Research Project

BipedalWalker-V2

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

ABSTRACT
Abstract Abstract!!

Categories and Subject Descriptors
I.2.6 [**Artificial Intelligence**]: Learning

General Terms
Reinforcement Learning, Policy Gradient

Keywords
Actor-Critic, Deep-q-learning

1. INTRODUCTION

2. GAME-ENVIRONMENT

`BipdelWalker-v2` environment is provided from `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.

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 $R(a_t, s_t) \in \mathbb{R}$ 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 $s_t, t = 1, 2, \ldots, n$. The ten trailing numbers in this vector reflects 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 range $[-1, 1]$. As stated above, in each state s_t, the environment takes as input an action vector a_t and immediately generates the current reward $R(a_t, s_t)$, 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 provide a real-time visualization of the game, Figure 1 illustrates a random position of the robot monitored directly from the rendering environment.

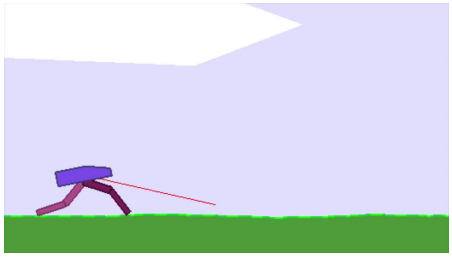

Figure 1: BipedalWalker Screenshot

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

SNACS '18 Social Network Analysis for Computer Scientists, Master CS, Leiden University (`liacs.leidenuniv.nl/~takesfw/SNACS`).

$$State = \langle \mathtt{h}_0, \frac{\partial \mathtt{h}_0}{\partial \mathtt{t}}, \frac{\partial \mathtt{x}}{\partial \mathtt{t}}, \frac{\partial \mathtt{y}}{\partial \mathtt{t}}, \mathtt{h}_1, \frac{\partial \mathtt{h}_1}{\partial \mathtt{t}}, \mathtt{k}_1, \frac{\partial \mathtt{k}_1}{\partial \mathtt{t}}, \mathtt{h}_2, \frac{\partial \mathtt{h}_2}{\partial \mathtt{t}}, \mathtt{k}_2, \frac{\partial \mathtt{k}_2}{\partial \mathtt{t}}, \iota_{leg1}, \iota_{leg2}, \mathtt{lidar} \in \mathbb{R}^{10} \rangle$$

$$Action = \langle \mathtt{H}_1, \mathtt{K}_1, \mathtt{H}_2, \mathtt{K}_2 \rangle$$

Regarding the $State$ vector, $\mathtt{h}_0$ is 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 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.

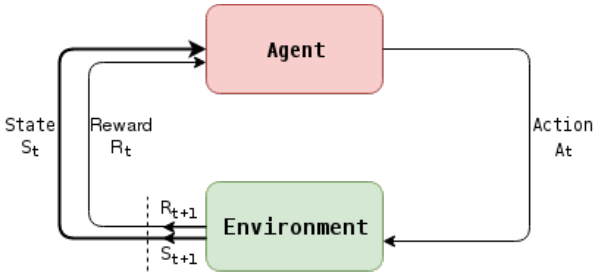

Figure 2: Agent-Environment Feedback Loop
Image inspired by Sanyam Kapoor blog

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 [8] 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 will be 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 attempts to learn the value of being in a given state, $s_t \in \mathcal{S}$, and making a specific action, $a_t \in \mathcal{A}$. So when the robot is located at a current position s_t, it chooses an action $\hat{a}$, and it 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:

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

The above equation is the well-known $Bellman\text{-}equation$ used in Q-Learning, where $R(a, s) \in \mathbb{R}$ 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. That will be achieved through the above procedure using a model which predicts $\max Q(\hat{a}, s')$ and is trained using the $Bellman\text{-}equation$ outcome. Algorithm 2 describes the above procedure from [8].

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** from **s** using **policy** obtained from **Q**
      â ← **policy(s,Q)**
      Operate **â**, observe **r,s'**
      Update $Bellman\text{-}equation$:
      $\mathbf{Q}_{new} \leftarrow (1 - \alpha) \cdot \mathbf{r} + \alpha \cdot \gamma \max \mathbf{Q}(\hat{\mathbf{a}}, \mathbf{s'})$
      Update state **s**: **s** ← **s'**
    end for
  end for
end procedure

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 is to predict the next 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, $R(a, s)$, 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 opportunity to the robot 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 $\epsilon\text{-}greedy$ algorithm which manipulates the balance between random choices and the predicted ones. This algorithm will be described in the following sections.

