Reinforcement Learning algorithms in the BipedalWalker-v2 Environment

Georgios Kyziridis
LIACS, Leiden University
s2077981@umail.leidenuniv.nl

ABSTRACT

In the last decades, one of the main interests in the field of Artificial Intelligence is robot locomotion. The morphology of humanoid robots is similar to that of humans which provides the ability to traverse complex terrains that are easily accessible to humans. There are various methods and algorithms based on the robot locomotion task. This paper is an endeavor to explore and describe some of the existing approaches. In particular, we explored some Reinforcement Learning algorithms and applied them on the openAI environment called *Bipedal-Walker*. We implemented some of the latest approaches for robot locomotion and explain them in the following sheet. The paper starts with some naive algorithms describing the whole experimentation and some of the Reinforcement Learning preliminaries and gradually moves on to more complex algorithms exploring their differences. Finally, all the results of the experiments are stated, visualizing the performance of the different algorithms.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning

General Terms

Reinforcement Learning, Policy Gradient

Keywords

Actor–Critic, Deep-Q-learning

1. INTRODUCTION

The `BipdelWalker-v2` environment is provided from the openAI organization and it can be loaded in Python as a framework through the `openAI.gym` module. It provides its own functional API, helping users test their AI–algorithms. The specific environment illustrates a bipedal–robot in 2D which tries to walk, so the main goal, is to build an algorithm which trains the robot to walk. The problem is posed as a finite–horizon terrain where the robot has to learn how to choose correct actions in order to stand up and walk until the end. This is basically a Markov–decision–process (MDP) where a suitable action has to be chosen at each step under the current policy.

With the term policy, we refer to the actions that the robot chooses from a set of possible actions $\mathcal{A}$ on each state $s \in \mathcal{S}$. The robot has to be trained through a trial and error procedure in order to learn how to choose correct actions according to the policy. There are various approaches based on the policy learning which are explained in the following sections.

The current paper explains the training procedure and the experimentation set–up of different reinforcement learning algorithms. The findings of the current research lead to a general conclusion about the performance of these different algorithms. According to the results, only few algorithms seem to be capable of training the robot efficiently up to the far end.

2. GAME–ENVIRONMENT

The environment generates observations according to the corresponding actions that robot exerts. More precisely, at each time–stamp $t \in T$, in state $s_t \in \mathcal{S}$, the robot operates a specific action $a_t \in \mathcal{A}$, derived from the environment's action–space, $\mathcal{A} = [-1, 1]^4$. Subsequently the environment produces observations such as reward $R(s_t) \in \mathbb{R}$ and new state $s_{t+1} \in \mathcal{S}$. All states and rewards are generated from the environment after each action. Rewards define how good was the operated action a_t at the time–stamp t, from current state to the new one, $s_t \rightarrow s_{t+1}$. Intuitively, the purpose of our algorithm is to force the robot to select actions that maximize rewards. All finite–MDP have at least one optimal policy (which can give the maximum reward) and among all the optimal policies at least one is stationary and deterministic.

A state observation $s_t \in \mathcal{S}$ is represented by a 24–length vector in continuous space. It contains information about the position, the angle and the speed of the robot at every state s_t, for t: $1, 2, \ldots, T$. The ten trailing numbers in the state vector, s, reflect the lidar–observation. The robot has its own mechanism of eyesight using lidar–vision. This feature is more useful in the hardcore version of the game `BipdealWalker-v2-Hardcore` which is the same environment but with some extra obstacles and pitfalls that robot has to come out.

Actions $a_t \in \mathcal{A}$, are represented by a 4-length vector which indicates the speed and the angle of the four robot joints. Each number in the action vector a_t, is a continuous value in the

Research Project '18 Computer Science & Advanced Data Analytics, Master CS, Leiden University (https://liacs.leidenuniv.nl/), supervised by Walter Kosters and Wojtek Kowalczyk

range $[-1, 1]$ which represents both the speed and the angle of the joints. As stated above, in each state s_t, the environment takes as input an action vector a_t and immediately generates the current reward $R(s_t)$, and the new state s_{t+1} as well. The environment also relocates the robot from the old state s_t to the new position, s_{t+1}. The `OpenAI` environment provides a real time visualization of the game. Figure 1 illustrates a random position of the robot monitored directly from the rendering environment.

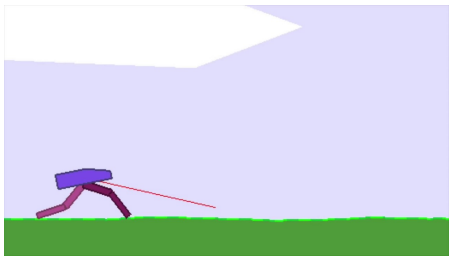


Figure 1: BipedalWalker screenshot

Figure 2 and Algorithm 1, describe the environment interaction. It is just an example of how we can provide an action as input and immediately get the new observation directly from the environment. However, before that, it is crucial to formulate the aforementioned vectors, *action* and *state*, in mathematical terms.

$$State = \langle \mathtt{h_0}, \frac{\partial \mathtt{h_0}}{\partial \mathtt{t}}, \frac{\partial \mathtt{x}}{\partial \mathtt{t}}, \frac{\partial \mathtt{y}}{\partial \mathtt{t}}, \mathtt{h_1}, \frac{\partial \mathtt{h_1}}{\partial \mathtt{t}}, \mathtt{k_1}, \frac{\partial \mathtt{k_1}}{\partial \mathtt{t}}, \mathtt{h_2}, \frac{\partial \mathtt{h_2}}{\partial \mathtt{t}}, \mathtt{k_2}, \frac{\partial \mathtt{k_2}}{\partial \mathtt{t}}, \iota_{leg1}, \iota_{leg2}, \mathtt{lidar} \rangle$$

$$Action = \langle \mathtt{H_1}, \mathtt{K_1}, \mathtt{H_2}, \mathtt{K_2} \rangle$$

Regarding the $State$ vector, $\mathtt{h_0}$ represents the hull angle and $\mathtt{h_1}, \mathtt{h_2}$, $\mathtt{k_1}, \mathtt{k_2}$ are the hips and knees angle respectively. Furthermore, $\mathtt{lidar} \in \mathbb{R}^{10}$ reflects the lidar readings that robot observes at each time stamp t_i, represented by a 10-length vector. In the $action$ vector, $\mathtt{H_1}, \mathtt{H_2}$ and $\mathtt{K_1}, \mathtt{K_2}$ indicate accordingly the hips and knees torque, and speed as well. The above information is provided by the `openAI-wiki` github repository.

