A Review of Self-balancing Robot Reinforcement Learning Algorithms

Aistis Raudys(✉) and Aušra Šubonienė(✉)

Institute of Computer Science, Vilnius University, 47 Didlaukio, 08303 Vilnius, Lithuania
{aistis.raudys,ausra.suboniene}@mif.vu.lt
https://mif.vu.lt/lt3/en/about/structure/institute-of-computer-science/

Abstract. We analyse reinforcement learning algorithms for self balancing robot problem. This is the inverted pendulum principle of balancing robots. Various algorithms and their training methods are briefly described and a virtual robot is created in the simulation environment. The simulation-generated robot seeks to maintain the balance using a variety of incentive training methods that use *non-model-based* algorithms. The goal is for the robot to learn the balancing strategies itself and successfully maintain its balance in a controlled position. We discuss how different algorithms learn to balance the robot, how the results depend on the learning strategy and the number of steps. We conclude that different algorithms result in different performance and different strategies of keeping the robot balanced. The results also depend on the model training policy. Some of the balancing methods can be difficult to implement in real world.

Keywords: Self-balancing robot · Reinforcement learning · Neural networks

1 Introduction

We analyse two-wheeled robot balancing problem. The movement of such a robot is modelled using an inverted pendulum model as the robot's centre of mass is above the pivot point. This model is inherently unstable, and must be actively balanced in order to remain upright. Various sensors and state measurements can be used, but the most common are wheel encoders and IMU sensors, using a combination of accelerometers and gyroscopes. Sensors directly measure robot's velocity, angular velocity of wheels as well as robot's angle.

These measurements are then used by the controller to provide commands to actuators in order to achieve the desired behaviour of a robot. The controller creates low power signals, which are passed through the amplifier and then sent to the actuators, which create robot forces and torques. The movement and forces of a robot are measured using sensors, which feed the measurements back to the controller. Because of such feedback, the process is called closed loop

A. Lopata et al. (Eds.): ICIST 2020, CCIS 1283, pp. 159–170, 2020.
https://doi.org/10.1007/978-3-030-59506-7_14

control. Power disturbances and sensor errors are often included in the model control cycle.

Majority of such algorithms are human coded and do not pay attention to real world factors such as slipping, load, wear-and-tear and so on. The analysis is often simplified making an assumption that amplifiers and actuators are perfect at generating control forces and angular moments which are required by the controller. It is also assumed that the sensors measure the performance of the robot perfectly. The model is also simplified by ignoring the fact that the controller is typically implemented at a finite frequency. Instead, it is assumed that control rules operate in continuous time. Then the control scheme of the robot can be simplified into a control loop of a controller feeding required forces to the actuators. They change the state of a robot which is measured by the sensors and then used again by the controller to make the next step.

Traditionally control loop mechanisms such as proportional–integral–derivative controller (PID), linear–quadratic regulator (LQR) or fuzzy logic controllers or their variations were widely used in robotics control systems. PID and its variants are some of the most common controllers in balance control [4,5]. Even though smaller balancing errors can be achieved using LQR instead of PID controller [1], the mathematical model is needed in order to achieve better results. Also, settling times using LQR can be longer than using PID controller [4]. Fuzzy logic controllers are also used to solve the balance problem. Although both PID and fuzzy logic controller can achieve extremely small steady state errors [6], fuzzy logic controller can be more stable than conventional PID controller [6,7]. In addition to this, fuzzy logic controllers are also combined with neural networks, which results in improved stability and adaptability of the robot [15,17].

In addition to controllers mentioned above, neural networks are also proven to provide good control mechanisms. Either using simple neural networks alone [4], or using recurrent neural networks [8,12,19], improved adaptability to the changes in terrain or mass can be achieved. Even though various solutions provide good results in solving the balancing problem, it is often very difficult to compare different controllers, especially the ones using various neural networks algorithms due to different models of the robot that is used for testing, different experimental conditions and varying parameters of the system. All these possible changes complicate the analysis of different algorithms for balance control. The analysis and comparison of different reinforcement learning algorithms for the balancing problem will result in comparable results for any future work. Several different algorithms will be analysed using a controlled and fully reproducible environment, which allows for direct comparison between different reinforcement learning algorithms which are either mentioned here or created later.

