Upside Down Reinforcement Learning (UDRL)

Iván Hidalgo
Gorka Arrausi
Iñigo Rojo

Index

Introduction

In this report we are presenting the technical report of Jürgen Schmidhuber on the topic of upside down reinforcement learning presented in the 5th of december of 2009. This new technique is a spin on the classical reinforced learning algorithms, by turning the problem into one of supervised learning allowing the agent to learn to decide actions receiving a desired reward and the observation of the environment as a base instead of the usual prediction of rewards and actions decided based on that prediction that traditional reinforced learning employs.

This new model of machine learning is very useful as it was thought to be impossible to fully solve the problems of reinforced learning with supervised learning, as the agent would receive feedback on the usefulness of an action, not the optimal way to solve it creating errors within a learned dataset.

To finish this introduction one of the main results of this report is presented, a way to train agents in general model free settings without using value based algorithms like Q-learning (in which a markov decision process can find optimal policies maximizing the value of the total reward over any successive steps, starting on the current one).

Theory

To understand Upside Down Reinforcement Learning(UDRL) we must first explain the concept of Reinforcement Learning(RL).
The concept is pretty simple, an agent who interacts with his environment through different actions. When the agent performs an action, the environment answers with a new state and with a reward that the agent is gonna use as a feedback to the performed action. With this feedback the agent can learn if doing that is a good idea or a bad idea, putting the agent on the right way to achieve his objective.

In normal RL the actions are used as inputs to calculate the output value(the reward), but in Upside Down Reinforcement Learning the desired reward is used as the input and the action to achieve that input is obtained as output.
So, we have a desired return(the reward) that we want in a maximum number of timesteps(the horizon), and we use it as input along with the actual state of the environment to predict, using the behavior function, what is the best action to take at that moment.

In short it manages to learn by interacting with the environment using gradient descent to map self-generated commands to corresponding action probabilities with the possibility of using this acquired knowledge to solve new problems.

An agent may interact with its environment throughout a single prolonged period of time. At a given time, the history of actions and vector-valued costs like time, energy, pain and reward signals,etc will contain all the agent can know about the present state of itself and the environment. Its task then will be to obtain a lot of reward before some period of time elapses. For all past periods of time it can evaluate and implement additional, consistent, vector-valued command inputs for itself trying to achieve similar reward with less cost or to try to surpass that possible reward that happened in the past. It's possible now to use the gradient based supervised learning to assign our computer to map sensory inputs alongside our desired commands of rewards and the timeframe(horizons) to the known action sequences. If we find different but equally costly actions between our starting point and towards some goal we can train our computer to approximate the expected value of the appropriate actions.

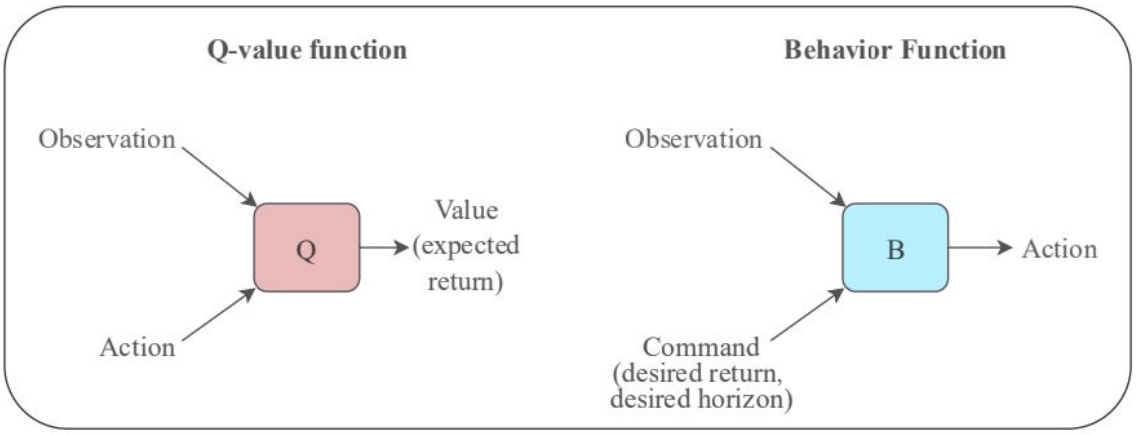
Through our single long periods of time our algorithm will learn to solve problems where there are logistical constraints of costs.

With this done we should be able to define commands of our own specific to problems affecting the user and hopefully our trained computer will be able to generalize based on what it has learned. This will not only help our problems, but also increase the experience of the computer which is ultimately what machine learning means: To learn from experience with respect to some task as measured by some method and to improve this performance as the experience grows.

