mathtools

Machine Learning
Benchmarking neural-fitted Actor Critic with state of the art reinforcement learning algorithms[*]

H. Maathuis M. Groefsema S. Warmelink T. Oosterhuis

University of Groningen, 9747AG Groningen

Abstract

1 Introduction

TODO:: current state of the art stuff, mention OpenAI, Alpha-Go,.. TODO:: Research question

Reinforcement learning is a machine learning technique which allows for agents to autonomously learn relatively complex behaviour in a variety of applications such as such as playing backgammon [**?**] or chess [**?**], real life quadripedal walking [**?**], or autonomous simulated driving [].

With reinforcement learning an agent is placed in an environment in a state where it can execute a number of actions. Each action the agent takes in a state entails a reward or a punishment for the agent as well as bringing it to a new state. By maximizing tt's reward the agent gradually learns the value of each state (either by keeping track of this value in a lookup table, or when the environment is to complex for this to be feasible, by approximating the value function for the environment with a function approximator such as a multi-layer perceptron - MLP), allowing it to autonomously learn the optimal policy for its environment.

When they were first developed reinforcement learning algorithms were designed to deal with discretized state and action spaces. In real life however state spaces aren't discritized and although action spaces can be, this is not ideal as different actions may require different levels of discritization to be accurate to get the desired reward. Furthermore the required level of discritization can not be known in all 11cases, and having to discover it is time intensive [1]. This potential problem can be solved by keeping the state and action space continuous in the environment representation of the agent.

Several new reinforcement learning algorithms and adaptations of existing reinforcement algorithms were developed to model continuous state and action spaces. In this paper we will benchamark one such algorithm, named Neural Fitted Actor Critic (NFAC), which was develeped in August 2016 [**?**] against two established (at the time of the writing of this paper) continuous reinforcement learning algorithms, named CACLA (Continuous Actor Critic Learning Automaton) [1] and SARSA (State Action Reward State Action) adapted for continuous state- and action-spaces [**?**] with gradient descent (referred to in this paper as GD-SARSA).

First a formal background will be given on connectionist reinforcement learning with the use of MLP's as function approximators, and on extending this to continuous state- and action-spaces. In the methods section the three algorithms that are compared in this paper the implementation of these algorithms are explained in detail. After that setup of the experiments will be discussed. Finally the results will be

[*]In case of an extended abstract refer to the original paper in a footnote such as "The full paper has been published in *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 13–20, 2013." Also, please keep the title and authors exactly the same as the original.

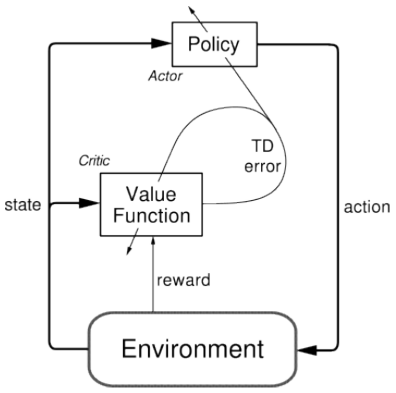

Figure 1: Actor Critic system. Reprinted from [2]

presented and evaluated, and the final conclusions will be drawn regarding NFAC compared to CACLA and GD-SARSA.

2 Background

3 Continuous Actor Critic Learning Automaton

A well established algorithm in Reinforcement Learning (RL) is Continuous Actor Critic Learning Automaton (CACLA) [1]. CACLA deals with undiscretized continuous state and action spaces. It implements an Actor Critic system in which both the Actor and Critic are represented by a Multi-Layer Perceptron. In an Actor-Critic system, the Actor is responsible for selecting the current action given the policy and the Critic is used in the calculation of the Temporal Difference (TD)-error which drives the learning of the Actor. The TD learning rule is characterized by equation (1), where σ_t is the TD-difference error as written in (2). The Actor-Critic system allows for a seperation between the representation of the policy and the value function. A visualization of the Actor-Critic system is shown in Figure 1. In RL problems it is important to be aware of the exploration versus exploitation trade-off; meaning that a decision has to be made between exploration which might lead to an improvement of the current policy or simply the action is selected according to the current policy. An exploration technique that is used often in CACLA is Gaussian Exploration, meaning that an action is calculated around the best action that the current policy provides. To pick a value around the best action, a sample is chosen from a gaussian distribution $N(action, standarddeviation)$. The sign of the TD-error ($\sigma_t > 0$) is used to determine whether the policy of the Actor has to be updated. To update the Actor in a connectionist way, the performed action a_t is used as the target when backpropgating the error using the state s_t as input. Since CACLA is an online RL algorithm, the data that is gathered from the interaction of the agent with the environment is used immediately to update the Critic and possibly the Actor.

$$V(S_{t+1}) = V(S_t) + \alpha_t \sigma_t \tag{1}$$

$$\sigma_t = r_t + \gamma V_t(s_{t+1}) - V_t(s_t) \tag{2}$$

4 SARSA

We implemented SARSA for the described environments, using a modification of [**?**]. In their work Newtons Method (NM) is used to generate continues actions with a high expected Q-value. It our work we use a slight variation of this approach. Instead of using NM, which requires first and second order partial-differentials to the input of the MLP, we use gradient-descend to obtain an action with a high expected Q-value. This method only requires first order partial differentials towards the input of the MLP, as shown in equation (3). To generate a action, first multiple starting-points are chosen. For each action-dimension four evenly spread values are taken, resulting in 4^D starting points, for D dimensions of the actionspace. From these starting points K iterations of gradient descend, as described, are used to obtain actions for our current state, with the highest Q-value, by taking the best action obtained from all starting points.

$$a = a + \lambda * \frac{\partial MLP}{\partial a} \tag{3}$$

5 Experimental setup

TODO:::: CONSTRUCT::: (What do we measure) : How well the agent performs TODO:::: OPERATIONALISATION!!!! (How do we measure it?)!! : Average reward last 100 epochs

We will briefly describe the experimental setup used to test the different algorithms. SARSA, CACLA and NFAC are compared in two continuous environments: MountainCar [**?**] and LunarLander [**?**]. Both environments are OpenAI Gym environments [**?**], which is a toolkit for comparing reinforcement learning algorithms.