2 Control Algorithms Without Reinforcement Learning

2.1 Proportional–Integral–Derivative Controller (PID)

The usual closed-loop controller in robotics is the PID (proportion-integral-derivative) controller. The three separate controllers (P, I and D) are connected to generate a control signal. The PID controller tries to maintain the output such that there is zero error between the process variable and the desired behavior.

The proportional, or P-controller, produces an output that is proportional to the current error. As long as there is an error (process variable at a non-desired point), Controller I will continuously increase or decrease the controller output value, thus reducing the error. If the error is high, integral mode will increase or decrease the controller output quickly. D controller's output depends on the error rate variation over time multiplied by the derivative constant which allows the system react faster when needed. By combining P, I and D controllers, a PID controller is obtained which is able to control the system so that the robot remains in a balanced position. To control not only the balance but also the displacement of the robot, i.e. to move in a plane, two PID controllers are used one for speed and one for tilt.

2.2 Linear–Quadratic Regulator (LQR)

Linear Quadratic Regulator (LQR) – is an algorithm which is concerned with operating a dynamic system at minimum cost. It can be considered as an automatic way of finding an appropriate state-feedback controller and is a controller that can be optimal in two aspects- balancing and lost cost. Having the system model expressed as $\dot{x} = Ax + Bu$, the feedback control rule minimizing the price value is $u = -Kx$, where K is found by $K = R^{-1}B^TP(t)$ and P is found by solving Riccatti's differential equation.

Then u is selected as an input to achieve the system control objective and obtain a closed loop system dynamics rule $\dot{x} = Ax + Bu$.

3 Reinforcement Learning Algorithms

3.1 LTSM and MLP Policies in Reinforcement Learning Algorithms

The policy defines robot's way of behaving at a given time. Algorithm must find such policy with maximum expected return. In this work two policies were studied with several reinforcement learning algorithms: multilayer perceptron (MLP) and long short-term memory (LSTM). MLP policy is often used in control applications where linear function is not sufficient. Neural network inputs are angles between individual parts of the robot or robot and the environment, and the speed at which those angles change. LSTM policy adds complexity to the robot's behaviour as it uses information learned at previous steps in order to make a decision on a certain action. Information is processed using write, read and keep gates as well as an information cell. The results not only return the output value, but also update the internal state. This way knowledge gained in previous states influence future decisions.

3.2 DeepQ Learning

DeepQ learning algorithm is based on the idea of Q-learning [16], which is a model-free reinforcement learning algorithm for solving Markov decision process. Q-learning finds the policy which maximises the expected value of total reward by iteratively computing the values for the action-value function. Q-learning was later combined with deep learning by DeepMind [10] into DeepQ learning algorithm as a way to approximate Q-values. Instead of updating individual Q-values, using DeepQ learning the updates are performed to the parameters of the network.

DeepQ learning also uses the experience replay which allows for greater data efficiency, behaviour distribution is averaged over many previous states, and randomizing batches breaks correlations between samples. Also, DeepQ learning derives Q-values in one forward pass where Q-values for are predicted for each action for a given state as opposed to Q-learning, where state and action are needed to be given as inputs resulting in Q-value for that particular state and action. In the robotics environment, DeepQ learning produces better results for robot balance compared to usual controllers such as LQR or PID, although PID could sometimes lead to more stable results [11].

3.3 Trust Region Policy Optimisation (TRPO)

TRPO is an on-policy algorithm, which updates policies not by keeping old and new policies close in parameter space, but by taking the largest possible step to improve performance within the bound of constraint [14]. This determines how close the new and old policies are allowed to be. As a result, TRPO avoids situations where small differences in parameter space could have very large differences in performance improving the balance quickly and monotonically.

TRPO uses single path procedues in order to collect state-action pairs, together with Monte Carlo estimates for Q-values. It then creates the predicted goals and constraints by averaging samples. Finally, the strategy parameter vector is updated using conjugate gradient algorithm, followed by line search.

