Path Planning for E-Puck Robot using Deep Reinforcement Learning algorithms

David Scarin[1], Gonçalo Dias[2], and Vicente Bandeira[3]
[1]up202108314@fc.up.pt, [2]up202108517@fc.up.pt, [3]up202106364@fc.up.pt

DCC: FCUP - Department of Computer Science of the Faculty of Sciences of the University of Porto, Portugal

Abstract. This paper reports the process of training a agent to learn to navigate an unknown environment in order to find the optimal path from a random origin position to a known target while avoiding obstacles. The agent is a Webots "e-puck" robot and the training and testing were simulated in a custom Webots world. The training process uses the robot's Light Detection and Ranging sensor (LiDAR), which provides point cloud readings as input to different neural networks; we experimented with a Deep-Q Learning (DQN) approach using a manually discretized action space where a policy network predicts the next optimal action from among a pre-specified set of actions, as well as with a Proximal Policy Optimization (PPO) approach, where the policy network attempts to return two continous values: the optimal linear and angular velocity for the robot's actuators. The reward function remained the same in both cases, being based on the known distance of the robot from the target as well as detecting collisions and instances where the target was reached.

Keywords: path planning, robot learning, deep reinforcement learning, machine learning, DQN, PPO, agent, real-time reaction

1 Introduction

Path planning (also known as motion planning or the navigation problem) is a computational problem that consists of determining the best route for a robot or autonomous system to travel from a starting point to a destination, while avoiding obstacles and optimizing desired criteria such as distance, time, or amount of energy utilized. This process uses various algorithms and techniques that consider the environment, the robot's abilities, and any given constraints. Path planning has several applications including, but not limited to the area of robotics, such as autonomous driving, urban planning, medical surgery, gaming, animation, simulation, among others.

We will be exploring the navigation problem by developing a mobile robot that must traverse a set of points in a map with obstacles, while avoiding collisions. In particular, we will attempt to apply Deep Reinforcement Learning (RL) algorithms to solve this problem, studying their efficiency and monitoring their

performance in finding the optimal path between the origin of the robot and a given target.[1] This will give us a better grasp of their usefulness in this type of problems and what type of Reinforcement Learning algorithms are better suited for this problem.

To solve this problem, we will be utilizing two Deep Reinforcement Learning algorithms, in particular the Deep Q-Network (DQN) and Proximal Policy Optimization (PPO) algorithms. We will apply these algorithms on a Webots-simulated environment with an 'e-puck' robot, equipped with a LiDAR sensor as its main source of sensory input from the world. We will compare their efficiency in training and their results on the behaviour of the robot.

This paper will firstly cover the details of our environment, as well as the robot agent that we will use and its sensors and actuators in a bit more detail (section 2). In sections 3 and 4, we will cover the algorithms used and elaborate on how they work, and focus on some other key methodological aspects. Furthermore, the experiments used to test our solution will be covered in detail, as well as the results obtained from the performance of our algorithms, before ending on the conclusions we retrieve from these results and suggesting new ways in which to elaborate on this work (section 5).

2 The Environment

In this section, we will delve into the specifics of the environment within which our robot operates. Understanding the setup is crucial for comprehending the robot's interactions and the subsequent results of our experiments.

2.1 Perception

The robot, an 'e-puck' mobile robot, is equipped with various sensors which can be accessed through the Webots Controller module. These sensors provide information about the environment and the agent's location.

We used LiDAR as our main sensor, serving as essentially our world state observation. LiDAR returns a point cloud of 201 points with 3 dimensions each; since our robot only moves in two dimensions, we only consider the X and Y sets of points. We convert these sets into a smaller tensor by dividing all points into 8 subsections and obtain the average of every point in each subsection across each dimension. Finally, we concatenate all points into a single tensor. At each training step, this is the tensor that is fed to our policy network.

