Alma Mater Studiorum · University of Bologna

Flatland Challenge

Deep learning course final project

Lorenzo borelli (lorenzo.borelli@studio.unibo.it)
Giovanni Minelli (giovanni.minelli2@studio.unibo.it)
Lorenzo Turrini(lorenzo.turrini4@studio.unibo.it)

June 27, 2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The **Flatland challenge** consists in tackling the vehicle scheduling problem, in a controlled and simulated environment. The main approach used here is reinforcement learning, however there are several other possibilities that fall under the umbrella of combinatorial optimization.
The aim of the challenge, from a RL standpoint, is to allow the agents to learn how to optimally reach their target in minimal time.

Chapter 2

The environment

2.1 Railway network

The flatland environment is simulated by a rectangular grid of fixed size, which can be set by the user. Each cell of the grid is either a rail cell, or an empty unusable cell, or a **target** cell. Rails are of different type, depending on **transitions**: there are 16 different transitions in flatland, since there are 4 different orientations of the agent, and 4 other directions of exit from the cell. Thus, each cell is equipped with a bitmap that represents the whole transition space.
However, not every transition is allowed in flatland, since the aim is to actually simulate a real railway system: effectively, only 2 exit directions are allowed from every orientation of the agent, which result in 8 different cell types (including empty ones).

Flatland offers also a *sparse_rail_generator* that randomizes the creation of a realistic railway structure, where clusters of cities are sparsely connected to each other, allowing to mimic as faithfully as possible real city railway networks.

Figure 2.1: Sparse railway

2.2 Agents

Trains have a number of important properties:

- **Position**: the current coordinates of the agent.
- **Target**: the position of the target cell.
- **Direction**: the current orientation, with 0 corresponding to North, 1 to East, 2 to South, 3 to West.
- **Movement**: a flag that tells whether the agent is moving or not.

Since every agent is liable to malfunctions, much like real trains, there are properties to keep track of that, as well as variables that store the agents' speed:

- **Malfunction rate**: the Poisson rate at which malfunctions occur
- **Malfunction**: a counter of the remaining time the agent will remain malfunctioning

- **Next malfunction**: number of steps until next malfunction
- **Number of malfunctions**: the total number of malfunctions for this agent
- **Max speed**: a fraction between 0 and 1: a speed of 1/2 means that the agent changes cell after 2 time steps.
- **Position fraction**: related to speed, indicates when the next action can be taken

2.3 Observations

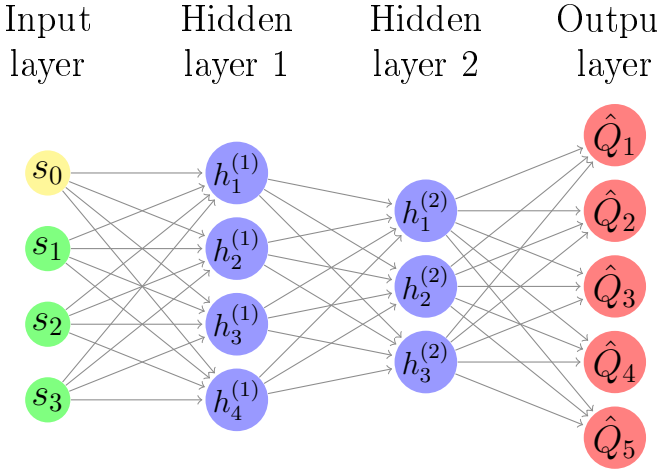
2.4 Our environment

Chapter 3

Reinforcement learning

3.1 DQN

We started by implementing a slight variation of DQN as described in [1], without the use of CNN, but by simply using fully connected layers with **ReLU**.



The units in the hidden layers default to 24 and 12, but the dimensions are fully customizable by the user, while the input layer is just made by the state of the environment, and the output layer has as many layers as there are actions in flatland. The model was also enriched with a few improvements:

- **Experience replay**: past experiences are recorded in a memory

buffer that is sampled, when needed, to train the network.

