and custom environment. The algorithms implementation comes from the *ReinforcementLearning* julia library. In the `./src/conf.jl` file, I define the algorithms hyper-parameters:

```julia
#JuliaRL/src/conf.jl

# Configuration values for the DQN algorithms are defined here
module Conf

# BasicDQN, DQN, PDQN

# duration in steps of the training
duration = 150000
# size of mini-batches
batchsize = 64
# number of transitions that should be experienced before updating the
↪  approximator
minreplayhistory = 100
# decaysteps for EpsilonGreedyExplorer
decaysteps = 3000
# capacity (in steps) of the experience buffer
capacity = duration
# frequency at which the agent is saved during training
savefreq = div(duration,2)

# DQN, PDQN

# the frequency of updating the approximator
updatefreq = 4
# the frequency of updating the target
targetupdatefreq = 100

end
```

The topology of the neural network model used is defined in `./src/shared.jl`:

```
# in ./src/shared.jl
function netmodel(ns::Int, na::Int, rng)
    Chain(
        Dense(ns, 64, leakyrelu; initW = glorotuniform(rng)),
        Dense(64, 64, leakyrelu; initW = glorotuniform(rng)),
        Dense(64, 32, leakyrelu; initW = glorotuniform(rng)),
        Dense(32, na; initW = glorotuniform(rng)),
    ) −¿ cpu
end
```

The hyper-parameters and the network topology choice is based on the article ”Solving Lunar Lander with Double Dueling Deep Q-Network and PyTorch” from Le Hoang Van [5], where different hyper-parameters configuration are compared.

4.0.3 Results

Here are shown the results of the training runs. For each run, the total loss per episode and the total reward per episode are plotted. Note that, even though all the runs have the same duration in number of steps (defined in `./src/conf.jl`), the number of episodes differs (since the agent behaviours determines how much an episode lasts). Also note that the plots show a running average over the episodes of the actual values (with the actual values shown with a faint line).