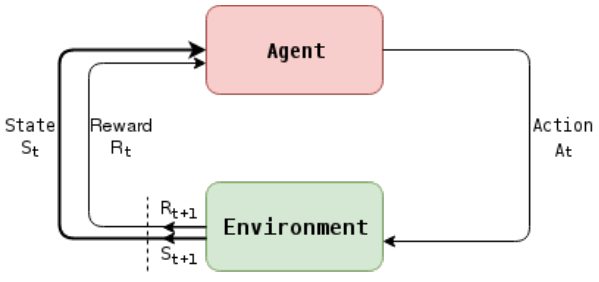


Figure 2: Agent-Environment feedback loop
Image inspired by Sanyam Kapoor blog

Algorithm 1 Environment Interaction Example

procedure ENVIRONMENT
  Load the environment
  env = **OpenaiMake**('BipedalWalker-v2')
  Reset get initial State S_t
  state = **ResetEnvironment**()
  Generate random Action A_t
  action = **Random**$(x_{min} = -1, x_{max} = 1, size = 4)$
  Interact: feed Action A_t, get new **observation**
  state_new, reward = **envStep**(action)
end procedure

3. Q-LEARNING ALGORITHM

Q–Learning [6] is a value-based reinforcement learning algorithm which is used to find the optimal action–selection policy using a Q function. In the following sections the algorithm is explained thoroughly. The reason we use this algorithm, is to approximate the reward value for the chosen predicted action.

3.1 Bellman Equation

The Q–learning algorithm learns a policy, which tells to an agent what action to take under what circumstances. It does not require a model of the environment and can handle problems with stochastic transitions and rewards. For any finite Markov decision process, Q–learning finds a policy that is optimal in the sense that it maximizes the expected value of the total reward over all successive steps, starting from the current state. So when the robot is located at a current position s_t, it chooses an action a_t, and moves to the next state $s_{t+1} \in \mathcal{S}$. At the new state s_{t+1}, the robot must select the next action which produces the highest reward–value and iteratively update the following equation:

$$Q_{new} \leftarrow (1 - \alpha) \cdot R + \alpha \cdot \gamma \cdot \max Q(\hat{a}, s \rightarrow s') \qquad (1)$$

The above equation is the well known $Bellman$–$equation$ used in Q–Learning, where $R(s_t) \in \mathbb{R}$ defines the reward gained from the current state s and an action a. Parameter $\gamma \in [0, 1]$ defines the decay rate, a constant parameter ranging from 0 to 1 and usually set to a value close to 1, e.g., $\gamma = 0.99$. Its role is to manipulate the way that agent considers future rewards. If γ is closer to 0, the agent will tend to consider more immediate rewards while if it is closer to 1, the agent will consider future rewards with greater weight. Parameter α, indicates the equation's learning rate, also known as the Q–$learning\ rate$, and it is usually set to a very small value, e.g., $\alpha = 10^{-4}$. It controls the amount of speed that the agent forgets previous states and how fast it learns from new experience. We consider Q_{new}, as the target value of our training model, y_{target}. The reader will note that our goal is to find the optimal policy of predicting actions $\hat{a}$, that maximize the current and future rewards. This will be achieved through the above procedure using a model which predicts $\max Q(\hat{a}, s')$ and is trained using the $Bellman$–$equation$ outcome. Algorithm 2 describes the above procedure from [6].

4. NAIVE APPROACH

This section explains the implementation of different neural-network models. It refers to naive models based on a heuristic derived from the Q–learning idea. The aim of these networks

Algorithm 2 Q-Learning Rule

procedure Q-LEARNING
  Initialize $Q(a, s)$ arbitrarily
  for each episode **do**
    Reset environment & initialize s
    for each step in $episode$ until s is terminal **do**
      Choose a at s using policy obtained from Q
      $a \leftarrow policy(s, Q)$
      Operate a, observe r, s'
      Update $Bellman–equation$:
      $Q(a, s) \leftarrow Q(a, s) + (1-\alpha) \cdot r + \alpha \cdot \gamma \max Q(a', s')$
      Update state s: $s \leftarrow s'$
    end for
  end for
end procedure

is to predict the action $\hat{a} \in \mathcal{A}$, by understanding the data structure and produce actions according to the current policy. Note that the model has to predict actions that maximize future rewards, $R(s_t)$, and force the robot to walk. At each time–stamp, $t \in T$, the robot has two choices: either select a random action or select the predicted action generated from the model. In this way, we give the robot the opportunity to explore its environment by making completely random actions and collect significant observations which are saved in experience–replay–memory. The chance of selecting a random action is based on the $\epsilon–greedy$ algorithm which manipulates the balance between random choices and the predicted ones. This algorithm will be described in the following sections.

4.1 First Model

The first model consists of a Deep–$\mathcal{Q}$–Learning neural network with two hidden layers. Specifically, the first hidden layer contains 49 nodes with a Relu activation function while the second contains 25 hidden nodes with the tanh activation function. The choice of tanh activation function in the last hidden layer is due to the environment's action domain $\mathcal{A} = [-1, 1]^4$. Therefore, the output of the network is a 4–length vector which represents a single action of the robot, $\hat{a}$. The model takes as input the concatenation of the current and the previous state observation $s_t, s_{t+1} \in \mathcal{S}$, and predicts the next action, $\hat{a} \in \mathcal{A}$. Figure 3 visualizes the network architecture.