The Distance Sensor is used for collision detection by defining, through manual observation, a distance threshold for which we can affirm that the robot is too close to an object and consider that event a collision. These readings are also obtained at every time-step and passed through a boolean function that outputs if a collision happened or not, which returns the most negative reward for the agent if true.

The GPS Sensor is used to determine the location of the robot at a given time-step. It returns (x,y,z) coordinates although only the (x,y) coordinates are considered for this work since there is no movement along the z-axis.

2.2 Actuation

The way the robot interacts with the world is dictated essentially by its movement. It is a small differential wheeled mobile robot, which means it can control its wheels independently, allowing it not only to move forward and backward but also turn at will. We have at our disposal a function to command the robot's velocity which takes as input a the linear and angular velocity of the robot to be applied at each time-step.

Through this configuration, we can define either a continuous action space, where we attempt to find 2 values corresponding to these 2 velocities at each given step or a discrete action space where we define a set of actions which have associated with them predefined velocities, such as "moving forward", where the linear velocity would be maximized and the angular velocity would be null. We will explain the action space more in depth when talking about the learning algorithms used, since each has their own specific action space.

2.3 Reward function

In reinforcement learning, a reward is a signal given to the robot to indicate how good or bad its actions are. The goal of the agent is to maximize the total reward it receives over time.
The reward function used for every experiment in this project is calculated by the sum of the negative of the distances from the agent to the target at each time-step. In addition, a reward of 3 is added to the total if the robot reaches the target, while a reward of -3 is added if the robot collides with an obstacle.
To calculate the distance between the robot and the target, two different distance types were used: Manhattan distance and Euclidean distance.

3 Deep Q-Learning (DQN)

Deep Q-learning is a variation of the Q-learning algorithm. While Q-learning is model-free, in DQN we utilize two neural networks: a policy net which should return the action with the highest expected reward and a target network, used to stabilize the policy net's training. Like all RL algorithms, the idea is that the agent will act based on its perception of the world; in practice, our network will handle as input robot sensor data and produce as output the optimal action for the robot to take, while being optimized based on rewards the agent will receive during the training process.

3.1 Action Space

Due to the formulation of this algorithm, the action spaced used had to be discrete. This is performed by a function which according to the input performs one of 3 possible actions. For a timestep of 0.5 seconds by default, the robot can either move: "forward", moving at 0.1 m/s; "left", rotating $\frac{\pi}{2}$ degrees; "right",

rotating -$\frac{\pi}{2}$ degrees. This way, the robot has full mobility. The policy network, similarly, will have 3 neurons in its output layer, each corresponding to a possible action.

3.2 Replay Memory

As the agent observes the current state of the environment, it selects and applies and action causing the environment to switch to another state, which also corresponds to a reward given by our reward function. We save this transition [state, action, next state, reward], allowing us to reuse this data later in optimizing the network.

3.3 Policy and Target Networks

Our model is a feed forward network consisting of 3 Fully Connected (FC) layers with 2 Rectified Linear Unit (ReLU) activations on the last 2 layers. The number of neurons on the first layer corresponds to the size of our world tensor, or, in practice, twice the number of divisions we select for the LiDAR Point Cloud (because we concatenate all points along the X and Y dimensions). The output layer has a fixed size of 3 neurons, each corresponding to a different action the agent can perform according to our formulation. This is referred to as the Policy Network.

The Target Network is an identical neural network used to compute the expected Q-values V(st+1) resulting in added stability for optimizing the Policy Net. This network is updated with a soft update at each time-step according to a parameter τ.

3.4 Action Selection

At each time-step, our action is chosen according to an Epsilon-Greedy strategy, meaning it is either chosen at random or chosen by the network, if the episode threshold is lower than a randomly drawn sample. The chosen action corresponds to the index of the tensor returned by the network when fed the state tensor which corresponds to the highest value, or, if chosen at random, simply chosen between 0, 1 and 2. Each integer corresponds to a different action as described in the environment.