Figure 4.1: Training with BasicDQN and original environment

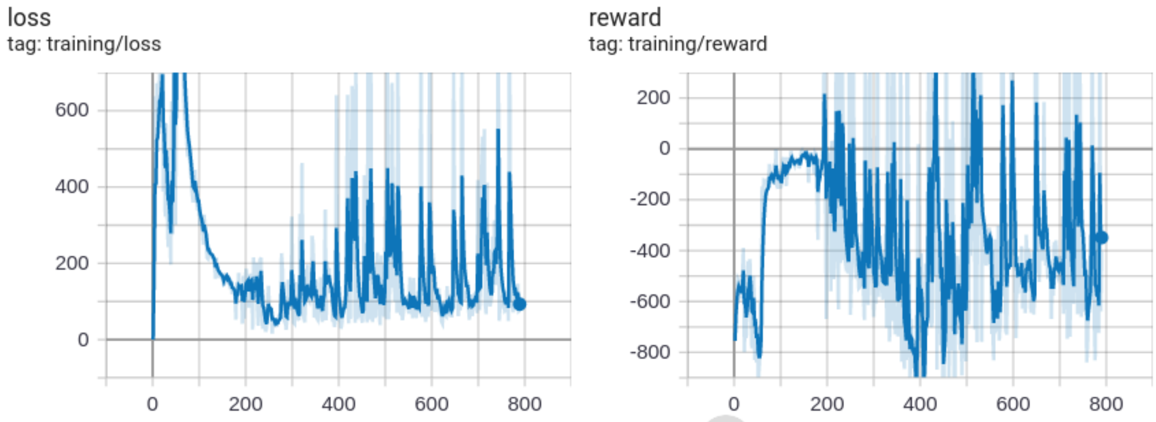

Figure 4.2: Training with BasicDQN and custom environment

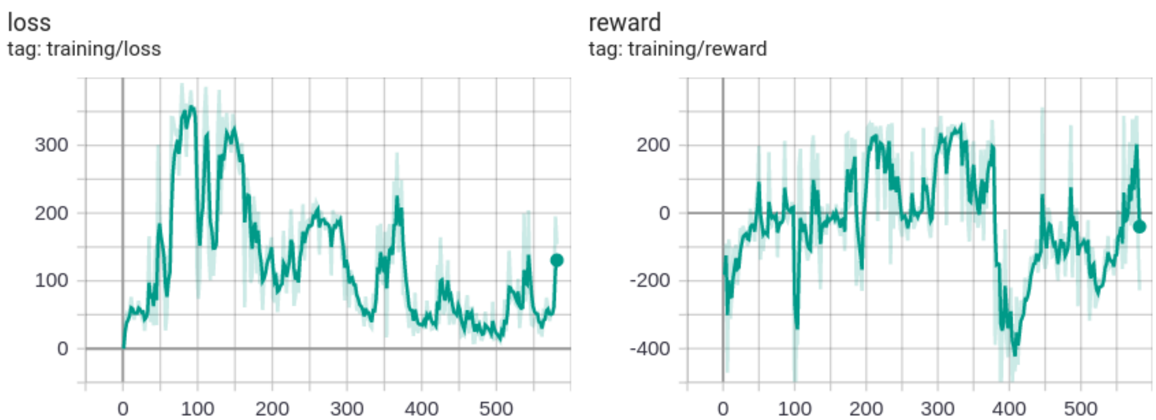

Figure 4.3: Training with DQN and original environment

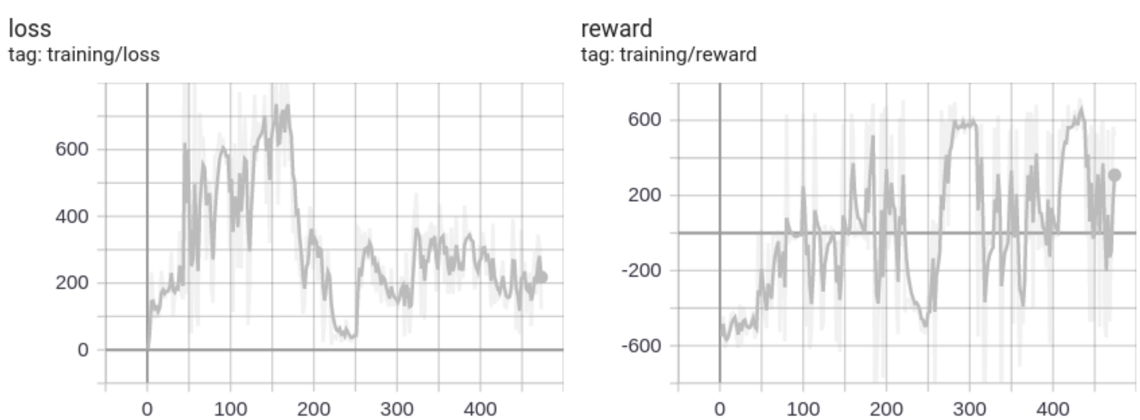

Figure 4.4: Training with DQN and custom environment

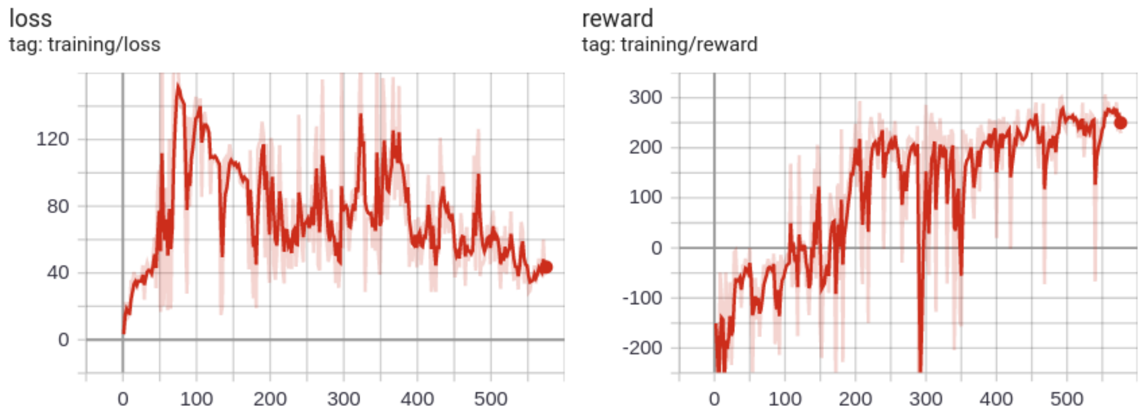

Figure 4.5: Training with PDQN and original environment

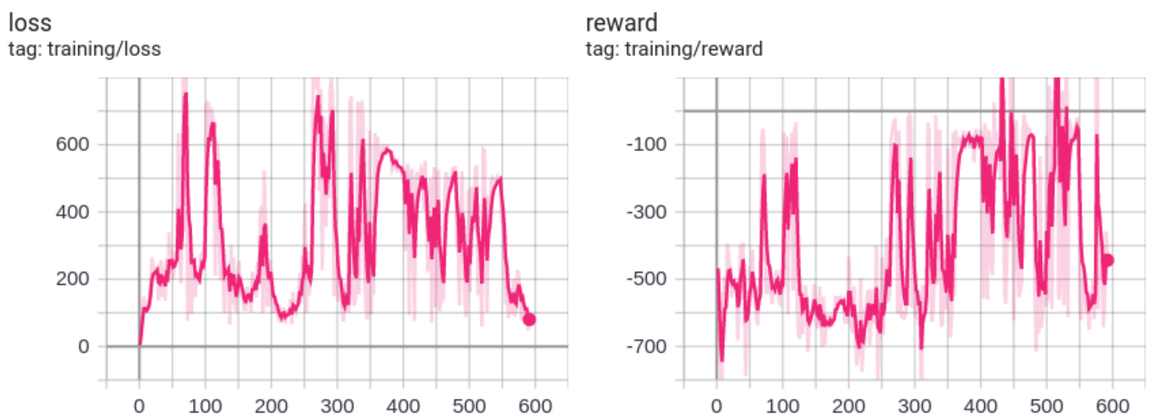

Figure 4.6: Training with PDQN and custom environment

Looking at the plots, we can see that only the PDQN algorithm, applied to the original environment, converges successfully to a solution.
We can notice that the use of a target network did not make DQN much better than BasicDQN at solving the problem (although both the loss and the reward do appear to be more stable).
Conversely, PDQN seems to perform much better than DQN, thanks to the use of a priority system in the transitions sampling. However, even PDQN fails to converge on the custom environment. This is somewhat expected, since the introduction of a secondary objective has made the reward function much more ”noisy”.
It is therefore reasonable to assume that the model requires more time to solve the problem posed by the custom environment. Figure 4.5 shows the plot relative to a run of $4e5$ steps,

using PDQN on the custom environment. We can see that the model converges to a solution that can achieve consistently a total episodic reward of 600 or more (but also of -200 or more, and is therefore far from being an optimal agent).

Figure 4.7: Training with PDQN and custom environment ($4e5$ steps)

Chapter 5

Conclusions and future works

This work shows that using DQN techniques, even a simple environment can become much harder to solve when a secondary, non-correlated goal is introduced in the reward function. It also shows that using a prioritized sampling of the experience replay buffer (PDQN) brought a noticeable improvement to the solving of the environment. Further work might explore other classes of algorithms (such as *Actor-Critic*) or different ways of implementing a multi-objective environment.

Concerning to the implementation, I have found Julia to be a flexible and very expressive programming language, although it took some work before I could find myself comfortable with it.

