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 [9] or chess [4], real life quadripedal walking [6], 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 [5]. 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 [10] against two established (at the time of the writing of this paper) continuous reinforcement learning algorithms, named CACLA (Continuous Actor Critic Learning Automaton) [5] and SARSA (State Action Reward State Action) adapted for continuous state- and action-spaces [7] 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.

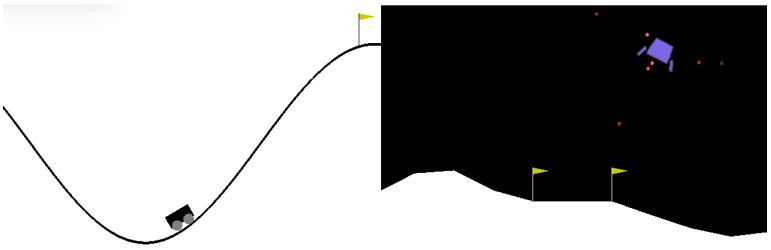

Figure 1: Left: MountainCar Environment, Right: LunarLander Environment.

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

2 Background

2.1 Reinforcement Learning

In Reinforcement Learning problems are modeled as Markov Decision Processes (MDP's). An MDP in the context of Reinfocement Learning is a four-valued tuple (S, A, R, T), where S is a set of states that together make up the environment that a RL agent is in, A is a set of actions the agent can take, $R : S \times A \times S \rightarrow \mathbb{R}$, is a reward function mapping a state the agent is in s_t, an action by the agent a_t and the resulting new state of the agent s_{t+1} to a reward $R(s_t, a_t, s_{t+1})$ and $T : S \times A \times S \rightarrow [0, 1]$ is a series of transition probabilites of $T(s_t, a_t, s_{t+1})$, the probability of the agent ending up in a possible state s_{t+1}, when it executes action a_t in state s_t.
The policy of an agent $\pi : S \times A \rightarrow [0, 1]$ is the probability the agent choosing action a_t in state s_t. The agent learns by storing values for each state or for each combination of possible states and actions, allowing it to optimize its policy by maximizing its expected total discouned reward (see equation 1, also [10]).

$$\pi^* = \arg\max_\pi \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \times R(s_t, \pi_t(s_t))\right] \tag{1}$$

In equation 1 t is a time step and γ is the discount factor, which weights the value the current reward against the value of potential future rewards. The discount factor is traditionally defined as being part of a Markov Decision process, but for Reinforcement Learning purposes the discount factor is seen as part of the algorithms and not as part of the MDP, because different algorithms require different discount factors to perform optimally when the model (S, A, R, T) is kept the same [5]. In the RL setting of this paper the agent must learn the optimal policy model-free, meaning R and T are not known. State values or Q-values (values of a certain action in a certain state) are updated, for example on each time step, see the function in equation 2, where α is the learning rate.

$$Q_t + 1(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t * (r_t + \gamma Q_t(s_t + 1, a_t + 1) - Q_t(s_t, a_t)) \tag{2}$$

2.2 Continuous State Space

When S is continuous storing the values of states, or state action combinations is not possible. However the function $V(s_t)$ that maps states (or $Q(s_t, a_t)$ that maps combinations of states and actions) can be approximated with a Function Approximator. We used Multi-Layer Perceptrons as Function Approximators and updated the MLP weights using back-propagation.

2.3 Continuous Action Space

When A is continuous as well, naively applying the argmax operator for action selection is no longer possible. One solution to this to use an MLP as a Function Approximator for action selection as well as

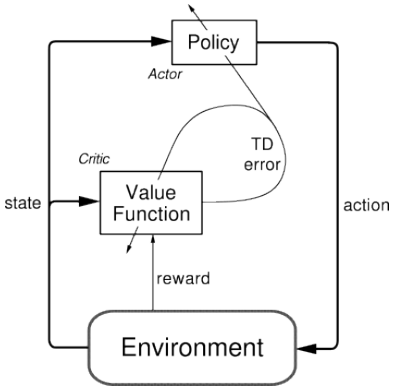

Figure 2: Actor Critic system. Reprinted from [8]

for determining the value. Another solution is to use a discretized action space as a starting point to find a number of local maxima in the action space, for example by iteratively applying gradient descent on the starting discrete actions, and to finally take the argmax of the local maxima.

3 Continuous Actor Critic Learning Automaton

A well established algorithm in Reinforcement Learning (RL) is Continuous Actor Critic Learning Automaton (CACLA) [5]. 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 (3), where σ_t is the TD-difference error as written in (4). 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 2. 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.

$$V(S_{t+1}) = V(S_t) + \alpha_t \sigma_t \tag{3}$$

$$\sigma_t = r_t + \gamma V_t(s_{t+1}) - V_t(s_t) \tag{4}$$

4 Neural Fitted Actor Critic

Another Reinforcement Learning that uses an Actor-Critic system is Neural Fitted Actor-Critic (NFAC). Where CACLA is an offline Actor-Critic algorithm, NFAC is an online Actor-Critic algorithm. This means that a data collection D_π is built which is filled with experience tuples of the form $\{s_t, u_t, a_t, r_{t+1}, s_{t+1}\}$. s_t is the state at time t, u_t is the action the Actor MLP provides, a_t is the exploration action which can deviate from u_t, the reward at time $t+1$ is given by r_{t+1}, and finally s_{t+1} is the resulting state after applying u_t. Every time the agent interacts in the environment, D_π will be extended with one tuple.