3.5 Model Optimization

The model optimization is done by batches. The optimization process only begins if the replay memory has enough samples to form a batch according to a predetermined size. Non-terminal next states are concatenated for further use, while current states, actions, and rewards are also concatenated into their respective tensors. The loss is calculated using the Huber loss function which measures the difference between the predicted Q-values, obtained by the target net, and the

expected Q-values, determined by combining immediate reward with discounted future rewards. The weights of the policy network are updated through standard backpropagation while using gradient clipping to prevent issues with exploding gradients.

3.6 Training

We run the training loop for a fixed number of episodes. In each episode, we reset the environment and obtain the initial world state. The episode runs for any number of timesteps until it is done, defined as either truncated (when the agent gets a negative reward because of a collision) or terminated (if the agent gets a positive reward for reaching its target). We select a move according and transition to the next state, saving the transition as described earlier. If the batch of transitions is big enough, the model is optimized.

4 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a popular algorithm in deep reinforcement learning that helps an agent to learn the best actions to take in different situations by adjusting its strategy in a stable and efficient way.[2] The agent interacts with the environment by taking actions according to its current policy - the function that maps the states perceived by the agent's sensors to the best action it can take. After each action is taken, the agent receives a reward and its advantage is calculated. An advantage is a measure of how much better or worse that action is compared to the average action for that state. Then, the policy is updated by adjusting its parameters in order to maximize the expected reward, although this is done carefully to avoid making drastic changes that could destabilize learning. This is achieved by the use of a special function called the "clipped surrogate objective" which controls the size of policy updates. This function limits how much the new policy can differ from the old policy, keeping changes within a safe range.

4.1 Action Space

In contrast with the previously covered DQN, Proximal Policy Optimization is best used with a continuous action space. Instead of being limited to the actions of "forward", "turn right" and "turn left", each timestep is associated with a pair of values that correspond to the robot's linear and angular velocity. Notably, this gives the robot a lot more freedom in the way it maneuvers across the environment, but at the cost of being much more susceptible to noise and thus leading to a slower convergence.

4.2 Policy Network

The policy network is a feed forward network consisting of 3 Fully Connected (FC) layers with 2 Rectified Linear Unit (ReLU) activations on the last 2 layers.

The number of neurons on the first layer is 16, with these being the average of the x coordinate and the average of the y coordinate for each one of the 8 subsections of the LIDAR point cloud. The second layer has 128 neurons while the output layer has 2 neurons, one corresponding to the linear velocity of the robot and the other corresponding to the angular velocity.

4.3 Action Selection

At each time-step, the network returns two normal distributions corresponding to the linear and angular velocities. Each velocity is sampled from its distribution and these are passed to the function that commands the robot's actuators.

4.4 Loss calculation

PPO uses a clipped surrogate objective function to calculate the loss of the model after each set of predictions. The advantage estimator, often using Generalized Advantage Estimation (GAE), computes how much better or worse the taken actions are compared to the average action. The key component in PPO's loss function is the probability ratio between the new policy and the old policy for the actions taken. This ratio is clipped within a small range to prevent significant deviations from the old policy. The clipped surrogate objective function is then defined as the minimum of the unclipped objective and the clipped objective, ensuring that updates do not move the policy too far from the old one, thus maintaining training stability while allowing for effective policy improvement.

4.5 Training

In each training batch, we ran the loop for a fixed number of episodes. Each episode has a fixed maximum number of timesteps but can be stopped earlier if the robot either collides with an obstacle or reaches a target. After each episode, the surrogate loss is calculated and backpropagated through the network.
In the next episode, the normal distributions from which the actions will be selected use the new weights calculated during the backpropagation process.

5 Results Analysis

5.1 Results discussion

We ran these tests taking into account two different measurements: Euclidean distance and Manhattan distance. We expect that the model should show significant results if trained on 500 episodes with 600 timesteps each.

With PPO, we obtained lackluster results that we decided not to display. Due to the noisy training process of a Policy Network which produced continuous values, we failed to see convergence in such a short time frame.