Although TRPO performs well for certain applications, it is computationally expensive, as it calculates H matrix for each iteration of the algorithm. It is unable to scale to big networks and also suffers from sample inefficiency.

3.4 Advantage Actor-Critic (A2C)

Advantage Actor-Critic method [9] is a variant of more general actor-critic algorithms which combine value-based methods and policy based methods. In actor-critic methods both value function and policy function are learned. Q-value is learned by parametrising Q-function using neural network. Critic updates the parameters of value function, which could be action value or state value, depending on the algorithm. Actor then updates policy parameters in the direction suggested by the critic.

In A2C algorithm Q-values can be expressed by combining the state value function V(s) and the advantage value $A(s_t, a_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)$, which is used to determine how better one action is compared to the other action at a given state, as opposed to the value function, which captures only how rewarding the current state is. Then the update equation becomes:

$$\nabla_\theta J(\theta) \sim \sum_{t=0}^{T-1} \nabla_\theta log\pi_\theta(a_t, s_t)(r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)) \quad (1)$$

Then instead of the critic learning the Q-values, it learns the advantage values, which is possible using only one neural network for the state-value function V(s). In this way the action is evaluated not only on the basis of how good it is, but also how much it can be improved. The advantage function in A2C makes the model more stable and reduces the high variance of the policy network.

3.5 Sample Efficient Actor-Critic with Experience Replay (ACER)

ACER uses a combination of ideas used in several other algorithms, some of which are discussed above. ACER uses multiple worker threads like A2C, a replication buffer, RETRACE algorithm and trust region optimisation. On the other hand ACER introduces several new approaches, such as truncated importance sampling with bias correction, stochastic dueling network architectures, and a new trust region policy optimization method [18].

Policy network is used to estimate the probabilities of actions. During a learning phase data sample is taken from categorical action distribution, related to these probabilities. During a testing phase the actions related to the highest probabilities are used.

During every policy update these steps are performed: the state values are found, then Q-retrace is calculated, followed by collecting gradients and calculating policy gradients, also the trust region is updated, which is used to minimise the difference between the updated policy and mean policy to ensure the stability of the algorithm [18].

3.6 Proximal Policy Optimization (PPO)

Instead of trying to limit or optimise the size of the strategy update step as in TRPO or ACER algorithms, which lead to difficult implementation or issues in practical use for algorithms which have shared parameters for policy and value functions, PPO uses clipped probability rations. It creates a pessimistic policy evaluation (the lower threshold). In order to optimise the policy, data selection and sample creation using policy are constantly changed through multi-epoch optimisation of data samples.

PPO [13] uses fixed-length trajectory segments. During each iteration, each of the N actors acting in parallel runs the policy in the environment for a fixed number of steps T and collects data from those steps. Then the advantage estimates are computed. After this has been done for every actor, the surrogate loss function is constructed and optimised, and the network parameters are updated.

In the neural network architecture, where strategy and value function share common parameters, the loss function uses a policy substitute and a value function error element. Also, the objective function is supplemented by adding an entropy element to ensure sufficient exploration.

4 Robot Model and Environment

The model of the robot was created in OpenAI Gym environment [2]. The model consists of one rectangular parallelepiped of size 20 cm $\times$ 5 cm $\times$ 40 cm, imitating the body of the robot. The mass of the body is 0.8 kg, and the center of mass is at the center of the body. Two cylindrical wheels were attached to the body, with a diameter of 10 cm and width of 2 cm. Each wheel weighs 0.1 kg. The robot starts each simulation from a slight angle in order to start balancing (Fig. 1).