Appendix A

Code

The code I have developed for this work can be read at the GitHub repository `FeLusiani/JuliaRL` [3]. However, it is also shown below for completeness. I have omitted the file `lunarlander.py`, since the modifications to the original code pertain to a relatively small fraction of lines.

```
#JuliaRL/src/JuliaRL.jl

include("./BasicDQN.jl")
include("./DQN.jl")
include("./PDQN.jl")
include("./A2C.jl")

include("./display.jl")
include("./showstats.jl")
```

```
#JuliaRL/src/conf.jl

# Configuration values for the DQN algorithms are defined here
module Conf

# BasicDQN, DQN, PDQN

# duration in steps of the training
duration = 150000
# size of mini-batches
batchsize = 64
# number of transitions that should be experienced before updating the
↪  approximator
```

```
minreplayhistory = 100
# decaysteps for EpsilonGreedyExplorer
decaysteps = 3000
# capacity (in steps) of the experience buffer
capacity = duration
# frequency at which the agent is saved during training
savefreq = div(duration,2)

# DQN, PDQN

# the frequency of updating the approximator
updatefreq = 4
# the frequency of updating the target
targetupdatefreq = 100

end
```

```
#JuliaRL/src/shared.jl

using ReinforcementLearning
using ReinforcementLearningEnvironments
using Flux
using Dates
using Suppressor


"""
    pyenv2env(pyenv::PyObject)

Returns a `ReinforcementLearningEnvironments.GymEnv` environment
from the `gym` environment `pyenv`.
Code is from `ReinforcementLearningEnvironments.GymEnv(::String)` function.
"""
function pyenv2env(pyenv::PyObject)
    obsspace = convert(AbstractSpace, pyenv.observationspace)
    actspace = convert(AbstractSpace, pyenv.actionspace)
    obstype = if obsspace isa Union-MultiContinuousSpace,MultiDiscreteSpace
        PyArray
    elseif obsspace isa ContinuousSpace
        Float64
    elseif obsspace isa DiscreteSpace
        Int
```

```
    elseif obsspace isa VectSpace
        PyVector
    elseif obsspace isa DictSpace
        PyDict
    else
        error(”don't know how to get the observation type from observation space of $obsspace”)
    end
    env = GymEnv-obstype,typeof(actspace),typeof(obsspace),typeof(pyenv)(
        pyenv,
        obsspace,
        actspace,
        PyNULL(),
    )
    RLBase.reset!(env) # reset immediately to init env.state
    env
end



”””
Returns a `LunarLander` environment from the `CustomGym` python module
as a `ReinforcementLearningEnvironments.GymEnv`
”””
function LunarLander(landingFactor=1, legFirstBonus=0)
    gym = pyimport(”CustomGym”)
    # reload the python module, otherwise changes to the code
    # won't be effective until you restart julia
    gym = pyimport(”importlib”).reload(gym.lunarlander)
    gym.LunarLander(landingfactor=landingFactor, legfirstbonus=legFirstBonus) –¿
    pyenv2env –¿
    ActionTransformedEnv(a -¿ a-1;mapping= a -¿ a+1)

end


function netmodel(ns::Int, na::Int, rng)
    Chain(
        Dense(ns, 64, leakyrelu; initW = glorotuniform(rng)),
        Dense(64, 64, leakyrelu; initW = glorotuniform(rng)),
        Dense(64, 32, leakyrelu; initW = glorotuniform(rng)),
        Dense(32, na; initW = glorotuniform(rng)),
    ) –¿ cpu
```

```
end


function shallownetmodel(ns::Int, na::Int, rng)
    Chain(
        Dense(ns, 128, leakyrelu; initW = glorotuniform(rng)),
        Dense(128, 64, leakyrelu; initW = glorotuniform(rng)),
        Dense(64, na; initW = glorotuniform(rng)),
    ) −¿ cpu
end


”””
@timeRet expr

Works like the macro `@time` from `Base`,
but returns the printed statistics as a string.
”””
macro timeRet(ex)
    quote
        while false; end # compiler heuristic: compile this block (alter this if the heuristic changes)
        local stats = Base.gcnum()
        local elapsedtime = timens()
        local val = $(esc(ex))
        elapsedtime = Base.timens() - elapsedtime
        local diff = Base.GCDiff(Base.gcnum(), stats)
        local output = @captureout Base.timeprint(
            elapsedtime, diff.allocd,
            diff.totaltime,
            Base.gcalloccount(diff)
        )
        print(output)
        println()
        output
    end
end


”””
    logtraininginfo(infos, agent, savedir)

Creates file `traininginfos.txt` inside of the `savedir` directory,
containing training infos (agent structure and elapsed time)
”””
```

```
function logtraininginfo(infos, agent, savedir)
    filepath = joinpath(savedir, ”traininginfos.txt”)
    open(f-¿write(f, infos*”“n”, string(agent)), filepath, ”w”)
end
```

```
#JuliaRL/src/BasicDQN.jl

using ReinforcementLearning
using PyCall
using ReinforcementLearningEnvironments
using Random
using Flux
using TensorBoardLogger
using Dates
using Logging
using BSON
using Suppressor
include(”./conf.jl”)
include(”./shared.jl”)


”””
    runBasicDQN(savedir, landingfactor=1, legfirstbonus=0)