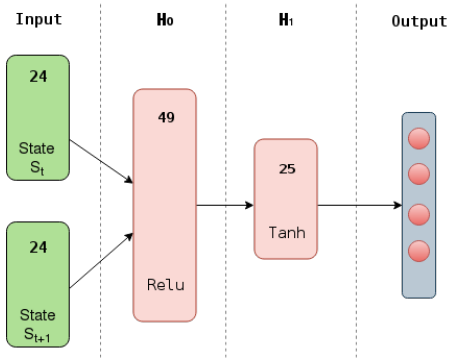


Figure 3: First naive network

4.2 Second Model

The second model is derived from the previous network. It is a softly upgraded version of the first model regarding the network's input. The intuition behind this model is, that in matter of robot–locomotion, the network should include some elements of the recurrency concept. More precisely, the idea was to upgrade the network of the previous section, by plugging in the output of the last layer as an extra input and consequently force it to capture the structure of locomotion data more efficiently. In this way, we formulate a recurrent network which returns the last–layer's outcome as an input together with the concatenation of the state vectors, s_t and s_{t+1}. Figure 4 visualizes the aforementioned network which is based on the first model.

The differences between the first and the second model, arise from the architecture differentiation. Namely, the addition of the predicted action as input, and the increased number of hidden nodes in the hidden layer as well. Both networks use the same loss–function, optimizer and same update rule, too. The training process of the two previous networks is described in the following sections.

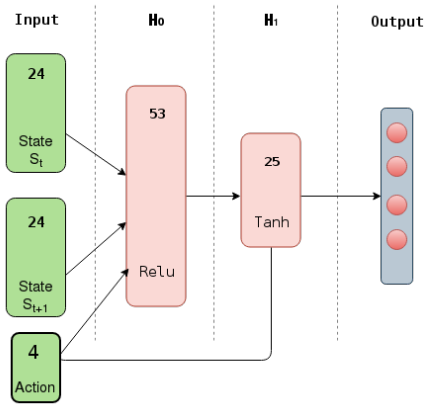


Figure 4: Upgraded naive network

4.3 Q–Learning Heuristic

According to the description from Section 3, the goal of our algorithm is to predict actions that maximize future rewards following the current policy. So we need a model which predicts an action, $\hat{a}_t$, for each state $s_t, t = 1, 2, \ldots, T$, and a function that calculates the approximated reward for the predicted action. We will use a neural network model for predicting the action, $\hat{a}$ at each state s. The training process of the network, using classical Q–learning, requires a function to criticize the predicted action and produce the approximated reward value, $Q(\hat{a}_{t+1}, s_{t+1})$. This function is usually another neural network as we will see in the next sections, but for the time being, we use the following heuristic: instead of calculating the approximated $Q(\hat{a}, s_{t+1})$ value, we transform the discounted reward to a vector (just by replicating it) and add it to the predicted action, $\hat{a}$. In this way we produce a vector $\vec{Q}_{new}$ whose values represent the sum of the action vector $\hat{a}$ and the discounted reward vector $(1 - \alpha)R(a, s)$. The output vector $\vec{Q}_{new}$ indicates the target value y_{target} of the neural network. Thus, we examine the intuition that if we set

network's target to be the sum of the predicted action and the reward, we can force the network to predict actions that leads to higher rewards. Algorithm 3 describes the above heuristic.

Algorithm 3 Heuristic

procedure Q-Learning Heuristic
  Run episode collect observations
  Observations: $\langle state, action, reward, state', flag \rangle$
  Save observations as $\langle s, a, r, s', f \rangle$ in memory
  After episode ends get random.sample from memeory
  for each $\langle s, a, r, s', f \rangle$ **in** random.sample **do**
    if f **then**
      $Q_{new} \leftarrow (r, r, r, r)$
    else
      $Q_{new} \leftarrow (1-\alpha) \cdot (r, r, r, r) + \alpha \cdot \gamma \cdot NN(s, s')$
    end if
    NetworkTrain NN(input = states, output = Q_{new})
  end for
end procedure

4.4 Loss Function

This section explains the error function of the neural network. It is the standard mean squared error function and is basically used for regression problems since it measures the averaged squared distance between target and predicted value. Regarding the *target* part of Equation (4), it could be translated as a sum of $(1-\alpha) \cdot 100\%$ of the current state's reward $R(s_t)$, and $\alpha \cdot 100\%$ of the discounted estimated value, $Q(\hat{a}, s')$. Therefore, in the *prediction* part, $\hat{Q}(\hat{a}, s)$ represents the network's outcome. Combining all the above, the aim is to reduce the deviation between the predicted value from the model and the target value derived from the $Bellman–equation$. In this way, we are confident that the outcome of the network leads to an optimal sequence of actions by training the weights into this direction. So we have:

$$loss = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{2}$$

$$= \frac{1}{n}\sum_{i=1}^{n}(Q_{new}(a,s)_i - \hat{Q}(\hat{a},s)_i)^2 \tag{3}$$

$$= \frac{1}{n}\sum_{i=1}^{n}[\underbrace{(1-\alpha)\cdot R_i + \alpha \cdot \gamma \cdot Q(\hat{a}, s')_i}_{target} - \underbrace{\hat{Q}(\hat{a},s)_i}_{prediction}]^2 \tag{4}$$

Where y_i and $\hat{y}_i$ indicate the target value obtained from the $Bellman–equation$ and the predicted one, respectively.

4.5 Epsilon-Greedy

The exploration and exploitation ratio is a great challenge in reinforcement learning which has a strong impact on the quality of results. On the one hand, exploration mode prevents from maximizing short-term rewards due to the fact that many random actions will yield low rewards. On the other hand, exploration provides the opportunity to the robot to collect data from the environment by exerting random actions. In exploitation mode, the robot uses the collected data from the exploration phase and predicts an action based on the policy. Exploiting uncertain environment data, prevents the maximization of the long–term rewards because the selected actions are probably not optimal. Consequently, we need balance between the two modes (exploration & exploitation) in order to use them adequately. All the above are summarized in the well known *Dilemma of exploration and exploitation* [6].