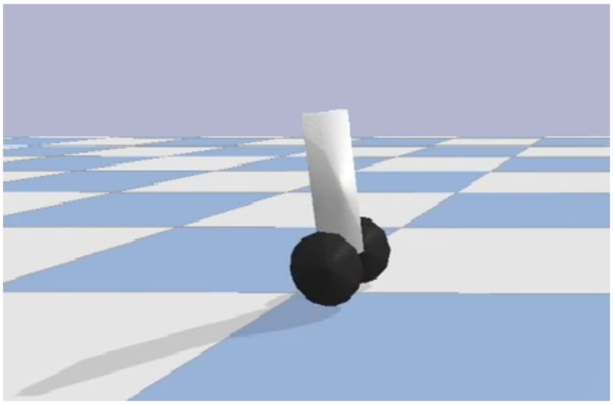

Fig. 1. Visualisation of robot in OpenAI environment

The simulation environment for the robot was created using OpenAI Gym [2] toolkit together with PyBullet physics engine [3]. A plane was created through x and z axis, and standard acceleration of free fall set as 10 m/s^2. In order to learn, the robot was able to choose from 9 different discrete actions which allow the robot to increase or decrease current angular velocity of wheels by 0, 0.1, 0.2, 0.5 or 1 rad/s. This allowed the robot to fine tune small oscillations in order to maintain the balanced position as well as sharply increase the angular velocity or even change the direction of movement if needed.

The state of the environment consists of the tilt angle of the robot, the angular velocity of the robot and the angular velocity of the wheels. Angular velocity of each wheel is not used in order to avoid cheating. This would allow the robot to learn undesirable balancing strategies, such as using angular velocity of the same amplitude but different direction for each wheel, which would result in the robot maintaining the balance by spinning around its axis.

The reward of for each state at time t is calculated using the formula:

$$r_t = 1 - |\alpha| \cdot 0.1 - |v_c - v_d| \cdot 0.01 \tag{2}$$

where α - tilt angle of the robot (rad), v_c - current angular velocity of wheels (rad/s), v_d - desired velocity of wheels (rad/s). In order to achieve a balanced position with no movement back and forth (as opposed to moving in one direction in a stable position) desired velocity $v_d = 0$ is used. Such reward was chosen in order to keep the cumulative reward in a relatively low order and to deduct points for deviating from desired angle of 0 rad more heavily than deviating from desired velocity. This means that the primary goal of the robot should become maintaining a tilt close to 0 rad. As long as the robot maintains a reasonably upright position, it should have little concern about the velocity used.

5 Results

The robot was trained using five different reinforcement learning algorithms: DeepQ learning, TRPO, A2C, ACER and PPO. The simulation was stopped and started again when the robot was falling i.e. position of center of the robot's body was below 15 cm (approximately 1.4 rad tilt) or relatively stable position was maintained for 1500 steps. Simulation data was read at the rate of 100 steps per second in the simulation environment. Figure 2 shows the results of balancing the robot using different reinforcement learning algorithms and MLP policy. A2C, ACER, PPO and TRPO algorithms did not learn to balance the robot during given time frame of 30,000 total steps. While testing these algorithms and running the simulation with models learned, the robot lost its balance and fell within the first 2 s. DeepQ algorithm achieved good results within the given time frame and was successful in balancing the robot within 0.5 rad angle range. Other algorithms were not successful. However, increasing the learning limit to 50000 total steps PPO algorithm (Fig. 2, PPO 2) was successful in learning to maintain the balance. However increasing the time for learning not necessarily results in successfully maintaining the balance. The same experiment of increasing learning time to 50000 steps was tried with ACER algorithm, which did not produce good results compared with PPO algorithm, and the robot still fell during the first few seconds.

The two algorithms that were successful in maintaining the balance after learning using MLP policy resulted in different strategies to accomplish the task. DeepQ algorithm balanced the robot using within much smaller angle range than PPO algorithm (DeepQ - about 0.179 rad, PPO - 1.6376 rad). Also, PPO algorithm used much greater angular velocity of the wheels than DeepQ algorithm, which was in general more stable (Fig. 3). While PPO algorithm used greater angular velocity constantly, DeepQ algorithm generally used quite small velocities, except for periodical angle adjustments using greater velocities than usual, but still usually smaller than biggest velocities using PPO algorithm.