- **Fixed Q-targets**: since in a reinforcement learning setting we do not have target values prepared, we would need to use the same network for both the model's weights and for the target, leading to instability of the target values. Thus, we use 2 equal networks, one to just compute the target Q value, and the other for training. Every 100 time step, a tunable hyperparameter, the target network is *soft updated* with a portion of the current weights determined by this formula

$$\theta_i = \tau * \theta_i + (1 - \tau) * \theta_i$$

and by the hyperparameter τ. Furthermore, the model itself is updated every 4 time steps, which are customizable as well, so as to increase training speed.

- **Huber loss**: [2] explains the advantage in using a loss function that clips the gradient to a certain threshold, to avoid **exploding gradient**.

> *We also found it helpful to clip the error term from the update $r + \gamma max'_a Q(s', a'; \theta_i^-) - Q(s, a : \theta_i)$ to be between -1 and 1. Because the absolute value loss function $|x|$ has a derivative of -1 for all negative values of x and a derivative of 1 for all positive values of x, clipping the squared error to be between -1 and 1 corresponds to using an absolute value loss function for errors outside of the (-1,1) interval.This form of error clipping further improved the stability of the algorithm.*

While the squared loss is too sensitive to outliers, the Huber loss, after a certain threshold, maintains a constant derivative (see 3.1).

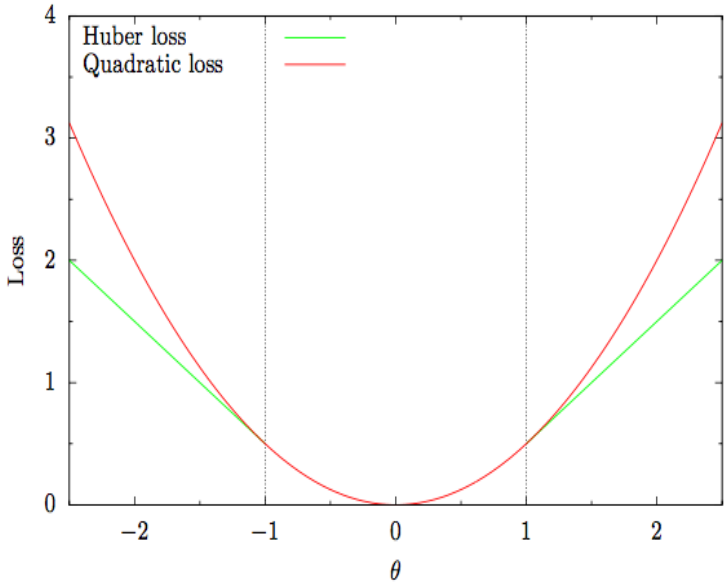


Figure 3.1: Huber and squared loss

- **Kaiming initialization**: a method for weights matrix initialization, described in [3], which draws samples from a standard normal distribution, to avoid **vanishing** and **exploding gradient**, and then adjusts this distribution to the ReLU activation function, by doubling the variance, since ReLU halves it w.r.t. to the original standard normal distribution (see 3.2).