Considering the current reinforcement learning problem, at each state $s_t \in \mathcal{S}, t: 1, 2, \ldots, n$, the robot exerts an action $\hat{a}_t$. That action is chosen either completely at random, or obtained from the current policy π. The referent ϵ-greedy algorithm, handles the trade–off between exploration and exploitation mode by controlling the parameter of randomness. Let $\epsilon \in [0, 1]$ be the probability that the robot selects an action completely at random and not based on the current policy π. The idea of ϵ–greedy is to control the frequency of selecting a random action, by reducing this probability. The following equation describes the binary decision between random and non–random action.

$$\pi(s) = \begin{cases} \text{random action from} \mathcal{A}(s), & \text{if } \xi < \epsilon \\ \text{argmax}_{a \in \mathcal{A}(s)} \mathcal{Q}(s,a), & \text{otherwise} \end{cases} \tag{5}$$

where ξ is a uniform random number drawn at each time step, t. All in all, the above equation is described thoroughly in [6, 9]. The goal is to handle the balance between the two modes, providing the opportunity to choose if we need more exploration or exploitation. Each time the robot selects a random action, the ϵ value is decreased by a discount factor $\delta \in [0, 1]$ which is set to a value close to 1, e.g., 0.999. In this way, we slightly decrease the chance of a random action selection according to our requirements. Ideally, exploration mode is preferable in the early stages when the robot is completely uncertain about the action selection due to its zero experience. After the robot collects observations from the environment during the exploration phase, it will exploit its knowledge and select actions derived from the policy. Figure 5 illustrates the value of ϵ for 100000 consecutive episodes with two different discount factors $\delta_1 = 0.999, \delta_2 = 0.9999$, showing the significant impact of parameter δ. In each experiment the ϵ value is initialized to 1.

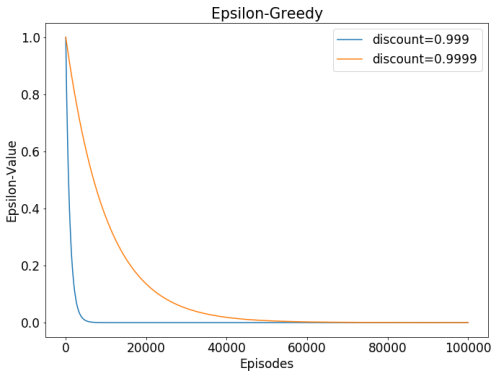


Figure 5: ϵ-greedy example

4.6 Training

Taking into account the intuition and the heuristic from Section 4.3, we trained the naive models in such a way that the loss function measures the average squared distance between

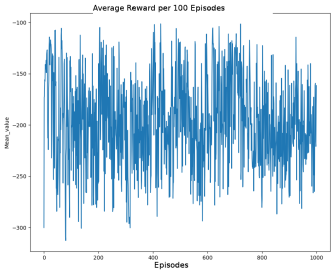
the target vector and predicted vector. More precisely, the *target* part of Equation (4) is represented by a vector $\vec{Q}_{new}$, which is the sum of the discounted reward (as vector) and the predicted action for the new state, $\hat{a}$. The following equation states the loss function following our intuition:

$$loss = \frac{1}{n}\sum_{i=1}^{n}[\underbrace{(1-\alpha)\cdot\vec{r}_i + \alpha\cdot\gamma\cdot\hat{\vec{a}}}_{target} - \underbrace{\hat{\vec{a}}_i}_{prediction}]^2 \quad (6)$$

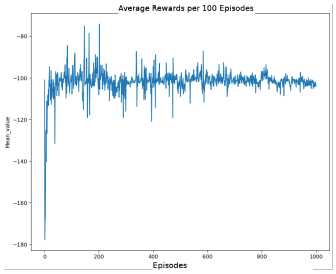
where $\hat{\vec{a}}$, is the neural network outcome using as input the current pair of observation–states s_t, s_{t+1}. The training procedure starts after the end of each episode inside a mini–batch routine where a random sample of observations is chosen from the experience–replay memory [1]. The network is trained for each one of the observation–tuples $\langle s, a, r, s', f\rangle$, using the aforementioned heuristic.

4.7 Naive Models Results

The plots below illustrate the outcome of both networks by visualizing the average reward per 100 episodes. It is obvious that the above naive models are not capable of capturing the structure of data correctly and they are not producing the desirable outcome. It can be easily observed that the first model (left hand side), reaches the maximum average of -100 reward. The second model produced better outcome but it is not reaching the desirable winning reward of +300 as well. In conclusion, it seems that both naive models are not sufficient to produce robust results. So, probably we need more advanced methods and network architectures for this specific task. Moreover, it seems that our intuition and the corresponding heuristic from Algorithm 3 did not meet the success point of this task. The reader could consider all the above, as preliminary information based on simplified methods and naive heuristics.



(a) Average rewards Model 1



(b) Average rewards Model 2

5. ACTOR–CRITIC LEARNING

This section is related to some specialized neural network architecture called Actor–Critic. This approach is used to represent the policy function independently of the value function. Before moving on a thorough explanation of Actor–Critic algorithms, we briefly mention some basic preliminaries from [6].

Reinforcement Learning Objective: *Maximize the expected reward following a parametrized policy* π.

$$J(\theta) = \mathbb{E}_\pi[r(\tau)] \quad (7)$$

where r indicates the reward for the trajectory τ.

5.1 Policy Gradient

The Policy–Gradient (PG) algorithm [7], optimizes a policy π, by computing estimates of the gradient of the expected reward for the current predicted action, $\hat{a}$, following the current policy. Then, it updates the policy in the gradient direction. Ideally, the algorithm visits various training examples of high rewards from good actions and negative rewards from bad actions.

Policy Gradient Theorem: *The derivative of the expected reward is the expectation of the product of the reward and gradient of the log of the policy* π.

$$\nabla\,\mathbb{E}_\pi[r(\tau)] = \mathbb{E}_\pi\Big[r(\tau)\cdot\nabla\log\pi(\tau)\Big] \quad (8)$$

Let us extend this policy term π in a mathematical way since it is essential for our objective. The following equations brake down Equation (8) and prove the above theorem.

$$\pi(\tau) = P(s_0)\prod_{t=1}^{T}\pi(a_t|s_t)\rho(s_{t+1}, r_{t+1}|s_t, a_t) \quad (9)$$

$$\log\pi(\tau) = \log P(s_0) + \sum_{t=1}^{T}\log\pi(a_t|s_t) + \sum_{t=1}^{T}\log\rho(s_{t+1}, r_{t+1}|s_t, a_t) \quad (10)$$

$$\nabla\log\pi(\tau) = \sum_{t=1}^{T}\nabla\log\pi(a_t|s_t) \quad (11)$$

$$\nabla\,\mathbb{E}_\pi[r(\tau)] = \mathbb{E}_\pi\left[r(\tau)\cdot\left(\sum_{t=1}^{T}\nabla\log\pi(a_t|s_t)\right)\right] \quad (12)$$

The above equation is the classical *Policy Gradient* where value τ indicates the trajectory, P the ergodic distribution of starting randomly at a state s_0, and ρ expresses the dynamics of the environment. The result of the above equation shows that the ergodic distribution of the states P, and environment dynamics ρ, are not really needed in order to predict the expected reward. The above procedure is known as a *Model–Free Algorithm* due to the fact that we do not actually model the environment.