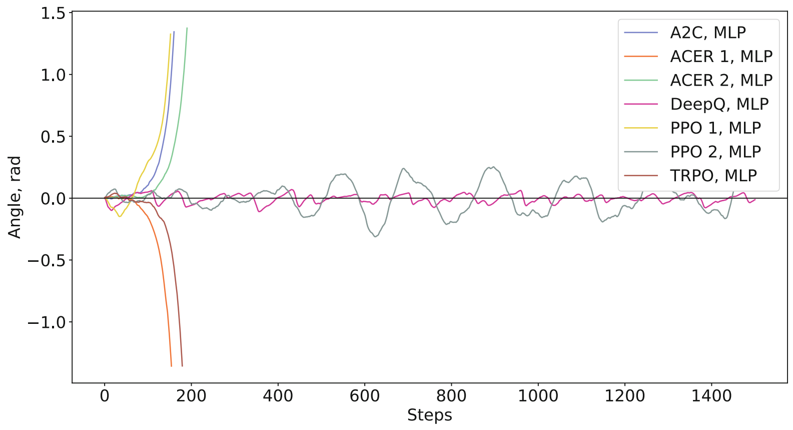

Fig. 2. Changes in tilt angle after learning with MLP policy

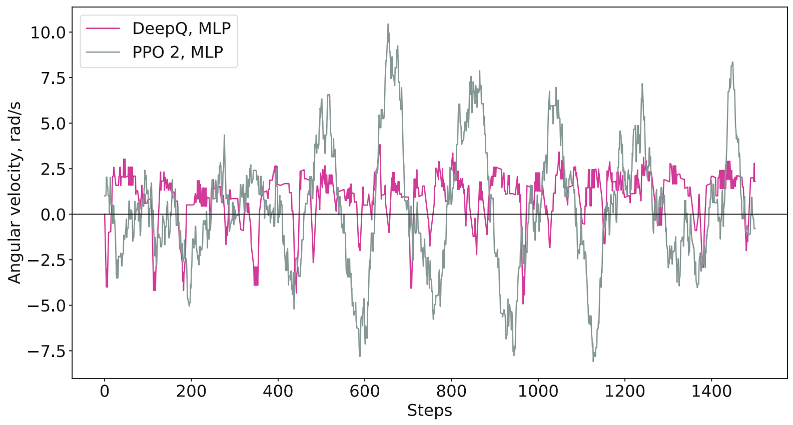

Fig. 3. Changes in angular velocity of wheels after learning with MLP policy

LSTM policy was tried with three algorithms: A2C, ACER and DeepQ, all of which learned to successfully balance the robot within the given time frame of 30000 steps. As it can be seen in Fig. 4, A2C algorithm was visibly worse than ACER and DeepQ algorithms in maintaining a steady balanced position. A2C balanced the robot within 0.5023 rad range, while ACER - 0.1336, and DeepQ - 0.1828 rad range. Besides balancing the robot in the smallest angle range, ACER algorithm also required the smallest range of angular velocity of wheels to maintain the balance (about 5.57 rad/s amplitude). While DeepQ algorithm used almost twice as big velocity range as ACER (10.29 rad/s velocity range), the full range was never used within short time intervals. As shown in Fig. 5, the velocity that was used followed the angle of robot's tilt closely, and changed from peak to peak during intervals of about 2.5 s, while A2C oscillated between