Practice

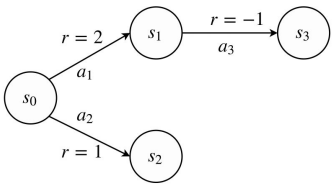
This section is based in the second paper about UDRL [2], in this paper they explain how this new technique works in some agents and the results of the experiments they had made.

First of all we have to understand in a more visual way how does UDRL works and what is the difference with traditional RL. This is the image that explains it:



The left function shows how the traditional RL works (in this case with Q-learning), as it has been explained before it receives an observation like the actual state and an action to make. Then with the value function Q we receive an output value, the reward.
The right function shows how the UDRL works (with a behaviour function). In this case the action is the output (the roles have been changed compared with traditional RL) and the inputs also have changed. The observation has not changed, but the command is a new input. This command contains the desired return (the desired reward) and the desired horizon (in how many steps do you want to obtain that reward). At first this may sound strange, with the next example explained in the paper it will be easier to understand.

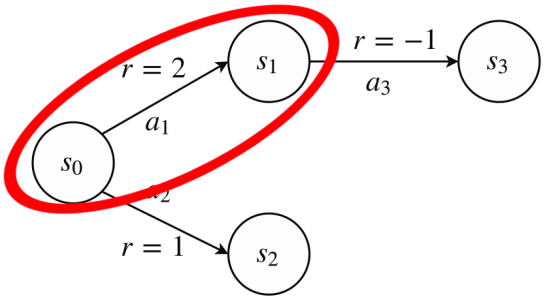
Imagine the next situation, with s_0 or s_1 as initial states of the trajectories and s_2 or s_3 as final states. The table of the right has the possible commands we can pass to the Behavior funcion, with its desired return and horizon.



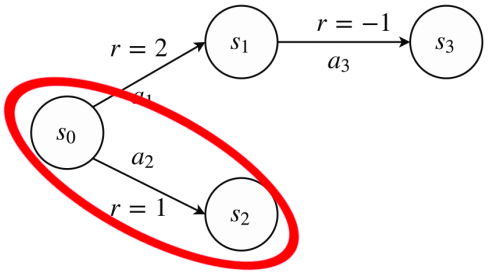
State	Desired Return	Desired Horizon	Action
s_0	2	1	a_1
s_0	1	1	a_2
s_0	1	2	a_1
s_1	−1	1	a_3

We will explain a bit what is in the table to understand what it means or represents each attribute. For example, if we are in state s_0 we can have three situation:

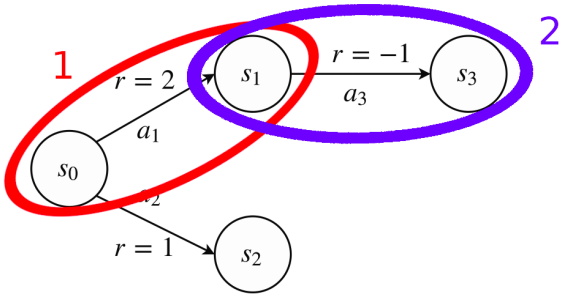
- "I want to obtain a reward = 2 in just 1 step", in this case the table will tell that the action that has to be made is a_1 (and that will be the output it will give). This will be its representation in the graph:



- "I want to obtain a reward = 1 in just 1 step", in this case the table will tell that the action that has to be made is a_2 (and that will be the output it will give). This will be its representation in the graph:

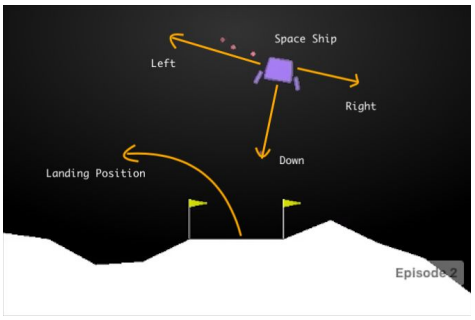


- "I want to obtain a reward = 1 in 2 step". This case is diferent because we have 2 steps, but it can only give the action for the first step; the next action is not important now, it will have to be decided later. That is why in this case the table will tell that the action that has to be made is a_1 (and that will be the output it will give). It decides a_1 because then it will decide a_2 , doing this the reward will be what we wanted: $r_t = r_1 + r_2 = 2 + (-1) = 1$. This will be its representation in the graph:



In the paper, they use two environments to show how UDRL works: LunarLander-v2 and TakeCover-v0. But I will only talk about LunarLander because I think its results are more interestings.