One term that remains untouched in our treatment above is the reward of the trajectory $r(\tau)$. However, we can make use of a discount function G which returns the discounted reward G_t. Hence, if we replace the element $r(\tau)$ with the discounted result, G_t, we can arrive at the classic algorithm called *Reinforce* which is based on *Policy Gradient*. Then Equation (12) change to:

$$\nabla\,\mathbb{E}_\pi[r(\tau)] = \mathbb{E}_\pi\left[G_t\cdot\left(\sum_{t=1}^{T}\nabla\log\pi(a_t|s_t)\right)\right] \quad (13)$$

One way to realize the problem is to re–imagine the RL objective defined above, as a Maximum Likelihood Estimate problem. In `MLE` setting we follow the intuition that, no matter how bad the initial estimates are, according to data, the model will converge to the true parameters. However, in a setting where the data samples are characterized by high variance, stabilizing model parameters can be notoriously hard. In our

context, any trajectory can cause a sub–optimal shift in the policy distribution.

5.2 Deterministic Policy Gradient

In the robotics field, the control policy is the main objective due to the fact that in such environments it is hard to build a stochastic policy. The proposed algorithm learns directly a deterministic action for a given state instead of learning probability distributions. We refer to deterministic actions according to the equation below where μ indicates the action with the highest Q–value for a current state s:

$$\mu^{k+1}(s) = \underset{a}{\operatorname{argmax}}\ Q^{\mu^k}(s,a) \tag{14}$$

However, the maximization is quite infeasible so there is no other way than to search the entire vector space for a given action–value function. So, we can build a function which approximates the argmax value and train it in order to produce robust approximations, imitating the rewards from the environment. More precisely, our objective is to generate the approximated reward–value for the given predicted action and use this value in the training process based on the $Bellman$–$equation$ from Section 3. The Deterministic Policy Gradient [4] is briefly described in the following equations.

$$J(\theta) = \mathbb{E}_{s\sim p^\theta}\Big[r\Big(s,\mu_\theta(s)\Big)\Big] \tag{15}$$

$$\nabla J(\theta) = \mathbb{E}_{s\sim p^\theta}\Big[\nabla_\theta \mu_\theta(s) \nabla_a Q^{\mu\theta}(s,a)\Big|_{a=\mu_\theta(s)}\Big] \tag{16}$$

Equation (15) describes the objective and Equation (16) illustrates the deterministic policy gradient theorem. All in all, the above equations summarize the idea of DPG where μ_θ: S $\rightarrow \mathcal{A}$ and parameter vector $\theta \in \mathbb{R}^n$. The performance objective is defined by $J(\theta)$ and p indicates the probability distribution $p(s \rightarrow s', t, \mu)$.

5.3 Actor–Critic Architecture

The Actor–Critic learning algorithm from [2, 8], represents the policy function independently of the value function. The policy function structure regards the actor, and the value function the critic. As Actor–Critic, we consider two models where the actor produces an action given the current state and the critic produces an error signal for the taken action. The critic is estimating the action–value $Q(s,a)$, using the actor's outcome. The output of the critic leads the learning process for both models. More precisely, in this task, the actor model will produce the action $\hat{a} \in \mathcal{A}$, and the critic will produce the approximated value given that action, $Q(s,\hat{a})$. In this way, the critic judges the corresponding predicted action by calculating the approximated future reward. In Deep Reinforcement Learning, neural networks can be used to represent the actor and critic structures.

Figure 7 illustrates a simple Actor–Critic network architecture. It can be observed that the actor network gets as input the current state vector, $s_t \in \mathcal{S}$, and generates the predicted action, $\hat{a} \in \mathcal{A}$ as outcome. Consequently, the critic network takes as input the state s_t, and the actor's output $\hat{a}$ and its goal is to capture the structure of the data. That is achieved by feeding the input into different layers of hidden neurons. The output of both layers is merged into the final hidden layer which ends up to a single value through a linear

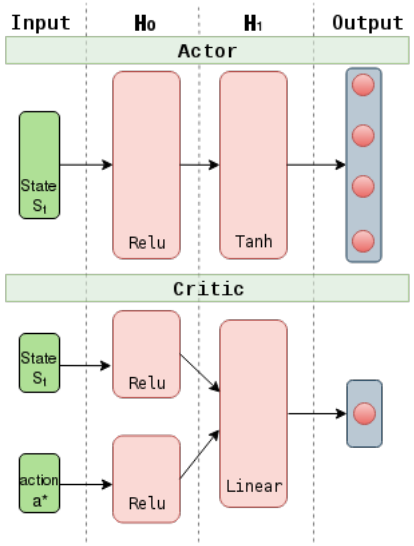


Figure 7: Actor-Critic architecture

activation function. This value represents the corresponding $\hat{Q}(s,\hat{a})$ value as an approximation of the corresponding reward for the predicted action $\hat{a}$.

The general idea of the above approach is to represent the action-value Q which criticizes the action in the current state. So, consider a player who asks for feedback for each action that it chooses, that feedback is generated from the critic network which takes as input the state s_t and the predicted action $\hat{a}_t$. The feedback is just a potential reward which punishes the bad predicted actions or praises the good ones. The reader could consider Actor–Critic networks as two friends, one of them plays the game (actor) and asks the other (critic) for feedback. Then, learning from that feedback, the actor will update its policy towards the direction provided from the critic, which also updates its way to provide more accurate feedback. Both networks are trained in parallel and learn from the game experience.

$$Actor\colon \pi(s,a,\theta)$$
$$Critic\colon \hat{Q}(s,a,w)$$

Having two models (Actor and Critic), implies that we have two set of weights (θ for our Actor and w for our Critic) that must be updated and optimized separately as it is described in the following equations:

$$\mathcal{P}\colon \theta = \alpha\nabla_\theta\Big(\log\pi_\theta(s,a)\Big)\hat{Q}_w(s,a)$$
$$\mathcal{V}\colon w = \beta\Big(\underbrace{R+\gamma\hat{Q}_w(s_{t+1},a_{t+1})-\hat{Q}_w(s_t,a_t)}_{TD}\Big)\underbrace{\nabla_w\hat{Q}_w(s_t,a_t)}_{\nabla Value\ Function}$$

Here α, β represent the learning rate of the two networks, respectively, π_θ the current policy and $\hat{Q}_w(s,a)$ the value function approximation for action $\hat{a}$ and state s. Temporal Difference (TD) measures the difference between the estimation of the $\hat{Q}$-value at the states s_t and s_{t+1}. The specific formula of TD used in the above equation, is called TD(0) and it is a variant of the general algorithm called TD(λ) [6].