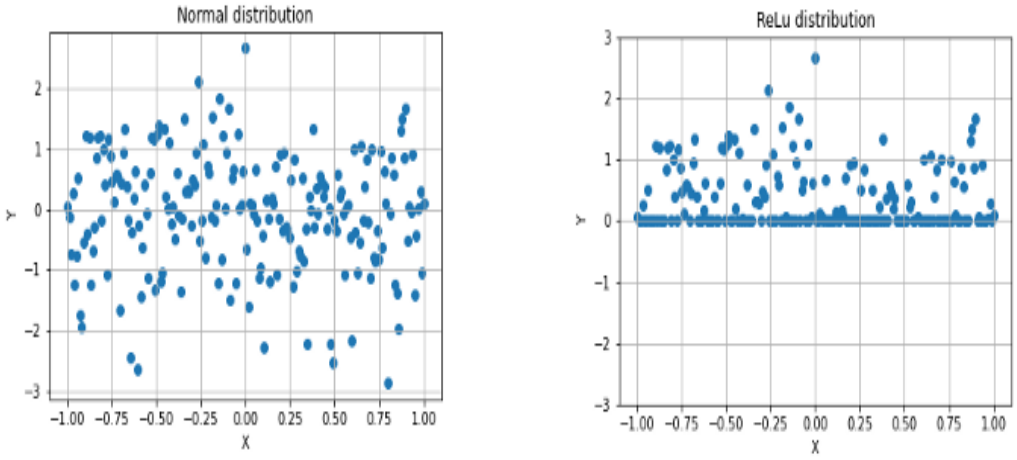


Figure 3.2: Normal and ReLU distributions

3.1.1 Dueling

The model was further improved with **dueling** architecture. The main issue of standard DQN lies in the fact that the **Bellman operator** tends to overestimate the Q value: as stated in [4], the solution seems to be to separate the computation of the Q value in 2 separate functions,

$$Q_\pi(s,a) = V_\pi(s) + A_\pi(s,a)$$

The **advantage** $A(s,a)$, which computes the "advantage" of taking the action a in state s w.r.t. the other possible actions in that state, and the **value** $V(s)$ which simply measures how good the state s is, regardless of the possible actions; this is particularly useful whenever we're dealing with environments that are not affected by actions in every state, like Flatland, where actions are only really relevant at switches.

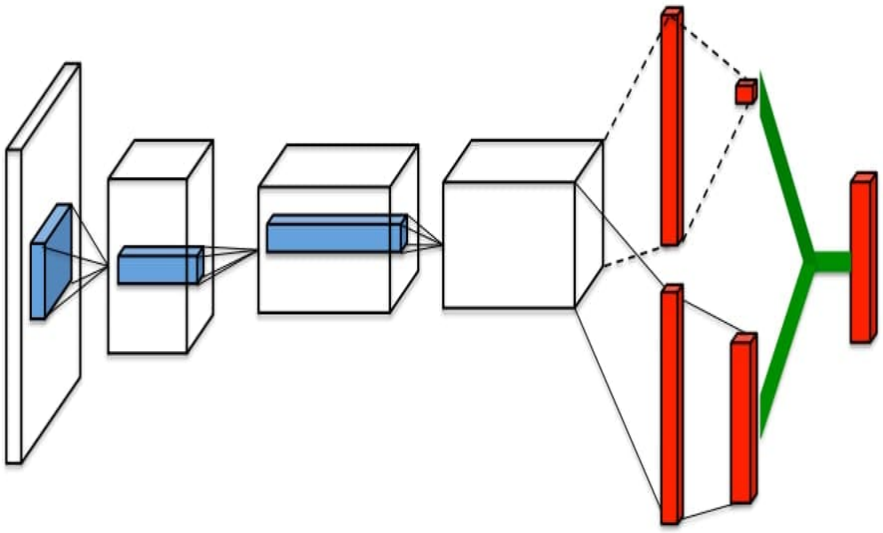


Figure 3.3: Dueling DQN

Advantage and value still need to be recombined, and since the standard formula doesn't allow to extract A and V from the Q value, [4] suggested to force the advantage to be 0 at the chosen best action, which results in the output formula for Q

$$Q_\pi(s,a) = V_\pi(s) + (A_\pi(s,a) - max_{a' in A} A(s,a'))$$

It also highlights, however, that empirically, subtracting the average advantage yields better results.

Bibliography

[1] Mnih et Al. *Playing Atari with Deep Reinforcement learning.* In: (2013). cite arxiv:1312.5602 NIPS Deep Learning Workshop 2013. url: http://arxiv.org/abs/1312.5602.

[2] Mnih et Al. *Human Level Control Through Deep Reinforcement Learning.* Nature, 518, pages 529–533, 2015.

[3] He et Al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.* ICCV '15: Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV), 2015. pp. 1026–1034 https://doi.org/10.1109/ICCV.2015.123

[4] Wang, Ziyu and Schaul, Tom and Hessel, Matteo and Hasselt, Hado and Lanctot, Marc and Freitas, Nando *Dueling Network Architectures for Deep Reinforcement Learning* Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48. ICML'16.New York, NY, USA: JMLR.org, 2016, pp. 1995–2003.