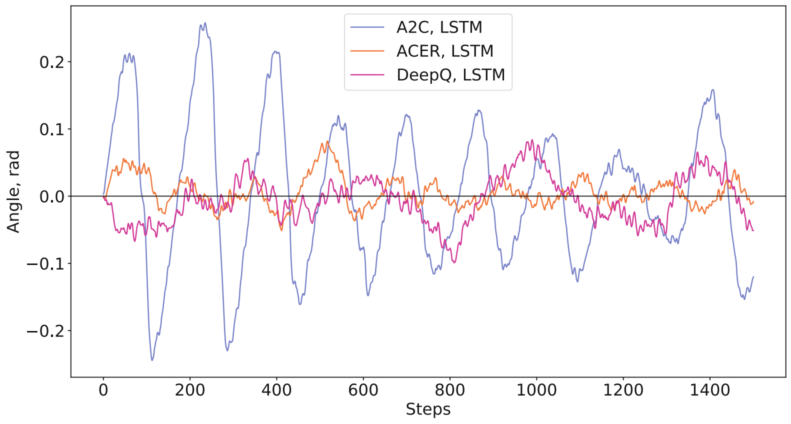

Fig. 4. Changes in tilt angle after learning with LSTM policy

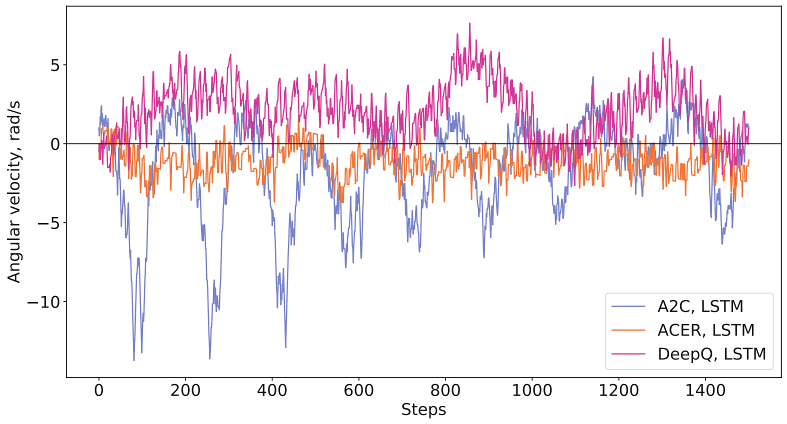

Fig. 5. Changes in angular velocity of wheels after learning with LSTM policy

highest velocities in one direction and highest velocities in different one about every 0.5 s. This makes A2C performance less stable compared with the other two algorithms used.

Detailed results are shown in Table 1. ACER and DeepQ algorithms learned to balance the robot within a smaller tilt range than A2C and PPO algorithms. ACER and DeepQ algorithms also needed a smaller range of angular velocity than A2C or PPO to keep the balance. ACER together with LSTM policy learned the most stable way to keep the balance using smallest tilt and angular velocity range, while both DeepQ MLP and DeepQ LSTM were comparably close. A2C, ACER, TRPO and PPO algorithms combined with MLP policy was unsuccessful in learning the task, although results can be improved in some cases either by changing to LSTM policy, or by increasing the learning steps allowed although this does not guarantee successful balance of the robot.

Table 1. Summary of training results.

Algorithm	Policy	Steps during learning	Tilt range, rad	Angular velocity of wheels, rad/s
A2C	MLP	30000	–	–
	LSTM	30000	0.5023	17.98
ACER	MLP	30000	–	–
	MLP	50000	–	–
	LSTM	30000	0.1336	5.57
DeepQ	MLP	30000	0.179	8.76
	LSTM	30000	0.1828	10.29
TRPO	MLP	30000	–	
PPO	MLP	30000	–	–
	MLP	50000	0.565	18.54

6 Discussion

Different reinforcement learning algorithms require different number of steps to learn to maintain a balanced position. Algorithms that were successful in balancing the robot do so in different ways - trying to keep the tilt angle as low as possible, allowing the tilt angle to fluctuate within a certain radian range, using very sharp changes in wheel rotation to balance, or maintaining the changes in wheels' motion as small as possible, only occasionally adjusting the position with sharp movements to bring the robot back into the upright position.

In most cases, MLP policies produced poor results in comparison to LSTM policies. MLP policies resulted in less sustainable balance of the robot, or at all failed to keep the robot upright. This could be explained by LSTM monitoring the state and evaluating results in the context of past actions, whereas MLP policy only evaluates the current situation, or a buffer of previous situations consisting of single-case observations. This facilitates the LSTM policy in the balancing task by analysing what lead to the current state, paying more attention to the connection between past actions and the current state. It appears that monitoring historical actions are beneficial in robot's stability.