5.4 Actor–Critic Training

The training process of the networks starts when a specific number of trajectories is collected from the running simulation in the experience–replay buffer. The critic network takes as input the states and the corresponding operated actions which were resulted from the actor network. The critic, using the $\mathcal{Q}$–learning rule from Algorithm 2, tries to optimize its predictions according to the direction of the gradients of the mean square error loss function. Basically, it tries to predict the future reward given the state s_t and the action a_t. This prediction aims to approximate the rewards R provided by the game–environment. In that way, the critic network judges the actor's choices by punishing bad policies with a low Q–value and rewards good policies with a higher one. In the classical Actor–Critic algorithm, actor network takes as input only the state and predicts the action according to the current policy π_θ. In order to optimize the policy, actor uses the critic's gradients and optimizes its loss function according to the direction of the critic gradients. More precisely, we have two models from each network (Actor & Critic), local and target. The local actor model is capable of predicting the action $\hat{a}_t$ and the local critic predicts the Q–value for the current state s_t. Target models are used for the calculation of the new action $\hat{a}_{t+1}$ and Q–value, $\hat{Q}(s_{t+1}, \hat{a}_{t+1})$, in the training process. Afterwards, local and target models exchange their weights iteratively using a small learning rate τ, according to the following formula:

$$\begin{aligned} \textit{Actor}&: W_{target} \leftarrow \tau \cdot W_{local} + (1-\tau) \cdot W_{target} \\ \textit{Critic}&: W_{target} \leftarrow \tau \cdot W_{local} + (1-\tau) \cdot W_{target} \end{aligned}$$

where W indicates the network weights and τ: $0 < \tau < 1$, the learning rate of the weights update.

5.5 Experiments

For the specific task of robot locomotion we have performed various experiments using different kinds of neural network architectures. There are several approaches for the same task varying from deep convolutional neural networks to long–short term memory. However, the first thing we wanted to experiment with, is the loss function.

5.5.1 Loss Function

In the initial experiment setup the mean square error loss function was used in both networks. So, both actor and critic were optimizing upon the MSE. After further exploration we decided to experiment with a loss function derived from the Proximal Policy Optimization (*PPO*) algorithm [3]. PPO is a recently developed algorithm which is not covered thoroughly in this paper except from some basic information regarding its loss function. In that way we are combining the two approaches *DDPG & PPO*. This loss function is based on the idea to limit, as much as is possible, the changes of the policy, π_θ, in each iteration through the training simulation. More precisely, we want to reduce the variations between the current policy and the new one in order to bring stability and smoothness to our policy optimization. For that reason, we experimented with the following equation described in [3].

$$\begin{aligned} \textit{new}: &\quad \pi_\theta(a_t|s_t) \\ \textit{old}: &\quad \pi_{\theta\,old}(a_t|s_t) \\ \textit{ratio}: &\quad \tilde{r} = \frac{\pi_\theta}{\pi_{\theta\,old}} \end{aligned}$$

$$\mathcal{L}_\theta(\theta) = \mathbb{E}_{t\sim\pi_\theta}\left[\sum_{t=0}^{T} \min\left(r_t(\theta)A, \text{clip}(r_t, 1-\epsilon, 1+\epsilon)A\right)\right] \tag{17}$$

Equation (17) describes the loss function of the actor neural network, where r_t indicates the ratio between the new and the old policy and A the advantage which is just the difference between the actual reward R and the predicted Q–value $\hat{Q}$, A: $A = R(s_t) - Q(a_t, s_t)$. The ϵ value indicates a very small number, e.g., 10^{-6}, which is used for the clipping. More precisely, we want to manipulate our policy to make soft changes through time and for that reason we use the clipping function with a very small value ϵ in order to set bounds $[1-\epsilon, 1+\epsilon]$ for the policy.

5.5.2 Network Architectures

We experimented with various network architectures and configurations since small differences of the number of nodes or hidden layers impacted the results significantly. The initial model consists of two simple networks (actor & critic) with only one hidden layer each, and 64 hidden nodes. The experiments' results shown that very deep networks with a big number of nodes are not functional, probably because of the gradient vanishing phenomenon. On the other hand, very small and simple networks are not capable to capture the data structure and in many simulations the gradients were stuck into local minima. The final configuration of the networks consists of two hidden layers with 400 and 300 nodes for actor and critic, respectively. The input of the actor network is the lidar observation and the state mapped onto different hidden dense layers. Then the concatenation of the layers' output is the input of the next hidden layer which ends up to the output layer producing the action. The critic network takes as input the state and the lidar observation having the same architecture as the actor, plus the predicted action from actor network. The concatenation of these three inputs is plugged to the second hidden layer which leads to the output layer, which is one single value representing the $\hat{Q}(s, a)$. During the experimentation we tried to formulate an idea derived from [5] and use an LSTM hidden layer before the output. Long short–term memory neural networks, are trained in such a way that understand which information is useful to remember and which to forget using the well known *forget–gate* in the LSTM cell. They are basically used in time–series tasks and they tend to produce promising results. So, in that way, using the LSTM cell in both networks, the Actor–Critic model will consider the input as time–series. Consequently, the critic network will judge actor's actions with better knowledge of the previous replay history and the actor network will produce actions based on a better memory management. This improvement, benefits the whole training procedure of the robot.