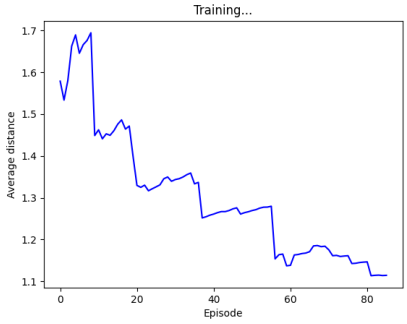

Fig. 1. Average Distance per Episode: Manhattan Distance

As for DQN, convergence appears to be achieved more easily. Our results show a positive increase in reward and episode time as well as a decrease in the distance to the target (perhaps the most important indicator) over time as the number of episodes increased.

The training is noisy due to the fact that the robot's starting position is random, resulting in some episodes where the duration is very high and therefore the cumulative reward is also much less than in others. This impact of this noise should diminish as the number of episodes increases and does not change the overall trends observed.

We observe and hope to see further expression with more training of a lower average distance, meaning the agent is approaching the target; as well as reduced reward and increased episode duration, also indicating that the agent is approaching the target and actively avoiding collisions.

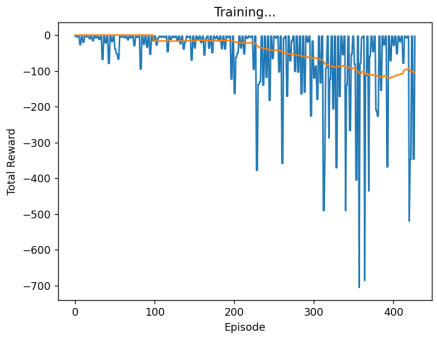

Fig. 2. Reward per Episode: Euclidean Distance

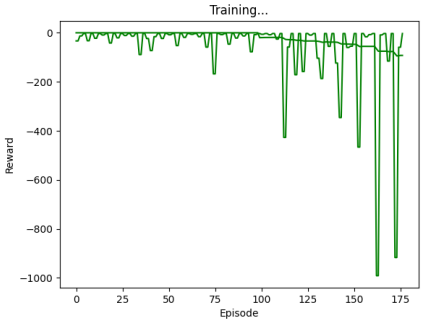

Fig. 3. Reward per Episode: Manhattan Distance

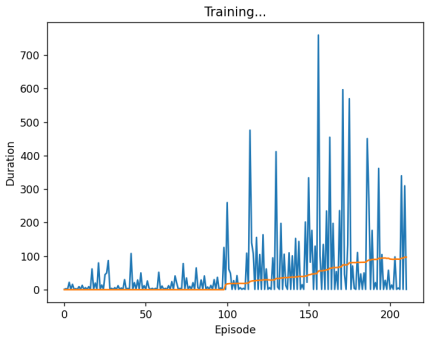

Fig. 4. Duration per Episode: Euclidean Distance

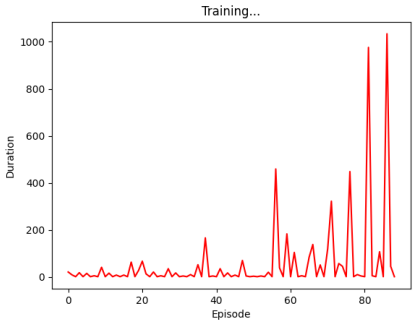

Fig. 5. Duration per Episode: Manhattan Distance

6 Conclusions and Future Work

We successfully demonstrated the feasibility and effectiveness of RL algorithms to navigate and optimize the agent's path in the Webots environment. Our ex-

perimental results showed significant improvements in the robot's performance, particularly in the case of the DQN algorithm, providing a reliable approach in the case of a discrete action space.

Future research could explore improved reward structures, refining the reward function to better reflect the navigation goals as well as investigating the scalability and computational efficiency of our approaches if they were to be used in a pratical deployment scenario.

References

1. Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. *Advances in neural information processing systems*, 27, 2014.
2. John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.