Trains the modified LunarLander agent using BasicDQN
from the ReinforcementLearning library.
Results will be saved at `savedir`.

Using the default values for landingfactor and legfirstbonus
will make the environement behave like the original LunarLander.
For more information, see the
 ↪  JuliaRL/CustomGym/CustomGym/lunarlander.py file.
”””
function runBasicDQN(savedir::T, landingfactor=1, legfirstbonus=0)
 ↪  where -T¡:AbstractString
    # clear savedir directory
    isdir(savedir) && rm(savedir; force=true, recursive=true)

    lg = TBLogger(joinpath(savedir, ”tblog”), minlevel = Logging.Info)
    rng = MersenneTwister(123)

    env = LunarLander(landingfactor, legfirstbonus)

    ns, na = length(getstate(env)), length(getactions(env))
```

```julia

    agent = Agent(
        policy = QBasedPolicy(
            learner = BasicDQNLearner(
                approximator = NeuralNetworkApproximator(
                    model = netmodel(ns, na, rng),
                    optimizer = ADAM(),
                ),
                batchsize = Conf.batchsize,
                minreplayhistory = Conf.minreplayhistory,
                lossfunc = huberloss,
                rng = rng,
            ),
            explorer = EpsilonGreedyExplorer(
                kind = :exp,
                stable = 0.01,
                decaysteps = Conf.decaysteps,
                rng = rng,
            ),
        ),
        trajectory = CircularCompactSARTSATrajectory(
            capacity = Conf.capacity,
            statetype = Float32,
            statesize = (ns,),
        ),
    )

    stopcondition = StopAfterStep(Conf.duration)

    global episodeloss = 0
    totalrewardperepisode = TotalRewardPerEpisode()
    timeperstep = TimePerStep()
    hook = ComposedHook(
        totalrewardperepisode,
        timeperstep,
        DoEveryNStep() do t, agent, env
            global episodeloss += loss = agent.policy.learner.loss
        end,
        DoEveryNEpisode() do t, agent, env
            withlogger(lg) do
                global episodeloss
                @info "training" loss = episodeloss
                @info "training" reward =
                 ↪  totalrewardperepisode.rewards[end]
                logstepincrement = 0
```

```
                episodeloss = 0
            end
        end,
        DoEveryNStep(Conf.savefreq) do t, agent, env
            RLCore.save(savedir, agent)
            BSON.@save joinpath(savedir, "stats.bson") totalrewardperepisode timeperstep
        end,
    )


    infos = @timeRet run(agent, env, stopcondition, hook)
    logtraininginfo(infos, agent, savedir)
end
```

```
#JuliaRL/src/DQN.jl

using ReinforcementLearning
using PyCall
using ReinforcementLearningEnvironments
using Random
using Flux
using TensorBoardLogger
using Dates
using Logging
using BSON
using Suppressor
include("./conf.jl")
include("./shared.jl")


"""
    runDQN(savedir, landingfactor=1, legfirstbonus=0)

Trains the modified LunarLander agent using DQN
from the ReinforcementLearning library.
Results will be saved at `savedir`.

Using the default values for landingfactor and legfirstbonus
will make the environement behave like the original LunarLander.
For more information, see the JuliaRL/CustomGym/CustomGym/lunarlander.py file.
"""
```

```
function runDQN(savedir::T, landingfactor=1, legfirstbonus=0) where
 ↪ -T¡:AbstractString
    # clear savedir directory
    isdir(savedir) && rm(savedir; force=true, recursive=true)

    lg = TBLogger(joinpath(savedir, ”tblog”), minlevel = Logging.Info)
    rng = MersenneTwister(123)

    env = LunarLander(landingfactor, legfirstbonus)
    ns, na = length(getstate(env)), length(getactions(env))

    agent = Agent(
        policy = QBasedPolicy(
            learner = DQNLearner(
                approximator = NeuralNetworkApproximator(
                    model = netmodel(ns, na, rng),
                    optimizer = ADAM(),
                ),
                targetapproximator = NeuralNetworkApproximator(
                    model = netmodel(ns, na, rng),
                    optimizer = ADAM(),
                ),
                lossfunc = huberloss,
                stacksize = nothing,
                batchsize = Conf.batchsize,
                updatehorizon = 1,
                minreplayhistory = Conf.minreplayhistory,
                updatefreq = Conf.updatefreq,
                targetupdatefreq = Conf.targetupdatefreq,
                rng = rng,
            ),
            explorer = EpsilonGreedyExplorer(
                kind = :exp,
                stable = 0.01,
                decaysteps = Conf.decaysteps,
                rng = rng,
            ),
        ),
        trajectory = CircularCompactSARTSATrajectory(
            capacity = Conf.capacity,
            statetype = Float32,
            statesize = (ns,),
        ),
    )