5.5.3 Training Batch

The training process of the networks starts after the specified number of collected trajectories reaches the batch size. There are various methods for training on batches. In the current approach the networks are trained on a specific number of trajectories collected from the replay–memory. This number has a very significant impact on the training procedure regarding the speed and the quality of the results. It has been observed, that small number of batch size, e.g., 256, does not

lead to efficient results, probably due to fact that the networks are trained frequently using small amounts of information. On the other hand, bigger batches e.g.,1024 help the networks to produce better results because they use bigger amounts of trajectories. That fact slows the speed of the learning procedure but enables the networks to exploit the variance of the different trajectories collected through the episodes, which leads to more robust results. In some of the experiments a threshold k was set. So, when the total reward of an episode was higher than that threshold, $(R > k)$, the networks were trained only on this episode. This implementation has impact on the results because it forces the networks to be trained only on "good" trajectories, leading to higher results.

6. RESULTS

Figure 8 visualizes the outcome of the Proximal Policy Optimization algorithm [3]. These two simulations produced the highest results among all the experiments. The Boosted PPO algorithm, is a variant of the original PPO and its differentiation is that it uses synthetic data simulating the original data of the environment. It uses these data to train the old and current policy which are plugged into the master model. The master network is trained using Equation (17). It can be observed, that Boosted PPO produced better results than the original PPO algorithm. None of them was able to train the robot to reach the number of 300 total episode–reward in 10,000 consecutive episodes. However, both algorithms seem to teach the robot to take actions towards the direction of maximizing the rewards.

The maximum number of average rewards for Boosted PPO and the simple one is 252.155 and 144.60, respectively. The following plots illustrate the rewards over the whole simulation for both algorithms. In Figure 9 we can see that after almost 4000 episodes, the algorithm starts to produce very high rewards close to 300. The highest episode reward achieved using this algorithm was 298.89. In Figure 10, it can be easily observed that the total episode rewards were not close to 300, the highest reward was 215.49. Both algorithms seem to work efficiently and produced better results than the other algorithms of the experimentation.

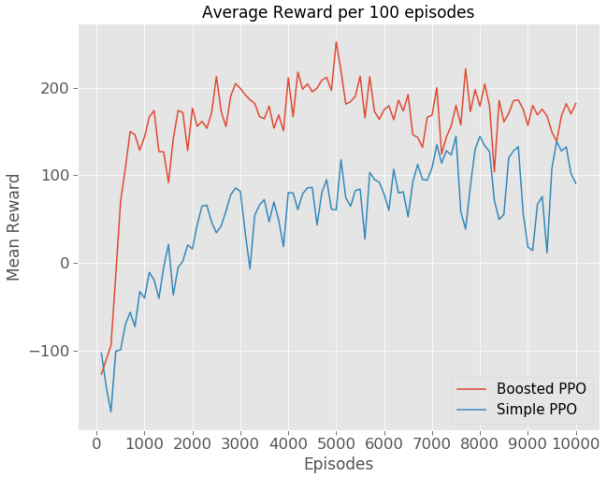


Figure 8: Average reward per 100 episodes

Figures 11 & 12 visualize the results of the Deterministic Policy Gradient simulations. After many different experiments with different configurations we ended up with the best results in terms of the average reward per 100 episodes. Figure 11, illustrates three different variants of the Deterministic Policy Gradient algorithm. The first one is a combination of the DDPG and PPO algorithms which provide the best results among all the experiments of DDPG. It is basically the standard Deterministic Policy Gradient algorithm but with loss function from the Proximal Policy Optimization algorithm described in Subsection 5.5.1. The second curve in the plot below, illustrates the results of the Recurrent Deterministic Policy Gradient which is another variant of the DDPG algorithm using LSTM layers for the Actor–Critic networks. The recurrent networks have the ability to learn better the data structures using old information from the game history. We tried to simulate the approach from [5], but without using the parallel agents due to limited resources. The final curve in the graph below represents the simple version of the Deterministic Policy Gradient method, which had the weakest performance in this simulation. However, all three algorithms seem to did train towards a good direction for the robot, because according to the plot, their results seem to have an ascending tendency. This means that probably for many more episodes and iterations they will be able to reach the final goal which is 300 average reward in 100 consecutive episodes.

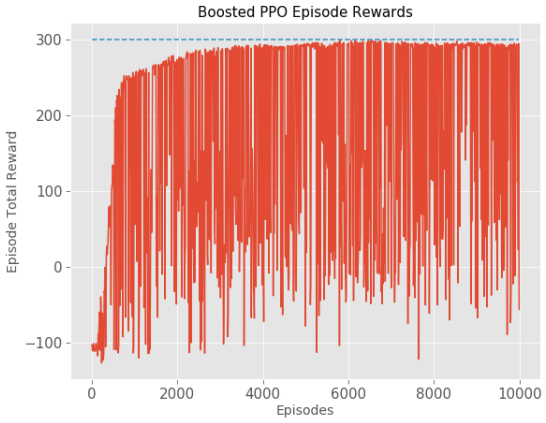


Figure 9: Boosted PPO total rewards

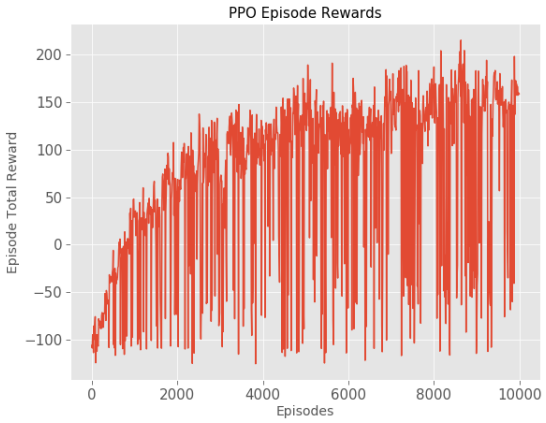


Figure 10: Simple PPO total rewards

Figure 12 illustrates two other experiments of the Deterministic Policy Gradient algorithm using different network set–up for Actor & Critic. The first line of the plot visualizes the *Big–DDPG* which consists of deep neural–networks with convolution layers for the lidar readings. It includes a big number of hidden nodes, e.g., 800, and apart from the convolution, their peculiarity is that they are bigger and deeper than the rest of the networks. The *Small–DDPG* curve in the graph represents the outcome of a very simple Actor–Critic architecture using networks with only two layers and very small number of nodes. Although *Small–DDPG* seems to not train the robot efficiently, Figure 13 illustrates the outcome of the same model in many more episodes. It is obvious that there is an ascending tendency especially after 20,000 episodes.