4.1 Model 1

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 tanh activation function. We used the tanh activation function in the last hidden layer due to 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 network architecture.

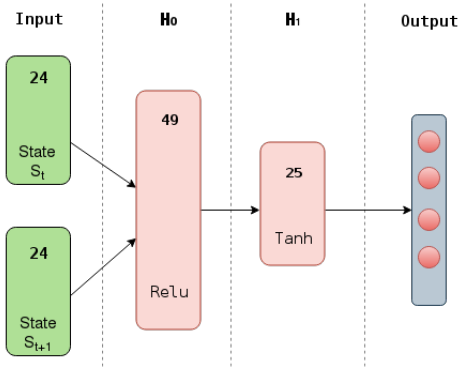

Figure 3: First Naive Network

4.2 Model 2

The second model is derived from the previous network. It is a softly upgraded version of Model1 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 Models 1 and 2 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.

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 pursuant to the current policy. So we need a model which predicts an action, $\hat{a}$, for each state $s_t, t = 1, 2, \ldots, n$ 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 that criticizes the predicted action and produces the approximated reward value, $Q(\hat{a}, s_{t+1})$. This function is usually another neural network as we will see in the next sections, but for the time being, we used 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

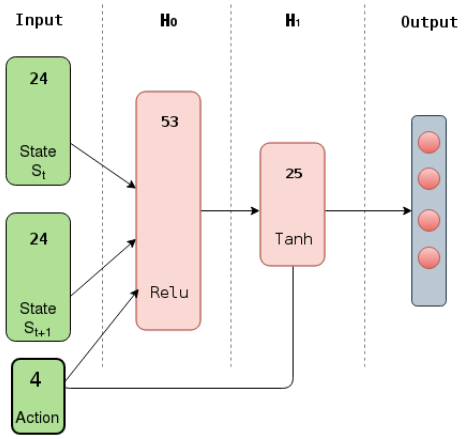

Figure 4: Upgraded Naive Network

a vector $\vec{Q}_{new}$ whose values represent the sum of the action vector $\hat{a}$ and the discounted reward vector $(1-a)R(a, s)$. 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**: ⟨**state**, **action**, **reward**, **state′**, **flag**⟩
Save **observations** as ⟨**s**, **a**, **r**, **s′**, **f**⟩ in **memory**
After **episode** ends get **random.sample** from **memeory**
for each **s, a, r, s′, f** 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 $R(a, s)$, 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 train-

ing the weights into this direction.

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

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

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

4.5 Epsilon-Greedy

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 not probably the 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* [8]

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 algorithm ϵ-greedy handles the trade-off between exploration and exploitation mode by controlling the parameter of randomness. More precisely, ϵ-greedy controls the probability $0 \leq \epsilon \leq 1$ that the robot selects an action at random, instead of selecting an action from the learned experience according to the current policy π derived from the Q-function:

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

where ξ is a uniform random number drawn at each time step, t. All in all, the above equation is described thoroughly in [8][10], brings 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 $\delta_1 = 0.999, \delta_2 = 0.9999$, explaining the significant impact of parameter δ. In each experiment ϵ value is initialized to 1.

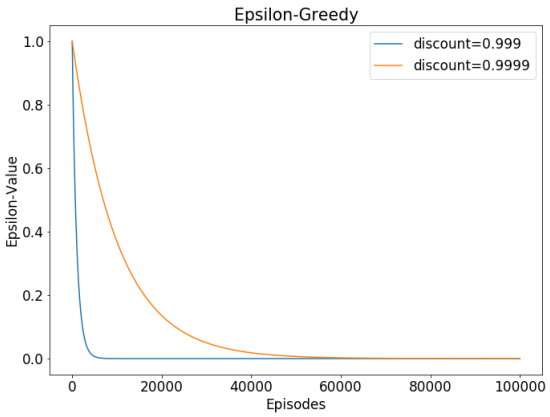

Figure 5: ϵ-greedy example

4.6 Training

Regarding the intuition and the heuristic in 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:

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

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 [4]. The network is trained for each one of the observation-tuples $\langle s, a, r, s', flag\rangle$, using the aforementioned heuristic.