The discount factors used during the experiments were 0.20, 0.40, 0.60, 0.80, 0.90, 0.99 and 0.999. The learning rates used were 0.001, 0.01 and 0.05.

The number of hidden nodes was varied from 20 to 200. Each environment-algorithm pair had its own optimal number of hidden nodes. A table can be found below.

Each simulation run consists of 2000 epochs, each of which has a maximum of 10000 time steps.

SARSA-GD used ϵ-greedy exploration, which was first set to 0.1, meaning a 10% chance to take a random action during a time step. This exploration rate decayed over epochs, and is equal to $0.1 * 0.99^{N-1}$, where N is the current epoch. Since our experiments run for 2000 epochs, this means that the final exploration rate is $1.86 * 10^{-10}$.

NFAC and CACLA both used gaussian exploration.

TODODODODO:: In NFAC wordt bij gaussian exploration de sigma alleen maar geupdatet wanneer het goal wordt gehaald in een epoch. In CACLA wordt elke epoch de sigma kleiner gemaakt onafhankelijk van of het goal wordt gehaald. Elk epoch updaten is beter te verdedigen. Daarnaast wordt zoals hierboven te zien is de exploration na like 400 epochs al nagenoeg nul. Als op dat punt nog geen successen zijn behaald convergeert het waarschijnlijk naar bagger.

5.1 MountainCar

TODO:: Add image (or in intro) In the MountainCar environment a car is situated in between two mountains. Its goal is to reach the top of the rightmost mountain. In order to do this, it first has to drive up the left mountain in order to generate enough momentum to reach the top of the rightmost mountain.

The action space of the MountainCar environment is a float in the range [-1,1]. This float dictates the force applied to the car in either the left or right direction. The state space consists of two floats:

The car position, in the range [-1.2,0.6]

The car velocity, in the range [-0.07,0.07]

The reward function, where r_t is the reward at time step t and v_t is the car velocity at time step t, is given by the OpenAI Gym environment and defined as follows:

$$r_t = \begin{cases} +100 - v_t^2 * 0.1 & \text{if goal is reached} \\ -v_t^2 * 0.1 & \text{otherwise} \end{cases} \tag{4}$$

The total reward is defined as the sum of all rewards gained during an epoch. The MountainCar action space vector is in the $[-1, 1]$ range.

5.2 LunarLander

TODO:: Add image (or in intro) In the LunarLander environment, the lunar lander's goal is to safely land on the lunar surface. It can do this by firing its left or right engine and controlling its overall thrust power. The action space of the LunarLander environment consists of two floats in the range $[-1,1]$: one for controlling the left or right engine and one for controlling the main engine. In the first case, a value in between -1.0 and -0.5 means the left engine is firing, a value in between 0.5 and 1.0 means the right engine is firing, and any other value means neither engine is firing. For the main engine, a value in between -1.0 and 0 means that the main engine is not firing, while from 0 to 1.0 the engine is throttled from 50% to 100% power. The LunarLander state space consists of 6 floats, all in the range $[-1,1]$ and two booleans: x_position, y_position, x_velocity, y_velocity, angle, rotation and the booleans left_leg_contact and right_leg_contact.

For the reward function, as given by the OpenAI Gym environment, we read the state at time t $state_t$. It is defined as such:

$$state_t = -100 * d_t - 100 * v_t - 100 * a_t + 10 * l_t + 10 * r_t \tag{5}$$

where d_t is the distance from the lunar lander to landing zone at time t, v_t is the velocity of the lunar lander at time t, a_t is the lunar lander's angle at time t, and l_t and r_t are booleans of respectively the left and right foot touching the ground at time t.

Using this $state_t$, we can determine the change in state which is needed for the reward function (equation 7).

$$\delta_{state} = \begin{cases} 0 & \text{for epoch 1} \\ state_t - state_{t-1} & \text{otherwise} \end{cases} \tag{6}$$

The reward function, where r_t is the reward at time step t, m_t is the main engine power and lr_t is the left-right engine power.

$$r_t = \begin{cases} \delta_{state} - m_t * 0.30 - lr_t * 0.03 - 100 & \text{after crash} \\ \delta_{state} - m_t * 0.30 - lr_t * 0.03 + 100 & \text{after landing} \\ \delta_{state} - m_t * 0.30 - lr_t * 0.03 & \text{otherwise} \end{cases} \tag{7}$$

6 Conclusions and further work

7 Contributions

References

[1] H. Van Hasselt and M.A. Wiering. Reinforcement learning in continuous action spaces. In *Proceedings of the IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning, 2007*, pages 272–279. IEEE, 2007.

[2] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.