	learning rate	discount factor	number of hidden nodes
SARSA	N	P	M
CACLA	A	B	actor: F, critic: G
NFAC	X	Y	actor: H, critic: I

Table 1: MountainCar network parameters per algorithm

	learning rate	discount factor	number of hidden nodes
SARSA	N	P	M
CACLA	A	B	actor: F, critic: G
NFAC	X	Y	actor: H, critic: I

Table 2: LunarLander network parameters per algorithm

In CACLA, first the Critic is updated and afterwards the Actor. However since the Critic is dependent on the value of the Actor when updating in batches, the Actor is updated first. The Actor is updated towards the exploration action if the TD-error is > 0. This is identical for CACLA. NFAC deviates from CACLA by also updating the Actor when the TD-error is ≤ 0. In that case the update of the Actor is towards the action that the Actor MLP provides. The Critic update is similar to CACLA, but in NFAC the Critic is updated for each experience in D_π. Note that the data collection needs to be emptied after every epoch since the targets used in estimating the new value function are dependent on the current value function.

5 SARSA

TODO::: INTRODUCE 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 (5). 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.

$$a = a + \lambda * \frac{\partial MLP}{\partial a} \tag{5}$$

6 Experimental setup

In this section, the experimental setup used to test the different algorithms will be described. SARSA, CACLA and NFAC are compared in two continuous environments: MountainCar [3] and LunarLander [1]. Both environments are OpenAI Gym environments [2], which is a toolkit for comparing reinforcement learning algorithms. Agent performance is measured by looking at the average reward over the best 100 epochs. The reward function is given by the OpenAI environment and shown in more detail in their respective subsections. Each simulation run consisted of 2000 epochs, each of which had a maximum of 10000 time steps. Learning rates, discount factors and numbers of hidden nodes for the MountainCar and LunarLander environments can be found in table 6 and table 6 respectively.

(GD) SARSA 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 the experiments run for 2000 epochs, this means that the final exploration rate is $1.86 * 10^{-10}$.

NFAC and CACLA both used gaussian exploration. A random value sampled from a gaussian distribution($\mu = 0$ $\sigma = 1$) was multiplied by a value $\Sigma = 10$. This value was added to the output of

the MLP and finally clamped to be in the range [−1,1]. This Σ decayed over epochs and is equal to $10 * 0.99^{N-1}$, where N is the current epoch. Since the experiments run for 2000 epochs, this means that the final exploration rate is $1.86 * 10^{-8}$. Since initially the exploration rate is very high, this ensures that the agent will quickly reach its goal. [[The exploration rate is diminshed over time since then it can rely more on its learned behavior rather than the noise added by the gaussion value.]]

6.1 MountainCar

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:

$$r_t = \begin{cases} +100 - v_t^2 * 0.1 & \text{if goal is reached} \\ -v_t^2 * 0.1 & \text{otherwise} \end{cases} \tag{6}$$

The total reward is defined as the sum of all rewards gained during an epoch.

6.2 LunarLander

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: the floats x_position, y_position, x_velocity, y_velocity, angle and rotation, and the booleans left_leg_contact and right_leg_contact.

For the reward function, as given by the OpenAI Gym environment, the state at time t st_t is determined. It is defined as such:

$$st_t = -100 * d_t - 100 * v_t - 100 * a_t + 10 * l_t + 10 * r_t \tag{7}$$

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 st_t, the change in state which is needed for the reward function can be determined (equation 8).

$$\delta_{state} = \begin{cases} 0 & \text{for epoch 1} \\ st_t - st_{t-1} & \text{otherwise} \end{cases} \tag{8}$$

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, is given here (equation 9):

$$r_t = \begin{cases} \delta_{state} - m_t * 0.30 - lr_t * 0.03 - 100 & \text{after crash} \\ \delta_{state} - m_t * 0.30 - lr_t * 0.03 + 100 & \text{after landing} \\ \delta_{state} - m_t * 0.30 - lr_t * 0.03 & \text{otherwise} \end{cases} \tag{9}$$

The total reward is defined as the sum of all rewards gained during an epoch.

7 Conclusions and further work

8 Contributions

References

[1] OpenAI Gym lunarlander environment. `https://gym.openai.com/envs/LunarLanderContinuous-v2`. Accessed: 02-06-2017.

[2] OpenAI Gym main webpage. `https://gym.openai.com/`. Accessed: 02-06-2017.

[3] OpenAI Gym mountaincar environment. `https://gym.openai.com/envs/MountainCarContinuous-v0`. Accessed: 02-06-2017.

[4] J. Baxter, A. Tridgell, and L. Weaver. Knightcap: a chess program that learns by combining td (lambda) with game-tree search. *arXiv preprint cs/9901002*, 1999.

[5] 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.

[6] N. Kohl and P. Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 3, pages 2619–2624. IEEE, 2004.

[7] B.D. Nichols and D.C. Dracopoulos. Application of newton's method to action selection in continuous state-and action-space reinforcement learning. In *Proceedings of the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2014)*. ESANN, 2014.

[8] R. Sutton and A.G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[9] G. Tesauro. Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134(1-2):181–199, 2002.

[10] M. Zimmer, Y. Boniface, and A. Dutech. Neural fitted actor-critic. In *Proceedings of the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2016)*. ESANN, 2016.