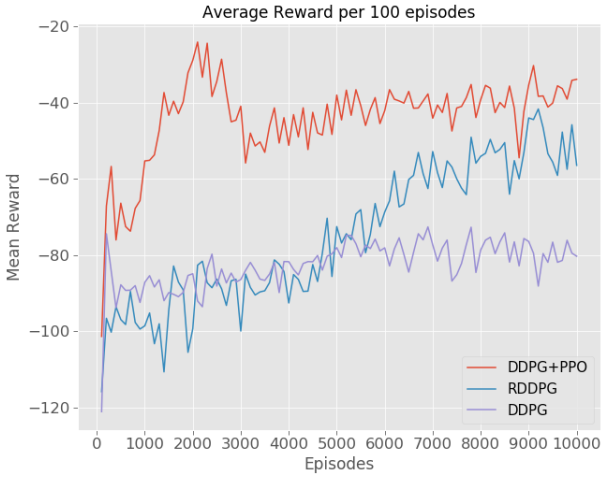


Figure 11: Mixed Deterministic Policy Gradient results

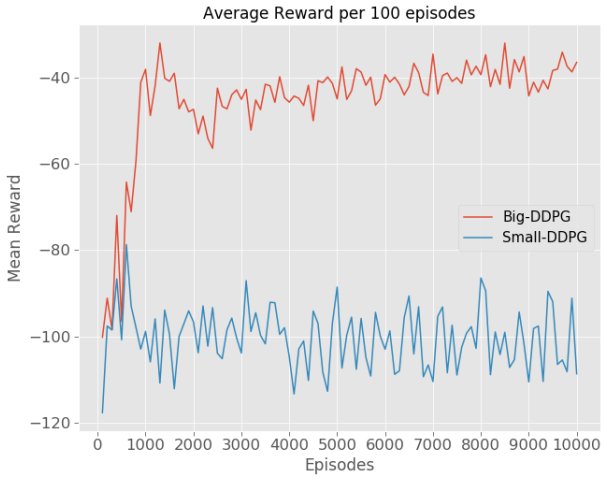


Figure 12: Deterministic Policy Gradient results

Table 1 provides information about the performance of the aforementioned algorithms. We measured the efficiency of the results using the maximum reward on average and the total episode reward in the whole simulation. Of course, the main performance metric is the average reward curve per 100 episodes. All the simulations ran on GPU machines in the *Data Science Lab* in LIACS. The source code can be found at the Kyziridis/BipedalWalker repository.

Algorithm	Max Average Reward	Max Episode Reward	Time in hours
Boosted PPO	252	298	3
Simple PPO	144	215	2.3
DDPG+PPO	−24	11	2.7
Big DDPG	−32	7	3.5
RDPG	−41	9	12
DDPG	−72	3	2
Small DDPG	−78	5	1.2

Table 1: Final results

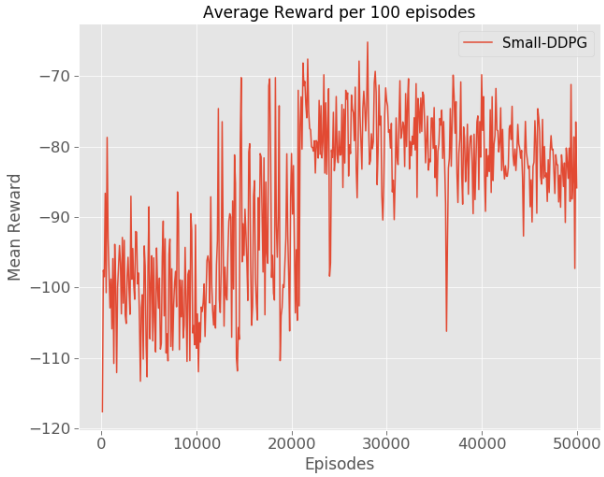


Figure 13: Simple-DDPG on 50,000 episodes

7. CONCLUSIONS

It is obvious that the best results, in terms of the average reward per 100 episodes, are produced from the *Boosted–PPO* algorithm visualized in Figure 8. Unfortunately this paper does not contain much information about the Proximal Policy Optimization algorithm because the main research interest was on the Deterministic Policy Gradient. However, many of the simulations using only the DDPG algorithm seem to have an ascending tendency regarding the rewards. We discuss the results of PPO because it is the latest approach that provides good results for the specific task of robot locomotion. The idea was to compare our implementations to the ones of PPO. Finally, we can draw the conclusion that the Deterministic Policy Gradient algorithm has many limitations on the specific task, however, its variants such as RDPG, could probably be able to train the robot up to the far end, using the parallel–agents approach from [5].

The task of robot locomotion constitutes a main scientific research especially in the AI field. Problems like this one need various experimentation with different approaches and different experimentation set–up. For future work, it would be interesting if we tried to figure out an efficient combination of the Deterministic Policy Gradient and Proximal Policy Optimization. Moreover, the implementation of the Recurrent Deterministic Policy Gradient is an intelligent idea which could lead to satisfying results. There are already various approaches which combine different algorithms such as PPO and Q–learning.

8. REFERENCES

[1] N. Heess, J. J. Hunt, T. P. Lillicrap, and D. Silver. Memory-based control with recurrent neural networks. *arXiv preprint arXiv:1512.04455*, 2015.

[2] V. R. Konda and J. N. Tsitsiklis. Actor-critic algorithms. In *Advances in Neural Information Processing Systems*, pages 1008–1014, 2000.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[4] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *ICML*, 2014.

[5] D. R. Song, C. Yang, C. McGreavy, and Z. Li. Recurrent deterministic policy gradient method for bipedal locomotion on rough terrain challenge. 2017.

[6] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, 2ns edition, 2018.

[7] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, pages 1057–1063, 2000.

[8] V. Tangkaratt, A. Abdolmaleki, and M. Sugiyama. Guide actor-critic for continuous control. *arXiv preprint arXiv:1705.07606*, 2017.

[9] M. Tokic. Adaptive ε-greedy exploration in reinforcement learning based on value differences. In *Annual Conference on Artificial Intelligence*, pages 203–210. Springer, 2010.