```

```
    global episodeloss = 0
    stopcondition = StopAfterStep(Conf.duration)
    totalrewardperepisode = TotalRewardPerEpisode()
    timeperstep = TimePerStep()
    hook = ComposedHook(
        totalrewardperepisode,
        timeperstep,
        DoEveryNStep() do t, agent, env
            if agent.policy.learner.updatestep % agent.policy.learner.updatefreq == 0
                global episodeloss += agent.policy.learner.loss
            end
        end,
        DoEveryNEpisode() do t, agent, env
            withlogger(lg) do
                global episodeloss
                @info "training" loss = episodeloss
                @info "training" reward = totalrewardperepisode.rewards[end]
                logstepincrement = 0
                episodeloss = 0
            end
            episodeloss = 0
        end,
        DoEveryNStep(Conf.savefreq) do t, agent, env
            RLCore.save(savedir, agent)
            BSON.@save joinpath(savedir, "stats.bson") totalrewardperepisode timeperstep
        end,
    )


    infos = @timeRet run(agent, env, stopcondition, hook)
    logtraininginfo(infos, agent, savedir)
end
```

```
#JuliaRL/src/PDQN.jl

using ReinforcementLearning
using PyCall
using ReinforcementLearningEnvironments
using Random
using Flux
using TensorBoardLogger
```

```julia
using Dates
using Logging
using BSON
using Suppressor
include(”./conf.jl”)
include(”./shared.jl”)


”””
    runPDQN(savedir, landingfactor=1, legfirstbonus=0)

Trains the modified LunarLander agent using PDQN
from the ReinforcementLearning library.
Results will be saved at `savedir`.

Using the default values for landingfactor and legfirstbonus
will make the environement behave like the original LunarLander.
For more information, see the
↪   JuliaRL/CustomGym/CustomGym/lunarlander.py file.
”””
function runPDQN(savedir::T, landingfactor=1, legfirstbonus=0) where
↪   -T¡:AbstractString
    # clear savedir directory
    isdir(savedir) && rm(savedir; force=true, recursive=true)

    lg = TBLogger(joinpath(savedir, ”tblog”), minlevel = Logging.Info)
    rng = MersenneTwister(123)

    env = LunarLander(landingfactor, legfirstbonus)
    ns, na = length(getstate(env)), length(getactions(env))

    agent = Agent(
        policy = QBasedPolicy(
            learner = PrioritizedDQNLearner(
                approximator = NeuralNetworkApproximator(
                    model = netmodel(ns, na, rng),
                    optimizer = ADAM(),
                ),
                targetapproximator = NeuralNetworkApproximator(
                    model = netmodel(ns, na, rng),
                    optimizer = ADAM(),
                ),
                lossfunc = huberlossunreduced,
                stacksize = nothing,
                batchsize = Conf.batchsize,
```

```
                updatehorizon = 1,
                minreplayhistory = Conf.minreplayhistory,
                updatefreq = Conf.updatefreq,
                targetupdatefreq = Conf.targetupdatefreq,
                rng = rng,
            ),
            explorer = EpsilonGreedyExplorer(
                kind = :exp,
                stable = 0.01,
                decaysteps = Conf.decaysteps,
                rng = rng,
            ),
        ),
        trajectory = CircularCompactPSARTSATrajectory(
            capacity = Conf.capacity,
            statetype = Float32,
            statesize = (ns,),
        ),
    )

    global episodeloss = 0
    stopcondition = StopAfterStep(Conf.duration)
    totalrewardperepisode = TotalRewardPerEpisode()
    timeperstep = TimePerStep()
    hook = ComposedHook(
        totalrewardperepisode,
        timeperstep,
        DoEveryNStep() do t, agent, env
            if agent.policy.learner.updatestep %
             ↪  agent.policy.learner.updatefreq == 0
                global episodeloss += agent.policy.learner.loss
            end
        end,
        DoEveryNEpisode() do t, agent, env
            withlogger(lg) do
                global episodeloss
                @info "training" loss = episodeloss
                @info "training" reward =
                 ↪  totalrewardperepisode.rewards[end]
                logstepincrement = 0
                episodeloss = 0
            end
            episodeloss = 0
        end,
        DoEveryNStep(Conf.savefreq) do t, agent, env
```

```
            RLCore.save(savedir, agent)
            BSON.@save joinpath(savedir, ”stats.bson”)
             ↪  totalrewardperepisode timeperstep
        end,
    )


    infos = @timeRet run(agent, env, stopcondition, hook)
    logtraininginfo(infos, agent, savedir)
end
```

```
#JuliaRL/src/A2C.jl

using ReinforcementLearning
using PyCall
using ReinforcementLearningEnvironments
using Random
using Flux
using TensorBoardLogger
using Dates
using Logging
using BSON
using Suppressor
include(”./conf.jl”)
include(”./shared.jl”)


”””
    runA2Csavedir, landingfactor=1, legfirstbonus=0)

Trains the modified LunarLander agent using A2C
from the ReinforcementLearning library.
Results will be saved at `savedir`.

Using the default values for landingfactor and legfirstbonus
will make the environement behave like the original LunarLander.
For more information, see the
 ↪  JuliaRL/CustomGym/CustomGym/lunarlander.py file.
”””
function runA2C(savedir::T, landingfactor=1, legfirstbonus=0) where
 ↪  -T¡:AbstractString
    # clear savedir directory
    isdir(savedir) && rm(savedir; force=true, recursive=true)