4.7 Naive Models Results

Figure 6 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 performed 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 a very naive heuristic and simple network implementation.

(a) Average Rewards Model1 (b) Average Rewards Model2

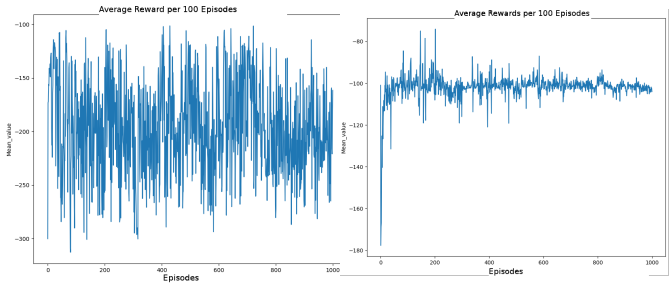

Figure 6: Average Rewards per 100 Episodes

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 first mention some basic preliminaries from [8].

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

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

where r indicates the reward for the trajectory, τ.

5.1 Policy Gradient

Policy-Gradient (PG) algorithm [9], 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, 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.

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

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

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

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

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

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

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 defines that 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 *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 the Equation (12) will be:

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

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 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 robotics field, 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 bellow where μ indicates the action with the highest Q-value for a current state, s.

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

However, the maximization is quite infeasible wherefore 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. Deterministic Policy Gradient [7] 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 $\mu_{\theta} : S \rightarrow A$ 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

Actor-Critic learning algorithm [6], 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 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 $\hat{a} \in A$, and the critic will produce the approximated value given that action, $Q(s, \hat{a})$. In this way, 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.

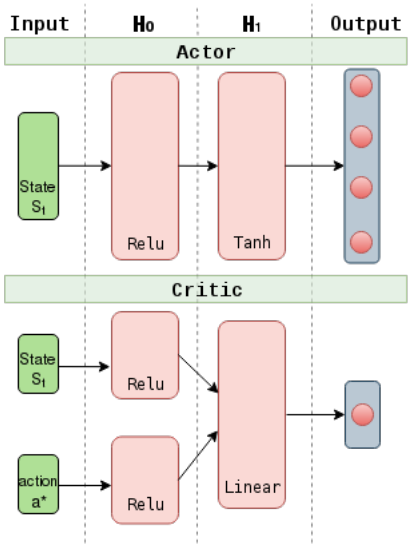

Figure 7: Actor-Critic Network

Figure 7 illustrates a simple actor-critic network architecture. It can be easily observed that the actor network gets as input the current state vector, $s_t \in \mathcal{S}$, and generates the predicted action, $\hat{a} \in A$ 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 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 critices the taken action in the current state. So, consider an 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 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 critic, which also updates its way to provide more accurate feedback. Both networks are trained in parallel and learn from the game experience.

$$Actor : \pi(s, a, \theta)$$

$$Critic : \hat{Q}(s, a, w)$$

Because we have two models (Actor and Critic), it means 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 described in the following equations:

$$Policy : \theta = \alpha \nabla_\theta \Big(\log \pi_\theta(s, a) \Big) \hat{Q}_w(s, a)$$

$$Value : w = \beta \Big(\underbrace{R(s, a) + \gamma \hat{Q}_w(s_{t+1}, a_{t+1}) - \hat{Q}_w(s_t, a_t)}_{TD} \Big) \underbrace{\nabla_w \hat{Q}_w(s_t, a_t)}_{\text{Value Function Gradient}}$$

,where 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) [8].

5.4 Experiments

5.5 Training

5.6 Results

APPENDIX

A. REFERENCES

[1] M. Bowman, S. K. Debray, and L. L. Peterson. Reasoning about naming systems. *ACM Trans. Program. Lang. Syst.*, 15(5):795–825, November 1993.

[2] J. Braams. Babel, a multilingual style-option system for use with latex's standard document styles. *TUGboat*, 12(2):291–301, June 1991.

[3] M. Clark. Post congress tristesse. In *TeX90 Conference Proceedings*, pages 84–89. TeX Users Group, March 1991.

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

[5] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993.

[6] V. R. Konda and J. N. Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

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

[8] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[9] 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.

[10] M. Tokic. Adaptive ε-greedy exploration in reinforcement learning based on value differences. In

Annual Conference on Artificial Intelligence, pages
203–210. Springer, 2010.