LunarLander is a Markovian environment where the objective is to land a spacecraft between two flags only using 3 engines (one in each side and another one in the bottom of the spacecraft). It is used as a AI Gym and is available in the Gym RL library. This is an image of LunarLander:



In LunarLander-v2 at the end of the episode there is a reward of +100 for landing well and -100 if not. But in this first environment during the episode, in each step there are also rewards to ensure that it is getting to the landing position.
This was the first set up:

- All agents were implemented using Artificial NN.
- The behaviour function of UDRL has been implemented with Fully-Connected Feed-Forward Networks.
- The command inputs suffered a small modification.

Then 20 seeds were used for each environment and algorithm and were sampled like this:

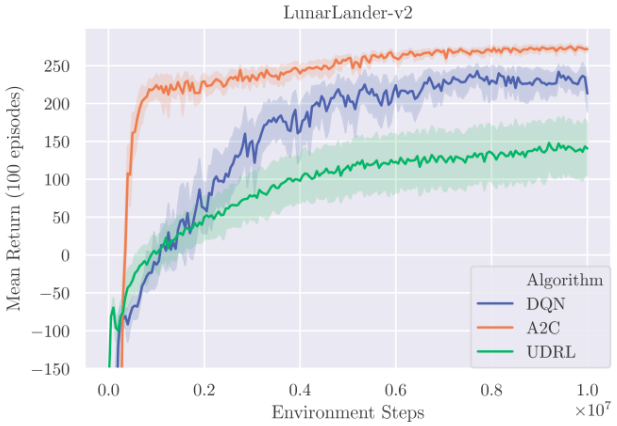
- [1M, 10M) for training
- [0.5M, 1M) for evaluation during hyperparameters tuning
- [1, 0.5M) for final evaluation with best hyperparameters

*These were the hyperparameters:

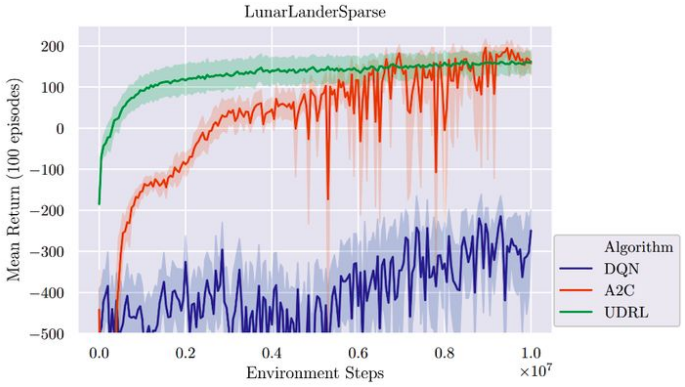
Table 2: A summary of UDRL hyperparameters

Name	Description
batch_size	Number of (input, target) pairs per batch used for training the behavior function
horizon_scale	Scaling factor for desired horizon input
last_few	Number of episodes from the end of the replay buffer used for sampling exploratory commands
learning_rate	Learning rate for the ADAM optimizer
n_episodes_per_iter	Number of exploratory episodes generated per step of UDRL training
n_updates_per_iter	Number of gradient-based updates of the behavior function per step of UDRL training
n_warm_up_episodes	Number of warm up episodes at the beginning of training
replay_size	Maximum size of the replay buffer (in episodes)
return_scale	Scaling factor for desired horizon input

In this case as you can see in the next graphic, UDRL has not gone very well. It does not do so bad, some of the agents learn quickly to land successfully like A2Q and DQN, but there are other agents who do not.



Then they changed a bit the Reward Structure. Now, all rewards were accumulated until the end of each episode and they were given to the agent at the last time step; to make this the rewards in other time steps were zero. The results of the las 20 runs are in the next graphic:



This change made UDRL much better than the other two, because they were having trouble with the delayed rewards and also with sparse rewards. But UDRL has been capable to train agents with both sparse and dense rewards; but sometimes sparse rewards worked better than dense depending on the environment.
It is curious how the same task but with different Rewards Structures the results can change so much and this is a thing to have into account.

Bibliography

[1] Paper - Reinforcement Learning Upside Down: Don't Predict Rewards -- Just Map Them to Actions

[2] Paper - Training Agents using Upside-Down Reinforcement Learning

[3] Blog - Demystifying Upside-Down Reinforcement Learning

[4] Blog - Is Upside-Down Reinforcement Learning = Imitation Learning?

[5] YouTube - Reinforcement Learning Upside Down: Don't Predict Rewards -- Just Map Them to Actions

[6] YouTube - Upside-Down Reinforcement Learning