```

```
    lg = TBLogger(joinpath(savedir, ”tblog”), minlevel = Logging.Info)
    rng = MersenneTwister(123)

    NENV = 16
    UPDATEFREQ = 10
    env = MultiThreadEnv([
        LunarLander(landingfactor, legfirstbonus) for i in 1:NENV
    ])

    ns, na = length(getstate(env[1])), length(getactions(env[1]))




    # RLBase.reset!(env, isforce = true)
    agent = Agent(
        policy = QBasedPolicy(
            learner = A2CLearner(
                approximator = ActorCritic(
                    actor = smallnetmodel(ns, na),
                    critic = smallnetmodel(ns, 1),
                    optimizer = ADAM(),
                ) −¿ cpu,
                 = 0.99f0,
                actorlossweight = 1.0f0,
                criticlossweight = 0.5f0,
                entropylossweight = 0.001f0,
            ),
            explorer = BatchExplorer(GumbelSoftmaxExplorer()),#= seed = nothing =#
        ),
        trajectory = CircularCompactSARTSATrajectory(;
            capacity = UPDATEFREQ,
            statetype = Float32,
            statesize = (ns, NENV),
            actiontype = Int,
            actionsize = (NENV,),
            rewardtype = Float32,
            rewardsize = (NENV,),
            terminaltype = Bool,
            terminalsize = (NENV,),
        ),
    )
    stopcondition = StopAfterStep(Conf.duration)
    totalrewardperepisode = TotalBatchRewardPerEpisode(NENV)
```

```
    timeperstep = TimePerStep()
    hook = ComposedHook(
        totalrewardperepisode,
        timeperstep,
        DoEveryNStep() do t, agent, env
            withlogger(lg) do
                @info(
                    "trainingAC",
                    actorloss = agent.policy.learner.actorloss,
                    criticloss = agent.policy.learner.criticloss,
                    entropyloss = agent.policy.learner.entropyloss,
                    loss = agent.policy.learner.loss,
                )
                for i in 1:length(env)
                    if getterminal(env[i])
                        @info "trainingAC" reward =
                         ↪  totalrewardperepisode.rewards[i][end]
                         ↪  logstepincrement =
                            0
                        break
                    end
                end
            end
        end,
        DoEveryNStep(Conf.savefreq) do t, agent, env
            RLCore.save(savedir, agent)
            BSON.@save joinpath(savedir, "stats.bson")
             ↪  totalrewardperepisode timeperstep
        end,
    )

    infos = @timeRet run(agent, env, stopcondition, hook)
    logtraininginfo(infos, agent, savedir)
end
```

```
#JuliaRL/src/display.jl

using ReinforcementLearning
using PyCall
using Random
using ReinforcementLearningEnvironments
using Flux
using Logging