The results were obtained by performing experiments in a simulation, and were not tested with the real life robot. To obtain similar results using a real robot, many unsuccessful attempts to balance would be carried out. In the simulation environment up to 271 falls were necessary for an algorithm to learn to balance the robot. This could result in hardware damage or measurement errors during the impact or introduce measurement noise for subsequent tries. However, even with a maximum protection of robot's hardware the learned ways of keeping the robot in a balanced position would not be feasible. No constraints were enforced on the change of direction of wheels' rotation. The agent could have chosen to go as far as rotating the wheels at any speed to one direction, then in 10 ms reduce the speed by 1 rad/s. This could also result in the constant change in the direction of wheels' rotation every 10 ms, which is hardly sustainable for typical motors used in the models of two-wheeled robots. As a result, not all of the trained models that were obtained in simulation could be implemented using the real robot.

For future research we plan two directions. One: to make our learning methods more sophisticated and penalize for excessive energy consumption, excessive swinging and other undesirable behavior, to train the robot to use different loads, go uphill/downhill and ride uneven terrain. The other direction is to apply these models on real world two-wheeled robots and continue to train from real life data.

References

1. Bature, A.A., et al.: A comparison of controllers for balancing two wheeled inverted pendulum robot. Int. J. Mech. Mechatron. Eng. **14**(3), 62–68 (2014)
2. Brockman, G., et al.: Openai gym. arXiv preprint arXiv:1606.01540 (2016)
3. Coumans, E., Bai., Y.: PyBullet, a Python module for physics simulation in robotics, games and machine learning (2017)
4. Glushchenko, A.I., Petrov, V.A., Lastochkin, K.A.: On development of neural network controller with online training to control two-wheeled balancing robot. In: International Russian Automation Conference (RusAutoCon), IEEE 2018, pp. 1–6 (2018)
5. Imtiaz, M.A., et al.: Control system design, analysis & implementation of two wheeled self balancing robot (TWSBR). In: 2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), pp. 431–437 (2018)
6. Kharola, A., et al.: A comparison study for control and stabilisation of inverted pendulum on inclined surface (IPIS) using PID and fuzzy controllers. Perspect. Sci. **8**, 187–190 (2016)
7. Kim, H.-W., Jung, S.: Fuzzy logic application to a two-wheel mobile robot for balancing control performance. Int. J. Fuzzy Logic Intell. Syst. **12**(2), 154–161 (2012)
8. Liang, S., Gan, F.: Balance control of two-wheeled robot based on reinforcement learning. In: Proceedings of 2011 International Conference on Electronic & Mechanical Engineering and Information Technology, IEEE 2011, vol. 6, pp. 3254–3257 (2011)
9. Mnih, V., et al.: Asynchronous methods for deep reinforcement learning. In: International Conference on Machine Learning, pp. 1928–1937 (2016)
10. Mnih, V., et al.: Playing Atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013)
11. Rahman, M.D.M., Rashid, S.M.H., Hossain, M.M.: Implementation of Q learning and deep Q network for controlling a self balancing robot model. Robot. Biomimetics **5**(1), 1–6 (2018). https://doi.org/10.1186/s40638-018-0091-9
12. Ren, H., Ruan, X.: Bionic self-learning of two-wheeled robot based on skinner's operant conditioning. In: 2009 International Conference on Computational Intelligence and Natural Computing, IEEE 2009, vol. 1, pp. 389–392 (2009)
13. Schulman, J., et al.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)

14. Schulman, J., et al.: Trust region policy optimization. In: International Conference on Machine Learning, pp. 1889–1897 (2015)
15. Kuo-Ho, S., Chen, Y.-Y., Shun-Feng, S.: Design of neural-fuzzy-based controller for two autonomously driven wheeled robot. Neurocomputing **73**(13–15), 2478–2488 (2010)
16. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (2018)
17. Tatikonda, R.C., Battula, V.P., Kumar, V.: Control of inverted pendulum using adaptive neuro fuzzy inference structure (ANFIS). In: Proceedings of 2010 IEEE International Symposium on Circuits and Systems, IEEE 2010, pp. 1348–1351 (2010)
18. Wang, Z., et al.: Sample efficient actor-critic with experience replay. arXiv preprint arXiv:1611.01224 (2016)
19. Xia, P., Li, Y.: The control of two-wheeled self-balancing vehicle based on reinforcement learning in a continuous domain. In: 32nd Youth Academic Annual Conference of Chinese Association of Automation (YAC), IEEE 2017, pp. 1084–1089 (2017)