```

```
include(”./shared.jl”)

”””
    displayagent(savedir, duration::Int = 1000)

Runs the agent saved at `savedir`, rendering the environment,
up to `duration` steps.
”””
function displayagent(savedir, duration::Int = 5000)
    env = LunarLander()
    agent = RLCore.load(savedir, Agent)
    Flux.testmode!(agent)

    stopcondition = StopAfterStep(duration)
    disphook = DoEveryNStep(1) do t, agent, env
        env.env.pyenv.render()
    end

    disphook = DoEveryNStep(1) do t, agent, env
        env.env.pyenv.render()
    end

    printhook = DoEveryNEpisode(1) do t, agent, env
        println(”- Ep N $t”)
    end

    hook = ComposedHook(disphook, printhook)

    run(agent, env, stopcondition, hook)
end
```

```
#JuliaRL/src/showstats.jl

using BSON
using PyPlot


”””
    showstats(savedir)

Plots the stats saved in the `stats.bson` file inside of `savedir`.
Stats are total reward per episode, and time per step.
”””
function showstats(savedir::T) where -T¡:AbstractString
```

```
    BSON.@load joinpath(savedir,"stats.bson") totalrewardperepisode
     ↪  timeperstep

    figure(figsize=(10, 5))
    subplot(121)
    title(savedir)
    ylabel("Total reward")
    xlabel("Episode")
    plot(totalrewardperepisode.rewards)

    subplot(122)
    ylabel("Time")
    xlabel("Step")
    x = (1:length(timeperstep.times)) * 100
    y = timeperstep.times / 100
end
```

Bibliography

[1] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

[2] Kaelbling, Leslie P.; Littman, Michael L.; Moore, Andrew W. "A fast and elitist multiobjective genetic algorithm: NSGA-II". In: *Journal of Artificial Intelligence Research* 4.1 (1996), pp. 237–285.

[3] Federico Lusiani. *JuliaRL [GitHub repository]*. URL: https://github.com/FeLusiani/JuliaRL. (accessed: 2020-11-06).

[4] Matiisen, Tambet; Computational Neuroscience Lab. *Demystifying Deep Reinforcement Learning*. URL: https://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/. (accessed: 2020-11-06).

[5] Le Hoang Van. *Solving Lunar Lander with Double Dueling Deep Q-Network and PyTorch*. URL: https://drawar.github.io/blog/2019/05/12/lunar-lander-dqn.html. (accessed: 2020-11-06).